National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

LIST OF FIGURES

2.1	A class description.	8
2.2	Producer Consumer Problem — a solution	15
2.3	Producer consumer problem — the class hierarchy	15
2.4	MLog code for classes main, buffer, and bounded buffer.	16
2.5	MLog code for classes producer and consumer.	17
2.6	Inheritance of instance variables — one approach.	22
3.1	Translation process	38
3.2	Basic translation of a class into a package.	40
3.3	Use of the <i>success</i> argument with a conjunction of messages.	42
3.4	Use of the <i>success</i> argument with a disjunction of messages.	42
3.5	Use of the <i>success</i> argument with a conjunction of disjunctions of messages.	43
3.6	Use of the <i>success</i> argument in a multiple clause method.	44
3.7	Translation of the cut in a multiple clause method.	46
3.8	Translation of tail recursive method into a loop.	47
3.9	Translation of a repeat-loop.	48
3.10	Restructuring a recursive method — an example	52
3.11	Translation of inheritance using <i>call-through</i> procedures.	57
3.12	Translating inheritance for messages with target <i>self</i> that select redefined methods.	60
3.13	Translating inheritance for messages with explicit targets.	62
3.14	Translation of a message with an explicit target.	64
4.1	Translation of an unannotated message to an explicit target.	67
4.2	Message to an anonymous target — annotated version.	70
4.3	Pseudo-code for Transformation 1 of an extended rendezvous.	73
4.4	Pseudo-code for Transformation 2 of an extended rendezvous.	74
4.5	Example of unfolding messages.	77
4.6	Example of unfolding messages in entry methods.	77
4.7	Algorithm 1: To determine messages that require unfolding.	80
4.8	Example: fold methodB/0 into methodA/0.	81
4.9	Unfold methodA/0 with respect to methodB/0 — a specific case.	82
4.10	Unfold methodA/0 with respect to methodB/0 — a general version.	83
4.11	Unfold methodA/0 with respect to methodB/0 — a complete annotation.	83
4.12	Restructuring post-rendezvous processing code.	94
4.13	Algorithm 2: For consistent typing of polymorphic messages in the body of a single clause method.	106
4.14	Algorithm 3: For consistent typing of polymorphic/monomorphic methods with multiple clauses.	108
5.1	Translating unification — an intermediate step.	112
5.2	Example of unification transformation.	114

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

MLog to Ada Transformations

by

Christine M. Stanczyk, B.Math.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Science
Department of Systems and Computer Engineering
Carleton University Ottawa, Ontario
December 1989

© copyright
1989, C.M.Stanczyk

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-59053-X

ACCEPTANCE SHEET

The undersigned hereby recommends to the Faculty of
Graduate Studies and Research acceptance of the thesis

“MLog to Ada Transformations”

submitted by Christine M. Stanczyk, B. Math., in partial
fulfillment of the requirements for the degree of Master of Science

Professor G.M. Karam, Ph.D.

Chairman,
Department of Systems and
Computer Engineering

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
January, 1990

ABSTRACT

This thesis investigates the use of MLog as a rapid prototyping language for Ada systems. MLog is a language designed to take advantage of object-oriented and declarative programming techniques. Additionally, a powerful set of inter-process communication (IPC) primitives is provided to support concurrent programming.

A cohesive set of transformations is presented that allow the translation of a substantial portion of the MLog language into Ada. The transformation process occurs in two steps: (1) pretranslation processing where the MLog code is annotated/restructured and (2) translation processing where the transformations are applied to the annotated code. The transformations maintain equivalent functionality; no attempt is made to maintain the object-oriented, declarative and IPC characteristics of MLog in the Ada code.

ACKNOWLEDGEMENTS

I would like to express my appreciation to Dr. Gerald M. Karam, my thesis adviser, for his enthusiasm, encouragement and help during the creation of this thesis.

My family and friends are thanked for their unfailing support and encouragement which made this thesis possible.

TABLE OF CONTENTS

ACCEPTANCE SHEET	ii
ABSTRACT	iii
ACKNOWLEDGEMENTS	iv
LIST OF FIGURES	vii
LIST OF TABLES	ix
CHAPTER 1 Introduction	1
1.1 Motivation	1
1.2 Thesis Objectives	3
1.3 Thesis Overview	4
CHAPTER 2 Background	6
2.1 The Language: MLog	6
2.1.1 Structure	7
2.1.2 Behaviour	10
2.1.3 An Example — The Producer Consumer Problem	14
2.2 Ada: An Object-Oriented Language?	18
2.2.1 Object-oriented constructs and Ada — Booch	19
2.2.2 Object-oriented constructs and Ada — Scidewitz	21
2.2.3 InnovAda	24
2.2.4 Classic-Ada	25
2.2.5 Ada++	27
2.2.6 DRAGOON (Distributed Resuable Ada Generated from an Object-Oriented Notation)	28
2.3 Declarative Programming and Ada	29
2.4 Polymorphism in MLog and Ada	33
CHAPTER 3 MLog to Ada Transformations	37
3.1 Structure	39
3.2 Control	40
3.2.1 Conjunctions, Disjunctions and Cuts	40
3.2.2 Loops	45
3.2.3 Object environment	46
3.3 Message Passing	49
3.4 Inheritance	56
3.4.1 Message target is self	58
3.4.2 Messages with explicit targets	61

CHAPTER 4	Pretranslation Processing	65
4.1	Inheritance and message passing	65
4.2	Communications and Concurrency	68
4.2.1	Messages to anonymous targets	69
4.2.2	Conditions for rendezvous	71
4.2.3	Rendezvous location	75
4.3	Declarative versus Procedural Semantics	85
4.3.1	Determinism and directionality	85
4.3.2	Meta programming	88
4.4	Polymorphism and MLog	90
4.4.1	Towards a type inference program (TIP) for MLog	91
4.4.2	Preprocessing typed MLog	96
4.5	Summary	109
CHAPTER 5	Transformations revisited	111
5.1	Unification	111
5.2	Inter-process communications	113
5.2.1	Unguarded Rendezvous	115
5.2.2	Guarded rendezvous	116
5.2.3	Extended rendezvous	119
5.2.4	Anonymous messages	127
5.3	Polymorphic MLog	128
CHAPTER 6	Applying the Transformations	132
6.1	Emulating MLog system support	132
6.2	Translation of a user-defined class	134
6.2.1	Transforming a class into a package	138
6.2.2	Transforming a class into a task type	138
6.3	An example: The producer consumer problem	139
6.3.1	Pre-translation processing	139
6.3.2	Translation processing	142
CHAPTER 7	Conclusions	148
7.1	Summary	148
7.2	Future enhancements	152
BIBLIOGRAPHY		155
APPENDIX A	Techniques for the Translation of MLog Components into Ada . . .	159
A.1	Translation of user-defined classes	159
A.2	Translation of MLog system classes	166
APPENDIX B	The Producer Consumer Problem: An Ada Solution	170

LIST OF FIGURES

2.1	A class description.	8
2.2	Producer Consumer Problem — a solution	15
2.3	Producer consumer problem — the class hierarchy	15
2.4	MLog code for classes main, buffer, and bounded buffer.	16
2.5	MLog code for classes producer and consumer.	17
2.6	Inheritance of instance variables — one approach.	22
3.1	Translation process	38
3.2	Basic translation of a class into a package.	40
3.3	Use of the <i>success</i> argument with a conjunction of messages.	42
3.4	Use of the <i>success</i> argument with a disjunction of messages.	42
3.5	Use of the <i>success</i> argument with a conjunction of disjunctions of messages.	43
3.6	Use of the <i>success</i> argument in a multiple clause method.	44
3.7	Translation of the cut in a multiple clause method.	46
3.8	Translation of tail recursive method into a loop.	47
3.9	Translation of a repeat-loop.	48
3.10	Restructuring a recursive method — an example	52
3.11	Translation of inheritance using <i>call-through</i> procedures.	57
3.12	Translating inheritance for messages with target <i>self</i> that select redefined methods.	60
3.13	Translating inheritance for messages with explicit targets.	62
3.14	Translation of a message with an explicit target.	64
4.1	Translation of an unannotated message to an explicit target.	67
4.2	Message to an anonymous target — annotated version.	70
4.3	Pseudo-code for Transformation 1 of an extended rendezvous.	73
4.4	Pseudo-code for Transformation 2 of an extended rendezvous.	74
4.5	Example of unfolding messages.	77
4.6	Example of unfolding messages in entry methods.	77
4.7	Algorithm 1: To determine messages that require unfolding.	80
4.8	Example: fold methodB/0 into methodA/0.	81
4.9	Unfold methodA/0 with respect to methodB/0 — a specific case	82
4.10	Unfold methodA/0 with respect to methodB/0 — a general version.	83
4.11	Unfold methodA/0 with respect to methodB/0 — a complete annotation.	83
4.12	Restructuring post-rendezvous processing code.	94
4.13	Algorithm 2: For consistent typing of polymorphic messages in the body of a single clause method.	106
4.14	Algorithm 3: For consistent typing of polymorphic/monomorphic methods with multiple clauses.	108
5.1	Translating unification — an intermediate step.	112
5.2	Example of unification transformation.	114

5.3	Example of the translation of a unguarded rendezvous.	117
5.4	Example of the translation of a guarded rendezvous in a single clause method.	118
5.5	Example of the translation of a guarded multiple clause method. . .	120
5.6	Pseudo-code for Transformation 1 modified to individually accept waiting callers.	122
5.7	Transformation 1 modified for accept/! messages.	124
5.8	Transformation 2, modified to support a passive caller object. . . .	126
5.9	Translation of the anonymous message.	129
6.1	Package Global	134
6.2	The package representing some user defined class	136
6.3	Annotated code for producer consumer problem, classes main, producer and consumer.	143
6.4	Annotated code for the producer consumer problem, classes buffer and boundedBuffer.	144
A.1	Translation of a class into a package.	160
A.2	Translation of a user's method into a procedure	161
A.3	Translation of a message within a procedure	162
A.4	Translation of a class into a task type.	163
A.5	Translation of a method into a task entry	164
A.6	Translation of a message within a task entry.	165

LIST OF TABLES

3.1	Criteria for translating messages from passive objects.	53
3.2	Criteria for translating messages from an active object to itself. . .	54
3.3	Criteria for translating messages from an active object to another object.	55
3.4	Criteria for translating anonymous messages.	56
5.1	Transformation of waitToAccept/1 and Accept/1 methods.	115
5.2	Translations of monomorphic/polymorphic methods.	130
7.1	Run-time costs for message passing and object creation.	151
A.1	Summary of translations for class <i>object of nil</i>	168
A.2	Summary of translations for class <i>behaviour of object</i>	169

Chapter 1

Introduction

1.1 Motivation

Prototyping is the process of constructing software for the purpose of obtaining information about the adequacy and appropriateness of the designers' conception of a software product. Prototyping is usually done as a precursor to writing a production system, and a prototype is distinguished from a production system by typically being more quickly developed, more readily adapted, less efficient and/or complete, and more easily instrumented and monitored. Prototyping is useful to the extent it enables information to be gained quickly and at low cost.

The above definition was presented in the Draft Report on Requirements for a Common Prototyping System [1]. The report was driven by a realization within DoD¹ that its standard software life cycle did not identify errors and omissions introduced in the early stages of software development until after the implementation stage. Furthermore, it estimated that the cost of correcting errors or bad decisions increases by a factor of ten for each phase of the life cycle through which it passes undetected [1]. Prototyping is a tool for identifying problems early in the software life cycle.

The rapid prototyping idea, borrowed from other engineering disciplines, relies on producing cheap, simple prototype versions of a system rapidly and early in the project. The prototype becomes a learning device for both end user and developer [2]. The customer gets increased visibility into the software system, which encourages the early identification, and resolution of discrepancies between user expectations and system capabilities. The developer of the system has a tool for testing the feasibility, functionality and performance of alternate design strategies. Prototypes are used to answer three questions: is it a solution to the posed problem, does it have acceptable performance, production cost and reliability, and is it a solution to the right problem [2]?

"MLog is a multitasking, object-oriented Prolog for which the primary application is prototyping embedded concurrent systems" [3]. The object-oriented elements of MLog

¹ DoD is an acronym for the United States Department of Defense.

are attractive for rapid prototyping; the inheritance hierarchy encourages the reuse of code and alternative structural decompositions of the problem are easily defined through classes and instances. The Prolog element of MLog provides the declarative semantics that allows complex problems to be programmed with relative ease. Due to syntactic compactness and semantic power, Prolog programs have been shown to be approximately ten times shorter than equivalent programs in Pascal, Fortran or Basic [4]. Since MLog is based on the Prolog model, MLog exhibits equivalent program compactness which directly contributes to rapid prototype development. As in Ada, concurrency in MLog is based on large grain communicating sequential processes and the rendezvous. However the interprocess communication (IPC) primitives are more powerful and allow designers to solve complex IPC problems quite easily [5, 3]. A detailed analysis of MLog as a rapid prototyping tool presented by Bond [6] and Wilson [5], concluded that MLog is a good rapid prototyping language for concurrent systems.

Prototypes have a wide range of life expectancies. They can be seen as *throw-aways*, models that are discarded immediately after use, or they can become the initial implementation of the software system. In the latter case, inefficiencies that were acceptable in the prototype must be removed in the production version by means of transformations. This is referred to as a *generative basis* where production implementations are derived from prototypes by applying transformations to improve efficiency [1]. The possibility of using a MLog prototype as the generative basis for a production system motivated this thesis. Specifically, the transformations between MLog and the target production language, Ada, are investigated. If the quickly developed prototype can become the implementation system or a part of the implementation system with relatively little effort, two advantages occur.

- The time required to develop the implementation system will be significantly decreased which by default will reduce the cost of development. Furthermore, the implementation will be functionally equivalent to the prototype assuming that the transformation algorithms are correct.

- The potential exists for the prototype to continue to model the system/subsystem through the remainder of its life cycle. As a system is modified the changes can be modelled first on the prototype with minimum effort, and once verified, can be transformed into the required Ada code. This use of the prototype will require a clean, well defined interface to the remainder of the system.

MLog prototypes, whether used as the production system or as throw-away models cannot provide accurate performance analysis, with respect to timings, due to the differences in the IPC primitives of MLog and the target language; (for example, the more powerful IPC primitives result in fewer rendezvous in MLog programs than in functionally equivalent Ada programs). The availability of semi-automated transformations from MLog to Ada, allows the program in the target language to provide the performance data on timings.

Ada is the natural choice for the target language. The IPC paradigms of both MLog and Ada are based on synchronous communications utilizing the rendezvous. In addition, Ada is designed to support embedded concurrent systems and is the DoD's language of choice for military applications which form a significant portion of embedded concurrent systems. This is indicated in the Draft Report on Requirements for a Common Prototyping System, which states that one of the target languages supported must be Ada.

1.2 Thesis Objectives

This thesis explores the feasibility of developing an automated or semi-automated tool for generating reasonably efficient Ada code from MLog code such that the programs are functionally equivalent. This study addresses the following areas.

- The evaluation of methods to emulate the functionality of object-oriented paradigms in Ada. Where efficiency conflicts with a strict emulation of the object-oriented structure, efficiency shall be the over-riding criteria.

- The identification of MLog code that must be restructured to accommodate reasonably efficient transformations. The restructuring of code will be assessed with respect to the object-oriented, logic programming, and IPC elements of MLog.
- The identification of a set of annotations for MLog code that will act as translator directives, which will support maximized automation of the required code restructuring.
- The identification of transformations, from annotated MLog code to Ada code, that are suitable for automated processing.

1.3 Thesis Overview

This thesis begins with a synopsis of the MLog language. Its object-oriented paradigm provides a built-in structure for programs, its declarative nature provides compact, easy to understand code, and the powerful IPC primitives allow complex communications to be presented in a straight forward manner. A summary of these capabilities and their interfaces is provided. Also, Chapter two will review the object-oriented features of Ada and approaches used to convert object-oriented and logic programming techniques into Ada.

Chapter three presents the transformations that convert the structural and control elements of MLog into Ada. An analysis of how the object-oriented hierarchy of MLog can be emulated in Ada is presented, as well as a detailed analysis of the message passing semantics and how they interface with the emulated control structure in Ada.

Chapter four presents an analysis of all aspects of MLog with respect to the feasibility of efficient transformations into Ada. This analysis determines aspects of the language that are unsuitable for translation and that must be restructured by the user. These decisions are based on the current lack of an efficient solution, or on a basic incompatibility between the two languages; for example, self modifying code is supported in MLog, but is totally foreign to Ada. This chapter also determines where and how the MLog code should be annotated to facilitate automated translation.

Chapter five presents some further transformations. Specifically, the translations of unification, IPC primitives, and polymorphic MLog methods will be discussed.

Chapter six illustrates the proposed transformations by example using the producers/consumers problem and chapter seven presents conclusions and future enhancements.

Chapter 2

Background

2.1 The Language: MLog

MLog is a concurrent, object-oriented Prolog, designed to facilitate the rapid prototyping of concurrent systems. The structure of the language is determined from the object-oriented model, the behaviour is derived primarily from the logic programming model, and the concurrency is based on the inter-process communication of large grain sequential processes. The use of the three programming language paradigms in MLog is summarized in the following sections. A complete description is presented by G. Karam in [7] from which this summary is drawn.

The nomenclature used to describe MLog is based on that found in object-oriented languages. A brief glossary of MLog terminology is provided below.

Class — A class is a program module used to group methods and instance variables.

A class has a superclass and may have subclasses. The system class object has superclass `nil`; i.e., it has no superclass.

Instance variable — A persistent data store. Each object of a class has a private copy of the instance variables defined for the class. An object's persistent state is defined as the value of its instance variables.

Message — This is the specification of an operation to be performed by an object. A message consists of a target which identifies the object that must process the message, and a selector that identifies the method to be invoked.

Method — A Prolog-like procedure used to implement the response to a message. A method consists of one or more clauses, and is defined by its functor and arity.

Method clause — This is a single clause within a method, and corresponds to a Prolog clause. Each clause consists of a head and an optional body.

Object — An object encapsulates some private data and a set of operations (methods) on that data. An object stores data in its instance variables and responds to messages by executing methods. Every object is an instance of a class.

Selector — A part of a message. The selector identifies the method to be invoked by the target object on receipt of a message.

Self — The default target for messages that do not contain explicit targets. A message to self means the current object is sending the message to itself.

Super — The target of a message. Super identifies the current object as the message recipient. The difference between the targets self and super relates to where the system begins the search for a method to match the message selector. If the message target is self the search begins in the class of the current object. If the message target is super the search begins in the superclass of the class where the message is lexically located.

Target — The part of a message that identifies the object that is the receiver of the message. A message can have an explicit target, (e.g., X-msgA), or an implicit target, (e.g., msgA). The implicit target is defined to be self.

2.1.1 Structure

An MLog program is designed as a collection of classes. Each class may define instance variables, class methods, and instance methods. Figure 2.1 illustrates a class description. In MLog, method variable names begin with an upper case character; the names of instance variables and other literals begin with a lower case character.

Each program usually defines a class main as a subclass of class object. The MLog system begins a program's execution by creating an instance of class main and passing control to that instance. The class main object acts as the main-line code of the program and is typically used to initialize the program by creating the other required objects.

Instance variables provide persistent data stores for objects of the class, and are defined by the statement:

```
ivars: [instvar1, instvar2=value, ...].
```

```

class: classA of object.      % defines a class

ivars: [var1,var2=1].         % define two instance variables,
                                % var1 is free,
                                % var2 has default value 1
methods: class.               % defines class methods

methodA(X) :-                 % methodA/1 is a class method
    classB-methodB(X).        % message to the object, classB,
                                % with selector=methodB(X)

methods: instance.           % defines instance methods

methodC(X,Y) :-               % methodC/2 is an instance method
    Y-methodD(X),              % message with target=Y, and
                                % selector=methodD(X)
    theValueOf(var2,Value),    % message to self to read var2
    Z is Value + X,            % message to self to invoke is/2
    makeTheValueOf(var1,Z),    % message to update var1
    abc(W)-methodE.            % message to anonymous target in
                                % class abc

```

Figure 2.1 A class description.

The `ivars:` statement provides a list of all instance variables defined in the class. The list can include default values for some, all, or none of the instance variables in the class description; e.g., `instvar2` is defined with a default value. The default values can be over-ridden at object creation using the method `new/3`. After an object is created only the object has access to its instance variables.

Methods are equivalent to Prolog procedures, and provide the functionality available to objects of the class. Methods consist of one or more clauses with the same functor and arity. Each clause of a method consists of a head, and an optional body. The neck, “:-”, is used to separate the head and body of a clause. The body of a method consists of a conjunction/disjunction of messages which are equivalent to the subgoals of a Prolog clause. A message can have two possible formats: (1) the message consists of a target object followed by “-”, followed by a message selector, (e.g., `X-msgA(args...)`), or

(2) the message can consist of just the selector, (e.g. `msgA(args...)`). In the latter case the implicit target of the message is defined to be `self`.

Instance methods are those following the statement: `methods:instance`. These methods are accessed by instances of the class. When no `methods:` statement exists in a class definition, the methods are instance methods by default.

Class methods are defined as those methods following the statement: `methods:class`. They can only be accessed by objects that are classes. In MLog, like Smalltalk, classes are objects; a class is an instance of a meta-class. A meta-class has only one instance unlike other classes which can have multiple instances. Class methods represent the instance methods of the class object's meta-class. A complete description of the class hierarchy is found in [7].

Objects

The basic component of an MLog program is the object. An object is an instance of a class and encapsulates private copies of the instance variables defined for the class and has access to the methods defined for the class. Objects are created dynamically in MLog using the method: `new(Instance,Mode,[InitVals])`.

The first argument (`Instance`) identifies the new object. The argument can be: (1) an uninstantiated variable in which case the system provides an object-id to identify the instance, or (2) a user defined symbolic name which is used to identify the instance. The second argument (`Mode`) determines the mode of the new object and must have the value *active* or *passive*. An active object is capable of autonomous behaviour; a passive object is not. The last argument (`[InitVals]`) allows instance variables in the newly created object to be initialized. The argument consists of a list of instance variable names and their associated initial values. An empty list indicates no create-time instance variable initialization is required. When an active object is created, the `begin/0` method of the class is automatically invoked.

Inheritance

MLog supports hierarchical inheritance. Each class has only one superclass and can have none or multiple subclasses [8]. A class inherits all methods and instance variables from its superclass unless they are over-ridden in the class. Since each superclass has a superclass, a class inherits all methods and instance variables defined in its superclass or any class above its superclass in the inheritance hierarchy.

Methods can be redefined (over-ridden) in a subclass by providing a new method for an old method's selector. Instance variables can be redefined also in MLog, unlike Smalltalk. The instance variable is included in the `ivars:` statement and given a new initial value. When a method or variable appears more than once in the inheritance hierarchy of a class, the definition found in the closest superclass is the version inherited.

The inheritance hierarchy is defined by the `class:` statements. In figure 2.1, `classA` identifies the class, and `object` defines its superclass. All classes have a superclass except class `object` whose superclass is defined as `nil`.

2.1.2 Behaviour

Execution proceeds in MLog as a result of objects sending messages to each other; i.e., message passing. When a message is sent to a target object, the object uses the message selector to invoke the correct method. The method selected by the message is determined using inheritance. If the class of the accepting object defines the matching method, then that method is selected and processed, otherwise, the superclass of this class is searched. A method matches a selector when their functor and arity are the same. The search is repeated until the method is found or until the class `nil` is reached. When the class `nil` is reached an error occurs since the message has selected a method that does not exist for the target object; i.e., it is not defined in or inherited by the target's class.

The only exception to the above process occurs when the target is `super`. `Super` indicates that: (1) the target of the message is the current object, and (2) the search for the selected method should start in the superclass of the class where the message is lexically located (instead of with the current object's class). The target `super` is used to add new

behaviour to an existing method. In the following example new behaviour is added to methodA/1 of classA. MethodA/1 in classB is modified to call methodB/0 before processing the version of methodA/1 defined in classA (the superclass).

Example:

```
class: classB of classA.
methodA(X) :- methodB, super-methodA(X) .
```

Choosing the correct method to invoke relies on the object-oriented characteristics of MLog, but how the methods are processed relies on the declarative semantics of Prolog goal reduction and the mode and identity of the receiving object. A message passed to a passive object has the semantics of a procedure call. A message passed to an active object has the semantics of a rendezvous. A message passed by an object to itself has the semantics of a procedure call.

Messages to Passive Objects

When a method is invoked as a result of matching the message selector, the processing of the message is based on the conventional goal reduction of the Prolog model. The clause heads of the method are scanned from top to bottom until a clause is found that unifies with the message selector, if no clause is found the method terminates in failure. If a clause is found, then each message in the clause body is processed sequentially, and returns success or failure. If all messages return success, then the method terminates successfully. If a message terminates as failed, then the remainder of the clause is abandoned, and backtracking occurs to solve the clause. If the clause fails, then unification is attempted with a new clause head, selected from those clauses of the method that follow the last clause selected, and the process repeats. If the method fails (no clause succeeds), then the message that selected the method fails.

Messages to Active Objects

A message to an active object is processed as a rendezvous although it has the same syntax as a message to a passive object. When a caller object sends a message to an active object the message is placed on the FIFO queue of messages of the acceptor object, and

the caller object is blocked until its message is processed successfully (the rendezvous terminates). The acceptor object decides to accept a rendezvous by sending itself the `waitToAccept/1` message. The `waitToAccept/1` message causes the argument of the selector to be used to select the next method. The rendezvous consists of two steps: eligibility and evaluation; both steps must succeed for the rendezvous to succeed.

The eligibility step succeeds if the message selector on the FIFO queue unifies with the argument of the `waitToAccept/1` message. This argument can be an uninstantiated variable which means any message can pass the eligibility step, or the argument can be a method name and arguments which means only messages that invoke the specific method and unify with its arguments are eligible. When unification succeeds, the method that matches the argument of the `waitToAccept/1` message is invoked by the target object.

The evaluation step consists of processing the selected method using unification and goal reduction as for messages to passive objects. If the method succeeds then the rendezvous terminates. The caller is unblocked, and proceeds to process its next message. The acceptor's `waitToAccept/1` message succeeds, and the acceptor proceeds to process its next message. However, if the method fails, then the evaluation step fails, and the rendezvous fails.

When a rendezvous fails the message is returned to the FIFO queue of the acceptor object in its original position, and the acceptor tries the next message. The failed message remains on the queue and the caller remains blocked until the message can be processed successfully. A message from a caller that results in a rendezvous cannot fail unless an undefined method is selected. The acceptor object processes the messages on its FIFO queue sequentially until one successfully passes both the eligibility and evaluation steps, at which time the rendezvous terminates. If no message can pass both steps the acceptor blocks until a new message arrives. At that time, both steps of the rendezvous are attempted again. Messages that remain on the queue (their attempt to rendezvous failed) are attempted again on the next invocation of `waitToAccept/1` as the queue is always processed from front to back. This preserves the FIFO ordering of the queue

subject to the messages' ability to pass the eligibility and evaluation steps.

After a rendezvous completes there is no connection between the caller and acceptor. If the message that caused the rendezvous, in either the caller or acceptor object, is backtracked through, only the bindings caused by the rendezvous in that object are undone, no effect is felt in the other object.

MLog supports the nested rendezvous; i.e., an object accepts another rendezvous before the first one is completed. This is accomplished by sending another `waitToAccept/1` message before completing an existing `waitToAccept/1` message. Nested rendezvous in MLog must terminate in the reverse order of their initiation; the last rendezvous accepted must terminate first, as in Ada.

The previous discussion has focused on the `waitToAccept/1` method as the means to accept a rendezvous. MLog also provides the `accept/1` method which causes an object to accept a rendezvous without blocking. If no message on the FIFO queue can successfully complete the rendezvous, then the acceptor object does not block, it fails and backtracking occurs to find the next message to process.

In MLog, an active object accepts a rendezvous based on caller and/or acceptor data. A MLog rendezvous that relies on caller data is referred to as an *extended rendezvous*.

Messages to anonymous targets

Messages to anonymous targets allow a caller to send a message to a class of active instances; one of the active instances accepts the rendezvous and processes the message. The syntax of a message to an anonymous target is: `classname(X)-methodA`, where `classname` is the class of objects, `X` is an uninstantiated variable that will be unified with the pointer of the object that processed the message, and `methodA/0` is the method to be processed. On completion of the rendezvous, the acceptor, depending on its internal state can accept another rendezvous. These messages are referred to as *anonymous messages*.

2.1.3 An Example — The Producer Consumer Problem

In the producer consumer problem an autonomous producer object stores items in a shared buffer and an autonomous consumer object retrieves items from the shared buffer. The shared buffer is a bounded buffer that maintains a last in first out (LIFO) ordering. Figure 2.2 illustrates the proposed solution using Buhr's [9] iconic notation.. The solution contains five classes of objects: producer, consumer, buffer, boundedBuffer and main. The class hierarchy is shown in figure 2.3, and the code is shown in figures 2.4 and 2.5.

Class Main. An active instance of class main creates an active instance of the producer, consumer and bounded buffer classes, prints the message that all active tasks are initialized, and then terminates. The instance of class main is created by the MLog system which transfers control to the object to start the program.

Class BoundedBuffer. The boundedBuffer object which was given the user defined symbolic name common [cref. figure 2.4] is active and on start-up it executes the begin/0 method. Since a begin/0 method is not defined in the class or its superclass (buffer) the object inherits the default begin/0 method defined in the class object:

```
begin :- repeat, waitToAccept(X), fail.
```

The method causes the object to loop endlessly waiting to accept messages. The object accepts the store/1 method (defined in the class), and the retrieve/1 method (inherited from the superclass). The store/1 method checks if the bounded buffer is full by comparing the value of the instance variable, bound (which defines the maximum buffer size) with the actual length of the buffer, (determined by the inherited method size/1). If the buffer is not full, a message to invoke the method store/1 defined in the superclass is sent by using the target super. The method store/1 defined in class buffer is selected and causes the data to be stored in the instance variable myBuffer of the object common.

Class Producer. The producer object executes its begin/0 method on start-up which creates a passive buffer object initialized to a list of fruit. The create-time initialization of the instance variable myBuffer is used to over-ride the default value (an empty list)

implemented either manually, by conditional statements written by the programmer, or automatically, by the programming language and the run-time environment [11]. Ada, due to its strict typing and its restriction of procedures as first class data types, requires manual implementation while MLog, like Smalltalk, provides dynamic binding as an element of the language. In Ada, dynamic situations use access types or pointer variables with conditional statements [11]. However, this approach does not support the software engineering principles of extendibility, reuseability and compatibility as it scatters *case* discriminations throughout the program, and does not allow a new choice to be added without modifying all existing case statements for that type [16, 11].

Seidewitz suggests that dynamic binding can be emulated in Ada to some degree by parameterizing generic packages with types and subprograms. This allows a new class to be declared which can use any existing class, that contains certain needed operations, by instantiating the equivalent generic package. The needed operations are specified by the formal subprogram parameters of the generic package. Also, generic facilities can be used to defer the implementation of some operations to subclasses by making these operations formal subprogram parameters which are defined in the subclass. In neither case is dynamic binding occurring since in Ada generic entities are elaborated at compile time and separate copies of the generic are made for each instantiation of the package. The use of generics as a solution for dynamic binding implies that enough knowledge exists at compile time to allow the correct generic packages to be instantiated; this is not generally the case for methods that are redefined (defined more than once in the class hierarchy). Instantiating generic packages representing superclasses in a package is analogous to explicitly copying each inherited method into the class. Messages that select methods that have been redefined and that are directed to the targets *self* can be achieved only by true dynamic binding. The problem of selecting the method at run-time, instead of at compile time, still exists. Generics do provide an efficient way of defining multiple subclasses of a class when the differences between the subclasses can be defined by the formal parameters of the generic.

% CLASS MAIN.

class: main of object.

begin :-

```
    boundedBuffer-new(common,active,[bound=5]),
    producer-new(P,active,[]),
    consumer-new(C,active,[]),
    mainConsole-writeln("All tasks initialized").
```

% CLASS BUFFER.

class: buffer of object.

ivars: [myBuffer=[]].

store(DATA) :-

```
    theValueOf(myBuffer,B),
    makeTheValueOf(myBuffer,[DATA|B]).
```

retrieve(DATA) :-

```
    size(S), S > 0,
    theValueOf(myBuffer,[DATA|B]),
    makeTheValueOf(myBuffer,B).
```

size(S) :-

```
    theValueOf(myBuffer,B),
    listLength(B,S).
```

% CLASS BOUNDED BUFFER.

class: boundedBuffer of buffer.

ivars: [bound].

```
store(DATA) :- theValueOf(bound,MAX),
    size(S), S < MAX,
    super-store(DATA).
```

Figure 2.4 MLog code for classes main, buffer, and bounded buffer.

% CLASS PRODUCER.

class: producer of object.

```
begin :-
    buffer-new(X,passive,[myBuffer=[apples,oranges,pears,kiwi]]),
    producerLoop(X).
```

```
producerLoop(X) :-
    X-retrieve(D),
    mainConsole-(write("sending "),writeln(D)),
    common-store(D),
    producerLoop(X).
producerLoop(X).
```

% CLASS CONSUMER.

class: consumer of object.

```
begin :-
    buffer-new(X,passive,[]),
    consumerLoop(X).
```

```
consumerLoop(X) :-
    mainConsole-writeln("waiting to receive"),
    common-retrieve(D),
    X-store(D),
    X-size(S),
    mainConsole-(write(D),write(" received"),nl),
    mainConsole-(write(S),write(" elements received"),nl),
    consumerLoop(X).
```

Figure 2.5 MLog code for classes producer and consumer.

defined in the class buffer description. Continuing the `begin/0` method, the producer object sends itself the message `producerLoop/1`. The matching method repeatedly retrieves an element from the producer object's passive buffer and stores the element in the active bounded buffer, `common`. Due to the mode of the objects, messages to retrieve data have the semantics of a procedure call, and messages to store data result in a rendezvous. When the passive buffer is empty the message `retrieve/1` fails which causes the method `producerLoop/1` to succeed and the object terminates.

Class Consumer. The consumer object uses the `begin/0` message to create a passive buffer object, and then invoke the method `consumerLoop/1`. The method retrieves data from the active bounded buffer object, `common`, (results in a rendezvous), and stores the data in its passive buffer object, (results in a procedure call). The consumer object does not terminate; it blocks on the message queue of the bounded `_buffer` object, waiting to retrieve more data from the shared bounded buffer.

Class Buffer. This class represents unbounded buffer objects. The producer consumer solution uses two objects of this class, both passive, to hold the private data of the producer and consumer objects. The buffer objects do not terminate since they are passive; to delete these objects the user must send the `removeInstance/1` message to the buffer class.

2.2 Ada: An Object-Oriented Language?

Although Ada has been called an object-oriented language [10, 11], it is not equivalent to MLog. MLog, modelled on Smalltalk, supports objects, classes, inheritance, and provides linguistic support for dynamic binding. Ada supports data abstraction and information hiding. Wegner [12] differentiates between object-based and object-oriented languages. Ada is a object-based language as it provides support for objects, where an object is defined as follows.

Object: An object has a set of operations and a state that remembers the effect of the operations. Objects communicate by sending each other messages to perform operations. Objects may be contrasted to functions which have no memory. Whereas function values

are completely determined by their arguments, an object may learn from experience, its reaction to an operation being determined by its invocation history.

Object-oriented languages then are defined as:

object-oriented = object-based + classes + class inheritance

Under this definition Smalltalk and MLog are both object-oriented languages.

Booch [10], who considers Ada an object-oriented language, and Seidewitz [13], who considers Smalltalk an object-oriented language, have both analysed the use of Ada to support object-oriented programming. The approaches suggested are evaluated with respect to their applicability for transforming MLog into Ada.

Four systems have been developed that support object-oriented programming in Ada. InnovAda², Classic-Ada³, Ada++, and DRAGOON all provide an interface that allows classes, methods, instance variables, and objects to be defined. None of the systems provides an automatic translation of another language into Ada code. The user provides the Ada code, but the structure of classes, inheritance, message passing, etc. are transformed by these systems into compilable Ada code. The implementation strategies for the transformation of object-oriented constructs into Ada are described in varying levels of detail, but are summarized here to show approach and experience with the transformations. All the systems appear to provide a complete emulation of the object-oriented paradigm as defined above. However, only InnovAda, Classic-Ada and DRAGOON provide support for dynamic binding.

2.2.1 Object-oriented constructs and Ada — Booch

Booch [10] suggests that classes of objects be denoted by packages that export private or limited private types: (1) objects are instances of the private or limited private types and (2) methods are implemented as subprograms exported from the package specification. This mapping maintains the information hiding and data abstraction properties of MLog. In MLog all objects are first class data types. Since instances of a class have been given a type under this mapping, they are first class data types and can be passed as data.

² InnovAda is copyrighted by Harris Corporation, 1987.

³ Classic-Ada is a trademark of Software Productivity Solutions Inc.

However, packages which cannot be passed as data represent classes, thus classes are not represented as first class data objects under the transformation, and cannot be passed as data. In MLog, all classes are objects (instances of their meta-class), therefore this translation is valid for instances of classes excluding meta-classes.

The object's state (the value of its instance variables) resides in the declared object for instances of private or limited private types. The Smalltalk/V [14] description also equated instance variables to a data record in conventional languages.

Booch differentiates between classes and anonymous classes. The latter are classes with only one object and can be represented as packages that serve as abstract state machines. There is no requirement for instances of the class as the class definition can represent the single object. In these cases, the object state resides in the body of the package. This approach has two problems with respect to MLog to Ada transformations.

- The object is not a first class data type.
- Due to the dynamic creation of objects in MLog, in general it is not known at compile time how many instances of a class will be created.

Although Booch does not believe inheritance is a necessary condition for object-orientedness, he suggests a form of inheritance can be achieved in Ada using derived types. By representing a class of objects, as instances of non-private types, subclasses can be built by deriving from the first type. The derivation inherits all subprograms of the superclass that use the derived type as a parameter or as the returned value in a function call. Since the type is not encapsulated, new subprograms can be added and existing subprograms can be replaced. In addition to voiding the protection provided by Ada's strong typing and encapsulation abilities, this approach is not a complete answer to the transformation of MLog inheritance for the following reasons.

- It prohibits the redefinition of inherited instance variables to a new type.
- It does not provide an inheritance mechanism for superclasses without instance variables.

- Subprograms in a superclass that do not use the derived type in their interface are not derived (inherited).

These problems also apply to Perez' [15] use of derived types to emulate inheritance. Perez, unlike Booch, represents objects as instances of private data types. However with private types, new instance variables cannot be added directly to the derived type (the inherited instance variable) in Ada. Perez solves the problem by defining: (1) a non-Ada syntax that allows additional instance variables to be defined in the subclass and (2) a set of transformation rules to transform the non-Ada syntax into functionally equivalent legal Ada code.

2.2.2 Object-oriented constructs and Ada — Seidewitz

Seidewitz [13] differentiates between classes of objects that are passed as data and those that are not passed as data. Classes whose objects are passed as data need instantiation as a type. The transformation suggested, corresponds to that by Booch; classes are represented by packages, objects are instantiations of private types, methods are represented as subprograms and object state resides in the private type. Classes of objects that are not passed as data do not require a type, and can be defined in Ada by using generic packages. The generic package is considered the class definition while instances of the package represent objects. This approach cannot be used when transforming MLog, since all objects in MLog are created dynamically while generic instantiations in Ada are elaborated at compile time. Furthermore, MLog objects are all first class data types, capable of being passed as data.

Inheritance of methods can be emulated in Ada by a package specification defining all methods of the class including inherited methods. The inherited methods are implemented in the package body by *call-through* procedures to the superclass. Call-through procedures are procedures whose only function is to call another procedure. This transformation results in one extra procedure call for each level of the class hierarchy that the inherited method travelled, and does not always select the correct method if the method has been redefined [cref. §3.4].

```

type IVARS is private;
...
private
  type IVARS is record
    IVAR_1 : TYPE1;
    IVAR_2 : SUPERCLASS.IVARS;
  end record;

```

The IVARS record type defines the instance variables for the current class. IVAR_1 is an instance variable defined in the class; IVAR_2 is a record defining all instance variables inherited from the superclass.

Figure 2.6 Inheritance of instance variables — one approach.

Inheritance of instance variables is supported by using the private type definition in the superclass (which declares the superclass' instance variables) to declare part of the instance variables of the subclass, see figure 2.6. This approach, which is also the basis of Perez' [15] automated transformations, requires that all access to inherited instance variables use the methods supplied in the superclass since the inherited instance variables are private types. As a result, operations on inherited instance variables cannot be overridden in subclasses, and instance variables cannot be redefined in subclasses. The latter restraint is valid for the Smalltalk language, but provides an unacceptable restriction on the MLog language which permits the redefinition of instance variables in subclasses. Finally, accessing inherited private types forces a ladder of procedure calls since at each level of the class hierarchy the procedure call must be routed to the next superclass until the class is reached where the private type is defined.. Seidewitz relates the problem with this approach to the strict typing of Ada which does not recognize subclassing. The type of the subclass' instance variables is not a subtype of the superclass' instance variables, therefore it is not possible to pass an argument of the subclass' instance variable type to a procedure expecting the superclass' instance variable type. Generics may produce a partial solution, but inheritance has been proved to be more powerful than genericity [16].

In addition to objects and classes, (which provide data abstraction and information hiding), and inheritance, dynamic binding is often considered to be a characteristic of object-oriented languages [13, 17, 18, 19, 20]. Dynamic binding refers to the run-time selection of the method that is invoked by a message. Dynamic binding can be

implemented either manually, by conditional statements written by the programmer, or automatically, by the programming language and the run-time environment [11]. Ada, due to its strict typing and its restriction of procedures as first class data types, requires manual implementation while MLog, like Smalltalk, provides dynamic binding as an element of the language. In Ada, dynamic situations use access types or pointer variables with conditional statements [11]. However, this approach does not support the software engineering principles of extendibility, reuseability and compatibility as it scatters *case* discriminations throughout the program, and does not allow a new choice to be added without modifying all existing case statements for that type [16, 11].

Seidewitz suggests that dynamic binding can be emulated in Ada to some degree by parameterizing generic packages with types and subprograms. This allows a new class to be declared which can use any existing class, that contains certain needed operations, by instantiating the equivalent generic package. The needed operations are specified by the formal subprogram parameters of the generic package. Also, generic facilities can be used to defer the implementation of some operations to subclasses by making these operations formal subprogram parameters which are defined in the subclass. In neither case is dynamic binding occurring since in Ada generic entities are elaborated at compile time and separate copies of the generic are made for each instantiation of the package. The use of generics as a solution for dynamic binding implies that enough knowledge exists at compile time to allow the correct generic packages to be instantiated; this is not generally the case for methods that are redefined (defined more than once in the class hierarchy). Instantiating generic packages representing superclasses in a package is analogous to explicitly copying each inherited method into the class. Messages that select methods that have been redefined and that are directed to the targets *self* can be achieved only by true dynamic binding. The problem of selecting the method at run-time, instead of at compile time, still exists. Generics do provide an efficient way of defining multiple subclasses of a class when the differences between the subclasses can be defined by the formal parameters of the generic.

2.2.3 InnovAda

InnovAda [17] is an object-oriented development system that was created to port systems developed in Flavours or Smalltalk into Ada. Under InnovAda the user can define flavours (classes), methods, instance variables, and inheritance. Classes are defined by providing:

- a class name;
- the name and type of instance variables;
- the names of superclasses, (multiple inheritance is supported);
- the names of other packages that must be visible;
- directives that tell the InnovAda system to generate methods to access, update, or re-initialize instance variables.

Methods are defined by providing: (1) a method name, (2) a class name, (3) the argument names and their types and (4) the actual Ada code required to implement the method's functionality. Objects are created either statically or dynamically by providing the class name, object name and values for the instance variables. Message passing is implemented using the function SEND which has the parameters object name, message name, and message arguments. By evaluating the message name as a variable at run time, InnovAda achieves dynamic binding to the selected method.

InnovAda supports inheritance, dynamic binding, message passing, multiple inheritance, and active objects. Each class is supplied with a corresponding active object implemented as a task (since independent processes are required to support simulations). Active objects have the same capabilities as normal objects with the exception that messages are sent as entry calls (as opposed to the function calls used by passive objects).

Each class is implemented as a Ada package, however other implementation details were not provided.

Ada systems produced through InnovAda were evaluated with respect to development time, execution speed, and application size and maintainability. Their experience showed a "dramatic" decrease in code development time, (despite the user providing the application code for methods); this decrease is credited towards the object-oriented design

process and the automated generation of Ada code from the InnovAda specification. The measurements on the speed of the application were based on the comparison of the time required to pass a message to an object, and to make a straight function call. The results supported those of Cox [11]; message passing took approximately 2.5 times longer. The code mass was greater for small InnovAda produced systems due to the code required to support the object-oriented concepts. However, as the applications became larger, the InnovAda produced systems became smaller due to code reuse as a result of inheritance.

The InnovAda system has shown that significant savings in development time and program size can be achieved. The increased execution time can be minimized by reducing the amount of message passing in the system. Message passing and inheritance both require dynamic binding which is run-time expensive.

Enhancements are planned to the present system as follows.

- Dynamic binding of messages will be replaced by static binding where possible.
- The translation of inheritance will be optimized by removing intermediate levels of inheritance.
- Compiler errors which presently reference the InnovAda generated Ada code, will be mapped (by a parser) back onto the user generated Ada code for the method definitions.

InnovAda is easy to learn for those who know Ada and the principles of object-oriented programming.

2.2.4 Classic-Ada

Classic-Ada [18, 21] is a toolset to support object-oriented programming within the Ada language. This is not a transformation of an existing language into Ada, but a collection of language extensions that permit the specification of classes, objects, inheritance lattices, and message passing. Since Classic-Ada is a hybrid system, features of the Ada language are still available. Within Classic-Ada the developer can define classes using a format similar to Ada packages. The class construct has a visible specification component, and

a private body component. The superclass, methods and instance variables are identified within the class construct. Classes also contain the instance creation routines. Message passing is provided by a `send` function which has the arguments: object-id, method name and method arguments. The ability to incorporate standard Ada is accomplished through a package refinement feature which allows the developer to hand tune the class by adding *with* clauses, data types etc.

The Classic-Ada code is fed into a code generator that produces a system that is compiled as DoD standard Ada “with polymorphic message passing protocol, specification of class attributes with inheritance, dynamic binding, and definition of instance attributes” [18]. The code generator consists of five components which are described as follows.

Language processor/object generator — The Classic-Ada processor converts the object definitions consisting of the data-types, methods, and messages into Ada code. Part of the process includes the incorporation of these objects into an object base.

Object-Base/Dictionary — The object-base maintains class and method definitions for all Classic-Ada units processed by the object generator.

Builder — The builder extracts information from the object-base and builds supporting data structures for dynamic binding and inheritance.

Object-Base Reporter — The reporter provides status information regarding the object-base’s contents, classes available, and other information.

Executive — The executive is a collection of packages generated by the builder and used at run-time to resolve messages references and support the inheritance facility.

The system allows the object-oriented development of systems and the delivery of DoD standard Ada to the customer. The use of the executive component during run-time indicates that the advantages gained from object-oriented design are offset by a level of indirection in the code of the production system. However, this price is inevitable unless the Ada language is changed to provide direct support of object-oriented programming. The description of Classic-Ada makes no reference to active objects or the concurrency capability of Ada, and gave no details on the implementation approach used in the code generator to transform the object-oriented constructs into standard Ada constructs.

2.2.5 Ada++

Ada++ [22] is a superset of Ada that allows the definition of: (1) classes, (2) the inheritance lattice (multiple inheritance is supported), (3) slots (instance variables) and (4) the reserved word *sendsuper* to indicate the call of a method defined in a parent class. Ada++ is a statically strongly typed language; i.e., each object is typed at compile time. In Ada++, a class defines a type. The dynamic binding of object-oriented programming is not supported.

Classes and objects are implemented as Ada packages and records, however the inheritance mechanism provides a new operational semantics. Inheritance is solved using static type checking with an extended version of the usual overloading algorithm that includes a provision for inheritance. Method identification is achieved by subprogram typing resolution, and thus message binding will be static. The selection of the correct method uses the following steps.

- Determine the actual parameter types of the method.
- Determine the unique definition corresponding to these types.

Each slot in a class has a specific name and stands for a typed value. Objects consist of the set of all local and inherited slots. Inherited slots that are defined to be private types are inherited in subclasses with respect to the logical structure of the instances of the subclass, but they are not directly accessible outside of the class declaration. (This is more restrictive than MLog where all instance variables are polymorphic and can be accessed directly by methods defined in subclasses.)

Ada++ is implemented as a two step preprocessor that produces compilable Ada code as follows.

1. Produce syntactically correct Ada code with classes and objects translated into packages and records, but with expressions that may be ill-typed in Ada.
2. Resolve any ill-typing by use of the *overloading + inheritance resolution algorithm* that consists of two phases:

- a. type checking for Ada expressions;
- b. type conversion when inheritance arises.

Ada++ visibility rules determine for each expression and statement, two sets of types:

- regular types resulting from Ada's overloading resolution;
- inherited types induced by the class hierarchy.

Method selection is treated as type determination and standard binding rules apply. To be correct, an expression must support, with respect to its context, exactly one type.

Ada++, unlike the previous two systems, eliminated dynamic binding in favour of static binding which maintains the static philosophy of Ada and the efficiency gained from static type checking. However, like the previous systems, Ada++ expects the user to code in Ada.

2.2.6 DRAGOON (Distributed Resuable Ada Generated from an Object-Oriented Notation)

DRAGOON [20] is a language that is based largely on Ada and augmented by object-oriented features. Classes and multiple inheritance are supported, and the basic inheritance model is extended to handle concurrency, distribution and reconfiguration. DRAGOON supports *behavioural* classes that contain descriptions of temporal relationships (e.g., mutual exclusion, serialization). A concurrent application is designed using multiple inheritance — a non-concurrent class inherits from a behavioural and a non-behavioural class, this causes a new concurrent class with appropriate functional and temporal behaviour.

Classes are translated as packages, and instance variables are represented as access types to data records. The state of an object is not represented by a single record as in previous systems, it is represented by a linked list of records, each node in the list stores the state added by a subclass in the inheritance hierarchy. This supports DRAGOON's ability to dynamically assign objects to different instance variables. (Note: it appears that classes represent types in DRAGOON, therefore objects of different

classes have different types.) The creation and manipulation of these lists use Ada's `UNCHECKED_CONVERSION`.

Unlike InnovAda and Classic-Ada which use the `send` function to implement dynamic binding of methods, DRAGOON supports dynamic binding by generating two Ada subprograms for each method. One subprogram contains the Ada code equivalent to the DRAGOON method; the other subprogram provides the run-time selection from the currently existing versions of the method.

Active objects are represented as tasks, the thread of the active object is mapped into a task type which allows the creation of multiple active objects. To allow the task to manipulate the state of the object the task requires an initialization entry which receives the object's state list as a parameter. The behavior of an active object is implemented by a dynamically created task which provides methods that implement operations such as *signal* and *wait*. The specified behaviour is obtained by the method selection subprogram calling these behaviour methods immediately before and after the implementing method subprogram.

DRAGOON's approach to dynamic binding and inheritance of both instance variables and methods is significantly different to the previous systems. Although DRAGOON also supports a greater functionality, its implementation is more complex.

2.3 Declarative Programming and Ada

MLog execution is based on the Prolog model which uses top-down resolution with unification and backtracking [3]. The translation of MLog's declarative semantics into Ada which provides no support for declarative programming results in similar problems to those encountered when translating Prolog into Ada or other imperative languages. The Prolog transformations can provide a basis for the transformation of MLog, but the MLog transformation is inherently more complex due to its support for concurrency and object-oriented programming.

A Prolog clause base cannot be transformed into a set of Ada subprograms since: (1) the static compilation model of Ada is incompatible with the dynamically changing Prolog clause base and (2) the call-return model of Ada subprograms is incompatible with the backtracking model of Prolog [23]. Most deductive language implementations in an Ada environment implement an interpreter which is coded in Ada [23]. The Adalog system [24] is an example of an interpreter in Ada. A second approach is to compile Prolog clauses into instructions for a machine. This approach relies on the procedural semantics of Prolog and static analysis that generates code for each clause. A compiler can increase efficiency 10 to 20 times compared to an interpreter, but may induce some restrictions on the dynamic characteristics of the Prolog programs processed [23].

The problems in compiling Prolog are [25]:

- compilation of non-deterministic control structures (clauses);
- compilation of terms into appropriate data structures;
- compilation of unification.

Annotating Prolog code with additional information on the determinism of procedures, the syntactic types, and the directionality (mode) of procedure arguments partially eliminates these problems and allows more efficient code to be produced. Deterministic procedures do not require the extensive overhead that non-determinism invokes, and the type and directionality information allows assignment and efficient pattern matching to replace unification [26]. Matching is the unidirectional version of unification and is used in the *function call* of imperative programming languages [27].

Warren compiled Prolog into a specialized intermediate language that is designed so that Prolog's control and data structures map in a straight forward manner to a sequence of instructions in this intermediate language and forms the basis of the Warren Abstract Machine (WAM) [26]. Instead of building a machine onto which Prolog could be easily mapped, Mellish [28] examined the subset of procedures that do not require the overhead of an abstract machine but could be directly mapped onto a conventional machine. He observed that most Prolog programs are both directional and deterministic. Programs can

be annotated to provide this information without undue hardship since the programmer is usually aware of the directionality and determinism of predicates [26].

Drabent [29] pursued Mellish's hypothesis "that many Prolog programs are not radically different from programs written in conventional languages" [28]. Conventional programs were assumed to be a transformation of input data into output data, and *simple logic programs* were defined as programs that transform input ground terms into output ground terms. A simple logic program has the property that at every step of its computation all input positions of its actual subgoal are ground and when the subgoal succeeds, all its output positions are ground. This definition of simple logic programs allows non-determinism but imposes directionality on the Prolog procedures. Simple logic programs can be executed using pattern matching instead of unification if delayed unification on output positions occurs. Delayed unification on output positions means that at a procedure call only the input positions are unified. Unification of the output positions is done on success of the call. Delaying unification may itself increase efficiency since unification is performed only if the procedure succeeds. However, delaying unification may cause unnecessary backtracking when the unification performed at the time of the procedure call would have failed [29]. The latter disadvantage disappears for deterministic code. Drabent checked a sample of Prolog programs to see if they met the criteria for simple logic programs; his results confirm Mellish's hypothesis. About half the programs satisfied the restrictions and substantial parts of the remaining ones were "conventional" too. However, the sample also revealed some programming techniques which are unique to logic programming.

Compilation of Prolog into imperative languages has used continuation passing to implement backtracking [28, 26, 30, 25, 31] which causes a procedure to indicate success by invoking the next procedure to be executed and returning only on failure. This technique is facilitated by a language that supports "linked record data structures and recursive procedures (accepting parameter procedures)" [25]. However, Ada does not support procedures as subprogram parameters.⁴ The translation of Prolog to Ada

⁴ Since Ada does not treat procedures as types, procedures cannot be passed directly as arguments to a subprogram.

requires the implementation of abstract inference machines. These have been combined with compiled clause bases to provide more efficient implementations in the Slate [32] and Adlog [23] systems which provide Ada with access to declarative programming.

Slate system. The Slate inference engine performs full unification and supports forward and backward chaining. (The backward chaining implementation is based on the Prolog model.) The Slate system uses dynamic Ada tasking to decouple search of the rule base from the inference engine which asserts and retracts bindings and provides control. Search tasks retrieve sets of rules from the compiled rule base. One search task is created per backtrack point. This system may realize significant throughput savings in a multi-processor environment [32], but the high cost (with respect to time) of rendezvous raises the question of the efficiency of this implementation on a single processor machine.

Adlog system. This system provides a deductive kernel that can be used by the applications to process deductive code. The system consists of: (1) a compiler that transforms the rule base into an Ada form and (2) a run-time executive containing the inference engine and the primitives used to manipulate the compiled data base. Adlog emulates a Prolog abstract machine in Ada; it is based on the WAM modified to facilitate implementation in Ada. Abstract codes representing Prolog clauses are stored in the form of Ada data, which constitutes programs that can be executed by the abstract machine. "The Adlog system was evaluated as providing good performance in comparison with purely interpretive structures" [23].

The previous discussion of Prolog implementations has indicated two approaches to the translation of MLog.

- Fully emulate the deductive semantics of MLog in Ada through the use of an abstract inference machine. Multiple copies of portions of the machine would be required to accommodate the multiple threads of control in concurrent MLog applications. This approach would be further complicated by the object-oriented aspects of MLog.

This restriction can be circumvented through the use of a pointer used in conjunction with a case statement. A pointer is passed as a argument to the procedure, A, which uses the pointer in a case statement to determine the new procedure to execute. However, the procedure, A, must know at compile time all possible choices for the pseudo parameter procedure.

- Restrict the declarative semantics of MLog to those characteristics that can be mapped directly onto a conventional machine using Ada.

This thesis follows the second option. The restrictions on MLog are considered justified due to the increased efficiency of the resulting code and the significant portion of declarative code that is programmed in a directional and deterministic manner.

2.4 Polymorphism in MLog and Ada

Carelli & Wegner [33] define polymorphic languages as those that allow some values and variables to have one or more types, polymorphic functions/procedures are those whose parameters can have more than one type, and polymorphic types are defined as types whose operations are applicable to operands of more than one type. Parametric polymorphism is obtained when a function works uniformly on a range of types; these types normally exhibit some common structure. In parametric polymorphism the uniformity of type structure is achieved by type parameters; i.e., a polymorphic function has an explicit or implicit type parameter which determines the type of the argument for each application of the function. In terms of implementation, a parametric polymorphic function executes the same code for arguments of any admissible type. MLog meets the requirements of a polymorphic language.

Ada is a primarily monomorphic language [33]. In a monomorphic language every value and variable can be interpreted to be of one and only one type. The polymorphism of Ada's generic procedures is specialized to particular kinds of parameters — types, values, and procedures. However, generic polymorphism in Ada is syntactic since generic instantiation is performed at compile time with actual values that must be determined at compile time. The semantics of generic procedures can be considered abbreviation for sets of monomorphic procedures. In true polymorphic systems, code is generated only once for every generic procedure.

To translate MLog's polymorphic methods into Ada's generic subprograms, the actual values of the type parameters and procedure parameters must be determined from the

static MLog code. This requires that MLog be typed. Again the Prolog experience with typing is used as a starting point for investigating an MLog solution.

Type checking and type inference systems have both been studied with respect to typing Prolog.

- Type checking systems [34, 35] use programmer supplied type declarations which define the legal types for polymorphic methods and variables. These declarations can then be used by a type checker to ensure consistent typing in the program.
- Type inference systems infer the types of expressions when little or no type information is given explicitly [33]. This approach is attractive since user input into the typing system appears to be minimal.

Horiuchi & Kanamori [36], and Xu & Warren [37] both developed type inference programs for Prolog based on semantic types and using abstract interpretation.

Horiuchi & Kanamori [36] consider types as the set of all terms satisfying the definition of type predicates which must be supplied by the user. These type definitions are used to classify the set of all terms (the Herbrand universe) into various sets of terms (types) [36]. In addition to the user supplied definitions which must define disjoint sets, the system adds two additional types: (1) *any* — set of all terms, and (2) ϕ — the empty set. Using abstract interpretation, all arguments of a program are typed with a user supplied type, *any*, or ϕ . *Any* is used when more than one of the user types can correctly solve the predicate; ϕ is used when no types can satisfy the predicate. The use of *any* as an argument type is valid in Prolog and MLog, however it does not provide sufficiently fine grain information for Ada translation purposes.

Xu & Warren [37] consider types as subsets of values that language expressions can take, and are therefore subsets of a universal domain of values. Although the system does not require any input from the programmer because it uses data available in the program to derive types, these types are not useful for a practical application unless augmented by user-defined types. In order to obtain a more specific typing the user must provide additional type information in the form of type definitions and type declarations. A type

definition is a mapping from user-supplied type names to types which are sets of Herbrand terms. The following example defines a parametric list; i.e., a list with elements of type A.

Example: `list(nil,A).`

`list(cons(X,Y),A) :- A(X), list(Y,A).`

A type declaration specifies the type for arguments of a predicate. Multiple type declarations for a predicate are allowed. The following example declares that `append/3` can be used with integer lists as the type of each argument.

`append(X,Y,Z) ---> append(int_list,int_list,int_list).`

The type inference procedure consists of four steps.

- Construct a type base according to the type definitions in the program. In the worst case each variable, constant, and function in the program is considered a separate type so there is a one to one correspondence between the Herbrand universe and the type base.
- Transform the program by replacing each constant with its type.
- Use the inference engine (a Prolog-like theorem prover) to evaluate each predicate in the program to form the set of typings for the predicate.
- Collect the set of typings for a predicate, and combine them into a single formula that becomes the typing of the predicate. User supplied type definitions (if provided) allow the collect function to provide a set of types "preferred by the user". A type is preferred if any of the following conditions are met.
 - a. The type results from a user-supplied type definition.
 - b. The type has one element and that element does not belong to any other type definition.
 - c. The type corresponds to a function symbol in the program, and that function type does not appear in any other type definition.

The time complexity of this system is: $O(|TB|^N * |C| * L * P)$ where :
 $|TB|$ is the magnitude of the type base,

N is the bound on the maximum arity of a predicate,

$|C|$ is the number of clauses in the program,

L is the bound on the number of goals in a clause, and

P is the number of predicates in the program.

The system does not require that user type definitions result in disjoint sets. However, this system also defines new types as the the union and intersection of existing types, which can result in the final typing of the program not having a sufficiently fine grain for Ada translation. Judicious use of type declarations and type definitions by the user can minimize this problem. This system does not support meta-predicates; (i.e., a predicate used as an argument in a goal).

Chapter 3

MLog to Ada Transformations

MLog is an integrated language that combines the characteristics of object-oriented programming and logic programming with the use of large grain communicating processes. The transformations presented in this thesis allow users to design and code in MLog and then translate the code into Ada. This approach retains the efficiencies of object-oriented design and logic programming while using the transformations to produce more efficient Ada code. The user continues to do all development in MLog, and theoretically would require no knowledge of the Ada language. This differs from the InnovAda and Classic-Ada systems where the user provides the Ada code for his methods.

The previous chapter summarized translation approaches from object-oriented languages to Ada and from Prolog to Ada. The emulation of either of these programming paradigms in Ada involves some level of run-time cost. In order to limit the overhead required to support object-oriented programming in Ada, a full emulation of an object-oriented system is not attempted. Only the functionality required to support the specific application being translated is provided. For example, a complete inheritance mechanism is not provided, rather those classes whose objects require specific inherited methods are provided access to them. To further maximize the efficiency of the resulting Ada code, the declarative semantics of MLog are not fully supported in the translated code as this requires the use of an abstract inference machine. Only those portions of the MLog language that can be mapped directly onto a conventional machine are supported by the proposed transformations. The third feature MLog supports is concurrent processing. The inter-process communication (IPC) used are based on the rendezvous, but are much more powerful; these IPC constructs must be translated into Ada, but unless additional data is provided by the developer, these implementations are not very efficient.

The transformations are based on a static analysis of the code, therefore the complete program must be submitted for translation. This is as a result of not completely emulating the object-oriented constructs.

Figure 3.1 Translation process

The MLog code cannot be transformed directly into Ada due to:

- the decision not to fully translate MLog's declarative semantics;
- the differences in the IPC semantics of the two languages;
- the lack of built in language support in Ada for inheritance and dynamic binding.

The MLog code must be processed prior to translation. This pretranslation processing consists of two parts.

- User intervention to annotate the code to either: (1) increase the efficiency of the resulting Ada code, (e.g., to minimize the dynamic binding of procedure calls), or (2) restructure code that is not supported by the transformations, (e.g., remove non-deterministic programming as backtracking does not map directly onto a conventional machine).
- An automated static analysis of the MLog program to collect data used in the transformations.

The restructured and/or annotated code is then processed by transformations which produce compilable Ada code. The overall structure of the translation process is shown in figure 3.1.

The remainder of this chapter presents some basic transformations of MLog to Ada. The following chapter provides the summary of and rationale for the pretranslation processing.

3.1 Structure

MLog programs are composed of building blocks such as classes, methods, instances and instance variables. Within a class, the objects have individual persistent states defined by instance variables, but the operations (methods), are common for all objects of the class. The transformation from MLog to Ada requires that equivalent representations be found for the structural features of MLog; the following representations are used.

- Classes are represented as packages.
- Methods are represented as subprograms.
- A MLog instance (active or passive) is represented as a data record which associates the object with its persistent state.

An Ada package encapsulates the subprograms and data records required to define the objects of the equivalent MLog class. Methods that are accessed by external objects or that are accessed as inherited methods, (i.e., a subclass' object invokes the method), appear as subprograms in the package specification (the external interface). Ada subprograms include functions and procedures. Methods without output arguments can be represented as procedures, or Ada functions that return a success or fail status on completion. For simplicity, methods used as examples, in this thesis, are translated consistently into procedures.

The data records representing instance variables are also visible in the package specification. This visibility of object state diverges from the information hiding approach suggested by [10, 13] [cref. §2.2]. A non-private type is selected to represent an object since in MLog the instance variables can be modified by: (1) the creating object using the new/3 method, and (2) the created object using the methods theValueOf/2, makeTheValueOf/2. These objects typically belong to different classes, (different packages in Ada); therefore, a non-private data type was chosen to simplify implementation. Furthermore, within the MLog program, the integrity of the object's state is protected and as the transformations do not add new functionality to the programs, the integrity of the object's state in the Ada program is maintained.

in MLog: class: classA of object. ivars: [X]. methodA :- methodB.	in Ada: package CLASSA is procedure METHODDA; task type ACTIVE_CLASSA is entry METHODDA; end ACTIVE_CLASSA; type IVAR is record OBJECT : OBJ_NAME; X : TYPE_OF_X; end record; end CLASSA;
--	---

Figure 3.2 Basic translation of a class into a package.

In Ada active entities are represented as tasks. To accommodate active MLog objects, each package defines a task type that is instantiated for each active object in the equivalent MLog class. The task type defines entries equivalent to the subprograms that appear in the package specification. The `begin/0` method of a class is translated as the mainline code in the equivalent task type. Figure 3.2 illustrates a class and the equivalent package specification. The state of an object, active or passive, is represented as an instance of the visible data type defined in the package specification. Also, active objects are associated with an instance of the task type which provides the autonomous control associated with active objects.

3.2 Control

3.2.1 Conjunctions, Disjunctions and Cuts

Execution control is exercised in MLog using elements found in the logic programming and object-oriented disciplines. The basic computation is driven by top-down resolution using unification and backtracking in accordance with the Prolog model [3]. The `“,”/2`, `“;”/2`, and `“!”/0` methods, which represent the conjunction of messages, the disjunction

of messages, and the cut respectively, have nearly the same operational semantics as the equivalent Prolog predicates.⁵ These *control methods* are used within a method to determine which message will be processed next based on the success or failure of the previously processed message. The control methods are demonstrated using examples.

- `methodA :- methodB, methodC.`

MethodA/0 succeeds if methodB/0 and methodC/0 succeed.

- `methodA :- methodB; methodC.`

MethodA/0 succeeds if methodB/0 or methodC/0 succeeds.

- `methodA :- methodB, !, methodC.`

`methodA :- methodD.`

MethodA/0 succeeds if the first clause succeeds or if the first clause fails prior to the cut and the second clause succeeds. If the first clause fails after the cut (methodC/0 fails), then methodA/0 fails since the cut prevents any further clauses from being tried.

In the above examples all messages appearing in clause bodies can represent a single message, or can represent structures which are conjunctions/disjunctions of messages.

The execution path through a method is dependent on the success or failure of its messages. This control is incorporated into the equivalent Ada subprogram by adding an argument, *success*. The argument will return a boolean value indicating whether the subprogram succeeded or failed. On entering a procedure *success* will be set to true, and will be reset by a failure during pattern matching (the code resulting from the translation of unification, [cref. §5.1]) or by a failure in a subprogram call. The use of the success argument to control the flow of program execution is shown in conjunction with each of the control methods of MLog. The control methods are not translated as procedure calls in Ada, but as in-line code.

- The transformation of a method that consists of a conjunction of messages is shown in figure 3.3. If methodB/0, methodC/0 or methodD/0 fail then methodA/0

⁵ The cut is transparent through a disjunction of messages in Prolog, this is not the case in MLog.

in MLog: methodA :- methodB, methodC, methodD.

in Ada:

```

procedure METHODDA(SUCCESS : in out BOOLEAN ) is
    . . .
    SUCCESS := TRUE;
    METHODDB(SUCCESS);
    if SUCCESS then
        METHODDC(SUCCESS);
    end if;
    if SUCCESS then
        METHODDD(SUCCESS);
    end if;
end METHODDA;

```

Figure 3.3 Use of the *success* argument with a conjunction of messages.

in MLog: methodA :- methodB; methodC.

in Ada:

```

procedure METHODDA(SUCCESS : in out BOOLEAN ) is
    . . .
    SUCCESS := TRUE;
    METHODDB(SUCCESS);
    if SUCCESS then
        null;
    else
        METHODDC(SUCCESS);
    end if;
end METHODDA;

```

Figure 3.4 Use of the *success* argument with a disjunction of messages.

```

in MLog:
methodA :- (methodB;methodC),
           (methodD;methodE),
           methodF.

in Ada:
procedure METHODDA(SUCCESS : in out BOOLEAN) is
    . . .
    SUCCESS := TRUE;
    METHODDB(SUCCESS);
    if SUCCESS then
        null;
    else
        METHODDC(SUCCESS);
    end if;
    if SUCCESS then
        METHODDD(SUCCESS);
        if SUCCESS then
            null;
        else
            METHODE(SUCCESS);
        end if;
    end if;
    if SUCCESS then
        METHODDF(SUCCESS);
    end if;
end METHODDA;

```

Figure 3.5 Use of the *success* argument with a conjunction of disjunctions of messages.

fails, or equivalently if any of the subprograms METHODDB, METHODDC, or METHODDD fail then the subprogram METHODDA fails. In methods with multiple clauses, the *if* statements do not cause the procedure (e.g. METHODDA) to return immediately on failure of a subprogram.

- The transformation of a method that consists of a disjunction of messages is shown in figure 3.4. The procedure METHODDA succeeds if one of the subprograms METHODDB or METHODDC succeeds.
- The transformation of a method that consists of a combination of disjunctions and conjunctions of messages is illustrated in figure 3.5. The procedure METHODDA

in MLog:

```
methodA :- methodB.
methodA :- methodC.
```

in Ada:

```
procedure METHODDA(SUCCESS : in out BOOLEAN ) is
    . . .
    SUCCESS := TRUE;
    METHODDB(SUCCESS);
    if SUCCESS then
        return;
    end if;
    SUCCESS := TRUE;
    METHODDC(SUCCESS);
end METHODDA;
```

Figure 3.6 Use of the *success* argument in a multiple clause method.

succeeds if one of METHODDB and METHODDC succeed, and one of METHODDD and METHODE succeed, and METHODDF succeeds.

- The transformation of a method that consists of multiple clauses is illustrated in figure 3.6. The translation is for a disjunction of clauses as opposed to a disjunction of methods, which was discussed previously. The method succeeds when one of its clauses succeeds. The body of each clause is processed: when a clause succeeds the procedure is exited, and methodA/0 succeeds or equivalently the subprogram METHODDA succeeds.

Cuts

The cut (the “!”/0 method), is also used to control execution flow by controlling backtracking in MLog. MLog accepted by the translator will be deterministic (cref. §4.3.1); since a deterministic method has only one solution, backtracking is limited to the selection of clauses in a multiple clause method. The cut is translated as follows.

- A method contains a cut and consists of a single clause; the cut is not translated.

example: `methodA :- methodB, !, methodC.`

The cut is designed to prevent a second choice being used for `methodA/0` or `methodB/0` if `methodC/0` fails. `MethodB/0` is deterministic and `methodA/0` has only one clause (one choice), therefore the cut has no effect on program execution and can be ignored on translation.

- A method contains a cut, and consists of multiple clauses. The cut must be translated, since it modifies the possible execution paths through the method. The cut is represented as a boolean variable which is set true when a cut is executed. In Ada, the cut variable is used together with the success argument to determine program flow. An extension to the transformation proposed for methods with multiple clauses is required when the method contains a cut(s) as shown in figure 3.7. Since the “!”/0 method always succeeds, a success argument and a separate test for success or failure in the translated code is not required; the cut is translated as an assignment statement.

3.2.2 Loops

MLog allows the modification of execution flow through the use of loops. Looping is implemented through recursive methods or through failure-driven loops.

Recursive methods are translated into: (1) recursive Ada procedures, or (2) iterative loops if the method is tail recursive with no *out* arguments. The translation of tail recursion by an iterative loop is illustrated in figure 3.8. This transformation must be used for recursive `begin/0` methods since they form the main-line code in tasks. The `loop` statement begins the translation of the method, and the `end loop` statement is lexically the last executable statement in the method.

Failure-driven loops are replaced by iterative loops in Ada. A failure-driven loop uses failure with its associated backtracking to cause the re-evaluation of messages. Since MLog code that is to be translated is deterministic [cref.4.3.1], backtracking occurs only on the system method `repeat/0`, which always succeeds; i.e., it always provides a new choice point for the resolution algorithm. `Repeat/0` messages appearing in methods are translated as loop statements. The lexical location of the `end loop` statement occurs at

```

in MLog:
methodA :- methodB, , methodC.
methodA :- methodD.

in Ada:
procedure METHODDA(SUCCESS : in out BOOLEAN) is
. . .
- translation of first clause
  SUCCESS := TRUE;
  METHODDB(SUCCESS);
  if SUCCESS then
    CUT := TRUE;
    METHODDC(SUCCESS);
  end if;
  if SUCCESS or CUT then
    return;
  end if;
- translation of second clause
  SUCCESS := TRUE;
  METHODD(SUCCESS );
end methodA;

```

Figure 3.7 Translation of the cut in a multiple clause method.

the end of the conjunction of messages that contains the `repeat/0` message. The end loop statement is preceded by a check on the `success` argument, if `success` is true the loop is exited, if it is false the loop reiterates. The system method `fail/0` is often used in failure-driven loops to force failure. `Fail/0` is translated as an assignment of the value *false* to the `success` argument. Figure 3.9 illustrates the translation of repeat-loops.

3.2.3 Object environment

MLog processing causes messages to be sent between objects. Messages consist of a target and the message selector which designates the method that the target is to execute. The control features discussed previously determine the execution flow within the method but make no provision to ensure that the correct object receives the message.

```

in MLog:

begin :- methodA,begin.
begin :- methodB.
begin :- methodC,begin.

in Ada:
task body ... is
    ...
    loop
        METHODDA(...);
        if SUCCESS then
            null;
        else
            SUCCESS := TRUE;
            METHODDB(...);
            if SUCCESS then
                exit;
            else
                SUCCESS := TRUE;
                METHODDC(...);
                if SUCCESS then
                    null;
                else
                    exit;
                end if;
            end if;
        end if;
    end loop;
end;

```

Note The tests for success that result in null can be omitted, but are shown here to parallel the transformation of disjunctions of methods in figure 3.4.

Figure 3.8 Translation of tail recursive method into a loop.

The information on the current object and the current process in a program is available automatically from the MLog system, but must be determined explicitly in the equivalent Ada code. To accomplish this an environment argument, (ENVIRONMENT) is added to each subprogram or task entry representing a MLog method. The environment argument

```
in MLog:
methodC :- methodA, repeat, methodB.
```

```
in Ada:
procedure METHODDC(...) is
    ...
    SUCCESS := TRUE;
    METHODDA(...);
    if SUCCESS then
        loop
            METHODDB(...);
            if SUCCESS then
                exit;
            end if;
        end loop;
    end if;
end METHODDC;
```

Figure 3.9 Translation of a repeat-loop.

is a record containing the following fields.

- `cur_object` identifies the current object which is the object the subprogram is acting on.
- `cur_class` identifies the class of the current object.
- `cur_process` identifies the active object currently controlling execution.

The current object and current process are initially the object of the class *main of object* which acts as the mainline of the MLog program; the current class is set to *main*. During program execution, these values are updated when a message is sent to another object (for a detailed analysis see section 3.3). The environment argument is required for the translation of inheritance, message passing and for the correct accessing of instance variables.

Example:

in MLog:

```
class: classA of object.
methodA(X) :- X-methodB.
```

in Ada:

```
package CLASSA is
    . . .
    procedure METHODA(ENVIRONMENT : in ENV_RECORD_T;
                     X: in STRING;
                     SUCCESS: in out BOOLEAN);
end CLASSA;
```

The translation of the example assumes that the argument, *X*, has a mode *in*, and a type *string*. The mode and type of all method arguments will be available prior to the transformations being applied as a result of pretranslation processing (cref. §4.3.1,4.4.2).

3.3 Message Passing

Communications between objects in MLog is achieved through message passing. A message can be directed to a class of objects (an anonymous message), or to a single object.

The operational semantics of message passing depends on the *mode* of the calling and accepting objects. The mode of an object⁶ is either passive or active. The operational semantics of message passing obey the following rules.

- A message sent to a passive object has the semantics of a procedure call.
- A message sent to an active object has the semantics of a rendezvous unless the object is calling itself.
- A message sent by an object to itself has the semantics of a procedure call.

In addition to the mode of the caller object and acceptor object, the identity of the current process also impacts translation. For example, a passive object sends a message to an active object which is also the current process; effectively the current process is sending

⁶ This should not be confused with the *mode* of a method or subprogram argument. Argument modes are *in*, *out*, and *in out*.

a message to itself indirectly through the passive object. In Ada, this is equivalent to a task calling itself indirectly through a procedure and results in deadlock. The translation must ensure that an active object does not try to rendezvous with itself indirectly. The characteristics of the method invoked by the message also effect the required translation as outlined below.

- *The method uses the extended rendezvous.*

Since Ada does not support the extended rendezvous directly, special transformations are required (cref. §5.2.3).

- *The method contains a waitToAccept/1 or accept/1 message.*

In MLog, any method in an active object can be used to accept a rendezvous.

In Ada, procedures cannot accept rendezvous, only task entries can. The translation must ensure that a procedure does not attempt to accept a rendezvous.

- *The method is recursive.*

In MLog, the recursive calls to a method are treated as procedure calls since the object is calling itself. In Ada, a recursive task entry results in deadlock.

Special transformations are required [cref. tables 3.2, 3.3, and figure A.5], and then only some cases are translatable.

An analysis of possible combinations of caller, acceptor, and message selector characteristics is presented for messages to a single object. Table 3.1 analyses messages from a passive object, table 3.2 analyses messages from an active object to itself, and table 3.3 analyses messages from an active object to some other object, active or passive. The characteristics of the method can be determined by static analysis, but the characteristics of the caller, acceptor and current process require run-time tests.

The analysis of factors affecting the translation of message passing result in a limited number of distinct transformations.

- The message is translated as a procedure call when the acceptor is passive or the current process equals the acceptor object.

- The message is translated as a call to a task entry when the acceptor object is active and the current process does not equal the acceptor object.
- The message is translated as a call to a task entry and the task entry consists of a call to a recursive procedure. This is used to translate recursive methods that do not contain `waitToAccept/1` or `accept/1` messages.
- The message is translated as a call to a task entry under one of the transformations for an extended rendezvous (cref. §5.2.3).

The characteristics of the caller object, acceptor object and current process determine when the values of the current object, current class and current process must be updated to reflect the acceptor object. The analysis also determined three cases that cannot be translated directly and require restructuring of the MLog code by the user.

- An active object sends itself a message indirectly through a passive object, and the method invoked contains a `waitToAccept/1` or `accept/1` message.
- An active object sends itself a message directly, and the method invoked contains a `waitToAccept/1` or `accept/1` message.
- An active or passive object sends a second active object a message that invokes a recursive method that contains a `waitToAccept/1` or `accept/1` message.

In general, a recursive method invoked by an active object and containing a `waitToAccept/1` and/or an `accept/1` message cannot be translated without user intervention. However, if the method is tail recursive with no *out* arguments, the method can be translated as an iterative loop using the transformation illustrated in figure 3.8. Otherwise, the recursive method must be restructured; this can sometimes be accomplished by replacing the recursive method with a functionally equivalent iterative method as illustrated in figure 3.10. Alternate methods for restructuring MLog code are discussed in section 4.2.3.

MLog code:

```
methodA(0,Y) :- theValueOf(result,Y).
methodA(X,Y) :- waitToAccept(Z), X1 is X-1, methodA(X1,Y).
mode: methodA(in,out).
```

Restructured MLog code:

```
methodA(X,Y) :- makeTheValueOf(count,X), repeat,
                theValueOf(count,X1),
                waitToAccept(Z), X2 is X1-1,
                makeTheValueOf(count,X2), X2=0,
                theValueOf(result,Y).
```

Figure 3.10 Restructuring a recursive method — an example

	Acceptor characteristics	Method characteristics	Comments	Translation
1	passive caller = acceptor		A passive object is sending a message to itself.	a procedure call The current object, class, and process are unchanged.
2	passive caller \= acceptor			a procedure call The current object and class are updated. The current process is unchanged.
3	active caller \= acceptor current process \= acceptor	Does not use an extended rendezvous	The case of caller = acceptor is not possible since the caller and acceptor have different modes. The method is invoked indirectly from the current process;(i.e., an external object).	a task entry call The current object, class, and process are updated.

Table 3.1: Criteria for translating messages from passive objects. (Continued . . .)

	Acceptor characteristics	Method characteristics	Comments	Translation
4	active caller $\neq$ acceptor current process $\neq$ acceptor	Does use an extended rendezvous.	as per item 3.	a task entry call under an extended rendezvous transformation (cref. section 4.2.2). The current object, class, and process are updated.
5	active caller $\neq$ acceptor current process = acceptor	Does not contain a WaitToAccept/1 message.	When the current process equals the acceptor, an active object is calling itself indirectly. This causes the message to have the semantics of a procedure call.	a procedure call The current object and class are updated. The current process is unchanged.
6	active caller $\neq$ acceptor current process = acceptor	Does contain a WaitToAccept/1 message.	as per item 5 Cannot translate as a procedure since the method contains a waitToAccept/1 which can only be translated in Ada tasks. Cannot translate as a task entry since the the object is calling itself and this causes deadlock in Ada.	none These cases must be removed prior to translation (cref. section 4.2.3).
notes: (1) Occurrences of 'waitToAccept/1' in the table imply 'waitToAccept/1' or 'accept/1.'				

Table 3.1: Criteria for translating messages from passive objects.

	Acceptor characteristics	Method Characteristics	Comments	Translation
1	active caller = acceptor	Does not contain a waitToAccept/1 message. The message is not a recursive call to the current method. note(1)	An object sending a message to itself results in the message having the semantics of a procedure call.	a procedure call The current object, class and process are unchanged.
2	active caller = acceptor	Does contain a waitToAccept/1 message. The message is not a recursive call to the current method. note(1).	Cannot translate as a procedure since Ada procedures cannot accept a rendezvous. Cannot translate as a task entry call since this represents the acceptor task calling one of its own entries.	none This case is resolved by preprocessing the code using fold/unfold transformations (cref. section 4.2.3).
3	active caller = acceptor	Does contain a waitToAccept/1 message. Message is a recursive call to the current method. note(1).	Cannot translate as a procedure since Ada procedures cannot accept rendezvous. Cannot translate as a task entry since task entries cannot be recursive.	none A recursion must be removed prior to translation (cref. section 4.2.3).
notes: (1) methodA :- . . . , methodA. This is an example of a message being a recursive call to the current method. (2) Instances of 'waitToAccept/1' in the table imply 'waitToAccept/1' or 'accept/1.'				

Table 3.2: Criteria for translating messages from an active object to itself.

	Acceptor characteristics	Method characteristics	Comments	Translation
1	active caller $\neq$ acceptor	Does contain waitToAccept/1 message. Method is recursive.	Cannot translate as a procedure since two active objects want to communicate. Cannot translate as a task entry since entries are not recursive.	none. These cases must be removed prior to translation, [cref. section 4.2.3].
2	active caller $\neq$ acceptor	Does not use extended rendezvous.		a task entry call The current object, class, and process are updated.
3	active caller $\neq$ acceptor	Does use extended rendezvous.	The method is invoked directly by an external active object.	a task entry call under an extended rendezvous transformation The current object, class, and process are updated.
4	active caller $\neq$ acceptor	Does use extended rendezvous.	Caller originates the message from a method that is translated as a procedure (IAW table 3.1 item5, table 3.2 item 2) note(1).	a task entry call under an extended rendezvous transformation The current object, class, and process are updated.
5	passive			a procedure call The current object and class are updated. The current process is unchanged.
notes: (1) Item4 is treated the same as table3.1 item 4, since in the translated code a procedure is calling a task entry which uses the extended rendezvous. (2) Instances of 'waitToAccept/1' in the table imply 'waitToAccept/1' or 'accept/1'.				

Table 3.3: Criteria for translating messages from an active object to another object.

caller object characteristics	acceptor class characteristics	method characteristics	MLog comments	translation
active/passive	some active objects exist note(1)	may or may not contain a waitToAccept/1 or accept/1 message. note(2)	Caller will rendezvous with an active object of the acceptor class.	Call to a task entry under the anonymous message transformation
active/passive	no active objects exist note(1)	as above	Caller will block until an active object becomes available.	as above
notes: (1). The acceptor of an anonymous message must be an active object by definition. (2). Methods invoked as a result of anonymous messages are limited to using a rendezvous that does not depend on caller or acceptor data, due to the constraints of the anonymous message transformation (cref. section 5.2.4).				

Table 3.4 Criteria for translating anonymous messages.

Semantically an anonymous message results in a rendezvous between the calling object and one of the active objects of the acceptor class. Table 3.4 presents a summary of possible modes for callers and acceptors of anonymous messages. All anonymous messages result in the same transformation (cref. §5.2.4).

3.4 Inheritance

Inheritance allows new classes to be defined as extensions of previously defined ones. Ada does not support inheritance directly. Therefore, a package can completely emulate a class only if the package encapsulates the subprograms equivalent to the methods of a class, including the inherited methods. One solution is to place a copy of all methods inherited by a class into the equivalent Ada package. This emulation of full inheritance by the Ada code would produce functionally equivalent Ada and MLog code, however it has the disadvantage of adding overall code bulk and its attendant program obscurity [19, 16]. The potential increase in program size eliminates this transformation for any systems where space is a constraint.

in MLog:

```
class: classB of classA.
methodA :- methodB.
```

note: methodB/0 is inherited from classA.

in Ada:

```
package CLASSB is
    procedure METHODA(...);
    procedure METHODB(...);
end CLASSB;
```

```
package body CLASSB is
    . . .
    procedure METHODA(...) is
        METHODB(...);
    end METHODA;

    procedure METHODB(...) is
        CLASSA.METHODB(...);
    end METHODB;
end CLASSB;
```

Figure 3.11 Translation of inheritance using *call-through* procedures.

Seidewitz [13] presented an alternative transformation based on inherited operations being implemented as *call-throughs* to the operations of the superclass where they are defined. Under this transformation every method of a class including inherited methods would be represented as a procedure in the class' equivalent Ada package as illustrated in figure 3.11 where methodB/0 has been implemented as a call through to its superclass. The call-throughs are inefficient due to a large number of redundant procedure calls, however, these transformations do emulate full inheritance, if methods are not redefined.

The transformations from MLog to Ada do not attempt to emulate complete inheritance. The functional equivalence required between the MLog and Ada code is achieved when packages contain only a translation of the inherited methods that are *required* by the class. An inherited method is required by a class when: (1) it is invoked by mes-

Chapter 4

Pretranslation Processing

MLog has melded the three separate programming approaches of object-oriented programming, logic programming and concurrent programming into a single language. Ada, on the other hand, is a procedural language that supports concurrent programming and some object-oriented features. The differences in the languages are too great to allow a simple translation of MLog code into Ada code. Preprocessing of MLog code is required to provide additional information for the translator which will allow the resolution of these differences. Information generated by automated preprocessing and information provided explicitly by the user are both necessary. This chapter will specify the preprocessing requirements of the translation in terms of:

- what extra information is required;
- why the information is required;
- how the information will be acquired.

The preprocessing will be discussed with respect to inheritance and message passing, communications and concurrency, declarative versus procedural semantics and polymorphism in MLog.

4.1 Inheritance and message passing

Inheritance and message passing utilize MLog's linguistic support for dynamic binding. In Ada, the user provided `case` statements emulate this dynamic binding but are cumbersome and inefficient. The previous chapter translates messages to self with minimal use of dynamic binding. This is possible because: (1) the choice of methods that match the message selector is limited to those on a single inheritance path in the class hierarchy and (2) all messages to self are translated as procedure calls, (the exact class is not required for the translation, instead the class from which the method is inherited is sufficient).

This inheritance transformation works correctly if the inherited method does not contain messages that select methods which are redefined within that part of the class hierarchy spanning the class that defines the method and the class that inherits the method. Any message to `self` can be bound statically to the selected method if the method is defined only once in the class hierarchy path containing the inheriting and defining classes of the method. If there is more than one definition of the method on the path, the selection of the correct method requires dynamic binding since it depends on the class of the current object. The identification of the current object is run-time dependent.

Figure 3.12 illustrates this situation. Executing `methodE/0` causes `methodA/0` inherited from the superclass to be processed which sends the message `methodB/0`. Since the current object belongs to `classC`, `methodB/0` defined in `classC` is processed. Attempts to match a message to a method always begin in the class of the current object unless the message includes the explicit target `super`. If the example were translated using the suggested transformations, procedure `CLASSC.METHODC` would call procedure `CLASSB.METHODA`. Procedure `CLASSB.METHODA` would call procedure `CLASSB.METHODB` in accordance with Ada's scoping rules which have no built in support for inheritance. Under a correct transformation, procedure `CLASSB.METHODA` would call procedure `CLASSC.METHODB`.

This anomaly in the transformation is removed by reintroducing dynamic binding into the translated code of methods whose messages select redefined methods. A case statement is used in the procedure initiating the call to a redefined method, which determines the correct version to call based on the class of the current object; (i.e., the argument `ENVIRONMENT.CUR_CLASS`), [cref. figure 3.12].

A case statement is required to translate a message that invokes a method which is defined more than once on a path in the MLog class hierarchy. The classes that appear in the case statement are those that define or inherit the multiply-defined method, and are on a single path of the class hierarchy tree. A static analysis of the MLog code can then determine the correct copy of the method to invoke for each possible value of the current class. The current class will always be one of the classes in the case statement as

in MLog:

```
class: classC of classB.   class: classB of classA.
methodB :- methodD.       methodA :- methodB.
methodE :- methodA.       methodB :- methodC.
```

notes: classB is the superclass of classC.

methodB/0 is defined in both classC and classB.

methodE/0 is being processed by the current object.

The current object belongs to classC.

in Ada:

package body CLASSC is

```
...
procedure METHODE(...) is
...
  CLASSB.METHODA(...);
end METHODE;
```

```
procedure METHODB(...) is
...
  METHODDD(...);
end METHODB;
end CLASSC;
```

package body CLASSB is

```
...
procedure METHODA(ENVIRONMENT: in ENV_RECORD;
                  SUCCESS: in out BOOLEAN) is
...
  case ENVIRONMENT.CUR_CLASS is
    when CLASSB =>
      CLASSB.METHODB(...);
    when CLASSC =>
      CLASSC.METHODB(...);
  end case;
end METHODA;
```

```
procedure METHODB(...) is
  METHODDC(...);
end METHODB;
end CLASSB;
```

Figure 3.12 Translating inheritance for messages with target *self* that select redefined methods.

these are the only classes whose objects can access the method. Due to Ada's visibility rules, any method that is redefined (appears in a case statement), must appear in the specification of packages corresponding to the classes in the case statement.

The previous discussion referred to messages sent to `self` that were translated within procedures encapsulated in packages (which are non-active entities). These translations require case statements since the current object and its class can be determined only at run-time. In contrast, messages sent to `self` that are translated within tasks have a known current class since each task is associated with a fixed current object (object-id) and current class. Therefore, the correct version of a redefined method can always be determined statically from the class hierarchy, and case statements are not required.

3.4.2 Messages with explicit targets

When an external object sends the message that invokes an inherited method of the class, the translator uses a call-through to implement the inherited method. The call-through is translated as both a procedure and a task entry since static analysis cannot determine the mode of the explicit target. Figure 3.13 illustrates the transformation.

This is equivalent to the transformation proposed by Seidewitz. Figure 3.13 shows the task type that models active objects of classB. A procedure and a task entry are defined to accommodate the call-through for `methodB/0`. The procedure call is not replaced by a call to the actual superclass since that would expand the alternatives of the case statements used to implement dynamic binding for redefined methods. At present the only alternatives in the case statement consist of classes on a single path of the tree. If objects from classes on different paths of the tree could call directly, then the case alternatives would have to consider all classes where the method is defined or inherited in the program.

Dynamic binding and its associated case statements are not required in the transformation for messages with explicit targets regardless of whether the selected method is redefined. For active objects the object-id uniquely identifies the task to call, for passive objects the call-through procedure is lexically located in the target's class, and the correct

in MLog:

```
class: classB of classA.
methodA(X) :- X-methodB.
notes: methodB/0 is inherited from classA.
      X is an instance of classB.
in Ada:
```

```
package CLASSB is
  procedure METHODDA(...);
  procedure METHODDB(...);
  task type ACTIVE_CLASSB is
    entry METHODDA(...);
    entry METHODDB(...);
  end ACTIVE_CLASSB;
end CLASSB;

package body CLASSB is
  ...
  procedure METHODDA(...) is
    - set current object to X
    - set current class to classA
    if (mode of X is passive) or
       (current object equals current process) then
      METHODDB(...);
    else
      - get PTR, the pointer to the task representing the current object
      - set current process to X
      PTR.METHODB(...);
    end if;
  end METHODDA;

  procedure METHODDB(...) is
    CLASSA.METHODB(...);
  end METHODDB;
end CLASSB;
```

Figure 3.13 Translating inheritance for messages with explicit targets.

version of the method to call is statically bound in the call through procedure. How the target's class is determined is addressed in section 4.1.

Since the message passing syntax in MLog is identical whether the explicit target is `super` or an object name, the translation must allow for both cases.⁷ Since `super`, by definition, is the current object sending a message to itself, the message will always result in a call to a procedure in Ada. Determining whether an explicit target is instantiated to `super` usually is not possible by a static analysis of the code. A run-time check is incorporated into the equivalent Ada code by using an *if* statement. However if the target is `super`, the lexical location of the message allows static binding to the matching method for the message selector. Figure 3.14 illustrates the translation of a message with an explicit target. The current object, current class and current process are not modified when the explicit target is `super`. Refinements to the translation of messages with explicit targets are presented in §4.2.3.

⁷ The explicit target could also be instantiated to *self* which means the object is sending the message to itself. Self is the default when the message does not include an explicit target. The translations presented assume messages to self are restricted to the default syntax.

in MLog:

```
class: classB of classA.
methodA(X) :- X-methodB.
notes: methodB/0 is inherited from classA.
      x is an instance of classB.
```

in Ada:

```
procedure METHODDA(ENVIRONMENT : in ENV_REC_T;
                  X : in STRING;
                  SUCCESS : in out BOOLEAN) is
    . . .
    if X = super then
        CLASSA.METHODB(ENVIRONMENT, SUCCESS);
    else
        -- set current object to X
        -- set current class to classA
        if (mode of X is passive) or
           (ENVIRONMENT.CUR_PROCESS = X) then
            CLASSB.METHODB(NEW_ENVIRONMENT, SUCCESS);
        else
            -- get PTR, the pointer to the task
            -- representing the current object
            -- set current process to X
            PTR.METHODB(NEW_ENVIRONMENT, SUCCESS);
        end if;
    end if;
end METHODDA;
```

Figure 3.14 Translation of a message with an explicit target.

Chapter 4

Pretranslation Processing

MLog has melded the three separate programming approaches of object-oriented programming, logic programming and concurrent programming into a single language. Ada, on the other hand, is a procedural language that supports concurrent programming and some object-oriented features. The differences in the languages are too great to allow a simple translation of MLog code into Ada code. Preprocessing of MLog code is required to provide additional information for the translator which will allow the resolution of these differences. Information generated by automated preprocessing and information provided explicitly by the user are both necessary. This chapter will specify the preprocessing requirements of the translation in terms of:

- what extra information is required;
- why the information is required;
- how the information will be acquired.

The preprocessing will be discussed with respect to inheritance and message passing, communications and concurrency, declarative versus procedural semantics and polymorphism in MLog.

4.1 Inheritance and message passing

Inheritance and message passing utilize MLog's linguistic support for dynamic binding. In Ada, the user provided case statements emulate this dynamic binding but are cumbersome and inefficient. The previous chapter translates messages to self with minimal use of dynamic binding. This is possible because: (1) the choice of methods that match the message selector is limited to those on a single inheritance path in the class hierarchy and (2) all messages to self are translated as procedure calls, (the exact class is not required for the translation, instead the class from which the method is inherited is sufficient).

Messages to explicit targets require dynamic binding because: (1) if the target is active, the class of the target object is required and (2) if the selector invokes a method that is multiply defined, the version selected depends on the class of the current object. Both the previous conditions are run-time dependent.

Message passing to an explicit target is accomplished in MLog by the message `X-methodA`, where `X` is instantiated with the object-id of the target, and `methodA/0` is the name of the method to be executed. Since the value of `X` is usually not explicitly given as a symbolic name in the MLog code, the emulation of dynamic binding causes extensive case statements in the translated Ada code in order to determine the correct procedure call to make based on the value of `X`, or more precisely its class. These case statements, illustrated in figure 4.1, are required for every translation of a message to an explicit target, where the target is an instantiated variable. Depending on the size of the application, these case statements could be quite large as they must include alternatives for each class that defines or inherits the method in the program. This overhead is not conducive to producing efficient code, therefore the user will be required to annotate each explicit target in the MLog code with its classname. For example, `X-methodB` will become `X(classname)-methodB`; this allows the dynamic binding of the method to be replaced by static binding to the method in the class shown in the annotated message. In figure 4.1, if `methodB/0` is defined in `classA` then the entire case statement is replaced by the statement `CLASSA.METHODB(...)`.

When the target is annotated, the classname is derived as follows.

- If the target is an object-id, the classname is the class of the target object.
- If the target is `super`, the classname is the lexical location of the message.

This semantic difference in annotated classnames allows a hierarchy of calls to `super`, but does not allow the same target to be instantiated to `super` and to an object-id at different times during program execution (unless the annotated classnames are the same).

This annotation of messages with explicit targets also permits other optimizations in the translated code, as outlined below.

In MLog:
 class: classA of object.
 methodA(X) :- X-methodB.

In Ada:

```
package body CLASSA is
  ...
  procedure METHODDA(ENVIRONMENT : in ENV_RECORD_T;
                    X : in OBJ_NAME;
                    SUCCESS : in out BOOLEAN) is
    ...
  begin
    Y := GET_CLASS_OF(X);
    case Y is
      when CLASSA=>
        CLASSA.METHODB(...);
      when CLASSB=>
        CLASSB.METHODB(...);
      ...
    end case;
  end METHODDA;
  ...
end CLASSA;
```

note: The case statement must include all classes that define or inherit methodB/0.

Figure 4.1 Translation of an unannotated message to an explicit target.

- It allows some minimization of the number of entries on tasks. All MLog active instances are defined as tasks in Ada. The message X-methodA indicates that methodA/0 must be translated as a task entry in the equivalent Ada code. If methodA/0 is defined in, or inherited by several classes, then, each class must translate methodA/0 as a task entry. After annotating the target with its class name; (e.g.,X(classB)-methodA), the message indicates that only the translation of classB needs to include methodA/0 as a task entry.
- It increases the efficiency of typing MLog since messages to explicit targets that are annotated with a class name, result in at most two choices for the invoked method.

The choice for the invoked method depends on whether the target is instantiated to an object-id or to super. This is discussed in detail in section 4.4.2.

In addition to annotating messages with explicit targets the correct translation of inheritance requires that two sets of information be collected.

- Any method, including system methods, can be redefined by the user. A list of all methods that have been redefined in subclasses is required. Each list element will contain the name of a method, and the class(es) where it is lexically defined/redefined. The list can be generated automatically and is used to ensure that the correct version of a method is invoked in the code produced by the translation; i.e., used to format the case statements that translate a message to self where the selected method is redefined.
- A class inherits all the methods of its superclasses, but usually requires only a subset of the methods. A list, for each class, of all methods that it needs to inherit is required. Each element of the list contains: (1) the name of a method used in the class (selected by a message with an explicit target originating from another object) but not lexically located in the class and (2) the name of the class where the method is lexically located. The list can be generated automatically and is used to ensure that all methods required in a class are present in the equivalent Ada code.

The above information will allow the Ada translation to be functionally equivalent to the MLog code without supporting complete inheritance in Ada.

4.2 Communications and Concurrency

Interprocess communications (IPC) appears to be handled equivalently by MLog and Ada. Both use the semantics of a procedure call when the acceptor is passive and the semantics of a rendezvous when the acceptor is active. However MLog has substantially expanded on the IPC facilities available in Ada. The MLog ability to send anonymous messages is not available in Ada, although it can be emulated to some extent by a dispatcher paradigm [38]. MLog's rendezvous is more powerful since it considers both caller data (message

content) and acceptor data when an acceptor decides to initiate a rendezvous; Ada's rendezvous considers only the acceptor's data. A more subtle distinction between the two languages relates to where a rendezvous can be accepted. In MLog, the acceptance can occur in any method, not just the `begin/0` method which is considered the mainline code of the active instances of a class. In Ada, rendezvous acceptance can occur only in a task entry (which is lexically located in the mainline code of a task); that is, it cannot be accepted within procedures.

The IPC differences relate to: (1) messages to anonymous targets, (2) conditions considered when accepting a rendezvous, and (3) rendezvous location. The impact of each of these differences and the preprocessing requirements they generate are discussed in the following sections.

4.2.1 Messages to anonymous targets

Anonymous messages in MLog, allow a caller to send a message to a class of instances; i.e., one of the active instances will accept the rendezvous and process the message. MLog has an underlying dispatcher algorithm that supports anonymous messages. In Ada, the dispatcher algorithm must be coded explicitly in order to support translation of anonymous messages. A caller initiating an anonymous message, would call the dispatcher to get the name of an instance, send a message to the instance for action, and then release the instance back to the control of the dispatcher.

To effect the transformation from the anonymous message to the explicit dispatcher the following data is required.

- Classes, whose instances are the target of anonymous messages require dispatchers. A list of these classes can be generated as part of the translation process due to the distinctive syntax of the anonymous message in MLog. The list will be used to ensure that dispatchers are generated and that they are informed of all active instances they control.
- In Ada, the dispatcher must know when an instance is returned so that new work can be assigned. The user can most easily provide this information by inserting

caller's code: `methodA :- classA(X)-methodB`

new caller's code:

```
methodA :- classA(X)-methodB,
           dispatcherForClassA-releaseInstance(X) .
```

Figure 4.2 Message to an anonymous target — annotated version.

the `releaseInstance/1` message into the caller's code at the point where the instance is no longer required. The information is not required in MLog's anonymous messages since the acceptor task queries its own state and the caller's data to determine if it can accept a new rendezvous (a message or an anonymous message). In figure 4.2, the message selector, `releaseInstance/1` indicates to the dispatcher of classA that instance X is no longer required by the caller.

The option of having the acceptor instance inform the dispatcher when it was free was not chosen since the acceptor would have to check its state explicitly at the end of each method to determine if it was free, and when necessary, call its dispatcher with the `releaseInstance/1` message. This extra check of internal state is an unnecessary overhead both with respect to runtime efficiency and user coding, and is avoided if the caller instances send the `releaseInstance/1` message to the dispatchers.

- The previous discussion assumed that all objects controlled by the dispatcher have the same internal state; i.e., all objects can accept a message from any caller. This assumption underlies the standard dispatcher paradigm. In MLog, anonymous messages can generate extended rendezvous where the satisfactory completion of the rendezvous depends on caller data and acceptor state. This ability to query the acceptor's state implies that all objects may not have the same state, and therefore all objects may not be able to accept a message from any caller. For the proposed translation of anonymous messages to function correctly, the user must ensure that all objects under the control of a dispatcher have identical states. This is accomplished when all anonymous messages result in simple unguarded rendezvous. In these cases,

the state of the acceptor is irrelevant to the successful completion of the rendezvous, and the dispatcher paradigm will work correctly in the equivalent Ada code.

4.2.2 Conditions for rendezvous

In MLog, an active object can accept callers for rendezvous based on caller and/or acceptor data; in Ada callers for rendezvous are selected based only on acceptor data. In MLog a caller and an acceptor can complete a rendezvous in the following ways.

- The rendezvous always completes; this is equivalent to a unguarded entry in Ada.
- The rendezvous completes based only on the state of the acceptor; this is equivalent to the guarded entry in Ada.
- The rendezvous completes based on sender data or on a combination of sender data and acceptor state.

In the first case, the translator can generate the Ada rendezvous directly from the MLog code with no input from the user. In the second case, there is no way of statically determining, for a given MLog method, if the rendezvous is dependent on acceptor state. The user must provide this information by embedding a `conditionsEnd/0` message in the acceptor method following the messages that act as guards on the rendezvous.

Example:

```
caller:  method :- X-methodA(W,Y) .
```

```
acceptor: methodA(W,Y):-theValueOf(status,go),
           conditionsEnd, ... .
```

Note: `theValueOf/2` message is the guard condition on `methodA/2` .

The user inserts the `conditionsEnd/0` message to indicate to the translator that the rendezvous will now complete successfully; i.e., the guard conditions are now satisfied.

In the third case, where the completion of the rendezvous depends on caller and acceptor data, Ada has no simple equivalent code. In Ada, to see the caller data the acceptor must start the rendezvous, and once the rendezvous is started the caller can only be released by terminating the rendezvous. The caller cannot be put back on the

queue of the entry which is equivalent to what MLog does. Karam et al [39] determined two possible transformations each based on the idea of a private semaphore. Effectively, the rendezvous is started and if the caller data and acceptor data are satisfactory, the rendezvous proceeds normally. If the caller data and acceptor data are not satisfactory then the caller is told to wait on another entry; the caller's message and its arguments are placed on a pending callers queue. The messages on the queue are evaluated each time the acceptor becomes active and any messages that can be evaluated successfully are processed. The caller is then informed at the entry it is waiting on. These transformations are given in detail in figures 4.3 and 4.4. These transformations differ with respect to where the caller waits when the conditions for rendezvous are unsatisfactory. Under transformation 1, the caller waits on an entry in the acceptor task, and under transformation 2 the caller waits on an entry in its own task.

The pretranslation requirements depend on which transformation is selected. If transformation 2 is used, no further input from the user is required. If transformation 1 is used then information is required from the user. Two approaches are possible.

- Each caller has its own entry on the acceptor, where it waits when the conditions for rendezvous are unsatisfactory. In this case the user must provide the names of all callers to the acceptor. This can be derived from the application when the user has coded his program to name explicitly each active or passive instance that calls objects of the acceptor's class.
- The caller is provided with one of a pool of entries available in the acceptor. In this case the user must provide the maximum number of waiting callers at any one time. If the user underestimates the maximum number of simultaneously waiting callers a possible error condition has been added to the translated application (deadlock).

Additional discussion on the advantages/disadvantages of the two transformations with respect to their translation into Ada is presented in section 5.2. The decision on which transformation to choose is application dependent, and therefore should be at the discretion of the user.

RESULTING ALGORITHM, CALLER-SIDE

```

call Acceptor.entry_name(x,index)
if index assigned begin
    call Acceptor.wait[index](x)
end

```

RESULTING ALGORITHM, ACCEPTER-SIDE

```

while ( not done ) begin
    scan pending caller queue (front to back) for
    some caller with extended rendezvous conditions
    that are now true
    if conditions begin
        remove caller info from queue
        set guard(index) = true
        do service for caller
    end else begin
        select begin
            accept entry_name(x,next_entry) do
                if conditions begin
                    do service for entry_name
                end else begin
                    assign some entry index to next_entry
                    append x on pending caller queue
                    using assigned next_entry as caller id
                end
            end accept
        or
            when guard(index) ==> accept wait[index](x) do
                end accept
        end select
    end
end while

```

Figure 4.3 Pseudo-code for Transformation 1 of an extended rendezvous.

From the previous discussion, there is no requirement for the insertion of a conditionsEnd/0 message to translate an extended rendezvous. However, its insertion prevents the generation of extra Ada code required to translate a method as if it were using an extended rendezvous, when in fact it is using a standard Ada rendezvous; i.e., an

RESULTING ALGORITHM, CALLER-SIDE

```

call Acceptor.Entry_name(x,status)
if status = rejected begin
    accept Wait_entry(x) do
    end accept;
end

```

RESULTING ALGORITHM, ACCEPTER-SIDE

```

while ( not done ) begin
    scan pending caller queue (front to back) for
    some caller with extended rendezvous conditions
    that are now true
    if conditions begin
        remove caller info from queue
        do service for caller
        call Caller.wait_entry(x)
    end else begin
        select begin
            accept entry_name(x,status) do
                if conditions begin
                    do service for entry_name
                end else begin
                    append x and caller id on
                    pending caller queue
                    status = rejected
                end
            end accept
        or
            ...
        end select
    end
end while

```

Figure 4.4 Pseudo-code for Transformation 2 of an extended rendezvous.

unguarded accept. For this reason the user will be required to explicitly indicate that an extended rendezvous exists in the method by inserting a conditionsEnd/0 message at the point where all conditions for a successful rendezvous are satisfied.

In MLog, some or all of the conditions for a successful rendezvous can be embedded in a method's clause head, and unification determines if the conditions for a successful

rendezvous are met. When all the conditions are decided by unification, a static analysis of the code can determine if an extended rendezvous is used (since the code is annotated with argument modes), and the `conditionsEnd/0` message need not be added to the method.

Example:

```
caller:    methodA :- Z-methodB(X,Y) .
```

```
acceptor: methodB(a,Y) :- methodC .
```

Note: The condition for successful rendezvous is $X = a$.

In the example when $X \neq a$, the caller is placed on the pending caller queue. Since caller data does not change while on the queue, the condition $X=a$ will never be met, and the caller will remain blocked forever. Whenever an extended rendezvous depends on caller data, and not on acceptor state, the rendezvous either succeeds immediately or remains blocked indefinitely.

In addition to the insertion of `conditionsEnd/0` messages by the user, the pretranslation of the extended rendezvous creates a list for each class, of all methods that use the extended rendezvous. This list is generated automatically, and is used to ensure that callers to these methods are translated correctly.

4.2.3 Rendezvous location

MLog accepts a rendezvous when an active object sends itself a `waitToAccept/1` or an `accept/1` message. These messages can occur in the body of any MLog method. MLog methods are translated as Ada procedures, but in Ada, procedures cannot accept a rendezvous; a rendezvous can be accepted only in a task entry. MLog methods that are translated as Ada task entries are the methods in an object that process messages from external objects. These methods are considered *entries* into the active object(s) of a class. Within a class the set of methods that contain the `waitToAccept/1` or `accept/1` messages may not be equivalent to the set of methods that are translated into Ada task entries. When the sets are not equivalent, *unfold* transformations must be applied to the MLog code. Unfolding refers to the replacement of a message in a method by an

inline expansion of the method [40]. Unfold transformations preserve partial correctness (non-logical features, such as the cut can be handled by preprocessing) [40].

Example:

```
methodA :- methodB,methodC.
methodB :- methodD,methodF.
```

Unfolding methodA/0 with respect to methodB/0 gives:

```
methodA :- methodD,methodF,methodC.
```

Note: methodB/0 is said to be folded into methodA/0.

methodA/0 is unfolded with respect to methodB/0.

Unfold transformations are applied to ensure that waitToAccept/1 and accept/1 messages occur only in methods that are entries. Note, the begin/0 method of an object is considered the mainline of the Ada task equivalent to the object and as such can accept a rendezvous. The begin/0 method can contain waitToAccept/1 and accept/1 messages. The begin/0 method is considered an entry for the purpose of unfold transformations.

To determine which messages must be unfolded, it is necessary to know which methods are entries, as these methods terminate the folding process, and which methods contain the waitToAccept/1 and/or accept/1 messages, as these methods determine the messages that require unfolding. Figure 4.5 illustrates the unfolding process.

The fact that waitToAccept/1 and accept/1 messages always occur in methods that are entries is not always a sufficient condition for correct Ada code. The pathological case exists when a method, that is an entry, invokes a second method, that is an entry and is in the same object, and contains a waitToAccept/1 or accept/1 message. Since all messages to *self* are treated operationally as procedures (and are translated as such in Ada), the call to the second method results in the equivalent translated code consisting of an Ada procedure that is trying to accept a rendezvous. To avoid this error condition, the second method must be folded into the first method, even though both methods are entries.

Before Unfolding:

```

class: classA of object.
begin :- waitToAccept(X).
methodA :- methodB, methodC.
methodB :- methodD, methodE.
methodD :- methodF, waitToAccept(X).

```

After Unfolding:

```

class: classA of object.
begin :- waitToAccept(X).
methodA :- methodF, waitToAccept(X), methodE, methodC.
methodB :- methodD, methodE.
methodD :- methodF, waitToAccept(X).

```

Note.

Assume that within classA the only entries are begin/0 and methodA/0. When the waitToAccept/1 message appears in one of these methods, the folding process can terminate. In classA, methodD/0 must be folded into methodB/0, since methodB/0 is not an entry, the unfolded version of methodB/0 must be folded into methodA/0. Since methodA/0 is an entry, the folding process terminates.

Figure 4.5 Example of unfolding messages.

before unfolding:

```

class: classA of object.
begin :- waitToAccept(X).
methodA :- methodB, methodD.

methodB :- methodC,
           waitToAccept(X).

```

after unfolding:

```

class: classA of object.
begin :- waitToAccept(X).
methodA :- methodC,
           waitToAccept(X),
           methodD.
methodB :- methodC,
           waitToAccept(X).

```

Note: methodA/0, methodB/0 and begin/0 are all entries into objects of classA.

Figure 4.6 Example of unfolding messages in entry methods.

In figure 4.6, `methodB/0` remains as an entry for objects of classA since it continues to process messages from external objects. Apart from the pathological case just discussed, Algorithm 1 in figure 4.7 will determine which methods require unfolding.

To this point, the discussion has focused on why unfold transformations are required, and how to determine which methods must be folded/unfolded. Methods that must be folded may require preprocessing (for example to remove recursion and cuts). The user is expected to restructure the MLog code to facilitate fold/unfold transformations when required. These transformations are discussed in the literature [40, 41, 42] and are not pursued here with two exceptions that are peculiar to MLog.

- The method to be folded appears as a message selector in the method to be unfolded.

Example: Fold `methodB/0` into `methodA/0`.

```
methodA :- Y-methodB.
methodB :- methodC, waitToAccept(Z).
```

- The method to be folded contains a `conditionsEnd/0` message, which acts as a directive to the translator. After folding, the `conditionsEnd/0` message is not always required in the unfolded message.

Example: Fold `methodB/0` into `methodA/0`.

```
methodA :- methodB, conditionsEnd, methodD.
methodB :- methodC, conditionsEnd, waitToAccept(Z).
```

After normal unfolding `methodA/0` would contain two `conditionsEnd/0` messages.

The unfold transformations for these cases, which are unique to MLog, are described in the following two sections.

ALGORITHM 1: TO DETERMINE WHICH MESSAGES REQUIRE UNFOLDING

```

-- Prior to commencing this algorithm there exists:
-- (1) a set A of all methods in the class (including inherited
--     methods).

-- (2) a set E of all methods (including inherited methods)
--     of the class that are called from other instances of the
--     class or from other classes; (i.e., the set of entry
--     methods)

begin

step 1 — Collect all methods used to accept a rendezvous.
Form set S of all methods in set A that contain
waitToAccept/1 and/or accept/1 predicates;

step 2 — For each method used to accept a rendezvous, collect any messages that invoke
         the method.
For each method X, in set S, find the set of methods
in set A whose bodies contain a message to method X. Call
this set S1;

step 3 — Unfold any messages that invoke the method used to accept the rendezvous.
Replace each method in S1 by an equivalent method that
has method X folded into it. Call this set S2;

step 4 — If any of the unfolded methods are also in the set of entry methods, replace the entry
         method by its equivalent unfolded version.
-- The methods in S2 become entries if they are in set E.

Replace methods in set E by their equivalent unfolded methods
from S2 where possible;

step 5 — If unfolded methods exist that are not entry methods, repeat the algorithm.
S := S2
repeat from step 2. until all S1 are empty;

end;

```

notes: 1. This algorithm will work more efficiently if classes

Figure 4.7 Algorithm 1: To determine messages that require unfolding. (Continued ...)

higher in the inheritance tree are processed first. Subclasses will inherit the unfolded versions of superclass methods and will not have to repeat the processing for those cases.

2. On completion of the algorithm there exists a set E of methods that require translation as task entries, and a set A of methods that require translation as procedures.
3. If the methods of set A are used for translation as Ada procedures, the procedures will not include any methods that have been modified by unfolding.
4. If the unfolded version of methods is to be used for translation as Ada procedures the set would be formed by replacing methods in set A by equivalent methods in set S2 when possible at step 4.

Figure 4.7 Algorithm 1: To determine messages that require unfolding.

Unfolding messages containing explicit targets and message selectors

The message passing syntax in MLog is identical when an object sends a message to super and when an object sends a message to another object. However, the message passing semantics are different, and these differences have an impact on the transformations applied to unfold the message selector.

In figure 4.8, the unfolding transformation depends on the values of X and Y.

- *X is instantiated to an object name.*

Folding is not required since methodB/0, by definition, acts as an entry in the object, X. (Folding is required only to remove waitToAccept/1 or accept/1 messages from methods that do not act as entries).

- *X is instantiated to super, and Y is instantiated to an object name.*

On unfolding, the message super(classB)-methodB in methodA/0 is replaced by the body of methodB/0 that is lexically located in the superclass of the

Example: Fold methodB/0 into methodA/0.

```
class: classB of classA
methodA :- X(classB)-methodB.
```

```
class: classA of classC
methodB :- Y(classA)-methodC, waitToAccept(Z).
```

Note: classA is the superclass of classB.

Figure 4.8 Example: fold methodB/0 into methodA/0.

class containing methodA/0. In figure 4.9, this transformation results in replacing the message `super(classB):-methodB` in methodA/0 (found in classB) by the body of methodB/0 (found in classA, the superclass of classB). In the general case, the transformation in figure 4.9 is only correct when explicit targets are annotated with their class. The reason for this restriction is explained in the next case.

- *X is instantiated to super, and Y is instantiated to super.*

The standard unfolding of methodA/0 in figure 4.10 would result in:

```
methodA:-super(classB)-(super(classA)-methodC,waitToAccept(X)).
```

There is no concept of `super-(super-pred1)` in MLog, since `super` refers to the superclass of the class where the message selector is lexically located. When an explicit target is annotated with its class, this information is preserved, otherwise on folding, this information on the physical location of the message selector is lost. To accomplish the unfolding of methodA/0 replace the message `super(classB)-methodB` with a copy of the body of methodB/0 as defined in the superclass of classB. This is the more general transformation and subsumes the previous case. After unfolding, the unannotated MLog indicates that the message `methodC/0` to be used is found in the parent of classB, instead of the parent of classA as originally intended. However, with the annotated MLog, the correct class for the `super` call is carried with it, and can be used to ensure a correct translation.

At compile time, the instantiated values of X and Y in the example, presented at figure 4.8, are unknown, therefore the unfolding must accommodate all the situations

Before unfolding:

```
class: classB of classA.
methodA :- super(classB)-methodB.
```

```
class: classA of classC.
methodB :- objectA(classA)-methodC,
          waitToAccept(X).
```

After unfolding:

```
class: classB of classA.
methodA :- objectA(classA)-methodC,
          waitToAccept(X).
```

```
class: classA of classC.
methodB :- objectA(classA)-methodC,
          waitToAccept(X).
```

Figure 4.9 Unfold methodA/0 with respect to methodB/0 — a specific case.

discussed. The unfold transformations result in a disjunction of messages replacing unfolded messages as follows.

- If the explicit target, in a message that is to be unfolded, is instantiated to an object name, then the unfolding is not required, and the message is unchanged.
- If the explicit target in a message is instantiated to `super` unfolding is required. Specifically, the message is replaced by a copy of the body of the selected method as defined in the superclass of the current class.

The unfold transformation is illustrated in figure 4.11.

Before unfolding:

```
class: classB of classA.
methodA :- super(classB)-methodB.
```

```
class: classA of classC.
methodB :- super(classA)-methodC,
          waitToAccept(X).
```

After unfolding:

```
class: classB of classA.
methodA :- super(classA)-methodC,
          waitToAccept(X).
```

```
class: classA of classC.
methodB :- super(classA)-methodC,
          waitToAccept(X).
```

Note: If the explicit targets were not annotated with their class after unfolding, then the result would be:

```
class: classB of classA.
methodA :- super-methodC,
          waitToAccept(X).
```

Figure 4.10 Unfold methodA/0 with respect to methodB/0 — a general version.

Before unfolding:

```
class: classB of classA.
methodA :- X(classB)-methodB.
```

```
class: classA of classC.
methodB :- Y(classA)-methodC,
          waitToAccept(Z).
```

After unfolding:

```
class: classB of classA.
methodA :- (X\=super, X(classB)-methodB);
          (X=super, Y(classA)-methodC, waitToAccept(Z)).
```

Figure 4.11 Unfold methodA/0 with respect to methodB/0 — a complete annotation.

Folding and conditionsEnd/0 messages

ConditionsEnd/0 messages are used to inform the translator: (1) when all conditions for a rendezvous to complete successfully are met (messages preceding them act as Ada guards) and (2) that methods containing conditionsEnd/0 messages are entries into active objects. Folding these methods implies that one entry is being folded into another. This situation occurs since MLog allows entries to be treated as procedures within an active instance. There are two cases.

Case 1:

When a guarded entry, methodB/0 is folded into a unguarded entry methodA/0, the conditionsEnd/0 message can be eliminated.

example: Unfold methodA/0 with respect to methodB/0.

before unfolding:

```
methodA :- methodB, methodC.
methodB :- methodD, conditionsEnd, waitToAccept(X).
```

after unfolding:

```
methodA :- methodD, waitToAccept(X), methodC.
```

If methodA/0 is unguarded, the user is stating that the rendezvous will always be processed immediately on acceptance. If methodA/0, prior to unfolding, calls methodB/0 which has a guard, then when the guard conditions are not met, methodB/0 will fail and the methodA/0 rendezvous will be queued instead of processed. By not guarding methodA/0, the user is stating that the conditions on methodB/0 will always be met when methodB/0 is called from methodA/0. If this is not the case then there is a program error. Therefore the conditionsEnd/0 message in methodB/0 can be eliminated when methodB/0 is folded into methodA/0.

Case 2:

When a guarded entry methodB/0 is folded into a guarded entry methodA/0 the conditionsEnd/0 message of methodA/0 remains valid, and the one in methodB/0 can be eliminated on folding.

example: Unfold `methodA/0` with respect to `methodB/0`.

before unfolding:

```
methodA :- methodB, conditionsEnd, methodC.
```

```
methodB :- methodD, conditionsEnd, waitToAccept(X).
```

after unfolding:

```
methodA :- methodD, waitToAccept(X),
           conditionsEnd, methodC.
```

The `conditionsEnd/0` message of `methodA/0` states that from that message to the end of the method the rendezvous will complete without queuing the caller. Before unfolding, if `methodB/0` appears after the `conditionsEnd/0` message of `methodA/0`, then as in the previous case, the user is saying the guard conditions for `methodB/0` are also met. If `methodB/0` appears before the `conditionsEnd/0` message, then `methodB/0` forms part of the guard conditions on `methodA/0`. In either situation, the `conditionsEnd/0` message of `methodA/0` has precedence. The one exception to the above occurs when `methodB/0` immediately precedes the `conditionsEnd/0` message in `methodA/0`. In this case, the two `conditionsEnd/0` messages have the same effect after folding so either could be retained; however, for consistency the `conditionsEnd/0` message of `methodA/0` will be retained.

4.3 Declarative versus Procedural Semantics

The declarative nature of MLog results in several language characteristics that are not supported in Ada. MLog is non-deterministic⁸, multidirectional⁹ [28] and supports meta programming; Ada is deterministic, directional and does not support meta programming.

4.3.1 Determinism and directionality

A non-deterministic method in MLog can return success or failure and one or many correct answers for a given set of arguments. A deterministic method can also return

⁸ The term non-deterministic is used exclusively in the declarative sense in this thesis. The term is not used to reference the non-determinism inherent in concurrent processing.

⁹ Mellish described a multidirectional Prolog procedure, which is equivalent to a MLog method, as a procedure where there is no distinction between inputs and outputs, the unknown values being computed from the values in whatever way is required.

success or failure but can only return one correct answer for a given set of arguments. A deterministic method is one where:

- each clause apart from the last includes a cut as a conjunct; and,
- each message which occurs not before a cut in one of the clauses is itself determinate [28].

Since Ada is a deterministic language and does not support backtracking. MLog code that is to be translated must be made deterministic by the user prior to applying the transformations. This restriction is placed on the MLog code since the translation of non-deterministic code would require an Ada interpreter/compiler for MLog which would be quite inefficient [cref. §2.3].

MLog supports multidirectional methods which are equivalent to Mellish's multidirectional Prolog procedures. The direction of a method is determined by the mode of each of its arguments. The modes available in MLog and Ada are *in*, *out* and *inout*. In Ada, procedures are directional, and if a multidirectional procedure is required, it is written as separate directional procedures. This restriction will be imposed on the MLog code; furthermore the user must state explicitly the direction of each argument of a method for use by the translator. If multiple modes are allowed for all arguments of methods then there are two options for translation; both inefficient.

- Option 1 — The translator assigns a unique name to a method for each set of argument modes, and then inserts the appropriate new message name each time the method is called. This option would require the explicit addition of modes by the user to each message that selected a multidirectional method.
- Option2 — The translator adds a new argument to the equivalent Ada procedure to indicate which of the possible modings of the procedure was being used. The procedure would then use a case statement to determine the appropriate code to execute. This option incurs a runtime cost.

Both options effectively produced separate procedures for each possible set of argument modes in the corresponding MLog method. The cost of this overhead in the translation

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

to and/or execution of the Ada code is not considered worthwhile since much MLog code is used in one direction. MLog code is based on the Prolog model and much Prolog code is used in only one direction [28]. Since MLog methods will be directional, the mode for each argument of a method need only be set once; i.e., where the method is defined, instead of on each call to the method.

The restriction of MLog to directional methods is limited to user-defined methods. MLog system methods whose arguments are bi-directional may be used as multidirectional; however, the mode of each argument of the methods must be given on each call, so that a correct translation can be produced.

Within a directional method, only the modes *in* and *out* are supported by the translator. The *inout* mode in MLog is used for partially instantiated structures. These are disallowed as method arguments due to the runtime expense of translating these partially instantiated data structures. The expense results from two Ada limitations:

- it is unable to pass variables as arguments (as opposed to the value represented by the variable);
- it is unable to unbind variables if a procedure is deemed to have failed.

The first case will be discussed in the next section on meta programming.

In MLog, unbinding variables is handled automatically during backtracking. In Ada, an audit trail would be required so that variables in partially instantiated data structures could be unbound if the equivalent MLog method eventually fails. Since the variables can be set in any method that terminates, the audit trail could conceivably be started on the first procedure call and continue to grow for the duration of the program's execution. Furthermore since Ada is concurrent, an audit trail would be required for each thread of control in the program. These audit trails are a form of backtracking control and are therefore more appropriately discussed under Ada compiled MLog. The execution time overhead generated by these audit trails would mitigate against the transformations producing reasonably efficient code.

Since partially instantiated arguments in MLog methods are disallowed, the user must ensure that all *in* arguments are completely instantiated, and that all *out* arguments are uninstantiated variables for each message in the method. With this restriction, if a method fails, then there is no requirement for audit trails in the translated code, since there is no effect on the value of arguments. *In* arguments are unchanged, and *out* arguments regardless of if they have been set incorrectly by a failed method, are treated as variables on the next call. This is possible because in MLog the value of the *out* argument was unbound when the method failed. In Ada the equivalent result is obtained by adding an new argument to each procedure to indicate whether the procedure failed or succeeded. If the procedure has failed, the *out* arguments are treated as variables; if the procedure has succeeded then, the *out* arguments are treated as values. Thus, modes in the equivalent Ada code will be limited to *in* and *out*, and the *in out* mode will not be utilized.

4.3.2 Meta programming

MLog supports methods equivalent to Prolog's meta-logical predicates which "query the state of the proof, treat variables (rather than the terms they denote) as objects of the language, and allow the conversion of data structures to goals." [43]. Meta programming has been disallowed for MLog programs that will be translated into Ada.

Since Ada does not support variables as program objects (as opposed to the value they represent), the translation from MLog to Ada code would have to include code capable of recognizing when arguments were data and when they were variables. One solution would be to represent each variable as a record with two fields; a value field, and a tag field to indicate whether the value was set or not set. Additional code would also be required to update these data tags as program execution proceeded. This solution results in time and space costs. An alternative to data tags is to use default values for all data types in Ada; these default values would indicate that the value was not set. Since Ada supplies only a default value for access types, the use of default values would unilaterally restrict the user from the complete range of values available for a type under Ada. For example, the character type includes the ASCII character set; to support default

values, one element of the ASCII set would be chosen arbitrarily to act as the default value. Neither solution is acceptable, therefore, the ability to treat variables as program objects is not supported by the translator.

Queries on data argument types can be done in Ada only if the type information is passed explicitly, as a parameter in the procedure call, or if the variable is made into a record with the second field indicating type information. As discussed previously, the overhead this entails is not worthwhile and the ability to query the type of an argument is disallowed. Also, MLog code is to be typed as part of the translation process, thus, the ability to query type information will not be required. Argument types are queried directly in MLog through the method `isA/2` (e.g., `X isA integer`), and indirectly through unification, when the argument in a method is not the same type in different clauses of a method.

Example:

```
methodA(1) :- methodC.
methodA([X]) :- methodD.
```

`methodA/1` can be rewritten as follows:

```
methodA(X) :- X isA integer, X == 1, methodC.
methodA(X) :- X isA list, methodD.
```

In the example, the type of the argument is being used to determine which clause will be selected, thus, the argument of `methodA/1` does not have a consistent type in all clauses. The arguments of a method must be consistent from clause to clause if the method is to be translated into Ada, since Ada procedures accept only one type for each argument. Both direct and indirect querying of type data are disallowed by the translator.

Meta-programming also allows the conversion of data structures into program code, effectively allowing self-modifying programs. This is counter to Ada's design goal of easily maintained reliable software for the support of systems through the life cycle [44]. Since the translator algorithms depend on a static global analysis of MLog code, this capability is disallowed with one notable exception in the case of rendezvous processing. Rendezvous processing is accomplished through the system methods, `waitToAccept/1`

and `accept/1`; the argument of these methods is processed as the next message.

Post-rendezvous processing is done by the acceptor object, as a result of the rendezvous, after the caller object is released (unblocked). The post-rendezvous processing is required to satisfy the caller's request but does not require the synchronization of the caller and acceptor objects. In general, post-rendezvous processing is accomplished in MLog through the conversion of a message into a data structure. After a rendezvous terminates there is more than one choice (at compile time) for the post-rendezvous processing since it depends on which method in the object processed the rendezvous. The following example shows MLog code required to perform post-rendezvous processing.

Example:

```
begin:-repeat, waitToAccept(X), postrendez(X), fail.
methodA :- methodB.
methodC :- methodD.
postrendez(methodA) :- methodX.
postrendez(methodC) :- methodY.
```

Note: `methodA/0` and `methodC/0` are both entries.

In the example, the `waitToAccept/1` accomplishes the rendezvous and executes the method given by `X`. The correct post-rendezvous processing for a method is selected by the unification of `X` with the clause heads of `postrendez/1`. If the method, `postrendez/1` is translated as an Ada procedure, the type of the argument of `postrendez/1` must be made consistent from clause to clause. The translator will treat post-rendezvous processing as a special case. The `postrendez/1` method will be the only means used to implement post-rendezvous processing, as the method name will act as a translator directive.

4.4 Polymorphism and MLog

Declarative languages are often polymorphic and MLog falls into this category of typeless languages. Ada, however, is a strictly typed language. Since there is no simple way to emulate polymorphic variables in Ada [13], the polymorphism in MLog must be made explicit by:

- determining the possible types of all instance variables and method arguments in the program (assign types to the MLog code);
- determining the unique set of argument types required on a specific invocation of a method or the unique set of instance variable types required in a specific instance of a class.

The first requirement is an area of current research with respect to Prolog programming [36, 37, 34], since typed programs tend to be less prone to run-time errors because compile time checks catch errors due to type inconsistencies.

The second requirement is needed because Ada does not support polymorphic reuse of code. Ada emulates polymorphic reuse of code by generic packages and subprograms.

Each of these requirements is discussed in some detail.

4.4.1 Towards a type inference program (TIP) for MLog

The current known type inference programs infer types for a restricted set of Prolog. These systems provide a starting point but not a solution to the problem of typing MLog. In order to develop a TIP for MLog a Prolog TIP must be expanded to include support for:

- inheritance,
- concurrency,
- instance variables.

Inheritance

The TIP developed by Xu & Warren [37] used a method of abstract interpretation driven by a Prolog theorem prover. Replacing the Prolog engine by a MLog engine allows the abstract interpreter to correctly use the class hierarchy to process inheritance in an MLog program. However, since the abstract interpreter only recognizes an object-id as a variable, or if instantiated with a type, as a string, no data is available to tell the MLog engine where to start the search for a matching method.

If the matching method is defined only once in the class hierarchy, the abstract interpretation can determine the next method to process.

If the method is defined more than once in the class hierarchy, the MLog engine has no data to indicate the correct method to process. Therefore, all copies of the redefined method must be processed as possible threads of control. If the message is to `self`, the MLog machine can use the class hierarchy to select the correct method. If the message has an explicit target the MLog machine cannot determine the correct version of the method to invoke. As a result, all copies of the method must be matched as possible threads of control in the abstract interpretation. These parallel threads can be expected to introduce some extraneous typings into the solution (the typing of the program). This problem can be solved by using the classname which has been appended to each explicit target in code due for translation. By modifying the abstract interpreter to recognize the annotated class on explicit targets, sufficient information is provided to tell the engine where to start the search for a matching method. Hence, the redundant threads of control are eliminated.

The previous discussion of explicit targets has assumed all targets are object-ids. The explicit target could also be instantiated to `super`. This possibility can re-introduce duplicate threads of control. For `X(classA)-methodA` the following cases exist.

- If `methodA/0` is defined in `classA` and in a class from which `classA` inherits, then two possible methods must be processed by the abstract interpretation.
- If `methodA/0` is defined in `classA`, and not defined in any class from which `classA` inherits, then the target can not be instantiated to `super` and only one choice for `methodA/0` exists.
- If `methodA/0` is inherited by `classA` (not defined in `classA`) then both instantiations of the target select the same method, and only one choice must be processed.

Since both possible selections for the correct method must be processed, duplicate threads of control can be introduced into the abstract interpretation with the attendant probability of introducing extraneous solutions.

Concurrency

The abstract interpreter has no notion of concurrency; all messages are processed sequentially. In order to provide typings for all threads of control in the program, each class with active objects is processed through the TIP using the `begin/0` method as the initial message. The selection of initial messages to input into the TIP should be done by the user, since an automated selection of classes that have active objects is not always possible. This data is available from scanning the MLog code only when the mode argument of `new/3` messages is not an instantiated variable.

The previous discussion on inheritance has approached concurrency from the caller's perspective. The problem for abstract interpretation is more complex from the acceptor's perspective. A discontinuity occurs when a `waitToAccept/1` or `accept/1` message is sent. The TIP cannot recognize either the argument type of the message or the identity of the next logical method to process since it is represented as a meta-variable. Having the abstract interpreter ignore these messages results in part of the execution thread being lost (the part triggered by the rendezvous) and the next method executed is the one normally executed on return from the rendezvous. This does not result in error if the variables appearing in the `waitToAccept/1` or `accept/1` message, do not appear in any other term in the method (except the `postrendez/1` message). In the following example, the abstract interpretation can ignore the `waitToAccept/1` message since its variables do not appear elsewhere within `methodA/1`. The typing of the method represented as `X` is achieved under the caller's thread of control.

Example: `methodA(Z) :- waitToAccept(X),methodB(Z,C).`

When the argument is a single variable as in the example, the variable is always unique to the `waitToAccept/1` message since the translator has disallowed messages as data structures in MLog with the exception of those required for post-rendezvous processing.

Post-rendezvous processing presents a problem since the argument, `X`, in the example can appear as the argument of `postrendez/1` as discussed in section 4.3.2. Although the TIP would provide a type for `postrendez/1` based on its definition, it would not

Before restructuring:

```
methodA(X) :- methodB.
methodE(X) :- methodF.
postrendez(methodA(Z)) :- methodC.
postrendez(methodE(Z)) :- methodD.
```

After restructuring:

```
methodA(X) :- methodB, postrendez_methodA(X).
methodE(X) :- methodF, postrendez_methodE(X).
postrendez_methodA(Z) :- methodC.
postrendez_methodE(Z) :- methodD.
```

Figure 4.12 Restructuring post-rendezvous processing code.

associate this type to the method types the argument represented. The recommended solution is to process the MLog code before using a TIP, to concatenate the post-rendezvous processing of a method directly onto the method as illustrated in figure 4.12.

This processing of the `postrendez/1` method causes the post-rendezvous processing required by each method to be typed by the TIP as part of the method which will minimize extraneous typings. The post-rendezvous processing of a method can then be translated as part of the equivalent task entry code in Ada.

When the argument of `waitToAccept/1` uses variables found elsewhere in the message, the recommended solution is to have the user explicitly type the messages containing the common variables. This situation occurs when the argument is partially instantiated (by the method name) as illustrated in the following example.

Example:

```
methodA :- waitToAccept(methodB(X,Y)), methodC(X).
methodC(X) :- methodD(X,Z), methodB(Z,W).
```

The variable `X` is shared by the `waitToAccept/1` and `methodC/1` messages. When the message `methodB/2` can be completely typed without `methodC/1` being typed, then the types derived for `methodB/2` can be used to type `methodC/1`. In the example, this situation does not necessarily occur. A complete typing of `methodB/2` includes

types derived from the typing of `methodC/1`. In this case user supplied type declarations are required.

Instance variables

Instance variables which are persistent data stores have no equivalent in Prolog. The MLog abstract interpretation engine must recognize the types for `theValueOf/2` and `makeTheValueOf/2` methods. This process involves: (1) selecting the instance variable from the correct class (this is equivalent to inheritance processing) and (2) selecting the correct instance variable from the instance variables available in the class. Unless the message is typed indirectly from other messages, the correct instance variable can be selected only if the first argument of the method is the symbolic name of the instance variable; i.e., not an instantiated variable. This places the restriction on the programmer that the messages `theValueOf/2` and `makeTheValueOf/2` contain the symbolic name of the instance variable. Otherwise, for classes with multiple instance variables, multiple threads of control are introduced. It is recommended that all instance variables be typed by the user prior to using the TIP. The abstract interpreter can use this information to statically type variables defined by instance variables. This is equivalent to treating those variables as constants.

Summary

The Prolog TIP requires significant modifications before it can be used to type MLog as follows.

- The Prolog engine must be replaced by an MLog engine.
- Code must be added to take advantage of the user supplied information on typed instance variables.
- The abstract interpretation of messages with explicit targets must:
 - a. recognize annotated class names;
 - b. explore two paths when necessary (due to `super`).

- The abstract interpretation of `waitToAccept/1` and `accept/1` messages must allow for two possible execution paths.
 - a. When the message argument is a variable ignore the message.
 - b. When the message argument is a symbolic method name, utilize the programmer supplied typings of the message argument to type shared variables in the method.

Despite these modifications the typing produced for an MLog program may not provide sufficient data to allow Ada translation.

The use of a TIP is not recommended at this time. Further work is required to resolve the following drawbacks.

- The Prolog TIP has a high time complexity. The TIP for MLog is expected to have a higher time complexity due to the addition of multiple threads of control (to support concurrency) and duplicate paths (to accommodate MLog's syntactic cohesiveness¹⁰).
- The duplicate paths can cause extraneous typings to be introduced for methods. These typings result in more complex Ada code as extra copies of generic subprograms and their associated task entries now are required.
- To support the TIP the user must provide type definitions and may provide method declarations. The TIP may still return a set of types that are unwieldy for Ada to support, and may have to be rerun after additional user information is provided.

The recommended solution is to have the user: (1) define types if required — a standard set of types would be pre-defined and (2) provide type declarations for each method. These declarations are used as the basis for determining the typing of messages within each method.

4.4.2 Preprocessing typed MLog

After the MLog code is typed, a list of possible types for each method is available. The typing¹¹ of a method consists of the method name and a type for each argument,

¹⁰ Syntactic cohesiveness refers to the ability of a MLog message to have multiple meanings; the correct meaning is selected based on the instantiation of the message target..

¹¹ The term typing is used to refer to the association of language expressions with types [37].

for example a type for the method `append(X, Y, Z)` is `append( $\alpha$ ,  $\alpha$ ,  $\alpha$ )`, where α is a distinct data type; i.e., a list of integers. However, Ada also needs to know which specific typing of the method is required on a given call.

example:

```
abc(X, Y) :- append(X, Y, Z) .
```

typings:

```
abc(int_list, int_list)
```

```
append(int_list, int_list, int_list)
```

```
append(char_list, char_list, char_list)
```

When `append/3` is called in the method `abc/2` the correct typing for `append/3` must be derived from the choices provided.

In Ada, different typings of a procedure do not execute the same code as occurs in MLog, rather each executes a specific instantiation of the procedure's generic code that corresponds to the typing. The translation of MLog methods into Ada procedures requires the identification of the typing of the message selector so that the correct instantiation of the generic procedure can be invoked. The combinations of monomorphic and polymorphic method heads and message selectors are shown for the sample code:

```
abc(X, Y) :- append(X, Y, Z) .
```

- `abc/2` is monomorphic and `append/3` is monomorphic.
- `abc/2` is polymorphic and `append/3` is monomorphic.
- `abc/2` is monomorphic and `append/3` is polymorphic.
- `abc/2` is polymorphic and `append/3` is polymorphic.

Only the last two cases require preprocessing in order to determine the correct typing of a polymorphic message in the clause body of a method for each possible typing of the method's clause head.

As a result of user declarations, each method's clause head has a set of typings. Where the method is defined in multiple classes, a set of typings exists for each class where the method is defined. The basic premise when translating a message within a method, is that given a typing for the method's clause head, sufficient type information

is available to uniquely identify which typing of any polymorphic messages in the clause body is to be invoked by the translated code. Where a message in the clause body selects a redefined method, the correct version of the method is identified as follows.

- When the message is to `self` the inheritance hierarchy determines the correct version of the method to use in typing the message selector.
- When the message has an explicit target, the classname (annotated on the target) is used to determine the correct version of the method to use for typing the message selector. If two typings must be pursued to accommodate an explicit target of `super` or an object-id (as discussed in the previous section), then the typings of the variables must be equivalent or the code must be restructured. The transformations only support code that can be typed statically at compile time.

Having identified: (1) which set of typings applies, (which version of a method is selected), for each message in a method and (2) a typing for the method's clause head, it is necessary next to determine a single type for each variable occurring in the method (clause head and body). Two algorithms are presented in figures 4.13 and 4.14, which provide a partial solution to the problem of statically typing MLog methods.

- Algorithm 2 (figure 4.13) determines a set of consistent types for variables in methods that consist of a single clause.
- Algorithm 3 (figure 4.14) determines a set of consistent types for variables in methods that consist of multiple clauses.

As a result of these algorithms each method should have a consistent set of type tuples corresponding to each typing of its clause head. A type tuple consists of a variable name and its associated type; e.g., (A, integer). A consistent set of type tuples includes one tuple for each variable appearing in the method.

Both algorithms make a basic assumption — that a consistent set of types (a set of type tuples) can be derived from the typings of the method's clause head and the typings of all the methods selected by its messages. This assumption is not always valid due to: (1) MLog's instance variables, and (2) MLog's ability to pass pointers to objects

(object-ids) as variables. In these cases the algorithms require user intervention to supply additional type data.

The basic assumption was made since Ada must know the typing of all procedures at compile time; i.e., Ada does not support polymorphism directly and if a unique typing of a message is not available for a given typing of the method head then the method cannot be translated efficiently and the MLog code must be restructured.

Why does the assumption not always hold? New variables in an MLog method can be introduced in the clause head or in any *out* argument position in the messages of the clause body. An anomaly occurs with instance variables since their value/type although introduced in an *out* position of the message `theValueOf/2` is totally independent of the remainder of the clause. It depends on the type of the instance variable which in turn depends on the class of its object which is run-time dependent. Since the same method (`theValueOf/2`) is used by multiple objects, the type of the output argument cannot be determined statically. This problem is not eliminated by restricting instance variables to being monomorphic, as this restriction does not eliminate the instance variable being redefined with a new type in a subclass. Thus, the problem occurs since the type of the instance variable (required at compile-time) depends on the value of the current object and its class (available at run-time). The previous discussion has assumed that the instance variable is identified by its symbolic name and not an instantiated variable. In the latter case the identity of the instance variable and hence its type, is also run-time dependent. The translation of methods containing `theValueOf/2` with the first argument a variable would require a case statement which chooses an alternative based on the identity of the instance variable, however the problem of identifying the correct type of the instance variable still exists. The determination of the type of the instance variable requires a run-time decision that decides the version of the message to implement based on the class of the current object. Since a case statement is used to allow the variable to assume different types in the different case alternatives, the remainder of the method must be translated in each case alternative (using declare blocks) since Ada does not

allow a variable to change type within a single scope. The code bulk this generates is not acceptable, and this option is disallowed.

The problem becomes more complex when an object-id is used to indirectly invoke a message whose typing is dependent on an instance variable. In the following example, `methodA/1` cannot be statically typed from an examination of the method since `X` represents an object-id, and the type of `Y` depends on the type of `D`; (i.e. the type of the instance variable belonging to the object `X`). Effectively the type of `Y` depends on the value of `X`. Even using annotated MLog code and abstract interpretation two possible types can be available since the target may be instantiated to `super` or an object-id.

Example:

```
methodA(X) :- methodB(X,Y) .
methodB(X,Y) :- X(someclass)-retrieve(D), methodC(D,Y) .
retrieve(D) :- theValueOf(some_ivar, D) .
typings: methodA(string)
         methodB(string,integer), methodB(string, character)
         retrieve(integer)
         methodC(integer, integer), methodC(integer, character)
modes: methodA(in), methodB(in,out), retrieve(out), methodC(in,out)
```

A fully automated system for typing all variables in an MLog method has not been determined, and is left as an open question. The following approaches may yield a solution.

- Use the algorithms presented in conjunction with an abstract interpreter to determine type information by using data flow when necessary.
- Develop a type inference system for MLog that records the complete typing of the method instead of just the typing of the method head.

Until a fully automated system is derived, algorithms two and three are used to type a method's clause bodies. These algorithms are augmented by user supplied data when required.

Algorithm 2

Algorithm 2 provides consistent types for the arguments of methods whose bodies contain polymorphic messages. The algorithm assumes the method consists of a single clause.

Algorithm 2 uses the type of the arguments appearing in the monomorphic messages in the body of the method and the type of the arguments appearing in the head of the method, to derive the type of the arguments of the polymorphic messages in the method body. An iterative approach is used; each time a variable is typed, all untyped messages are checked to see if the additional argument type allows a unique type for a message to be determined. The method outlined in Algorithm 2 is used to type all variables in the method `abc/1` in the following example.

Example: Apply Algorithm 2 to the method `abc/1`.

```
abc(X) :- msgA(X,Y,Z), msgB(X,Y,W), msgC(C).
Typings:
abc(int)
msgA(int,int_list,char), msgA(int,char_list,char), msgA(char,char_list,int)
msgB(int,int_list,int), msgB(int,token_list,int)
msgC(int)
```

step 1:

Collect all variables in the method that have a fixed type into set S.

Collect all polymorphic messages into a set B (messages that require typing).

Form the set, S, of known variable types from monomorphic messages in the method; form the set, B, of polymorphic messages.

from `msgC/1` $S = \{(C, \text{int})\}$

$B = \{\text{msgA}(X, Y, Z), \text{msgB}(X, Y, W)\}$

step 2:

Add to the set of variables with fixed types, one typing of the clause head. This forms an initial set of variables with known types.

Add a typing of the clause head to S.

$S = \{(C, \text{int}), (X, \text{int})\}$

step 3:

Check if variables with fixed types completely type any of the polymorphic messages, if so those messages are removed from the set of messages that need typing.

Not required since set S does not completely type
any element of set B.

step 4:

For each polymorphic message, try to determine a unique type for each of its variables that does not have a fixed type. For each variable determine its type by forming the set of all types the variable can assume in the message; (some typings of the message may be disqualified due to variables in the message that have been given a fixed type).

Form set M of possible type tuples for each
polymorphic message.
for msgA(X,Y,Z) get M = {A1,A2}
where A1 = {(X,int), (Y,int_list), (Z,char)}
A2 = {(X,int), (Y,char_list), (Z,char)}
element by element union of A1 and A2 =
{(X,int), {(Y,int_list), (Y,char_list)}, {(Z,char)}}
therefore Z has type char
S = {(C,int), (X,int), (Z,char)}

repeat step 3 - set S does not type any element
of set B

repeat step 4 for msgB(X,Y,W),
processing derives W has type int
S = {(C,int), (X,int), (Z,char), (W,int)}

repeat step 3 - set S does not type any element
of set B

step 5:

Determine a variable's type by examining the types it can assume in each of the messages where it appears within the method. The sets of types obtained for the variable from each message are intersected, if the resulting set has only one element, then the variable has been given a fixed type. Steps 3 and 4 are repeated if a variable is given a fixed type.

Form set, T, of untyped arguments appearing in method.
set T = {Y}
set of possible types for Y from msgA/3 is
{int_list, char_list}

```

set of possible types for Y from msgB/3 is
    {int_list, token_list}
intersection of sets = {int_list}
therefore Y has type int_list
S = {(C,int), (X,int), (Z,char), (W,int), (Y,int_list)}
Repeat step 3 - Remove msgA(X,Y,Z) and msgB(X,Y,W)
                from set B since they are typed
                by set S.
                B = {}

```

step 6:

If messages still exist without a fixed typing, user intervention is required, otherwise the type for each variable is saved.

```

    save set S, S is a consistent set of type tuples
    for the method abc/1.

```

step 7:

Repeat the algorithm for each typing of the clause head.

```

    not required since no other typing for abc/1

```

The algorithm assumes that a unique typing of the method head can produce a unique typing of a polymorphic message in its body, otherwise a type error exists.

example:

```

abc(X,Y) :- ..., append(A,B,C), ....

```

```

abc/2 and append/3 are both polymorphic

```

In the example, for a given typing of abc/2, there is only one typing for append/3; i.e., this typing will not change on different calls to abc/2 unless the typing of abc/2 also changes.

ALGORITHM 2.

```

-- From user declarations, each method has a set of typings.
-- If the set of typings consists of one element then the
-- method is monomorphic, otherwise it is polymorphic.

begin
  step 1.
  Form a set S of arguments and their types from any
  monomorphic messages in the body of the method.

  Form a set B of polymorphic messages in the method body;
  there is a 1-1 correspondence between multiple occurrences
  of a polymorphic message in the method and in the set B.

  step 2.
  From a typing of the method head (obtained from user
  declarations) add the variables and their types to the set S.

  step 3.
  Check if any message(s) in set B is completely typed by
  set S; (i.e., if all variables in a message are also in
  set B). If the typing is consistent with one declared by
  the user then remove the message from set B; otherwise
  the typing is not consistent, and user intervention
  is required.

  step 4.
  updateFlag:= true;
  while (set B not empty) and (updateFlag = true)
    updateFlag:= false;
    Match element X from set B, as typed from S against
    its set of typings from user declarations to get set
    of types M = {A1,A2...} for element X. Set M is a subset
    of the types of X provided by the user.

    if (M ={}) then
      inconsistent types-user intervention required;
      return;
    endif;

```

Figure 4.13 Algorithm 2: For consistent typing of polymorphic messages in the body of a single clause method. (Continued ...)

Determine if any arguments of X have been uniquely identified by taking the union of sets A1,A2,... with respect to each variable successively.

```

if for a given variable only one type results from
    the union then
    if the argument is not in S then
        Add the variable and its type to S;
        updateFlag:=true;
        repeat step 3;
    endif;
endif;
end while;

```

step 5.

```

if set B not empty then
    Form set T from arguments that remain untyped,
    (i.e., arguments not in S but appear in the method);
    if argument A, in set T, appears in more than one
        element of B then

```

step 5a.

Form a set of typings for argument A, from each element of set B in which it appears (from the TIP typings of the element of B).

```

    if the intersection of the sets derived in
        step 5a results in a single type then
        Add the argument and the type to set S;
        repeat from step 3;
    endif;
endif;
endif;

```

repeat step 5 until all elements of set T are processed;

step 6.

```

if set B not empty then
-- method exists without all arguments typed
   inconsistent typing-user intervention required;

```

Figure 4.13 Algorithm 2: For consistent typing of polymorphic messages in the body of a single clause method. (Continued ...)

```

    return;
else
    save set S containing a type for each variable appearing
    in the method.
endif;

step 7.
if not end of list of typings for the method head then
    repeat from step 2;
endif;
end;

```

Figure 4.13 Algorithm 2: For consistent typing of polymorphic messages in the body of a single clause method.

Algorithm 3

Algorithm 3 is used to completely type a method with multiple clauses. These methods are first normalized. A normalized method meets the following conditions.

- Each argument in the method's clause head(s) is represented as a single variable.
- Variables appearing in more than one clause body must also appear in the clause head.
- The variable representing the i th argument of the method's clause head must be identical in all clauses of the method.

Normalized methods are required to accommodate the different scope of variables in an MLog method and an Ada procedure. In MLog, the scope of a variable is a single clause in the method; in Ada the scope of a variable is the procedure which is equivalent to all clauses of the method.

Algorithm 3 processes an MLog method as follows.

- Normalize the method;
- Form a typing for each method by invoking algorithm 2 to type each clause.

For a polymorphic method, the translation of each occurrence of a polymorphic message within the method depends on whether the occurrence is used polymorphically or monomorphically. A polymorphic message is used monomorphically if the message has the same typing for all typings of the method head. Otherwise, the typing of the polymorphic message varies with the typing of the method head, and the message is used polymorphically.

In a monomorphic method, all polymorphic messages are treated monomorphically, by definition. In a polymorphic method, the preprocessing must determine for each polymorphic message whether it is used monomorphically or polymorphically, and annotate each message accordingly. Where a method contains multiple occurrences of a message selector, each occurrence is annotated since the case exists where distinct occurrences of a message are treated monomorphically and polymorphically. This case is illustrated below.

example:

```
abc(X) :- msgA(X,Y), msgB(X,Z).
abc(X) :- msgC(X,Z), msgB(Z,K).
```

Typings:

abc(int), abc(char)

msgA(int,int), msgA(char,int)

msgB(int,int), msgB(char,int)

msgC(int,int), msgC(char,int)

The first occurrence of msgB/2, msgB(X,Z), is used polymorphically as it has typings:

{(X,int),(Z,int)} for abc(int), and {(X,char),(Z,int)} for abc(char).

The second occurrence of msgB/2, msgB(Z,K) is used monomorphically as its typing is:

{(Z,int), (K,int)} for both typings of abc(X).

ALGORITHM 3.

```

begin

    step 1.
    Normalize the method.

    step 2.
    For a given typing of the method head apply Algorithm 2
    successively to each clause of the method. Algorithm 2
    returns a set of type tuples for each clause. The union
    of the sets of type tuples from all clauses forms a
    typing of the complete method.
    Step 2 is repeated for each typing of the method head.

```

Figure 4.14 Algorithm 3: For consistent typing of polymorphic/monomorphic methods with multiple clauses.

Thus at the completion of preprocessing MLog code the following information exists that uniquely identifies the required translation of each method:

- A set of consistent type tuples for variables in the method exists for each typing of the method head.
- Each polymorphic message in the method is annotated as being used monomorphically or polymorphically.

Polymorphic Instance Variables

To this point polymorphic methods have been addressed; MLog also supports polymorphic instance variables (ivars). Again Ada cannot support this polymorphism directly. In MLog an instance variable can be used to store variables of any type. This is not possible in Ada, where a given variable can only be used to store one type of data. The impact of this constraint is that MLog code that utilizes an instance variable polymorphically within an instance of the class must be restructured by the user prior to translation into Ada, to remove this polymorphic use of the instance variable. Since each instance of a class has a unique copy of the instance variables, Ada can support the translation

of a MLog class where separate instances of the class have different typings of their instance variables, but not efficiently. An example illustrates these points. ClassB contains methodA and the instance variable *buff* initialized to an empty list. Objects, instance1 and instance2 have been instantiated from classB. Within instance1, if methodA stores integers and characters in *buff*, then the MLog code must be reworked to remove this polymorphism. If methodA always places integers in *buff* for instance1 and always places characters in *buff* for instance2; then the code for classB can be translated into Ada.

When instance variables are restricted to being monomorphic, each class has a known set of typed ivars as a result of user declarations. The creation of instances in MLog, achieved by the statement `classname-new(name, active/passive, [])` is sufficient to allow the translator to create the instance in Ada code since `classname` effectively defines the typing for its set of instance variables. When the restriction is removed, the instance variables become polymorphic, and there is no longer sufficient information provided at compile time by `classname` to allow the translator to create Ada code to initialize the instance variables. The solution requires the use of run-time selection of the required type of the instance variable. This solution is unacceptable and all instance variables within a class must be monomorphic with respect to all instances of the class.

4.5 Summary

The result of pretranslation processing is MLog code that can be transformed into equivalent Ada code. The preprocessing can be divided into three categories: that provided explicitly by the user, that generated automatically by the pretranslation and that dependent on a user selected switch.

Preprocessing that must be done directly by the user is summarized below:

- make the MLog code deterministic;
- make the MLog code directional, ensuring all *in* arguments are fully instantiated, and all *out* arguments are uninstantiated;

- add `releaseInstance/1` messages to callers using anonymous sends;
- add `conditionsEnd/0` messages to acceptors' methods that act as guarded entries;
- restructure the code to remove recursive entries that contain a nested rendezvous, and to remove indirectly recursive entries that accept a nested rendezvous;
- annotate all messages to explicit targets with the appropriate classname depending on whether the target is instantiated to `super` or an `object-id`;
- remove all meta programming from the MLog code with the exception of that required for post-rendezvous processing;
- restructure methods as necessary to facilitate unfold transformations;
- provide type declarations and definitions as required.

The order presented above is the recommended order for preprocessing. Specifically, methods should not be unfolded, until all other restructuring of the code is done, since decisions on which methods require unfolding are difficult without the final restructured code to analyse. To minimize processing, programs should not be typed until after any required unfolding of methods is accomplished, since those methods that are modified by unfolding will have to have their type information modified.

Data that is generated automatically by the preprocessor includes the following:

- a list of classes that accept messages to anonymous targets;
- a list, for each class, of all methods that use the extended rendezvous;
- a list of methods that are redefined in subclasses and the classes that define/redefine them;
- a list, for each class, of all methods it needs to inherit and where each method is defined;
- type polymorphic messages in the body of methods using algorithms 2 and 3.

The user has one area of control over the transformations to be applied to the MLog code — the selection the transformation to be employed on the extended rendezvous. If transformation 1 is selected, the user must provide the maximum number of concurrent callers.

Chapter 5

Transformations revisited

Chapter 3 presented some basic transformations, and chapter 4 presented the restructuring and annotations required during pretranslation processing. This chapter addresses the transformation of MLog language characteristics that generally have been restructured and/or annotated during the pretranslation phase. The transformation of: (1) unification, (2) the interprocess communication primitives and (3) polymorphic methods are presented in the following sections.

5.1 Unification

Unification is used in MLog to pass values between calling and receiving objects. Unification subsumes: (1) data assignment, (2) parameter passing, (3) record allocation, and (4) read/write-once field access in records [43]. The effects of unification must be translated in Ada to the conventional memory management operations described above. In MLog, the unification process effects execution flow by its success or failure. Therefore the code resulting from translation of unification must interface to the execution control code of the translated program.

The unification algorithm attempts to match the arguments of the message selector with those in the clause head of the matching method. Each argument in MLog represents a *term*; a term is defined as: (1) a constant, (2) a variable, (3) a functor. A functor represents a compound term, and is characterized by its name and arity; arity indicates the number of arguments. For example, the term $f(t_1, t_2, \dots, t_n)$ is represented as f/n where f is the functor name, n is the arity, and $t_1, \dots, t_n$ are terms. Functors allow data of different types to be grouped into a single entity in typed-MLog, and are intuitively equivalent to data records in Ada. Lists are used to group data of different types in polymorphic MLog, but are limited in typed-MLog to values of a single type.

The conditions for two terms *X* and *Y* to unify follow.

- When *X* and *Y* are identical constants or variables they unify.
- When *X* is a variable that does not occur in *Y* they unify, and *X* is instantiated to *Y*.
- When *Y* is a variable that does not occur in *X* they unify, and *Y* is instantiated to *X*.
- When *X* and *Y* are functors and their functor name and arity are the same, and each term of *X* and *Y* unify then *X* and *Y* unify.

Otherwise, the unification of *X* and *Y* fails.

The translation of MLog unification into Ada data manipulation statements must make explicit the unifications which are done implicitly when a method is selected. A MLog method can be rewritten to make explicit the unifications that require translation. MLog code to be translated is fully ground for all *in* arguments and uninstantiated for all *out* arguments on initiation of a message¹². On completion of the message all *out* arguments are ground. The translation implements delayed unification; i.e., unification occurs with *in* arguments on method entry and with *out* arguments on method completion. Figure 5.1 illustrates: (1) the replacement of implicit unification with explicit unification, (this is equivalent to the normalization of a single clause method) and (2) the location of the explicit unifications within the method. This intermediate translation step occurs implicitly and is presented here to clarify the Ada code produced by the translation.

MLog code with implicit unifications:

```
methodA(a, [H|T], f(H, B, d), [c|K]) :- methodB(c, B, K).
modes: methodA(in,in,in,out)
       methodB(in,in,out)
typing: methodA(char,int_list,f_record,char_list)
       methodB(char,char,char_list)
```

Intermediate translation with explicit unifications:

```
methodA(X, Y, W, Z) :- X=a, Y=[H|T], W=f(H, B, d),
                      methodB(c, B, K), Z=[c|K].
```

Figure 5.1 Translating unification — an intermediate step.

¹² These restrictions on the *in* and *out* parameters apply to the message selector, but not to the selected method's clause head(s).

The translation of the unification of two arbitrary terms, *X* and *Y*, into equivalent Ada code follows. Since MLog code to be translated is typed and directional, unification checks for type and instantiation are not required.

- When *X* and *Y* are constants or instantiated variables, the translation is a comparison to determine if *X* equals *Y*. The case where *X* and *Y* are both uninstantiated variables does not occur since *in* arguments are always ground, and *out* arguments are not unified until completion of the method when they also are ground.
- When *X* is a variable, the translation assigns the value of *Y* to *X*.
- When *Y* is a variable, the translation assigns the value of *X* to *Y*.
- When *X* and *Y* are functors, their functor name and arity are always the same since these determine the type of the variable (and MLog has been typed for translation). Each term of *X* is unified with each term of *Y*. A special case exists when the functor is a list; i.e., “.”/2. Due to the built-in list type of MLog, extra information about the list’s length is passed during unification. For example, [*H*|*T*] has a length greater than zero, and [*A*, *B*, *C*] has a length of three. The translation of unification for list functors compares the length of the lists, if the list lengths are compatible, then the terms of the list functors are unified.

Since the success or failure of unification results in changes to program flow, the effects of unification are interfaced with the control of program execution by the *SUCCESS* argument which is set to *false* if unification fails. Since assignment statements cannot fail, the *SUCCESS* argument is set/reset only when a comparison is generated by the translation of unification.

Figure 5.2 illustrates the translation of unification for the method presented in figure 5.1.

5.2 Inter-process communications

Chapter 2 presented the criteria for determining when message passing is translated as a procedure call, and when it is translated as a rendezvous, and chapter 3 outlined the

Translation of methodA/4 given in figure 5.1:

```

procedure METHODDA(ENVIRONMENT : in ENV_RECORD;
                  X : in CHARACTER; Y : in INTEGER_LIST;
                  W : in F_RECORD; Z : out CHARACTER_LIST;
                  SUCCESS : in out BOOLEAN) is
  --declare local variables
  H : INTEGER;
  T : INTEGER_LIST;
  B : CHARACTER;
  K : CHARACTER_LIST;
begin
  SUCCESS := TRUE;
  --unify first argument
  if X = 'a' then
    null;
  else
    SUCCESS = FALSE;
  end if;
  --unify second argument
  if SUCCESS then
    if LENGTH_OF(Y) > 0 then
      H := HEAD_OF(Y);
      T := TAIL_OF(Y);
    else
      SUCCESS := FALSE;
    end if;
  end if;
  --unify third argument (TERM represents the name c' a field in the F_RECORD)
  if H = W.TERM1 then
    B := W.TERM2;
  else
    SUCCESS := FALSE;
  end if;
  if SUCCESS then
    if 'd' = W.TERM3 then
      null;
    else
      SUCCESS := FALSE;
    end if;
  end if;
  --process body of methodA/4 clause
  if SUCCESS then
    METHODDB(ENVIRONMENT, 'c', B, K, SUCCESS);
  end if;
  --unify fourth argument
  if SUCCESS then
    Z := K;
    CONSTRUCT('c', Z);
  end if;
end METHODDA;

```

Note: LENGTH_OF, HEAD_OF, TAIL_OF, and CONSTRUCT are list processing subprograms presented by Booch.

Figure 5.2 Example of unification transformation.

Method	Translation when argument is instantiated	Translation when argument is uninstantiated
<code>waitToAccept/1</code>	accept statement for the alternative specified by the instantiated argument	selective wait statement with accept alternatives for all entry methods in the accepting object
<code>accept/1</code>	a selective wait statement with an else part, the selective wait accepts the alternative specified by the instantiated argument. see note below	a selective wait statement with an else part, and accept alternatives for all entry methods in the accepting object see note below
Note: The else alternative sets the success argument to false to indicate the method failed.		

Table 5.1 Transformation of `waitToAccept/1` and `Accept/1` methods.

preprocessing of the code that is necessary for the translation of the MLog rendezvous, including messages to anonymous targets. The actual translation of the rendezvous is presented here. A rendezvous can occur in MLog after an active object sends itself a `waitToAccept/1` or `accept/1` message. The basic translation of these messages, which depends on whether the argument is instantiated, is presented in table 5.1. Since these messages are translated as in-line code and not as a procedure or a task entry, the transformations support only the `waitToAccept/1` or `accept/1` messages when the target object is `self`; (i.e., the message, `X-waitToAccept (Y)`, can only be translated when `X` is instantiated to `self`).

The types of rendezvous generated by these messages are: (1) a unguarded rendezvous, (2) a guarded rendezvous, and (3) an extended rendezvous. The translation of these rendezvous types and of anonymous messages is discussed in the following sections.

5.2.1 Unguarded Rendezvous

An unguarded rendezvous does not depend on caller or acceptor data. The translator determines that a rendezvous is unguarded when the following conditions are met.

- The method selected in the accepting object is not annotated with a `conditionsEnd/0` message.
- The clause heads of the method selected do not contain constants or the explicit list functor (i.e. [...]). Other functors may appear in the clause head since they impose no implicit conditions on the caller's data.
- In multiple clause methods, if one clause meets the previous conditions then the method is unguarded. (Note: Within a method, if some clauses are guarded and other clauses are unguarded, an error has probably occurred in annotating the code with `conditionsEnd/0` messages, since the unguarded clause will always accept the rendezvous).

The translation of an unguarded rendezvous is a direct application of the transformations in table 5.1 and is illustrated in figure 5.3.

5.2.2 Guarded rendezvous

A guarded rendezvous depends on acceptor data. The translator determines that a rendezvous is guarded when all the following conditions are met.

- The method selected in the acceptor object is annotated with a `conditionsEnd/0` message.
- The messages preceding the `conditionsEnd/0` message do not contain any variables common to the *in* arguments of the clause head. These messages form the guard conditions.
- The clause head of the method does not contain any constants or explicit list functors in the *in* arguments as these are implicit conditions on the caller data.

In a guarded rendezvous, the translated code of the messages that form the guard conditions is placed before the selective wait statement required in Ada for a guarded accept statement. The success or failure of these messages' equivalent Ada procedures or rendezvous, determine whether the guard is open or closed. The messages following the `conditionsEnd/0` message are translated within the accept statement. An example of the guarded rendezvous is found in figure 5.4.

in MLog:

```
class: classA of object.
begin :- waitToAccept(methodB),
        methodC.
methodB/0 is an entry into objects of classA.
methodC/0 is not an entry.
```

in Ada:

```
task type ACTIVE_CLASSA is
    entry METHODB(ENVIRONMENT : in ENV_RECORD_T;
                SUCCESS : in out BOOLEAN);
    ...
end ACTIVE_CLASSA;

task body ACTIVE_CLASSA is
    ...
begin
    accept METHODB(ENVIRONMENT : in ENV_RECORD_T;
                SUCCESS : in out BOOLEAN) do
-       process methodB
    end;
    if SUCCESS then
        METHODDC(ENVIRONMENT, SUCCESS);
    end if;
end ACTIVE_CLASSA;
```

Figure 5.3 Example of the translation of a unguarded rendezvous.

Two potential problems exist with the translation of a guarded rendezvous.

- Common variable names in messages forming the guard conditions of separate entries can cause erroneous values as variables are unintentionally reset. This situation is avoided by renaming variables prior to translation.
- Methods with multiple clauses are only guarded when each clause is guarded since the guard on the accept alternative becomes the disjunction of the conditions on each clause. When the conditions on each clause are identical, this degenerates into the single method case illustrated in figure 5.4. When the conditions on separate clauses are different, the translation does not provide a way of determining which part of the accept alternative should be entered (which clause's guard messages

in MLog:
 methodA :- methodB,methodC,conditionsEnd,methodE.
 Assume methodA is an entry in a class that uses the default begin/0 method.

in Ada: (only the task body is shown)

```

task body ACTIVE_CLASS is
  ...
begin
  loop
    -- evaluate guard conditions
    SUCCESS := TRUE;
    METHODDB(ENVIRONMENT, SUCCESS);
    if SUCCESS then
      METHODDC(ENVIRONMENT, SUCCESS);
    end if;
    if SUCCESS then
      METHODDA_GUARD := TRUE;
    else
      METHODDA_GUARD := FALSE;
    end if;
    -- accept rendezvous

    select
      when METHODDA_GUARD =>
        accept METHODDA(ENVIRONMENT : in ENV_RECORD_T,
                      SUCCESS : in out BOOLEAN) do
          SUCCESS := TRUE;
          METHODE(ENVIRONMENT, SUCCESS);
        end;
      or
        ...
    end select;
  end loop;
end ACTIVE_CLASS

```

Figure 5.4 Example of the translation of a guarded rendezvous in a single clause method

evaluated to *true*). In these cases the translator will repeat the Ada code for the guard messages within the accept statement. This causes the messages to be evaluated twice, however it does not cause unexpected results since guard conditions should not cause

permanent side effects. In MLog if the conditions are not met, the guard messages are also executed more than once, and though MLog will unbind variables on backtracking, permanent side effects, which includes the setting of instance variables, cannot be unbound. The MLog and Ada code are functionally equivalent, but the Ada code is less efficient. The code associated with guard conditions can be expected to be brief in a well designed MLog program since by definition the conditions are expected to be executed more than once.

5.2.3 Extended rendezvous

An extended rendezvous depends on caller and acceptor data. The translator determines that an extended rendezvous exists when the conditions for a unguarded or guarded rendezvous are not met. As in the guarded rendezvous, messages preceding the `conditionsEnd/0` message form the guard conditions for the rendezvous. Chapter 4 outlined two possible transformations of the extended rendezvous in figures 4.3 and 4.4.

The criteria for deciding when the extended rendezvous conditions are met for a caller on the pending caller's queue depends on whether the `waitToAccept/1` or `Accept/1` message that initiates the rendezvous has an instantiated or uninstantiated argument.

- *Uninstantiated argument.* The rendezvous is translated by a selective wait with accept alternatives for all entries into the task. The conditions that must be met by the objects on the pending caller queue are those specified by the guard messages of the invoked method.
- *Instantiated argument.* The rendezvous is translated by an accept statement which accepts the entry corresponding to the instantiated argument. The conditions that must be met by the objects on the pending callers queue are those outlined above, plus the caller must be requesting service of the type specified by the instantiated argument; i.e., the eligibility step of the MLog rendezvous must be satisfied.

This difference is transparent in MLog as unification makes the entry name part of the rendezvous selection criteria. However in Ada, the different criteria must be explicitly used in searching the pending callers queue. Since the `waitToAccept/1`

in MLog:

```
class: classA of object.
begin :- waitToAccept(methodA).
methodA :- methodB,conditionsEnd,methodC.
methodA :- methodD,conditionsEnd,methodE.
```

in Ada: (only the task body is shown)

```
task body ACTIVE_CLASSA is
    . . .
    SUCCESS := TRUE;
    METHODDB(ENVIRONMENT, SUCCESS);
    if SUCCESS then
        null;
    else
        SUCCESS:= TRUE;
        METHODDD(ENVIRONMENT, SUCCESS);
    end if;
    if SUCCESS then
        METHODDA_GUARD := TRUE;
    else
        METHODDA_GUARD := FALSE;
    end if;
    select
        when METHODDA_GUARD =>
            accept METHODDA(ENVIRONMENT : in ENV_RECORD_T,
                           SUCCESS : in out BOOLEAN) do
                SUCCESS := TRUE;
                METHODDB(ENVIRONMENT, SUCCESS);
                if SUCCESS then
                    METHODDC(ENVIRONMENT, SUCCESS);
                end if;
                if SUCCESS then
                    null;
                else
                    SUCCESS := TRUE;
                    METHODDD(ENVIRONMENT, SUCCESS);
                    if SUCCESS then
                        METHODE(ENVIRONMENT, SUCCESS);
                    end if;
                end if;
            end METHODDA;
        end select;
end ACTIVE_CLASS;
```

Figure 5.5 Example of the translation of a guarded multiple clause method.

or `accept / 1` can specify only a willingness to rendezvous on one entry (a instantiated argument) or on all entries (an uninstantiated argument), discussion of the criteria to use when searching the pending callers queue for other combinations of accept alternatives in a selective wait is not required.

In order to maintain the FIFO preprocessing of callers (within the constraints posed by guard conditions), the pending callers queue must be scanned prior to accepting any extended rendezvous from an external caller. If an accept alternative representing a guarded or unguarded rendezvous appears in the Ada code, the pending callers queue need not be checked since calls to those entries do not place callers on the queue. If the selective wait contains accept alternatives for all types of rendezvous, the pending callers queue must be scanned.

Both transformations of the extended rendezvous require that the caller be removed from the pending caller queue prior to providing the requested service. This is critical when the requested service includes accepting a nested extended rendezvous. In these cases, if the caller's data remains on the queue, the nested extended rendezvous could cause the callers data to be processed a second time; (i.e. it would be reselected when the pending caller's queue was scanned to find a caller to respond to the nested accept of an extended rendezvous). Similarly, if the guard conditions of an extended rendezvous include a `waitToAccept / 1` or `accept / 1`, that results in an extended rendezvous, the caller's data must be removed from the queue prior to evaluating the guard conditions.

A modification to transformation 1 is made to decrease code bulk and increase runtime efficiency. An accept alternative within a selective wait statement cannot occur within a loop construct, thus each indexed wait entry must have an individual accept alternative. The code bulk this generates can be eliminated by accepting the appropriate wait entry immediately after the caller's service is completed instead of opening the entry by setting the guard. This change results in a potentially much smaller selective wait statement which increases run-time efficiency. The modified transformation 1 is presented in figure 5.6.

Resulting Algorithm, Caller-side

– no change

Resulting Algorithm, Acceptor-side

```

while ( not done ) begin
  scan pending caller queue (front to back) for
  some caller with extended rendezvous conditions
  that are now true
  if conditions begin
    remove caller info from queue
    do service for caller
    accept wait[index](x) do
      end accept
  end else begin
    select begin
      accept entry_name(x,next_entry) do
        if conditions begin
          do service for entry_name
        end else begin
          assign some entry index to next_entry
          append x on pending caller queue
          using assigned next_entry as caller id
        end
      end accept
    or
      ...
    end select
  end
end while

```

Figure 5.6 Pscudo-code for Transformation 1 modified to individually accept waiting callers.

Using transformation 1, extended rendezvous are translated by a straight forward application of the algorithm in figure 4.3 when the rendezvous is initiated by a wait-ToAccept/1 message. The pool of wait entries is used by callers to different entries and therefore with different argument lists. Ada variant records are used to describe the argument lists and records on the pending callers queue. The name of the entry originally accessed provides the discriminant for these records.

When the extended rendezvous is initiated by an accept/1 message transformation

1 must be modified slightly. In Ada the selective wait with an else part accepts a caller if possible, or it executes the else alternative. If a caller is accepted its requested service is performed. This is not the case with an extended rendezvous — multiple callers may be accepted and put on the pending callers queue before a caller's request for service is processed (rendezvous completes successfully), or before the else alternative is executed. An extra flag `RENDEZVOUS_DONE`, has been added to transformation 1 which allows the selective accept loop to be exited when: (1) the else alternative is processed — no rendezvous could complete successfully, or (2) one of the caller's has had a successful rendezvous. The `SUCCESS` flag continues to be used to indicate whether the `accept / 1` message terminated in success or failure. The changes are shown in bold type in figure 5.7. Similar changes are required for transformation 2.

A second option with transformation 1 is to provide individual entries (as opposed to a pool of entries) where each caller waits on its private entry. This solution requires more data to be collected from the user, specifically the name of every object that uses an extended rendezvous when calling the acceptor object. The translator would have to devise individual entry names for callers. Individual entry names are not feasible since the same code is used by multiple caller objects in a class. The caller's code would require a case statement that would select the appropriate wait entry depending on the caller object's identity. Thus, specific wait entries achieve greater clarity in the Ada code at the expense of runtime efficiency. Furthermore, greater user input is required (the names of all callers instead of the total number of concurrent callers for an accepting object) and more translator code is required to process the user input. The option of individual wait entries is disallowed on the basis of its inefficiency.

Under transformation 2, extended rendezvous are translated by a straightforward application of the algorithm in figure 4.4 when the object originating the call is active; i.e., the call is originated in a task's mainline code. The arguments of wait entries in the caller are again implemented by variant records. This allows one wait entry to be defined on the caller object for all entries called by the caller object. The record discriminant

Resulting Algorithm, Caller-side

- no change

Resulting Algorithm, Acceptor-side

```

while ( not rendezvous_done ) begin
  scan pending caller queue (front to back) for
  some caller with extended rendezvous conditions
  that are now true
  if conditions begin
    remove caller info from queue
    do service for caller
    accept wait[index](x) do
    end accept
    set rendezvous_done true
  end else begin
    select begin
      accept entry_name(x,next_entry) do
        if conditions begin
          do service for entry_name
          set rendezvous_done true
        end else begin
          assign some entry index to next_entry
          append x on pending caller queue
          using assigned next_entry as caller id
        end
      end accept
    or
      ...
    else
      set success false
      set rendezvous_done true
    end select
  end
end while

```

Note: The above code translates the accept/1 message, if the message occurs within a loop then the loop requires additional translation.

Figure 5.7 Transformation 1 modified for accept/1 messages.

is composed of the names of all entries called by the object that require an extended rendezvous.

Transformation 2 as presented expects both the caller and acceptor to be tasks. A modification to the transformation is required when the caller is a procedure; i.e., if the caller object is passive. In MLog, a passive object calling an active object can result in an extended rendezvous but under transformation 2, the passive object would try to accept a rendezvous, which is an error in Ada. The error situation is avoided by modifying the caller side. The passive caller object, prior to initiating an extended rendezvous, creates a temporary task which initiates the extended rendezvous on the caller object's behalf. On completion of the rendezvous, the temporary task terminates. This extension to the transformation is shown in figure 5.8.

The change to the caller-side, presented in figure 5.8, is transparent to the acceptor-side of the algorithm since the caller is identified, in both variations of the algorithm, as a pointer to a task type.

Transformation 1 presented in figure 5.6 is the preferred translation of the extended rendezvous. The translation is constant regardless of the mode of the caller object, and can have greater run-time efficiency. When a passive object originates the call for a rendezvous, transformation 2 incurs the run-time overhead of creating a task and making an extra rendezvous to invoke the created task.

An optimization of transformations 1 and 2 is achieved by evaluating the pending caller queue only after a rendezvous terminates satisfactorily, rather than every time the task is made active. If it is made active by a rendezvous whose conditions are not met and in which the caller gets placed on the pending caller queue then there has been no change to the acceptor's state, and therefore data that the pending callers require to complete the rendezvous has not changed; i.e., neither the state of the acceptor nor the caller data has changed. This assumes that guard conditions do not cause permanent side effects [cref §5.2.2].

Resulting Algorithm, Caller-side

```
TEMP_CALLER := new TEMP_TASK;
TEMP_CALLER.ENTRY_NAME(X, TEMP_CALLER);
...
```

Definition of temporary task type.

task type TEMP_TASK is

```
    entry ENTRY_NAME(X : in out ARGS_RECORD;
                     CALLER_ID : in out TASK_PTR);
- any other entry names that require an extended rendezvous and that are
- called by this class of objects.
```

```
    ...
    entry WAIT_ENTRY(X : in out ARGS_RECORD);
end TEMP_TASK;
task body TEMP_TASK is
    ...
    select
        accept ENTRY_NAME(X : in out ARGS_RECORD;
                           CALLER_ID : in out TASK_PTR) do
            call ACCEPTOR.ENTRY_NAME(X, CALLER_ID, STATUS);
            if STATUS = rejected then
                accept WAIT_ENTRY(X : in out ARGS_RECORD) do
                    end accept;
            end if;
        end accept;
    or
        ...
    end select;
end TEMP_TASK;
```

Resulting Algorithm, Acceptor-side

- No change.

Figure 5.8 Transformation 2, modified to support a passive caller object.

5.2.4 Anonymous messages

The anonymous message is translated as a dispatcher paradigm in Ada [cref. §4.2.1]. The following steps indicate the restructuring of the MLog code required to replace an anonymous message with a dispatcher in MLog.

- Create a dispatcher class with two methods, `getInstance/1`, `releaseInstance/1`. The argument of these methods is an object pointer.
- In the `begin/0` method of the class `main` of object create a dispatcher object for each class that accepts anonymous messages.
- In the `begin/0` method in each class that accepts anonymous messages insert a message, `dispatcherForClass-releaseInstance/1`, which tells the dispatcher object that a new active object is available for assignment to callers.
- Replace every anonymous message with a message to the dispatcher object of the accepting class.

example:

using anonymous message:

```
methodA :- classA(X)-methodB.
```

using the dispatcher paradigm:

```
methodA :- dispatcherForClassA-getInstance(X),
           X(classA)-methodB.
```

- Insert the message `releaseInstance/1` into the callers method to tell the dispatcher when the instance is being returned. The user has done this during pretranslation processing of the code [cref §4.2.1].

The translation from MLog to MLog does not actually occur, but the translation of an anonymous message into Ada is effectively a translation of the transformed MLog code outlined above. The anonymous message results in the following transformation.

- A dispatcher task type is defined. Since all dispatchers control object pointers, (a single type), a generic package containing the task is not required.
- The mainline program creates an instance of the dispatcher task type for each class in MLog that accepts anonymous messages.

- The mainline of each task, that results from the translation of a MLog class which accepts anonymous messages, calls the `release` entry of the class' dispatcher task to inform the dispatcher of its availability.
- When the anonymous message syntax appears in MLog it is translated with a call to the acceptor class' dispatcher task to get an acceptor instance; the instance is called to request work, and when all work is completed, the dispatcher task is called to return the acceptor instance.

In figure 5.9, the calls to the dispatcher instance do not track the environment since these calls emulate system calls in MLog and are transparent to the MLog program. The call to the `methodB` entry of task X, will always succeed since it is a rendezvous, however for simplicity the argument is carried and checked for both procedure and rendezvous calls.

5.3 Polymorphic MLog

The basic premise of the translation of a method is that messages in the clause body can uniquely determine the method to be selected. Furthermore, when the selected method is polymorphic, the message selector provides sufficient information to determine which instantiation of the method to select. A polymorphic method is one whose arguments have more than one set of types. Polymorphic methods are translated as generic subprograms, and instances are declared for each different typing of the method. Table 5.2 presents a more detailed translation of polymorphic methods which analyses the effects of polymorphism in the method's clause head and/or clause body.

Table 5.2 outlines how a method is selected for translation as a subprogram or as a generic subprogram. The underlying assumption in all cases is that the message passing which invokes the corresponding method always generates a procedure call. The possibility of the method being invoked as a result of a rendezvous must also be considered.

Since task entries cannot be generic in Ada, a polymorphic method that is translated as a task entry results in each typing of the method being coded as a separate entry. The

MLOG ANONYMOUS MESSAGE:

```
methodA :- classA(X)-methodB,...,
          dispatcherForClassA-releaseInstance(X),... .
```

STEPS IN ADA TRANSLATION:

Define dispatcher task type:

```
task type DISPATCHER is
  entry RELEASE_INSTANCE(X : in OBJ_NAME);
  entry GET_INSTANCE(X : out OBJ_NAME);
end dispatcher;
type DISPATCHER_PTR is access DISPATCHER;
```

In mainline code of program:

```
DISPATCHER_FOR_CLASSA : DISPATCHER_PTR;
...
DISPATCHER_FOR_CLASSA := new DISPATCHER;
```

In mainline code of task representing objects of classA:

```
task body CLASSA_ACTIVE is
  ...
begin
  accept INIT(X : in OBJ_NAME) do
    ...
  end accept;
  DISPATCHER_FOR_CLASSA.RELEASE_INSTANCE(X);
  ...
end CLASSA_ACTIVE;
```

In the code of some class that initiates an anonymous message to classA:

```
procedure METHODDA(ENVIRONMENT : in ENV_RECORD;
                  SUCCESS : in out BOOLEAN) is
  ...
  DISPATCHER_FOR_CLASSA.GET_INSTANCE(X : out OBJ_NAME);
  ENVIRONMENT := (CUR_OBJECT => X, CUR_CLASS => CLASSA,
                  CUR_PROCESS => X);
  CLASSA.METHODB(ENVIRONMENT, SUCCESS);
  if SUCCESS then
    -   do any other work required
  end if;
  DISPATCHER_FOR_CLASSA.RELEASE_INSTANCE(X);
  ...
end METHODDA;
```

Figure 5.9 Translation of the anonymous message.

	Method's clause head	Messages in clause body	Comments	Translation
1	monomorphic	all monomorphic		The method is translated into a subprogram.
2	polymorphic	all monomorphic	The method is polymorphic but all methods selected by its messages are monomorphic. This case is impossible to achieve in typed MLog.	nil
3	monomorphic	some/all polymorphic	The method is monomorphic but some/all of the methods selected by its messages are polymorphic.	The method is translated as a subprogram. The subprogram name(s) generated by the correct instantiation of the polymorphic method(s) is determined and this new subprogram name appears in place of the polymorphic message name in the body of the subprogram. note1.
4	polymorphic	some/all polymorphic	All polymorphic messages are used monomorphically; (i.e., regardless of the typing of the clause head a polymorphic message has the same typing). This is the same as case 2.	nil
5	polymorphic	some/all polymorphic	Some polymorphic messages are used monomorphically; some are used polymorphically; (i.e., a message is used polymorphically when for different typings of the clause head the typing of the polymorphic message changes).	The method is translated as a generic subprogram with formal type parameters for those variables that are polymorphic, and a formal subprogram parameter for each polymorphic message that is used polymorphically. Polymorphic messages that are used monomorphically have the correct instantiation of the selected method inserted in the body of the generic subprogram as per item 3. note2.
notes: (1) The subprogram name and the polymorphic message name may be the same if Ada's support for overloading subprogram names is used. (2) When duplicate occurrences of a polymorphic message within a method are distinctly typed, and the occurrences are used polymorphically, the message name appears twice as a formal generic parameter. This approach utilizes Ada's overloading support.				

Table 5.2 Translations of monomorphic/polymorphic methods.

caller object can then access the appropriately typed entry. The number of separate entries for the polymorphic method can be limited to those actually required for that class of objects. After the MLog code is typed, the translator can determine all methods that call a class and the typing of those methods. Some of the typings of a polymorphic method result from messages to `self` and therefore only need be translated into procedures.

Chapter 6

Applying the Transformations

In the previous chapters the transformation of individual language elements has been discussed. A technique is presented now for using these transformations to produce functionally equivalent MLog and Ada programs. The technique provides a logical progression for: (1) applying the previously discussed transformations to a given MLog program and (2) interfacing these transformations to code that emulates MLog system support. The producer/consumer problem is used to illustrate both the pretranslation processing of the MLog code and the application of the transformations.

6.1 Emulating MLog system support

In addition to transforming each MLog class into an equivalent Ada package, the translator must produce housekeeping code that provides the same services to the translated program as the MLog system provides to the MLog program. The function of this housekeeping code is summarized below.

Build a mainline program for the application.

The mainline program creates an active object (instantiates a task type) in package `main`, and then transfers control to that object. This parallels the operation of the MLog system when it starts an application program using the *run* command. If the application program uses anonymous messages the mainline program also defines a dispatcher task type, and instantiates dispatchers for those classes requiring them.

Build and maintain global data structures that provide information on the status of objects in the program.

The status of objects in the program is maintained automatically by the MLog system but must be explicitly coded in an Ada translation of the program. The package `GLOBAL` which is made visible to all other packages in the system encapsulates the global data structures, that provide information on the status of objects, and

the procedures and task entries used to access these data structures. The package GLOBAL fulfills two functions: (1) it controls the assignment of object-ids to newly created objects, and (2) it maintains a list of records, each of which contains the object-id, mode, class, and symbolic name for an object. The symbolic name is of type *string*, and can be assigned by the user. When the user does not provide a name, the symbolic name is derived implicitly from the object-id. Since strings are inefficient when used as keys into data structures, an object-id based on the integer type is also assigned to each object.

The structure of the GLOBAL package is given in figure 6.1. The data-store NEXT_NAME is provided with mutual exclusion protection by the task OBJ_NAME_CONTROL to ensure that two objects never receive the same name. The function of the list of instance records (INSTANCE_LIST) is to provide the mode of an object given its object-id, and to provide the object-id of an object given its symbolic name. These queries are made in order to identify the mode of an explicit target, therefore, any attempt to protect this data structure (from concurrent access by readers) with a task results in an extra rendezvous occurring in the program for each message sent to an explicit target (excluding super). This unacceptable overhead is avoided by implementing the data store as a list with the following access constraints.

- New records are added only to the head of the list;
- Records on the list are read only.

Since connecting a record to the head of a list involves updating one memory location, and assuming this is an atomic operation, it is impossible for a process that requires a read operation on the list to obtain a partially updated list. Two cases exist: (1) the calling object receives the list with the old head, and (2) the calling object receives the list with the new head. In either case, the list is searched for the required data. If the data is not found, the situation is equivalent to calling an object that has not been created; in MLog and Ada this results in an error.

Figure 6.1 Package Global

This implementation imposes the restriction that objects cannot be deleted from the system. This constraint is acceptable since the number of objects in a program is restricted by the nature of MLog objects. Unlike Smalltalk, in MLog, most data structures are not objects. Objects are instances of a class and normally represent a useful model of some part of the problem domain. Furthermore, at present the deletion of objects in MLog is not fully implemented.

To prevent two objects accessing the list concurrently for the purpose of adding a new head element, write access to this list is protected by mutual exclusion. The task `OBJ_NAME_CONTROL` prevents concurrent access to the list for writing.

6.2 Translation of a user-defined class

Each user defined class is transformed into a package of the same name. In addition to transformed copies of the user defined methods, each package is provided with an individual copy of the MLog system method `new/3` (renamed to `MAKE_NEW` since *new* is a reserved word in Ada). In addition, classes that define or inherit instance variables receive a data-store to contain pointers to the instance variable records, and an individual copy of `theValueOf/2` and `makeTheValueOf/2` for each instance variable. Unless a class is known (from static inspection) to have only passive objects, classes also are provided with (1) a task type to define active objects, (2) a data-store to contain pointers

to active objects, and (3) a procedure to access the data-store. The lists of pointers to active objects and pointers to instance variable records are implemented equivalently to the global `INSTANCE_LIST`. Thus each list allows concurrent readers and one writer, but prevents concurrent access by multiple writers. The structural composition of a user defined class with instance variables and active objects is given in figure 6.2, and its contents are summarized below.

- *Data structures and procedures for classes with instance variables.*

`IVAR_PTR_REC` — This record associates an object's name with a pointer to the record containing its instance variables, including inherited instance variables. These records are saved on the list `IVAR_PTR_REC_LIST` which corresponds to the data store in figure 6.2.

Three MLog methods can access the instance variables of an object: (1) `new/3` which performs create time instantiation of values, (2) `theValueOf/2` which reads the value of instance variables and, (3) `makeTheValueOf/2` which updates the value of instance variables. An individual copy of the translated `new/3` method is required for each class. Each of the methods `theValueOf/2` and `makeTheValueOf/2` is translated as a set of procedures in the class. Each instance variable has a private copy of these methods which is renamed to include the instance variable name. For example, `theValueOf(bound, X)` becomes procedure `TheValueOf_BOUND(...)`. This approach avoids the *case* statements that would otherwise be required in the Ada code when instance variables have the same type; (i.e., Ada's overloading could not discriminate).

- *Data structures and procedures for classes with active objects.*

- `ACTIVE_CLASSNAME` — This task type defines active objects of the class.
- `TASK_ID` — This access type defines a pointer to an active object.
- `TASK_NAME_T` — This record type associates the object-id of an active object with its access pointer. These records are stored on the list `ACTIVE_LIST` which corresponds to the data-store shown in figure 6.2.

Figure 6.2 The package representing some user defined class

- d. **ACTIVE_MUTEX** — This task provides mutual exclusion for processes updating **ACTIVE_LIST**.

The procedure **GET_TASK_PTR**, produced by the translator, is used to access the records on the **ACTIVE_LIST**. Given an object-id, the procedure returns the pointer to the task representing the active object.

Additional data types must be defined if the objects in the class accept extended rendezvous. A pending caller queue must be defined to hold callers' arguments. A variant record is defined to hold the arguments of a caller. This record type is capable of storing the arguments of any entry that accepts the extended rendezvous.

The translation of the system methods **new/3**, **theValueOf/2** and **makeTheValueOf/2** is presented below, to clarify their operational semantics.

new/3 —

The **MAKE_NEW** procedure does the following.

- Calls the task **OBJ_NAME_CONTROL** which provides the following functions.

- a. Gets a new object-id from the GLOBAL database to reference the object.
- b. Determines the symbolic name for the object as follows.
 - If the mode of the name argument in `new/3` is *in*, the value of the user name is the string provided by the user.
 - If the mode of the name argument is *out*, the string representation of the object-id becomes the symbolic name, (the value is always of type string to accommodate Ada's strict typing rules).
- c. Update the global database with the object-id, its mode, its class and its symbolic name.
- If the class defines or inherits an instance variable, performs the following functions.
 - a. Creates an `IVAR_RECORD` and initializes it with all compile time values of the object's instance variables.
 - b. Modifies the `IVAR_RECORD` to reflect any create-time initializations of instance variables of the object.
 - c. Saves the pointer to the `IVAR_RECORD` in the `IVAR_PTR_REC_LIST` for the class.
- If the mode of the new object is active, creates a new task, adds the new active object to the list of active objects (`ACTIVE_LIST`) for that class, and sends the task an `INIT` message to inform the object of its id.

`theValueOf/2, makeTheValueOf/2-`

Each procedure generated by the translation of the methods performs as follows.

- Gets a copy of the object's `IVAR_RECORD` through the `IVAR_PTR_REC_LIST` of the package.
- Returns or updates the value of the requested instance variable.

6.2.1 Transforming a class into a package

The major functions required to translate a user-defined class into a package are summarized as:

- defining¹³ types for the data structures required to store instance variables and pointers to active objects;
- defining a task type to represent active objects;
- translating and inserting the methods `new/3`, `theValueOf/2`, `makeTheValueOf/2` as required;
- inserting the procedure `GET_TASK_PTR` and the mutex task if required;
- translating any user-defined methods;
- inserting call-through procedures required to accept calls to inherited methods.

An expanded presentation of the translation of MLog classes into Ada packages is found in appendix A which provides the following pseudo-code algorithms.

- Figure A.1 translates a class into a package.
- Figure A.2 translates a method into a procedure.
- Figure A.3 translates a message within a procedure.
- Figure A.4 translates a class into a task type.
- Figure A.5 translates a method into a task entry.
- Figure A.6 translates a message within a task entry.

6.2.2 Transforming a class into a task type

The task type defined as part of the translation of a class is built as follows.

- The task specification which appears in the package specification contains an `INIT` entry and an entry corresponding to each method that acts as entry into the objects of the class.

¹³ The term *define* is used to indicate the definition of a data type.

- The task body which appears in the package body is formed as follows.
 - a. Declare¹⁴ local variables required to support unification, message arguments, and guards for entries.
 - b. Add the predefined code for the INIT entry.
 - c. If the class accepts anonymous messages, insert the code to inform the dispatcher of the objects existence, [cref. § 5.2.4].
 - d. Translate the begin/0 method for the class. In order to translate the begin/0 method of a class whose active objects accept messages (from other objects), it is necessary to translate the entry methods of the class into task entries.

6.3 An example: The producer consumer problem

6.3.1 Pretranslation processing

The producer consumer problem introduced in chapter two is used now to illustrate the pretranslation and translation processing of the MLog code. The pretranslation follows the steps proposed in section 4.5. The annotated code resulting from preprocessing is found in figures 6.3 and 6.4. The user's input into the pretranslation processing of the problem follows.

1. *Make the code deterministic.*
No changes are required as the code is deterministic.
2. *Make the code directional, ensuring all in arguments are instantiated and all out arguments are uninstantiated.*

Mode declarations are added to the code for each user defined method, and for each occurrence of multi-directional system methods. The multi-directional system methods are `listLength/2`, `theValueOf/2`, and `new/3`. In the method `retrieve/1` the second argument of the message, `theValueOf(myBuffer, [DATA|B])` (figure 2.4), is partially instantiated and is considered to have mode *out*. Since the argument is not fully ground, the code must be

¹⁴ The term `declare` is used to indicate the elaboration of a data object in Ada.

restructured; this is accomplished by rewriting the message to replace the implicit unification with an explicit unification statement. The resulting code is:

```
theValueOf(mybuffer,B), B=[DATA|B2].
```

The messages `makeTheValueOf/2`, `">/2` and `"</2` are directional system methods, and therefore do not require any mode information from the user.

3. *Add `releaseInstance/1` messages to the code of callers using anonymous messages.*

The code does not contain messages to anonymous targets.

4. *Add `conditionsEnd/0` messages to methods that use a guarded or an extended rendezvous.*

The `retrieve/1` method of the class `buffer`, and the `store/1` method of the class `boundedBuffer` use a guarded rendezvous. The `retrieve/1` method accepts a rendezvous when the buffer is not empty; i.e., the list length is greater than zero, therefore the `conditionsEnd/0` message is inserted after the message `S>0`. The `store/1` method only accepts a rendezvous when the buffer is not full; i.e., the list length is less than the upper bound; therefore the `conditionsEnd/0` message is inserted after the message `S<MAX`.

5. *Restructure the code to remove recursive entries that accept nested rendezvous.*

No recursive entries exist in the code.

6. *Annotate all messages to explicit targets with the class of the target object.*

Although the targets that are represented as symbolic names (e.g., `common` and `super`), can be annotated with their class by an automated scanning of the code, for consistency here, all targets are annotated manually.

7. *Remove meta programming from the code.*

No meta programming exists in the code.

8. *Restructure methods to facilitate unfold transformations.*

Since the problem uses `waitToAccept/1` only in the `begin/0` methods, unfold transformations are not required.

9. *Provide type definitions and declarations for the MLog code.*

The type *fruit* is defined explicitly as an enumeration type, and is used to describe the contents of the buffers; i.e., the producer produces fruit and the consumer consumes fruit. In this program, after all user-defined methods and instance variables have been typed, it is determined that they are all monomorphic. Further typing of the clause bodies is required only if some of the messages in the user-defined methods invoke polymorphic methods.

The typing of `new/3` is predefined as `new(string, mode_t, _)`. The underscore in the third position indicates no type is required from the user. A record type is derived by the system; each field of the record has the type of one of the instance variables of the target class. The typing of `theValueOf/2` and `makeTheValueOf/2` depends on the type of the accessed instance variable; these messages are implicitly typed when the instance variable is typed.

After the user has produced the annotated code automated preprocessing derives the following data for use during the translation.

1. There are no classes that accept anonymous messages.
2. There are no methods that use an extended rendezvous.
3. The methods that are redefined include `store/1` which is defined in the class `buffer` and the class `boundedBuffer`, and the methods `theValueOf/2` and `makeTheValueOf/2`. The latter two are redefined only with respect to the instance variable `myBuffer`. The redefinition occurs since the instance variable is inherited.
4. Methods are inherited as follows.
 - a. The class `buffer` must inherit the method `listLength/2` from class `object` to satisfy a message to `self` occurring in the method `size/1`.
 - b. The class `boundedBuffer` must inherit the method `size/1` from class `buffer` to satisfy a message to `self` occurring in the method `store/1`. This class also inherits method `retrieve/1` to satisfy a call from an object of class `consumer` that uses the method `consumerLoop/1`.

The translation of each class receives its private copy(s) of the methods `new/3`, `theValueOf/2` and `makeTheValueOf/2`, therefore these methods are not included on the lists of inherited methods. Similarly, system methods that are translated as in-line code are not included; for example `"<"/2`.

5. Typing polymorphic messages in the body of methods is required only for the message `listLength/2` which is found in the method `size/1`. All other messages invoke monomorphic methods. The algorithm presented in figure 4.13, derives a typing of: `listLength(fruit_list, integer)`.

6.3.2 Translation processing

The translation of the producer consumer problem produces the package `GLOBAL` and the procedure `START_UP` which acts as the main-line of the translated program. Next, each of the user-defined classes is translated. The translated code of the entire producer consumer problem is presented in appendix B. In the translated code, lists are represented by an abstract data type which is implemented by the generic packages `LIST_BASIC` and `LIST_SEARCH`. The package `LIST_BASIC` provides basic list functions; the package `LIST_SEARCH` builds on the basic functions to provide *find* and *remove* utilities for lists. Both list packages draw extensively from the work of Booch on Ada software components [45]. The translation of the class `boundedBuffer` is examined in detail.

From pretranslation processing, it is known that the bounded buffer does not accept anonymous messages or respond to extended rendezvous, however it does contain three redefined methods: `store/1`, `theValueOf(mybuffer, _)` and `makeTheValueOf(mybuffer, _)`. These methods are redefined in the class `boundedBuffer`, and were defined originally in the class `buffer`. The translation of the class uses the technique presented in figure A.1.

The class contains the instance variables `bound` and `mybuffer` (inherited from class `buffer`), therefore: (1) a record type `IVAR_REC_T` is defined to group an object name with its instance variables, (2) the list abstraction `IVAR_PTR_REC_LIST_T` allows a list of `IVAR_PTR_REC_T` records (records associating an object name with the pointer to its

CLASS MAIN:

```
class: main of object.
```

```
begin :-
```

```
%      mode: new(in,in,in), type: new(string,mode_t,_).
      boundedBuffer-new(common,active,[bound=5]),
%      mode: new(out,in,in), type: new(string,mode_t,_).
      producer-new(P,active,[]),
%      mode: new(out,in,in), type: new(string,mode_t,_).
      consumer-new(C,active,[]),
      mainConsole-writeln("All tasks initialized").
```

CLASS PRODUCER:

```
class: producer of object.
```

```
begin :-
```

```
%      mode: new(out,in,in), type: new(string,mode_t,_).
      buffer-new(X,passive,[myBuffer=[apples,oranges,pears,kiwi]]),
      producerLoop(X).
```

```
%      mode: producerLoop(in).
%      type: producerLoop(string).
```

```
producerLoop(X) :-
      X(buffer)-retrieve(D),
      mainConsole-(write("sending "),writeln(D)),
      common(boundedBuffer)-store(D),
      producerLoop(X).
```

```
producerLoop(X).
```

CLASS CONSUMER:

```
class: consumer of object.
```

```
begin :-
```

```
%      mode: new(out,in,in), type: new(string,mode_t,_).
      buffer-new(X,passive,[]),
      consumerLoop(X).
```

```
%      mode: consumerLoop(in).
%      type: consumerLoop(string).
```

```
consumerLoop(X) :-
      mainConsole-writeln("waiting to receive"),
      common(boundedBuffer)-retrieve(D),
      X(buffer)-store(D),
      X(buffer)-size(S),
      mainConsole-(write(D),write(" received"),nl),
      mainConsole-(write(S),write(" elements received"),nl),
      consumerLoop(X).
```

Figure 6.3 Annotated code for producer consumer problem, classes main, producer and consumer.

CLASS BUFFER:

```

class:  buffer of object.

%      type definition: fruit = {apples, oranges, pears, kiwi}.
%      type:  [mybuffer : fruit_list].
ivars:  [myBuffer=[]].

%      mode:  store(in).
%      type:  store(fruit).
store(DATA) :-
    theValueOf(myBuffer, B),
    makeTheValueOf(myBuffer, [DATA|B]).

%      mode:  retrieve(out).
%      type:  retrieve(fruit).
retrieve(DATA) :-
    size(S), S > 0,
    conditionsEnd,
    theValueOf(myBuffer, B), B=[DATA|B2],
    makeTheValueOf(myBuffer, B2).

%      mode:  size(out).
%      type:  size(natural).
size(S) :-
    theValueOf(myBuffer, B),
    listLength(B, S).

```

CLASS BOUNDED BUFFER:

```

class:  boundedBuffer of buffer.

%      type: [bound : natural].
ivars:  [bound].

%      mode: store(in).
%      type: store(fruit).
store(DATA) :- theValueOf(bound, MAX),
    size(S), S < MAX,
    conditionsEnd,
    super(boundedBuffer)-store(DATA).

```

Figure 6.4 Annotated code for the producer consumer problem, classes buffer and boundedBuffer.

instance variables) to be declared, and (3) the translation of `theValueOf/2` generates the procedures `THEVALUEOF_BOUND(...)` and `THEVALUEOF_MYBUFFER(...)`. The translation of the method `makeTheValueOf/2` also generates a separate procedure for each instance variable.

The class does have active objects, therefore: (1) the task type `ACTIVE_CLASS` is defined with an `INIT` entry and entries corresponding to the methods `retrieve/1` and `store/1`, (2) the record type `TASK_NAME_T` is defined to associate the object names and pointers to the equivalent task, and (3) the procedure `GET_TASK_PTR` and the mutex task are defined; the procedure is used by caller objects to obtain the task pointer of the active object they wish to call. The task is used to prevent concurrent access by processes adding new elements to the list. The method `new/3` is translated as the procedure `MAKE_NEW`. The user defined method `store/1` is translated as a procedure, and since `retrieve/1` is (1) inherited, and (2) invoked by a message with an explicit target the procedure `RETRIEVE` is added to the package. This procedure is implemented as a call-through to the procedure `RETRIEVE` in the package `BUFFER_CL`.

Using figures A.2 and A.3, the translation of method `store/1` into a procedure declares local variables to represent the variables used as message arguments (`MAX`, `S`); then each message is translated as follows.

- `theValueOf(bound, MAX)` is translated by a call to the procedure `THEVALUEOF_BOUND(...)`.
- `size(S)` which is inherited from class `buffer` is translated by a call to the equivalent procedure in the package `BUFFER_CL`.
- `S < MAX` is translated by an in-line *if* statement, this is followed by the assignment of the correct value to the `SUCCESS` variable.
- `conditionsEnd` is ignored since the method is being translated into a procedure.
- `super(boundedBuffer) - store(DATA)` is translated as a call to the equivalent procedure in the superclass, `buffer`. Note that, although `store/1` is a redefined method, a *case* statement is not required to enumerate all choices for the correct

version to call, since the explicit target has uniquely identified which version to call.

Using the code presented in figure 6.4, the translation of the class `boundedBuffer` into the task type `ACTIVE_CLASS` is built as follows.

- The entry methods `store/1` and `retrieve/1` become the entries in the task specification. The `INIT` entry is defined also.
- The local variables, required by the messages in all entry methods of the class, are declared. In addition the guard variables are declared for the guarded entries (`STORE`, `RETRIEVE`).
- The accept alternative for the `INIT` entry begins the body of the task `ACTIVE_CLASS`; this entry is used to initialize the task's environment record.
- The `begin/0` method is translated. In the `boundedBuffer` class this is the default method provided by the system. This method becomes the main-line code for the task. The translation consists of a loop containing a selective wait with an accept entry generated for each entry method in the class.

Using figure A.5 the methods `store/1` and `retrieve/1` are translated into entries. Both methods are guarded, (as indicated by the `conditionsEnd/0` messages), and are translated as per section 5.2.2. The guard methods are translated and the code is placed before the selective wait statement; code is added to set the entry's guard. Since the `retrieve/1` method is guarded it is translated completely in the bounded buffer's task type. The `retrieve/1` method contains the messages, `theValueOf(mybuffer,B)` and `makeTheValueOf(mybuffer,B2)`. Since the instance variable `mybuffer` is inherited by class `boundedBuffer` these two methods are defined both in class `buffer` and class `boundedBuffer`, therefore as per section 3.4.1 and figure A.3 *case* statements are inserted in the translated code of the equivalent procedure to the `retrieve/1` method to ensure the correct version of the redefined methods are invoked.

The bounded buffer class demonstrates the transformation of classes with instance variables and active objects. The translated code of the buffer class demonstrates the transformation of classes without active objects. An inspection of the code, (i.e., all

new/3 messages directed to buffer) reveals that no active objects are required, therefore the code normally produced to support active objects is not required. This has a side effect on the producer and consumer objects which call buffer objects. In a caller object, the code resulting from the translation of a message to an explicit target, includes a check to determine if the mode of the target is active or passive, and the appropriate code is inserted in each case. Since class `buffer` objects are all passive, the translated code only considers the passive case. The translation of the producer and consumer classes also demonstrates the transformation of classes without instance variables, and the transformation of strictly caller classes. Strictly caller classes only accept messages to self, therefore the procedure `GET_TASK_PTR` is omitted from the generated Ada code.

Chapter 7

Conclusions

7.1 Summary

MLog is a rapid prototyping language designed to take advantage of object-oriented and declarative programming techniques. Additionally, a powerful set of IPC primitives is provided to support concurrent programming. This thesis has presented a cohesive set of transformations that allow the translation of a substantial portion of the MLog language into Ada, a standard production language for embedded systems.

Due to the significant differences between MLog and Ada, and the cohesiveness of the MLog syntax, the transformation process occurs in two steps: (1) pretranslation which is suitable for semi-automated processing and (2) translation which is suitable for automated processing.

Pretranslation processing

Pretranslation processing involves restructuring and/or annotating the MLog code. An analysis of methods used to transform object-oriented programs into procedural Ada programs showed that object-oriented paradigms could be emulated completely [17, 18, 20] although these solutions exacted a price in run-time efficiency. An equivalent situation exists with respect to declarative paradigms. In order to minimize the run-time costs only a subset of MLog's capabilities are supported, and the pretranslation processing is designed to restructure the code to remove those capabilities that would exact a high cost in run-time efficiency. A less severe solution than restructuring the code is annotating the code. This approach preserves the function in the MLog code but the extra information encoded in the annotation increases the efficiency of the resulting Ada code. The object-oriented, declarative and IPC features of MLog are summarized to identify those features that cause restructuring and those that cause annotation.

The object-oriented constructs in MLog do not require restructuring of the code during pretranslation processing. However, annotation of the MLog code is required to provide the class of the target on all messages to explicit targets. This allows static binding to replace dynamic binding in most cases.

The declarative aspects of MLog are the least supported by the transformations. The code must be restructured to remove non-deterministic methods, meta-programming (except for rendezvous/post-rendezvous processing), and all message arguments that are partially instantiated; (methods must be restructured to ensure all *in* arguments are instantiated and all *out* arguments are uninstantiated). The elimination of non-deterministic methods also eliminates the requirement to emulate backtracking between methods in Ada. Backtracking that occurs between the clauses of a method is supported by the transformations. The restructured MLog code can be translated without building an Ada inference engine or emulating variables as data types in Ada, however the code requires annotation to provide the type and mode of method arguments.

The majority of IPC constructs are supported under the transformations, but translation is not possible when a recursive method is used to accept a rendezvous; i.e., the MLog code must be restructured in these cases. Annotation of IPC code in MLog is used to indicate if a rendezvous is guarded or extended. This allows progressively more efficient transformations for the unguarded, guarded, and extended rendezvous which are all available in MLog. MLog's messages to anonymous targets also are annotated to indicate when the anonymous target is available to rendezvous with a new caller. Post-rendezvous processing is annotated implicitly by requiring the use of a predefined method name.

Translation processing

The translation of the restructured, annotated MLog code produces compilable Ada code. Classes, methods, instance variables and objects are represented as packages, subprograms, and data records, respectively. To support concurrency each active object is associated with a task, in addition to being identified with a data record. True inheritance is not translated, rather static calls are made to the invoked procedure. The identification

of the invoked procedure is obtained by static analysis of the inheritance hierarchy for calls to `self`, and by static inspection of the annotated class on messages to explicit targets. Dynamic binding is provided (by case statements) for messages that invoke methods which are redefined within a class hierarchy path.

The operational semantics of the declarative paradigms in the restructured/annotated code are emulated in the translated code by the use of conditional statements that check whether a subprogram succeeds or fails. A set of algorithms for translating the MLog IPC primitives is presented. The choice of which algorithm is used depends on the message passing characteristics specifically the characteristics of the caller object, acceptor object (current object), current process and the characteristics of the invoked method.

The transformations were applied manually to the producer consumer problem. The complete program required 1120 lines of Ada code, of which 916 lines resulted from the translation of user defined classes. It is difficult to assess the overall efficiency of the transformed programs, however an indication of the run-time costs of object creation and message passing are shown in table 7.1.

The efficiency of the resulting Ada code was the primary criteria in the selection of the MLog features that would be supported by the transformations; i.e. during the pretranslation analysis. While the translation analysis strived to produce efficient code, the stress was placed on a thorough analysis and straightforward transformations that increased the clarity of the resulting Ada code. The emphasis on code clarity introduced some inefficiencies into the transformed code. However, the optimizations which are required to remove the inefficiencies are not considered significant since the optimizations can easily be incorporated into an automated system. For example, the `ENVIRONMENT` argument in calls to task entries is not required since tasks have a predefined environment, but its inclusion provides a consistency to the code in the examples.

Further study is required to minimize the code bulk generated by the transformations. The code bulk results from Ada's natural verbosity, in addition to the following characteristics of the transformations.

Operation	Number of rendezvous	Number of procedure calls
create an object	1-3 note 1	1 (to send the message 'new/3')
send a message to 'self'	0	1 (to send the message)
send a message to a passive object (excluding 'self')	0	1 (to send the message) note2
send a message to an active object (excluding 'self')	1 (to send the message) note3	1 (to obtain the task pointer) note2
<p>Note 1: A rendezvous is required to update: (1) the global list of objects in the program, (2) the appropriate class' list of objects (instance variable records) and (3) the appropriate class' list of active objects (task pointers). The latter two rendezvous may not be required for all objects.</p> <p>Note 2: Two additional procedure calls may be required to obtain the mode and object_id of the target.</p> <p>Note 3: A second rendezvous may be required if the message generates an extended rendezvous in MLog.</p>		

Table 7.1 Run-time costs for message passing and object creation.

- A large number of *if* statements is required to emulate MLog's declarative semantics. At present each subprogram invocation (equivalent to sending a message) is encased in an *if* statement. This can be optimized to some degree by using nested *if* statements. However, if it can be determined that a message never fails; (i.e., the method it selects never fails), then the encasing *if* statement is not required. For example, messages following a `conditionsEnd/0` message should never fail. This type of analysis was not attempted.
- Some methods (those selected by messages with explicit targets) are translated as both procedures and task entries. This duplication of code can be minimized if the mode of each explicit target is known at compile time since methods that are invoked only by passive objects do not need to be translated as task entries. This data could be obtained from an analysis of the annotated code if all objects of a class had the

same mode.¹⁵ Alternatively, if objects in a class are not limited to a single mode, the user could annotate each explicit target with its mode. Duplication of code cannot be totally eliminated since an active object accessing its own method is translated as a procedure call while any other object accessing that method is translated as a task entry call.

Bond [6] suggested a set of MLog methods be defined to provide commonly used data structure manipulations; (e.g., list operations). The translation of these new MLog methods could be hand tailored to maximize efficiency, and the use of libraries to store the translated methods would reduce code bulk.

7.2 Future enhancements

Automated typing of MLog code is the outstanding problem that must be solved if transformation of MLog to Ada is to be achieved with only minimal user intervention. This thesis investigated an approach to typing MLog based on the Prolog model of TIPs that use abstract interpretation. Major difficulties, that result from MLog's syntactic cohesiveness and support for concurrent processes, must be resolved if this approach is to be successful. Furthermore, Prolog TIPs remain an area of current research, and no Prolog TIP was discovered that supported meta-programming or addressed concurrency. An alternative method for typing MLog is to use the object-oriented approach to typing where classes are types and sub-classes are sub-types [12]. Ada++ [22] appears to have pursued this approach partially, by assigning classes a type and then modifying the type resolution algorithm to include inheritance. However, the problems of the Ada++ application are limited as follows.

- The Ada++ typing procedure is required to resolve only type inconsistencies relating to class types and inheritance. Ada++ is a superset of Ada, therefore, the user has coded all the methods in Ada and de facto has included all required standard Ada type information.

¹⁵ The code has been annotated with the classname of each explicit target.

- The Ada++ literature does not reference active objects.

The MLog typing problem encompasses the typing of all variables, and must provide the type information for all threads of control in a program.

Further investigation into typing MLog programs should include both the Prolog and object-oriented approaches perhaps used in combination. However, since all entities in MLog are not objects, and since there is a lack of any type data in an MLog program, it is felt that the Prolog model for deriving type data is the more promising.

The second outstanding problem in transforming MLog into Ada is the implementation of an automated transformation process. The investigation of the transformations was aimed at facilitating the use of MLog prototypes as the basis of accurate timing data for new systems, and as the generative basis of the production systems. This will occur only if the process of applying the transformations is automated.

Finally, the correctness of the transformations presented must be proved if the user is to have confidence in the equivalency of the MLog and the Ada code. Karam et al [39] proved the correctness of the transformations for the extended rendezvous that are used in this thesis; however, work is still required to prove the correctness of the remaining transformations.

Final Comment

This thesis has provided a set of transformations that translate a significant subset of MLog's capability into Ada. The manual application of these transformations has been successfully accomplished. The restrictions imposed on the MLog code by the transformations may or may not require significant program restructuring. Significant changes should not be required if the MLog program's eventual translation into Ada is a requirement in the design phase of the program's development.

The MLog to Ada transformations presented, in this research, are the first known attempt to simultaneously map object-oriented, declarative and concurrent code into Ada. Furthermore, transformations from MLog to Ada do not require detailed programming

knowledge of Ada by the user, unlike the other known object-oriented systems that map into Ada. Although MLog's complete functionality is not supported and further work is required, the presented transformations indicate that the use of MLog prototypes for Ada systems is a viable option.

BIBLIOGRAPHY

- [1] "Draft report on requirements for a common prototyping system," *SIGPLAN Notices*, vol. 24, no. 3, pp. 93–165, March 1989. R.P. Gabriel, ed.
- [2] S. Hekmatpour and D. Ince, *Software Prototyping, Formal Methods and VDM*. International Computer Science Series, Don Mills, Ontario: Addison-Wesley, 1988.
- [3] G. Karam, "Mlog: A language for prototyping concurrent systems," *tr sce-88-3*, Dept. of Systems and Computer Eng., Carleton University, Ottawa, Ontario, CANADA, November 1988.
- [4] R. Venken and M. Bruynooghe, "Prolog as a language for prototyping of information systems," in *Approaches to Prototyping* (R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Zullighoven, eds.), pp. 447–458, New York: Springer-Verlag, 1984.
- [5] J. Wilson, "An evaluation of MLog in a CASE environment," in *tr sce-89-9*, pp. 1–56, Dept. of Systems and Computer Eng., Carleton University, Ottawa, Ontario, Canada, April 1989.
- [6] G. Bond, "An evaluation of MLog in a CASE environment," in *tr sce-89-9*, pp. 1–18, Dept. of Systems and Computer Eng., Carleton University, Ottawa, Ontario, Canada, April 1989.
- [7] G. Karam, "The MLog user guide," *tr-sce-89-15*, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, May 1989.
- [8] M. Stefik and D. Bobrow, "Object-oriented programming: Themes and variations," *AI Magazine*, vol. VI, no. 4, pp. 40–62, Winter 1986.
- [9] R. Buhr, *Practical Visual Techniques in System Design: with applications to Ada*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1990.
- [10] G. Booch, "Object-oriented development," *IEEE Transactions on Software Engineering*, vol. SE12, no. 2, pp. 211–221, February 1986.
- [11] B. Cox, *Object-Oriented Programming, An Evolutionary Approach*. Don Mills, Ontario, Canada: Addison-Wesley, 1987.

- [12] P. Wegner, "The object-oriented classification paradigm," in *Research Directions in Object-Oriented Programming* (B. Shriver and P. Wegner, eds.), pp. 479-560, Cambridge, Massachusetts: MIT Press, 1987.
- [13] E. Seidewitz, "Object-oriented programming in Smalltalk and Ada," *OOPSLA '87 Conference Proceedings, SIGPLAN Notices*, pp. 202-213, October 1987.
- [14] *Smalltalk/V Tutorial and Programming Handbook*. Los Angeles, California: digitalk inc., 1986.
- [15] E. Perez, "Simulating inheritance with Ada," *Ada Letters*, vol. VIII, no. 5, pp. 37-46, September/October 1988.
- [16] B. Meyer, "Genericity versus inheritance," *OOPSLA '86 Conference Proceedings, SIGPLAN Notices*, pp. 391-405, September 1986.
- [17] R. Simonian and M. Crone, "Innovada. True object-oriented programming in Ada," *J. of Object-Oriented Programming*, pp. 14-21, November/December 1988.
- [18] V. Kovarik Jr. and S. Nies, "Supporting object-oriented programming within Ada: Extending the paradigm," in *Fourth Annual Conference on Artificial Intelligence and Ada, AIDA-88*, pp. 13-1-13-6, November 1988.
- [19] G. Pascoe, "Elements of object-oriented programming," *BYTE Magazine*, pp. 15-20, August 1986.
- [20] A. Di Maio, C. Cardigno, R. Bayan, C. Destombes, and C. Atkinson, "DRAGOON: an Ada-based object oriented language for concurrent, real-time, distributed systems," in *Proceedings of the Ada-Europe International Conference Madrid 13-15 June 1989* (A. Alvarez, ed.), pp. 3-15, Cambridge: Cambridge University Press, 1989.
- [21] C. Donaldson, "Dynamic binding and inheritance in an object-oriented Ada design," in *Proceedings of the Ada-Europe International Conference Madrid 13-15 June 1989* (A. Alvarez, ed.), pp. 16-25, Cambridge: Cambridge University Press, 1989.
- [22] J. Forestier, C. Fomarino, and P. Franchi-Zannettacci, "Ada++ A class and inheritance extension for Ada," in *Proceedings of the Ada-Europe International Conference Madrid 13-15 June 1989* (A. Alvarez, ed.), pp. 3-15, Cambridge: Cambridge University Press, 1989.
- [23] G. Pitette, "Adlog : an Ada components set to add logic to Ada," in *Proceedings of the 7th Joint Ada Europe/AdaTEC Conference*, pp. 174-188, 1988.

- [24] I. Attali and P. Franchi-Zannettacci, "Inference system environment for Ada," in *Ada components : libraries and tools: proceedings of the Ada-Europe International Conference* (S. Tafvelin, ed.), pp. 3–18, Cambridge, Cambridgeshire: Cambridge University Press, May 1987.
- [25] J. Nilsson, "On the compilation of a domain-based Prolog," in *Information Processing 83* (R. Mason, ed.), pp. 293–298, Amsterdam: Elsevier Science Publishers B.V.(North Holland), 1983.
- [26] J. Weiner and S. Ramakrishnan, "A piggy-back compiler for Prolog," in *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, (Atlanta, Georgia), pp. 288–296, June 1988.
- [27] K. Knight, "Unification: A multidisciplinary survey," *ACM Computing Surveys*, vol. 21, no. 1, pp. 93–124, March 1989.
- [28] C. Mellish, "Some global optimizations for a Prolog compiler," *J. Logic Programming*, vol. 2, no. 1, pp. 43–66, April 1985.
- [29] W. Drabent, "Do logic programs resemble programs in conventional languages?," in *Proceedings of 4th Symposium on Logic Programming, San Francisco, Sept 1987*, pp. 389–396, Washington, D.C.: Computer Society Press of the IEEE, 1987.
- [30] J. Boyd and G. Karam, "Prolog in C," tech. rep., Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, Ottawa, Ontario, Canada, March 1989.
- [31] J. Cohen, "Describing Prolog by its interpretation and compilation," *Communications of the ACM*, vol. 28, no. 12, pp. 1311–1324, December 1985.
- [32] D. Hardin and K. Albin, "Taking inference to task," in *Proceedings of AIDA-88, Fourth Annual Conference on Artificial Intelligence & Ada*, pp. 8–1 – 8–9, Dept of Computer Science, George Mason University, November 1988.
- [33] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *ACM Computing Surveys*, vol. 17, pp. 471–522, 1985.
- [34] M. Bruynooghe, "Adding redundancy to obtain more reliable and more readable Prolog programs," in *Proceedings 1st International Conference on Logic Programming*, (Marseille), pp. 129–134, ADDP-GIA, Faculte des Sciences de Luminy, 1982.

- [35] A. Mycroft and R. O'Keefe, "A polymorphic type system for Prolog," *Artificial Intelligence*, vol. 23, pp. 295-307, 1984.
- [36] K. Horiuchi and T. Kanamori, "Polymorphic type inference in Prolog by abstract interpretation," in *Logic Programming '87, Proceedings of the 6th Conference, Lecture Notes in Computer Science*, pp. 195-214, New York: Springer-Verlag, 1987.
- [37] J. Xu and D. Warren, "A theory of types and type inference in logic programming," tr 88/15, State University of New York, Stony Brook, New York, October 1988.
- [38] R. Buhr, *System Design with Ada*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1984.
- [39] G. Karam, C. Stanczyk, and G. Bond, "Critical races in Ada programs," *IEEE Transactions on Software Engineering*, vol. 15, no. 11, pp. 1471-1480, November 1989.
- [40] S. Debray, "Unfold/fold transformations and loop optimization of logic programs," in *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, (Atlanta, Georgia), pp. 297-307, June 1988.
- [41] H. Tamaki and T. Sato, "Unfold/fold transformation of logic programs," in *Proceedings of the Second International Logic Programming Conference*, (Uppsala), pp. 127-138, 1984.
- [42] T. Kanamori and K. Horiuchi, "Construction of logic programs based on generalized unfold/fold rules," in *Logic Programming Proceedings of the Fourth International Conference* (J. Lassez, ed.), vol. 2, pp. 744-768, Cambridge, Massachusetts: MIT Press, 1987.
- [43] L. Sterling and E. Shapiro, *The Art of Prolog*. Cambridge, Massachusetts: MIT Press, 1986.
- [44] U.S. Department of Defense, *The programming language Ada Reference Manual, ANSI/MIL-STD-1815a*. New York: Springer-Verlag, 1983.
- [45] G. Booch, *Software Components with Ada, Structures, Tools, and Subsystems*. Don Mills, Ontario, Canada: The Benjamin/Cummings Publishing Co., 1987.

Appendix A

Techniques for the Translation of MLog Components into Ada

A.1 Translation of user-defined classes

A technique for translating classes into packages is presented in figure A.1. The transformations of components of a class are presented in the following figures.

- Figure A.2 translates a method into a procedure.
- Figure A.3 translates a message within a procedure.
- Figure A.4 translates a class into a task type.
- Figure A.5 translates a method into a task entry.
- Figure A.6 translates a message within a task entry.

The translation of messages in a task entry is similar to the translation in a procedure and uses the method presented in figure A.2 modified to accommodate the differences outlined below.

- If the message is `waitToAccept / 1` or `accept / 1` then it is translated as a nested `accept` as per table 5.1.
- If the target of the message is `self`, it is translated as a procedure call to the package where the method is defined. Case statements are not required in a task entry, regardless of whether the method is redefined, [cref. § 3.4.1] since the target (`self`) has uniquely identified the correct class to call (i.e., the class of the current object, or the superclass in the class hierarchy where the method is defined).
- If the message contains an explicit target there is no requirement to check if the target equals the current process. In a task, if the target equals the current process, then by definition it also equals the current object, but if the target equals the current object, a message is being sent to self. In MLog code to be translated, messages to self are restricted to the default syntax (no explicit target).

TRANSLATION OF A CLASS INTO A PACKAGE:

```

begin
  if (the class has instance variables) then
    -define a record to store an object's instance
      variables;
    -define a list to store pointers to the records
      containing instance variables;
    -define a mutex task to protect the list from
      concurrent writers;
    -define a copy of theValueOf/2 and makeTheValueOf/2
      for each instance variable in the class;
  end if;
  if (class has active objects) then
    -define the task type that represents active objects
      [cref. section 6.2.2];
    -define the records that associate a task pointer
      and an object name, and define a list to store them;
    -define a mutex task to protect the list from
      concurrent writers;
    -define the GET_TASK_PTR procedure;
    if (active objects accept the extended rendezvous)
      then
        -define the record type used by the pending
          caller queue, and the discrete type that defines
          the variant of the record;
        -define the pending caller queue;
      end if;
  end if;
  -translate the method sw/3 as the procedure MAKE_NEW;
  -translate each user defined method in the class as
    a procedure;
  if (any methods are inherited by the class and invoked
    by messages with explicit targets) then
    -translate the methods as call-through procedures,
      [cref section 3.4];
  end if;
end;

```

Figure A.1 Translation of a class into a package.

TRANSLATION OF A METHOD INTO A PROCEDURE:

```

begin

    -declare local variables required to support
    unification of the clause(s) head and variables for
    arguments in the messages

    -make required instantiations of polymorphic methods that
    are invoked by the current method

    while (untranslated clauses exist) loop
        -translate unification found in the clause head's
        in arguments;
        while (untranslated messages in the clause) loop
            -get next message
            if (message = waitToAccept/1 or
                message = accept/1) then
                -translate as SUCCESS := FALSE and terminate
                translation of the current clause;
            else
                -translate the message in accordance with
                figure A.3;
            end if;
        end loop;
        -translate unification found in the clause head's
        out arguments;
    end loop;
end;

```

Figure A.2 Translation of a user's method into a procedure

TRANSLATION OF A MESSAGE WITHIN A PROCEDURE:

```

begin
    -get message;
    if (target = self) then
        if (message is not redefined) then
            translate as a procedure call to package where
            method is defined;(see note);
        else
            -translate as a case statement that provides
            the correct procedure call for each class of
            object that use the selected method;
            [cref. section 3.4.1];
        end if;
    else if (target \= self) then
        if (the message is an anonymous message) then
            -translate as a call to the dispatcher for
            the class, followed by a call to the assigned
            object; [cref. section 5.2.4]
        else if (target is a symbolic name equal to super) then
            -translate as a procedure call to the package
            that defines the selected method;
        else if (target is a symbolic name not equal to super)
            then
            do steps (2) and (3) below;

        else if (target is a variable) then
            -translation includes [cref. figure 3.14]:
            (1) adding code to check if target is instantiated
            to super and inserting the required procedure
            call,[cref. section 4.2.2];
            (2) adding code to check if target is passive or
            if the target equals the current process, and
            inserting the required procedure call;
            (3) otherwise inserting a call to a task entry;
            if (the task entry uses an extended
                rendezvous [cref. table 3.1, 3.3])
                then
                -translation includes adding code
                required to support an extended
                rendezvous (in the caller object),
                [cref. section 4.2.2, 5.2.3]
            end if;
        end if;
    end if;
end;

```

Note: In some cases, system methods can be translated as in-line code, and a procedure call will not be required. These methods are summarized in table A.1.

Figure A.3 Translation of a message within a procedure

TRANSLATION OF A CLASS INTO A TASK TYPE:

begin

- insert the task type definition in the package specification;**
- insert the task body in the package body;**
 - the task body contains the following:**
 - (1) declaration of local variables required to support unification, message arguments, and guards for entries;**
 - (2) the predefined code for the INIT entry;**
 - (3) the code to inform the dispatcher of the objects existence, [cref. section 5.9];**
 - (4) translation of the begin/0 method for the class;**

end;

Note: In order to translate the begin/0 method of a class whose active objects accept messages (from other objects), it is necessary to translate the entry methods of the class into task entries.

Figure A.4 Translation of a class into a task type.

TRANSLATION OF A METHOD INTO A TASK ENTRY:

```

begin
  -get an entry method
  if (the entry method is unguarded) then
    if (method does not contain waitToAccept/1 or
        accept/1 messages) then
      -translate the body of the entry by a call to
        the procedure version of the method;
    else
      -translate the method message by message in
        the entry body;
    end if;
  if (the method requires a guarded rendezvous) then
    if (the method is not recursive) then
      -translate as a guarded entry, [cref. 5.2.2];
    else if (the method is recursive) then
      -translate the guard messages as per section 5.2.2
        but replace the body of the entry with a call to
        the procedure version of the method; (see note below);
    end if;
  if (the method requires an extended rendezvous) then
    if (the method is not recursive) then
      -translate as an extended rendezvous entry,
        [cref. 5.2.3];
    else if (the method is recursive) then
      -translate the guard messages within the entry body
        but replace the remaining messages of the method
        with a call to the procedure version of the method;
        (see note below);
    end if;
  end if;
end if;
end;

```

note: The call to the equivalent procedure causes the guard messages to be reprocessed but avoids the deadlock that a recursive entry causes in Ada.

Figure A.5 Translation of a method into a task entry

TRANSLATION OF A MESSAGE WITHIN A TASK ENTRY:

```

begin
  -get message;
  if (target = self) then
    translate as a procedure call to package where
      method is defined; (see note);
  else if (target \= self) then
    if (the message is an anonymous message) then
      -translate as a call to the dispatcher for
        the class, followed by a call to the assigned
        object; [cref. section 5.2.4]
    else if (target is a symbolic name equal to super) then
      -translate as a procedure call to the package
        that defines the selected method;
    else if (target is a symbolic name not equal to super)
      then
        do steps (2) and (3) below;
    else if (target is a variable) then
      -translation includes [cref. figure 3.14:
        (1) adding code to check if target is instantiated
          to super and inserting the required procedure
          call, [cref. section 4.2.2];
        (2) adding code to check if target is passive, and
          inserting the required procedure call;
        (3) otherwise inserting a call to a task entry;
          if (the task entry uses an extended
            rendezvous [cref. table 3.1, 3.3])
            then
              -translation includes adding code
                required to support an extended
                rendezvous (in the caller object),
                [cref. section 4.2.2, 5.2.3]
            end if;
          end if;
    end if;
  end if;
end;

```

Note: In some cases, system methods can be translated as in-line code, and a procedure call will not be required. These methods are summarized in table 6.1.

Figure A.6 Translation of a message within a task entry.

A.2 Translation of MLog system classes

The MLog system classes, *object of nil* and *behaviour of object* are translated as packages. Within these classes some methods are not supported, some are translated as subprograms which may or may not be generic, and some are translated by inline code. The translation of methods in class *object of nil* is outlined in table A.1, and the translation of methods in class *behaviour of object* is outlined in table A.2.

System methods that are translated as in-line code must ensure that the SUCCESS variable is set correctly. If the translation results in an assignment statement, the SUCCESS variable is not updated. If the translation results in a comparison of values, the SUCCESS variable is updated to reflect the results of the comparison.

Methods in class *behaviour* are used by both classes and metaclasses [7] and many are not supported by the translation as they require meta programming . Generally, those methods that query the status or definition of a class, an object or the class hierarchy can be translated, but those attempting to change the system components can not. The data structures required to support these methods can be derived from inspection of the static code, but should be built only if the method(s) requiring them are invoked by the program. Data structures required by the methods are indicated in table A.2.

Other system classes are not supported by the transformations; this includes the input/output classes.

Method	Comments	Translation
<code>!</code>	cut	section 3.2.1
<code>"/2</code>	conjunction of messages	section 3.2.1
<code>"/2</code>	disjunction of messages	section 3.2.1
<code>"="/2</code>	unifies	section 5.1
<code>"==" /2</code> <code>"\==" /2</code> <code>"@<" /2</code> <code>"@<" /2</code> <code>"@>" /2</code> <code>"@>" /2</code>	term comparisons (only strings and numbers are supported in MLog)	in-line code
<code>"=:" /2</code> <code>"\=" /2</code> <code>"<" /2</code> <code>"<" /2</code> <code>">" /2</code> <code>">" /2</code>	arithmetic comparisons	in-line code
<code>accept/1</code> <code>waitToAccept/1</code>	accept a message from a caller	section 5.2
<code>begin/0</code>	default main line for active objects	in-line code consisting of a loop that waits to accept any message
<code>class/1</code> <code>myself/1</code>	obtain the value from the environment record	procedure
<code>concatenate/3</code> <code>cocatenateList/2</code>		procedure
<code>convert/3</code>	type can be "number", "string" or "expression",	"number" and "string" are supported only
<code>delay/1</code>		in line code
<code>fail/0</code>		in-line code sets the SUCCESS variable to false
<code>is/2</code>		in-line code
<code>isKindOf/1</code>	a non-backtrackable version is supported	procedure
<code>isMode/1</code>	value is found in the GLOBAL data-store	procedure

Table A.1 Summary of translations for class *object of nil*. (Continued . . .)

Method	Comments	Translation
listLength/2	polymorphic	generic procedure
makeCharList/2		procedure
makeTheValueOf/2 theValueOf/2	used to access instance variables	section 6.2
name/1	value found in the GLOBAL data-store	procedure
not/1		in-line code
prefixOf/2		procedure
psort/4 sort/2		generic procedure
repeat/0		section 3.2.2
stringIndex/3 stringLength/2 substring/3 substring/4		procedure
debug/0 doesNotUnderstand/1 findall/3 hostCall/1 isA/2 isPriority/1 perform/1 performOnce/1 preempt/0 structure/3		not supported

Table A.1 Summary of translations for class *object of nil*.

Method	Comments	Translation
allInstances/1	data available in GLOBAL instance records	procedure
allSuperclasses/1	requires the class hierarchy to be stored	procedure
canUnderstand/1	requires a list of methods for each class and the class hierarchy to be stored	non-backtracking version supported procedure
canUnderstandList/1	as above	procedure
includesSelector/1	requires the list of methods defined for each class to be stored	non-backtracking version supported procedure
inheritsFrom/1	requires the class hierarchy to be stored	non-backtracking version supported procedure
instVarNames/1	requires names of instance variables to be stored	procedure
new/3	a unique version is produced for each class	procedure section 6.2
subclasses/1 superclass/1	requires the class hierarchy to be stored	procedure
Note: All other methods in the class are not supported.		

Table A.2 Summary of translations for class *behaviour of object*.

Appendix B: The Producer Consumer Problem: An Ada Solution

PACKAGE OBJECT_CL SPECIFICATION:

```
package OBJECT_CL is
  generic
    type LIST is private;
    with function LENGTH_OF(B : in LIST) return INTEGER;
    procedure LISTLENGTH(B : in LIST; S : out NATURAL);
end OBJECT_CL;
```

PACKAGE OBJECT_CL BODY:

```
package body OBJECT_CL is

  procedure LISTLENGTH(B : in LIST; S : out NATURAL) is
  begin
    S := LENGTH_OF(B);
  end LISTLENGTH;
end OBJECT_CL;
```

PACKAGE GLOBAL SPECIFICATION:

```
package GLOBAL is

  type      MODE_T           is (ACTIVE, PASSIVE);
  type      CLASS_T          is (MAIN, PRODUCER, CONSUMER, BUFFER,
    BOUNDED_BUFFER);
  type      ARG_MODE_T       is (MODE_IN, MODE_OUT);
  subtype   OBJ_NAME         is INTEGER;
  type      USER_NAME_REC(MAX_LENGTH : NATURAL) is record
    CUR_LENGTH : NATURAL;
    NAME       : STRING(1 .. MAX_LENGTH);
  end record;
  type      USER_INSTANCE_NAME_PTR is access USER_NAME_REC(40);

  type      INSTANCE_REC_T is record
    INSTANCE_NAME      : OBJ_NAME;
    CLASS_NAME         : CLASS_T;
    MODE_TYPE          : MODE_T;
    USER_INSTANCE_NAME : USER_INSTANCE_NAME_PTR;
```



```

end record;

type ENV_RECORD_T is record
  CUR_OBJECT      : OBJ_NAME;
  CUR_CLASS       : CLASS_T;
  CUR_PROCESS     : OBJ_NAME;
end record;

generic
  type ITEM is private;
  type LIST is private;
  with procedure CONSTRUCT(DATA_ITEM : in ITEM;
                           DATA_LIST : in out LIST);

package LIST_MUTEX is
  task MUTEX_TASK is
    entry UPDATE_LIST(DATA_ITEM : in ITEM;
                      DATA_LIST : in out LIST);

  end MUTEX_TASK;
end LIST_MUTEX;

task OBJ_NAME_CONTROL is
  entry GET_INSTANCE_ID(
    USER_NAME      : in out USER_INSTANCE_NAME_PTR;
    USER_NAME_MODE : in ARG_MODE_T;
    MODE_TYPE       : in MODE_T;
    CLASS_NAME      : in CLASS_T;
    X               : out OBJ_NAME);
end OBJ_NAME_CONTROL;

procedure GET_MODE(X : in OBJ_NAME; Y : out MODE_T);
procedure GET_OBJ_NAME(USER_NAME : in USER_INSTANCE_NAME_PTR;
                       SYS_NAME  : out OBJ_NAME);

end GLOBAL;

```

PACKAGE GLOBAL BODY:

```

with TEXT_IO, LIST_BASIC, LIST_SEARCH;
package body GLOBAL is
  package INSTANCE_LIST_T is new LIST_BASIC(INSTANCE_REC_T);

  function IS_EQUAL_INSTANCE_LIST(LEFT  : in OBJ_NAME;
                                   RIGHT : in INSTANCE_REC_T)
    return BOOLEAN;

  package SEARCH_INSTANCE_LIST is new LIST_SEARCH(

```

```

KEY => OBJ_NAME,
ITEM => INSTANCE_REC_T,
LIST => INSTANCE_LIST_T.LIST,
IS_EQUAL => IS_EQUAL_INSTANCE_LIST,
IS_NULL => INSTANCE_LIST_T.IS_NULL,
HEAD_OF => INSTANCE_LIST_T.HEAD_OF,
TAIL_OF => INSTANCE_LIST_T.TAIL_OF,
SWAP_TAIL => INSTANCE_LIST_T.SWAP_TAIL);

function IS_EQUAL_INSTANCE_LIST_USER_NAME(
LEFT : in USER_INSTANCE_NAME_PTR;
RIGHT : in INSTANCE_REC_T) return BOOLEAN;

package SEARCH_INSTANCE_LIST_USER_NAME is new LIST_SEARCH(
KEY => USER_INSTANCE_NAME_PTR,
ITEM => INSTANCE_REC_T,
LIST => INSTANCE_LIST_T.LIST,
IS_EQUAL => IS_EQUAL_INSTANCE_LIST_USER_NAME,
IS_NULL => INSTANCE_LIST_T.IS_NULL,
HEAD_OF => INSTANCE_LIST_T.HEAD_OF,
TAIL_OF => INSTANCE_LIST_T.TAIL_OF,
SWAP_TAIL => INSTANCE_LIST_T.SWAP_TAIL);

INSTANCE_LIST : INSTANCE_LIST_T.LIST;
procedure UPDATE_INSTANCE_LIST(X : in OBJ_NAME;
                                Y : in CLASS_T;
                                Z : in MODE_T;
                                W : in USER_INSTANCE_NAME_PTR);

package body LIST_MUTEX is
  task body MUTEX_TASK is
    begin
      loop
        select
          accept UPDATE_LIST(DATA_ITEM : in ITEM;
                              DATA_LIST : in out LIST) do
            CONSTRUCT(DATA_ITEM, DATA_LIST);
          end UPDATE_LIST;
        or
          terminate;
        end select;
      end loop;
    end MUTEX_TASK;
  end LIST_MUTEX;

  task body OBJ_NAME_CONTROL is
    NEXT_NAME : OBJ_NAME := 1;
  begin
    loop
      accept GET_INSTANCE_ID(

```

```

USER_NAME      : in out USER_INSTANCE_NAME_PTR;
USER_NAME_MODE : in ARG_MODE_T;
MODE_TYPE      : in MODE_T;
CLASS_NAME     : in CLASS_T;
X              : out OBJ_NAME) do

    X := NEXT_NAME;

    if USER_NAME_MODE = GLOBAL.MODE_IN then
        null;
    else
        USER_NAME.CUR_LENGTH := OBJ_NAME'IMAGE(NEXT_NAME)'LENGTH;
        USER_NAME.NAME(1 .. USER_NAME.CUR_LENGTH) :=
            OBJ_NAME'IMAGE(NEXT_NAME);
    end if;
    UPDATE_INSTANCE_LIST(NEXT_NAME, CLASS_NAME,
        MODE_TYPE, USER_NAME);
    NEXT_NAME := OBJ_NAME'SUCC(NEXT_NAME);

end GET_INSTANCE_ID;

end loop;
end OBJ_NAME_CONTROL;

function IS_EQUAL_INSTANCE_LIST(LEFT : in OBJ_NAME;
                                RIGHT : in INSTANCE_REC_T)
    return BOOLEAN is
begin
    return (LEFT = RIGHT.INSTANCE_NAME);
end IS_EQUAL_INSTANCE_LIST;

function IS_EQUAL_INSTANCE_LIST_USER_NAME(
    LEFT : in USER_INSTANCE_NAME_PTR;
    RIGHT : in INSTANCE_REC_T) return BOOLEAN is
    LENGTH : NATURAL;
begin
    LENGTH := RIGHT.USER_INSTANCE_NAME.CUR_LENGTH;
    return (LEFT.NAME(1 .. LEFT.CUR_LENGTH) =
        RIGHT.USER_INSTANCE_NAME.NAME(1 .. LENGTH));
end IS_EQUAL_INSTANCE_LIST_USER_NAME;

procedure UPDATE_INSTANCE_LIST(X : in OBJ_NAME;
                                Y : in CLASS_T;
                                Z : in MODE_T;
                                W : in USER_INSTANCE_NAME_PTR) is

    INSTANCE_REC : INSTANCE_REC_T;
begin
    INSTANCE_REC.INSTANCE_NAME := X;
    INSTANCE_REC.CLASS_NAME := Y;

```

```

    INSTANCE_REC.MODE_TYPE := Z;
    INSTANCE_REC.USER_INSTANCE_NAME := W;
    INSTANCE_LIST_T.CONSTRUCT(INSTANCE_REC, INSTANCE_LIST);
end UPDATE_INSTANCE_LIST;

procedure GET_MODE(X : in OBJ_NAME; Y : out MODE_T) is
    INSTANCE_RECORD : INSTANCE_REC_T;
begin
    SEARCH_INSTANCE_LIST.FIND(X, INSTANCE_RECORD, INSTANCE_LIST);
    Y := INSTANCE_RECORD.MODE_TYPE;
end GET_MODE;

procedure GET_OBJ_NAME(USER_NAME : in USER_INSTANCE_NAME_PTR;
                        SYS_NAME : out OBJ_NAME) is

    INSTANCE_RECORD : INSTANCE_REC_T;
begin
    SEARCH_INSTANCE_LIST_USER_NAME.FIND(USER_NAME, INSTANCE_RECORD,
    INSTANCE_LIST);
    SYS_NAME := INSTANCE_RECORD.INSTANCE_NAME;
end GET_OBJ_NAME;
end GLOBAL;

```

PACKAGE BUFFER_CL SPECIFICATION:

```

with GLOBAL, LIST_BASIC, TEXT_IO, OBJECT_CL, LIST_SEARCH;
use GLOBAL;
package BUFFER_CL is

    type FRUIT is (APPLES, ORANGES, PEARS, KIWI);
    package FRUIT_LIST_T is new LIST_BASIC(FRUIT);

    type IVAR_REC_T is record
        FRUIT_LIST : FRUIT_LIST_T.LIST;
        INSTANCE_NAME : GLOBAL.OBJ_NAME;
    end record;

    type IVAR_PTR_T is access IVAR_REC_T;

    type IVAR_PTR_REC_T is record
        INSTANCE_NAME : GLOBAL.OBJ_NAME;
        IVAR_PTR : IVAR_PTR_T;
    end record;

    procedure LISTLENGTH_FRUIT_LIST is new OBJECT_CL.LISTLENGTH(
        LIST=>BUFFER_CL.FRUIT_LIST_T.LIST,

        LENGTH_OF=>BUFFER_CL.FRUIT_LIST_T.LENGTH_OF);

```

```

type MASK is array(1 .. 1) of BOOLEAN;

procedure MAKE_NEW(USER_NAME      : in out USER_INSTANCE_NAME_PTR;
                   USER_NAME_MODE : in GLOBAL.ARG_MODE_T;
                   MODE_TYPE       : in GLOBAL.MODE_T;
                   VALUES         : in IVAR_REC_T;
                   VALUES_FLAG    : in MASK);

procedure RETRIEVE(ENVIRONMENT : in ENV_RECORD_T;
                  DATA        : out FRUIT;
                  SUCCESS      : in out BOOLEAN);

procedure STORE(ENVIRONMENT : in ENV_RECORD_T;
                X            : in FRUIT;
                SUCCESS      : in out BOOLEAN);

procedure SIZE(ENVIRONMENT : in ENV_RECORD_T;
               S           : out INTEGER;
               SUCCESS      : in out BOOLEAN);

end BUFFER_CL;

```

PACKAGE BOUNDED_BUFFER_CL SPECIFICATION:

```

with GLOBAL, LIST_BASIC, TEXT_IO, OBJECT_CL, LIST_SEARCH, BUFFER_CL;
use GLOBAL;
package BOUNDED_BUFFER_CL is

  type IVAR_REC_T is record
    BOUND      : NATURAL;
    FRUIT_LIST : BUFFER_CL.FRUIT_LIST_T.LIST;
    INSTANCE_NAME : GLOBAL.OBJ_NAME;
  end record;

  type IVAR_PTR_T is access IVAR_REC_T;

  type IVAR_PTR_REC_T is record
    INSTANCE_NAME : GLOBAL.OBJ_NAME;
    IVAR_PTR      : IVAR_PTR_T;
  end record;

  task type ACTIVE_BOUNDED_BUFFER is
    entry INIT(X : in GLOBAL.OBJ_NAME);
    entry RETRIEVE(ENVIRONMENT : in ENV_RECORD_T;
                  DATA        : out BUFFER_CL.FRUIT;
                  SUCCESS      : in out BOOLEAN);
    entry STORE(ENVIRONMENT : in ENV_RECORD_T;
                X            : in BUFFER_CL.FRUIT;

```

```

                SUCCESS      : in out BOOLEAN);
end ACTIVE_BOUNDED_BUFFER;

type TASK_ID is access ACTIVE_BOUNDED_BUFFER;
type MASK     is array(1 .. 2) of BOOLEAN;

procedure MAKE_NEW(USER_NAME      : in out USER_INSTANCE_NAME_PTR;
                   USER_NAME_MODE : in GLOBAL.ARG_MODE_T;
                   MODE_TYPE       : in GLOBAL.MODE_T;
                   VALUES          : in IVAR_REC_T;
                   VALUES_FLAG     : in MASK);

procedure GET_TASK_PTR(CUR_OBJECT : in GLOBAL.OBJ_NAME;
                      TASK_PTR    : out TASK_ID);

procedure THEVALUEOF_MYBUFFER(
  ENVIRONMENT : in ENV_RECORD_T;
  DATA       : out BUFFER_CL.FRUIT_LIST_T.LIST;
  SUCCESS     : in out BOOLEAN);

procedure MAKETHEVALUEOF_MYBUFFER(
  ENVIRONMENT : in ENV_RECORD_T;
  DATA       : in BUFFER_CL.FRUIT_LIST_T.LIST;
  SUCCESS     : in out BOOLEAN);

procedure RETRIEVE(ENVIRONMENT : in ENV_RECORD_T;
                  DATA       : out BUFFER_CL.FRUIT;
                  SUCCESS     : in out BOOLEAN);

procedure STORE(ENVIRONMENT : in ENV_RECORD_T;
               DATA       : in BUFFER_CL.FRUIT;
               SUCCESS     : in out BOOLEAN);

end BOUNDED_BUFFER_CL;

```

PACKAGE BUFFER_CL BODY:

```

with BOUNDED_BUFFER_CL;
package body BUFFER_CL is

```

```

  package IVAR_PTR_REC_LIST_T is new LIST_BASIC(IVAR_PTR_REC_T);

  function IS_EQUAL_IVAR_PTR_REC_LIST(LEFT  : in OBJ_NAME;
                                       RIGHT : in IVAR_PTR_REC_T)
    return BOOLEAN;

```

```

package SEARCH_IVAR_PTR_REC_LIST is new LIST_SEARCH(
  KEY => OBJ_NAME,
  ITEM => IVAR_PTR_REC_T, LIST => IVAR_PTR_REC_LIST_T.LIST,
  IS_EQUAL => IS_EQUAL_IVAR_PTR_REC_LIST,
  IS_NULL => IVAR_PTR_REC_LIST_T.IS_NULL,
  HEAD_OF => IVAR_PTR_REC_LIST_T.HEAD_OF,
  TAIL_OF => IVAR_PTR_REC_LIST_T.TAIL_OF,
  SWAP_TAIL => IVAR_PTR_REC_LIST_T.SWAP_TAIL);

IVAR_PTR_REC_LIST : IVAR_PTR_REC_LIST_T.LIST;

function IS_EQUAL_IVAR_PTR_REC_LIST(LEFT : in OBJ_NAME;
                                     RIGHT : in IVAR_PTR_REC_T)
  return BOOLEAN is
begin
  return (LEFT = RIGHT.INSTANCE_NAME);
end IS_EQUAL_IVAR_PTR_REC_LIST;
package IVAR_LIST_MUTEX is new LIST_MUTEX(
  ITEM => IVAR_PTR_REC_T,
  LIST => IVAR_PTR_REC_LIST_T.LIST,
  CONSTRUCT => IVAR_PTR_REC_LIST_T.CONSTRUCT);

procedure MAKE_NEW(USER_NAME          : in out USER_INSTANCE_NAME_PTR;
                   USER_NAME_MODE    : in GLOBAL.ARG_MODE_T;
                   MODE_TYPE           : in GLOBAL.MODE_T;
                   VALUES              : in IVAR_REC_T;
                   VALUES_FLAG .      : in MASK) is

  CLASS_NAME      : GLOBAL.CLASS_T := GLOBAL.BUFFER;
  Y                : GLOBAL.OBJ_NAME;
  IVAR_REC         : IVAR_REC_T;
  IVAR_PTR_REC     : IVAR_PTR_REC_T;

begin
  OBJ_NAME_CONTROL.GET_INSTANCE_ID(USER_NAME, USER_NAME_MODE,
    MODE_TYPE, CLASS_NAME, Y);

  --do MLog compile time initializations of ivars
  IVAR_REC.INSTANCE_NAME := Y;
  --do MLog create time initialization of ivars
  if VALUES_FLAG(1) then
    FRUIT_LIST_T.CLEAR(IVAR_REC.FRUIT_LIST);
    FRUIT_LIST_T.COPY(VALUES.FRUIT_LIST, IVAR_REC.FRUIT_LIST);
  end if;
  IVAR_PTR_REC.IVAR_PTR := new IVAR_REC_T;
  IVAR_PTR_REC.IVAR_PTR.all := IVAR_REC;
  IVAR_LIST_MUTEX.MUTEX_TASK.UPDATE_LIST(IVAR_PTR_REC,
    IVAR_PTR_REC_LIST);

end MAKE_NEW;

```

```

procedure THEVALUEOF_MYBUFFER(ENVIRONMENT : in ENV_RECORD_T;
                               DATA       : out FRUIT_LIST_T.LIST;
                               SUCCESS      : in out BOOLEAN) is
    IVAR_PTR_REC : IVAR_PTR_REC_T;

begin
    SUCCESS := TRUE;
    SEARCH_IVAR_PTR_REC_LIST.FIND(ENVIRONMENT.CUR_OBJECT,
    IVAR_PTR_REC, IVAR_PTR_REC_LIST);
    DATA := IVAR_PTR_REC.IVAR_PTR.FRUIT_LIST;
end THEVALUEOF_MYBUFFER;

procedure MAKETHEVALUEOF_MYBUFFER(
    ENVIRONMENT : in ENV_RECORD_T;
    DATA       : in FRUIT_LIST_T.LIST;
    SUCCESS      : in out BOOLEAN) is
    IVAR_PTR_REC : IVAR_PTR_REC_T;

begin
    SUCCESS := TRUE;
    SEARCH_IVAR_PTR_REC_LIST.FIND(ENVIRONMENT.CUR_OBJECT,
    IVAR_PTR_REC, IVAR_PTR_REC_LIST);
    IVAR_PTR_REC.IVAR_PTR.FRUIT_LIST := DATA;
end MAKETHEVALUEOF_MYBUFFER;

procedure RETRIEVE(ENVIRONMENT : in ENV_RECORD_T;
                   DATA       : out FRUIT;
                   SUCCESS      : in out BOOLEAN) is
    B, B2 : FRUIT_LIST_T.LIST;
    S      : INTEGER;

begin
    SUCCESS := TRUE;
    SIZE(ENVIRONMENT, S, SUCCESS);
    if SUCCESS then
        if S > 0 then
            SUCCESS := TRUE;
        else
            SUCCESS := FALSE;
        end if;
    end if;
    case ENVIRONMENT.CUR_CLASS is
        when GLOBAL.BUFFER =>
            THEVALUEOF_MYBUFFER(ENVIRONMENT, B, SUCCESS);
        when GLOBAL.BOUNDED_BUFFER =>
            BOUNDED_BUFFER_CL.THEVALUEOF_MYBUFFER(ENVIRONMENT,
            B, SUCCESS);
        when others =>
            null;
    end case;
end RETRIEVE;

```



```

end case;
if FRUIT_LIST_T.LENGTH_OF(B) > 0 then
    DATA := FRUIT_LIST_T.HEAD_OF(B);
    B2 := FRUIT_LIST_T.TAIL_OF(B);
    SUCCESS := TRUE;
else
    SUCCESS := FALSE;
end if;

if SUCCESS then
    case ENVIRONMENT.CUR_CLASS is
        when GLOBAL.BUFFER =>
            MAKETHEVALUEOF_MYBUFFER(ENVIRONMENT, B2, SUCCESS);
        when GLOBAL.BOUNDED_BUFFER =>
            BOUNDED_BUFFER_CL.MAKETHEVALUEOF_MYBUFFER(
                ENVIRONMENT, B2, SUCCESS);
        when others =>
            null;
    end case;
end if;

end RETRIEVE;

procedure STORE(ENVIRONMENT : in ENV_RECORD_T;
                X             : in FRUIT;
                SUCCESS       : in out BOOLEAN) is

    B : FRUIT_LIST_T.LIST;
begin
    SUCCESS := TRUE;
    case ENVIRONMENT.CUR_CLASS is
        when GLOBAL.BUFFER =>
            THEVALUEOF_MYBUFFER(ENVIRONMENT, B, SUCCESS);
        when GLOBAL.BOUNDED_BUFFER =>
            BOUNDED_BUFFER_CL.THEVALUEOF_MYBUFFER(
                ENVIRONMENT, B, SUCCESS);
        when others =>
            null;
    end case;
    if SUCCESS then
        FRUIT_LIST_T.CONSTRUCT(X, B);
        case ENVIRONMENT.CUR_CLASS is
            when GLOBAL.BUFFER =>
                MAKETHEVALUEOF_MYBUFFER(ENVIRONMENT, B, SUCCESS);
            when GLOBAL.BOUNDED_BUFFER =>
                BOUNDED_BUFFER_CL.MAKETHEVALUEOF_MYBUFFER(
                    ENVIRONMENT, B, SUCCESS);
            when others =>
                null;
        end case;
    end if;
end STORE;

```

```

        end case;
    end if;
end STORE;

procedure SIZE(ENVIRONMENT : in ENV_RECORD_T;
               S           : out INTEGER;
               SUCCESS      : in out BOOLEAN) is
    B : FRUIT_LIST_T.LIST;
begin
    SUCCESS := TRUE;
    case ENVIRONMENT.CUR_CLASS is
        when GLOBAL.BUFFER =>
            THEVALUEOF_MYBUFFER(ENVIRONMENT, B, SUCCESS);
        when GLOBAL.BOUNDED_BUFFER =>
            BOUNDED_BUFFER_CL.THEVALUEOF_MYBUFFER(
                ENVIRONMENT, B, SUCCESS);
        when others =>
            null;
    end case;
    LISTLENGTH_FRUIT_LIST(B, S);
end SIZE;

end BUFFER_CL;

PACKAGE BOUNDED_BUFFER_CL BODY:

package body BOUNDED_BUFFER_CL is

    package IVAR_PTR_REC_LIST_T is new LIST_BASIC(IVAR_PTR_REC_T);

    function IS_EQUAL_IVAR_PTR_REC_LIST(LEFT  : in OBJ_NAME;
                                         RIGHT : in IVAR_PTR_REC_T)
        return BOOLEAN;

    package SEARCH_IVAR_PTR_REC_LIST is new LIST_SEARCH(
        KEY => OBJ_NAME,
        ITEM => IVAR_PTR_REC_T, LIST => IVAR_PTR_REC_LIST_T.LIST,
        IS_EQUAL => IS_EQUAL_IVAR_PTR_REC_LIST,
        IS_NULL => IVAR_PTR_REC_LIST_T.IS_NULL,
        HEAD_OF => IVAR_PTR_REC_LIST_T.HEAD_OF,
        TAIL_OF => IVAR_PTR_REC_LIST_T.TAIL_OF,
        SWAP_TAIL => IVAR_PTR_REC_LIST_T.SWAP_TAIL);
    -- define a list of pointers to ivar records; each record is
    -- equivalent to an object
    IVAR_PTR_REC_LIST : IVAR_PTR_REC_LIST_T.LIST;

    type TASK_NAME_T is record
        PTR : TASK_ID;
        INSTANCE_NAME : GLOBAL.OBJ_NAME;
    end record;

```

end record;

```
package ACTIVE_BOUNDED_BUF_LIST is new LIST_BASIC(TASK_NAME_T);
ACTIVE_LIST : ACTIVE_BOUNDED_BUF_LIST.LIST;
function IS_EQUAL_ACTIVE_LIST(LEFT  : in OBJ_NAME;
                              RIGHT : in TASK_NAME_T)
    return BOOLEAN;
```

```
package SEARCH_ACTIVE_LIST is new LIST_SEARCH(KEY => OBJ_NAME,
ITEM => TASK_NAME_T, LIST => ACTIVE_BOUNDED_BUF_LIST.LIST,
IS_EQUAL => IS_EQUAL_ACTIVE_LIST,
IS_NULL => ACTIVE_BOUNDED_BUF_LIST.IS_NULL,
HEAD_OF => ACTIVE_BOUNDED_BUF_LIST.HEAD_OF,
TAIL_OF => ACTIVE_BOUNDED_BUF_LIST.TAIL_OF,
SWAP_TAIL => ACTIVE_BOUNDED_BUF_LIST.SWAP_TAIL);
```

```
function IS_EQUAL_IVAR_PTR_REC_LIST(LEFT  : in OBJ_NAME;
                                     RIGHT : in IVAR_PTR_REC_T)
    return BOOLEAN is
```

```
begin
    return (LEFT = RIGHT.INSTANCE_NAME);
end IS_EQUAL_IVAR_PTR_REC_LIST;
```

```
function IS_EQUAL_ACTIVE_LIST(LEFT  : in OBJ_NAME;
                              RIGHT : in TASK_NAME_T)
    return BOOLEAN is
```

```
begin
    return (LEFT = RIGHT.INSTANCE_NAME);
end IS_EQUAL_ACTIVE_LIST;
```

```
package ACTIVE_LIST_MUTEX is new LIST_MUTEX(
    ITEM => TASK_NAME_T,
    LIST => ACTIVE_BOUNDED_BUF_LIST.LIST,
    CONSTRUCT => ACTIVE_BOUNDED_BUF_LIST.CONSTRUCT);
```

```
package IVAR_LIST_MUTEX is new LIST_MUTEX(ITEM => IVAR_PTR_REC_T,
    LIST => IVAR_PTR_REC_LIST_T.LIST,
    CONSTRUCT => IVAR_PTR_REC_LIST_T.CONSTRUCT);
```

```
procedure MAKE_NEW(USER_NAME      : in out USER_INSTANCE_NAME_PTR;
                  USER_NAME_MODE : in GLOBAL.ARG_MODE_T;
                  MODE_TYPE       : in GLOBAL.MODE_T;
                  VALUES         : in IVAR_REC_T;
                  VALUES_FLAG    : in MASK) is
```

```
TASK_NAME      : TASK_NAME_T;
CLASS_NAME     : GLOBAL.CLASS_T := GLOBAL.BOUNDED_BUFFER;
Y              : GLOBAL.OBJ_NAME;
IVAR_REC       : IVAR_REC_T;
IVAR_PTR_REC   : IVAR_PTR_REC_T;
```

```

begin
  OBJ_NAME_CONTROL.GET_INSTANCE_ID(USER_NAME, USER_NAME_MODE,
    MODE_TYPE, CLASS_NAME, Y);

  -- do MLog compile time initialization of ivars
  IVAR_REC.INSTANCE_NAME := Y;
  -- do MLog create time initialization of ivars
  if VALUES_FLAG(1) then
    IVAR_REC.BOUND := VALUES.BOUND;
  end if;
  if VALUES_FLAG(2) then
    BUFFER_CL.FRUIT_LIST_T.CLEAR(IVAR_REC.FRUIT_LIST);
    BUFFER_CL.FRUIT_LIST_T.COPY(VALUES.FRUIT_LIST,
      IVAR_REC.FRUIT_LIST);
  end if;

  IVAR_PTR_REC.IVAR_PTR := new IVAR_REC_T;
  IVAR_PTR_REC.IVAR_PTR.all := IVAR_REC;
  IVAR_LIST_MUTEX.MUTEX_TASK.UPDATE_LIST(
    IVAR_PTR_REC, IVAR_PTR_REC_LIST);

  -- creates a task, associates it with the
  -- object name, puts record on list of
  -- active bounded buffer objects, and initializes
  -- the task;
  if MODE_TYPE = GLOBAL.ACTIVE then
    TASK_NAME.PTR := new ACTIVE_BOUNDED_BUFFER;
    TASK_NAME.INSTANCE_NAME := Y;
    ACTIVE_LIST_MUTEX.MUTEX_TASK.UPDATE_LIST(
      TASK_NAME, ACTIVE_LIST);
    TASK_NAME.PTR.INIT(Y);
  end if;
end MAKE_NEW;

procedure GET_TASK_PTR(CUR_OBJECT : in GLOBAL.OBJ_NAME;
  TASK_PTR : out TASK_ID) is

  TEMP_LIST : ACTIVE_BOUNDED_BUF_LIST.LIST := ACTIVE_LIST;
  TEMP_NAME_REC : TASK_NAME_T;
  DONE : BOOLEAN := FALSE;

begin
  SEARCH_ACTIVE_LIST.FIND(CUR_OBJECT, TEMP_NAME_REC, ACTIVE_LIST);
  TASK_PTR := TEMP_NAME_REC.PTR;

end GET_TASK_PTR;

procedure THEVALUEOF_MYBUFFER(
  ENVIRONMENT : in ENV_RECORD_T;

```

```

DATA      : out BUFFER_CL.FRUIT_LIST_T.LIST;
SUCCESS   : in out BOOLEAN) is

    IVAR_PTR_RECORD : IVAR_PTR_REC_T;
begin
    SUCCESS := TRUE;
    SEARCH_IVAR_PTR_REC_LIST.FIND(ENVIRONMENT.CUR_OBJECT,
        IVAR_PTR_RECORD, IVAR_PTR_REC_LIST);
    DATA := IVAR_PTR_RECORD.IVAR_PTR.FRUIT_LIST;
end THEVALUEOF_MYBUFFER;

procedure MAKETHEVALUEOF_MYBUFFER(
    ENVIRONMENT : in ENV_RECORD_T;
    DATA       : in BUFFER_CL.FRUIT_LIST_T.LIST;
    SUCCESS     : in out BOOLEAN) is

    IVAR_PTR_RECORD : IVAR_PTR_REC_T;
begin
    SUCCESS := TRUE;
    SEARCH_IVAR_PTR_REC_LIST.FIND(ENVIRONMENT.CUR_OBJECT,
        IVAR_PTR_RECORD, IVAR_PTR_REC_LIST);
    IVAR_PTR_RECORD.IVAR_PTR.FRUIT_LIST := DATA;
end MAKETHEVALUEOF_MYBUFFER;

procedure THEVALUEOF_BOUND(ENVIRONMENT : in ENV_RECORD_T;
                           MAX          : out NATURAL;
                           SUCCESS     : in out BOOLEAN) is

    IVAR_PTR_RECORD : IVAR_PTR_REC_T;
begin
    SUCCESS := TRUE;
    SEARCH_IVAR_PTR_REC_LIST.FIND(ENVIRONMENT.CUR_OBJECT,
        IVAR_PTR_RECORD, IVAR_PTR_REC_LIST);
    MAX := IVAR_PTR_RECORD.IVAR_PTR.BOUND;
end THEVALUEOF_BOUND;

procedure MAKETHEVALUEOF_BOUND(ENVIRONMENT : in ENV_RECORD_T;
                               MAX          : in NATURAL;
                               SUCCESS     : in out BOOLEAN) is

    IVAR_PTR_RECORD : IVAR_PTR_REC_T;
begin
    SUCCESS := TRUE;
    SEARCH_IVAR_PTR_REC_LIST.FIND(ENVIRONMENT.CUR_OBJECT,
        IVAR_PTR_RECORD, IVAR_PTR_REC_LIST);
    IVAR_PTR_RECORD.IVAR_PTR.BOUND := MAX;

```

3 OF/DE 3

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

```
end MAKETHEVALUEOF_BOUND;
```

```
procedure RETRIEVE(ENVIRONMENT : in ENV_RECORD_T;
                   DATA       : out BUFFER_CL.FRUIT;
                   SUCCESS      : in out BOOLEAN) is
```

```
begin
    BUFFER_CL.RETRIEVE(ENVIRONMENT, DATA, SUCCESS);
```

```
end RETRIEVE;
```

```
procedure STORE(ENVIRONMENT : in ENV_RECORD_T;
                DATA       : in BUFFER_CL.FRUIT;
                SUCCESS      : in out BOOLEAN) is
```

```
    MAX, S : NATURAL;
begin
    SUCCESS := TRUE;
    THEVALUEOF_BOUND(ENVIRONMENT, MAX, SUCCESS);
    if SUCCESS then
        BUFFER_CL.SIZE(ENVIRONMENT, S, SUCCESS);
    end if;
    if SUCCESS then
        if S < MAX then
            SUCCESS := TRUE;
        else
            SUCCESS := FALSE;
        end if;
    end if;
    if SUCCESS then
        BUFFER_CL.STORE(ENVIRONMENT, DATA, SUCCESS);
    end if;
end STORE;
```

```
task body ACTIVE_BOUNDED_BUFFER is
```

```
    ENVIRONMENT      : ENV_RECORD_T;
    SUCCESS           : BOOLEAN := TRUE;
    B, B2             : BUFFER_CL.FRUIT_LIST_T.LIST;
    RETRIEVE_GUARD, STORE_GUARD : BOOLEAN;
    MAX               : NATURAL;
    S                 : INTEGER;
```

```
begin
    accept INIT(X : in GLOBAL.OBJ_NAME) do
        ENVIRONMENT := (X, GLOBAL.BOUNDED_BUFFER, X);
        TEXT_IO.PUT_LINE("BOUNDED_BUFFER TASK INITIALIZED");
    end INIT;
loop
```

```

-- evaluate guard conditions for store entry
THEVALUEOF_BOUND(ENVIRONMENT, MAX, SUCCESS);
if SUCCESS then
    BUFFER_CL.SIZE(ENVIRONMENT, S, SUCCESS);
end if;
if SUCCESS then
    if S < MAX then
        SUCCESS := TRUE;
    else
        SUCCESS := FALSE;
    end if;
end if;
if SUCCESS then
    STORE_GUARD := TRUE;
else
    STORE_GUARD := FALSE;
end if;
-- evaluate guard conditions for retrieve entry
BUFFER_CL.SIZE(ENVIRONMENT, S, SUCCESS);
if SUCCESS then
    if S > 0 then
        SUCCESS := TRUE;
    else
        SUCCESS := FALSE;
    end if;
end if;
if SUCCESS then
    RETRIEVE_GUARD := TRUE;
else
    RETRIEVE_GUARD := FALSE;
end if;
select
    when RETRIEVE_GUARD =>
        accept RETRIEVE(ENVIRONMENT : in ENV_RECORD_T;
                        DATA       : out BUFFER_CL.FRUIT;
                        SUCCESS     : in out BOOLEAN) do

            SUCCESS := TRUE;
            THEVALUEOF_MYBUFFER(ENVIRONMENT, B, SUCCESS);
            if BUFFER_CL.FRUIT_LIST_T.LENGTH_OF(B) > 0 then
                DATA := BUFFER_CL.FRUIT_LIST_T.HEAD_OF(B);
                B2 := BUFFER_CL.FRUIT_LIST_T.TAIL_OF(B);
                SUCCESS := TRUE;
            else
                SUCCESS := FALSE;
            end if;

            if SUCCESS then
                MAKETHEVALUEOF_MYBUFFER(ENVIRONMENT, B2, SUCCESS);
            end if;

```



```

        end RETRIEVE;
    or
        when STORE_GUARD =>
            accept STORE(ENVIRONMENT : in ENV_RECORD_T;
                        X           : in BUFFER_CL.FRUIT;
                        SUCCESS      : in out BOOLEAN) do
                BUFFER_CL.STORE(ENVIRONMENT, X, SUCCESS);
            end STORE;
        end select;
    end loop;
end ACTIVE_BOUNDED_BUFFER;

end BOUNDED_BUFFER_CL;

```

PACKAGE PRODUCER_CL SPECIFICATION:

```

with GLOBAL, LIST_BASIC, TEXT_IO;
use GLOBAL;
package PRODUCER_CL is

```

```

    task type ACTIVE_PRODUCER is
        entry INIT(X : in GLOBAL.OBJ_NAME);
    end ACTIVE_PRODUCER;

```

```

    type TASK_ID is access ACTIVE_PRODUCER;

```

```

    procedure MAKE_NEW(USER_NAME      : in out USER_INSTANCE_NAME_PTR;
                      USER_NAME_MODE : in GLOBAL.ARG_MODE_T;
                      MODE_TYPE       : in GLOBAL.MODE_T);

```

```

end PRODUCER_CL;

```

PACKAGE PRODUCER_CL BODY:

```

with BUFFER_CL, BOUNDED_BUFFER_CL;
package body PRODUCER_CL is

```

```

    procedure PRODUCERLOOP(ENVIRONMENT : in ENV_RECORD_T;
                          X           : in USER_INSTANCE_NAME_PTR;
                          SUCCESS      : in out BOOLEAN);

```

```

    type TASK_NAME_T is record
        PTR : TASK_ID;
    end record;

```

```

    INSTANCE_NAME : GLOBAL.OBJ_NAME;
end record;

package ACTIVE_PROD_LIST is new LIST_BASIC(TASK_NAME_T);
ACTIVE_LIST : ACTIVE_PROD_LIST.LIST;

task body ACTIVE_PRODUCER is

    ENVIRONMENT      : ENV_RECORD_T;
    NEW_ENVIRONMENT  : ENV_RECORD_T;

    SUCCESS          : BOOLEAN          := TRUE;

    TEMP_REC         : BUFFER_CL.IVAR_REC_T;
    TEMP_FLAG        : BUFFER_CL.MASK;
    TEMP             : BUFFER_CL.FRUIT_LIST_T.LIST;
    ID               : USER_INSTANCE_NAME_PTR := new USER_NAME_REC(40);

begin
    accept INIT(X : in GLOBAL.OBJ_NAME) do
        ENVIRONMENT := (X, GLOBAL.PRODUCER, X);
        TEXT_IO.PUT_LINE("Producer TASK INITIALIZED");
    end INIT;

    TEMP_FLAG(1) := TRUE;
    BUFFER_CL.FRUIT_LIST_T.CONSTRUCT(BUFFER_CL.APPLES, TEMP);
    BUFFER_CL.FRUIT_LIST_T.CONSTRUCT(BUFFER_CL.ORANGES, TEMP);
    BUFFER_CL.FRUIT_LIST_T.CONSTRUCT(BUFFER_CL.PEARS, TEMP);
    BUFFER_CL.FRUIT_LIST_T.CONSTRUCT(BUFFER_CL.KIWI, TEMP);
    TEMP_REC.FRUIT_LIST := TEMP;
    BUFFER_CL.MAKE_NEW(ID, MODE_OUT, GLOBAL.PASSIVE, TEMP_REC,
        TEMP_FLAG);
    PRODUCERLOOP(ENVIRONMENT, ID, SUCCESS);
end ACTIVE_PRODUCER;

procedure PRODUCERLOOP(ENVIRONMENT : in ENV_RECORD_T;
                        X            : in USER_INSTANCE_NAME_PTR;
                        SUCCESS      : in out BOOLEAN) is

    package FRUIT_IO is new TEXT_IO.ENumeration_IO(BUFFER_CL.FRUIT);
    XNEW : GLOBAL.OBJ_NAME;
    D : BUFFER_CL.FRUIT;
    Z : GLOBAL.OBJ_NAME;
    Z_MODE : GLOBAL.MODE_T;
    PTR_BB : BOUNDED_BUFFER_CL."ASK_ID";
    USER_NAME : USER_INSTANCE_NAME_PTR := new USER_NAME_REC(40);
    NEW_ENVIRONMENT : ENV_RECORD_T;

begin
    SUCCESS := TRUE;

```

```

-- translate X(buffer)-retrieve(D), since buffer class has
-- no active objects, the mode of X is known to be passive.
if X.NAME(1 .. X.CUR_LENGTH) = "super" then
  TEXT_IO.PUT_LINE(
    "ERROR-PRODUCERLOOP NOT DEFINED IN SUPERCLASSES");
else
  XNEW := GLOBAL.OBJ_NAME'VALUE(X.NAME(1 .. X.CUR_LENGTH));
  NEW_ENVIRONMENT := (XNEW, GLOBAL.BUFFER,
    ENVIRONMENT.CUR_PROCESS);
  BUFFER_CL.RETRIEVE(NEW_ENVIRONMENT, D, SUCCESS);

  if SUCCESS then
    TEXT_IO.PUT("sending ");
    FRUIT_IO.PUT(D);
    TEXT_IO.NEW_LINE;
    USER_NAME.NAME(1 .. 6) := "common";
    USER_NAME.CUR_LENGTH := 6;
    GLOBAL.GET_OBJ_NAME(USER_NAME, Z);
    GLOBAL.GET_MODE(Z, Z_MODE);
    if (Z_MODE = GLOBAL.PASSIVE) or
      (Z = ENVIRONMENT.CUR_PROCESS) then
      NEW_ENVIRONMENT := (Z, GLOBAL.BOUNDED_BUFFER,
        ENVIRONMENT.CUR_PROCESS);
      BOUNDED_BUFFER_CL.STORE(NEW_ENVIRONMENT, D, SUCCESS);
    else
      BOUNDED_BUFFER_CL.GET_TASK_PTR(Z, PTR_BB);
      NEW_ENVIRONMENT := (Z, GLOBAL.BOUNDED_BUFFER, Z);
      PTR_BB.STORE(NEW_ENVIRONMENT, D, SUCCESS);
    end if;
  end if;
  if SUCCESS then
    PRODUCERLOOP(ENVIRONMENT, X, SUCCESS);
  end if;
end if;
if SUCCESS then
  return;
else
  SUCCESS := TRUE;
end if;
end PRODUCERLOOP;

package ACTIVE_LIST_MUTEX is new LIST_MUTEX(
  ITEM => TASK_NAME_T,
  LIST => ACTIVE_PROD_LIST.LIST,
  CONSTRUCT => ACTIVE_PROD_LIST.CONSTRUCT);

procedure MAKE_NEW(USER_NAME : in out USER_INSTANCE_NAME_PTR;
  USER_NAME_MODE : in GLOBAL.ARG_MODE_T;
  MODE_TYPE : in GLOBAL.MODE_T) is

```

```

TASK_NAME : TASK_NAME_T;
CLASS_NAME : GLOBAL.CLASS_T := GLOBAL.PRODUCER;
Y          : GLOBAL.OBJ_NAME;

begin
  OBJ_NAME_CONTROL.GET_INSTANCE_ID(USER_NAME, USER_NAME_MODE,
    MODE_TYPE, CLASS_NAME, Y);

  --makes an active task & associates it with instance id
  --& puts on list of active objects
  if MODE_TYPE = GLOBAL.ACTIVE then
    TASK_NAME.PTR := new PRODUCER_CL.ACTIVE_PRODUCER;
    TASK_NAME.INSTANCE_NAME := Y;
    ACTIVE_LIST_MUTEX.MUTEX_TASK.UPDATE_LIST(
      TASK_NAME, ACTIVE_LIST);
    TASK_NAME.PTR.INIT(Y);
  end if;
end MAKE_NEW;
end PRODUCER_CL;

```

PACKAGE CONSUMER_CL SPECIFICATION:

```

with GLOBAL, BUFFER_CL, BOUNDED_BUFFER_CL, LIST_BASIC, TEXT_IO;
use GLOBAL;
package CONSUMER_CL is

  task type ACTIVE_CONSUMER is
    entry INIT(X : in GLOBAL.OBJ_NAME);
  end ACTIVE_CONSUMER;

  type TASK_ID is access ACTIVE_CONSUMER;

  procedure MAKE_NEW(USER_NAME      : in out USER_INSTANCE_NAME_PTR;
                     USER_NAME_MODE : in GLOBAL.ARG_MODE_T;
                     MODE_TYPE      : in GLOBAL.MODE_T);

end CONSUMER_CL;

```

PACKAGE CONSUMER_CL BODY:

```

package body CONSUMER_CL is

  type TASK_NAME_T is record
    PTR          : TASK_ID;
    INSTANCE_NAME : GLOBAL.OBJ_NAME;
  end record;

```

```

package ACTIVE_CONS_LIST is new LIST_BASIC(TASK_NAME_T);
ACTIVE_LIST : ACTIVE_CONS_LIST.LIST;

procedure CONSUMERLOOP(ENVIRONMENT : in ENV_RECORD_T;
                        X             : in USER_INSTANCE_NAME_PTR;
                        SUCCESS       : in out BOOLEAN);

task body ACTIVE_CONSUMER is

    ENVIRONMENT      : ENV_RECORD_T;
    NEW_ENVIRONMENT  : ENV_RECORD_T;
    SUCCESS          : BOOLEAN           := TRUE;

    TEMP_REC         : BUFFER_CL.IVAR_REC_T;
    TEMP_FLAG        : BUFFER_CL.MASK;
    TEMP             : BUFFER_CL.FRUIT_LIST_T.LIST;
    ID               : USER_INSTANCE_NAME_PTR := new USER_NAME_REC(40);

begin
    accept INIT(X : in GLOBAL.OBJ_NAME) do
        ENVIRONMENT := (X, GLOBAL.CONSUMER, X);
        TEXT_IO.PUT_LINE("Consumer TASK INITIALIZED");
    end INIT;

    TEMP_FLAG(1) := FALSE;
    BUFFER_CL.MAKE_NEW(ID, MODE_OUT, GLOBAL.PASSIVE,
                      TEMP_REC, TEMP_FLAG);
    CONSUMERLOOP(ENVIRONMENT, ID, SUCCESS);
end ACTIVE_CONSUMER;

procedure CONSUMERLOOP(ENVIRONMENT : in ENV_RECORD_T;
                        X             : in USER_INSTANCE_NAME_PTR;
                        SUCCESS       : in out BOOLEAN) is
    package INT_IO is new TEXT_IO.INTEGER_IO(INTEGER);
    package FRUIT_IO is new TEXT_IO.ENUMERATION_IO(BUFFER_CL.FRUIT);
    XNEW : GLOBAL.OBJ_NAME;
    D : BUFFER_CL.FRUIT;
    S : INTEGER;
    Z : GLOBAL.OBJ_NAME;
    Z_MODE : GLOBAL.MODE_T;
    PTR_BB : BOUNDED_BUFFER_CL.TASK_ID;
    USER_NAME : USER_INSTANCE_NAME_PTR := new USER_NAME_REC(40);
    NEW_ENVIRONMENT : ENV_RECORD_T;

begin
    SUCCESS := TRUE;
    TEXT_IO.PUT_LINE("waiting to recieve");
    USER_NAME.NAME(1 .. 6) := "common";
    USER_NAME.CUR_LENGTH := 6;
    GLOBAL.GET_OBJ_NAME(USER_NAME, Z);

```

```

GLOBAL.GET_MODE(Z, Z_MODE);
if (Z_MODE = GLOBAL.PASSIVE) or (Z = ENVIRONMENT.CUR_PROCESS) then
    NEW_ENVIRONMENT := (Z, GLOBAL.BOUNDED_BUFFER,
                        ENVIRONMENT.CUR_PROCESS);
    BOUNDED_BUFFER_CL.RETRIEVE(NEW_ENVIRONMENT, D, SUCCESS);
else
    BOUNDED_BUFFER_CL.GET_TASK_PTR(Z, PTR_BB);
    NEW_ENVIRONMENT := (Z, GLOBAL.BOUNDED_BUFFER, Z);
    PTR_BB.RETRIEVE(NEW_ENVIRONMENT, D, SUCCESS);
end if;
if SUCCESS then
    -- translate X(buffer)-store(D), since the buffer class has no
    -- active objects, the mode of X is passive.
    if X.NAME(1 .. X.CUR_LENGTH) = "super" then
        TEXT_IO.PUT_LINE(
            "ERROR- STORE/1 NOT DEFINED IN SUPERCLASSES");
        SUCCESS := FALSE;
    else
        XNEW := GLOBAL.OBJ_NAME'VALUE(X.NAME(1 .. X.CUR_LENGTH));
        NEW_ENVIRONMENT := (XNEW, GLOBAL.BUFFER,
                            ENVIRONMENT.CUR_PROCESS);
        BUFFER_CL.STORE(NEW_ENVIRONMENT, D, SUCCESS);
    end if;
end if;
if SUCCESS then
    -- translate X(buffer)-size(S), since the buffer class has
    -- no active objects, the mode of X is passive.
    if X.NAME(1 .. X.CUR_LENGTH) = "super" then
        TEXT_IO.PUT_LINE(
            "ERROR- SIZE/1 NOT DEFINED IN SUPERCLASSES");
        SUCCESS := FALSE;
    else
        NEW_ENVIRONMENT := (XNEW, GLOBAL.BUFFER,
                            ENVIRONMENT.CUR_PROCESS);
        BUFFER_CL.SIZE(NEW_ENVIRONMENT, S, SUCCESS);
    end if;

    if SUCCESS then
        FRUIT_IO.PUT(D);
        TEXT_IO.PUT(" recieved ");
        TEXT_IO.NEW_LINE;
        INT_IO.PUT(S);
        TEXT_IO.PUT(" elements recieved.");
        TEXT_IO.NEW_LINE;
        CONSUMERLOOP(ENVIRONMENT, X, SUCCESS);
    end if;
end if;
end CONSUMERLOOP;

package ACTIVE_LIST_MUTEX is new LIST_MUTEX(

```

```

ITEM => TASK_NAME_T,
LIST => ACTIVE_CONS_LIST.LIST,
CONSTRUCT => ACTIVE_CONS_LIST.CONSTRUCT);

procedure MAKE_NEW(USER_NAME      : in out USER_INSTANCE_NAME_PTR;
                  USER_NAME_MODE : in GLOBAL.ARG_MODE_T;
                  MODE_TYPE       : in GLOBAL.MODE_T) is

    TASK_NAME : TASK_NAME_T;
    CLASS_NAME : GLOBAL.CLASS_T := GLOBAL.CONSUMER;
    Y          : GLOBAL.OBJ_NAME;

begin
    OBJ_NAME_CONTROL.GET_INSTANCE_ID(USER_NAME, USER_NAME_MODE,
    MODE_TYPE, CLASS_NAME, Y);

    --makes an active task & associates it with instance id
    -- & puts on list of active objects
    if MODE_TYPE = GLOBAL.ACTIVE then
        TASK_NAME.PTR := new CONSUMER_CL.ACTIVE_CONSUMER;
        TASK_NAME.INSTANCE_NAME := Y;
        ACTIVE_LIST_MUTEX.MUTEX_TASK.UPDATE_LIST(
            TASK_NAME, ACTIVE_LIST);
        TASK_NAME.PTR.INIT(Y);
    end if;
end MAKE_NEW;
end CONSUMER_CL;

```

PACKAGE MAIN_CL SPECIFICATION:

```

with GLOBAL, BOUNDED_BUFFER_CL, BUFFER_CL, PRODUCER_CL, CONSUMER_CL,
LIST_BASIC, TEXT_IO;
use GLOBAL;
package MAIN_CL is

    task type ACTIVE_MAIN is
        entry INIT(X : in GLOBAL.OBJ_NAME);
    end ACTIVE_MAIN;

    type TASK_ID is access ACTIVE_MAIN;

    procedure MAKE_NEW(USER_NAME      : in out USER_INSTANCE_NAME_PTR;
                      USER_NAME_MODE : in GLOBAL.ARG_MODE_T;
                      MODE_TYPE       : in GLOBAL.MODE_T);

end MAIN_CL;

```

PACKAGE MAIN_CL BODY:

```
package body MAIN_CL is
```

```
  type TASK_NAME_T is record
    PTR          : TASK_ID;
    INSTANCE_NAME : GLOBAL.OBJ_NAME;
  end record;
```

```
  package ACTIVE_MAIN_LIST is new LIST_BASIC(TASK_NAME_T);
  ACTIVE_LIST : ACTIVE_MAIN_LIST.LIST;
```

```
  task body ACTIVE_MAIN is
```

```
    ENVIRONMENT : ENV_RECORD_T;
    TEMP        : BOUNDED_BUFFER_CL.IVAR_REC_T;
    TEMP_FLAG   : BOUNDED_BUFFER_CL.MASK;
    ID          : USER_INSTANCE_NAME_PTR := new USER_NAME_REC(40);
```

```
  begin
```

```
    accept INIT(X : in GLOBAL.OBJ_NAME) do
      ENVIRONMENT := (X, GLOBAL.MAIN, X);
      TEXT_IO.PUT_LINE("Main TASK INITIALIZED");
    end INIT;
    ID.NAME(1 .. 6) := "common";
    ID.CUR_LENGTH := 6;
    TEMP_FLAG(1) := TRUE;
    TEMP_FLAG(2) := FALSE;
    TEMP.BOUND := 5;
    BOUNDED_BUFFER_CL.MAKE_NEW(ID, MODE_IN, ACTIVE, TEMP, TEMP_FLAG);
```

```
    ID := new USER_NAME_REC(40);
    PRODUCER_CL.MAKE_NEW(ID, MODE_OUT, ACTIVE);
    ID := new USER_NAME_REC(40);
    CONSUMER_CL.MAKE_NEW(ID, MODE_OUT, ACTIVE);
    TEXT_IO.PUT_LINE("All Tasks Initialized");
  end ACTIVE_MAIN;
```

```
  package ACTIVE_LIST_MUTEX is new LIST_MUTEX(
    ITEM => TASK_NAME_T,
    LIST => ACTIVE_MAIN_LIST.LIST,
    CONSTRUCT => ACTIVE_MAIN_LIST.CONSTRUCT);
```

```
  procedure MAKE_NEW(USER_NAME      : in out USER_INSTANCE_NAME_PTR;
                     USER_NAME_MODE : in GLOBAL.ARG_MODE_T;
                     MODE_TYPE      : in GLOBAL.MODE_T) is
```

```
    TASK_NAME : TASK_NAME_T;
    CLASS_NAME : GLOBAL.CLASS_T := GLOBAL.MAIN;
    Y          : GLOBAL.OBJ_NAME;
```



```
begin
  OBJ_NAME_CONTROL.GET_INSTANCE_ID(USER_NAME, USER_NAME_MODE,
    MODE_TYPE, CLASS_NAME, Y);

  --makes an active task & associates it with instance
  --id & puts on list of active main objects
  if MODE_TYPE = GLOBAL.ACTIVE then
    TASK_NAME.PTR := new MAIN_CL.ACTIVE_MAIN;
    TASK_NAME.INSTANCE_NAME := Y;
    ACTIVE_LIST_MUTEX.MUTEX_TASK.UPDATE_LIST(
      TASK_NAME, ACTIVE_LIST);
    TASK_NAME.PTR.INIT(Y);
  end if;
end MAKE_NEW;

end MAIN_CL;

PROCEDURE START_UP:

with GLOBAL, MAIN_CL;
use GLOBAL;
procedure START_UP is
  ID : USER_INSTANCE_NAME_PTR := new USER_NAME_REC(40);
begin
  MAIN_CL.MAKE_NEW(ID, GLOBAL.MODE_OUT, GLOBAL.ACTIVE);
end START_UP;
```

END

2002/91

FIN