1

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

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.

Automatic Code Generation for Real Time Systems

by

Hazel Jean Scott, B.A.Sc.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master of Engineering

Department of Systems and Computer Engineering,
Carleton University,
Ottawa, Ontario, Canada

July 19, 1991

© Hazel Jean Scott 1991

In between these two extremes are code generators which perform transformations from lower level specifications or designs to source languages. Essentially, code generators are capable of providing a greater degree of automation when the specification or design is detailed and the programming domain being supported by the code generator is narrow.

In the remainder of this section, a number of code generators and more general program transformation systems will be presented under the four categories of transformation techniques established by Rich and Waters [36]: procedural, schematic, deductive, and inspection. The discussion is by no means complete; however, it provides insight into the various transformation techniques being studied. A good summary of research into program transformation systems can be found in "Program Transformation Systems" by Partsch et al. [33].

2.1.1 Procedural Transformation Techniques

Procedural transformation techniques are the most developed of all the transformational techniques as procedural programming methods are well established. Most commercial transformation systems and numerous transformation systems developed at research establishments use procedural transformation techniques.

Systems employing procedural transformation techniques specify both *what* the transformation rules are and *how* to apply the transformation rules within a series of procedures. The order in which transformation rules are applied is controlled by the order in which the procedures are invoked. All the information about the transformations is contained within the code generation program.

The major advantage to using procedural techniques is that highly automatic code generators can be created as both the transformation rules and instructions to apply the transformation rules can be easily encoded. However, code generators which use procedural techniques are limited to narrow programming domains mainly because it is not feasible to encode all the transformation rules and instructions which can be applied to general programming environments. Furthermore, code generators using procedural techniques are fairly inflexible. It is not easy to tailor a procedural code generator to each programming project without signifi-

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

Automatic Code Generation for Real Time Systems

submitted by Hazel Jean Scott, B.A.Sc., in partial fulfilment of the requirements for the degree of Master of Engineering.

Chair,
Department of Systems and
Computer Engineering

Thesis Supervisor

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
August 27, 1991

Abstract

This thesis examines the development of an automatic code generation facility for TimeBench, a CASE tool for designing real time software. The approach used in this thesis is novel in that a meta-code generator called *MetaGen* is used to produce code generators for the code generation facility. The reasons for using the meta-code generator approach are three-fold: (1) it is easier to create and maintain code generators at a more abstract level; (2) the code generators are developed individually and therefore are easier to automate; and (3) the TimeBench code generation facility can be configured to suit any real time environment simply by linking in the appropriate code generators.

Two code generators have been developed using MetaGen in the course of this research. The first, *AdaGen*, transforms TimeBench designs to Ada implementations, while the second, *C-Gen*, transforms TimeBench designs to C/JNX implementations (JNX is a real time operating system). Both code generators have been tested on a number of TimeBench designs, the most significant being the communication system design which is shown in part in Appendix B. The development of the two code generators illustrates two points: (1) MetaGen can successfully produce code generators for a variety of target environments; and (2) code generators which automatically generate fully formatted, fully executable implementations can be defined using MetaGen.

This thesis also briefly identifies the need for user guidance during the code generation process since it is possible to produce more than one implementation from a single TimeBench design. The proposed method of implementing user guidance is through annotations to the TimeBench design.

Acknowledgements

I would first like to thank Ron Casselman and Greg Franks for the countless hours of their time I took trying to learn how to use TimeBench and trying to interface my software with their software. Without their help, the research for my thesis would not have progressed as quickly as it did.

I would next like to thank Gerald Karam, my thesis supervisor. Not only did he convince me that the brick walls I periodically ran into were just figments of my imagination, but he also spent many hours reading and providing suggestions for my thesis.

I would also like to thank my family and friends for the support they have provided over the past two years. I would especially like to thank Leon as he tended to suffer the consequences of all my successes and failures (and still volunteered to read the final draft of my thesis).

Finally, I would like to thank NSERC and the Department of Systems and Computer Engineering for the funding they provided.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Thesis Objectives	4
1.3	Thesis Overview	5
2	Background	7
2.1	Transformation Techniques	7
2.1.1	Procedural Transformation Techniques	8
2.1.2	Schematic Transformation Techniques	10
2.1.3	Deductive Transformation Techniques	13
2.1.4	Inspection Transformation Techniques	15
2.1.5	Summary	16
2.2	TimeBench	17
2.2.1	Overview	17
2.2.2	The TimeBench Design Tree	21
2.2.3	Accessing Objects in the TimeBench Design Tree	24
2.2.4	General Observations	28
3	MetaGen: A Generator of Code Generators	30
3.1	Overview	30
3.2	The MetaGen Specification Language	32
3.3	Writing MSL Specifications	33
3.4	Translation of MSL to Objective C	36
4	The Ada Code Generator	44
4.1	Overview	44
4.2	The Transformation of TimeBench Designs to Ada Programs	46
4.2.1	The Structural Components in the MachineChart Notation	48
4.2.2	MCL Statements	48
4.2.3	The Behavioural Components of the MachineChart Notation	50
4.2.4	ACM's	50
4.2.5	MCL Data Types and Expressions	52
4.3	Using AdaGen	55

5	The C/JNX Code Generator	70
5.1	Overview	70
5.2	The Transformation of TimeBench Designs to C/JNX Implementations	71
5.2.1	The Structural Components in the MachineChart Notation . .	71
5.2.2	MCL Statements	74
5.2.3	The Behavioural Components of the MachineChart Notation .	76
5.2.4	ACM's	76
5.2.5	MCL Data Types and Expressions	78
5.3	Using C-Gen	80
6	User Annotations	96
6.1	Overview	96
6.2	Implementing User Annotations	97
6.3	Where User Annotations are Needed	98
6.3.1	Annotations for AdaGen	99
6.3.2	The Buffer Component Revisited	102
7	Conclusions	107
7.1	Research Summary	107
7.2	Experimental Results	111
7.3	Future Work	112
A	MSL Grammar	119
B	Excerpts from the Communication System Design	122
C	Excerpts of the Ada Program Generated from the Communication System Design	133
D	AdaGen Rules Used in the Transformation of the Communication System Design	142
E	A Sampling of the Ada Library Used to Support MCL Expressions and Data Types	167
F	Excerpts of the C/JNX Program Generated from the Communication System Design	172
G	C-Gen Rules Used in the Transformation of the Communication System Design	184
H	A Sampling of the C/JNX Library Used to Support MCL Expressions and Data Types	220

List of Figures

2.1	Top level of control flag design.	19
2.2	The contents of the control flag reactor	20
2.3	A sample state machine.	21
2.4	Sample MCL code.	21
2.5	Segment c of the TimeBench Design Tree for top level of control flag design.	25
2.6	Segment of the TimeBench Design Tree for the internals of FlagACM.	26
2.7	Segment of the TimeBench Design Tree for the internals of gate button CLR.	27
3.1	The relationship of MetaGen to TimeBench.	31
3.2	Sample MSL rules.	34
3.3	Objective C code generated by MetaGen.	39
3.4	Objective C code generated by MetaGen containing tail recursion optimization.	40
4.1	Skeletal Ada package used to represent ACM's in TimeBench.	53
4.2	A Timer Component in TimeBench.	56
4.3	Ada code generated from the information contained in actor timer	57
4.4	Transformation rule for creating Ada task specifications.	58
4.5	Transformation rule for creating Ada task bodies.	59
4.6	Transformation rule for generating Ada assignment statements.	59
4.7	Transformation rule for generating Ada loop statements.	60
4.8	Transformation rule for generating Ada select statements.	61
4.9	Transformation rule for generating Ada variable declarations.	62
4.10	Transformation rule for generating Ada procedures.	62
4.11	A Buffer Component in TimeBench.	65
4.12	Ada code generated from the information contained in actor buffer	66
4.12	Continued	67
4.13	Transformation rule for generating an Ada procedure call from string and list slice assignments in MCL.	67

4.14	Transformation rule for generating an Ada expression which determines the lengths of MCL strings, lists, or sets.	68
5.1	TimeBench design which will not translate to C/JNX properly.	74
5.2	Skeletal C/JNX code showing scoping changes.	74
5.3	Revised TimeBench design to accomodate scoping in C/JNX	76
5.4	C/JNX code generated from the information contained in actor timer	79
5.4	Continued	80
5.5	Transformation rule for creating C/JNX task declarations.	82
5.6	Transformation rule for creating C/JNX task bodies.	83
5.7	Transformation rule for generating C/JNX assignment statements.	84
5.8	Transformation rule for generating C/JNX loop statements.	84
5.9	Transformation rule for generating C/JNX code which receives mail.	85
5.10	Transformation rule for generating simple C/JNX variable declarations.	86
5.11	Transformation rule for generating C/JNX functions.	86
5.12	C/JNX code generated from the information contained in actor buffer	91
5.12	Continued	92
5.13	Transformation rule for generating a C/JNX function from string and list slice assignments in MCL.	93
5.14	Transformation rule for generating a C/JNX expression which determines the lengths of MCL strings, lists, or sets.	93
6.1	Type declarations generated for unannotated producer/consumer design.	101
6.2	Annotated TimeBench design.	103
6.3	Ada code generated from annotated buffer component.	104
6.3	Continued	105
6.4	Type declarations generated for annotated producer/consumer design.	106
B.1	Overall communications system design.	123
B.2	Contents of actor source	124
B.3	Contents of one of the courier actors.	126
B.4	Contents of box comm1	127
B.5	Contents of actor comm1 (Part 1).	128
B.6	Contents of actor comm1 (Part 2).	129
B.7	Contents of actor timer	130
B.8	Contents of ACM altbit1	131

List of Tables

3.1	Transformation of MSL actual arguments to Objective C.	42
4.1	Transformation of MachineChart structural components to Ada constructs.	47
4.2	Transformation of MCL statements to Ada statements.	49
4.3	Transformation of MachineChart behavioural components to Ada constructs.	51
5.1	Transformation of MachineChart structural components to C/JNX constructs.	72
5.2	Transformation of MCL statements to C/JNX statements.	75
5.3	Transformation of MachineChart behavioural components to C/JNX constructs.	77

Chapter 1

Introduction

1.1 Motivation

The traditional software life cycle is flawed in that the steps to convert requirements to specification and then specification to implementation are both informal and undocumented. The lack of formalized procedures contributes to inefficient development and maintenance of software systems, and, furthermore, contributes to error prone implementations [4, 5]. This leads to four significant problems in software development.

- The implementation does not always satisfy the original design. Because deriving an implementation from an abstract definition of the problem: (the design) is complex, the implementor often loses sight of the original goals [34].
- Software quality is often poor because implementation methods are error prone and testing procedures are not formalized [34].
- Maintenance of software is difficult as it is performed on the implementation. The implementation has been optimized by the original programmer which tends to spread out the implementation information derived from the specification. Furthermore, no documentation exists to explain the mapping from specification to implementation. Thus, the maintainer of the software often

does not fully understand how small changes will affect the overall operation of the code [4, 5, 34].

- Often software at the implementation level is not in a reusable form because it contains implementation details and optimizations that are suitable for one system but not for another [20].

To improve the software life cycle, there needs to be a shift in focus from traditional forms of programming [36]. More emphasis should be placed on designing the software and less emphasis should be placed on implementing the software. Software should be tested, maintained and designated for reuse at the design phase of the software life cycle rather than at the implementation phase. To meet these objectives, computer assisted programming is required [4]. Ideally, automatic code generation of compilable implementations from an abstract specification of the software is sought [3, 4, 5, 24, 26, 30, 36, 40].

The most successful automatic programming systems developed thus far are those that target narrow programming domains [2, 7, 30, 40] and those that perform source-to-source translations [1, 10, 17, 31, 33]. It is possible to automate the transformation process in a narrow domain programming environment since all or at least most of the knowledge about the environment can be encoded. It is possible to automate systems which perform source-to-source translations as the information encoded in the source code of one language is usually sufficiently detailed to produce source code for another language.

Less successful systems in terms of automation are those that try to transform very general specifications into compilable implementations [5, 6, 19, 20, 22, 24, 33]. Essentially, these systems provide many of the transformation rules and perform any necessary bookkeeping when the transformation is applied. However, it is up to the user to select where and when the appropriate transformation rules apply.

One area which would greatly benefit from the widespread introduction of automatic programming systems is that of computer-aided software engineering (CASE) tools. Most CASE tools developed thus far are "front end" CASE tools in that they provide an environment for managing, designing, and analysing software systems [23, 32]. In general, automatic code generation is not available in these tools. A

number of CASE tools have been developed which concentrate solely on generating code from abstract specifications [29]. However, these tools neglect “front end” software development.

TimeBench [12, 13, 16, 18, 26], a CASE tool for developing real time and embedded software, fully supports the software paradigm advocated by Balzer et al [4] as the tool provides the designer with the means to concentrate on designing the software (or, later, redesigning the software during the maintenance phase of the software life cycle). Ultimately, the TimeBench environment will allow the designer to quickly sketch out the structure of the design at a very abstract level and then fill in the structural and behavioural details as more is learned about the design.¹ Once the designer has a sufficiently detailed design, the designer will be able to execute the design to see if it meets operational requirements. Furthermore, the designer will be able to study the temporal behaviour and performance of the design using temporal and performance analysis facilities contained within TimeBench. Once the design has been fully tested, the designer will then be able to automatically generate correct target code for the design.

Ideally, the code generation facility within TimeBench will be able to perform the majority of the transformations from design to implementation automatically. Implementation details which cannot be derived automatically from the design information will be added to the design through user annotations. These details will then be automatically integrated with the design information to generate a software implementation. The major features of the “ideal” code generation facility should be as follows.

- The facility should be able to produce code for more than one target environment.
- The facility should be able to produce optimal implementations.
- The facility should be able to produce structured implementations (e.g., into modules, into files, etc.).

¹The design methodology used in TimeBench is called architectural design [14]. Architectural design emphasizes the structural and behavioural aspects of the design.

- The facility should be able to transform high level data types.
- The facility should be able to produce high quality, legible, documented implementation code.

1.2 Thesis Objectives

The primary objective of this thesis is to develop a code generation facility for TimeBench which is capable of automatically producing implementations of TimeBench designs for a wide range of target environments. This objective must be restricted somewhat as there are a huge number of potential target environments for real time software. Therefore, the code generation facility should be configurable to the target environments of choice. This implies that the code generation facility should be composed of individual code generators each of which produces implementations for a specific target environment.

To meet this objective, this thesis examines the feasibility of using a *meta-code generator* called *MetaGen* to produce individual code generators for the TimeBench code generation facility. To produce each code generator, MetaGen automatically transforms an abstract specification of the transformation rules used by the code generator into an implementation of the code generator which can be easily integrated with TimeBench. The language used to specify the code generators is called the *MetaGen Specification Language* or MSL for short.

The success of the meta-code generator approach depends on two factors: (1) the expressiveness of MSL; and (2) the algorithms used to transform MSL to the implementation language. MSL should be both simple to use, yet sufficiently general to create code generators which produce implementations for a variety of target environments. Simplicity is a necessity as the purpose of MetaGen is to reduce the time needed to develop and maintain the code generators. Furthermore, MetaGen should be able to automatically generate code generators from their respective MSL specifications. Otherwise gains in time made by using MSL to specify the code generators could be wholly or partially lost.

Two prototype code generators are created with MetaGen to demonstrate that

the meta-code generator approach satisfies the goals stated above. The first code generator called *AdaGen* produces Ada implementations. The second code generator called *C-Gen* produces C code which runs in conjunction with a real time operating system called JNX.² Both code generators are tested on a fairly large TimeBench design (see Appendix B for excerpts of the design) to show that the code generators can automatically generate significant implementations. The differences between the two target languages are significant enough to be used as proof towards the success or failure of the meta-code generator approach.

Finally, recognizing the fact that some implementation details cannot be derived automatically from TimeBench designs either because the information is not present or is insufficient to generate the "best" possible implementation, this thesis also identifies areas in which user guidance will be beneficial. The proposed method of user guidance is through annotations to the design. The annotations will contain implementation specific information which the code generators will automatically integrate with the design information to produce implementations.

1.3 Thesis Overview

Chapter 2 presents background material for the research carried out in this thesis. The first part of the chapter describes related work and compares and contrasts the various transformation methods used. The second part of the chapter provides an overview of the TimeBench tool. It briefly describes the philosophy behind the tool and then presents in some detail the underlying data structure used to store the TimeBench designs.

Chapter 3 discusses MetaGen. It describes the syntax and semantics of MSL and presents in some detail the algorithms used by MetaGen to transform MSL to Objective C.

Chapters 4 and 5 discuss AdaGen and C-Gen, respectively. Both chapters describe the issues which had to be resolved before the transformation rules could be developed for each of the code generators. Furthermore, each chapter discusses the

²JNX is a real time operating system for IBM-PC's used at Carleton University.

testing carried out on the code generators, extracting examples from the tests to better illustrate the functionality of the code generators.

User annotations are discussed in Chapter 6. This chapter briefly describes the changes to TimeBench and MetaGen which are necessary to accommodate user annotations and identifies where annotations will be useful. A detailed example is also presented which clearly illustrates the envisaged use of annotations. Chapter 7 presents conclusions and future work.

Appendix A presents the MSL grammar in detail. Appendices B, C, and F contain excerpts from the TimeBench communication system design, excerpts from the automatically generated Ada implementation of the communication design, and excerpts from the automatically generated C/JNX implementation of the communication design, respectively. Appendices D and G contain the AdaGen and C-Gen rules used to transform the communication design into its respective implementations. Finally, Appendices E and H contain samples of the functions used to support the transformation of MCL data types and expressions to Ada and C/JNX data types and expressions, respectively.

Chapter 2

Background

2.1 Transformation Techniques

Transformation plays a key role in automatic code generation. Code generators apply transformation rules to their input (e.g., specifications, programming languages) in order to produce their output (some target language). The techniques used to apply the rules, the degree of automation achieved, and the types of input and output vary from code generator to code generator.

Many code generators have been developed both commercially and in academia. The code generators are either stand-alone or part of a software development tool and support various levels of automation. The code generators range from source-to-source translators to specification-to-source translators.¹ The source-to-source translators are generally of two types: the input and output languages are the same, and the input and output languages are different. In the former case, the purpose of the code generator is to perform optimizations on the source code. In the latter case, the purpose of the code generator is to promote software reuse.

Code generators which perform transformations from high level specifications to source languages must be “intelligent” as it is necessary to make numerous implementation decisions during the transformation process. In most cases, the intelligence of these code generators is provided by the user of the system.

¹Source languages are languages that can be compiled using current technology.

In between these two extremes are code generators which perform transformations from lower level specifications or designs to source languages. Essentially, code generators are capable of providing a greater degree of automation when the specification or design is detailed and the programming domain being supported by the code generator is narrow.

In the remainder of this section, a number of code generators and more general program transformation systems will be presented under the four categories of transformation techniques established by Rich and Waters [36]: procedural, schematic, deductive, and inspection. The discussion is by no means complete; however, it provides insight into the various transformation techniques being studied. A good summary of research into program transformation systems can be found in "Program Transformation Systems" by Partsch et al. [33].

2.1.1 Procedural Transformation Techniques

Procedural transformation techniques are the most developed of all the transformational techniques as procedural programming methods are well established. Most commercial transformation systems and numerous transformation systems developed at research establishments use procedural transformation techniques.

Systems employing procedural transformation techniques specify both *what* the transformation rules are and *how* to apply the transformation rules within a series of procedures. The order in which transformation rules are applied is controlled by the order in which the procedures are invoked. All the information about the transformations is contained within the code generation program.

The major advantage to using procedural techniques is that highly automatic code generators can be created as both the transformation rules and instructions to apply the transformation rules can be easily encoded. However, code generators which use procedural techniques are limited to narrow programming domains mainly because it is not feasible to encode all the transformation rules and instructions which can be applied to general programming environments. Furthermore, code generators using procedural techniques are fairly inflexible. It is not easy to tailor a procedural code generator to each programming project without signifi-

cantly complicating the interface to the code generator or, in some cases, without modifying the code generator itself.

GRAMPS

GRAMPS [17] is a method for developing metaprograms such as source-to-source translators. The idea behind the method is that a fixed set of procedures for manipulating the parse tree of the languages input to and output from a metaprogram can be derived systematically from the extended BNF of the languages. These procedures can then be used by a metaprogrammer to create a metaprogram which performs source-to-source translation.

GEM [25] is a tool which automates the GRAMPS method. GEM makes it possible to quickly generate metaprogramming environments for a host of applications by automatically deriving the procedures described above. It is, however, still up to the metaprogrammer to write the metaprogram that controls the application of these procedures.

PascAda

PascAda [1] is a system which can perform automatic transformations from Pascal code to Ada code and vice versa. PascAda depends on two highly compatible sublanguages to support the bidirectional transformations. These sublanguages are PascalA and AdaP, respectively. Each of the sublanguages is constructed from a subset of Pascal and Ada's language components which can be easily translated from one language to the other.

Transformation from one language to the other is performed as a series of steps through the different language levels. For instance, in transforming Pascal to Ada, the Pascal code is first converted to PascalA code which in turn is converted to AdaP code.

The individual transformations are executed as a series of parse tree manipulations. When a subtree within the parse tree satisfies the applicability condition of a transformation rule, the subtree is replaced with a new subtree which implements the transformation specified in the rule.

ASCGEN

ASCGEN [40] automatically transforms CAEDE iconic designs into Skeletal Ada code. The design notation used in CAEDE is based on the notation developed by Buhr [11] to graphically represent Ada components. Skeletal Ada is a subset of Ada and only deals with the structural aspects (such as packages) of the Ada language.

ASCGEN is written in C Prolog; therefore, before the transformation can take place, the designs must be first converted to a Prolog representation (which is written in the Prolog Design Language). Then through a series of recursive steps, ASCGEN analyses the design, translates the individual design components into Skeletal Ada, and then generates Skeletal Ada code.

ASCGEN performs the transformation from the design to Skeletal Ada in a single pass. The analysis, translation, and code generation phases are interleaved. Thus, there is no need for an intermediate data structure to store Skeletal Ada code. Instead, Skeletal Ada code is output as the transformations occur.

ADDS

ADDS [2] is an example of a class of highly specialized code generation facilities called program generators [30]. The primary purpose of program generators is to allow the average computer user to customize software packages to meet the needs of their environment.

ADDS automatically generates user interfaces for Ada programs. ADDS takes as input an annotated Ada package specification which describes the desired user interface and automatically generates the package body to go with the specification. Essentially ADDS contains a number of templates for user interfaces written in Ada. The user selects and customizes one of these templates through the package specification.

2.1.2 Schematic Transformation Techniques

Schematic transformation techniques use pattern matching and pattern replacement to perform transformations. The transformation rules consist of an input pattern

which syntactically matches the element to be transformed, an applicability condition which usually contains semantic information about the transformation, and a replacement pattern which implements the transformation.

The transformation rules are applied to the input by some algorithm which makes repeated passes through an intermediate representation of the input (such as a parse tree) until a terminating condition is met. The selection of which transformation rules to apply in a given pass, the order of application of the rules, and the terminating condition depend on each individual transformation system.

The major advantage to using schematic transformation techniques is that the transformation rules can be expressed clearly and concisely. The transformation rules are written using a standard format. Furthermore, they are separate from the algorithm used to apply the rules rather than being integrated as in the procedural transformation method. Thus, the transformation rules are much easier to modify. Ideally, transformation rules can be added to and removed from the system without affecting the algorithm used to apply the rules.

The major weakness of systems using schematic transformation techniques is the algorithm used to apply the transformation rules. Only program transformation systems which operate in fairly narrow programming domains have algorithms which can automatically apply all the transformation rules. The algorithms used in general purpose transformation systems require user assistance for the selection and ordering of the transformation rules.

TAMPR

TAMPR [10] transforms LISP programs into FORTRAN programs. Because LISP programs are far more expressive than FORTRAN programs, the transformation is broken down into a series of manageable transformations between intermediate language levels. During each step, TAMPR performs an exhaustive post-order walk on the parse tree of the language level being transformed. The transformation rules associated with that phase are applied when the input pattern in the rule matches with a subtree within the parse tree. Transformation for that step terminates when no more transformation rules can be applied.

With the exception of a few initial transformations, TAMPR is fully automatic. However, it takes a very long time to transform a LISP program to a FORTRAN program. Four hours to transform 1300 lines of LISP is the example cited in the paper.

PDS

PDS [19, 20] is a complete environment for program development of which transformation is a small part. The purpose of the transformation system within PDS is to transform an "abstract" implementation of a program into an optimal implementation of the program.

All aspects of program transformation are controlled by the user of PDS. The user must specify the transformation rules, the scope over which the transformation rules apply, and, if needed, the specific order in which the transformation rules must be applied. This information is then input to PDS in a refinement module. The PDS transformation algorithm traverses the parse tree top down as many times as is necessary to apply all the transformation rules contained in the refinement module according to the instructions provided.

CIP

The CIP project [6] is concerned with formalizing program development. To achieve this end, a system is being developed which transforms formal specifications into implementations. The approach used in this project is unique in that a wide spectrum language supports all levels of transformation. Wide spectrum languages are capable of expressing both abstract constructs needed to write formal specifications as well as low level constructs needed to write compilable implementations.

The user of the CIP transformation system is responsible for all creative aspects of the transformational process. First, the user must derive a catalogue of transformation rules from the small set of fundamental transformation rules provided by the CIP transformation system. Then during the transformation process, the user must select which transformation rules to apply. The CIP system takes care of the application of the selected transformation rules automatically. As an alternative to

selecting the transformation rules one by one, the user can input a metaprogram to the transformation system which specifies a sequence of transformation rules to apply to the formal specification.

ZAP

ZAP [22] transforms recursive program definitions into recursive implementations. Transformation from the original program definition to an intermediate representation must be performed by hand. ZAP then transforms the intermediate representation to the final implementation.

The user of the ZAP system must input a metaprogram which contains the schematic transformation rules as well as guidelines for applying the rules. Using these guidelines and the input patterns in the transformation rules, ZAP finds every location in the intermediate representation where the rules may be applied. To test if the rule is applicable at a given location, ZAP unfolds both the input pattern and the replacement pattern according to both the guidelines and the information contained in the intermediate representation. If the two patterns match after unfolding, the rule is applicable and the input pattern is replaced by the replacement pattern.

2.1.3 Deductive Transformation Techniques

Deductive transformation techniques use theorem proving as a basis for automatic code generation. Given an initial set of facts (the input program or specification to transform), the code generator tries to deduce goals from the facts using a series of rules about the programming environment. In the process of deriving the goals, the transformation takes place.

Systems using deductive transformation techniques have the same weakness as those using schematic transformation techniques. User input is required to direct the transformational process for general purpose programming environments. The reasons for this are twofold: first, a set of heuristics to mimic human thought processes during implementation of a program from a specification have not been developed (and cannot be until this process is formalized); and second, it takes time to build a knowledge base which deals with all aspects of program transformation.

The TI Project

The TI project [5, 24] has been developed to experiment with various aspects of computer-assisted software development. One area within the TI project which has received much attention is computer-assisted transformations from high level specifications to implementation level languages. To this end, a formal specification language called GIST and two transformation facilities called PROPART and GLITTER have been created. GLITTER is the transformation facility to be covered in this section as it uses deductive transformation techniques.

GLITTER is an expert system which formalizes design decisions and strategies that are used when creating a detailed specification from an abstract specification. Initial transformation goals and strategies usually have to be selected by the user because the abstract specification does not contain enough information for GLITTER to proceed automatically. However, GLITTER does guide the user through this initial phase by asking appropriate questions.

As more details about the design are input, GLITTER makes more and more decisions automatically using selection rules either input by the user or contained in GLITTER's knowledge base to resolve any competing strategies. GLITTER has the potential to perform at greater levels of automation as it gains more knowledge about programming.

CHI

The CHI project [28, 37, 41] has been undertaken to study aspects of knowledge-based software environments. The CHI project centers around the development of a wide spectrum language called V and the development of a description knowledge base called DKB.

V can be used for expressing both high level and low level constructs. Therefore, abstract specifications as well as source programs can be written in the same language. All the prototype systems developed for CHI manipulate V programs in some way using the information contained in the DKB.

DKB contains knowledge about the software environment supported by CHI. The kernel of DKB describes all the V language constructs. Further descriptions about

the environment are added as the knowledge base is used. These descriptions include information about programming knowledge (in the form of schematic transformation rules), information about V programs under development, the development history of V programs, and information about conceptual models of various programming domains.

Two of the prototype systems developed for CHI are a rules compiler and a data structure synthesizer. The rules compiler converts rules expressed in V to procedural V and then to LISP. The data synthesizer transforms high level V data types to lower level V data types. Both systems interact with users and the DKB to perform the transformations. Users guide the transformation process by choosing between competing implementations stored in the DKB and, if necessary, supplying more information to the DKB. Every phase of the transformation process is recorded in the DKB. Therefore, if the user decides that one branch of the transformation is unsatisfactory, it is possible to backtrack to an appropriate point and continue along another branch.

The results of the research into the CHI software environment have been incorporated into a commercial product called REFINE.

2.1.4 Inspection Transformation Techniques

Inspection transformation techniques perform code generation by integrating fragments of well known algorithms. These techniques cannot be the sole transformation technique used as programs cannot be completely formed by combining a number of well known algorithms. Other details such as data declarations, special purpose procedures, and connecting code between the algorithms are needed as well.

GIBBSGEN

GIBBSGEN [7] is a facility for generating FORTRAN code from Gibbs chapters. Gibbs chapters are components of the Gibbs data base which has been developed to promote reuse of ideas. Gibbs chapters contain both an English language description and a formal description of an algorithm, optimization, or data definition.

Corresponding to each Gibbs chapter is a Prolog module which semiautomatically generates FORTRAN code from the formal specification contained in the Gibbs chapter. The user must provide some information to the Prolog module to clear up any ambiguities that arise during code generation. FORTRAN programs are created by combining a number of Gibbs chapters and executing the Prolog modules associated with each chapter. The Prolog modules are written by hand and not generated automatically from the Gibbs chapters.

2.1.5 Summary

Code generators which use procedural techniques are most common mainly because procedural programming methods are well established. The major advantage of procedural code generators is that high levels of automation can be achieved since it is fairly easy to encode all the transformation information as a series of steps. However, procedural code generators are limited to narrow programming domains and tend to be inflexible.

The thrust of research has been to develop code generators which use schematic transformation techniques. These systems are advantageous in that the transformation rules are separate from the algorithm used to apply them. Therefore, systems using schematic transformation techniques ideally should be able to produce multiple implementations from a single specification simply by accessing different sets of transformation rules. Unfortunately, the only systems in which high levels of automation have been achieved are those that support a narrow programming domain (e.g., TAMPR). The remaining systems require a significant amount of human assistance in order to complete the transformations.

Less common are systems using deductive transformation techniques and inspection transformation techniques. Systems using deductive transformation techniques have much potential. However, until all methods used to transform software specifications into software implementations are formalized, it will not be possible to build a complete knowledge base for the deductive system. Therefore, as with schematic transformation systems, deductive transformation systems require a significant amount of user guidance.

Inspection transformation techniques can be integrated with procedural, schematic, or deductive techniques but cannot be used alone. Systems depending solely on inspection transformation techniques are missing too much knowledge to ever become completely automated.

2.2 TimeBench

2.2.1 Overview

TimeBench, a CASE tool being developed at Carleton University, provides designers with an environment for designing real time and embedded systems. TimeBench can be used to develop, analyse, test, maintain and automatically implement time-intensive software systems.

The key component within the TimeBench environment is the design tool which uses the architectural design method developed by Buhr [14]. Architectural design iterates back and forth between developing the structure of the software system and developing the behaviour of the software system.

Fundamental to Buhr's design method is a visual design notation called *MachineCharts*. For the most part, the visual notation represents the structure of the design. However, some visual annotations express high level behavioural characteristics.

In addition to the visual notation, Buhr has provided two mechanisms through which detailed behaviour can be expressed: *abstract controller machines* (ACM's) and *engines*. ACM's contain *state machines* (SM's) which represent the abstract behaviour of some component in the design. The ACM acts as an interface between the abstract behaviour of the SM and the "concrete" behaviour of the surrounding design. In effect, ACM's allow abstract behaviour to be plugged directly into the design.

Engines provide a place in which the functional behaviour of the design can be described in detail. Engines contain *agenda's* which sequence the high level behaviour of the visual annotations and fill in any missing operational details. Buhr leaves the contents of the engines whether it be pseudocode, a natural language

description, etc., up to the designer. However, as designs in TimeBench must be operational, the contents of the engine have been limited to a high level design language called *MachineChart Language* (MCL).

In order to introduce the TimeBench design notation, a small TimeBench design is presented in the remainder of this section. A complete description of the TimeBench design notation can be found in the *User's Guide for TimeBench* [35].

The design, shown at its most abstract level in Figure 2.1, is a simple control flag problem. There are three autonomous customers who would all like to access a shared data structure, the control flag. To prevent more than one customer from accessing the control flag at a time, mutual exclusion is provided through a semaphore called **ControlFlag**. The notation used in Figure 2.1 to express these concepts is summarized as follows.

- The customers are denoted by *actors*. Actors are drawn as large parallelograms and represent autonomous entities within the design.
- The control flag is protected by a *reactor*. Reactors are drawn as trapezoids and provide mutually exclusive access to their contents. Depending on the implementation, reactors may or may not become autonomous entities.
- Access to the control flag is provided by *gate buttons*. Gate buttons are the small parallelograms drawn on the edges of the reactor. Gate buttons act as rendezvous points for both reactors and actors.
- The fact that each customer interacts with the control flag is illustrated by the use of *push fingers*. Push fingers in are represented by arcs which, in this case, run from the actors to the gate buttons.
- The flow of data from the control flag to the customer is denoted by *data flows*. Data flows are drawn as arrows terminated by circles.

The control provided by the semaphore is shown in more detail in Figure 2.2. In addition to the gate buttons which are described above, there are two more constructs.

Figure 2.1: Top level of control flag design.

- Access to the control flag is controlled by an ACM. The ACM is drawn as a box with a circle in the middle.
- The control flag is contained within a *data store*. The data store is drawn as an oval in the figure.

The SM contained in the ACM is displayed in Figure 2.3. The trapezoid denotes an *initial state*, whereas the circle denotes an *intermediate state*. The arcs between the states represent *state transitions*. *Events* and/or *conditions* which trigger the transition from one state to another may or may not be specified for all transition

Figure 2.2: The contents of the control flag reactor.

Figure 2.3: A sample state machine.

arcs; however, the *actions* which are performed when the transition occurs will always be specified.

The SM for the control flag problem grants access to a requesting customer only when it is in state YES. Once access is granted, the state changes to NO. The state returns to YES when the customer accessing the control flag signals that it is finished.


```

engine CLR is
begin CRITICAL_REGION
    anEvent := FlagACM.Done;
    push GetAction2;
end CRITICAL_REGION;
begin
end;

```

Figure 2.4: Sample MCL code.

The detailed operation of the semaphore is described in the agenda's of the *implicit* engines contained within the gate buttons. The MCL code contained in the engine labelled CLR is shown in Figure 2.4. MCL code in gate buttons is always divided into two regions. This is because all interactions with gate buttons involve rendezvous. The region between the **begin CRITICAL_REGION** and **end CRITICAL_REGION** markers contains code which is executed during the rendezvous. The region between the **begin** and **end** markers contains code which is executed after the rendezvous. In this example, there is no post-rendezvous code.

The first MCL statement in the critical region is an *assignment* statement. The second is a *push* statement which causes the implicit gate engine to activate the push finger labelled *GetAction2* in Figure 2.2. In this case, the push statement can be equated to a subprogram call.

2.2.2 The TimeBench Design Tree

TimeBench is implemented in Objective C; therefore, an Objective C data structure (which for the purposes of this thesis will be called the *TimeBench Design Tree*) is used to store TimeBench designs. To provide some background for the discussion on the TimeBench Design Tree, a brief description of object oriented programming concepts is presented. The concepts outlined in the following paragraphs are taken from Booch's *Object Oriented Design: With Applications* [9].

Objects are the building blocks of any program using object-oriented techniques. Booch defines an object as follows:

An object has state, behaviour, and identity; the structure and behaviour of similar objects are defined in their common class; the terms instance and object are interchangeable.

The state of an object describes the object's structure and attributes. The behaviour of the object describes the actions that can be performed on and by the object. These actions affect the state of the object. The object's identity uniquely distinguishes the object from all other objects.

Objects which have similar structure and behaviour can be collected together in a single *class*. For instance, in the control flag example, all three customers are objects which belong to the *actor class* as they all have the same structure (that of a parallelogram) and they all exhibit the same behaviour (that of autonomous entities). Each customer is an instance of the actor class as each customer has its own identity (*Cust1*, *Cust2*, and *Cust3*) and its own attributes (e.g., each customer performs a different task).

Actions which can be performed on or by an object are specified in *methods*. Methods are typically written for an entire class of objects rather than a single object and can be applied to any object contained in that class. Methods are invoked by sending *messages* to objects.

Classes of objects are often related to one another in a hierarchical fashion. For instance, the push fingers in the control flag example belong to the *push arc class* which contains a detailed description of the behaviour exhibited by push fingers. The *push arc class* is a subclass of a more abstract class called the *arc object class*. The *arc object class* describes the general behaviour of all arc objects. Push fingers *inherit* the methods and attributes described in the *arc object class* and are further classified by the methods and attributes found in the *push arc class*.

The TimeBench Design Tree is arranged as a hierarchy of objects; each object describing a specific component in the design. There are four general classes into which the objects fall:

- a class of objects describing components of the MachineChart visual notation;
- a class of objects describing components of SM's;

- a class of objects describing components of MCL; and
- a class of objects describing interactions between the design components.

For the purposes of this thesis, these classes will be called the *MachineChart class*, the *ACM class*, the *MCL class*, and the *arc object class*, respectively.

Objects which belong to the MachineChart class have a hierarchical relationship with one another in the TimeBench Design Tree. This relationship reflects the hierarchical nature of TimeBench designs. It is evident from the control flag example that TimeBench designs are hierarchical in structure since details about the implementation of the control flag semaphore are contained inside the reactor. In the example, the reactor object is the parent of all the MachineChart objects shown in Figure 2.2.

MachineChart objects can have both *external* and *internal* children. External children correspond to MachineChart icons which act as interface components in the design. For example, the gate buttons in Figure 2.1 become external children of the reactor in the TimeBench Design Tree since the buttons act as an interface to the reactor. Internal children correspond to MachineChart icons which are hidden inside other icons. For instance, the ACM and data store in Figure 2.2 are internal children of the reactor in the TimeBench Design Tree.

Objects belonging to the ACM class are found in specific subtrees of the TimeBench Design Tree. These subtrees have ACM objects at their root node. The relationship between ACM objects in the TimeBench Design Tree is arbitrary as each object in this class has specific relationships with the other objects in this class.

Objects belonging to the MCL class also are found in specific subtrees of the TimeBench Design Tree. In this case, the subtrees have engine objects at their root node. The relationship between MCL objects is also hierarchical but in this case the hierarchy is based on the structure of the language. Each language element corresponds to a unique object in the TimeBench Design Tree.

The arc object class exhibits more than just parent/child relationships with other objects. Arc objects (such as the push fingers and transition arcs in the control flag example) denote interactions among either MachineChart objects or ACM objects.

Therefore, arc objects can be related to objects located in other branches of the TimeBench Design Tree as well.

Selected fragments of the TimeBench Design Tree used to store the control flag design are shown in Figures 2.5 through 2.7. More specifically, Figure 2.5 contains the fragment of the design tree corresponding to the design components shown in Figures 2.1 and 2.2. All the objects contained in this fragment of the design tree are MachineChart objects or arc objects. Figure 2.6 contains the fragment of the design tree corresponding to the design components in Figure 2.3. This subtree contains ACM objects and arc objects. Finally, Figure 2.7 contains the fragment of the design tree corresponding to the design components in Figure 2.4. This section of the design tree contains only MCL objects.

The solid lines drawn between objects in Figures 2.5 through 2.7 represent the hierarchical relationship between the objects. The dotted lines represent the special relationship between interacting objects. If the arrowhead on the dotted line is pointing towards an arc object, then the object located at the other end of the line is the source of the arc. Otherwise, the object is the destination of the arc.

Solid lines followed by three dots denote branches of the TimeBench Design Tree which have not been expanded.

2.2.3 Accessing Objects in the TimeBench Design Tree

A logical interface written in Objective C is provided to access objects in the TimeBench Design Tree. The interface is composed of a hierarchy of class definitions. Abstract class definitions group together large numbers of objects which exhibit similar properties (e.g., the MachineChart class, the MCL class, and the ACM class). The most detailed class descriptions group together objects which are associated with the same design component (e.g., the actor class, the reactor class, etc.).

The hierarchical structuring of the class definitions means that methods which are applicable to a large number of objects can be located in a single class. These methods are inherited by objects contained in subclasses of the original class.

The methods of interest for this thesis identify objects in the TimeBench Design Tree and determine their relationships to other objects in the TimeBench Design

Figure 2.5: Segment of the TimeBench Design Tree for top level of control flag design.

Figure 2.6: Segment of the TimeBench Design Tree for the internals of FlagACM.

Tree. To provide a basic understanding of how the logical interface works, some of the steps necessary to retrieve the objects from the segments of the TimeBench Design Tree shown in Figures 2.5 to 2.7 are presented in the remainder of this section.

Initial access to the objects in any TimeBench Design Tree is provided through a special root node. To retrieve the children of the root node the message **internals** is sent to the root node object. A list of the children is returned upon invocation of the corresponding method.

Figure 2.5 contains two general types of objects: those belonging to the MachineChart class; and those belonging to the arc object class. To retrieve the children of objects belonging to the MachineChart class two general methods apply: **internals** and **externals**. The first method retrieves internal children of the object and the second method retrieves external children of the object. If the object

Figure 2.7: Segment of the TimeBench Design Tree for the internals of gate button CLR.

has no external children, the value `nil` is returned.

To retrieve the objects at the source and destination of any arc object the messages `ultimateSrcNode` and `ultimateDstNode`, respectively, are sent to the arc object.

The data flows in Figure 2.5 are only associated with one type of arc object, that of push fingers. Therefore, the method for retrieving data flows is located in the class description for the push finger class and does not apply to all arc objects. The method used to retrieve data flows is `ultimateDataFlows`.

Most of the methods used to retrieve information from objects in Figure 2.6 are located in detailed class descriptions as the tasks performed by each object are very specific. For instance, to retrieve the events and actions associated with the transition arcs, the messages `event` and `returnAction` are sent to the transition arcs.

The objects in Figure 2.7 are all MCL objects. It is important that these ob-

jects are processed in order to preserve the meaning of the MCL code. MCL code subtrees are retrieved from the TimeBench Design Tree by sending the message `mcl` to engines.

The assignment statement and selected component in Figure 2.7 are examples of a common subclass of MCL objects, two-sided objects. To retrieve children of two-sided objects, the messages `left` and `right` are sent to the object.

The push statement in Figure 2.7 does not have any children of its own. Instead, push statements access associated push fingers. Thus, none of the information required for executing the push statement has to be stored in the MCL subtree. Information such as the data flowing between objects and the destination object can be retrieved by querying the push finger. The message sent to the push statement to retrieve the corresponding push finger is `parseNode`.

2.2.4 General Observations

The design of the TimeBench Design Tree restricts the design of the automatic code generation facility in the following ways.

- The TimeBench Design Tree is essentially read only. The logical interface does provide methods to add extra information to the nodes of the design tree, but it is not possible to restructure the design tree by deleting nodes or moving nodes to other locations in the tree. Therefore, the code generator cannot manipulate the TimeBench Design Tree.
- Many of the methods for accessing ACM and MCL objects are case specific. There is no general method such as `retrieveAllChildren` which can be applied to these objects. Thus, the code generator has to deal with each of these objects individually.
- The TimeBench Design Tree is an Objective C data structure and the logical interface to the objects in the design tree is written in Objective C. For compatibility, the code generator should also be written in Objective C.

The effects of these restrictions are described in greater detail in later chapters.

The first two restrictions can be removed if the logical interface to the TimeBench Design Tree is enhanced. The improvements which can be made to the code generation facility once these restrictions are lifted are outlined briefly in the conclusions under "Future Work."

Chapter 3

MetaGen: A Generator of Code Generators

3.1 Overview

MetaGen is a tool implemented in Prolog which automatically produces code generators for TimeBench. The input to MetaGen is a code generator specification written in the *MetaGen Specification Language* (MSL), a Prolog-like language, and the output from MetaGen is a code generator written in Objective C, the language used to implement TimeBench. When linked into the TimeBench code generation facility, a code generator produced by MetaGen will automatically transform TimeBench designs into software implementations for a target programming environment (e.g., Ada, C/JNX) supported by the code generator. A separate code generator must be created for every target environment supported by the TimeBench tool. Figure 3.1 illustrates the relationship of MetaGen to TimeBench.

The code generators produced by MetaGen use procedural transformation techniques. Because the TimeBench code generation facility is composed of a number of individual code generators each of which transform the TimeBench design notation to a specific implementation language, the programming domain that each code generator has to deal with is narrow. Therefore, it is possible to create highly automatic code generators for the TimeBench code generation facility, thus satisfying

Figure 3.1: The relationship of MetaGen to TimeBench.

one of the objectives of this thesis.

One of the problems with using procedural transformation techniques is the resulting inflexibility of the code generator. This is unsatisfactory as the other goal of this thesis is to produce a code generation facility which can be configured to suit the user's needs. Using MetaGen to create the code generators for TimeBench removes much of this inflexibility since MSL makes it easier to specify and later modify the code generators.

MetaGen also uses procedural transformation techniques to transform MSL to a small subset of Objective C constructs. Because MetaGen deals with a very narrow programming domain and can take advantage of the powerful pattern matching capabilities of Prolog [39], MetaGen is fully automatic.

The remaining sections of this chapter describe MetaGen in detail. The next two sections deal with MSL and the final section describes the transformations used by MetaGen to produce Objective C code generators from MSL specifications.

3.2 The MetaGen Specification Language

The MetaGen Specification Language (MSL) is a Prolog-like language used to specify the rules for transforming TimeBench designs into corresponding implementations. The reasons for choosing a Prolog-like language are twofold:

- The Prolog parser can be used to input the specification to MetaGen.
- The transformation rules can be stated logically without any need to worry about implementation details such as data types.

An MSL specification is composed of *facts* and *rules* which are syntactically defined as follows:

Fact : Head.
Rule : Head $\longrightarrow$ Body.

The *head* of a fact or rule is composed of a *functor* and a *formal argument list* and has the following format:

$$functor(A_1, +A_2, \dots, -A_N)$$

The argument list may contain 0 to N arguments all of which must be variable names. The '+' and '-' symbols attached to the arguments signify the mode of the argument. The '+' symbol means the argument is being passed in; the '-' symbol means the argument is being passed out. The default mode of an argument is mode *in*.

The *body* of a rule is composed one or more *expressions*. The most fundamental expressions found in the body are composed of *functors* and *actual argument lists* as follows:

$$functor(B_1, B_2, \dots, B_N)$$

The arguments in the actual argument list can be string constants, integer constants or variable names. Fundamental expressions which access *built-in rules* are enclosed in curly braces ($\{\}$). Built-in rules will be explained in the next section.

Logical relationships between expressions in the body are expressed using one of two constructs:

expression1, expression2
expression1|expression2

The first construct reads as *expression1* “and” *expression2* and the second construct reads as *expression1* “or” *expression2*. The logical relationship represented by the “|” construct can also be expressed by creating two MSL rules with the same head. For example the rule

$$a(A) \longrightarrow b(B)|c(C).$$

is equivalent to the following two rules:

$$\begin{aligned} a(A) &\longrightarrow b(B). \\ a(A) &\longrightarrow c(C). \end{aligned}$$

A detailed description of the MSL grammar can be found in Appendix A.

3.3 Writing MSL Specifications

MSL specifications are composed of a series of facts and rules which contain a complete set of instructions for transforming TimeBench designs to implementations. Because the TimeBench Design Tree cannot be modified, the MSL specification must integrate instructions for processing objects in the TimeBench Design Tree with both instructions for transforming the objects to implementation code and instructions for outputting the implementation code. This method of integrating the three phases of the transformation process is similar to that used by Van Der Loo [40].

Shown in Figure 3.2 are two MSL rules taken from the MSL specification of Ada-Gen. The two rules are a subset of the transformation rules used to convert a *push button* into an Ada subprogram specification. The evaluation of these rules depends on (1) the success or the failure of the fundamental expressions contained in the bodies of each rule; and (2) the logical constructs used to separate the expressions.

There are two types of fundamental expressions in MSL rules: those which derive their results from the invocation of other user defined MSL rules or facts; and those which derive their results from the invocation of built-in rules. Fundamental

```

subprogram_specification(ButtonNode) -->
    {print_literal("PROCEDURE ")},
    identifier(ButtonNode),
    {get_formal_parameters(ButtonNode, FormalParms)},
    ({is_empty(FormalParms)} |
     ({print_literal("(")},
      formal_parameter_list(FormalParms),
      {print_literal(")")})).

formal_parameter_list(FormalParms) -->
    {extract_element(FormalParms, FormalParm, FPTail)},
    parameter_specification(FormalParm),
    ({is_empty(FPTail)} |
     ({print_literal("; ")},
      formal_parameter_list(FPTail))).

```

Figure 3.2: Sample MSL rules.

expressions which derive their results from MSL facts always succeed. A rule or fact is invoked when the functor and arity (the number of arguments) in a fundamental expression matches with the functor and arity in the head of the rule or fact.

Upon transformation of MSL to Objective C, invocations of the built-in rules become function calls to predefined Objective C functions. Four libraries of predefined Objective C functions are supplied with MetaGen. The development of these libraries was inspired by the work of Cameron and Ito [17]. They derive a number of standard functions for operating on the parse trees of programming languages supported by the GRAMPS environment. Using these standard functions, metaprograms (such as source-to-source translators) can be developed. MSL specifications are very similar to the metaprograms described by Cameron and Ito in that the specifications order the MetaGen built-in rules in such a way that the appropriate transformations from the TimeBench design notation to a specific implementation language take place.

Each library contains predefined functions which all perform similar tasks. The tasks are categorized as follows.

- Fetch objects from given nodes in the TimeBench Design Tree. Succeed if the object can be returned successfully; fail if the object is not present.
- Determine the attributes of an object located at a given node in the TimeBench Design Tree.
- Operate on lists of objects. These operations are needed for processing the children of a given node in the TimeBench Design Tree.
- Generate code for the implementation language. This library contains sufficiently abstract functions to handle any target language.

The two logical constructs introduced in the previous section control the order in which the transformation rules are evaluated. The expression

$$expression1, expression2$$

succeeds only if both *expression1* and *expression2* succeed. If *expression1* fails, *expression2* will not be evaluated and the entire expression will fail. The expression

$$expression1|expression2$$

succeeds if *expression1* or *expression2* succeeds. *Expression2* is evaluated only if *expression1* fails. The entire expression fails when both *expression1* and *expression2* fail. Some care must be taken when writing rules which contain the “|” construct as fundamental expressions such as `print_literal/1` shown in Figure 3.2 produce side effects which cannot be undone.

The order in which expressions are evaluated can be further controlled by using parentheses. For example,

$$expression1, expression2|expression3$$

means “or” *expression2* and *expression3* and then “and” the result with *expression1*; whereas

$$(expression1, expression2)|expression3$$

means “and” *expression1* and *expression2* then “or” the result with *expression3*.

The evaluation of MSL is somewhat similar to Prolog in that MSL rules succeed or fail and MSL facts always succeed. However, unlike Prolog, MSL is deterministic as it does not support backtracking. Once MSL variables are assigned a value, the value remains bound to the variable even when the branch of the MSL rule containing the variable fails. It is much easier for MetaGen to transform a MSL specification to an Objective C implementation when MSL is deterministic. If MSL were nondeterministic like Prolog, MetaGen would also have to generate Objective C code to support Prolog’s unification algorithm [39], a nontrivial task.

One final point should be observed when writing MSL specifications. In order to integrate the resulting code generator with the TimeBench code generation facility, a special MSL rule is needed. The format of the rule is as follows:

$$\text{root}(\text{TimeBenchTree}) \longrightarrow \text{expression1}, \text{expression2}, \dots, \text{expressionN}$$

This rule becomes the Objective C function which is first invoked by TimeBench when the code generator is selected. Furthermore, the TimeBench Design Tree is passed to the code generator when this rule is invoked.

3.4 Translation of MSL to Objective C

MetaGen transforms MSL rules and facts to Objective C functions using four separate passes through each rule or fact. The first pass combines rules and facts having heads with the same functor and arity into a single rule. The second pass performs semantic checking. The third pass converts MSL constructs to Objective C constructs. Finally, the fourth pass generates local variable declarations for the resultant Objective C functions.

The first pass is necessary to resolve rule (or fact) renaming in MSL. For instance, the following rule definitions are acceptable in MSL:

$$\begin{aligned} a(A) &\longrightarrow b(A), c(B), d(B). \\ a(C) &\longrightarrow d(C), c(B), e(B). \end{aligned}$$

If each of these rules is simply converted to an Objective C function, two errors will occur. First, the Objective C compiler will not accept two function declarations having the same name, and second, the meaning of the MSL rules will be lost.

The first pass resolves this problem by combining rules and/or facts with the same name into a single rule which retains the meaning of the multiple definitions. For example, the two rules above are combined into the following rule:

$$a(A) \longrightarrow (b(A), c(B), c(B))|(d(A), c(D), e(D)).$$

The most difficult task performed by the first pass is the resolution of variable names when combining rules. Resolution of variables does not apply to facts. The algorithm to perform this task is as follows.

- The variables in the first rule are noted but otherwise left alone. There are two types of variables to be processed from this point on: variables located in the formal parameter list, and variables local to the body of the rule.
- All the variables in the body of the next rule are inspected.
 - If the variable's name matches with that of a variable in the formal parameter list, the name is changed to match that of the corresponding variable in the formal parameter list of the first rule.
 - If the variable is local to the rule, the variable's name is checked against the names of all the variables encountered in the first rule to see if it has a different name. If its name is the same as any of the variable names found in the first rule, the variable is given a new name. If there is more than one instance of the local variable in the rule, all the names must receive the same treatment.
- Continue inspecting all the variables in the bodies of the remaining rules. This step is essentially the same as the previous one except that the local variables' names are compared against all those which have previously been processed, not just those in the first rule.

The second pass checks to see if all the rules referenced in the body of the rule being processed are defined. At the same time, the second pass checks to see if the rule contains tail recursion. If there is tail recursion present, a flag is set to signal to the next pass that tail recursion optimization is required. Note, the flag is set only if the tail recursive rule has formal arguments with mode *m*. As of yet, the tail recursion optimization in MetaGen cannot handle formal arguments with mode *out*.

The third pass generates Objective C functions from MSL rules and facts. Many of the techniques used in the third pass are derived from Stanczyk's work [38] as she is also concerned with transforming a Prolog-like language to a procedural language although on a much larger scale.

Shown in Figures 3.3 and 3.4 are the Objective C functions generated from the MSL rules in Figure 3.2. The head of each rule becomes an Objective C function declaration and the body of each rule becomes an Objective C function body. The fundamental expressions in the body of each rule map to Objective C function calls while the logical constructs “,” and “|” map to Objective C “if statements.” The tail recursion in the second rule of Figure 3.2 maps to an iterative structure in the corresponding Objective C function.

Transformation of the head of a MSL rule or fact to an Objective C function declaration is fairly straightforward. The MSL functor becomes the Objective C function name and the MSL formal argument list becomes the Objective C formal parameter list. The greatest difficulty encountered during this transformation occurs when generating typed Objective C formal parameter declarations from untyped MSL formal arguments. A similar problem to this occurs later when generating local variable declarations for the Objective C functions.

One of the major problems encountered when transforming a polymorphic language such as MSL to a monomorphic language such as Objective C is the generation of type declarations [38]. There are two potential solutions to this problem. The first is to restrict the values which can be assigned to a polymorphic variable to a single type as is done in [38], and the second is to create a monomorphic variable (such as a pointer) which can accept values of any type. Objective C lends itself to

```

BOOL Ada_subprogram_specification( id A )
{
    BOOL Success;
    id B;
    print_literal( "PROCEDURE " );
    Success = Ada_identifier( A );
    if ( Success )
    {
        get_formal_parameters( A, &B );
        Success = is_empty( B );
        if ( ! Success )
        {
            print_literal( "(" );
            Success = Ada_formal_parameter_list( B );
            if ( Success )
            {
                print_literal( ")" );
            }
        }
    }
    return Success;
}

```

Figure 3.3: Objective C code generated by MetaGen.

the second solution because it is a weakly typed language and it supports a special type, type `id`, which describes all Objective C objects. As most variables in the code generators reference Objective C objects in the TimeBench Design Tree, this solution is feasible. The few items, such as string and numeric constants, which are not Objective C objects can be forced into variables of type `id` because of the language's weak typing.

If all the variables of type `id` in the formal parameter list of Objective C function declarations actually contained pointers to objects then there would be no need to worry about argument modes in MSL since any modifications to the objects would be visible inside and outside the function. However, sometimes the values passed to formal parameters are only type cast to type `id`. In this case, modifications to

```

BOOL Ada_formal_parameter_list( id A )
{
    BOOL Success;
    id C;
    id B;
    LOOP
    {
        Success = extract_element( A, &B, &C );
        if ( Success )
        {
            Success = Ada_parameter_specification( B );
            if ( Success )
            {
                Success = is_empty( C );
                if ( Success )
                {
                    break;
                }
                if ( ! Success )
                {
                    print_literal( "; " );
                    Success = YES;
                    A = C;
                }
            }
        }
        if ( ! Success )
        {
            break;
        }
    }
    return Success;
}

```

Figure 3.4: Objective C code generated by MetaGen containing tail recursion optimization.

the value contained within the formal parameter may not be visible outside the function. As there is no way to tell when the first case or second case applies, all the formal parameters must explicitly be passed in or out of the function. To do this, two Objective C types are provided: type *id* and type *id **. These two types correspond to mode *in* and mode *out* in MSL, respectively.

Fundamental expressions in the body of the rule are transformed into Objective C function calls. The functor in the fundamental expression becomes the Objective C function name, and the actual argument list in the fundamental expression becomes the actual argument list for the function call. A fairly complex algorithm is required to convert MSL actual arguments to Objective C actual arguments. The algorithm is summarized in Table 3.1.

The first column of the table contains a list of very simple MSL fundamental expressions. The actual argument contained inside each fundamental expression is either a constant or a variable. If the actual argument is a variable, the second column of the table describes where the variable is declared. The variable is either a local variable declaration or is declared in the formal argument list of the rule being processed. The third column describes the head of the MSL rule matching the fundamental expression. The purpose of this column is to describe the argument mode of the formal argument corresponding to the actual argument. The final column shows the Objective C code which is generated when the conditions in the first three columns are satisfied.

The logical constructs in MSL are converted into "if statements" in Objective C. The predefined boolean variable **Success** controls the execution of these statements. The mapping of two MSL expressions separated by an "," construct to an equivalent Objective C control sequence is

$$expression1, expression2 \Rightarrow \begin{array}{l} expression1 \\ \text{if (Success) \{ } \\ \quad expression2 \\ \} \end{array}$$

This transformation has been optimized so that built-in rules which always succeed (such as `get_formal_parameters`) are not followed by a test for success (see Figure 3.3). The mapping of two MSL expressions separated by an "|" construct to an

MSL Fundamental Expression	Source of Actual Argument	Head of MSL Rule	Objective C Function Call
F(Constant)	N/A	F(+FormalArg)	F(Constant);
F(Constant)	N/A	F(-FormalArg)	Error
F(Variable)	Local variable declaration	F(+FormalArg)	F(Variable);
F(Variable)	Local variable declaration	F(-FormalArg)	F(&Variable);
F(Variable)	Formal parameter declaration with mode <i>in</i>	F(+FormalArg)	F(Variable);
F(Variable)	Formal parameter declaration with mode <i>in</i>	F(-FormalArg)	Error
F(Variable)	Formal parameter declaration with mode <i>out</i>	F(+FormalArg)	F(*Variable);
F(Variable)	Formal parameter declaration with mode <i>out</i>	F(-FormalArg)	F(&TempVar); *Variable = TempVar;

Table 3.1: Transformation of MSL actual arguments to Objective C.

equivalent Objective C control sequence is

$$expression1|expression2 \Rightarrow \begin{array}{l} expression1 \\ \text{if (! Success) \{ } \\ \quad expression2 \\ \} \end{array}$$

Tail recursion optimization (see Figure 3.4) is performed on rules which have been flagged as tail recursive during the second pass. The tail recursive rule is transformed into an Objective C function containing an infinite loop which encloses the body of the function. The tail recursive fundamental expression simply becomes a reassignment of variables; the actual arguments in the fundamental expression are assigned to the formal arguments in the head of the rule. Finally, a “break statement” is inserted after the last fundamental expression in every non-recursive branch of the rule. The “break statement” causes an exit from the loop.

The transformation of the implicit body of a fact to an Objective C function body is trivial. The implicit body of a fact is **succeed**. The equivalent Objective C

structure is **return YES**.

During the processing of the actual arguments in the third pass, the names of the locally defined variables are stored in a table. The fourth pass uses this table to generate local variable declarations for each of the Objective C functions produced. All local variables are of type **id** (see above).

Chapter 4

The Ada Code Generator

4.1 Overview

The first code generator built for TimeBench using MetaGen transforms TimeBench designs into Ada programs. Ada was chosen as the first target language since Ada constructs closely parallel components of the TimeBench design notation.

- TimeBench designs are hierarchical; so are Ada programs. Thus, the scoping of components in TimeBench designs is similar to the scoping of constructs in Ada programs.
- Many of the structural components in the MachineChart notation can be mapped to similar Ada constructs.
- The concepts represented by many of the behavioural components in the MachineChart notation are easily represented in Ada.
- MCL is an Ada-like language. Many MCL statements have the same syntax as their Ada counterparts.

The most significant differences between the TimeBench design notation and the Ada language are ACM's and data types. Ada has no equivalent to the ACM and does not support the high level data types allowed in MCL. There are also numerous

other small differences which complicate the transformation of TimeBench designs to Ada programs.

To simplify the development of AdaGen some of the features of the "ideal" code generator discussed in Chapter 1 have been either left out or restricted. No attempt has been made to optimize the Ada code or to document the Ada program. Furthermore, although the Ada program is subdivided into packages and tasks according to the structure of the TimeBench design, all the Ada code generated by AdaGen is enclosed in a single package and is output to a single file.

It also is assumed that the TimeBench designs input to AdaGen are error free in that only valid combinations of TimeBench components are used, the designs are complete, and the syntax of MCL is adhered to. For the most part, this assumption is valid as the designs are passed through the TimeBench error checker before they are passed to AdaGen. However, there are two situations in which the user is responsible for insuring that AdaGen will not fail.

- Sometimes the TimeBench error checker generates warning messages as opposed to error messages when the design is not complete. If only warning messages appear, AdaGen can be invoked, but the consequences of running the design through AdaGen are unpredictable. Therefore, warnings which appear in the design should be fixed before AdaGen is invoked.
- If a TimeBench design component which is not yet supported by AdaGen is present in the design, code generation will fail. Thus, use of these components should be avoided.

Some rudimentary error handling is provided in the MetaGen libraries, but is not fully taken advantage of by AdaGen.

The feature which allows users to guide the generation of Ada code by annotating the design has been left out as well. To properly implement this feature, significant redesign of the logical interface to the TimeBench Design Tree as well as substantial changes to the MetaGen libraries are necessary. User guidance is needed because TimeBench designs often can be implemented more than one way; thus, the addition of this feature should be a high priority for future work on the code generation

facility. User annotations are discussed in more detail in Chapter 6.

Despite these simplifications, AdaGen can automatically transform significant TimeBench designs into equivalent Ada programs. The resulting Ada programs are executable, well structured, and fully formatted. Furthermore, complete support for the implementation of the MCL data types in Ada is provided.

The remaining two sections in this chapter discuss AdaGen in more detail. The first section deals with the transformation of TimeBench design components to Ada, while the last section describes AdaGen in some detail. It is assumed in the ensuing discussion that the reader is familiar with the Ada language. Comprehensive coverage of the Ada language can be found in Cohen's *Ada as a Second Language* [21].

4.2 The Transformation of TimeBench Designs to Ada Programs

The transformation rules used to transform TimeBench designs into Ada programs can be divided into five categories:

1. rules to transform the structural components in the MachineChart notation to equivalent Ada constructs;
2. rules to transform MCL statements to equivalent Ada statements;
3. rules to transform the behavioural components in the MachineChart notation to equivalent Ada constructs;
4. rules to transform ACM's to functionally equivalent Ada code; and
5. rules to transform MCL data types and expressions to functionally equivalent Ada abstract data types.

The first two categories contain the most straightforward transformation rules as most of the TimeBench design components in these categories have a one-to-one correspondence with equivalent Ada constructs. The third category contains more

MachineChart Components	Ada Constructs	Comments
Box	Package	One-to-one correspondence.
Actor	Task	One-to-one correspondence.
Reactor	Task	Engines in reactors are implicit. A special transformation rule is needed to map the implicit engine into an explicit task body.
Push Button	Procedure	One-to-one correspondence.
Gate Button	Task entry	Gate buttons are treated as separate entities in MachineCharts but are actually embedded inside of task bodies in Ada as Ada accept statements.
Data Store	Variable Declarations	One-to-one correspondence.
Engine in (a) Box	Package initialization	One-to-one correspondence.
(b) Actor	Task body	One-to-one correspondence.
(c) Push Button	Procedure body	One-to-one correspondence.
(d) Gate Button	Accept alternative body	In-rendezvous and post-rendezvous MCL code must be dealt with separately.
Scroll	?	Not implemented - depending on context scroll may or may not map to an Ada construct.
Robot Pool	Array of tasks	Cannot implement without user annotations.

Table 4.1: Transformation of MachineChart structural components to Ada constructs.

complex transformation rules. Although the TimeBench design components found in the third category represent concepts which can be expressed easily in Ada, a significant amount of processing is necessary in order to gather all the information needed to generate the Ada code. The final two categories have no equivalent in Ada. The transformation rules in these categories generate special purpose Ada code to to perform the same function as the TimeBench design components.

4.2.1 The Structural Components in the MachineChart Notation

Shown in Table 4.1 is a summary of the transformations from structural components to equivalent Ada constructs. The first column in the table lists the TimeBench components; the second column lists the equivalent Ada constructs; and the last column briefly describes any special processing which must be performed in order to gather all the information needed to create the Ada construct.

The transformation rules not only perform the transformations summarized in Table 4.1, but also impose an order on the way the MachineChart structural components are processed so that Ada's stringent order of compilation rules [21] are satisfied. The general guidelines to ensure that the transformation rules process the children of MachineChart objects in the correct order are as follows: (1) convert all ACM's to special Ada package specifications; (2) convert all data stores to Ada variable declarations; (3) convert all boxes, actors, reactors, and push buttons to Ada package specifications, task specifications and subprogram specifications; (4) convert all ACM's, boxes, actors, reactors, and push buttons to Ada package bodies, task bodies, and subprogram bodies. Note that many of the MachineChart structural components are processed twice to achieve the correct ordering of Ada constructs. Depending on the context of the transformation rule (e.g., rules dealing with the children of a box, rules dealing with the children of an actor, etc.) some or all of the guidelines are enforced.

4.2.2 MCL Statements

A summary of the transformations from MCL statements to Ada statements is presented in Table 4.2. In many cases, MCL statements and Ada statements have the same syntax; therefore, the transformations are straightforward. However, a few MCL statements have no Ada equivalent or make direct use of MCL data types not supported by Ada; thus, the transformations are more complicated.

Unlike structural components, MCL statement objects have a specific order in the TimeBench Design Tree. To preserve the meaning of MCL statements during

MCL Statements	Ada Statements	Comments
Assignment	Assignment or call to a predefined procedure	If the MCL assignment statement involves data types not directly supported by Ada, a call to a predefined procedure replaces the assignment statement during the transformation.
Exit	Exit	One-to-one correspondence.
Null	Null	One-to-one correspondence.
Print	Call to a predefined procedure	The MCL print statement can print any MCL data type. Ada supports I/O of specific data types. Thus, the print statement is replaced with calls to predefined procedures.
Pull	Simple accept or selective wait	The information needed to generate the appropriate Ada statement is gathered from the pull finger named in the MCL pull statement.
Push	Procedure call or task entry call or timed task entry call or delay	The information needed to generate the appropriate Ada statement is gathered from the push finger named in the MCL push statement.
Return	Return	One-to-one correspondence.
If	If	One-to-one correspondence.
Case	Case	One-to-one correspondence.
Simple Loop	Simple Loop	One-to-one correspondence.
While Loop	While Loop	One-to-one correspondence.
For Loop	For Loop	MCL for loops which iterate over MCL data types not directly supported by Ada require special handling.
Random	Predefined random function	A special library function has been created to support this transformation.

Table 4.2: Transformation of MCL statements to Ada statements.

the transformation, the transformation rules maintain this order of MCL statement objects.

4.2.3 The Behavioural Components of the MachineChart Notation

The transformations from behavioural components to Ada constructs is summarized in Table 4.3. The MachineChart behavioural components map to one or more Ada constructs; therefore, a considerable amount of processing must be carried out by each transformation rule in order to determine the context of the behavioural component. Furthermore, often more than one design component must be examined in order to generate the resulting Ada code.

4.2.4 ACM's

ACM's cannot be represented directly in Ada; however, an Ada package can be generated which is functionally equivalent to an ACM. The general format of this package is outlined in Figure 4.1.

The information contained in the angle brackets in Figure 4.1 is specific to the ACM being processed. The name of the package is derived from the name of the ACM in the TimeBench design. The event list is a list of all the valid events for the state machine contained inside the ACM. The valid actions which can be returned from the state machine are listed in the action list. The valid states are listed in the state list.

The procedure **DecodeEvent** mimics the ACM's state machine. A case statement enumerating all the states in the state machine is used to control state transitions. Associated with every case statement alternative are a number of "if" statements which are generated from the information stored in the state machine's transition arcs. One block of "if" statements is generated for every transition arc leaving the state named in the case alternative.

The transition arc selected depends on the state named in **CurStateVar**, the event from the event list passed to the ACM, and any special conditions located on the transition arcs leaving the state. The activation of the transition arc causes local actions to be performed, the **CurStateVar** to be updated with the name of the state located at the destination of the transition arc and a return action from the

MachineChart Components	Ada Constructs	Comments
Push finger to (a) Push Button	Procedure call	The button's name becomes the procedure's name and the data flows form the procedure's parameter list.
(b) Gate Button	Task entry call	The gate's name becomes the entry's name and the data flows form the entry's parameter list.
(c) Timer	Delay statement	The period of the delay is retrieved from the timer.
Pull finger to (a) Choice Frame	Selective accept statement	The alternatives for the selective accept statement are retrieved from the from the gate buttons attached to the choice frame.
(b) Gate Button	Simple accept statement	The code for the accept statement is retrieved from the gate button's engine
Install arc	?	Not implemented - depending on the context the arc may or may not map to an Ada construct.
Remove arc	?	Not implemented - depending on the context the arc may or may not map to Ada construct.
Jab finger	No equivalent	Not implemented.
Data flow	Actual parameter	One-to-one correspondence.
Waiting place	Guarded entry in Ada task	Need to specially declare a boolean variable in Ada to represent the waiting place. Also need to generate the guard condition for the Ada accept alternative when a waiting place is detected.
Timer attached to (a) Push finger	Timed task entry call	The destination of the push finger must be a gate button. Generates a call to an Ada entry which times out after a certain period of time.
(b) Pull finger	Delay alternative in selective accept statement	Produces an Ada selective accept statement which times out after a certain period of time.

Table 4.3: Transformation of MachineChart behavioural components to Ada constructs.

action list to be passed out of the ACM.

4.2.5 MCL Data Types and Expressions

The one area in which the current version of MCL differs significantly from Ada is in that of data types and expressions. MCL supports four simple data types: integer, real, boolean, and string; and two compound data types: lists and sets. The Ada types created to support the transformation of the MCL types are `MCLInteger`, `MCLReal`, `MCLBoolean`, `MCLString`, `MCLList`, and `MCLSet`, respectively.

The Ada types, `MCLInteger`, `MCLReal` and `MCLBoolean`, are defined in terms of the predefined Ada types, `Integer`, `floating point`, and `Boolean`. These types have been renamed in order to emphasize that the MCL types have different semantics than the predefined Ada types. The Ada type, `MCLString`, is created from the standard string type supplied with most Ada compilers [21]. The standard string type cannot be used directly as it is static whereas the MCL string type is dynamic (i.e., it can expand and shrink when necessary). As Ada does not support list types and set types directly, the types `MCLList` and `MCLSet` are created indirectly using basic Ada types.

Nearly all the MCL integer, real, and boolean expressions are syntactically identical to Ada integer, floating point, and boolean expressions. The Ada types, `MCLInteger`, `MCLReal`, and `MCLBoolean` inherit all the operations which can be performed on their predefined counterparts because of the way they have been declared. Thus, with a few exceptions, the transformation of these MCL expressions to Ada expressions is straightforward.

The exceptions occur when dealing with MCL integer and real expressions. A few of the MCL integer and real expressions have no Ada equivalent. Thus, these MCL expressions are defined by Ada functions. Furthermore, MCL integer and real predicate expressions are defined by Ada functions even though the operators used in both languages are the same. This is because the operators in Ada return a value of type `Boolean` as opposed to type `MCLBoolean` which causes typing conflicts during compilation of the Ada program due to Ada's strong typing rules [21]. The remaining MCL expressions have no Ada equivalents. Therefore, all these MCL


```

PACKAGE <ACMName> IS
  TYPE EVENT_TYPE IS (<event list>);
  TYPE ACTION_TYPE IS (IMPLICIT_ERROR,<action list>);
  PROCEDURE DecodeEvent(CurEventVar : IN EVENT_TYPE;
                        CurActionVar : OUT ACTION_TYPE;
                        <other formal variable declarations>);
END <ACMName>;

PACKAGE BODY <ACMName> IS
  TYPE STATE_TYPE IS (<state list>);
  CurStateVar : STATE_TYPE := <start state>;

  PROCEDURE DecodeEvent(CurEventVar : IN EVENT_TYPE;
                        CurActionVar : OUT ACTION_TYPE;
                        <other formal variable declarations>) IS
  BEGIN
    CASE CurStateVar IS
      WHEN <state type 1> =>
        IF CurEventVar = <event type 1> THEN
          IF <transition arc condition> THEN
            <transition arc action code>
            CurStateVar := <new state type>;
            CurActionVar := <transition arc return action>
            RETURN;
          END IF;
        END IF;
        IF CurEventVar = <event type 2> THEN
          etc.
          :
        WHEN <state type 2> =>
          etc.
          :
    END CASE;
    CurActionVar := IMPLICIT_ERROR;
  END DecodeEvent;
END <ACMName>;

```

Figure 4.1: Skeletal Ada package used to represent ACM's in TimeBench.

expressions are defined in terms of Ada functions and procedures.

Rather than generating all the code necessary to support MCL data types and expressions, a permanent Ada library has been created based on Booch's techniques for developing reusable software components [8]. The library is composed of six generic packages, one for each MCL type. The generic packages fall into two categories: the first simply defines additional functions which can be performed on data types which have been defined outside the package; and the second defines abstract data types. The packages supporting the MCL integer, real, and boolean types fall in the first category; the packages supporting MCL string, list, and set types fall in the second category. The type declarations and many of the Ada subprograms found in the generic packages belonging to the second category have been extracted from the examples found in Booch's *Software Components With Ada: Structures, Tools, and Subsystems* [8]. To better illustrate the contents of the Ada library, one generic package specification from each category of generic packages can be found in Appendix E.

The Ada code which instantiates the generic packages is automatically generated by AdaGen. Instantiation of generic packages in the first category must be preceded by a type declaration; whereas, instantiation of generic packages in the second category effectively declares the MCL type. The generic package instantiations are located near the beginning of the Ada program so support for MCL data types and expressions will be visible throughout the program.

Only one generic package instantiation is needed for each simple MCL type. Because only one instantiation is needed, the generic packages could have just as easily been implemented as packages. Generic packages have been used since future versions of AdaGen will permit the user to choose data types other than the default data type used by the code generator (e.g., Integer RANGE 1..10 as opposed to MCLInteger).

Compound MCL types are more complicated as there can be infinitely many types of lists and sets (e.g., list of integer, list of list of integer, etc.). One generic instantiation is needed for each type of list and set. In order to determine which generic instantiations are needed for lists and sets in an Ada program, all the variable

declarations contained in the TimeBench design are examined.

The final problem that arises when transforming MCL expressions to Ada expressions deals with type conversion. MCL is a weakly typed language whereas Ada is strongly typed. AdaGen only detects explicit type conversions in MCL expressions. Implicit type conversions are flagged by the Ada compiler.

4.3 Using AdaGen

The primary purpose of developing AdaGen is to show that a comprehensive set of rules for transforming TimeBench designs to Ada programs can be defined using MetaGen. To this end, AdaGen has been tested on a variety of TimeBench designs: the control flag design introduced in Chapter 2, the five dining philosophers design, the producer/consumer design, a design which tests most aspects of MCL, and a reasonably large communication system design.

The only major problem which may be encountered when transforming TimeBench designs to Ada programs is due to differences in the operational semantics of TimeBench and Ada. In TimeBench, actors are not scheduled to run until the entire design is initialized. However, in Ada, tasks can begin executing immediately after they are elaborated (which may occur before the entire program is initialized). This may lead to situations where tasks which are elaborated early in automatically generated Ada programs try to rendezvous with tasks that have not yet been elaborated and cause the Ada program to crash. For the time being, the only way to resolve this problem is to draw TimeBench designs in such a way that the resulting Ada tasks cannot rendezvous with other Ada tasks until the Ada programs which have been generated from the designs are completely initialized.

AdaGen successfully generates fully formatted and fully executable Ada programs from each of the TimeBench designs listed above. The only hand editing that needs to be performed on the resulting Ada code applies to the communication system design where the delay time must be changed to a value which suits the Ada environment. Furthermore, AdaGen generates Ada programs very quickly. The communication system design which produces approximately 750 lines of for-

Figure 4.2: A Timer Component in TimeBench.

```

TASK TYPE timerType IS
    ENTRY start;
    ENTRY stop;
END timerType;
timer : timerType;

TASK BODY timerType IS

    OnOrOff : MCLBoolean;

    PROCEDURE control(OnOrOff : IN OUT MCLBoolean);

    PROCEDURE control(OnOrOff : IN OUT MCLBoolean) IS
    BEGIN
        IF OnOrOff THEN
            OnOrOff := False;
            comm1.timeout;
        END IF;
    END control;

BEGIN
    OnOrOff := False;
    LOOP
        SELECT
            ACCEPT start DO
                OnOrOff := True;
            END start;
        OR
            ACCEPT stop DO
                OnOrOff := False;
            END stop;
        OR
            DELAY Duration(100.000000);
            control(OnOrOff => OnOrOff);
        END SELECT;
    END LOOP;
END timerType;

```

Figure 4.3: Ada code generated from the information contained in actor `timer`.

```

task_specification(RobotNode) -->
  {is_robot_pool(RobotNode),
   print_error("Robot pools cannot be transformed.")} |
  ({print_literal("TASK TYPE ")} ,
   identifier(RobotNode),
   {print_literal("Type IS"), print_newline,
    get_external_children(RobotNode,ChildList),
    get_all_gates(ChildList,GateNodes)},
   {is_empty(GateNodes)} |
   ({indent(3)},
    entry_declaration_list(GateNodes),
    {indent(-3)}}),
   {print_literal("END ")} , identifier(RobotNode),
   {print_literal("Type;"), print_newline},
   identifier(RobotNode), {print_literal(" : ")} ,
   identifier(RobotNode), {print_literal("Type;"),
   print_newline, print_newline,
   free_list(ChildList)}}).

```

Figure 4.4: Transformation rule for creating Ada task specifications.

matted Ada code is transformed in much less than 5 seconds on a SUN/SPARC1 workstation.

To demonstrate the functionality of AdaGen, the remainder of this section describes the transformation of two small TimeBench components to Ada code fragments. The first component is taken from the communication system design and serves to introduce the reader to the wide variety of transformation rules used in AdaGen, while the second component is taken from the producer/consumer design and introduces the reader to some of the transformation rules used to transform MCL list expressions to equivalent Ada expressions. A comprehensive example can be found in Appendices B, C and D. Appendix B contains selected portions of the communication system design, Appendix C contains the corresponding Ada code, and Appendix D contains a listing of the transformation rules used in the example.

```

task_body(RobotNode) -->
  {print_literal("TASK BODY ")}, identifier(RobotNode),
  {print_literal("Type IS"), print_newline, print_newline,
   indent(3),
   get_all_children(RobotNode,ChildList)},
  declarative_part(ChildList),
  {indent(-3),
   print_literal("BEGIN"), print_newline,
   indent(3)},
  ({get_engine_node(ChildList,EngineNode)},
   code_block(EngineNode)) |
  {print_literal("NULL;"), print_newline}},
  {indent(-3),
   print_literal("END ")}, identifier(RobotNode),
  {print_literal("Type;"), print_newline, print_newline,
   free_list(ChildList)}.

```

Figure 4.5: Transformation rule for creating Ada task bodies.

```

assignment_stmt(MCLStmt) -->
  {is_assignment_stmt(MCLStmt),
   get_lhs(MCLStmt,LHS),
   get_rhs(MCLStmt,RHS)},
  ({(is_slice(LHS)), slice_assignment(LHS,RHS)) |
   ({is_index_component(LHS)), index_assignment(LHS,RHS)) |
   ({is_var_name(RHS)), copy_assignment(LHS,RHS)) |
   (simple_var(LHS), {print_literal(" := ")},
    expression(RHS), {print_literal(";"), print_newline})}.

```

Figure 4.6: Transformation rule for generating Ada assignment statements.

Timer Component

Shown in Figure 4.2 is the timer component taken from the communication system design. The actor in the upper left hand corner of the figure represents the overall structure of the timer component, while the design components in the main window represent the internal details of the timer component.

```

loop_stmt(MCLStmt) -->
  ({is_simple_loop_stmt(MCLStmt)},
   {print_literal("LOOP"), print_newline,
    indent(3),
    get_code_block(MCLStmt,CodeBlock)},
   sequence_of_statements(CodeBlock),
   {indent(-3),
    print_literal("END LOOP;"), print_newline}) |
  ({is_while_loop_stmt(MCLStmt),
   get_iteration_scheme(MCLStmt,Condition)},
   {print_literal("WHILE ")}, expression(Condition),
   {print_literal(" LOOP"), print_newline,
    indent(3),
    get_code_block(MCLStmt,CodeBlock)},
   sequence_of_statements(CodeBlock),
   {indent(-3),
    print_literal("END LOOP;"), print_newline}) |
  ({is_for_loop_stmt(MCLStmt)},
   process_for_loop(MCLStmt)).

```

Figure 4.7: Transformation rule for generating Ada loop statements.

The timer component functions as follows. When the timer accepts a rendezvous at gate button **start**, the timer is turned on. Unless the timer accepts a rendezvous at gate button **stop** before the time out located on pull arc **recv** is activated, push button **control** is activated. When the timer is on, push button **control** activates push finger **timeout** which causes a time out to occur in the communication system. Time outs which occur when the timer is not turned on are ignored.

The Ada code generated from the timer component is shown in Figure 4.3. A sampling of the AdaGen rules used to generate the Ada code are found in Figures 4.4 to 4.10. The rule contained in Figure 4.4 generates the task specification for actor **timer**. The expression `print_literal/1` which appears throughout the rule is used to output Ada keywords and punctuation, while the expression `identifier/1` outputs the name of the actor. To generate the entry declarations for the task, the rule first invokes `get_external_children/2` and then `get_all_gates/2` to retrieve


```

select_stmt(PullArc) -->
  (({is_timed(PullArc)},
    ({is_choice(PullArc),
      get_all_destinations(PullArc,Gates),
      get_pull_timer(PullArc,Timer),
      print_literal("SELECT"), print_newline,
      indent(3)},
      accept_alternative_list(Gates),
      {indent(-3),
        print_literal("OR"), print_newline,
        indent(3)},
      delay_alternative(Timer),
      {indent(-3),
        print_literal("END SELECT;"), print_newline})) |
    ({get_destination(PullArc,Gate),
      get_pull_timer(PullArc,Timer)},
      {print_literal("SELECT"), print_newline,
      indent(3)},
      accept_alternative(Gate),
      {indent(-3),
        print_literal("OR"), print_newline,
        indent(3)},
      delay_alternative(Timer),
      {indent(-3),
        print_literal("END SELECT;"), print_newline}})) |
  ({is_choice(PullArc),
    get_all_destinations(PullArc,Gates)},
    {print_literal("SELECT"), print_newline,
    indent(3)},
    accept_alternative_list(Gates),
    {indent(-3),
      print_literal("END SELECT;"), print_newline})).

```

Figure 4.8: Transformation rule for generating Ada select statements.

```

object_declaration(DataNode) -->
  identifier(DataNode),
  {print_literal(" : "),
   get_type_node(DataNode,TypeNode)},
  subtype_indication(TypeNode),
  (({get_initial_expression(DataNode,InitExpression),
    print_literal(" := ")}),
   expression(InitExpression),
   {print_literal(";"), print_newline, print_newline}) |
   {print_literal(";"), print_newline, print_newline}).

```

Figure 4.9: Transformation rule for generating Ada variable declarations.

```

subprogram_body(ButtonNode) -->
  subprogram_specification(ButtonNode),
  {print_literal(" IS "), print_newline,
   indent(3),
   get_all_children(ButtonNode,ChildList)},
  declarative_part(ChildList),
  {indent(-3),
   print_literal("BEGIN"), print_newline,
   indent(3)},
  (({get_engine_node(ChildList,EngineNode)},
   code_block(EngineNode)) |
   {print_literal("NULL;"), print_newline}),
  {indent(-3),
   print_literal("END ")}, identifier(ButtonNode),
  {print_literal(";"), print_newline, print_newline,
   free_list(ChildList)}.

```

Figure 4.10: Transformation rule for generating Ada procedures.

the two gate buttons associated with actor **timer**. The entry declarations are then generated by invoking the rule **entry_declaration_list/1**.

The rule used to generate the task body for the actor is shown in Figure 4.5. This rule first invokes **get_all_children/2** which fetches all the children of actor **timer** and then invokes **declarative_part/1** which causes the children to be processed

according to the guidelines outlined in Section 4.2.1. In this example, an Ada data declaration, an Ada procedure specification and an Ada procedure body are generated from data store `onOrOff` and push button `control` when `declarative_part/1` is invoked.

Next, using `get_engine_node/2`, engine `timer` is extracted from the list of children. The contents of the engine are processed by invoking `code_block/1`. If no engine had been present in the actor's list of children, a "null" statement would have been generated instead.

The engine `timer` contains a MCL code fragment consisting of three MCL statements: an assignment statement, a loop statement and a pull statement. The MCL assignment statement is transformed into an Ada assignment statement using the rule in Figure 4.6. The first expression, `is_assignment_stmt/1` tests to see if the MCL statement passed to the rule is an assignment statement. Tests similar to this one are present in all the rules which transform MCL statements to Ada statements to ensure that the appropriate rule is invoked when processing a list of MCL statements. When invoked, the rules `get_lhs/2` and `get_rhs/2` fetch MCL expressions from the left and right hand side of the assignment operator, respectively. Because MCL assignment statements often deal with MCL expressions which cannot be transformed directly into Ada, the remainder of the rule tests for these cases. The last option in the series of tests is the direct transformation and is the case that applies to the MCL assignment statement located in engine `timer`.

The rule in Figure 4.7 transforms MCL loop statements into Ada loop statements. Three types of loops are supported in MCL: the simple loop, the while loop and the for loop. This is reflected by the three options contained in the transformation rule. As engine `timer` contains a simple loop, only the first option will be dealt with here. The transformation of the simple loop is straightforward. The rule generates the Ada keywords needed for the Ada simple loop and then processes the MCL statements contained inside the loop by invoking the rules `get_code_block/2` and `sequence_of_statements/1`.

The rule for transforming the MCL pull statement in engine `timer` into an Ada select statement is quite complex (see Figure 4.8) as not only does pull finger `recv`

(which is named in the pull statement) have to be retrieved, but also the context of the pull finger has to be determined. There are three separate contexts handled by this transformation rule: a choice frame and a time out attached to the pull finger, a time out and a gate button attached to the pull finger, and a choice frame attached to the pull finger. This example satisfies the first case as both the tests `is_timed/1` and `is_choice/1` succeed.

To generate the Ada select statement for this example, the gate buttons attached to choice frame `x` are retrieved using the rule `get_all_destinations/2` and the time out is retrieved using the rule `get_pull_timer/2`. The accept alternatives are generated by invoking the rule `accept_alternative_list/1`. The body of each accept statement is generated from the MCL code contained inside the engine of the gate button used to produce the accept statement. The delay alternative is generated by invoking the rule `delay_alternative/1`. The Ada delay statement in the delay alternative body is generated from the time out and the Ada procedure call is generated from the push finger between the time out and push button `control`.

The declaration of variable `OnOrOff` in Figure 4.3 is generated from the MCL data declaration contained inside store `onOrOff` using the transformation rule in Figure 4.9. The name of the variable is derived from the name of the data node used to store the MCL data declaration in the TimeBench Design Tree. The variable's type is generated by retrieving the type node associated with the data node using `get_type_node/2` and then processing the type node using the rule `subtype_indication/1`. Because the declaration of `OnOrOff` does not have an initial expression associated with it, the section of the rule starting with the expression `get_initial_expression/2` is not executed.

The rule used to transform push button `control` into an Ada procedure is found in Figure 4.10. The procedure's name and formal parameter list are generated when the expression `subprogram_specification/1` is evaluated. The invocation of the rule `declarative_part/1` has no effect on the Ada code as the only child of push button `control` is an engine. The sequence of statements inside the Ada procedure are generated from the MCL statements contained inside push button `control`'s engine using the rule `code_block/1`.

Figure 4.11: A Buffer Component in TimeBench.

```

TASK TYPE BufferType IS
    ENTRY Put(inVal : IN MCLInteger);
    ENTRY Get(outVal : OUT MCLInteger);
END BufferType;
Buffer : BufferType;

TASK BODY BufferType IS

    bufVal : MCLListOfInteger.List;
    notFull : MCLBoolean;
    notEmpty : MCLBoolean;

BEGIN
    notEmpty := False;
    notFull := True;
    bufVal := Construct;
    LOOP
        SELECT
            WHEN notEmpty = True =>
                ACCEPT Get(outVal : OUT MCLInteger) DO
                    outVal := Get_Item(bufVal,1);
                    IF Is_Equal(Length_Of(bufVal),1) THEN
                        bufVal := Construct;
                        notEmpty := False;
                    ELSE
                        bufVal := SubPart_Of(bufVal,2,
                                                Length_Of(bufVal));
                        notFull := True;
                    END IF;
                    Output(Initialize("After GET buffer contains:"));
                    Output(bufVal);
                    New_Line;
                END Get;
            OR

```

Figure 4.12: Ada code generated from the information contained in actor **buffer**.

```

    WHEN notFull = True =>
        ACCEPT Put(inVal : IN MCLInteger) DO
            Modify(bufVal,(Length_Of(bufVal) + 1),
                Length_Of(bufVal),
                Construct(inVal,Construct));
            notEmpty := True;
            IF Greater_Than_Equal(Length_Of(bufVal),5) THEN
                notFull := False;
            END IF;
            Output(Initialize("After PUT buffer contains:"));
            Output(bufVal);
            New_Line;
        END Put;
    END SELECT;
END LOOP;
END BufferType;

```

Figure 4.12: Continued ...

```

slice_assignment(LHS,RHS) -->
    {get_lhs(LHS,VarName),
     get_rhs(LHS,Range),
     get_lhs(Range,LowerLimit),
     get_rhs(Range,UpperLimit),
     print_literal("Modify(")}, identifier(VarName),
    {print_literal(",")}, expression(LowerLimit),
    {print_literal(",")}, expression(UpperLimit),
    {print_literal(",")}, expression(RHS),
    {print_literal(");"), print_newline}.

```

Figure 4.13: Transformation rule for generating an Ada procedure call from string and list slice assignments in MCL.

Buffer Component

Shown in Figure 4.11 is the buffer component taken from the producer/consumer design. The Ada code generated from the buffer component is found in Figure 4.12. The buffer is implemented as a list of integers with a maximum size of five. As long

```

length_operator(Operator, Expression) -->
  {is_length_operator(Operator)},
  print_literal("Length_Of("),
  expression(Expression),
  {print_literal(")").

```

Figure 4.14: Transformation rule for generating an Ada expression which determines the lengths of MCL strings, lists, or sets.

as there is room in the buffer, rendezvous can occur at gate button **Put** which causes an integer to be appended to the end of the list. When the buffer becomes full, the waiting place **notFull** is closed and rendezvous cannot take place at gate button **Put** until at least one integer is removed from the buffer. As long as the buffer is not empty, rendezvous can occur at gate button **Get**. When the buffer becomes empty, the waiting place **notEmpty** is closed until at least one integer is put in the buffer.

Many of the transformations that take place in this example are similar to the transformations already discussed in the previous example since both the timer component and buffer component are actors. The primary purpose of this example is to show how MCL list expressions are transformed to equivalent Ada expressions.

The mappings of the MCL list expressions used in the buffer component to Ada procedures and functions are as follows:

```

bufVal(low .. high) := list;  => Modify(bufVal, low, high, list);
#bufVal               => Length_Of(bufVal)
[]                    => Construct
[head|tail]           => Construct(head, tail)
bufVal(position)      => Get_Item(bufVal, position)
bufVal(low .. high)  => SubPart_Of(bufVal, low, high)

```

The rules which transform the first two expressions are found in Figures 4.13 and 4.14, respectively. The rule **slice_assignment/2** (Figure 4.13) is passed the expressions on the left hand side and the right hand side of the assignment operator. The left hand side expression is a slice expression which after a series of calls to **get_lhs/2** and **get_rhs/2** can be broken down into its individual components: the variable's name, the lower bound and the upper bound. Using these components

and the right hand side expression originally passed to the rule, the procedure call is generated. Because Ada allows overloading of subprogram names, the same name can be used for the procedure which deals with MCL string slices and the procedure which deals with MCL list slices.

The rule `length_operator/2` (Figure 4.14) deals with unary MCL expressions. The first expression in the rule `is_length_operator/1` ensures that this rule is invoked only when an expression containing the length operator is passed to it. The remainder of the rule generates the `Length_Of` function call. Once again the same name can be used for functions dealing with strings, lists or sets because of overloading in Ada.

The rules for the remaining expressions listed above are very similar to the rule `length_operator/2`. The first expression in each of the rules is a test to ensure the correct rule is being invoked and the remaining expressions generate the appropriate Ada function call.

Chapter 5

The C/JNX Code Generator

5.1 Overview

The second code generator (*C-Gen*) developed for TimeBench using MetaGen produces C/JNX implementations from TimeBench Designs. JNX is a real time operating system which provides the necessary functionality for real time programming in C. JNX “supports the creation and management of concurrent processes (tasks), and provides objects which allow those tasks to synchronize and communicate with each other” [15].

C/JNX has been chosen as a target implementation in order to test the versatility of MetaGen. First, the development of C-Gen illustrates the generality of the meta-generator approach. Although the transformation rules written for C-Gen differ significantly from those written for AdaGen, MSL and the MetaGen libraries are sufficiently general for use in C-Gen’s specification. Second, the development of C-Gen illustrates the definition of a code generator for a target language whose constructs do not closely parallel the TimeBench design notation. Components of the TimeBench design notation are rarely directly related to the C/JNX constructs although most of the concepts represented by the TimeBench design notation can also be represented using C/JNX constructs.

The same simplifications which have been applied to AdaGen have also been applied to C-Gen. However, the lack of multifile output and user annotations prove

to be more restrictive for C-Gen because C is not a modular language and there are significant scoping differences between TimeBench designs and their resulting C/JNX implementations.

The development of C-Gen progressed quickly as many of the ideas and transformation techniques used in AdaGen could be reused in C-Gen. As with AdaGen, C-Gen is capable of transforming significant TimeBench designs. However, there are a few gaps in the transformation rules and although designs which contain lists and sets can be transformed, they will not execute since the underlying C library functions to support these types are not complete.

The remainder of the chapter follows the same format as Chapter 4 for the purposes of comparison. It is assumed that the reader has both an understanding of C and of real time operating systems. The classic reference for the C language is Kernighan and Ritchies's *The C Programming Language* [27], while information on JNX can be found in *Design and Programming Concurrent, Real-Time Systems with Turbo C* by Buhr et al [15].

5.2 The Transformation of TimeBench Designs to C/JNX Implementations

5.2.1 The Structural Components in the MachineChart Notation

The transformations from MachineChart structural components to C/JNX constructs are summarized in Table 5.1. The transformation rules used in C-Gen are not as straightforward as those used in AdaGen since the information in the TimeBench Design Tree often has to be rearranged to create the C/JNX constructs. However, like the rules in AdaGen, these transformation rules impose an order on the way the structural components are processed.

The guidelines for ordering the structural components are more complex in C-Gen since C functions cannot be nested inside one another and MachineChart structural components can. Thus, not only must the transformation rules ensure that

MachineChart Components	C/JNX Constructs	Comments
Box	No equivalent	Modularity of the TimeBench design is lost in the transformation to C/JNX.
Actor	Task	The information contained in the actor must be rearranged before the task can be generated.
Reactor	Task	The information contained in the reactor must be rearranged before the task can be generated. Furthermore, a special transformation rule is needed to map the implicit reactor engine into the JNX body.
Push Button	Function	One - to - one correspondence if the flattening algorithm is ignored.
Gate Button	Mailbox	The information contained inside gate buttons is actually embedded inside the JNX task body.
Data Store	Variable Declarations	Cannot transform the initialization of MCL string, list or set variables.
Engine in (a) Box	Function	Becomes a special function which is invoked at the start of the C/JNX program.
(b) Actor	Task body	One-to-one correspondence.
(c) Push Button	Function body	One-to-one correspondence.
(d) Gate Button	Mailbox event code	One-to-one correspondence.
Scroll	No equivalent	Not implemented.
Robot Pool	Array of tasks	Cannot implement without user annotations.

Table 5.1: Transformation of MachineChart structural components to C/JNX constructs.

the MachineChart structural components are processed in an order which produces C/JNX code that satisfies the C compiler, but also the transformation rules must flatten the MachineChart structural components. To do this, two passes are made through the upper portion of the TimeBench Design Tree. On the first pass through the design tree, C global declarations, C forward function declarations, and C/JNX

task specifications are generated from the structural components found in the design tree. On the second pass through, C function definitions are generated. In both passes, the TimeBench Design Tree is traversed in reverse, with C/JNX code being generated for all the children before it is generated for the parent.

The major problem which arises when the TimeBench design is flattened is that of scoping. This is best illustrated by use of an example. The design fragment in Figure 5.1 shows the internals of a push button: a data store and another push button labelled **Internal**. According to the TimeBench scoping rules, any variables named in the data store are visible inside push button **Internal**. The skeletal C/JNX code fragment generated from the design in Figure 5.1 is shown in Figure 5.2. Because of flattening the variable declarations are not visible to the function **Internal**. To get around this problem, the design has to be modified. One possible modification is shown in Figure 5.3. In this figure, the variables named in the data store and used by the push button **Internal** are passed to the push button as parameters.

Naming is another problem that arises when the TimeBench design is flattened. This problem is compounded by the fact that C/JNX does not enforce modular programming. Names which are unique in TimeBench designs may not be unique in the resulting C/JNX implementation. The best solution to this problem is to use a naming convention which prepends the name of a TimeBench structural component to each variable or nested TimeBench component named inside the component. The convention is not too hard to implement for nested TimeBench components (although it is awkward). However, without major changes to MetaGen, there is no way to distinguish which variable belongs to which component when transforming MCL code to C/JNX code. Thus, at the moment, unique names have to be assigned to every component in the TimeBench design to ensure that naming conflicts do not arise.

The final issue to contend with is that C/JNX does not actively enforce modular programming. Thus, MachineChart boxes cannot be easily implemented in C/JNX. Pseudo-modules can be written in C/JNX by separating the C/JNX code into different files. C/JNX header files can be treated as the module's specification, and the corresponding C/JNX files can be treated as the module's body. Unfortunately,

Figure 5.1: TimeBench design which will not translate to C/JNX properly.

```

void Internal
{
    etc.
    :
}

void External
{
    <variable declarations from data store>
    etc.
    :
}

```

Figure 5.2: Skeletal C/JNX code showing scoping changes.

MetaGen presently produces code generators which output implementation code to a single file; therefore, even pseudo-modules cannot be created. Thus, any modularity introduced into the TimeBench designs using boxes is lost in the C/JNX implementation.

5.2.2 MCL Statements

The transformation rules used to transform MCL statements in C-Gen are similar to those used in AdaGen. The transformations are summarized in Table 5.2.

MCL Statements	C/JNX Statements	Comments
Assignment	Assignment or call to a predefined function	If the MCL assignment statement involves data types not directly supported by C/JNX, a call to a predefined function replaces the assignment statement during the transformation.
Exit	Break	The information is rearranged.
Null	No equivalent	Generate nothing.
Print	Call to a predefined function	The MCL print statement can print any MCL data type. C/JNX supports I/O of specific data types. Thus, the print statement is replaced with calls to predefined procedures.
Pull	Waiting on a mailbox event	The information to generate the C/JNX code which performs this function is gathered from the pull finger named in the MCL pull statement.
Push	Function call or function call to send mail or function call to start timing semaphore	The information to generate any of these C/JNX function calls is gathered from the push finger named in the MCL push statement.
Return	Return	One-to-one correspondence.
If	If	One-to-one correspondence.
Case	If	The information is rearranged.
Simple loop	For loop with no parameters	One-to-one correspondence.
While loop	While loop	One-to-one correspondence.
For loop	For loop	MCL loops which iterate over MCL data types not directly supported by C/JNX require special handling. These for loops have not been implemented. Otherwise, information is simply rearranged to generate the C/JNX for loop.
Random	Random function	A special library function has been created to support this transformation.

Table 5.2: Transformation of MCL statements to C/JNX statements.

Figure 5.3: Revised TimeBench design to accommodate scoping in C/JNX

5.2.3 The Behavioural Components of the MachineChart Notation

Most of the concepts represented by the MachineChart behavioural components can also be represented by C/JNX constructs. However, like AdaGen, a significant amount of processing must be performed to generate all the information needed for the C/JNX construct. The transformations from MachineChart behavioural components to C/JNX are summarized in Table 5.3.

5.2.4 ACM's

The algorithm used to map ACM's to C/JNX code is very similar to the one used in AdaGen (see Figure 4.1). The only significant difference between the algorithms is the naming convention used. Since C/JNX does not have a structure equivalent to the Ada package, the global type declarations, the global variable declarations and the procedure declaration needed for the ACM are visible to the entire C/JNX program. Thus, unique names must be given to all. The naming convention used is to prepend the ACM name to the front of the variable, type, or procedure name (e.g., `ACMName_event_type`, `ACMName_action_type`, etc.).

MachineChart Components	C/JNX Constructs	Comments
Push finger to (a) Push Button	Function call	The button's name becomes the function's name and the data flows form the function call's parameter list
(b) Gate Button	Function call to send mail	Creates C/JNX code which sends mail to a mailbox having the same name as the gate button.
(c) Timer	Function call to start timing semaphore	Creates C/JNX code which causes the JNX task to block until the semaphore times out. In effect causes a delay.
Pull finger to (a) Choice Frame	Waiting on a mailbox event	Creates C/JNX code which waits for a rendezvous on one of the mailboxes owned by the task. The mailboxes correspond to the gate buttons on the actor which are attached to the choice frame.
(b) Gate Button	Waiting for mail	Creates C/JNX code which causes the task to wait for mail from the mailbox having the same name as the gate button.
Install arc	?	Not implemented – depending on the context the arc may or may not map to a C/JNX construct.
Remove arc	?	Not implemented – depending on the context the arc may or may not map to a C/JNX construct.
Jab finger	No equivalent	Not implemented.
Data flow	Actual parameter	Have to make sure the actual parameter list has the same order as the formal parameter list in the function declaration.
Waiting place	Masked mailbox events	Generates C/JNX code which masks or unmask mailbox events depending on the state of the waiting place.
Timer attached to (a) Push finger (b) Pull finger	No equivalent Time out event	Cannot be translated. Generates C/JNX code which can trap either mailbox events or time out events when the task is waiting for a rendezvous.

Table 5.3: Transformation of MachineChart behavioural components to C/JNX constructs.

5.2.5 MCL Data Types and Expressions

The method developed for supporting the transformation of MCL data types and expressions to C/JNX equivalents is very similar to the method developed for AdaGen. Six new data types are defined in C/JNX to represent the MCL data types. Three are based on the predefined types integer, float and boolean and three are created from C's predefined types. The types are named `MCLInteger`, `MCLReal`, `MCLBoolean`, `MCLString`, `MCLList` and `MCLSet` and are declared in a header file which must be included with C/JNX programs generated from any TimeBench designs.

Most MCL expressions involving integer, real or boolean values can be converted to C/JNX expressions directly because C is weakly typed and has a large selection of operators and predefined functions to choose from. However, a few of these MCL expressions have no equivalent in C/JNX and require special C/JNX functions to support them.

The MCL expressions involving string, set or list values have no equivalent in C/JNX and therefore require a complete set of special C/JNX functions to support them. C/JNX functions to support the MCL string expressions are not difficult to write as the functions only have to deal with one type. However, C/JNX functions which support MCL list and set expressions have to be able to deal with multiple types (e.g., list of integer, list of list of integer, etc.). Fortunately, it is possible to define one function for every MCL list or set expression because C is weakly typed. However, the functions have to deal with each list or set type explicitly (each type is dealt with implicitly in AdaGen since generic packages are used).

A library of special C functions, similar to the Ada library built for AdaGen, has been developed to support transformation of MCL data types and expressions. The library is not complete, but it does provide enough support to run a few major examples. The C type definitions and a sampling of the function declarations found in the library can be found in Appendix H.

```

void control(MCLBoolean *formal_OnOrOff);

/* ----- timer_type Task Declaration ----- */

void timer_type();
task_parameter_block
    timer_parameters = {timer_type, 1000, 3, 20, 1};
task_id timer;
mailbox_id start, stop;
flag_id recv_timer;
typedef enum {start_event, stop_event, recv_event} timer_events;

void control(MCLBoolean *formal_OnOrOff)
{
    MCLBoolean OnOrOff;

    OnOrOff = *formal_OnOrOff;
    if (OnOrOff) {
        OnOrOff = False;
        send_mail(timeout, NULL);
    }
    *formal_OnOrOff = OnOrOff;
}

/* ----- timer_type Task Body ----- */

void timer_type()
{
    MCLBoolean OnOrOff;
    mail_id mail;
    timer_events event;

    recv_timer = create_flag(timer, recv_event, false);
    OnOrOff = False;
    for ( ; ; ) {
        signal_flag_after_time(100.000000, recv_timer);
        event = await_any_event(&mail);
    }
}

```

Figure 5.4: C/JNX code generated from the information contained in actor `timer`.

```

switch(event) {
    case start_event:
        OnOrOff = True;
        reply(start);
        break;
    case stop_event:
        OnOrOff = False;
        reply(stop);
        break;
    case recv_event:
        control(&OnOrOff);
        break;
}
reset_timed_flag(recv_timer);
}
}

```

Figure 5.4: Continued ...

5.3 Using C-Gen

The primary purpose of developing C-Gen is to show that MetaGen can be used to produce code generators for a wide variety of target languages. Testing of C-Gen has focussed on this purpose and, therefore, is not as comprehensive as the testing of AdaGen. C-Gen has been tested on most of the TimeBench designs mentioned in the previous chapter: the control flag design, the five dining philosophers design, the producer/consumer design, and the communication system design.

C-Gen produces fully formatted and fully executable C/JNX code for a simplified version of the producers/consumers design (the buffer is a single integer as opposed to a list of integers) and for the communication system design. C-Gen generates fully compilable C/JNX code for the more complex version of the producer/consumer design; however, the code cannot be executed as the underlying C library supporting list expressions is not complete. The control flag design and the five dining philosophers design can be transformed to C/JNX programs without error, but no attempt has been made to compile the resulting programs.

Two significant problems are encountered when transforming TimeBench designs to C/JNX programs. The first is due to naming conflicts and scoping differences as discussed in Section 5.2.1. All designs which originally produce C/JNX programs that will not compile because they contain naming conflicts and/or scoping errors have to be reworked in order to generate error free C/JNX programs. The second problem is due to the task parameter list which is associated with every task (see Figure 5.4). The task parameter list contains parameters specifying the maximum number of mailboxes and flags that can belong to the task, the size of the stack allocated to the task, etc. C-Gen can only generate default values for these parameters which may not suit the implementation (i.e., the program may crash because stack space runs out). Therefore, if necessary, the task parameter values in the resulting C/JNX program must be altered manually.

To demonstrate the functionality of C-Gen and to emphasize the similarities and differences between C-Gen and AdaGen, the transformation of the two TimeBench components presented in Chapter 4 (see Figures 4.2 and 4.11) to C/JNX code is described in the rest of this section. A more comprehensive example can be found in Appendices B, F, and G with Appendix B containing selected portions of the communication system design, Appendix F containing the corresponding C/JNX code, and Appendix G containing a listing of the transformation rules used in the example.

Timer Component

The C/JNX code generated from the timer component in Figure 4.2 is shown in Figure 5.4. A sampling of the rules used to transform the component to the C/JNX code are shown in Figures 5.5 to 5.11. On the whole, the rules are more complex than the ones in Chapter 4 since the TimeBench design components are not as similar to C/JNX constructs as they are to Ada constructs.

The transformation rule used to generate the C/JNX task declaration from the actor `timer` is shown in Figure 5.5. The rule is quite complex since all declarations for multitasking and rendezvous in C/JNX must be dealt with explicitly. The rule first generates the forward function declaration and task parameter block declaration

```

task_declaration(RobotNode) -->
{is_robot_pool(RobotNode),
 print_error("Robot pools cannot be transformed.")} |
({print_literal("/* ----- "), identifier(RobotNode),
 {print_literal("_type Task Declaration ----- */"),
  print_newline, print_newline,
  print_literal("void ")}, identifier(RobotNode),
 {print_literal("_type();"), print_newline,
  print_literal("task_parameter_block ")},
 identifier(RobotNode),
 {print_literal("_parameters = {")},
 identifier(RobotNode),
 {print_literal("_type, 1000, 3, 20, 1;"),
  print_newline,
  print_literal("task_id ")},
 identifier(RobotNode),
 {print_literal(";"), print_newline,
  get_all_children(RobotNode, ChildList),
  get_all_gates(ChildList, GateNodes),
  get_all_gates(ChildList, GateNodes2),
  get_all_gates(ChildList, GateNodes3),
  get_all_gates(ChildList, GateNodes4),
  get_all_pull_arcs(ChildList, PullArcs),
  get_all_pull_arcs(ChildList, PullArcs2)}},
 ({is_empty(GateNodes)} |
  ({print_literal("mailbox_id ")},
   mailbox_declaration_list(GateNodes),
   {print_literal(";"), print_newline},
   flag_declaration_list(PullArcs),
   {print_literal("typedef enum {")},
   mailbox_event_declaration_list(GateNodes2),
   flag_event_declaration_list(PullArcs2),
   {print_literal("} ")}, identifier(RobotNode),
   {print_literal("_events;"), print_newline},
   mailbox_parameter_declarations(GateNodes3))),
 gate_declaration_list(GateNodes4)),
 {print_newline,
  free_list(ChildList)}}.

```

Figure 5.5: Transformation rule for creating C/JNX task declarations.

```

task_body(RobotNode) -->
{print_literal("/* ----- "), identifier(RobotNode),
{print_literal("_type Task Body ----- */"),
  print_newline, print_newline,
  print_literal("void "), identifier(RobotNode),
{print_literal("_type()"), print_newline,
  print_literal("{"), print_newline,
  indent(3),
  get_all_children(RobotNode, ChildList),
  get_all_gates(ChildList, GateNodes1),
  get_all_gates(ChildList, GateNodes3),
  get_all_gates(ChildList, GateNodes4),
  get_all_pull_arcs(ChildList, PullArcs),
  get_all_data_nodes(ChildList, DataNodes)},
  local_gate_object_list(GateNodes1),
  gate_formal_parm_declaration_list(GateNodes4),
  object_declaration_list(DataNodes),
  ({is_empty(GateNodes3)} |
    {print_literal("mail_id mail;"),
      print_newline, print_newline},
    identifier(RobotNode),
    {print_literal("_events event;"),
      print_newline, print_newline}),
  ({get_engine_node(ChildList, EngineNode)},
    local_declarations(EngineNode),
    {get_all_outgoing_arcs(EngineNode, Arcs)},
    task_call_declarations(Arcs),
    create_flags(RobotNode, PullArcs),
    fetch_MCL_code(EngineNode)) |
    {print_literal("/* NULL */"), print_newline}),
  {indent(-3),
    print_literal("}"), print_newline, print_newline,
    free_list(ChildList)}.

```

Figure 5.6: Transformation rule for creating C/JNX task bodies.

```

assignment_stmt(MCLstmt) -->
  {is_assignment_stmt(MCLstmt),
   get_lhs(MCLstmt,LHS),
   get_rhs(MCLstmt,RHS)},
  (({is_slice(LHS)}, slice_assignment(LHS,RHS)) |
   ({is_index_component(LHS)}, index_assignment(LHS,RHS)) |
   ({is_var_name(RHS)}, copy_assignment(LHS,RHS)) |
   (simple_var(LHS), {print_literal(" = ")}, expression(RHS),
    {print_literal(";"), print_newline})).

```

Figure 5.7: Transformation rule for generating C/JNX assignment statements.

```

loop_stmt(MCLstmt) -->
  ({is_simple_loop_stmt(MCLstmt),
   print_literal("for ( ; ; ) {"), print_newline,
   indent(3),
   get_code_block(MCLstmt,MCLcode)},
   sequence_of_statements(MCLcode),
   {indent(-3),
    print_literal(")"), print_newline}) |
  ({is_while_loop_stmt(MCLstmt),
   get_iteration_scheme(MCLstmt,Condition),
   print_literal("while (")}, expression(Condition),
   {print_literal(") {"), print_newline,
   indent(3),
   get_code_block(MCLstmt,MCLcode)},
   sequence_of_statements(MCLcode),
   {indent(-3),
    print_literal(")"), print_newline}) |
  ({is_for_loop_stmt(MCLstmt)},
   process_for_loop(MCLstmt)).

```

Figure 5.8: Transformation rule for generating C/JNX loop statements.


```

select_stmt(Arc) -->
  ({is_timed(Arc)},
  ({is_choice(Arc),
    get_all_destinations(Arc,Gates1),
    get_all_destinations(Arc,Gates2),
    get_pull_timer(Arc,Timer)},
  waiting_place_evaluations(Gates1),
  start_delay_alternative(Arc,Timer),
  {print_literal("event = await_any_event(&mail);"),
   print_newline,
   print_literal("switch(event) {", print_newline,
   indent(3)},
  event_alternatives(Gates2),
  delay_alternative(Arc,Timer),
  {indent(-3),
   print_literal("}"), print_newline,
   print_literal("reset_timed_flag(")},
  identifier(Arc), {print_literal("_timer;"),
   print_newline}}) |
  ({get_destination(Arc,Gate),
   get_pull_timer(Arc,Timer)},
  start_delay_alternative(Arc,Timer),
  {print_literal("event = await_any_event(&mail);"),
   print_newline,
   print_literal("switch(event) {", print_newline,
   indent(3)},
  event_alternative(Gate),
  delay_alternative(Arc,Timer),
  {indent(-3),
   print_literal("}"), print_newline,
   print_literal("reset_timed_flag(")},
  identifier(Arc), {print_literal("_event;"),
   print_newline}})) |
  ({get_all_destinations(Arc,Gates1),
   get_all_destinations(Arc,Gates2)},
  waiting_place_evaluations(Gates1),
  {print_literal("event = await_any_mail(&mail);"),
   print_newline,
   print_literal("switch(event) {", print_newline,
   indent(3)},
  event_alternatives(Gates2),
  {indent(-3),
   print_literal("}"), print_newline}).

```

Figure 5.9: Transformation rule for generating C/JNX code which receives mail.

```

simple_type_declaration(DataNode) -->
  {get_type_node(DataNode,TypeNode)},
  simple_subtype_indication(TypeNode),
  identifier(DataNode),
  (({get_initial_expression(DataNode,InitExpression),
    print_literal(" = ")}),
    constant_expression(InitExpression),
    {print_literal(";"), print_newline, print_newline}) |
    {print_literal(";"), print_newline, print_newline}).

```

Figure 5.10: Transformation rule for generating simple C/JNX variable declarations.

```

subprogram_body(ButtonNode) -->
  subprogram_specification(ButtonNode), {print_newline,
  print_literal("{"), print_newline,
  indent(3),
  get_internal_children(ButtonNode,Internals),
  get_all_data_nodes(Internals,DataNodes)},
  object_declaration_list(DataNodes),
  {get_formal_parameters(ButtonNode,FormalDecls)},
  button_formal_parm_list(FormalDecls),
  (({get_engine_node(Internals,EngineNode)},
    local_declarations(EngineNode),
    {get_all_outgoing_arcs(EngineNode,Arcs)},
    task_call_declarations(Arcs),
    {get_formal_parameters(ButtonNode,FormalParms1)},
    assign_incoming_values(FormalParms1),
    fetch_MCL_code(EngineNode),
    {get_formal_parameters(ButtonNode,FormalParms2)},
    assign_outgoing_values(FormalParms2)) |
    {print_literal("/* NULL */"), print_newline}),
  {indent(-3),
  print_literal("}"), print_newline, print_newline,
  free_list(Internals)}.

```

Figure 5.11: Transformation rule for generating C/JNX functions.

2

P4-M-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

needed by JNX to create the task. Next a variable of type **task_id** is generated. This variable will eventually contain a pointer to the task once it has been created. Creation and scheduling of C/JNX tasks is performed by a start-up routine not shown in Figure 5.4.

The remainder of the declarations generated by the rule support rendezvous with the task. The variables of type **mailbox_id** are the first to be generated. These variables will eventually contain pointers to the mailboxes owned by the C/JNX task. One mailbox is created for each gate button contained in actor **timer**. The variable names are generated by the rule **mailbox_declaration_list/1** which retrieves the names of the two gate buttons contained in the actor **timer**. The entire declaration for the variable **recv_timer** is generated by the rule **flag_declaration_list/1**. This variable will eventually contain a pointer to a special timing flag owned by the C/JNX task. The variable declaration is generated because of the time out attached to pull arc **recv** in Figure 4.2.

The enumerated type declaration provides a set of unique labels for all the possible events which can occur while the C/JNX task is waiting for rendezvous. In this example, three possible events can occur: mail can be received at mailbox **start**; mail can be received at mailbox **stop**; or the flag **recv_timer** can time out. Two rules are used to generate the constants for the enumerated type. The first rule **mailbox_event_declaration_list/1** derives enumerated constants from the names of the two gate buttons contained in actor **timer** while the second rule **flag_event_declaration_list/1** derives an enumerated constant from the name of the pull arc. If there was no time out attached to the pull arc then the latter rule would have no effect.

As there are no waiting places or formal parameters associated with the gate buttons in Figure 4.2, neither of the rules **mailbox_parameter_declarations/1** and **gate_declaration_list/1** have any effect when invoked.

The rule for generating the C/JNX task body from the actor **timer** is shown in Figure 5.6. This rule first generates local variable declarations for the C/JNX task and then generates the C/JNX statements to be executed by the task. Five separate rules are invoked to generate local variable declarations for the C/JNX

task. Furthermore, a collection of rules to generate two additional variables, **mail** and **event** are executed since gate buttons are associated with actor **timer**.

The first rule, **local_gate_object_list/1**, generates local C/JNX variable declarations from the MCL variable declarations local to each of the gate buttons within an actor. As there are no data stores in either of the gate buttons in this example, the invocation of this rule has no effect.

The second rule **gate_formal_parm_declaration_list/1** generates local variable declarations from the formal parameters associated with each gate button. When C/JNX tasks rendezvous, they do not exchange information through formal parameter lists like gate buttons in TimeBench designs. Instead, the C/JNX tasks retrieve a pointer to a well known memory location containing the information (the mailbox). Although the formal parameter list in the gate buttons can be transformed to a well known memory location, references to the formal parameters which are found in the gate button engine's MCL code cannot be transformed. To accomodate this, the MCL formal parameters are transformed to C/JNX local variable declarations and the information contained in the well known memory location is assigned to these local variables when mail is received. No formal parameters are associated with either of the gate buttons in this example; therefore, the invocation of **gate_formal_parm_declaration_list/1** has no effect.

Local C/JNX variable declarations generated from any data stores contained within an actor are processed by the rule **object_declaration_list/1**. This rule generates one local variable declaration from store **onOrOff** in this example. If present, MCL variable declarations local to engine **timer** would be processed by the rule **local_declarations/1**. Finally, if a C/JNX task makes a call to another task's mailbox the rule **task_call_declarations/1** generates the local variables which will eventually become the well known memory location during the rendezvous. The invocation of this rule has no effect in this example.

The body of the task is generated by invoking two rules. The first rule invoked, **create_flags/1**, creates the special timing flag owned by the timer task while the second rule invoked, **fetch_MCL_code/1**, generates the C/JNX code which performs the same function as the three MCL statements contained in engine **timer**.

The rules used to transform the MCL assignment statement to a C/JNX assignment statement (see Figure 5.7) and the MCL loop statement to an infinite for loop in C/JNX (see Figure 5.8) are almost identical to the two rules used in AdaGen. However, the rule to transform the MCL pull statement (see Figure 5.9) is considerably different than the one used in AdaGen. Like the corresponding rule in AdaGen, there are three cases handled by `select_stmt/1`. The timer example satisfies the first case as both the rules `is_timed/1` and `is_choice/1` will succeed when they are invoked.

As there are no waiting places attached to the gate buttons in this example, the rule `waiting_place_evaluations/1` has no effect when it is invoked. The rule `start_delay_alternative/1` generates C/JNX code which causes the timing flag to be signalled after a set period of time. The next line of code generated causes the C/JNX task to wait for an event to occur. The event is processed using a C/JNX switch statement generated from the rules `event_alternatives/1` which processes the two gate buttons and `delay_alternative/2` which processes the time out. Code to reset the flag is generated after the C/JNX switch statement.

The rule to transform the MCL variable declaration in store `onOrOff` to a C/JNX variable declaration is shown in Figure 5.10. The rule for transforming MCL integer, real, and boolean data declarations in C-Gen is almost the same as the rule used for all MCL declarations in AdaGen. However, MCL strings, lists, and sets do not receive the same treatment in C-Gen because the C/JNX declarations for these latter types cannot contain initial expressions.

The rule for generating the C/JNX function from push button `control` is shown in Figure 5.11. For the most part the rules invoked by this rule are similar to the ones invoked by the rule `task_body/1` in Figure 5.6; however, three rules require some comment.

Push button `control` has a formal parameter of mode `in out` associated with it. In order to achieve the same effect in C/JNX, a pointer to the variable must be passed as a parameter to the C/JNX function. Thus, every time the formal parameter is used in the C/JNX function, the parameter must be dereferenced in order to manipulate the original variable. Unfortunately, when processing the MCL

code inside the push button's engine, there is no way to determine if a variable has been declared in the formal parameter list and, therefore, there is no way to know when to generate dereferenced variable names. To get around this problem, the formal parameter associated with push button `control` is transformed to a local variable in the C function using the rule `button_formal_parm_list/1` and a special formal parameter is created for the C function. The value passed into the C function through the newly defined formal parameter is then assigned to the newly defined local variable at the beginning of the function using code generated by the rule `assign_incoming_values/1`. The value contained inside the local variable at the end of the function is assigned back to the specially defined formal parameter using code generated by the rule `assign_outgoing_values/1`.

Buffer Component

The C/JNX code generated from the buffer component in Figure 4.11 is shown in Figure 5.12. Because the buffer component is also an actor, many of the transformations that take place in this example are very similar to the transformations already discussed in the previous example. However, because the gate buttons have both formal parameters and waiting places associated with them, a few more of the rules invoked in `task_declaration/1` (Figure 5.5) and `select_stmt/1` (Figure 5.9) take effect.

The two structure declarations in Figure 5.12 are generated when the rule `mailbox_parameter_declarations/1` is invoked by `task_declaration/1`. Tasks in C/JNX do not exchange messages through a formal parameter list like gate buttons in TimeBench. Instead, a message is exchanged by placing it in a memory location known to both the sender and receiver. To ensure that the message maintains the same format as the formal parameter list in each of the gate buttons, C/JNX structures are used. One C/JNX structure is generated for each gate button in Figure 4.11 with the entry in each structure definition corresponding to the formal parameter in each of the gate buttons.

The two `MCLBoolean` type declarations in Figure 5.12 are generated when the rule `gate_declaration_list/1` is invoked. The declarations are generated because

```

/* ----- Buffer_type Task Declaration ----- */

void Buffer_type();
task_parameter_block
    Buffer_parameters = {Buffer_type, 1000, 3, 20, 1};
task_id Buffer;
mailbox_id Put, Get;
typedef enum {Put_event, Get_event} Buffer_events;
typedef struct {
    MCLInteger inVal;
} Put_parms;
typedef struct {
    MCLInteger outVal;
} Get_parms;
MCLBoolean notFull;
MCLBoolean notEmpty;

/* ----- Buffer_type Task Body ----- */

void Buffer_type()
{
    MCLInteger inVal;
    MCLInteger outVal;
    MCLList bufVal;
    mail_id mail;
    Buffer_events event;

    notEmpty = False;
    notFull = True;
    bufVal = empty_list();
    for ( ; ; ) {
        if (notEmpty) {
            unmask_mailbox(Get);
        } else {
            mask_mailbox(Get);
        }
    }
}

```

Figure 5.12: C/JNX code generated from the information contained in actor **buffer**.


```

if (notFull) {
    unmask_mailbox(Put);
} else {
    mask_mailbox(Put);
}
event = await_any_mail(&mail);
switch(event) {
    case Get_event:
        outVal = list_element(bufVal, 1);
        if ((list_len(bufVal) == 1)) {
            bufVal = empty_list();
            notEmpty = False;
        } else {
            bufVal = sub_list(bufVal, 2, list_len(bufVal)) ;
            notFull = True;
        }
        print_string(
            create_string("After GET buffer contains:"));
        print_list(bufVal);
        print_newline();
        ((Get_parms *) mail)->outVal = outVal;
        reply(Get);
        break;
    case Put_event:
        inVal = ((Put_parms *) mail)->inVal;
        modify_sub_list(bufVal, (list_len(bufVal) + 1),
                        list_len(bufVal),
                        create_list(inVal, empty_list()));
        notEmpty = True;
        if ((list_len(bufVal) >= 5)) {
            notFull = False;
        }
        print_string(
            create_string("After PUT buffer contains:"));
        print_list(bufVal);
        print_newline();
        reply(Put);
        break;
    }
}
}

```

Figure 5.12: Continued ...

```

slice_assignment(LHS,RHS) -->
{get_lhs(LHS,VarName),
 get_rhs(LHS,Range),
 get_lhs(Range,LowerLimit),
 get_rhs(Range,UpperLimit)},
(({is_string_expression(RHS),
  print_literal("modify_sub_str("), identifier(VarName),
  {print_literal(", ")}, expression(LowerLimit),
  {print_literal(", ")}, expression(UpperLimit),
  {print_literal(", ")}, expression(RHS),
  {print_literal(");"), print_newline}) |
 ({is_list_expression(RHS),
  print_literal("modify_sub_list("), identifier(VarName),
  {print_literal(", ")}, expression(LowerLimit),
  {print_literal(", ")}, expression(UpperLimit),
  {print_literal(", ")}, expression(RHS),
  {print_literal(");"), print_newline}})).

```

Figure 5.13: Transformation rule for generating a C/JNX function from string and list slice assignments in MCL.

```

length_operator(Operator,Expression) -->
{is_length_operator(Operator)},
(({is_string_expression(Expression),
  print_literal("str_len("),
  expression(Expression),
  {print_literal(")")} |
 ({is_list_expression(Expression),
  print_literal("list_len("),
  expression(Expression),
  {print_literal(")")} |
 ({is_set_expression(Expression),
  print_literal("set_size("),
  expression(Expression),
  {print_literal(")"})))).

```

Figure 5.14: Transformation rule for generating a C/JNX expression which determines the lengths of MCL strings, lists, or sets.

of the waiting places found in the buffer component.

C/JNX code is also generated just before the switch statement in the task body because of the presence of the waiting places in the buffer component. Recall from Chapter 4, that the waiting places in the buffer component restrict access to the gate buttons depending on the state of each waiting place (open or closed). To achieve the same effect the C/JNX program, code is generated to mask or unmask mailboxes depending on the state of the C/JNX variables representing the waiting places. This code is generated when the rule `waiting-place-evaluations/1` is invoked by `select_stmt/1`.

The presence of formal parameters in each of the gate buttons also causes a special assignment statement to be generated for each of the event alternatives in the switch statement. This assignment statement copies data into and out of the well known memory location used by the mailbox which generated the event to exchange messages between the sender and receiver. The assignment statement is located near the end of the first event alternative as data is being passed back to the sender; whereas, the assignment statement is located at the beginning of the second event alternative as data is being passed to the receiver. Whether data is being passed to or from the receiver is determined from the argument modes associated with the formal parameters in the gate buttons.

The remainder of this discussion will briefly cover the transformation of MCL list expressions to C/JNX as the transformation rules used to transform MCL expressions in C-Gen are very similar to the transformation rules used to transform MCL expressions in AdaGen. The major difference between the two sets of rules is due to the fact that C/JNX functions cannot be overloaded like Ada functions and, therefore, distinct names must be used for each C/JNX function.

The mappings of the MCL list expressions used in the buffer component to C/JNX functions are as follows:

```

bufVal(low .. high) := list;  $\Rightarrow$ 
    modify_sub_list(bufVal,low,high,list);
#bufVal  $\Rightarrow$  list_len(bufVal)
[]  $\Rightarrow$  empty_list()
[head|tail]  $\Rightarrow$  create_list(head,tail)
bufVal(position)  $\Rightarrow$  list_element(bufVal,position)
bufVal(low .. high)  $\Rightarrow$  sub_list(bufVal,low,high)

```

The transformation rules used to transform the first two expressions are shown in Figures 5.13 and 5.14 respectively. The transformation rules used to transform the remaining expressions are very similar in format to the second rule as they first determine if the MCL expression is applicable to the rule and then generate the appropriate C/JNX function call using information contained in the MCL expression.

Chapter 6

User Annotations

6.1 Overview

Some degree of user guidance is needed for computer assisted code generation [4, 5, 6, 31, 36] mainly because so many of the steps from design to implementation are based on intuitive knowledge and experience rather than formal procedures. Complete automation cannot be achieved until all the steps from design to implementation are formalized.

Even if it were possible to completely automate the transformation from design to implementation, it may not be desirable to do so. The cost of achieving 100% automation may far outweigh the benefits of relieving the user of a few simple implementation steps. Moynihan et al. [31] discuss the trade-offs involved when deciding on the level of automation for a source-to-source translator. The issues raised in the paper also apply to design-to-source translators such as the ones in the TimeBench code generation facility.

At present, no form of user guidance is available to the code generators in the TimeBench code generation facility. However, it is envisaged that the user will be able to provide the code generators with extra transformation instructions by annotating the design with implementation information. The user will need to use annotations when there is insufficient information in the design to automatically derive the appropriate implementation.

The next section briefly discusses some of the changes to TimeBench and MetaGen that will be needed to incorporate user annotations into the TimeBench code generation facility. The final section presents some areas in which user annotations will provide implementation details which cannot be derived automatically from the design information.

6.2 Implementing User Annotations

The primary purpose of user annotations is to provide implementation specific information to the code generators. As implementation specific information can vary from one target environment to another, the annotations must be flexible; yet, at the same time, the annotations should disturb the design as little as possible. To accommodate flexibility, it is proposed that each code generator be responsible for parsing its own annotations. This way different annotations can be used for different target environments.

To disturb the design as little as possible, it is proposed that the TimeBench parser treat all user annotations like comments. This is easy to do for user annotations placed within MCL code fragments since comments are already part of MCL syntax. Adding annotations to MachineChart icons is somewhat more difficult as there is presently no facility for associating comments with the icons. The best location to add comments to icons is in the window used to input an icon's attributes. Once comments can be entered for all the design components, the ability to parse comments and store them as strings in the TimeBench Design Tree needs to be added to TimeBench.

A number of changes to both TimeBench and MetaGen are necessary in order to implement user annotations. As discussed above, some facility for entering the annotations into the TimeBench Design Tree has to be added to TimeBench. This involves changes to both the graphical interface (so comments can be added to the MachineChart icons) and the logical interface (so comments can be parsed and stored in the TimeBench Design Tree). Furthermore, methods which manipulate the TimeBench Design Tree such as deletion, rearrangement and replacement of objects

should be added to the logical interface to make it easier to process the annotations. Presently, the logical interface only contains methods to read information from the design tree and, at a primitive level, to add information to the tree.

Although the changes to MetaGen will be extensive, they will only affect the MetaGen libraries. Library functions will be added to parse the user annotations and to integrate the implementation information found in the annotations with the design information found in the TimeBench Design Tree.

With the addition of user annotations to the TimeBench Design Tree, the code generators produced by MetaGen will have to make at least two complete passes through the TimeBench Design Tree (which is why the methods to manipulate the tree should be added to the logical interface). The first pass will extract the implementation specific information from the annotations and modify the design tree accordingly. The second pass will then generate the implementation code based on the modified version of the TimeBench Design Tree.

6.3 Where User Annotations are Needed

Although specific annotations may vary from code generator to code generator, a number of areas in which user guidance is required have been identified. These areas are listed below.

- There are many ways to implement strings, lists and sets, some more efficient than others. The user should be given the opportunity to specify which implementation best suits the design.
- The user should be able to refine MCL data types. For example, the full range of integers is not needed for a data declaration dealing with the number of days in a month.
- The user should be able to specify optimizations to improve the performance of the code.
- The user should be given the ability to insert code fragments (e.g., in Ada, in C/JNX) or specify hooks to existing code libraries whenever it is appropriate.

- The user should be able to organize the implementation code (e.g., into different files, etc.) to best suit the environment.
- The user should be able to specify the values of constants. For example, a timer delay value which is meaningful in the TimeBench environment will not be meaningful in the Ada environment as different measures are used by the two environments.
- The user should be able to specify how error handling is dealt with in the implementation. For example, there is no way to represent Ada exception handling using the TimeBench design notation (even though this capability is included in the MachineChart notation). Therefore, instructions for generating Ada exceptions should be given to AdaGen through user annotations.
- The user should be able to supply task information such as priority, stack allocation, etc. Furthermore, the user should be able to specify which processor a task is scheduled to execute on.

To illustrate how user annotations will be used, a number of potential annotations for AdaGen are presented in the remainder of this section. First presented is a list of annotations along with a description of where the annotations will be used and what effect they will have on code generation. Second, a small example is presented to show how a sampling of the user annotations will affect the Ada code.

6.3.1 Annotations for AdaGen

\$new_file ::= *file_id*

This annotation is placed in the comment field of a box, an actor, a reactor or a push button and causes the Ada code generated from the component to be placed in the file labelled *file_id*. The "\$" symbol placed at the beginning of the command signals to the code generator that the comment being processed contains an user annotation.

\$task_priority ::= priority

This annotation is placed in the comment field of an actor or a reactor and causes an Ada priority pragma to be inserted in the Ada task generated from the component.

\$replace_hook ::= library_name, subprogram_name

This annotation is placed in the comment field of a push button. When this annotation is detected, the predefined subprogram *subprogram_name* which is contained in library *library_name* is used instead of the Ada code which would normally be generated from the push button.

\$insert_hook ::= library_name, subprogram_name, argument_list

This annotation is found in an MCL comment and causes a call to the subprogram *subprogram_name* in library *library_name* to be inserted into the Ada code according to the location of the MCL comment in the MCL code.

\$replace_fragment ::= "Ada code"

\$end_replace_fragment

These two annotations are located in MCL comments. When these annotations are detected, the MCL code contained between the annotations is replaced with the Ada code fragment contained in the first annotation.

\$insert_fragment ::= "Ada code"

This annotation is placed in an MCL comment and causes additional Ada code to be inserted into the generated Ada code according to the location of the MCL comment in the MCL code.

\$constrain_type ::= MCL_type, Ada_type, Ada_type_definition

This annotation is placed in an MCL comment field inside a data store. If *MCL_type* is integer, real or boolean, this annotation causes an extra type declaration to be generated. The resulting Ada type is named in *Ada_type* and is defined in *Ada_type_definition*. Furthermore, the appropriate generic package contained in the

```

TYPE MCLInteger IS RANGE Integer'First .. Integer'Last;
TYPE MCLReal IS DIGITS 15;
TYPE MCLBoolean IS NEW Boolean;

PACKAGE MCLIntProcs IS NEW Integer_Library(MCLInteger,MCLReal,
    MCLBoolean);
USE MCLIntProcs;

PACKAGE MCLRealProcs IS NEW Real_Library(MCLReal,MCLInteger,
    MCLBoolean);
SE MCLRealProcs;

PACKAGE MCLBoolProcs IS NEW Boolean_Library(MCLBoolean);
USE MCLBoolProcs;

PACKAGE MCLString IS NEW String_Unbounded_Unmanaged(MCLInteger,
    MCLBoolean);
USE MCLString;

PACKAGE MCLListOfInteger IS NEW List_Single_Unbounded_Unmanaged
    (MCLInteger,MCLBoolean,MCLInteger,Is_Equal,Output,Copy);
USE MCLListOfInteger;

```

Figure 6.1: Type declarations generated for unannotated producer/consumer design.

Ada library is instantiated to support operations on the constrained data type. In the case of MCL strings, lists and sets, this annotation is used to define the most efficient Ada implementation to use.

\$redefine_var ::= *variable_name, type_declaration*

This annotation is also placed inside data stores and allows users to redefine MCL variables in terms of the constrained data types defined in the previous annotation.

\$redefine_delay_value ::= *new_value*

One of the constants used in TimeBench designs which needs to be changed in the corresponding Ada programs is the delay time associated with the time out icon.

This annotation which is placed in the comment field of a time out allows users to modify the delay time so that a value which is meaningful to the Ada environment is generated.

6.3.2 The Buffer Component Revisited

The buffer component extracted from the producer/consumer design is found in Figure 4.11 and the Ada code currently generated from the buffer component by AdaGen is shown in Figure 4.12. The Ada type declarations generated from the producer/consumer design are shown in Figure 6.1. The Ada implementation in Figure 4.12 is correct, but is not necessarily the best implementation for a particular environment. To improve the implementation, say the following constraints should be applied to the design.

- The buffer will service the producer and consumer in a timely fashion if the buffer task is of a higher priority than either the producer task or the consumer task.
- The buffer will only receive integer values in the range 1 to 100.
- Because the buffer gets recreated every time it is emptied, the buffer will make more efficient use of memory if it is managed.
- The consumer will operate more efficiently if the items in the buffer are sorted.

The annotated version of the buffer component which will produce an Ada program satisfying the constraints listed above is shown in Figure 6.2. The annotation `$task_priority` sets the priority of the buffer task. The annotations `$constrain_type` and `$redefine_var` limit the values which can be stored in the buffer and redefine the buffer to be bounded and managed. Not shown in Figure 6.3 are two additional `$redefine_var` annotations which constrain the types of the formal parameters found in gate buttons `Put` and `Get` so that the parameters remain compatible with the new buffer type. Finally, the annotation `$insert_hook` generates a call to a predefined sort procedure which sorts the items in the buffer.

Figure 6.2: Annotated TimeBench design.

```

TASK TYPE BufferType IS
  ENTRY Put(inVal : IN Limited);
  ENTRY Get(outVal : OUT Limited);
  PRAGMA Priority(10);
END BufferType;
Buffer : BufferType;

WITH SortLib;
TASK BODY BufferType IS

  bufVal : BoundedListOfLimited.List;
  notFull : MCLBoolean;
  notEmpty : MCLBoolean;

BEGIN
  notEmpty := False;
  notFull := True;
  bufVal := Construct;
  LOOP
    SELECT
      WHEN notEmpty = True =>
        ACCEPT Get(outVal : OUT Limited) DO
          outVal := Get_Item(bufVal,1);
          IF Is_Equal(Length_Of(bufVal),1) THEN
            bufVal := Construct;
            notEmpty := False;
          ELSE
            bufVal := SubPart_Of(bufVal,2,
                                Length_Of(bufVal));
            notFull := True;
          END IF;
          Output(Initialize("After GET buffer contains:"));
          Output(bufVal);
          New_Line;
        END Get;
      OR

```

Figure 6.3: Ada code generated from annotated buffer component.

```

    WHEN notFull = True =>
        ACCEPT Put(inVal : IN Limited) DO
            Modify(bufVal, (Length_Of(bufVal) + 1),
                Length_Of(bufVal),
                Construct(inVal, Construct));
            Sort(bufVal);
            notEmpty := True;
            IF Greater_Than_Equal(Length_Of(bufVal), 5) THEN
                notFull := False;
            END IF;
            Output(Initialize("After PUT buffer contains:"));
            Output(bufVal);
            New_Line;
        END Put;
    END SELECT;
END LOOP;
END BufferType;

```

Figure 6.3: Continued ...

The resulting Ada code and the new Ada type declarations are shown in Figures 6.3 and 6.4, respectively.

The integration of user annotations with code generation leads to a new definition of the activity “implementation.” Rather than produce the implementation manually, the user guides the automatic generation of the implementation by inserting implementation code where needed, adding hooks to existing code, refining MCL data types, and providing implementation specific details about the environment. The code generator automatically integrates the information found in the annotations with the design information in order to produce a target implementation. The use of annotations also makes the maintenance phase of software development easier as a complete record of all implementation decisions is recorded along with the design.

```
TYPE MCLInteger IS RANGE Integer'First .. Integer'Last;
TYPE Limited IS RANGE 1..100;
TYPE MCLReal IS DIGITS 15;
TYPE MCLBoolean IS NEW Boolean;

PACKAGE MCLIntProcs IS NEW Integer_Library(MCLInteger,MCLReal,
    MCLBoolean);
USE MCLIntProcs;

PACKAGE LimitedProcs IS NEW Integer_Library(Limited,MCLReal,
    MCLBoolean);
USE LimitedProcs;

PACKAGE MCLRealProcs IS NEW Real_Library(MCLReal,MCLInteger,
    MCLBoolean);
USE MCLRealProcs;

PACKAGE MCLBoolProcs IS NEW Boolean_Library(MCLBoolean);
USE MCLBoolProcs;

PACKAGE MCLString IS NEW String_Unbounded_Unmanaged(MCLInteger,
    MCLBoolean);
USE MCLString;

PACKAGE BoundedListOfLimited IS NEW bounded_managed_list
    (MCLInteger,MCLBoolean,Limited,Is_Equal,Output,Copy);
USE BoundedList;
```

Figure 6.4: Type declarations generated for annotated producer/consumer design.

Chapter 7

Conclusions

7.1 Research Summary

With the introduction of the code generation facility to TimeBench, it is feasible for a developer to spend more time designing the software and less time implementing it. Using TimeBench, the developer can develop, test, and maintain the software at the design phase of the software life cycle since the code generation facility produces most of the implementation automatically.

The TimeBench code generation facility can contain one or more code generators, each targeting a different implementation environment. The facility is novel in that a meta-code generator called MetaGen is used to develop each code generator for the facility.

For the purposes of this thesis, the development of the code generation facility for TimeBench can be broken down into three related projects: the development of a meta-code generator, MetaGen; the development of an Ada code generator, AdaGen; and the development of a C/JNX code generator, C-Gen. The primary purpose of the latter two projects is to test the feasibility of MetaGen.

MetaGen

MetaGen is a tool implemented in Prolog which automatically produces code generators for the TimeBench code generation facility. MetaGen takes as input an MSL

specification of a code generator and outputs an Objective C implementation of the code generator. Objective C is the language in which TimeBench is implemented.

The advantages to using a meta-code generator approach are as follows.

- It is much easier to specify and maintain the code generators using MSL rather than Objective C. Simple logical constructs are used to specify how MSL rules are evaluated; MSL variables do not have to be declared; and, furthermore, MSL specifications hide details about accessing the underlying representation of TimeBench designs.
- Although the TimeBench code generation facility can automatically generate code for many different target environments, each code generator is developed individually. It is easier to achieve higher levels of automation when the transformation rules only have to deal with the TimeBench design notation and one implementation language rather than provide a generalized transformation for all target environments.
- The TimeBench code generation facility can be configured to produce implementations for any real time programming environment simply by linking in the appropriate code generators.

A MSL specification consists of rules which specify how the TimeBench design notation is to be transformed into the implementation language. MSL rules are Prolog-like in form in that they are composed of a head and a body, the body consisting of a series of expressions related by the logical constructs “,” (*AND*) and “|” (*OR*). Fundamental expressions in the rules either derive their values from built-in functions provided in the MetaGen libraries or other MSL rules.

There are four libraries of built-in Objective C functions supplied with MetaGen. The first three libraries provide functions which manipulate the TimeBench Design Tree. The fourth library supplies functions which output the code being generated by the code generators. Essentially, the purpose of a MSL specification is to order the references to the built-in functions in such a way that the appropriate code generator is formed.

The mapping between MSL and Objective C is straightforward and only targets a subset of Objective C constructs. MSL rules become Objective C functions while the fundamental expressions contained within the rules become Objective C function calls. The logical constructs within the rules become "if" statements in Objective C.

All the variables referenced in the MSL specification become variables of type `id` in Objective C. Type `id` is a special type declaration which describes Objective C objects. As most items in the TimeBench Design Tree are objects, this transformation is reasonable. Treating the variables as one type eliminates all the typing problems associated with translating a polymorphic language to a monomorphic language.

The execution of the Objective C code is somewhat similar to that of Prolog in that the functions always succeed or fail. However, unlike Prolog, the resulting Objective C code is deterministic as it does not support backtracking. Once a value is bound to a variable, the value remains in the variable even if a branch of the rule fails.

The Objective C code has been optimized so that built-in functions which always succeed are not followed by a test for success. Furthermore, MSL rules containing tail recursion are also optimized in the corresponding Objective C functions.

AdaGen

The first code generator built using MetaGen automatically transforms TimeBench designs into Ada programs. Ada constructs closely parallel the TimeBench design notation: the hierarchical nature of TimeBench designs can be easily expressed using hierarchical Ada structures; most of the MachineChart components in the design notation map to equivalent Ada constructs; and MCL is an Ada-like design language. Therefore, the transformation of TimeBench designs to Ada programs is fairly straightforward and serves as a good initial test of MetaGen.

The greatest difficulty in transforming MachineChart structural components to Ada occurs when trying to satisfy Ada's order of compilation rules. The TimeBench structural components must be processed twice, once to generate Ada variable dec-

larations and specifications and once to generate Ada bodies.

MachineChart behavioural components are somewhat more difficult to transform as these tend to represent concepts which can be represented in Ada rather than mapping to Ada constructs directly. However, by traversing the links in the TimeBench Design Tree created by the arcs and by looking ahead along the branches of the TimeBench Design Tree, all the information needed to generate equivalent Ada constructs can be found.

ACM's have no equivalent representation in Ada; therefore, a special Ada package template is used by AdaGen to produce Ada packages which have the same functionality as ACM's in TimeBench designs.

Most of the transformations from MCL statements to Ada statements are quite easy as the syntax of many of the MCL statements is very similar to that of Ada statements. However, there are a few MCL statements such as **push** and **pull** which do not have an Ada equivalent. Transformation of these statements is quite involved as they tend to turn into transformations of MachineChart behavioural components.

The most difficult transformations performed by AdaGen are those dealing with MCL expressions and data types. There are too many differences between MCL data types and expressions and Ada data types and expressions to resolve using transformation rules alone, especially as Ada is such a strongly typed language. Therefore, an acceptable solution is to provide an Ada library which supplements the transformation rules dealing with MCL data types and expressions by defining equivalent Ada data types where needed and supplying Ada functions which mimic MCL expressions.

C-Gen

The second code generator developed using MetaGen produces C/JNX programs from TimeBench designs. Unlike Ada, C/JNX constructs do not closely parallel the TimeBench design notation although the same concepts can be represented by both C/JNX constructs and the notation. Thus, the successful development of C-Gen illustrates two points: (1) MSL and the MetaGen libraries are sufficiently general to develop code generators for significantly different target environments; and (2)

code generators for target languages whose constructs do not closely parallel the TimeBench design notation can be defined using MetaGen.

Most of the transformation rules used in C-Gen are more complex than those used in AdaGen since information in the TimeBench Design Tree often has to be reordered before C/JNX constructs can be generated. Nevertheless, many of the techniques used in developing C-Gen's transformation rules are very similar to those used in developing AdaGen's transformation rules.

The rules for transforming MachineChart structural components to C/JNX constructs are further complicated by the lack of heirarchy and modularity in the C programming language. Because C/JNX functions cannot be nested inside one another, TimeBench designs must be flattened during the transformation. Furthermore, since C does not actively enforce modular programming, TimeBench components originally hidden inside boxes in a TimeBench design become C/JNX constructs which are visible to the entire C/JNX program. Both these deficiencies lead to scoping and naming problems which, currently, must be resolved manually.

7.2 Experimental Results

The current version of AdaGen installed in TimeBench is a working prototype. C-Gen is not to the same stage as AdaGen, but can be used transform most TimeBench designs which do not use lists and sets. Both AdaGen and C-Gen have been tried out on a number of TimeBench designs. The designs include the control flag design, the producer/consumer design, the five dining philosophers design, a comprehensive test of MCL, and a communication system design.

AdaGen successfully generates fully formatted and fully executable Ada programs for all the designs listed above. The communication system design had to be modified somewhat to accomodate for the differences in the operational semantics of TimeBench and Ada, and the delay time in the resulting Ada program had to be modified to suit the Ada environment. However, no other user intervention was required.

C-Gen successfully generates fully formatted and fully executable C/JNX pro-

grams for a simplified version of the producer/consumer design and the communication system design. Fully compilable C/JNX code is generated for the more complex producer/consumer design, but it cannot be executed as the design makes use of lists. The five dining philosophers design and the control flag design can be transformed but no attempt has been made to compile or execute them.

C-Gen requires more user intervention than does AdaGen. The TimeBench designs often had to be reworked to get rid of naming conflicts and scoping errors in the resulting C/JNX code. Furthermore, the default task parameter values generated by C-Gen sometimes had to be modified in the resulting C/JNX program in order to prevent the program from crashing. The delay time in the C/JNX implementation of the communication system design also had to be modified.

7.3 Future Work

The main thrust of future research into MetaGen and the TimeBench code generation facility should target the development of user annotations. Through annotations, the developer will be able to guide the code generators by providing details about the implementation environment which cannot be derived automatically from the design either because the information is missing entirely or is insufficient to generate the best implementation. Furthermore, the developer will have a complete record of implementation decisions contained within the user annotations which will make software maintenance easier to perform.

Although user annotations are an important addition to the TimeBench code generation facility, there are other areas of the facility which are open to development as well.

- AdaGen and C-Gen do not deal with some components of the MachineChart notation supported in the current version of TimeBench (install arcs, remove arcs, jabs, scrolls, robot pools). Algorithms to support these transformations have yet to be developed.
- TimeBench is a prototype and therefore is likely to change. AdaGen and C-Gen should keep up with any revisions made to the TimeBench design notation

(e.g., new icons, revised MCL data types, etc.).

- Currently, AdaGen and C-Gen generated undocumented programs. Any comments contained in the TimeBench design are lost in the transformation. This should be rectified in future versions of AdaGen and C-Gen.
- AdaGen and C-Gen should be able to optimize implementation code. For example, C-Gen often generates the following sequence of C/JNX statements because of the way it transforms formal parameters: $a = b; b = c;$. With the addition of an optimization pass, C-Gen could produce the following statement instead: $a = c;$. Optimizations like the one in the example would be performed automatically while other optimizations would require user guidance.
- Code generators for other target environments can be developed.

Before any further development of the TimeBench code generation facility takes place, the logical interface to the TimeBench Design Tree should be reworked. Currently, the tree is essentially read only; therefore, code generation tends to be inefficient when objects must be accessed out of order. There is some provision to add objects to the tree, but the methods to do this are primitive and difficult to use.

What is needed in the logical interface are methods which can add, delete and move any object. Then as long as a copy of the TimeBench Design Tree is passed to the code generator, the design tree can be used as an intermediate representation for both the design, and, as the transformation progresses, the implementation. For example, the first pass through the design tree can insert information from the user annotations into the tree, the second pass through the tree can rearrange the tree to resemble a parse tree for the implementation language, and the third pass through the tree can optimize the code contained in the parse tree. The ability to make multiple passes through the tree and record the changes to the tree will make it much easier to implement user annotations and some of the more difficult transformations (such as jabs).

With the changes to the TimeBench logical interface and the addition of user annotations, the code generation facility within TimeBench will become a powerful

tool. Software developers using TimeBench will no longer be concerned with implementing software in the traditional sense. The developer will be able to develop, test and maintain the software at the design phase of the software life cycle. Then rather than manually produce implementations from the design, the developer will annotate the design in order to guide the implementation process. The information contained in the annotations will be integrated with the design information by the TimeBench code generation facility to automatically produce an appropriate implementation of the design.

References

- [1] P. F. Albrecht, P. E. Garrison, S. L. Graham, R. H. Hyerle, P. Ip, and B. Krieg-Bruckner, "Source-to-Source Translation: Ada to Pascal and Pascal to Ada," *SigPlan Notices*, vol. 15, no. 11, pp. 183-193, November 1980.
- [2] P. Allen and A. Burns, "Program Generation for Ada - A Case Study," *Software - Practice and Experience*, vol. 18, no. 12, pp. 1125-1138, December 1988.
- [3] R. M. Balzer, "A Global View of Automatic Programming," in *Third International Joint Conference on Artificial Intelligence*, (Menlo Park, CA), pp. 494-499, Stanford Research Institute Publications Department, 1973.
- [4] R. Balzer, T. E. Cheatham, and C. Green, "Software Technology in the 1990's: Using a New Paradigm," *Computer*, vol. 16, no. 11, pp. 39-45, November 1983.
- [5] R. Balzer, "A 15 Year Perspective on Automatic Programming," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, pp. 1257-1267, November 1985.
- [6] F. L. Bauer, B. Moller, H. Partsch, and P. Pepper, "Formal Program Construction by Transformations - Computer-Aided, Intuition-Guided Programming," *IEEE Transactions on Software Engineering*, vol. 15, no. 2, pp. 165-179, February 1989.
- [7] D. Bergmark, "GIBBSGEN: Code Generation for GIBBS," in *A Second Conference on Software Development Tools, Techniques and Alternatives*, (Washington, D.C.), pp. 101-111, IEEE Computer Society Press, December 1985.
- [8] G. Booch, *Software Components With Ada, Structures, Tools, and Subsystems*. Don Mills, Ontario: The Benjamin/Cummings Publishing Company, Inc., 1987.
- [9] G. Booch, *Object Oriented Design with Applications*. Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1991.

- [10] J. M. Boyle and M. N. Muralidharan, "Program Reusability through Program Transformation," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, pp. 574-588, September 1984.
- [11] R. J. A. Buhr, *System Design With Ada*. New Jersey: Prentice Hall Inc., 1984.
- [12] R. J. A. Buhr, G. M. Karam, and R. Casselman, "TimeBench: A State Machine Syntax and Method for Machine Charts," SCE-89-6, Department of Systems and Computer Engineering, Carleton University, Ottawa, CA, March 1989.
- [13] R. J. A. Buhr, G. M. Karam, C. M. Woodside, R. Casselman, G. Franks, H. J. Scott, and D. Bailey, "TimeBench : a CAD Tool for Real-Time System Design," SCE-90-17, Department of Systems and Computer Engineering, Carleton University, Ottawa, CA, September 1990.
- [14] R. J. A. Buhr, *Practical Visual Techniques in System Design (with Application to Ada)*. Englewood Cliffs, NJ: Prentice-Hall, 1990.
- [15] R. J. A. Buhr, D. Bailey, J. Bryant, and W. Sullivan, "94.333 Course Notes: Design and Programming Concurrent, Real Time Systems with Turbo C," Winter, 1991.
- [16] R. J. A. Buhr, G. M. Karam, and R. Casselman, "Support for Specifying Temporal Behavior in Ada Designs," *Proceedings of the First Symposium on Environments and Tools for Ada*; in: *Ada Letters*, vol. 11, no. 3, pp. 91-101, Spring 1991.
- [17] R. D. Cameron and M. R. Ito, "Grammar-Based Definition of Metaprogramming Systems," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 1, pp. 20-54, January 1984.
- [18] R. Casselman, G. M. Karam, and R. J. A. Buhr, "TimeBench: A Short Description of the Machine Charts Textual Support Language," SCE-89-25, Department of Systems and Computer Engineering, Carleton University, Ottawa, CA, September 1989.
- [19] T. E. Cheatham, G. H. Holloway, and J. A. Townley, "Program Refinement by Transformation," in *5th International Conference on Software Engineering*, pp. 430-437, Computer Society Press, 1981.
- [20] T. E. Cheatham, "Reusability Through Program Transformations," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, pp. 589-594, September 1984.

- [21] N. Cohen, *Ada as a Second Language*. Toronto, CA: McGraw-Hill Book Company, 1986.
- [22] M. S. Feather, "A System for Assisting Program Transformation," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 1, pp. 1-20, January 1982.
- [23] H. Fersko-Weiss, "CASE Tools for Designing Your Applications," *PC Magazine*, vol. 9, no. 2, pp. 213-251, January 1990.
- [24] S. Fickas, "Automating the Transformational Development of Software," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, pp. 1268-1277, November 1985.
- [25] A. V. Goldberg and K. J. Lieberherr, "GEM: A Generator of Environments for Metaprogramming," in *A Second Conference on Software Development Tools, Techniques and Alternatives*, (Washington, D.C.), pp. 86-95, IEEE Computer Society Press, December 1985.
- [26] G. M. Karam, R. J. A. Buhr, C. M. Woodside, and R. Casselman, "Time-Bench: A Next Generation CASE Tool," SCE-89-18, Department of Systems and Computer Engineering, Carleton University, Ottawa, CA, July 1989.
- [27] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.
- [28] G. Kotik, "Knowledge-Based Compilation of High-Level Data Types," KES.U.83.5, Kestrel Institute, March 1984.
- [29] T. Lewis, "Code Generators," *IEEE Software*, pp. 67-70, May 1990.
- [30] P. A. Luker and A. Burns, "Program Generators and Generation Software," *The Computer Journal*, vol. 29, no. 4, pp. 315-321, 1986.
- [31] V. D. Moynihan and P. J. L. Wallis, "The Design and Implementation of a High-Level Language Converter," *Software - Practice and Experience*, vol. 21, no. 4, pp. 391-400, April 1991.
- [32] P. W. Oman, "CASE Analysis and Design Tools," *IEEE Software*, pp. 37-43, May 1990.
- [33] H. Partsch and R. Steinbruggen, "Program Transformation Systems," *Computing Surveys*, vol. 15, no. 3, pp. 199-236, September 1983.

- [34] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 2nd edition. Toronto, Canada: McGraw-Hill, Inc., 1987.
- [35] Real-Time and Distributed Systems Group, Carleton University, Ottawa, CA, *User's Guide for TimeBench*, October 1990.
- [36] C. Rich and R. C. Waters, "Automatic Programming: Myths and Prospects," *Computer*, vol. 21, no. 8, pp. 40-51, August 1988.
- [37] D. R. Smith, G. B. Kotik, and S. J. Westfold, "Research on Knowledge-Based Software Environments at Kestrel Institute," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, pp. 1278-1295, November 1985.
- [38] C. M. Stanczyk, "MLog to Ada Transformations," Master's thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa CA, January 1989.
- [39] L. Sterling and E. Shapiro, *The Art of Prolog, Advanced Programming Techniques*. Cambridge, Massachusetts: The MIT Press, 1987.
- [40] K. W. Van Der Loo, "Design and Implementation of ASCGEN, A Prolog Tool For Translating Graphical Designs into Ada Code," Master's thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, CA, April 1987.
- [41] S. Westfold, *Logic Specifications for Compiling*. PhD thesis, Kestrel Institute, Palo Alto, CA., June 1984.
- [42] H. Zimmermann, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications*, vol. COM-28, no. 4, pp. 425-432, April 1980.

Appendix A

MSL Grammar

The MSL grammar in BNF (Backus-Naur Form) is as follows:

rule	$\Rightarrow$	head $\rightarrow$ body.
fact	$\Rightarrow$	head.
head	$\Rightarrow$	symbol symbol(formal_arg_list)
body	$\Rightarrow$	expression expression_list
expression	$\Rightarrow$	symbol symbol(actual_arg_list) (expression_list) {expression_list} expression op expression
expression_list	$\Rightarrow$	expression expression,expression_list
formal_arg_list	$\Rightarrow$	formal_arg formal_arg,formal_arg_list
actual_arg_list	$\Rightarrow$	actual_arg actual_arg,actual_arg_list
formal_arg	$\Rightarrow$	mode variable

		variable
actual_arg	⇒	number string variable
op	⇒	' '
mode	⇒	+ -
symbol	⇒	lower_case_alpha lower_case_alpha alpha_numerics
variable	⇒	upper_case_alpha - upper_case_alpha alpha_numerics _ alpha_numerics
number	⇒	numeral_list + numeral_list - numeral_list
string	⇒	"symbol_list"
alpha_numerics	⇒	lower_case_alpha upper_case_alpha numeral - lower_case_alpha alpha_numerics upper_case_alpha alpha_numerics numeral alpha_numerics _ alpha_numerics
numeral_list	⇒	numeral numeral numeral_list
symbol_list	⇒	lower_case_alpha upper_case_alpha numeral symbol

```

lower_case_alpha symbol_list |
upper_case_alpha symbol_list |
numeral symbol_list |
symbol symbol_list

```

```

lower_case_alpha  =>  a|b|c|d|e|f|g|h|i|j|k|l|m|
                    n|o|p|q|r|s|t|u|v|w|x|y|z

```

```

upper_case_alpha  =>  A|B|C|D|E|F|G|H|I|J|K|L|M|
                    N|O|P|Q|R|S|T|U|V|W|X|Y|Z

```

```

numeral           =>  0|1|2|3|4|5|6|7|8|9

```

```

symbol            =>  ! | @ | # | $ | % | & | * | ( | ) | _ | - | + | = | ' | |
                    ' | { | } | [ | ] | ; | : | ' | " | , | . | < | > | ? | /

```

Appendix B

Excerpts from the Communication System Design

This appendix contains excerpts from the communications system design used to test the functionality of AdaGen and C-Gen. The design uses many components in the TimeBench design notation and therefore tests a significant portion of the transformation rules in AdaGen and C-Gen.

Shown in Figure B.1 is the overall design. The communication system consists of two nodes, a source node and a destination node. The source node sends messages to the destination node via a link running the alternating bit protocol. The actors **src** and **dst** represent the OSI [42] host layers and the packages **comm1** and **comm2** represent the OSI link layers in each of the communication nodes. The courier actors pass messages between the host layer and the link layer in each node and also simulate the physical channel between the nodes.

The source component is shown in Figure B.2. The purpose of the source component is to generate data to send to the destination node. This data is sent to the link layer when actor **courier1** rendezvous at gate **s_pduDn**. Although actor **source** has the capability to receive data at gate **s_pduUp**, this gate is permanently closed for the purposes of the example.

The contents of one of the courier tasks is shown in Figure B.3. The purpose of all the couriers is to transfer messages from one actor to another. To do this the couriers rendezvous (directly or indirectly through a package interface) with one

Figure B.2: Contents of actor source.

actor to fetch the data and then rendezvous with another actor to transfer the data.

Shown in Figure B.4 is the design of the source node's link layer. The alternating bit protocol used to control the flow of messages from the source node to the destination node is handled by actor `comm1`, while the timer, which causes actor `comm1` to resend messages when an acknowledgement is not received after a given period of time, is handled by actor `timer`.

Figures B.5 and B.6 show the contents of actor `comm1`. A message is sent from the host layer to the link layer of the source node when actor `courier1` rendezvous with actor `comm1` at gate button `sduDnG1`. The message is then sent to the destination node according to the rules specified by the alternating bit protocol. The state machine in ACM `altbit1` ensures that alternating bit protocol is followed. The message is sent to the destination node when actor `courier4` rendezvous with actor `comm1` at gate button `pduDnG1`. When the message is sent, actor `comm1` rendezvous with actor `timer` at gate button `start` to start the timer. If no acknowledgement is received from the destination node before the timer times out, actor `timer` rendezvous with actor `comm1` at gate button `timeout` which causes the message to be resent. An acknowledgment is received from the destination node when actor `courier3` rendezvous with actor `comm1` at gate button `pduUpG1`. If an acknowledgment is received, actor `comm1` rendezvous with actor `timer` at gate button `stop` to turn off the timer. Once the rendezvous with the timer is completed, actor `comm1` is ready to receive another message from the host layer.

Shown in Figure B.7 is the contents of the timer component. When actor `timer` accepts a rendezvous at gate button `start`, the timer is turned on. Unless the timer accepts a rendezvous at gate button `stop` before the time out located on pull arc `recv` is activated, push button `control` is activated. When the timer is on, push button `control` activates the push finger `timeout` which causes a time out to occur in actor `comm1`. Time outs which occur when the timer is not turned on are ignored.

The state machine found in ACM `altbit1` is shown in Figure B.8. The state machine is controlled by the events and conditions associated with each state transition arc. The resulting action is determined by the action code found in the return

Figure B.3: Contents of one of the courier actors.

Figure B.4: Contents of box comm1.

Figure B.5: Contents of actor `comm1` (Part 1).

Figure B.6: Contents of actor comm1 (Part 2).

Figure B.7: Contents of actor timer.

Figure B.8: Contents of ACM altbit1.

action field associated with each state transition arc. The events are passed to the state machine by the gate buttons in actor `comm1` (see Figure B.5), while the action code returned from the state machine is interpreted by push button `perform1` (see Figure B.6).

Appendix C

Excerpts of the Ada Program Generated from the Communication System Design

```
WITH Integer_Library, Real_Library, Boolean_Library,  
      String_Unbounded_Unmanaged, List_Single_Unbounded_Unmanaged,  
      Set_Simple_Sequential_Unbounded_Unmanaged;  
PACKAGE MainPackage IS  
  
  TYPE MCLInteger IS RANGE Integer'First .. Integer'Last;  
  TYPE MCLReal IS DIGITS 15;  
  TYPE MCLBoolean IS NEW Boolean;  
  
  PACKAGE MCLIntProcs IS NEW Integer_Library(MCLInteger,  
                                              MCLReal,  
                                              MCLBoolean);  
  
  USE MCLIntProcs;  
  
  PACKAGE MCLRealProcs IS NEW Real_Library(MCLReal,  
                                           MCLInteger,  
                                           MCLBoolean);  
  
  USE MCLRealProcs;  
  
  PACKAGE MCLBoolProcs IS NEW Boolean_Library(MCLBoolean);  
  USE MCLBoolProcs;  
  
  PACKAGE MCLString IS NEW String_Unbounded_Unmanaged(MCLInteger,  
                                                       MCLBoolean);  
  USE MCLString;  
  
  TASK TYPE srcType IS  
    ENTRY s_pduDn(pdu : OUT MCLString.String);  
    ENTRY s_pduUp;
```

```

END srcType;
src : srcType;

:

TASK TYPE courier1Type IS
  ENTRY Init1;
END courier1Type;
courier1 : courier1Type;

:

PACKAGE comm1 IS
  PROCEDURE sduDn1(sdu : IN MCLString.String);
  PROCEDURE sduUp1(sdu : OUT MCLString.String);
  PROCEDURE pduDn1(seq : OUT MCLInteger; typ : OUT MCLInteger;
    data : OUT MCLString.String);
  PROCEDURE pduUp1(typ : IN MCLInteger; seq : IN MCLInteger;
    data : IN MCLString.String);
END comm1;

:

PROCEDURE Main;

END MainPackage;

WITH Unchecked_Deallocation, Text_IO;
USE Text_IO;
PACKAGE BODY MainPackage IS

  TASK BODY srcType IS

    i : MCLInteger;
    data : MCLString.String;
    size : MCLInteger;
    closed1 : MCLBoolean;

  BEGIN
    i := 1;
    data := Initialize("This is a test.");
    size := Length_Of(data);
    closed1 := False;
  LOOP
    SELECT
      WHEN closed1 = True =>
        ACCEPT s_pduUp DO
          NULL;

```

```

        END s_pduUp;
    OR
    ACCEPT s_pduDn(pdu : OUT MCLString.String) DO
        pdu := Get_Item(data,i);
        Output(Initialize("SRC sending '"));
        Output(Get_Item(data,i));
        Output(Initialize(""));
        New_Line;
    END s_pduDn;
    i := (i MOD size);
    i := (i + 1);
END SELECT;
END LOOP;
END srcType;

```

:

TASK BODY courier1Type IS

```

    pdu : MCLString.String;
    sdu : MCLString.String;

BEGIN
    ACCEPT Init1 DO
        NULL;
    END Init1;
    LOOP
        src.s_pduDn(pdu => pdu);
        sdu := Copy(pdu);
        comm1.sduDn1(sdu => sdu);
    END LOOP;
END courier1Type;

```

:

PACKAGE BODY comm1 IS

```

    send_seq1 : MCLInteger;
    pduDn_data1 : MCLString.String;
    pduDn_type1 : MCLInteger;
    pduDn_seq1 : MCLInteger;
    recv_seq1 : MCLInteger;
    pduUp_type1 : MCLInteger;
    pduUp_seq1 : MCLInteger;
    pduUp_data1 : MCLString.String;
    sduUp1_data : MCLString.String;
    sduDn1_data : MCLString.String;

```

TASK TYPE comm1Type IS

```

ENTRY sduDnG1(sduDn : IN MCLString.String);
ENTRY sduUpG1(sduUp : OUT MCLString.String);
ENTRY pduUpG1(typUp : IN MCLInteger; seqUp : IN MCLInteger;
              dataUp : IN MCLString.String);
ENTRY pduDnG1(seqDn : OUT MCLInteger; typDn : OUT MCLInteger;
              dataDn : OUT MCLString.String);
ENTRY timeout;
END comm1Type;
comm1 : comm1Type;

TASK TYPE timerType IS
  ENTRY start;
  ENTRY stop;
END timerType;
timer : timerType;

PROCEDURE sduDn1(sdu : IN MCLString.String) IS
BEGIN
  comm1.sduDnG1(sduDn => sdu);
END sduDn1;

PROCEDURE sduUp1(sdu : OUT MCLString.String) IS
BEGIN
  comm1.sduUpG1(sduUp => sdu);
END sduUp1;

PROCEDURE pduDn1(seq : OUT MCLInteger; typ : OUT MCLInteger;
                 data : OUT MCLString.String) IS
BEGIN
  comm1.pduDnG1(typDn => typ, seqDn => seq, dataDn => data);
END pduDn1;

PROCEDURE pduUp1(typ : IN MCLInteger; seq : IN MCLInteger;
                 data : IN MCLString.String) IS
BEGIN
  comm1.pduUpG1(typUp => typ, seqUp => seq, dataUp => data);
END pduUp1;

TASK BODY comm1Type IS

  PACKAGE altbit1 IS
    TYPE EVENT_TYPE IS (timeout1, SEND_request1, DATA_response1,
                        init1, RECEIVE_request1);
    TYPE ACTION_TYPE IS (IMPLICIT_ERROR, DATA_request1,
                        RECEIVE_response1, initialize1,
                        data_received1, ack_received1);
    ACM_Error : Exception;
    PROCEDURE DecodeEvent(CurEventVar : IN EVENT_TYPE;
                        CurActionVar : IN OUT ACTION_TYPE;
                        typ : IN MCLInteger;

```

```

                                rcvd_seq : IN MCLInteger;
                                sent_seq : IN MCLInteger);
END altbit1;

anEvent1 : altbit1.EVENT_TYPE;
anAction1 : altbit1.ACTION_TYPE;
sduDnW : MCLBoolean;
sduUpW : MCLBoolean;
pduUpW : MCLBoolean;
pduDnW : MCLBoolean;

PROCEDURE perform1(anEvent1 : IN altbit1.EVENT_TYPE;
                   anAction1 : IN OUT altbit1.ACTION_TYPE);

PROCEDURE perform1(anEvent1 : IN altbit1.EVENT_TYPE;
                   anAction1 : IN OUT altbit1.ACTION_TYPE) IS
BEGIN
    altbit1.DecodeEvent(CurEventVar => anEvent1,
                        CurActionVar => anAction1,
                        typ => pduUp_type1,
                        sent_seq => send_seq1,
                        rcvd_seq => pduUp_seq1);

    CASE anAction1 IS
        WHEN altbit1.initialized =>
            send_seq1 := 0;
            rcv_seq1 := 0;
        WHEN altbit1.DATA_request1 =>
            pduDn_data1 := Copy(sduDn1_data);
            pduDn_seq1 := send_seq1;
            pduDn_type1 := 0;
            pduDnW := True;
            timer.start;
        WHEN altbit1.data_received1 =>
            sduUp1_data := Copy(pduUp_data1);
            pduDn_type1 := 1;
            pduDn_seq1 := pduUp_seq1;
            pduDnW := True;
            sduUpW := True;
            rcv_seq1 := (rcv_seq1 + 1) MOD 2;
        WHEN altbit1.action_received1 =>
            timer.stop;
            send_seq1 := ((send_seq1 + 1) MOD 2);
            sduDnW := True;
            pduUpW := True;
        WHEN altbit1.RECEIVE_response1 =>
            pduUpW := True;
        WHEN OTHERS =>
            Output(Initialize("comm1: Bogus action to perform"));
            New_Line;
    END CASE;

```

END perform1;

PACKAGE BODY altbit1 IS

TYPE STATE_TYPE IS (start1,estab1,ack_wait1);
CurStateVar : STATE_TYPE := start1;

PROCEDURE DecodeEvent(CurEventVar : IN EVENT TYPE;
CurActionVar : IN OUT ACTION_TYPE;
typ : IN MCLInteger;
rcvd_seq : IN MCLInteger;
sent_seq : IN MCLInteger) IS

BEGIN

CASE CurStateVar IS

WHEN start1 =>

IF (CurEventVar = init1) THEN
CurStateVar := estab1;
CurActionVar := initialize1;
RETURN;

END IF;

WHEN estab1 =>

IF (CurEventVar = SEND_request1) THEN
CurStateVar := ack_wait1;
CurActionVar := DATA_request1;
RETURN;

END IF;

IF (CurEventVar = RECEIVE_request1) THEN
CurStateVar := estab1;
CurActionVar := RECEIVE_response1;
RETURN;

END IF;

IF (CurEventVar = DATA_response1) THEN
IF (Is_Equal(typ,1)
AND Is_Equal(rcvd_seq,sent_seq)) THEN
CurStateVar := estab1;
CurActionVar := data_received1;
RETURN;

END IF;

END IF;

WHEN ack_wait1 =>

IF (CurEventVar = RECEIVE_request1) THEN
CurStateVar := ack_wait1;
CurActionVar := RECEIVE_response1;
RETURN;

END IF;

IF (CurEventVar = DATA_response1) THEN
IF (Is_Equal(typ,1)
AND Not_Equal(rcvd_seq,sent_seq)) THEN
CurStateVar := ack_wait1;
CurActionVar := data_received1;

```

        RETURN;
    END IF;
END IF;
IF (CurEventVar = DATA_response1) THEN
    IF (Is_Equal(typ,1)
        AND Is_Equal(rcvd_seq,sent_seq)) THEN
        CurStateVar := establ;
        CurActionVar := ack_received1;
        RETURN;
    END IF;
END IF;
IF (CurEventVar = timeout1) THEN
    CurStateVar := ack_wait1;
    CurActionVar := DATA_request1;
    RETURN;
END IF;
END CASE;
CurActionVar := IMPLICIT_ERROR;
END DecodeEvent;
END altbit1;

BEGIN
    sduUpW := False;
    pduDnW := False;
    sduDnW := True;
    pduUpW := True;
    anEvent1 := altbit1.init1;
    perform1(anEvent1 => anEvent1,anAction1 => anAction1);
LOOP
    SELECT
        ACCEPT timeout DO
            NULL;
        END timeout;
        anEvent1 := altbit1.timeout1;
        perform1(anEvent1 => anEvent1,anAction1 => anAction1);
    OR
        WHEN sduUpW = True =>
            ACCEPT sduUpG1(sduUp : OUT MCLString.String) DO
                sduUpW := False;
                anEvent1 := altbit1.RECEIVE_request1;
                perform1(anEvent1 => anEvent1,
                    anAction1 => anAction1);
                sduUp := Copy(sduUp1_data);
            END sduUpG1;
    OR
        WHEN pduUpW = True =>
            ACCEPT pduUpG1(typUp : IN MCLInteger;
                seqUp : IN MCLInteger;
                dataUp : IN MCLString.String) DO
                pduUpW := False;

```



```

        pduUp_type1 := typUp;
        pduUp_seq1 := seqUp;
        pduUp_data1 := Copy(dataUp);
    END pduUpG1;
    anEvent1 := altbit1.DATA_response1;
    perform1(anEvent1 => anEvent1, anAction1 => anAction1);
OR
    WHEN sduDnW = True =>
        ACCEPT sduDnG1(sduDn : IN MCLString.String) DO
            sduDnW := False;
            sduDn1_data := Copy(sduDn);
        END sduDnG1;
        anEvent1 := altbit1.SEND_request1;
        perform1(anEvent1 => anEvent1, anAction1 => anAction1);
OR
    WHEN pduDnW = True =>
        ACCEPT pduDnG1(seqDn : OUT MCLInteger;
                       typDn : OUT MCLInteger;
                       dataDn : OUT MCLString.String) DO
            pduDnW := False;
            dataDn := Copy(pduDn_data1);
            typDn := pduDn_type1;
            seqDn := pduDn_seq1;
        END pduDnG1;
    END SELECT;
END LOOP;
END comm1Type;

TASK BODY timerType IS

    OnOrOff : MCLBoolean;

    PROCEDURE control(OnOrOff : IN OUT MCLBoolean);

    PROCEDURE control(OnOrOff : IN OUT MCLBoolean) IS
    BEGIN
        IF OnOrOff THEN
            OnOrOff := False;
            comm1.timeout;
        END IF;
    END control;

BEGIN
    OnOrOff := False;
    LOOP
        SELECT
            ACCEPT start DO
                OnOrOff := True;
            END start;
        OR

```

```

        ACCEPT stop DO
            OnOrOff := False;
        END stop;
    OR
        DELAY Duration(100.000000);
        control(OnOrOff => OnOrOff);
    END SELECT;
END LOOP;
END timerType;

END comm1;

:

PROCEDURE Main IS
BEGIN
    courier1.Init1;
    courier2.Init2;
    courier3.Init3;
    courier4.Init4;
    courier5.Init5;
    courier6.Init6;
END Main;

END MainPackage;

WITH MainPackage;
PROCEDURE Start IS
BEGIN
    MainPackage.Main;
END Start;

```

Appendix D

AdaGen Rules Used in the Transformation of the Communication System Design

```
/* Root is the 'bootstrap' production rule as it is the rule that is
   invoked by TimeBench. Root generates the main package that encloses
   the entire Ada program. */
root(RootNode) -->
  {get_all_children(RootNode, ChildList),
   print_literal("WITH Integer_Library, Real_Library,
                 Boolean_Library,"),
   print_newline,
   print_literal("    String_Unbounded_Unmanaged,
                 List_Single_Unbounded_Unmanaged,"),
   print_newline,
   print_literal("    Set_Simple_Sequential_Unbounded_Unmanaged;"),
   print_newline,
   print_literal("PACKAGE MainPackage IS"),
   print_newline, print_newline, indent(3),
   get_all_compound_types(TypeList)},
  simple_type_declarations,
  compound_type_declaration_list(TypeList),
  basic_declaration_list(ChildList),
  {print_literal("PROCEDURE Main;"), print_newline, print_newline,
   indent(-3),
   print_literal("END MainPackage;"), print_newline, print_newline,
   print_literal("WITH Unchecked_Deallocation, Text_IO;"),
   print_newline,
   print_literal("USE Text_IO;"), print_newline,
   print_literal("PACKAGE BODY MainPackage IS"),
   print_newline, print_newline, indent(3)},
  body_declaration_list(ChildList),
  {print_literal("PROCEDURE Main IS"), print_newline,
   print_literal("BEGIN"), print_newline, indent(3)},
```

```

(({get_engine_node(ChildList,EngineNode)),
  code_block(EngineNode)) |
  {print_literal("NULL;"), print_newline}},
{indent(-3),
  print_literal("END Main;"), print_newline, print_newline,
  indent(-3),
  print_literal("END MainPackage;"),
  print_newline, print_newline},
start_up_proc,
{free_list(ChildList)}).

/* Generates the procedure needed by the Verdex Ada compiler to start
the program executing. */
start_up_proc -->
  {print_literal("WITH MainPackage;"), print_newline,
  print_literal("PROCEDURE Start IS"), print_newline,
  print_literal("BEGIN"), print_newline,
  indent(3),
  print_literal("MainPackage.Main;"), print_newline,
  indent(-3),
  print_literal("END Start;"), print_newline}.

/* Generates Ada type declarations for MCL's integer, real, boolean,
and string types. */
simple_type_declarations -->
  {print_literal("TYPE MCLInteger IS RANGE
    Integer'First .. Integer'Last;"), print_newline,
  print_literal("TYPE MCLReal IS DIGITS 15;"), print_newline,
  print_literal("TYPE MCLBoolean IS NEW Boolean;"),
  print_newline, print_newline,
  print_literal("PACKAGE MCLIntProcs IS NEW Integer_Library
    (MCLInteger,MCLReal,MCLBoolean);"), print_newline,
  print_literal("USE MCLIntProcs;"), print_newline, print_newline,
  print_literal("PACKAGE MCLRealProcs IS NEW Real_Library
    (MCLReal,MCLInteger,MCLBoolean);"), print_newline,
  print_literal("USE MCLRealProcs;"), print_newline, print_newline,
  print_literal("PACKAGE MCLBoolProcs IS NEW Boolean_Library
    (MCLBoolean);"), print_newline,
  print_literal("USE MCLBoolProcs;"), print_newline, print_newline,
  print_literal("PACKAGE MCLString IS NEW String_Unbounded_Unmanaged
    (MCLInteger,MCLBoolean);"), print_newline,
  print_literal("USE MCLString;"), print_newline, print_newline}.

/* These next three rules control the order in which the MachineChart
structural components are processed so that Ada's order of
compilation rules are satisfied. */
declarative_part(ITEMS) -->
  basic_declaration_list(ITEMS),
  body_declaration_list(ITEMS).

```

```

basic_declaration_list(ITEMS) -->
    {get_all_ACMs(ITEMS,ACMNodes)},
    an_ACM_declaration_list(ACMNodes),
    {get_all_data_nodes(ITEMS,DataNodes)},
    object_declaration_list(DataNodes),
    {get_all_gates(ITEMS,GateNodes)},
    local_gate_object_list(GateNodes),
    {get_all_gates(ITEMS,GateNodes)},
    gate_declaration_list(GateNodes),
    {get_all_buttons(ITEMS,ButtonNodes)},
    subprogram_declaration_list(ButtonNodes),
    {get_all_actors(ITEMS,ActorNodes)},
    task_declaration_list(ActorNodes),
    {get_all_reactors(ITEMS,ReactorNodes)},
    task_declaration_list(ReactorNodes),
    {get_all_boxes(ITEMS,BoxNodes)},
    package_declaration_list(BoxNodes).

body_declaration_list(ITEMS) -->
    {get_all_buttons(ITEMS,ButtonNodes)},
    subprogram_body_list(ButtonNodes),
    {get_all_actors(ITEMS,ActorNodes)},
    task_body_list(ActorNodes),
    {get_all_reactors(ITEMS,ReactorNodes)},
    special_task_body_list(ReactorNodes),
    {get_all_boxes(ITEMS,BoxNodes)},
    package_body_list(BoxNodes),
    {get_all_ACMs(ITEMS,ACMNodes)},
    an_ACM_body_list(ACMNodes).

/* These next two rules generate Ada variable declarations from the
   list of MCL variable declarations found in "DataNodes" */
object_declaration_list(DataNodes) -->
    {is_empty(DataNodes)} |
    ({extract_element(DataNodes,DataNode,DNTail)},
     object_declaration(DataNode),
     object_declaration_list(DNTail)).

object_declaration(DataNode) -->
    identifier(DataNode),
    {print_literal(" : ")},
    get_type_node(DataNode,TypeNode)},
    subty, indication(TypeNode),
    (({get_initial_expression(DataNode,InitExpression),
       print_literal(" := ")},
       expression(InitExpression),
       {print_literal(";"), print_newline, print_newline}) |
     {print_literal(";"), print_newline, print_newline}).

/* These next two rules generate the Ada subtype indication for the

```

```

variable declaration using the MCL type in "TypeNode" */
subtype_indication(TypeNode) -->
  simple_type_declaration(TypeNode) |
  selected_component(TypeNode).

simple_type_declaration(TypeNode) -->
  {is_integer(TypeNode), print_literal("MCLInteger")} |
  {is_real(TypeNode), print_literal("MCLReal")} |
  {is_boolean(TypeNode), print_literal("MCLBoolean")} |
  {is_string(TypeNode), print_literal("MCLString.String")} |
  ({is_list(TypeNode),
   get_parent_type(TypeNode,ListParentType),
   print_literal("MCLListOf")},
   complete_compound_name(ListParentType),
   {print_literal(".List")}) |
  ({is_set(TypeNode),
   get_parent_type(TypeNode,SetParentType),
   print_literal("MCLSetOf")},
   complete_compound_name(SetParentType),
   {print_literal(".Set")}).

/* These next two rules generate Ada procedure declarations from the
list of push buttons contained in "ButtonNodes". */
subprogram_declaration_list(ButtonNodes) -->
  {is_empty(ButtonNodes)} |
  ({extract_element(ButtonNodes,ButtonNode,BNTail)},
   subprogram_declaration(ButtonNode),
   subprogram_declaration_list(BNTail)).

subprogram_declaration(ButtonNode) -->
  subprogram_specification(ButtonNode),
  {print_literal(";"), print_newline, print_newline}.

/* Generates an Ada procedure specification from a Machine Chart push
button. */
subprogram_specification(ButtonNode) --> /* not supporting functions */
  {print_literal("PROCEDURE ")},
  identifier(ButtonNode),
  {get_formal_parameters(ButtonNode,FormalParms)},
  ({is_empty(FormalParms)} |
   ({print_literal("("),
    formal_parameter_list(FormalParms),
    {print_literal(")")})).

/* The next three rules generate Ada formal parameters from either
formal parameters associated with MachineChart push buttons or
formal parameters associated with MachineChart gate buttons. */
formal_parameter_list(FormalParms) -->
  {extract_element(FormalParms,FormalParm,FPTail)},
  parameter_specification(FormalParm),

```

```

({is_empty(FPTail)} |
  ({print_literal("; ")}),
  formal_parameter_list(FPTail))).

parameter_specification(FormalParm) -->
  identifier(FormalParm),
  {print_literal(" : ")},
  {get_mode_type(FormalParm,ModeType)},
  mode(ModeType),
  {get_type_node(FormalParm,TypeNode)},
  subtype_indication(TypeNode).

mode(ModeType) -->
  {is_in_mode(ModeType), print_literal("IN ")} |
  {is_out_mode(ModeType), print_literal("OUT ")} |
  {is_inout_mode(ModeType), print_literal("IN OUT ")}.

/* These next two rules generate task specifications from the list of
   actors or reactors in "RobotNodes". */
task_declaration_list(RobotNodes) -->
  {is_empty(RobotNodes)} |
  ({extract_element(RobotNodes,RobotNode,RNTail)},
   task_specification(RobotNode),
   task_declaration_list(RNTail)).

task_specification(RobotNode) -->
  {is_robot_pool(RobotNode),
   print_error("Robot pools cannot be transformed.")} |
  ({print_literal("TASK TYPE ")}),
  identifier(RobotNode),
  {print_literal("Type IS"), print_newline,
   get_external_children(RobotNode,ChildList),
   get_all_gates(ChildList,GateNodes)},
  ({is_empty(GateNodes)} |
   ({indent(3)},
    entry_declaration_list(GateNodes),
    {indent(-3)})),
  {print_literal("END ")}), identifier(RobotNode),
  {print_literal("Type;"), print_newline},
  (({is_dynamic(RobotNode), /* Robot is installed dynamically */
    print_literal("TYPE ")}),
   identifier(RobotNode), {print_literal("Access IS ACCESS ")}),
   identifier(RobotNode), {print_literal("Type;"), print_newline},
   identifier(RobotNode), {print_literal(" : ")}),
   identifier(RobotNode), {print_literal("Access;"), print_newline}) |
  (identifier(RobotNode), {print_literal(" : ")}),
   identifier(RobotNode), {print_literal("Type;"), print_newline})),
  {print_newline,
   free_list(ChildList)}).

```

```

/* These next two rules generate task entry declarations from the list
   of gate buttons contained in "GateNodes". */
entry_declaration_list(GateNodes) -->
  {extract_element(GateNodes, GateNode, GNTail)},
  entry_declaration(GateNode),
  ({is_empty(GNTail)} |
   entry_declaration_list(GNTail)).

entry_declaration(GateNode) -->
  {print_literal("ENTRY ")},
  identifier(GateNode),
  {get_formal_parameters(GateNode, FormalParms)},
  ({is_empty(FormalParms)} |
   ({print_literal("("),
    formal_parameter_list(FormalParms),
    print_literal(")"))},
   {print_literal(";"), print_newline}).

/* These next two rules generate Ada variable declarations for any
   waiting places associated with the list of gate buttons contained
   in "GateNodes". */
gate_declaration_list(GateNodes) -->
  {is_empty(GateNodes)} |
  ({extract_element(GateNodes, GateNode, GNTail)},
   ((gate_declaration(GateNode),
    gate_declaration_list(GNTail)) |
    gate_declaration_list(GNTail))).

gate_declaration(GateNode) -->
  {get_waiting_place(GateNode, WaitingPlace)},
  identifier(WaitingPlace),
  {print_literal(" : MCLBoolean;"), print_newline, print_newline}.

/* These next two rules generate Ada package specifications from the
   list of boxes contained in "BoxNodes". */
package_declaration_list(BoxNodes) -->
  {is_empty(BoxNodes)} |
  ({extract_element(BoxNodes, BoxNode, BNTail)},
   package_specification(BoxNode),
   package_declaration_list(BNTail)).

package_specification(BoxNode) -->
  {print_literal("PACKAGE ")}, identifier(BoxNode),
  {print_literal(" IS ")}, print_newline, print_newline,
  indent(3),
  get_external_children(BoxNode, ChildList)},
  basic_declaration_list(ChildList),
  {indent(-3),
   print_literal("END ")}, identifier(BoxNode),
  {print_literal(";"), print_newline, print_newline,

```



```

    free_list(ChildList)}.

/* These next two rules generate special Ada package specifications
   from the list of ACM's contained in "ACMNodes". */
an_ACM_declaration_list(ACMNodes) -->
  {is_empty(ACMNodes)} |
  ({extract_element(ACMNodes,ACMNode,ACMTail)},
   an_ACM_specification(ACMNode),
   an_ACM_declaration_list(ACMTail)).

an_ACM_specification(ACMNode) -->
  {print_literal("PACKAGE ")}, identifier(ACMNode),
  {print_literal(" IS ")}, print_newline, print_newline,
  indent(3)},
  event_type_declaration(ACMNode),
  action_type_declaration(ACMNode),
  {print_literal("ACM_Error : Exception;"), print_newline,
   print_literal("PROCEDURE DecodeEvent"),
   get_formal_parameters(ACMNode,FormalParms),
   print_literal("(CurEventVar : IN EVENT_TYPE;
                  CurActionVar : IN OUT ACTION_TYPE)"),
   ({is_empty(FormalParms)} |
    ({print_literal("; ")},
     formal_parameter_list(FormalParms))),
   {print_literal(");"), print_newline, print_newline,
    indent(-3),
    print_literal("END ")}},
  identifier(ACMNode),
  {print_literal(";"), print_newline, print_newline}.

/* Generates an Ada enumerated type declaration from the list of
   events associated with the ACM in "ACMNode". */
event_type_declaration(ACMNode) -->
  {print_literal("TYPE EVENT_TYPE IS ("),
   get_all_events(ACMNode,EventList)},
  enum_list(EventList),
  {print_literal(");"), print_newline}.

/* Generates an Ada enumerated type declaration from the list of
   actions associated with the ACM in "ACMNode". */
action_type_declaration(ACMNode) -->
  {print_literal("TYPE ACTION_TYPE IS (IMPLICIT_ERROR,"),
   get_all_actions(ACMNode,ActionList)},
  enum_list(ActionList),
  {print_literal(");"), print_newline}.

/* Generates an Ada enumerated type declaration from the list of
   states contained in "StateNodes". */
state_type_declaration(StateNodes) -->
  {print_literal("TYPE STATE_TYPE IS ("),

```

```

    enum_list(StateNodes),
    {print_literal(";"), print_newline}.

/* Generates a special Ada variable declaration to maintain the
   current state of the SM inside the ACM. The variable is initialized
   with the value of the start state. */
state_var_declaration(StartState) -->
    {print_literal("CurStateVar : STATE_TYPE := "),
     identifier(StartState),
     {print_literal(";"), print_newline}.

/* Generates the enumeration constants for the enumerated type
   declarations. */
enum_list(EnumList) -->
    {extract_element(EnumList,ID,EnumTail)},
    identifier(ID),
    ({is_empty(EnumTail)} |
     ({print_literal(", ")}),
     enum_list(EnumTail))).

/* These next two rules generate Ada procedures from the list of
   buttons contained in "ButtonNodes". */
subprogram_body_list(ButtonNodes) -->
    {is_empty(ButtonNodes)} |
    ({extract_element(ButtonNodes,ButtonNode,BNTail)},
     subprogram_body(ButtonNode),
     subprogram_body_list(BNTail)).

subprogram_body(ButtonNode) -->
    subprogram_specification(ButtonNode),
    {print_literal(" IS "), print_newline,
     indent(3),
     get_all_children(ButtonNode,ChildList)},
    declarative_part(ChildList),
    {indent(-3),
     print_literal("BEGIN"), print_newline,
     indent(3)},
    (({get_engine_node(ChildList,EngineNode)},
     code_block(EngineNode)) |
     {print_literal("NULL;"), print_newline}),
    {indent(-3),
     print_literal("END ")}, identifier(ButtonNode),
    {print_literal(";"), print_newline, print_newline,
     free_list(ChildList)}.

/* These next two rules generate task bodies from the list of actors
   contained in "RobotNodes". */
task_body_list(RobotNodes) -->
    {is_empty(RobotNodes)} |
    ({extract_element(RobotNodes,RobotNode,RNTail)},

```

```

    task_body(RobotNode),
    task_body_list(RNTail)).

task_body(RobotNode) -->
{print_literal("TASK BODY ")}, identifier(RobotNode),
{print_literal("Type IS"), print_newline, print_newline,
 indent(3),
 get_all_children(RobotNode,ChildList)},
declarative_part(ChildList),
{indent(-3),
 print_literal("BEGIN"), print_newline,
 indent(3)},
(({get_engine_node(ChildList,EngineNode)},
 code_block(EngineNode)) |
 {print_literal("NULL;"), print_newline}),
{indent(-3),
 print_literal("END ")}, identifier(RobotNode),
{print_literal("Type;"), print_newline, print_newline,
 free_list(ChildList)}.

/* These next two rules generate package bodies from the list of boxes
   contained in "BoxNodes". */
package_body_list(BoxNodes) -->
{is_empty(BoxNodes)} |
({extract_element(BoxNodes,BoxNode,BNTail)},
 package_body(BoxNode),
 package_body_list(BNTail)).

package_body(BoxNode) -->
{print_literal("PACKAGE BODY ")}, identifier(BoxNode),
{print_literal(" IS "), print_newline, print_newline,
 indent(3),
 get_internal_children(BoxNode,INTChildList)},
basic_declaration_list(INTChildList),
{get_all_children(BoxNode,ChildList)},
body_declaration_list(ChildList),
{indent(-3)},
(({get_engine_node(INTChildList,EngineNode),
 print_literal("BEGIN"), print_newline,
 indent(3)},
 code_block(EngineNode),
 {indent(-3),
 print_literal("END ")}, identifier(BoxNode)) |
 ({print_literal("END ")}, identifier(BoxNode))),
{print_literal(";"), print_newline, print_newline,
 free_list(INTChildList),
 free_list(ChildList)}.

/* These next two rules generate special package bodies from the list
   of ACM's contained in "ACMNodes". */

```

```

an_ACM_body_list(ACMNodes) -->
  {is_empty(ACMNodes)} |
  ({extract_element(ACMNodes,ACMNode,ACMTail)},
   an_ACM_body(ACMNode),
   an_ACM_body_list(ACMTail)).

an_ACM_body(ACMNode) -->
  {print_literal("PACKAGE BODY ")}, identifier(ACMNode),
  {print_literal(" IS"), print_newline, print_newline,
   indent(3),
   get_internal_children(ACMNode,ChildList),
   get_all_data_nodes(ChildList,DataNodes),
   get_all_states(ChildList,StateNodes1),
   get_all_states(ChildList,StateNodes2)},
  state_type_declaration(StateNodes1),
  {get_start_state(StateNodes2,StartState)},
  state_var_declaration(StartState),
  object_declaration_list(DataNodes),
  {print_newline,
   print_literal("PROCEDURE DecodeEvent"),
   get_formal_parameters(ACMNode,FormalParms),
   print_literal("(CurEventVar : IN EVENT_TYPE;
                  CurActionVar : IN OUT ACTION_TYPE)"),
   ({is_empty(FormalParms)} |
    ({print_literal("; ")},
     formal_parameter_list(FormalParms))),
   {print_literal(" IS"), print_newline,
    print_literal("BEGIN"), print_newline,
    indent(3),
    print_literal("CASE CurStateVar IS"), print_newline,
    indent(3)},
   an_ACM_case_stmt(StateNodes2,ACMNode),
   {indent(-3),
    print_literal("END CASE;"), print_newline,
    print_literal("CurActionVar := IMPLICIT_ERROR;"), print_newline,
    indent(-3),
    print_literal("END DecodeEvent;"), print_newline,
    indent(-3),
    print_literal("END ")}, identifier(ACMNode),
   {print_literal(";"), print_newline, print_newline,
    free_list(ChildList)}}.

/* Processes the MCL code contained inside the engines of push
   buttons, actors, and boxes. */
code_block(EngineNode) -->
  {get_MCL_code(EngineNode,MCL),
   get_local_declarations(EngineNode,DeclarationList1),
   get_local_declarations(EngineNode,DeclarationList2)},
  ({is_empty(MCL),
   print_literal("NULL;"), print_newline} |

```

```

    (({is_empty(DeclarationList1),
      free_list(DeclarationList2)},
      sequence_of_statements(MCL)) |
    ({print_literal("DECLARE"), print_newline,
      indent(3)},
      object_declaration_list(DeclarationList2),
      {indent(-3),
        print_literal("BEGIN"), print_newline,
        indent(3)},
      sequence_of_statements(MCL),
      {indent(-3),
        print_literal("END;"), print_newline,
        free_list(DeclarationList1)}})).

/* The next two rules process the list of MCL statements contained in
"MCL". The order of the list is maintained by the second rule. */
sequence_of_statements(MCL) -->
  {is_empty(MCL)} |
  ({extract_element(MCL,MCLStmt,MCLTail)},
   statement(MCLStmt),
   sequence_of_statements(MCLTail)).

statement(MCLStmt) -->
  null_stmt(MCLStmt) |
  assignment_stmt(MCLStmt) |
  exit_stmt(MCLStmt) |
  delay_stmt(MCLStmt) |
  push_stmt(MCLStmt) |
  return_stmt(MCLStmt) |
  if_stmt(MCLStmt) |
  loop_stmt(MCLStmt) |
  case_stmt(MCLStmt) |
  pull_stmt(MCLStmt) |
  reply_stmt(MCLStmt) |
  print_stmt(MCLStmt).

/* Generates an Ada null statement from an MCL null statement. */
null_stmt(MCLStmt) -->
  {is_null_stmt(MCLStmt),
   print_literal("NULL;"), print_newline}.

/* Kludge for Ron's code. */
reply_stmt(MCLStmt) -->
  {is_reply_stmt(MCLStmt)}.

/* Generates an Ada assignment statement or procedure call from an MCL
assignment statement. MCL only supports deep assignment. */
assignment_stmt(MCLStmt) -->
  {is_assignment_stmt(MCLStmt),
   get_lhs(MCLStmt,LHS),

```

```

    get_rhs(MCLStmt,RHS)),
    ({is_slice(LHS)}, slice_assignment(LHS,RHS)) |
    ({is_index_component(LHS)}, index_assignment(LHS,RHS)) |
    ({is_var_name(RHS)}, copy_assignment(LHS,RHS)) |
    (simple_var(LHS), {print_literal(" := ")}),
    expression(RHS), {print_literal(";"), print_newline})).

/* Generates an Ada function call to support deep assignment of
   strings, lists and sets when a variable is detected on the right
   hand side of the assignment operator in a MCL assignment statement. */
copy_assignment(LHS,RHS) -->
    {is_list_expression(RHS) |
    is_set_expression(RHS) |
    is_string_expression(RHS)},
    identifier(LHS),
    {print_literal(" := Copy(")}, identifier(RHS),
    {print_literal(");"), print_newline}.

/* Determines the type of Ada statement to generate from the push arc
   associated with the MCL push statement being processed. */
push_stmt(MCLStmt) -->
    {is_push_stmt(MCLStmt),
    get_arc(MCLStmt,Arc)},
    (procedure_call(Arc) |
    async_entry_call(Arc) |
    timed_entry_call(Arc) |
    regular_entry_call(Arc) |
    install_stmt(Arc) |
    remove_stmt(Arc) |
    an_ACM_call(Arc)).

/* Generates an Ada procedure call from the push arc whose destination
   is a MachineChart push button. */
procedure_call(Arc) -->
    {is_push_arc(Arc),
    get_destination(Arc,Destination),
    get_parent(Destination,Parent)},
    ({is_internal_button(Destination)},
    identifier(Destination)) |
    (identifier(Parent), {print_literal(".")},
    identifier(Destination)),
    {get_actual_parms(Arc,ActualParms)},
    ({is_empty(ActualParms)} |
    ({print_literal("(")},
    actual_parameter_list(ActualParms),
    {print_literal(")")})),
    {print_literal(";"), print_newline}.

/* The next two rules generate the actual parameters for procedure
   calls or task entry calls from the list of data flows contained in

```

```

"ActualParms". */
actual_parameter_list(ActualParms) -->
  {extract_element(ActualParms,ActualParm,APTail)},
  actual_parameter_part(ActualParm),
  ({is_empty(APTail)} |
   ({print_literal(", ")}),
   actual_parameter_list(APTail))).

actual_parameter_part(ActualParm) -->
  {get_formal_parm(ActualParm,FormalParm)},
  identifier(FormalParm),
  {print_literal(" => ")},
  identifier(ActualParm).

/* The next two rules generate an Ada entry call from a MachineChart
   push arc whose destination is a gatc button. */
regular_entry_call(Arc) -->
  {is_rendezvous_arc(Arc)},
  entry_call_stmt(Arc).

entry_call_stmt(Arc) -->
  {get_destination(Arc,Destination),
   get_parent(Destination,Parent)},
  (({is_root(Parent)},
    {print_literal("MainPackage.")}) |
   (identifier(Parent),
    {print_literal(".")})),
  (({is_robot_pool(Parent)},
    identifier(Destination),
    {print_literal("(Num)")} |
   identifier(Destination)),
  {get_actual_parms(Arc,ActualParms)},
  ({is_empty(ActualParms)} |
   ({print_literal("(")}),
   actual_parameter_list(ActualParms),
   {print_literal(")"}))),
  {print_literal(";"), print_newline}.

/* The next three rules generate an Ada if statement from an MCL if
   statement. */
if_stmt(MCLStmt) -->
  {is_if_stmt(MCLStmt),
   get_then_block(MCLStmt,ThenStmts),
   print_literal("IF "),
   get_condition(MCLStmt,Condition)},
  expression(Condition),
  {print_literal(" THEN"), print_newline,
   indent(3)},
  sequence_of_statements(ThenStmts),
  {indent(-3)},

```

```

(({get_elsif_block(MCLStmt,ElsIfStmts)},
  else_if_stmts(ElsIfStmts),
  (({get_else_block(MCLStmt,ElseStmts),
    print_literal("ELSE"), print_newline,
    indent(3)},
    sequence_of_statements(ElseStmts),
    {indent(-3),
      print_literal("END IF;"), print_newline})) |
    {print_literal("END IF;"), print_newline})) |
  ({get_else_block(MCLStmt,ElseStmts),
    print_literal("ELSE"), print_newline,
    indent(3)},
    sequence_of_statements(ElseStmts),
    {indent(-3),
      print_literal("END IF;"), print_newline})) |
    {print_literal("END IF;"), print_newline})).

else_if_stmts(ElseIfNodes) -->
  {is_empty(ElseIfNodes)} |
  ({extract_element(ElseIfNodes,ElseIfNode,EINTail)},
    else_if_stmt(ElseIfNode),
    else_if_stmts(EINTail)).

else_if_stmt(ElseIfNode) -->
  {print_literal("ELSIF "),
    get_condition(ElseIfNode,Condition)},
  expression(Condition),
  {print_literal(" THEN"), print_newline,
    indent(3),
    get_then_block(ElseIfNode,ThenStatements)},
  sequence_of_statements(ThenStatements),
  {indent(-3)}.

/* Generates an Ada simple loop, while loop or for loop statement from
   an MCL simple loop, while loop or for loop, respectively. */
loop_stmt(MCLStmt) -->
  ({is_simple_loop_stmt(MCLStmt)},
    {print_literal("LOOP"), print_newline,
      indent(3),
      get_code_block(MCLStmt,CodeBlock)},
    sequence_of_statements(CodeBlock),
    {indent(-3),
      print_literal("END LOOP;"), print_newline}) |
  ({is_while_loop_stmt(MCLStmt),
    get_iteration_scheme(MCLStmt,Condition)},
    {print_literal("WHILE ")}, expression(Condition),
    {print_literal(" LOOP"), print_newline,
      indent(3),
      get_code_block(MCLStmt,CodeBlock)},
    sequence_of_statements(CodeBlock),

```



```

        {indent(-3),
         print_literal("END LOOP;"), print_newline} |
        ({is_for_loop_stmt(MCLStmt)},
         process_for_loop(MCLStmt)).

/* The next six rules generate an Ada case statement from a MCL case
statement. */
case_stmt(MCLStmt) -->
    {is_case_stmt(MCLStmt),
     print_literal("CASE "),
     get_switch(MCLStmt,Switch)},
    expression(Switch),
    {print_literal(" IS"), print_newline,
     indent(3),
     get_alternatives(MCLStmt,Alternatives)},
    case_alternative_list(Alternatives),
    {indent(-3),
     print_literal("END CASE;"), print_newline}.

case_alternative_list(Alternatives) -->
    {is_empty(Alternatives)} |
    ({extract_element(Alternatives,Alternative,ATail)},
     case_alternative(Alternative),
     case_alternative_list(Alternatives)).

case_alternative(Alternative) -->
    {print_literal("WHEN "),
     get_choices(Alternative,Choices)},
    choice_list(Choices),
    {print_literal(" =>"), print_newline,
     indent(3),
     get_code_block(Alternative,MCL)},
    sequence_of_statements(MCL),
    {indent(-3)}.

choice_list(Choices) -->
    {extract_element(Choices,Choice,CTail)},
    choice(Choice),
    ({is_empty(CTail)} |
     ({print_literal(" | ")}),
     choice_list(CTail))).

choice(Choice) -->
    discrete_range(Choice) |
    others_default(Choice) |
    expression(Choice).

others_default(Choice) -->
    {is_others(Choice),
     print_literal("OTHERS")}.

```

```

/* Generates a special case statement to mimic the functions performed
   by the SM inside ACM's. One case alternative is generated for every
   state found in "StateNodes". */
an_ACM_case_stmt(StateNodes,ACMNode) -->
  {extract_element(StateNodes,StateNode,SNTail),
   print_literal("WHEN ")} identifier(StateNode),
  {print_literal(" =>"), print_newline,
   indent(3)},
  ({is_end_state(StateNode),
   print_literal("RAISE ACM_Error;")} |
   ({get_all_outgoing_arcs(StateNode,SrcArcs)},
    the_ACM_events_and_actions(SrcArcs))),
  {indent(-3)},
  ({is_empty(SNTail)} |
   an_ACM_case_stmt(SNTail,ACMNode)).

/* Generates Ada if statements from the information associated with
   every transition arc contained in "SrcArcs". Each arc in "SrcArcs"
   has the same state as its source. */
the_ACM_events_and_actions(SrcArcs) -->
  {is_empty(SrcArcs)} |
  ({extract_element(SrcArcs,Arc,Tail)},
   (an_ACM_if_event(Arc) |
    an_ACM_if_condition(Arc) |
    an_ACM_action(Arc) |
    (an_ACM_update_state(Arc),
     an_ACM_return_action(Arc))),
   the_ACM_events_and_actions(Tail)).

/* Generates an Ada if statement from the event list associated with
   the transition arc currently being processed. */
an_ACM_if_event(SrcArc) -->
  {get_arc_event_list(SrcArc,EventList),
   print_literal("IF ")} ,
  process_event(EventList),
  {print_literal(" THEN"), print_newline,
   indent(3)},
  (an_ACM_if_condition(SrcArc) |
   an_ACM_action(SrcArc) |
   (an_ACM_update_state(SrcArc),
    an_ACM_return_action(SrcArc))),
  {indent(-3),
   print_literal("END IF;"), print_newline}.

/* Generates the conditional part of an if statement based on the
   information contained in the event list mentioned above. */
process_event(EventList) -->
  ({is_single_event(EventList),
   print_literal("(CurEventVar = ")} ,

```

```

        identifier(EventList),
        {print_literal("")})} |
    ({get_lhs(EventList,Event),
     get_rhs(EventList,NewEventList)},
     {print_literal("(CurEventVar = ")},
     identifier(Event),
     {print_literal(") OR ")},
     process_event(NewEventList)).

/* Generates an Ada if statement from the condition associated with
the transition arc. */
an_ACM_if_condition(SrcArc) -->
    {get_arc_condition(SrcArc,Condition),
     print_literal("IF ")},
     expression(Condition),
     {print_literal(" THEN"), print_newline,
      indent(3)},
     (an_ACM_action(SrcArc) |
      (an_ACM_update_state(SrcArc),
       an_ACM_return_action(SrcArc))),
     {indent(-3),
      print_literal("END IF;"), print_newline}.

/* Generates a sequence of Ada statements based on the MCL action code
associated with the transition arc. */
an_ACM_action(SrcArc) -->
    {get_arc_action(SrcArc,ActionCode)},
     sequence_of_statements(ActionCode),
     an_ACM_update_state(SrcArc),
     an_ACM_return_action(SrcArc).

/* Generates an Ada assignment statement which updates the current
state in the Ada package to the state located at the destination of
the transition arc. */
an_ACM_update_state(SrcArc) -->
    {get_destination(SrcArc,DestinationState),
     print_literal("CurStateVar := ")},
     identifier(DestinationState),
     {print_literal(";"), print_newline}.

/* Generates an Ada assignment statement which sets the action
variable to the action value associated with the transition arc. */
an_ACM_return_action(SrcArc) -->
    {get_arc_return_action(SrcArc,ReturnAction),
     print_literal("CurActionVar := ")},
     identifier(ReturnAction),
     {print_literal(";"), print_newline,
      print_literal("RETURN;"), print_newline}.

/* Determines which Ada statement to generate from the pull arc

```

```

    associated with the pull statement being processed. */
pull_stmt(MCLStmt) -->
    {is_pull_stmt(MCLStmt),
      get_arc(MCLStmt, PullArc)},
    (select_stmt(PullArc) |
      simple_accept(PullArc)).

/* Generates an Ada select statement from the information contained in
the pull arc. */
select_stmt(PullArc) -->
    (({is_timed(PullArc)},
      ({is_choice(PullArc),
        get_all_destinations(PullArc, Gates),
        get_pull_timer(PullArc, Timer),
        print_literal("SELECT"), print_newline,
        indent(3)},
        accept_alternative_list(Gates),
        {indent(-3),
          print_literal("OR"), print_newline,
          indent(3)},
          delay_alternative(Timer),
          {indent(-3),
            print_literal("END SELECT;"), print_newline})} |
      ({get_destination(PullArc, Gate),
        get_pull_timer(PullArc, Timer)},
        {print_literal("SELECT"), print_newline,
          indent(3)},
          accept_alternative(Gate),
          {indent(-3),
            print_literal("OR"), print_newline,
            indent(3)},
            delay_alternative(Timer),
            {indent(-3),
              print_literal("END SELECT;"), print_newline}})) |
    ({is_choice(PullArc),
      get_all_destinations(PullArc, Gates)},
      {print_literal("SELECT"), print_newline,
        indent(3)},
        accept_alternative_list(Gates),
        {indent(-3),
          print_literal("END SELECT;"), print_newline})).

/* Generates an Ada delay alternative for the select statement from
the time out attached to the pull arc. */
delay_alternative(Timer) -->
    {print_literal("DELAY "),
      get_period(Timer, Time)},
    timer_expression(Time),
    {print_literal(";"), print_newline,
      get_all_outgoing_arcs(Timer, SrcArcs)},

```

```

    timer_arc_list(SrcArcs).

/* Generates the Ada delay period (which must be of type "Duration")
   from the delay period associated with the time out. */
timer_expression(Time) -->
    {print_literal("Duration(")},
    (integer_literal(Time) |
     real_literal(Time) |
     simple_var(Time)),
    {print_literal(")")}

/* Generates an Ada procedure call, entry call, or nested select
   statement if a push arc or a pull arc has the time out as its
   source. */
timer_arc_list(Arcs) -->
    {is_empty(Arcs)} |
    ({extract_element(Arcs,Arc,ATail)},
     process_arc(Arc),
     timer_arc_list(ATail)).

/* Determines the type of Ada statement to generate based on the type
   of arc having the time out as its source. */
process_arc(Arc) -->
    procedure_call(Arc) |
    async_entry_call(Arc) |
    timed_entry_call(Arc) |
    regular_entry_call(Arc) |
    an_ACM_call(Arc) |
    select_stmt(Arc) |
    simple_accept(Arc).

/* These next two rules generate Ada accept statements from the list
   of gate buttons in "Gates". */
accept_alternative_list(Gates) -->
    {extract_element(Gates,Gate,GTail)},
    ((when_condition(Gate),
     {indent(3)},
     accept_alternative(Gate),
     {indent(-3)}} |
     accept_alternative(Gate)),
    ({is_empty(GTail)} |
     ({indent(-3),
      print_literal("OR"), print_newline,
      indent(3)},
     accept_alternative_list(GTail))).

accept_alternative(Gate) -->
    {print_literal("ACCEPT ")},
    identifier(Gate),
    {get_formal_parameters(Gate,FormalParms)},

```

```

({is_empty(FormalParms)} |
  ({print_literal("("),
    formal_parameter_list(FormalParms),
    {print_literal(")"))}),
{print_literal(" DO"), print_newline,
  indent(3)},
(({get_engine(Gate, EngineNode),
  get_critical_MCL_code(EngineNode, MCLCritical),
  get_post_rdv_MCL_code(EngineNode, MCLPostRen)},
sequence_of_statements(MCLCritical),
  {indent(-3),
    print_literal("END ")}}, identifier(Gate),
{print_literal(";"), print_newline},
sequence_of_statements(MCLPostRen)) |
({print_literal("NULL;"), print_newline,
  indent(-3),
  print_literal("END ")}}, identifier(Gate),
{print_literal(";"), print_newline})).

/* If the gate button has a waiting place associated with it, this
   rule generates a guard for the accept alternative in the Ada select
   statement. */
when_condition(Gate) -->
  {get_waiting_place(Gate, WaitingPlace),
  print_literal("WHEN ")},
  identifier(WaitingPlace),
  {print_literal(" = True =>"), print_newline}.

/* Generates an Ada accept statement from a pull arc whose destination
   is a gate button. */
simple_accept(PullArc) -->
  {get_destination(PullArc, Gate)},
  accept_alternative(Gate).

/* The next two rules generate an Ada procedure call from an MCL print
   statement. */
print_stmt(MCLStmt) -->
  {is_print_stmt(MCLStmt),
  get_print_expressions(MCLStmt, PrintExpressions)},
  print_expression_list(PrintExpressions).

print_expression_list(Expressions) -->
  {is_empty(Expressions)} |
  ({extract_element(Expressions, Expression, ExpTail)},
  ({is_newline(Expression),
    print_literal("New_Line;"), print_newline} |
  ({print_literal("Output(")}, expression(Expression),
    {print_literal(");"), print_newline})),
  print_expression_list(ExpTail)).

```

```

/* Generates a special Ada procedure call to invoke the procedure
   contained in the special ACM package. Uses a push arc whose
   destination is an ACM to generate the appropriate information. */
an_ACM_call(Arc) -->
  {is_ACM_push_arc(Arc),
   get_destination(Arc, Destination)},
  identifier(Destination),
  {print_literal(".DecodeEvent"),
   get_actual_event_parm(Arc, ActualEvent),
   get_actual_action_parm(Arc, ActualAction),
   get_actual_parms(Arc, ActualParms),
   print_literal("(CurEventVar => ")}, identifier(ActualEvent),
  {print_literal(", CurActionVar => ")}, identifier(ActualAction),
  ({is_empty(ActualParms)} |
   ({print_literal(", "),
    actual_parameter_list(ActualParms)}),
   {print_literal(");"), print_newline}).

/* The next two rules control the conversion of MCL expressions into
   Ada expressions. This rule also controls the order in which the
   expressions are processed. */
expression_list(Expressions) -->
  {extract_element(Expressions, Expression, ETail)},
  expression(Expression),
  ({is_empty(ETail)} |
   ({print_literal(", "),
    expression_list(ETail)})).

expression(Expression) -->
  expression_component(Expression) |
  type_conversion(Expression) |
  indexed_component(Expression) |
  slice(Expression) |
  selected_component(Expression) |
  random_literal(Expression) |
  string_literal(Expression) |
  list_literal(Expression) |
  set_literal(Expression) |
  integer_literal(Expression) |
  real_literal(Expression) |
  boolean_literal(Expression) |
  simple_var(Expression).

/* Generates an Ada identifier from an MCL identifier. */
simple_var(VarName) -->
  {is_var_name(VarName)},
  identifier(VarName).

/* Generates an Ada function call which has the same functionality as
   a MCL indexing operator. */

```

```

indexed_component(Name) -->
    {is_index_component(Name),
      get_lhs(Name,Symbol),
      get_rhs(Name,Expression)},
    {print_literal("Get_Item("), identifier(Symbol),
      {print_literal(",")}, expression(Expression),
      {print_literal(")")}},

/* Generates an Ada selected component from a MCL selected component. */
selected_component(Name) -->
    {is_selected_component(Name),
      get_lhs(Name,Symbol1),
      get_rhs(Name,Symbol2)},
    identifier(Symbol1),
    {print_literal(".")},
    expression(Symbol2).

/* Generates an Ada function call to create a string which is
   equivalent to the MCL string literal. */
string_literal(Name) -->
    {is_string_literal(Name),
      get_string_literal(Name,Literal),
      print_literal("Initialize("), print_quote,
      print_literal(Literal), print_quote,
      print_literal(")")}},

/* Generates an Ada integer literal from a MCL integer literal. */
integer_literal(Name) -->
    {is_integer_literal(Name),
      get_integer_literal(Name,Literal),
      print_integer(Literal)},

/* Generates an Ada real literal from a MCL real literal. */
real_literal(Name) -->
    {is_real_literal(Name),
      get_real_literal(Name,Literal),
      print_real(Literal)},

/* Generates an Ada boolean literal from a MCL boolean literal. */
boolean_literal(Name) -->
    {is_boolean_literal(Name),
      get_boolean_literal(Name,Literal),
      print_boolean(Literal)},

/* Processes unary and binary expressions containing operators. */
expression_component(Expression) -->
    {is_expression(Expression)},
    (unary_expression(Expression) |
     binary_expression(Expression)).

```



```

/* Processes unary MCL expressions. */
unary_expression(Expression) -->
  {is_unary_expression(Expression),
   get_operator(Expression,Operator),
   get_rhs(Expression,RExpression)},
  (unary_plus_operator(Operator,RExpression) |
   unary_minus_operator(Operator,RExpression) |
   absolute_operator(Operator,RExpression) |
   length_operator(Operator,RExpression) |
   not_operator(Operator,RExpression)).

/* Generates an Ada function call to an Ada function which is
   functionally equivalent to the MCL length operator. */
length_operator(Operator,Expression) -->
  {is_length_operator(Operator)},
  ((({is_string_expression(Expression),
      print_literal("Length_Of(")},
      expression(Expression),
      {print_literal(")")} |
      ({is_list_expression(Expression),
          print_literal("Length_Of(")},
          expression(Expression),
          {print_literal(")")} |
      ({is_set_expression(Expression),
          print_literal("Length_Of(")},
          expression(Expression),
          {print_literal(")"})))).

/* Processes MCL binary expressions. */
binary_expression(Expression) -->
  {is_binary_expression(Expression),
   get_operator(Expression,Operator),
   get_lhs(Expression,LExpression),
   get_rhs(Expression,RExpression)},
  (plus_operator(Operator,LExpression,RExpression) |
   minus_operator(Operator,LExpression,RExpression) |
   multiply_operator(Operator,LExpression,RExpression) |
   divide_operator(Operator,LExpression,RExpression) |
   exponentiation_operator(Operator,LExpression,RExpression) |
   mod_operator(Operator,LExpression,RExpression) |
   max_operator(Operator,LExpression,RExpression) |
   min_operator(Operator,LExpression,RExpression) |
   equal_operator(Operator,LExpression,RExpression) |
   not_equal_operator(Operator,LExpression,RExpression) |
   greater_than_operator(Operator,LExpression,RExpression) |
   less_than_operator(Operator,LExpression,RExpression) |
   greater_than_equal_operator(Operator,LExpression,RExpression) |
   less_than_equal_operator(Operator,LExpression,RExpression) |
   and_operator(Operator,LExpression,RExpression) |
   or_operator(Operator,LExpression,RExpression) |

```

```

xor_operator(Operator,LExpression,RExpression) |
with_operator(Operator,LExpression,RExpression) |
in_operator(Operator,LExpression,RExpression) |
notin_operator(Operator,LExpression,RExpression) |
less_operator(Operator,LExpression,RExpression) |
subset_operator(Operator,LExpression,RExpression)).

/* In the case of MCL strings, lists, and sets, this rule generates
Ada function calls to Ada functions which are functionally
equivalent to the plus operator. In the case of MCL reals and
integers, this rule generates Ada expressions using the MCL plus
operator. */
plus_operator(Operator,LExpression,RExpression) -->
{is_plus_operator(Operator)},
(((is_string_expression(LExpression),
  is_string_expression(RExpression),
  print_literal("Concat("),
  expression(LExpression),
  {print_literal(",")},
  expression(RExpression),
  {print_literal(")"})) |
({is_list_expression(LExpression),
  is_list_expression(RExpression),
  print_literal("Concat("),
  expression(LExpression),
  {print_literal(",")},
  expression(RExpression),
  {print_literal(")"})) |
({is_set_expression(LExpression),
  is_set_expression(RExpression),
  print_literal("Union("),
  expression(LExpression),
  {print_literal(",")},
  expression(RExpression),
  {print_literal(")"})) |
({print_literal("("),
  expression(LExpression),
  {print_operator(Operator)},
  expression(RExpression),
  {print_literal(")"})))).

/* Generates an Ada expression using the mod operator. */
mod_operator(Operator,LExpression,RExpression) -->
{is_mod_operator(Operator),
  print_literal("("),
  expression(LExpression),
  {print_operator(Operator)},
  expression(RExpression),
  {print_literal(")"}).

```

```

/* Generates an Ada function call to an Ada function which has the
   same functionality as the equal operator. */
equal_operator(Operator, LExpression, RExpression) -->
    {is_equal_operator(Operator),
      print_literal("Is_Equal(")},
    expression(LExpression),
    {print_literal(",")},
    expression(RExpression),
    {print_literal(")")}

/* Generates an Ada function call to an Ada function which has the
   same functionality as the not equal operator. */
not_equal_operator(Operator, LExpression, RExpression) -->
    {is_not_equal_operator(Operator),
      print_literal("Not_Equal(")},
    expression(LExpression),
    {print_literal(",")},
    expression(RExpression),
    {print_literal(")")}

/* Generates an Ada expression using the and operator. */
and_operator(Operator, LExpression, RExpression) -->
    {is_and_operator(Operator),
      print_literal("(")},
    expression(LExpression),
    {print_operator(Operator)},
    expression(RExpression),
    {print_literal(")")}

/* Generates Ada identifiers from MachineChart components. */
identifier(Node) -->
    {get_node_string(Node, STRING),
      print_literal(STRING)}.

```

Appendix E

A Sampling of the Ada Library Used to Support MCL Expressions and Data Types

```
--                                Integer Library

-- Contains procedures and functions which support MCL operations on
-- integer types.

-- IMPORTS
--   Real_Type : The default MCL or user defined real type.
--   Integer_Type: The default MCL or user defined integer type.
--   Boolean_Type: The default MCL or user defined boolean type.

GENERIC

    TYPE Integer_Type IS RANGE <>;
    TYPE Real_Type IS DIGITS <>;
    TYPE Boolean_Type IS (<>);

PACKAGE Integer_Library IS

    FUNCTION Copy(Old_Int : IN Integer_Type) RETURN Integer_Type;
    -- Supports deep assignment of integer list elements.

    FUNCTION Max(Left : IN Integer_Type; Right : IN Integer_Type)
        RETURN Integer_Type;
    -- MCL Code Left max Right
    -- Returns the maximum of Left and Right.

    FUNCTION Min(Left : IN Integer_Type; Right : IN Integer_Type)
        RETURN Integer_Type;
    -- MCL Code Left min Right
    -- Returns the minimum of Left and Right.
```

```

FUNCTION "**"(Left : IN Integer_Type; Right : IN Integer_Type)
    RETURN Integer_Type;
-- MCL Code Left ** Right
-- Returns Left raised to the exponent Right.

FUNCTION Is_Equal(Left : IN Integer_Type; Right : IN Integer_Type)
    RETURN Boolean_Type;
-- MCL Code Left = Right
-- Returns True if Left is lexically equal to Right.

FUNCTION Not_Equal(Left : IN Integer_Type; Right : IN Integer_Type)
    RETURN Boolean_Type;
-- MCL Code Left /= Right
-- Returns True if Left is not equal to Right.

FUNCTION Greater_Than(Left : IN Integer_Type; Right : IN Integer_Type)
    RETURN Boolean_Type;
-- MCL Code Left > Right
-- Returns True if Left is greater than Right.

FUNCTION Less_Than(Left : IN Integer_Type; Right : IN Integer_Type)
    RETURN Boolean_Type;
-- MCL Code Left < Right
-- Returns True if Left is less than Right.

FUNCTION Greater_Than_Equal(Left : IN Integer_Type;
                           Right : IN Integer_Type)
    RETURN Boolean_Type;
-- MCL Code Left >= Right
-- Returns True if Left is greater than or equal to Right.

FUNCTION Less_Than_Equal(Left : Integer_Type; Right : IN Integer_Type)
    RETURN Boolean_Type;
-- MCL Code Left <= Right
-- Returns True if Left is less than or equal to Right.

FUNCTION ToReal(Value : IN Integer_Type) RETURN Real_Type;
-- MCL Code real(Value)
-- Returns the real valued equivalent of Value.

PROCEDURE Output(Value : IN Integer_Type);
-- MCL Code print Value
-- Outputs the integer value to standard I/O

END Integer_Library;

```

```

--                               Default List Library

-- Contains procedures and functions which support MCL operations on
-- lists. The lists can be composed of any MCL object (integer, reals,
-- boolean, strings, sets, or other lists). The lists are implemented
-- as dynamic singly linked lists. Memory is NOT managed. The code
-- for the list library has been adapted from the code in Booch's
-- "Software Components With Ada".

-- IMPORTS
--   Integer_Type: The default MCL or user defined integer type.
--   Boolean_Type: The default MCL or user defined boolean type.
--   Item: Any MCL or user defined type.
--   Equal_Element : A function which can test the equality of the
--                   items in the list.
--   Output_Element : A procedure which can output the items in
--                   the list.
--   Deep_Assign: A function that produces a copy of the item.

```

GENERIC

```

TYPE Integer_Type IS RANGE <>;
TYPE Boolean_Type IS (<>);
TYPE Item IS PRIVATE;
WITH FUNCTION Equal_Element(Left_Element : IN Item;
                           Right_Element : IN Item)
                           RETURN Boolean_Type,
WITH PROCEDURE Output_Element(Element : IN Item);
WITH FUNCTION Deep_Assign(Old_Element : IN Item) RETURN Item;

```

PACKAGE List_Single_Unbounded_Unmanaged IS

```

TYPE List IS PRIVATE;

FUNCTION Copy(From_The_List : List) RETURN List;
-- Needed to support deep assignment of lists.

FUNCTION Length_Of(The_List : IN List) RETURN Integer_Type;
-- MCL Code #The_List
-- Returns the length of the list.

FUNCTION Get_Item(The_List : IN List;
                 At_The_Position : IN Integer_Type) RETURN Item;
-- MCL Code The_List(At_The_Position)
-- Returns the element at offset At_The_Position in the list.
-- An error results if At_The_Position < 1 or At_The_Position >
-- the length of the list.

FUNCTION SubPart_Of(The_List : IN List;
                   From_The_Position : IN Integer_Type;
                   To_The_Position : IN Integer_Type) RETURN List;

```

```
-- MCL Code The_List(From_The_Position .. To_The_Position)
-- Returns a sublist of elements ranging from offset
-- From_The_Position to offset To_The_Position. An error results if
-- From_The_Position > To_The_Position, From_The_Position < 1, or
-- To_The_Position > the length of the list.
```

```
PROCEDURE Set_Item(The_List : IN OUT List;
                   At_The_Position : IN Integer_Type;
                   With_The_Item : IN Item);
-- MCL Code The_List(At_The_Position) := With_The_Item
-- The location at offset At_The_Position in the list is
-- assigned the value With_The_Item. An error results if
-- At_The_Position < 1 or At_The_Position > the length of the
-- list.
```

```
PROCEDURE Modify(The_List : IN OUT List;
                 From_The_Position : IN Integer_Type;
                 To_The_Position : IN Integer_Type;
                 The_SubList : IN List);
-- MCL Code The_List(From_The_Position .. To_The_Position)
-- := The_SubList
-- The sublist consisting of the elements from From_The_Position
-- to To_The_Position in the list are replaced with the sublist
-- passed to the procedure. This may result in the expansion or
-- shrinkage of list. If From_The_Position = To_The_Position + 1
-- then the sublist is inserted into the original list at
-- To_The_Position. If From_The_Position = To_The_Position + 1
-- and To_The_Position = the length of the list then the sublist
-- is appended to the original list. An error results if
-- From_The_Position < 1, From_The_Position > To_The_Position
-- + 1, or To_The_Position > the length of the list.
```

```
FUNCTION Concat(First_List : IN List;
                Second_List : IN List) RETURN List;
-- MCL Code First_List + Second_List
-- Returns a list which contains the first list concatenated
-- to the second list.
```

```
FUNCTION Copy_and_Concat(The_List : IN List;
                        Number_Copies : IN Integer_Type) RETURN List;
-- MCL Code The_List * Number_Copies
-- Returns a list which contains Number_Copies of the original
-- list all concatenated together.
```

```
FUNCTION Append(The_List : IN List;
                The_Element : IN Item) RETURN List;
-- MCL Code The_List with The_Element
-- Returns a list with the element appended to the original list.
```

```
FUNCTION Is_Equal(First_List : IN List;
                  Second_List : IN List) RETURN Boolean_Type;
```

```

-- MCL Code  First_List = Second_List
--      Returns true if both lists are identical and false otherwise.

FUNCTION Not_Equal(First_List : IN List;
                   Second_List : IN List) RETURN Boolean_Type;
-- MCL Code  First_List /= Second_List
--      Returns true if the lists are not identical and false
--      otherwise.

FUNCTION In_List(The_Element : IN Item;
                 The_List : IN List) RETURN Boolean_Type;
-- MCL Code  The_Element in The_List
--      Returns true if the element is in the list and false
--      otherwise.

FUNCTION Not_In_List(The_Element : IN Item;
                     The_List : IN List) RETURN Boolean_Type;
-- MCL Code  The_Element not in The_List
--      Returns true if the element is not in the list and false
--      otherwise.

FUNCTION Construct RETURN List;
-- MCL Code  []
--      Constructs an empty list

FUNCTION Construct(The_Head : IN Item;
                  The_Tail : IN List) RETURN List;
-- MCL Code  [List_Literal]
--      Constructs a list literal item by item.

PROCEDURE Output(The_List : IN List);
-- MCL Code  print The_List
--      Outputs the list to standard I/O.

Position_Error : exception;

PRIVATE

TYPE Node IS RECORD
    The_Item : Item;
    Next : List;
END RECORD;

TYPE List IS ACCESS Node;

END List_Single_Unbounded_Unmanaged;
```


Appendix F

Excerpts of the C/JNX Program Generated from the Communication System Design

```
#include <stdio.h>
#include <jnx.h>
#include <jnx-dos.h>
#include <math.h>

#include "typedefs.h"

#include "ListLib.h"
#include "SetLib.h"
#include "StrLib.h"
#include "IntLib.h"
#include "ReallLib.h"
#include "BoolLib.h"

semaphore_id timing_semaphore;

/* ----- src_type Task Declaration ----- */

void src_type();
task_parameter_block src_parameters = {src_type, 1000, 3, 20, 1};
task_id src;
mailbox_id s_pduDn, s_pduUp;
typedef enum {s_pduDn_event, s_pduUp_event} src_events;
typedef struct {
    MCLString pdu;
} s_pduDn_parms;
MCLBoolean closed1;
```

⋮


```

void perform1(altbit1_EVENT_TYPE anEvent1,
              altbit1_ACTION_TYPE *formal_anAction1);

/* ----- comm1_type Task Declaration ----- */

void comm1_type();
task_parameter_block comm1_parameters = {comm1_type, 1000, 3, 20, 1};
task_id comm1;
mailbox_id sduDnG1, sduUpG1, pduUpG1, pduDnG1, timeout;
typedef enum {sduDnG1_event, sduUpG1_event, pduUpG1_event, pduDnG1_event,
              timeout_event} comm1_events;
typedef struct {
    MCLString sduDn;
} sduDnG1_parms;
typedef struct {
    MCLString sduUp;
} sduUpG1_parms;
typedef struct {
    MCLInteger typUp;
    MCLInteger seqUp;
    MCLString dataUp;
} pduUpG1_parms;
typedef struct {
    MCLInteger seqDn;
    MCLInteger typDn;
    MCLString dataDn;
} pduDnG1_parms;
MCLBoolean sduDnW;
MCLBoolean sduUpW;
MCLBoolean pduUpW;
MCLBoolean pduDnW;

void control(MCLBoolean *formal_OnOrOff);

/* ----- timer_type Task Declaration ----- */

void timer_type();
task_parameter_block timer_parameters = {timer_type, 1000, 3, 20, 1};
task_id timer;
mailbox_id start, stop;
flag_id recv_timer;
typedef enum {start_event, stop_event, recv_event} timer_events;

/*                                     /***** */
                                     :
/* ----- src_type Task Body ----- */

```

```

void src_type()
{
    MCLString pdu;
    MCLInteger i;
    MCLString data;
    MCLInteger size;
    mail_id mail;
    src_events event;

    i = 1;
    data = create_string("This is a test.");
    size = str_len(data);
    closed1 = False;
    for ( ; ; ) {
        if (closed1) {
            unmask_mailbox(s_pduUp);
        } else {
            mask_mailbox(s_pduUp);
        }
        event = await_any_mail(&mail);
        switch(event) {
            case s_pduUp_event:
                reply(s_pduUp);
                break;
            case s_pduDn_event:
                pdu = str_element(data, i);
                print_string(create_string("SRC sending '"));
                print_string(str_element(data, i));
                print_string(create_string("'"));
                print_newline();
                ((s_pduDn_parms *) mail)->pdu = pdu;
                reply(s_pduDn);
                i = (i % size);
                i = (i + 1);
                break;
        }
    }
}

```

⋮

```
/* ----- courier1_type Task Body ----- */
```

```

void courier1_type()
{
    MCLString pdu;
    MCLString sdu;

```

```

mail_id mail;
courier1_events event;
s_pduDn_parms s_pduDn_mail;

await_mail(Init1, &mail);
/* NULL */
reply(Init1);
for ( ; ; ) {
    send_mail(s_pduDn, &s_pduDn_mail);
    pdu = s_pduDn_mail.pdu;
    sdu = copy_string(pdu);
    sduDn1(sdu);
}
}

:

/* -----PACKAGE BODY : comm1----- */

void sduDn1(MCLString sdu)
{
    sduDnG1_parms sduDnG1_mail;

    sduDnG1_mail.sduDn = sdu;
    send_mail(sduDnG1, &sduDnG1_mail);
}

void sduUp1(MCLString *formal_sdu)
{
    MCLString sdu;
    sduUpG1_parms sduUpG1_mail;

    send_mail(sduUpG1, &sduUpG1_mail);
    sdu = sduUpG1_mail.sduUp;
    *formal_sdu = sdu;
}

void pduDn1(MCLInteger *formal_seq, MCLInteger *formal_typ,
            MCLString *formal_data)
{
    MCLInteger seq;
    MCLInteger typ;
    MCLString data;
    pduDnG1_parms pduDnG1_mail;

    send_mail(pduDnG1, &pduDnG1_mail);
    typ = pduDnG1_mail.typDn;
    seq = pduDnG1_mail.seqDn;

```

```

    data = pduDnG1_mail.dataDn;
    *formal_seq = seq;
    *formal_typ = typ;
    *formal_data = data;
}

void pduUp1(MCLInteger typ, MCLInteger seq, MCLString data)
{
    pduUpG1_parms pduUpG1_mail;

    pduUpG1_mail.typUp = typ;
    pduUpG1_mail.seqUp = seq;
    pduUpG1_mail.dataUp = data;
    send_mail(pduUpG1, &pduUpG1_mail);
}

/* -----ACM Body : altbit1----- */

typedef enum {start1, establ, ack_wait1} altbit1_STATE_TYPE;
altbit1_STATE_TYPE altbit1_cur_state_var = start1;

void altbit1_decode_event(altbit1_EVENT_TYPE cur_event_var,
                          altbit1_ACTION_TYPE *cur_action_var,
                          MCLInteger typ, MCLInteger rcvd_seq,
                          MCLInteger sent_seq)
{
    if (altbit1_cur_state_var == start1) {
        if ((cur_event_var == init1)) {
            altbit1_cur_state_var = establ;
            *cur_action_var = initialize1;
            return;
        }
    }
    else if (altbit1_cur_state_var == establ) {
        if ((cur_event_var == SEND_request1)) {
            altbit1_cur_state_var = ack_wait1;
            *cur_action_var = DATA_request1;
            return;
        }
        if ((cur_event_var == RECEIVE_request1)) {
            altbit1_cur_state_var = establ;
            *cur_action_var = RECEIVE_response1;
            return;
        }
        if ((cur_event_var == DATA_response1)) {
            if (((typ == 1) && (rcvd_seq == sent_seq))) {
                altbit1_cur_state_var = establ;
                *cur_action_var = data_received1;
                return;
            }
        }
    }
}

```

```

    }
}
} else if (altbit1_cur_state_var == ack_wait1) {
    if ((cur_event_var == RECEIVE_request1)) {
        altbit1_cur_state_var = ack_wait1;
        *cur_action_var = RECEIVE_response1;
        return;
    }
    if ((cur_event_var == DATA_response1)) {
        if (((typ == 1) && (rcvd_seq != sent_seq))) {
            altbit1_cur_state_var = ack_wait1;
            *cur_action_var = data_received1;
            return;
        }
    }
    if ((cur_event_var == DATA_response1)) {
        if (((typ == 1) && (rcvd_seq == sent_seq))) {
            altbit1_cur_state_var = establ;
            *cur_action_var = ack_received1;
            return;
        }
    }
    if ((cur_event_var == timeout1)) {
        altbit1_cur_state_var = ack_wait1;
        *cur_action_var = DATA_request1;
        return;
    }
}
cur_action_var = altbit1_IMPLICIT_ERROR;
}

/*****/

void perform1(altbit1_EVENT_TYPE  anEvent1,
              altbit1_ACTION_TYPE *formal_anAction1)
{
    altbit1_ACTION_TYPE  anAction1;

    anAction1 = *formal_anAction1;
    altbit1_decode_event(anEvent1, &anAction1, pduUp_type1,
                        pduUp_seq1, send_seq1);
    if ((anAction1 == initialize1)) {
        send_seq1 = 0;
        recv_seq1 = 0;
    } else if ((anAction1 == DATA_request1)) {
        pduDn_data1 = copy_string(sduDn1_data);
        pduDn_seq1 = send_seq1;
        pduDn_type1 = 0;
    }
}

```

```

        pduDnW = True;
        send_mail(start, NULL);
    } else if ((anAction1 == data_received1)) {
        sduUp1_data = copy_string(pduUp_data1);
        pduDn_type1 = 1;
        pduDn_seq1 = pduUp_seq1;
        pduDnW = True;
        sduUpW = True;
        recv_seq1 = ((recv_seq1 + 1) % 2);
    } else if ((anAction1 == ack_received1)) {
        send_mail(stop, NULL);
        send_seq1 = ((send_seq1 + 1) % 2);
        sduDnW = True;
        pduUpW = True;
    } else if ((anAction1 == RECEIVE_response1)) {
        pduUpW = True;
    } else {
        print_string(create_string("comm1: Bogus action to perform"));
        print_newline();
    }
    *formal_anAction1 = anAction1;
}

/* ----- comm1_type Task Body ----- */

void comm1_type()
{
    MCLString sduDn;
    MCLString sduUp;
    MCLInteger typUp;
    MCLInteger seqUp;
    MCLString dataUp;
    MCLInteger seqDn;
    MCLInteger typDn;
    MCLString dataDn;
    altbit1_EVENT_TYPE anEvent1;
    altbit1_ACTION_TYPE anAction1;
    mail_id mail;
    comm1_events event;

    sduUpW = False;
    pduDnW = False;
    sduDnW = True;
    pduUpW = True;
    anEvent1 = init1;
    perform1(anEvent1, &anAction1);
    for ( ; ; ) {
        if (sduUpW) {

```



```

        unmask_mailbox(sduUpG1);
    } else {
        mask_mailbox(sduUpG1);
    }
    if (pduUpW) {
        unmask_mailbox(pduUpG1);
    } else {
        mask_mailbox(pduUpG1);
    }
    if (sduDnW) {
        unmask_mailbox(sduDnG1);
    } else {
        mask_mailbox(sduDnG1);
    }
    if (pduDnW) {
        unmask_mailbox(pduDnG1);
    } else {
        mask_mailbox(pduDnG1);
    }
    event = await_any_mail(&mail);
    switch(event) {
        case timeout_event:
            reply(timeout);
            anEvent1 = timeout1;
            perform1(anEvent1, &anAction1);
            break;
        case sduUpG1_event:
            sduUpW = False;
            anEvent1 = RECEIVE_request1;
            perform1(anEvent1, &anAction1);
            sduUp = copy_string(sduUp1_data);
            ((sduUpG1_parms *) mail)->sduUp = sduUp;
            reply(sduUpG1);
            break;
        case pduUpG1_event:
            typUp = ((pduUpG1_parms *) mail)->typUp;
            seqUp = ((pduUpG1_parms *) mail)->seqUp;
            dataUp = ((pduUpG1_parms *) mail)->dataUp;
            pduUpW = False;
            pduUp_type1 = typUp;
            pduUp_seq1 = seqUp;
            pduUp_data1 = copy_string(dataUp);
            reply(pduUpG1);
            anEvent1 = DATA_response1;
            perform1(anEvent1, &anAction1);
            break;
        case sduDnG1_event:
            sduDn = ((sduDnG1_parms *) mail)->sduDn;

```

```

        sduDnW = False;
        sduDn1_data = copy_string(sduDn);
        reply(sduDnG1);
        anEvent1 = SEND_request1;
        perform1(anEvent1, &anAction1);
        break;
    case pduDnG1_event:
        pduDnW = False;
        dataDn = copy_string(pduDn_data1);
        typDn = pduDn_type1;
        seqDn = pduDn_seq1;
        ((pduDnG1_parms *) mail)->seqDn = seqDn;
        ((pduDnG1_parms *) mail)->typDn = typDn;
        ((pduDnG1_parms *) mail)->dataDn = dataDn;
        reply(pduDnG1);
        break;
    }
}
}

```

```

void control(MCLBoolean *formal_OnOrOff)
{

```

```

    MCLBoolean OnOrOff;

    OnOrOff = *formal_OnOrOff;
    if (OnOrOff) {
        OnOrOff = False;
        send_mail(timeout, NULL);
    }
    *formal_OnOrOff = OnOrOff;
}

```

```

/* ----- timer_type Task Body ----- */

```

```

void timer_type()

```

```

    MCLBoolean OnOrOff;
    mail_id mail;
    timer_events event;

```

```

    recv_timer = create_flag(timer, recv_event, false);
    OnOrOff = False;
    for ( ; ; ) {
        signal_flag_after_time(100.000000, recv_timer);
        event = await_any_event(&mail);
        switch(event) {
            case start_event:
                OnOrOff = True;

```

```

        reply(start);
        break;
    case stop_event:
        OnOrOff = False;
        reply(stop);
        break;
    case recv_event:
        control(&OnOrOff);
        break;
}
reset_timed_flag(recv_timer);
}
}

```

/* */

:

```

void Main();
task_parameter_block Main_parameters = {Main, 400, 0, 0, 1};

```

```

void main()
{
    char *keystroke;

    initialize_multitasking(true,50,10,10000,0,0,1);
    initialize_DOS_package();
    timing_semaphore = create_non_counting_semaphore(false);
    (void) create_task(&Main_parameters);
    src = create_task(&src_parameters);
    s_pduDn = create_mailbox(src, s_pduDn_event, false);
    s_pduUp = create_mailbox(src, s_pduUp_event, false);
    dst = create_task(&dst_parameters);
    d_pduDn = create_mailbox(dst, d_pduDn_event, false);
    d_pduUp = create_mailbox(dst, d_pduUp_event, false);
    courier2 = create_task(&courier2_parameters);
    Init2 = create_mailbox(courier2, Init2_event, false);
    courier1 = create_task(&courier1_parameters);
    Init1 = create_mailbox(courier1, Init1_event, false);
    courier3 = create_task(&courier3_parameters);
    Init3 = create_mailbox(courier3, Init3_event, false);
    courier4 = create_task(&courier4_parameters);
    Init4 = create_mailbox(courier4, Init4_event, false);
    comm1 = create_task(&comm1_parameters);
    sduDnG1 = create_mailbox(comm1, sduDnG1_event, false);
    sduUpG1 = create_mailbox(comm1, sduUpG1_event, false);
    pduUpG1 = create_mailbox(comm1, pduUpG1_event, false);
    pduDnG1 = create_mailbox(comm1, pduDnG1_event, false);
}

```

```

    timeout = create_mailbox(comm1, timeout_event, false);
    timer = create_task(&timer_parameters);
    start = create_mailbox(timer, start_event, false);
    stop = create_mailbox(timer, stop_event, false);
    comm2 = create_task(&comm2_parameters);
    sduDnG2 = create_mailbox(comm2, sduDnG2_event, false);
    sduUpG2 = create_mailbox(comm2, sduUpG2_event, false);
    pduUpG2 = create_mailbox(comm2, pduUpG2_event, false);
    pduDnG2 = create_mailbox(comm2, pduDnG2_event, false);
    courier5 = create_task(&courier5_parameters);
    Init5 = create_mailbox(courier5, Init5_event, false);
    courier6 = create_task(&courier6_parameters);
    Init6 = create_mailbox(courier6, Init6_event, false);
    schedule_tasks();
    for( ; ) {
        if ( Pgetch(keystroke)) break;
    }
    terminate_DOS_package();
    terminate_multitasking();
}

void Main()
{
    send_mail(Init1, NULL);
    send_mail(Init2, NULL);
    send_mail(Init3, NULL);
    send_mail(Init4, NULL);
    send_mail(Init5, NULL);
    send_mail(Init6, NULL);
}

```

3 of/de 3

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

Appendix G

C-Gen Rules Used in the Transformation of the Communication System Design

```
/* Root is the 'bootstrap' production rule as it is the rule that is
   invoked by Timebench. */
root(RootNode) -->
    generate_includes,
    {get_all_children(RootNode,ChildList),
     print_literal("semaphore_id timing_semaphore;"),
     print_newline, print_newline},
    generate_declarations(ChildList),
    generate_bodies(ChildList),
    generate_main(ChildList).

/* Generates the C include directives for the header files required by
   all C/JNX programs produced by C-Gen. */
generate_includes -->
    {print_literal("#include <stdio.h>"), print_newline,
     print_literal("#include <jnx.h>"), print_newline,
     print_literal("#include <jnx-dos.h>"), print_newline,
     print_literal("#include <math.h>"), print_newline, print_newline,
     print_literal("#include "), print_quote, print_literal("typedefs.h"),
     print_quote, print_newline, print_newline,
     print_literal("#include "), print_quote, print_literal("ListLib.h"),
     print_quote, print_newline,
     print_literal("#include "), print_quote, print_literal("SetLib.h"),
     print_quote, print_newline,
     print_literal("#include "), print_quote, print_literal("StrLib.h"),
     print_quote, print_newline,
     print_literal("#include "), print_quote, print_literal("IntLib.h"),
     print_quote, print_newline,
     print_literal("#include "), print_quote, print_literal("RealLib.h"),
     print_quote, print_newline,
```

```

    print_literal("#include "), print_quote, print_literal("Boollib.h"),
    print_quote, print_newline, print_newline}.

/* The next three rules control the order in which the MachineChart
   structural components are processed so that C's order of
   compilation rules are satisfied. They also flatten the MachineChart
   structural components. */
generate_declarations(ITEMS) -->
    {get_all_ACMs(ITEMS,ACMNodes)},
    an_ACM_declaration_list(ACMNodes),
    {get_all_data_nodes(ITEMS,DataNodes)},
    object_declaration_list(DataNodes),
    {get_all_buttons(ITEMS,ButtonNodes)},
    subprogram_declaration_list(ButtonNodes),
    {get_all_actors(ITEMS,ActorNodes)},
    task_declaration_list(ActorNodes),
    {get_all_reactors(ITEMS,ReactorNodes)},
    task_declaration_list(ReactorNodes),
    {get_all_boxes(ITEMS,BoxNodes)},
    package_declaration_list(BoxNodes).

generate_internal_declarations(ITEMS) -->
    {get_all_ACMs(ITEMS,ACMNodes)},
    an_ACM_declaration_list(ACMNodes),
    {get_all_buttons(ITEMS,ButtonNodes)},
    subprogram_declaration_list(ButtonNodes),
    {get_all_actors(ITEMS,ActorNodes)},
    task_declaration_list(ActorNodes),
    {get_all_reactors(ITEMS,ReactorNodes)},
    task_declaration_list(ReactorNodes),
    {get_all_boxes(ITEMS,BoxNodes)},
    package_declaration_list(BoxNodes).

generate_bodies(ITEMS) -->
    {get_all_ACMs(ITEMS,ACMNodes)},
    an_ACM_body_list(ACMNodes),
    {get_all_buttons(ITEMS,ButtonNodes)},
    subprogram_body_list(ButtonNodes),
    {get_all_actors(ITEMS,ActorNodes)},
    task_body_list(ActorNodes),
    {get_all_reactors(ITEMS,ReactorNodes)},
    special_task_body_list(ReactorNodes),
    {get_all_boxes(ITEMS,BoxNodes)},
    package_body_list(BoxNodes).

/* Generates the main line code required by JNX to start up the
   program. */
generate_main(ITEMS) -->
    {get_engine_node(ITEMS,EngineNode),
    print_literal("void Main();"), print_newline,

```

```

print_literal(
    "task_parameter_block Main_parameters = {Main, 400, 0, 0, 1};",
    print_newline, print_newline,
    print_literal("void main()"), print_newline,
    print_literal("{"), print_newline,
    indent(3),
    print_literal("char *keystroke;"), print_newline, print_newline,
    print_literal("initialize_multitasking(true,50,10,10000,0,0,1);"),
    print_newline,
    print_literal("initialize_DOS_package();"), print_newline,
    print_literal(
        "timing_semaphore = create_non_counting_semaphore(false);"),
    print_newline,
    print_literal("(void) create_task(&Main_parameters);"),
    print_newline},
create_tasks_and_init_packages(ITEMS),
{print_literal("schedule_tasks();"), print_newline,
    print_literal("for( ; ;) {"), print_newline,
    indent(3),
    print_literal("if ( Pgetch(keystroke)) break;"), print_newline,
    indent(-3),
    print_literal("}"), print_newline,
    print_literal("terminate_DOS_package();"), print_newline,
    print_literal("terminate_multitasking();"), print_newline,
    indent(-3),
    print_literal("}"), print_newline, print_newline,
    print_literal("void Main()"), print_newline,
    print_literal("{"), print_newline,
    indent(3)},
local_declarations(EngineNode),
{get_all_outgoing_arcs(EngineNode,Arcs)},
task_call_declarations(Arcs),
fetch_MCL_code(EngineNode),
{indent(-3),
    print_literal("}"), print_newline, print_newline}.
generate_main(ITEMS) -->
{print_literal("void main()"), print_newline,
    print_literal("{"), print_newline,
    indent(3),
    print_literal("char *keystroke;"), print_newline, print_newline,
    print_literal("initialize_multitasking(true,50,10,10000,0,0,1);"),
    print_newline,
    print_literal("initialize_DOS_package();"), print_newline,
    print_literal(
        "timing_semaphore = create_non_counting_semaphore(false);"),
    print_newline},
create_tasks_and_init_packages(ITEMS),
{print_literal("schedule_tasks();"), print_newline,
    print_literal("for( ; ;) {"), print_newline,
    indent(3),

```



```

    print_literal("if ( Pgetch(keystroke)) break;"), print_newline,
    indent(-3),
    print_literal("{}"), print_newline,
    print_literal("terminate_DOS_package();"), print_newline,
    print_literal("terminate_multitasking();"), print_newline,
    indent(-3),
    print_literal("{}"), print_newline, print_newline}.

/* The next two rules go through all the MachineChart structural
   components so that tasks associated with actors and reactors can be
   created, mailboxes associated with the gate buttons in actors and
   reactors can be created, and functions associated with box engines
   can be called. */
create_tasks_and_init_packages(ITEMS) -->
    {is_empty(ITEMS)} |
    ({extract_element(ITEMS,ITEM,Tail)},
     task_or_package(ITEM),
     create_tasks_and_init_packages(Tail)).

task_or_package(ITEM) -->
    ({is_robot(ITEM),
     get_all_children(ITEM,ChildList)},
     create_task(ITEM),
     create_tasks_and_init_packages(ChildList)) |
    ({is_reactor(ITEM),
     get_all_children(ITEM,ChildList)},
     create_task(ITEM),
     create_tasks_and_init_packages(ChildList)) |
    ({is_box(ITEM),
     get_all_children(ITEM,ChildList)},
     ({get_engine_node(ChildList,EngineNode)},
      init_package(ITEM),
      create_tasks_and_init_packages(ChildList)) |
     create_tasks_and_init_packages(ChildList))) |
    ({is_engine(ITEM)}) |
    ({is_data_store(ITEM)}) |
    ({is_ACM(ITEM)}) |
    ({is_gate(ITEM)}) |
    ({get_all_children(ITEM,ChildList)},
     create_tasks_and_init_packages(ChildList)).

/* Generates C code which creates the JNX task associated with the
   actor passed to the rule. */
create_task(RobotNode) -->
    identifier(RobotNode), {print_literal(" = create_ta\t(&")},
    identifier(RobotNode), {print_literal("_parameters);"),
    print_newline,
    get_external_children(RobotNode,Externals),
    get_all_gates(Externals,GateNodes)},
    create_mailboxes(RobotNode,GateNodes).

```

```

/* Generates C code which creates JNX mailboxes associated with the
   list of gate buttons contained in "GateNodes". */
create_mailboxes(RobotNode, GateNodes) -->
  {is_empty(GateNodes)} |
  ({extract_element(GateNodes, GateNode, GTail)},
   identifier(GateNode), {print_literal(" = create_mailbox(")},
   identifier(RobotNode), {print_literal(", ")},
   identifier(GateNode), {print_literal("_event, false);"),
   print_newline},
   create_mailboxes(RobotNode, GTail)).

/* Generates a C function call which invokes a special C function
   generated from a box engine. */
init_package(BoxNode) -->
  {print_literal("initialize_")}, identifier(BoxNode),
  {print_literal("();"), print_newline}.

/* The next two rules generate special C/JNX code from the list of
   ACM's contained in "ACMNodes". */
an_ACM_declaration_list(ACMNodes) -->
  {is_empty(ACMNodes)} |
  ({extract_element(ACMNodes, ACMNode, ACMTail)},
   an_ACM_specification(ACMNode),
   an_ACM_declaration_list(ACMTail)).

an_ACM_specification(ACMNode) -->
  {print_literal("/* -----ACM Specification : ")},
  identifier(ACMNode),
  {print_literal("----- */"),
   print_newline, print_newline},
  event_type_declaration(ACMNode),
  action_type_declaration(ACMNode),
  {print_literal("void ")},
  identifier(ACMNode), {print_literal("_decode_event(")},
  identifier(ACMNode),
  {print_literal("_EVENT_TYPE cur_event_var, ")},
  identifier(ACMNode),
  {print_literal("_ACTION_TYPE *cur_action_var"),
   get_formal_parameters(ACMNode, FormalParms)},
  ({is_empty(FormalParms)} |
   ({print_literal(", "),
    formal_parameter_list(FormalParms)}),
  {print_literal(");"), print_newline,
   print_literal("          /*****/"),
   print_newline, print_newline}.

/* Generates a C enumerated type declaration from the event list
   associated with an ACM. */
event_type_declaration(ACMNode) -->

```

```

    {print_literal("typedef enum {",
      get_all_events(ACMNode,EventList)},
      enum_list(EventList),
      {print_literal("} ")}},
      identifier(ACMNode),
      {print_literal("_EVENT_TYPE;"), print_newline}.

/* Generates a C enumerated type declaration from the action list
   associated with an ACM. */
action_type_declaration(ACMNode) -->
  {print_literal("typedef enum {"), identifier(ACMNode),
  {print_literal("_IMPLICIT_ERROR;"),
  get_all_actions(ACMNode,ActionList)},
  enum_list(ActionList),
  {print_literal("} ")}},
  identifier(ACMNode),
  {print_literal("_ACTION_TYPE;"), print_newline}.

/* Generates the enumeration contents for the enumerated type
   declarations described above. */
enum_list(EnumList) -->
  {extract_element(EnumList,ID,EnumTail)},
  identifier(ID),
  ({is_empty(EnumTail)} |
  ({print_literal(", ")}},
  enum_list(EnumTail))).

/* The next two rules generate C variable declarations from the list
   of MCL variable declarations contained in "DataNodes". */
object_declaration_list(DataNodes) -->
  {is_empty(DataNodes)} |
  ({extract_element(DataNodes,DataNode,DTail)},
  object_declaration(DataNode),
  object_declaration_list(DataNodes)).

object_declaration(DataNode) -->
  simple_type_declaration(DataNode) |
  compound_type_declaration(DataNode).

/* Generates C variable declarations from MCL variable declarations of
   type integer, real and boolean as well as ACM variable types. */
simple_type_declaration(DataNode) -->
  {get_type_node(DataNode,TypeNode)},
  simple_subtype_indication(TypeNode),
  identifier(DataNode),
  ({(get_initial_expression(DataNode,InitExpression),
  print_literal(" = ")),
  constant_expression(InitExpression),
  {print_literal(";"), print_newline, print_newline}) |
  {print_literal(";"), print_newline, print_newline}).

```

```

/* Generates C variable declarations from MCL variable declarations of
   type list and set. Note, MCL string, list, and set types cannot be
   initialized in the C code because function calls are not allowed in
   C variable initializations. */
compound_type_declaration(DataNode) -->
{get_type_node(DataNode,TypeNode)},
compound_subtype_indication(TypeNode),
{print_literal(" ")},
identifier(DataNode),
({get_initial_expression(DataNode,InitExpression),
 print_error(
  "Cannot transform initialization of compound MCL data types.")} |
{print_literal(";"), print_newline, print_newline}).

/* The next three rules generate the subtype indication for the C
   variable declaration based on the MCL type contained in the MCL
   variable declaration. */
subtype_indication(TypeNode) -->
simple_subtype_indication(TypeNode) |
compound_subtype_indication(TypeNode).

simple_subtype_indication(TypeNode) -->
{is_integer(TypeNode), print_literal("MCLInteger ")} |
{is_real(TypeNode), print_literal("MCLReal ")} |
{is_boolean(TypeNode), print_literal("MCLBoolean ")} |
({is_selected_component(TypeNode), /* ACM variable types */
 get_lhs(TypeNode,LHS),
 get_rhs(TypeNode,RHS)},
 identifier(LHS), {print_literal("_")}, identifier(RHS),
 {print_literal(" ")}).

compound_subtype_indication(TypeNode) -->
{is_string(TypeNode), print_literal("MCLString ")} |
{is_list(TypeNode), print_literal("MCLList ")} |
{is_set(TypeNode), print_literal("MCLSet ")}).

/* The next two rules generate C variable declarations for every
   waiting place found with a gate button taken from the list of gate
   buttons in "GateNodes" */
gate_declaration_list(GateNodes) -->
{is_empty(GateNodes)} |
({extract_element(GateNodes,GateNode,GTail)},
 ((gate_declaration(GateNode),
  gate_declaration_list(GTail)) |
 gate_declaration_list(GTail))).

gate_declaration(GateNode) -->
{get_waiting_place(GateNode,WaitingPlace),
 print_literal("MCLBoolean ")},

```

```

        identifier(WaitingPlace),
        {print_literal(";"), print_newline}.

/* Because JNX tasks exchange information through mailboxes, rather
   than a formal parameter list as do the gate buttons, the formal
   parameters in the gate buttons must be converted to C variable
   declarations. The next two rules perform this function by
   processing the list of gate buttons in "GateNodes". */
gate_formal_parm_declaration_list(GateNodes) -->
{is_empty(GateNodes)} |
({extract_element(GateNodes, GateNode, Tail),
 get_formal_parameters(GateNode, FormalParms)},
 gate_parm_declaration_list(FormalParms),
 gate_formal_parm_declaration_list(Tail)).

gate_parm_declaration_list(FormalParms) -->
{is_empty(FormalParms)} |
({extract_element(FormalParms, FormalParm, Tail),
 get_type_node(FormalParm, TypeNode)},
 subtype_indication(TypeNode),
 {print_literal(" ")},
 identifier(FormalParm),
 {print_literal(";"), print_newline, print_newline},
 gate_parm_declaration_list(Tail)).

/* The next two rules generate C function forward declarations from
   the list of push buttons contained in "ButtonNodes". */
subprogram_declaration_list(ButtonNodes) -->
{is_empty(ButtonNodes)} |
({extract_element(ButtonNodes, ButtonNode, BTail),
 get_external_children(ButtonNode, Externals),
 get_internal_children(ButtonNode, Internals)},
 generate_declarations(Externals),
 generate_internal_declarations(Internals),
 {free_list(Externals),
 free_list(Internals)},
 subprogram_declaration(ButtonNode),
 subprogram_declaration_list(BTail)).

subprogram_declaration(ButtonNode) -->
subprogram_specification(ButtonNode),
{print_literal(";"), print_newline, print_newline}.

/* Generates a function declaration from the information associated
   with a push button. */
subprogram_specification(ButtonNode) -->
{print_literal("void ")},
identifier(ButtonNode),
{print_literal("("),
 get_formal_parameters(ButtonNode, FormalParameters)},

```

```

({is_empty(FormalParameters)}) |
  formal_parameter_list(FormalParameters)),
{print_literal(")")}).

/* The next three rules generate C formal parameters from formal
   parameters associated with a push button. */
formal_parameter_list(FormalParameters) -->
  {extract_element(FormalParameters, FormalParameter, Tail)},
  parameter_specification(FormalParameter),
  ({is_empty(Tail)}) |
  ({print_literal(", "),
   formal_parameter_list(Tail))).

parameter_specification(FormalParameter) -->
  {get_type_node(FormalParameter, TypeNode),
   get_mode_type(FormalParameter, Mode)},
  subtype_indication(TypeNode),
  mode(Mode),
  identifier(FormalParameter).

mode(Mode) -->
  {is_in_mode(Mode), print_literal(" ")} |
  {is_out_mode(Mode), print_literal(" *formal_")} |
  {is_inout_mode(Mode), print_literal(" *formal_")}.

/* The next two rules generate JNX task declarations from the list of
   actors or reactors contained in "RobotNodes". */
task_declaration_list(RobotNodes) -->
  {is_empty(RobotNodes)} |
  ({extract_element(RobotNodes, RobotNode, RTail),
   get_external_children(RobotNode, Externals),
   get_internal_children(RobotNode, Internals)},
   generate_declarations(Externals),
   generate_internal_declarations(Internals),
   {free_list(Externals),
    free_list(Internals)},
   task_declaration(RobotNode),
   task_declaration_list(RTail)).

task_declaration(RobotNode) -->
  {is_robot_pool(RobotNode),
   print_error("Robot pools cannot be transformed.")} |
  ({print_literal("/* ----- "), identifier(RobotNode),
   {print_literal("_type Task Declaration ----- */"),
    print_newline, print_newline,
    print_literal("void ")}, identifier(RobotNode),
   {print_literal("_type();"), print_newline,
    print_literal("task_parameter_block ")},
    identifier(RobotNode),
    {print_literal("_parameters = {"}}),

```

```

    identifier(RobotNode),
    {print_literal("_type, 1000, 3, 20, 1;"),
     print_newline,
     print_literal("task_id ")}},
    identifier(RobotNode),
    {print_literal(";"), print_newline,
     get_all_children(RobotNode, ChildList),
     get_all_gates(ChildList, GateNodes),
     get_all_gates(ChildList, GateNodes2),
     get_all_gates(ChildList, GateNodes3),
     get_all_gates(ChildList, GateNodes4),
     get_all_pull_arcs(ChildList, PullArcs),
     get_all_pull_arcs(ChildList, PullArcs2)},
    ({is_empty(GateNodes)} |
     ({print_literal("mailbox_id ")}},
     mailbox_declaration_list(GateNodes),
     {print_literal(";"), print_newline},
     flag_declaration_list(PullArcs),
     {print_literal("typedef enum {"}}},
     mailbox_event_declaration_list(GateNodes2),
     flag_event_declaration_list(PullArcs2),
     {print_literal("} ")}}, identifier(RobotNode),
     {print_literal("_events;"), print_newline},
     mailbox_parameter_declarations(GateNodes3))),
    gate_declaration_list(GateNodes4)),
    {print_newline,
     free_list(ChildList)}}.

/* Generates a list of C variables which will eventually become
   pointers to the JNX mailboxes created from the list of gate buttons
   in "GateNodes". */
mailbox_declaration_list(GateNodes) -->
{extract_element(GateNodes, GateNode, GTail)},
identifier(GateNode),
({is_empty(GTail)} |
 ({print_literal(", ")}},
  mailbox_declaration_list(GTail))).

/* Generates a list of C variable declarations which will eventually
   become pointers to special timing flags used to generate time outs
   when waiting a JNX task is waiting for mail. These variables are
   only declared when there is a time out attached to a pull arc. */
flag_declaration_list(PullArcs) -->
{is_empty(PullArcs)} |
({extract_element(PullArcs, PullArc, Tail)},
 ({is_timed(PullArc),
  print_literal("flag_id ")}}, identifier(PullArc),
  {print_literal("_timer;"), print_newline},
  flag_declaration_list(Tail)) |
flag_declaration_list(Tail))).

```

```

/* Generates an enumerated constant for each gate button in
   "GateNodes". The constant is used to determine which event must be
   processed by the JNX task when it is waiting for mail. */
mailbox_event_declaration_list(GateNodes) -->
  {extract_element(GateNodes, GateNode, GTail)},
  identifier(GateNode), {print_literal("_event")},
  ({is_empty(GTail)} |
   {print_literal(", ")}),
  mailbox_event_declaration_list(GTail))).

/* Generates an enumerated constant for each pull arc having a time
   out attached to it. The constant is used for the same purpose as
   the constants generated by the previous list. */
flag_event_declaration_list(PullArcs) -->
  {is_empty(PullArcs)} |
  ({extract_element(PullArcs, PullArc, Tail)},
   ({is_timed(PullArc),
    print_literal(", ")}), identifier(PullArc),
    {print_literal("_event")},
    flag_event_declaration_list(Tail)) |
  flag_event_declaration_list(Tail))).

/* The next two rules generate a C structure type declaration which
   will be used to store mail for each JNX mailbox created from the
   gate buttons in "GateNodes". The format of the structure is based on
   the format of the formal parameter list associated with each gate
   button. */
mailbox_parameter_declarations(GateNodes) -->
  {is_empty(GateNodes)} |
  ({extract_element(GateNodes, GateNode, GTail),
   get_formal_parameters(GateNode, FormalParms)},
   ({is_empty(FormalParms)} |
    ({print_literal("typedef struct {"), print_newline,
     indent(12)},
     mailbox_parameters(FormalParms),
     {indent(-4),
      print_literal("} ")}), identifier(GateNode),
      {print_literal("_parms;"), print_newline,
       indent(-8)}}),
   mailbox_parameter_declarations(GTail))).

mailbox_parameters(FormalParms) -->
  {extract_element(FormalParms, FormalParm, Tail),
   get_type_node(FormalParm, TypeNode)},
  subtype_indication(TypeNode),
  {print_literal(" ")}),
  identifier(FormalParm),
  {print_literal(";"), print_newline},
  ({is_empty(Tail)} |

```



```

    mailbox_paramoters(Tail)).

/* The next two rules flatten the components contained inside the list
   of boxes in "BoxNodes". The modularity provided by the box is lost
   upon transformation. */
package_declaration_list(BoxNodes) -->
    {is_empty(BoxNodes)} |
    ({extract_element(BoxNodes,BoxNode,BTail)},
     package_specification(BoxNode),
     package_declaration_list(BTail)).

package_specification(BoxNode) -->
    {print_literal("/* -----Package Specification : ")},
    identifier(BoxNode),
    {print_literal("----- */")},
    print_newline,print_newline,
    get_external_children(BoxNode,Externals)},
    generate_declarations(Externals),
    {free_list(Externals),
     print_literal("                          /*****"/)},
    print_newline, print_newline,
    print_literal("/* -----Package Body Declarations : ")},
    identifier(BoxNode),
    {print_literal("----- */")},
    print_newline, print_newline,
    get_internal_children(BoxNode,Internals)},
    generate_declarations(Internals),
    {free_list(Internals),
     print_literal("/*                          /*****/ "),
     print_newline, print_newline}

/* The next two rules generate special C/JNX code to mimic the SM's
   contained in the list of ACM's in "ACMNodes". */
an_ACM_body_list(ACMNodes) -->
    {is_empty(ACMNodes)} |
    ({extract_element(ACMNodes,ACMNode,ACMTail)},
     an_ACM_body(ACMNode),
     an_ACM_body_list(ACMTail)).

an_ACM_body(ACMNode) -->
    {print_literal("/* -----ACM Body : ")},
    identifier(ACMNode),
    {print_literal("----- */")},
    print_newline,print_newline,
    get_internal_children(ACMNode,Internals),
    get_all_data_nodes(Internals,DataNodes),
    get_all_states(Internals,StateNodes),
    get_start_state(StateNodes,StartState)},
    state_type_declaration(ACMNode,StateNodes),
    state_var_declaration(ACMNode,StartState),

```

```

object_declaration_list(DataNodes), {print_newline,
  print_literal("void ")}, identifier(ACMNode),
{print_literal("_decode_event(")},
  identifier(ACMNode), {print_literal("_EVENT_TYPE cur_event_var, ")},
  identifier(ACMNode), {print_literal("_ACTION_TYPE *cur_action_var")},
  get_formal_parameters(ACMNode, FormalParms1)},
  ({is_empty(FormalParms1)} |
    ({print_literal(", ")},
      formal_parameter_list(FormalParms1))),
{print_literal(")"), print_newline,
  print_literal("{"), print_newline,
  indent(3),
  get_formal_parameters(ACMNode, FormalParms2)},
  assign_incoming_values(FormalParms2),
  an_ACM_if_stmt(ACMNode),
  {print_literal("cur_action_var = ")}, identifier(ACMNode),
  {print_literal("_IMPLICIT_ERROR;"), print_newline,
    get_formal_parameters(ACMNode, FormalParms3)},
  assign_outgoing_values(FormalParms3),
  {indent(-3),
    print_literal("}"), print_newline, print_newline,
    free_list(Internals),
    print_literal("                                /*****/"),
    print_newline, print_newline}.

/* Generates a C enumerated type definition from the list of states
   contained in "StateNodes". */
state_type_declaration(ACMNode, StateNodes) -->
  {print_literal("typedef enum {"}},
  enum_list(StateNodes),
  {print_literal("} ")},
  identifier(ACMNode),
  {print_literal("_STATE_TYPE;"), print_newline}.

/* Generates a special C variable declaration used to store the
   current state of the SM in the ACM. The current state variable is
   initialized with the start state. */
state_var_declaration(ACMNode, StartState) -->
  identifier(ACMNode),
  {print_literal("_STATE_TYPE ")},
  identifier(ACMNode),
  {print_literal("_cur_state_var = ")},
  identifier(StartState),
  {print_literal(";"), print_newline}.

/* The next two rules generate a C function from the list of push
   buttons contained in "ButtonNodes". */
subprogram_body_list(ButtonNodes) -->
  {is_empty(ButtonNodes)} |
  ({extract_element(ButtonNodes, ButtonNode, BTail),

```

```

    get_all_children(ButtonNode,ChildList)),
    generate_bodies(ChildList),
    {free_list(ChildList)},
    subprogram_body(ButtonNode),
    subprogram_body_list(BTail)).

subprogram_body(ButtonNode) -->
    subprogram_specification(ButtonNode), {print_newline,
    print_literal("{"), print_newline,
    indent(3),
    get_internal_children(ButtonNode,Internals),
    get_all_data_nodes(Internals,DataNodes)},
    object_declaration_list(DataNodes),
    {get_formal_parameters(ButtonNode,FormalDecls)},
    button_formal_parm_list(FormalDecls),
    ({get_engine_node(Internals,EngineNode)},
    local_declarations(EngineNode),
    {get_all_outgoing_arcs(EngineNode,Arcs)},
    task_call_declarations(Arcs),
    {get_formal_parameters(ButtonNode,FormalParms1)},
    assign_incoming_values(FormalParms1),
    fetch_MCL_code(EngineNode),
    {get_formal_parameters(ButtonNode,FormalParms2)},
    assign_outgoing_values(FormalParms2)) |
    {print_literal("/* NULL */"), print_newline}),
    {indent(-3),
    print_literal(")"), print_newline, print_newline,
    free_list(Internals)}).

/* The next two rules generate C variable declarations for every JNX
   task call via a mailbox. Each variable is a structure type whose
   format is derived from the format of the formal parameter list of
   each gate button processed by the rule. The gate buttons are found
   by checking the destinations of all the arcs contained in "Arcs". */
task_call_declarations(Arcs) -->
    {is_empty(Arcs)} |
    ({extract_element(Arcs,Arc,Tail)},
    ((task_call_declaration(Arc),
    task_call_declarations(Tail)) |
    task_call_declarations(Tail))).

task_call_declaration(Arc) -->
    {get_destination(Arc,Node),
    is_gate(Node),
    get_formal_parameters(Node,FormalParms)},
    ({is_empty(FormalParms)} |
    (identifier(Node), {print_literal("_parms ")}),
    identifier(Node), {print_literal("_mail;"),
    print_newline, print_newline,
    free_list(FormalParms)})).

```

```

/* The next two rules generate C variable declarations for any of the
   formal parameters in the push button that are of mode out or mode
   in out, since the names of these formal parameters are changed
   during the transformation. */
button_formal_parm_list(FormalParms) -->
{is_empty(FormalParms)} |
({extract_element(FormalParms, FormalParm, Tail),
  get_mode_type(FormalParm, Mode)},
  button_formal_parm(FormalParm, Mode),
  button_formal_parm_list(Tail)).

button_formal_parm(FormalParm, Mode) -->
{is_in_mode(Mode)} |
({get_type_node(FormalParm, Type)},
  subtype_indication(Type), {print_literal(" ")},
  identifier(FormalParm), {print_literal(";")},
  print_newline, print_newline)).

/* The next two rules generate C assignment statements to assign the
   newly created C formal parameters to the C variables generated as
   place holders for the original MCL formal parameters. */
assign_incoming_values(FormalParms) -->
{is_empty(FormalParms)} |
({extract_element(FormalParms, FormalParm, Tail),
  get_mode_type(FormalParm, Mode)},
  assign_incoming_value(FormalParm, Mode),
  assign_incoming_values(Tail)).

assign_incoming_value(FormalParm, Mode) -->
{is_in_mode(Mode)} |
{is_out_mode(Mode)} |
(identifier(FormalParm), {print_literal(" = ")},
  {print_literal("*formal_")}, identifier(FormalParm),
  {print_literal(";")}, print_newline)).

/* The next two rules generate C assignment statements to assign the C
   variables generated as place holder for the original MCL formal
   parameters to the newly created C formal parameters. */
assign_outgoing_values(FormalParms) -->
{is_empty(FormalParms)} |
({extract_element(FormalParms, FormalParm, Tail),
  get_mode_type(FormalParm, Mode)},
  assign_outgoing_value(FormalParm, Mode),
  assign_outgoing_values(Tail)).

assign_outgoing_value(FormalParm, Mode) -->
{is_in_mode(Mode)} |
({print_literal("*formal_")}, identifier(FormalParm),
  {print_literal(" = ")}, identifier(FormalParm),

```

```

    {print_literal(";"), print_newline}}.

/* The next two rules generate C functions which are used as JNX task
bodies from the list of actors contained in "RobotNodes". */
task_body_list(RobotNodes) -->
    {is_empty(RobotNodes)} |
    ({extract_element(RobotNodes, RobotNode, RTail),
     get_all_children(RobotNode, ChildList)},
     generate_bodies(ChildList),
     {free_list(ChildList)},
     task_body(RobotNode),
     task_body_list(RTail)).

task_body(RobotNode) -->
    {print_literal("/* ----- "), identifier(RobotNode),
     {print_literal("_type Task Body ----- */"),
      print_newline, print_newline,
      print_literal("void ")}, identifier(RobotNode),
     {print_literal("_type()"), print_newline,
      print_literal("{"), print_newline,
      indent(3),
      get_all_children(RobotNode, ChildList),
      get_all_gates(ChildList, GateNodes1),
      get_all_gates(ChildList, GateNodes3),
      get_all_gates(ChildList, GateNodes4),
      get_all_pull_arcs(ChildList, PullArcs),
      get_all_data_nodes(ChildList, DataNodes)},
     local_gate_object_list(GateNodes1),
     gate_formal_parm_declaration_list(GateNodes4),
     object_declaration_list(DataNodes),
     ({is_empty(GateNodes3)} |
      {print_literal("mail_id mail;"), print_newline, print_newline},
      identifier(RobotNode),
      {print_literal("_events event;"), print_newline, print_newline}),
     ({get_engine_node(ChildList, EngineNode)},
      local_declarations(EngineNode),
      {get_all_outgoing_arcs(EngineNode, Arcs)},
      task_call_declarations(Arcs),
      create_flags(RobotNode, PullArcs),
      fetch_MCL_code(EngineNode)) |
      {print_literal("/* NULL */"), print_newline}},
    {indent(-3),
     print_literal(")"), print_newline, print_newline,
     free_list(ChildList)}.

/* Generates special C code to create all timing flags needed by the
task. This code is only generated if a time out is located on a
pull arc in the list of pull arcs contained in "PullArcs". */
create_flags(RobotNode, PullArcs) -->
    {is_empty(PullArcs)} |

```

```

({extract_element(PullArcs,PullArc,Tail)},
({{is_timed(PullArc)},
  identifier(PullArc), {print_literal("_timer = create_flag(")},
  identifier(RobotNode), {print_literal(", ")},
  identifier(PullArc), {print_literal("_event, false);"},
  print_newline},
  create_flags(RobotNode,Tail)) |
  create_flags(RobotNode,Tail))).

/* The next two rules process the contents of the list of boxes
   contained in "BoxNodes". */
package_body_list(BoxNodes) -->
  {is_empty(BoxNodes)} |
  ({extract_element(BoxNodes,BoxNode,BTail)},
   package_body(BoxNode),
   package_body_list(BTail)).

package_body(BoxNode) -->
  {print_literal("/* -----PACKAGE BODY : ")},
  identifier(BoxNode),
  {print_literal("----- */"),
   print_newline, print_newline},
  {get_all_children(BoxNode,ChildList)},
  generate_bodies(ChildList),
  (({get_engine_node(ChildList,EngineNode),
    print_literal("void initialize_")}, identifier(BoxNode),
    {print_literal("(")}, print_newline,
    print_literal("{"), print_newline,
    indent(3)}
    local_declarations(EngineNode),
    {get_all_outgoing_arcs(EngineNode,Arcs)},
    task_call_declarations(Arcs),
    fetch_MCL_code(EngineNode),
    {indent(-3),
     print_literal(")}, print_newline,
     print_literal("                                /*****"/)},
     print_newline, print_newline}) |
    {print_literal("                                /*****"/)},
     print_newline, print_newline)),
  {free_list(ChildList)}.

/* Generates C variable declarations which are local to an engine. */
local_declarations(EngineNode) -->
  {get_local_declarations(EngineNode,DataNodes)},
  object_declaration_list(DataNodes).

/* Processes the MCL code inside push button, robot, and box engines. */
fetch_MCL_code(EngineNode) -->
  {get_MCL_code(EngineNode,MCL)},
  (({is_empty(MCL)},

```

```

        print_literal("/* NULL */")) |
sequence_of_statements(MCL)).

/* The next two rules transform MCL statements to C/JNX statements.
   The rules maintain the order of the MCL statements. */
sequence_of_statements(MCL) -->
    {is_empty(MCL)} |
    ({extract_element(MCL,MCLstmt,Tail)},
     statement(MCLstmt),
     sequence_of_statements(Tail)).

statement(MCLstmt) -->
    null_stmt(MCLstmt) |
    assignment_stmt(MCLstmt) |
    exit_stmt(MCLstmt) |
    delay_stmt(MCLstmt) |
    push_stmt(MCLstmt) |
    return_stmt(MCLstmt) |
    if_stmt(MCLstmt) |
    loop_stmt(MCLstmt) |
    case_stmt(MCLstmt) |
    pull_stmt(MCLstmt) |
    reply_stmt(MCLstmt) |
    print_stmt(MCLstmt).

/* A MCL null statement translates to nothing in C; therefore, it is
   skipped over. */
null_stmt(MCLstmt) -->
    {is_null_stmt(MCLstmt)}.

/* Generates a C assignment statement or C function call from a MCL
   assignment statement depending on the expressions contained in the
   assignment statement. MCL only supports deep assignment. */
assignment_stmt(MCLstmt) -->
    {is_assignment_stmt(MCLstmt),
     get_lhs(MCLstmt,LHS),
     get_rhs(MCLstmt,RHS)},
    ({(is_slice(LHS)), slice_assignment(LHS,RHS)) |
     ({(is_index_component(LHS)), index_assignment(LHS,RHS)) |
     ({(is_var_name(RHS)), copy_assignment(LHS,RHS)) |
     (simple_var(LHS), {print_literal(" = ")}), expression(RHS),
     {print_literal(";"), print_newline}))}.

/* Generates a C function call from a MCL assignment statement
   containing a list, set or string variable on the right hand side of
   the assignment operator. This rule supports the deep assignment of
   these three types. */
copy_assignment(LHS,RHS) -->
    ({is_list_expression(RHS)},
     identifier(LHS), {print_literal(" = copy_list(")}),

```

```

        identifier(RHS), {print_literal(");"), print_newline}) |
    ({is_set_expression(RHS)},
    identifier(LHS), {print_literal(" = copy_set(")},
    identifier(RHS), {print_literal(");"), print_newline}) |
    ({is_string_expression(RHS)},
    identifier(LHS), {print_literal(" = copy_string(")},
    identifier(RHS), {print_literal(");"), print_newline}).

/* Processes the MCL push statement. */
push_stmt(MCLstmt) -->
    {is_push_stmt(MCLstmt),
    get_arc(MCLstmt,Arc)},
    (subprogram_call(Arc) |
    async_entry_call(Arc) |
    timed_entry_call(Arc) |
    regular_entry_call(Arc) |
    an_ACM_call_stmt(Arc) |
    install_arc(Arc) |
    remove_arc(Arc)).

/* Generates a C function call from a push arc whose destination is a
push button. */
subprogram_call(Arc) -->
    {is_push_arc(Arc),
    get_destination(Arc, Destination),
    get_formal_parameters(Destination, FormalParms)},
    identifier(Destination), {print_literal("(")},
    ({is_empty(FormalParms)} |
    actual_parameter_list(Arc, FormalParms)),
    {print_literal(");"), print_newline}.

/* Generates the actual parameters for a C function call from the data
flows located along the push arc. */
actual_parameter_list(Arc, FormalParms) -->
    {extract_element(FormalParms, FormalParm, Tail),
    get_actual_parms(Arc, ActualParms)},
    find_position(FormalParm, ActualParms),
    ({is_empty(Tail)} |
    ({print_literal(", ")},
    actual_parameter_list(Arc, Tail))).

/* This rule ensures that the actual parameter corresponds to the
correct formal parameter in the C function call. */
find_position(FormalParm, ActualParms) -->
    {extract_element(ActualParms, ActualParm, Tail),
    get_formal_parm(ActualParm, FormalParm2)},
    ({(element_equal(FormalParm, FormalParm2)},
    actual_parameter_part(ActualParm, FormalParm),
    {free_list(Tail)}) |
    find_position(FormalParm, Tail)).

```



```

/* Generates an actual parameter. */
actual_parameter_part(ActualParm,FormalParm) -->
  {get_mode_type(FormalParm,Mode)},
  ({(is_in_mode(Mode)), identifier(ActualParm)) |
   ({print_literal("&"), identifier(ActualParm))}).

/* Generates a C function call to the JNX mailbox which corresponds to
   the gate button located at the destination of the arc. */
regular_entry_call(Arc) -->
  {is_rendezvous_arc(Arc),
   get_destination(Arc, Destination),
   get_formal_parameters(Destination, FormalParms),
   get_actual_parms(Arc, ActualParms1),
   get_actual_parms(Arc, ActualParms2)},
  set_up_parms(Destination, ActualParms1),
  {print_literal("send_mail(")},
  identifier(Destination), {print_literal(", ")},
  ({is_empty(FormalParms),
   print_literal("NULL");"}, print_newline) |
   ({print_literal("&"), identifier(Destination),
    {print_literal("_mail");"}, print_newline}}),
  retrieve_parms(Destination, ActualParms2).

/* These two rules generate C assignment statements which assign the
   outgoing data flows (the MCL actual parameters) to the components
   of the structure used to send and receive mail for the mailbox being
   processed in the previous rule. */
set_up_parms(Node, ActualParms) -->
  {is_empty(ActualParms)} |
  ({extract_element(ActualParms, ActualParm, Tail)},
   set_up_parm(Node, ActualParm),
   set_up_parms(Node, Tail)).

set_up_parm(Node, ActualParm) -->
  {get_formal_parm(ActualParm, FormalParm),
   get_mode_type(FormalParm, Mode)},
  ({is_out_mode(Mode)} |
   (identifier(Node), {print_literal("_mail.")},
    identifier(FormalParm), {print_literal(" = ")},
    identifier(ActualParm), {print_literal(";"), print_newline})).

/* These two rules generate C assignment statements which assign the
   components of the structure used to send and receive mail for the
   mailbox being processed above to the incoming data flows. */
retrieve_parms(Node, ActualParms) -->
  {is_empty(ActualParms)} |
  ({extract_element(ActualParms, ActualParm, Tail)},
   retrieve_parm(Node, ActualParm),
   retrieve_parms(Node, Tail)).

```

```

retrieve_parm(Node,ActualParm) -->
    {get_formal_parm(ActualParm,FormalParm),
     get_mode_type(FormalParm,Mode)},
    ({is_in_mode(Mode)} |
     (identifier(ActualParm), {print_literal(" = ")}),
     identifier(Node), {print_literal("_mail.")}, identifier(FormalParm),
     {print_literal(";"), print_newline})).

/* Generates a C function call which invokes the function created to
   mimic the SM in the ACM located at the destination of the arc being
   processed by this rule. */
an_ACM_call_stmt(Arc) -->
    {is_ACM_push_arc(Arc),
     get_actual_event_parm(Arc,Event),
     get_actual_action_parm(Arc,Action),
     get_destination(Arc,Destination),
     get_formal_parameters(Destination,FormalParms)},
    identifier(Destination), {print_literal("_decode_event(")},
    identifier(Event), {print_literal(", &")},
    identifier(Action),
    ({is_empty(FormalParms)} |
     ({print_literal(", ")}), actual_parameter_list(Arc,FormalParms))),
    {print_literal(";"), print_newline}.

/* The next three rules generate a C if statement from a MCL if
   statement. */
if_stmt(MCLstmt) -->
    {is_if_stmt(MCLstmt),
     get_then_block(MCLstmt,ThenStatements),
     print_literal("if ("),
     get_condition(MCLstmt,Condition)},
    expression(Condition),
    {print_literal(") {"}, print_newline,
     indent(3)},
    sequence_of_statements(ThenStatements),
    {indent(-3)},
    (({get_elseif_block(MCLstmt,ElsIfStmts)},
     else_if_stmts(ElsIfStmts),
     ({get_else_block(MCLstmt,ElseStmts),
      print_literal("{} else {"}, print_newline,
      indent(3)},
      sequence_of_statements(ElseStmts),
      {indent(-3),
       print_literal(")"), print_newline})} |
     {print_literal(")"), print_newline}))) |
    (({get_else_block(MCLstmt,ElseStmts),
     print_literal("{} else {"}, print_newline,
     indent(3)},
     sequence_of_statements(ElseStmts),

```

```

        {indent(-3),
         print_literal("{}"), print_newline}} |
        {print_literal("{}"), print_newline})).

else_if_stmts(ElseIfNodes) -->
    {is_empty(ElseIfNodes)} |
    ({extract_element(ElseIfNodes,ElseIfNode,Tail)},
     else_if_stmt(ElseIfNode),
     else_if_stmts(Tail)).

else_if_stmt(ElseIfNode) -->
    {print_literal("{} else if (",
     get_condition(ElseIfNode,Condition)},
     expression(Condition),
     {print_literal("{} {", print_newline,
      indent(3),
      get_then_block(ElseIfNode,ThenStatements)},
     sequence_of_statements(ThenStatements),
     {indent(-3)}}.

/* The next two rules generates a special if statement to mimic the
   operations performed by the SM inside the ACM. One if or else if
   alternative is generated for every state found in "StateNodes". */
an_ACM_if_stmt(ACMNode) -->
    {get_internal_children(ACMNode,Internals),
     get_all_states(Internals,StateNodes),
     extract_element(StateNodes,StateNode,Tail),
     print_literal("if (", identifier(ACMNode),
     {print_literal("_cur_state_var == ")},
     identifier(StateNode), {print_literal("{} {", print_newline,
     indent(3)},
     ({is_end_state(StateNode),
      print_literal("/* no further actions from this state */")}) |
     ({get_all_outgoing_arcs(StateNode,SrcArcs)},
      the_ACM_events_and_actions(SrcArcs,ACMNode))),
     an_ACM_else_if_stmts(Tail,ACMNode),
     {indent(-3),
      print_literal("{}"), print_newline}.

an_ACM_else_if_stmts(StateNodes,ACMNode) -->
    {is_empty(StateNodes)} |
    ({extract_element(StateNodes,StateNode,Tail),
     indent(-3),
     print_literal("{} else if (", identifier(ACMNode),
     {print_literal("_cur_state_var == ")},
     identifier(StateNode), {print_literal("{} {", print_newline,
     indent(3)},
     ({is_end_state(StateNode),
      print_literal("/* No further actions from this state. */")}) |
     ({get_all_outgoing_arcs(StateNode,SrcArcs)},

```

```

    the_ACM_events_and_actions(SrcArcs,ACMNode))),
    an_ACM_else_if_stmts(Tail,ACMNode)).

/* Generates C if statements from the information associated with
every transition arc contained in "SrcArcs". Each arc in "SrcArcs" has
the same state as its source. */
the_ACM_events_and_actions(SrcArcs,ACMNode) -->
{is_empty(SrcArcs)} |
({extract_element(SrcArcs,Arc,Tail)),
 (an_ACM_if_event(Arc,ACMNode) |
  an_ACM_if_condition(Arc,ACMNode) |
  an_ACM_action(Arc,ACMNode) |
  (an_ACM_update_state(Arc,ACMNode),
   an_ACM_return_action(Arc))),
 the_ACM_events_and_actions(Tail,ACMNode)).

/* Generates a C if statement from the event list associated with the
transition arc currently being processed. */
an_ACM_if_event(SrcArc,ACMNode) -->
{get_arc_event_list(SrcArc,EventList),
 print_literal("if ("),
 process_event(EventList),
 {print_literal(") {"), print_newline,
  indent(3)},
 (an_ACM_if_condition(SrcArc,ACMNode) |
  an_ACM_action(SrcArc,ACMNode) |
  (an_ACM_update_state(SrcArc,ACMNode),
   an_ACM_return_action(SrcArc))),
 {indent(-3),
  print_literal(")"), print_newline}.

/* Generates the conditional part of an if statement based on the
information contained in the event list mentioned above. */
process_event(EventList) -->
{is_single_event(EventList),
 print_literal("(cur_event_var == "), identifier(EventList),
 {print_literal(")")} |
 ({get_lhs(EventList,Event),
  get_rhs(EventList,NewEventList),
  print_literal('(cur_event_var ==)'), identifier(Event),
  {print_literal(") || ")},
 process_event(NewEventList)).

/* Generates an Ada if statement from the condition associated with
the transition arc. */
an_ACM_if_condition(SrcArc,ACMNode) -->
{get_arc_condition(SrcArc,Condition),
 print_literal("if ("), expression(Condition),
 {print_literal(") {"), print_newline,
  indent(3)},

```

```

(an_ACM_action(SrcArc,ACMNode) |
  (an_ACM_update_state(SrcArc,ACMNode),
   an_ACM_return_action(SrcArc))),
{indent(-3),
 print_literal("{}"), print_newline}.

/* Generates a sequence of C statements based on the MCL action code
associated with the transition arc. */
an_ACM_action(SrcArc,ACMNode) -->
  {get_arc_action(SrcArc,ActionCode)},
  sequence_of_statements(ActionCode),
  an_ACM_update_state(SrcArc,ACMNode),
  an_ACM_return_action(SrcArc).

/* Generates a C assignment statement which updates the current state
of the function simulating the SM to the state located at the
destination of the transition arc. */
an_ACM_update_state(SrcArc,ACMNode) -->
  {get_destination(SrcArc,DestinationState)},
  identifier(ACMNode), {print_literal("_cur_state_var = ")},
  identifier(DestinationState), {print_literal(";"), print_newline}.

/* Generates a C assignment statement which sets the action variable
to the action associated with the transition arc. */
an_ACM_return_action(SrcArc) -->
  {get_arc_return_action(SrcArc,ReturnAction),
   print_literal("*cur_action_var = ")}, identifier(ReturnAction),
  {print_literal(";"), print_newline},
  print_literal("return;"), print_newline}.

/* Generates a C infinite loop, while loop, or for loop statement from
a MCL simple loop, while loop, or for loop statement. */
loop_stmt(MCLstmt) -->
  ({is_simple_loop_stmt(MCLstmt),
   print_literal("for ( ; ; ) {"}, print_newline,
   indent(3),
   get_code_block(MCLstmt,MCLcode)},
   sequence_of_statements(MCLcode),
   {indent(-3),
    print_literal("}"), print_newline}) |
  ({is_while_loop_stmt(MCLstmt),
   get_iteration_scheme(MCLstmt,Condition),
   print_literal("while ("}, expression(Condition),
   {print_literal(") {"}, print_newline,
   indent(3),
   get_code_block(MCLstmt,MCLcode)},
   sequence_of_statements(MCLcode),
   {indent(-3),
    print_literal("}"), print_newline}) |
  ({is_for_loop_stmt(MCLstmt)},

```

```

    process_for_loop(MCLstmt)).

/* The next five rules generate a C if statement from a MCL case
statement. */
case_stmt(MCLstmt) -->
{is_case_stmt(MCLstmt),
 get_switch(MCLstmt,Switch),
 get_alternatives(MCLstmt,Alternatives),
 extract_element(Alternatives,Alternative,Tail),
 get_choices(Alternative,Choices),
 extract_element(Choices,Choice,CTail),
 get_code_block(Alternative,MCLcode),
 print_literal("if ("),
 choice(Switch,Choice),
 choice_list(Switch,CTail),
 {print_literal(") {"), print_newline,
  indent(3)},
 sequence_of_statements(MCLcode),
 case_alternative_list(Switch,Tail),
 {indent(-3),
 print_literal(")"), print_newline}.

case_alternative_list(Switch,Alternatives) -->
{is_empty(Alternatives)} |
({extract_element(Alternatives,Alternative,Tail)},
 case_alternative(Switch,Alternative),
 case_alternative_list(Switch,Tail)).

case_alternative(Switch,Alternative) -->
{get_choices(Alternative,Choices),
 get_code_block(Alternative,MCLcode),
 extract_element(Choices,Choice,Tail)},
(({is_others(Choice),
  indent(-3),
  print_literal("} else {"), print_newline,
  indent(3)},
 sequence_of_statements(MCLcode)) |
({indent(-3),
  print_literal("} else if {"),
  choice(Switch,Choice),
  choice_list(Switch,Tail),
 {print_literal(") {"), print_newline,
  indent(3)},
 sequence_of_statements(MCLcode))).

choice_list(Switch,Choices) -->
{is_empty(Choices)} |
({extract_element(Choices,Choice,Tail),
  print_literal(" || "),
  choice(Switch,Choice),

```

```

choice_list(Switch,Tail)).

choice(Switch,Choice) -->
  ({is_discrete_range(Choice),
   get_lhs(Choice,Low),
   get_rhs(Choice,High),
   print_literal("("), expression(Switch), {print_literal(" >= ")},
   expression(Low), {print_literal(" && ")},
   expression(Switch), {print_literal(" <= ")}, expression(High),
   {print_literal(")"))} |
  ({print_literal("("), expression(Switch), {print_literal(" == ")},
   expression(Choice), {print_literal(")"))}).

/* Processes the MCL pull statement. */
pull_stmt(MCLstmt) -->
  {is_pull_stmt(MCLstmt),
   get_arc(MCLstmt,PullArc)},
  (select_stmt(PullArc) |
   simple_accept(PullArc)).

/* Generates a sequence of C statements which enables a JNX task to
   receive mail. The contents of the C statements depend upon the
   information associated with the pull arc being processed. */
select_stmt(Arc) -->
  ({is_timed(Arc)},
   ({is_choice(Arc),
    get_all_destinations(Arc,Gates1),
    get_all_destinations(Arc,Gates2),
    get_pull_timer(Arc,Timer)},
   waiting_place_evaluations(Gates1),
   start_delay_alternative(Arc,Timer),
   {print_literal("event = await_any_event(&mail);"), print_newline,
    print_literal("switch(event) {"), print_newline,
    indent(3)},
   event_alternatives(Gates2),
   delay_alternative(Arc,Timer),
   {indent(-3),
    print_literal(")"), print_newline,
    print_literal("reset_timed_flag("),
    identifier(Arc), {print_literal("_timer);"), print_newline}} |
   ({get_destination(Arc,Gate),
    get_pull_timer(Arc,Timer)},
   start_delay_alternative(Arc,Timer),
   {print_literal("event = await_any_event(&mail);"), print_newline,
    print_literal("switch(event) {"), print_newline,
    indent(3)},
   event_alternative(Gate),
   delay_alternative(Arc,Timer),
   {indent(-3),
    print_literal(")"), print_newline,

```

```

        print_literal("reset_timed_flag("),
        identifier(Arc), {print_literal("_event");"}, print_newline}})) |
({get_all_destinations(Arc,Gates1),
 get_all_destinations(Arc,Gates2)},
 waiting_place_evaluations(Gates1),
 {print_literal("event = await_any_mail(&mail);"), print_newline,
  print_literal("switch(event) {"), print_newline,
   indent(3)},
  event_alternatives(Gates2),
  {indent(-3),
   print_literal(")"), print_newline}}).

/* The next two rules generate C code which will mask or unmask the
   mailboxes if the corresponding gate buttons (contained in "Gates")
   have a waiting place associated with them. */
waiting_place_evaluations(Gates) -->
{is_mpty(Gates)} |
({extract_element(Gates,Gate,Tail)},
 ((waiting_place_evaluation(Gate),
  waiting_place_evaluations(Tail)) |
  waiting_place_evaluations(Tail))).

waiting_place_evaluation(Gate) -->
{get_waiting_place(Gate,WaitingPlace),
 print_literal("if ("), identifier(WaitingPlace),
 {print_literal(") {"), print_newline, indent(3),
  print_literal("unmask_mailbox("), identifier(Gate),
 {print_literal(");"), print_newline, indent(-3),
  print_literal("} else {"), print_newline,
   indent(3),
  print_literal("mask_mailbox("), identifier(Gate),
 {print_literal(");"), print_newline, indent(-3),
  print_literal(")"), print_newline}.

/* Generates a C function call to start the special timing flag based
   on the information associated with both the pull arc and the time
   out. */
start_delay_alternative(Arc,Timer) -->
{get_period(Timer,Time),
 print_literal("signal_flag_after_time("),
 expression(Time), {print_literal(",")},
 identifier(Arc), {print_literal("_timer");"}, print_newline}.

/* The next two rules generate the C code that gets executed for each
   event that can occur when the JNX task is waiting for mail. The
   code is retrieved from the list of gate buttons in "Gates" */
event_alternatives(Gates) -->
{is_empty(Gates)} |
({extract_element(Gates,Gate,Tail)},
 event_alternative(Gate),

```



```

event_alternatives(Tail)).

event_alternative(Gate) -->
{print_literal("case ")},
identifier(Gate), {print_literal("_event:"), print_newline,
indent(3)},
(({get_engine(Gate, EngineNode),
  get_formal_parameters(Gate, FormalParms1),
  get_formal_parameters(Gate, FormalParms2),
  get_critical_MCL_code(EngineNode, MCLCritical),
  get_post_rdv_MCL_code(EngineNode, MCLPostRen)}),
receive_incoming_data(Gate, FormalParms1),
sequence_of_statements(MCLCritical),
assign_outgoing_data(Gate, FormalParms2),
{print_literal("reply(")},
identifier(Gate), {print_literal(");"), print_newline},
sequence_of_statements(MCLPostRen),
{print_literal("break;"), print_newline,
indent(-3)})) |
(receive_incoming_data(Gate, FormalParms1),
{print_literal("/* NULL */"), print_newline,
print_literal("reply(")}, identifier(Gate),
{print_literal(");"), print_newline},
assign_outgoing_data(Gate, FormalParms2),
{print_literal("break;"), print_newline,
indent(-3)}))}.

/* Generates C code which is executed when the timing flag created
from the pull arc and time out times out. */
delay_alternative(Arc, Timer) -->
{print_literal("case ")},
identifier(Arc), {print_literal("_event:"), print_newline,
indent(3)},
get_all_outgoing_arcs(Timer, SrcArcs)},
timer_arc_list(SrcArcs),
{print_literal("break;"), print_newline,
indent(-3)}.

/* Processes any arc whose source is a time out. */
timer_arc_list(Arcs) -->
{is_empty(Arcs)} |
({extract_element(Arcs, Arc, Tail)},
process_arc(Arc),
timer_arc_list(Tail)).

/* Determines the C code to generate based on the type of arc passed
to the rule. */
process_arc(Arc) -->
subprogram_call(Arc) |
async_entry_call(Arc) |

```

```

    timed_entry_call(Arc) |
    regular_entry_call(Arc) |
    an_ACM_call_stmt(Arc) |
    select_stmt(PullArc) |
    simple_accept(PullArc).

/* Generates C code to receive mail from a single JNX mailbox. This
   rule is invoked when the pull arc's destination is a gate button and
   no time out is attached to the arc. */
simple_accept(Arc) -->
    {get_destination(Arc,Gate),
     print_literal("await_mail(")},
    identifier(Gate), {print_literal(", &mail);"), print_newline},
    ({get_engine(Gate,EngineNode),
     get_formal_parameters(Gate,FormalParms1),
     get_formal_parameters(Gate,FormalParms2),
     get_critical_MCL_code(EngineNode,MCLCritical),
     get_post_rdv_MCL_code(EngineNode,MCLPostRen)},
     receive_incoming_data(Gate,FormalParms1),
     sequence_of_statements(MCLCritical),
     assign_outgoing_data(Gate,FormalParms2),
     {print_literal("reply(")},
     identifier(Gate), {print_literal(";"), print_newline},
     sequence_of_statements(MCLPostRen)) |
    (receive_incoming_data(Gate,FormalParms1),
     {print_literal("/* NULL */"), print_newline,
      print_literal("reply(")},
      identifier(Gate), {print_literal(";"), print_newline},
      assign_outgoing_data(Gate,FormalParms2))).

/* The next two rules extract incoming mail from the mail box and
   assign it to variables having the same names as the gate buttons'
   formal parameters. */
receive_incoming_data(Gate,FormalParms) -->
    {is_empty(FormalParms)} |
    ({extract_element(FormalParms,FormalParm,Tail)},
     receive_data_element(Gate,FormalParm),
     receive_incoming_data(Gate,Tail)).

receive_data_element(Gate,FormalParm) -->
    {get_mode_type(FormalParm,Mode)},
    ({is_out_mode(Mode)} |
     (identifier(FormalParm), {print_literal(" = ((")},
      identifier(Gate),
      {print_literal("_parms *") mail)->"}), identifier(FormalParm),
      {print_literal(";"), print_newline})).

/* The next two rules extract the data from the variables having the
   same names as the gate buttons' formal parameters and assign the
   data to the well known memory location used by the mailbox to

```

```

transfer information. */
assign_outgoing_data(Gate, FormalParms) -->
{is_empty(FormalParms)} |
({extract_element(FormalParms, FormalParam, Tail)},
 assign_data_element(Gate, FormalParam),
 assign_outgoing_data(Gate, Tail)).

assign_data_element(Gate, FormalParam) -->
{get_mode_type(FormalParam, Mode)},
({is_in_mode(Mode)} |
 ({print_literal("("), identifier(Gate),
  {print_literal("_parms *") mail)->"}},
  identifier(FormalParam), {print_literal(" = ")}},
  identifier(FormalParam), {print_literal(";"), print_newline})).

/* The next four rules generate one or more C function calls from a
MCL print statement */
print_stmt(MCLStmt) -->
{is_print_stmt(MCLStmt),
 get_print_expressions(MCLStmt, PrintExpressions)},
print_expression_list(PrintExpressions).

print_expression_list(Expressions) -->
{is_empty(Expressions)} |
({extract_element(Expressions, Expression, Tail)},
 ({is_newline(Expression),
  print_literal("print_newline();"), print_newline} |
  ({is_var_name(Expression),
   get_var_type(Expression, Type)},
   print_var_expression(Expression, Type)) |
  print_expression(Expression)),
 print_expression_list(Tail)).

print_expression(Expression) -->
({is_integer_expression(Expression),
  print_literal("print_integer("), expression(Expression),
  {print_literal(");"), print_newline} |
  ({is_real_expression(Expression),
   print_literal("print_real("), expression(Expression),
   {print_literal(");"), print_newline} |
  ({is_boolean_expression(Expression),
   print_literal("print_boolean("), expression(Expression),
   {print_literal(");"), print_newline} |
  ({is_string_expression(Expression),
   print_literal("print_string("), expression(Expression),
   {print_literal(");"), print_newline} |
  ({is_list_expression(Expression),
   print_literal("print_list("), expression(Expression),
   {print_literal(");"), print_newline} |
  ({is_set_expression(Expression),

```

```

    print_literal("print_set("), expression(Expression),
    {print_literal(");"), print_newline}).

print_var_expression(Expression,Type) -->
    ({is_integer(Type),
     print_literal("print_integer("), expression(Expression),
     {print_literal(");"), print_newline}) |
    ({is_real(Type),
     print_literal("print_real("), expression(Expression),
     {print_literal(");"), print_newline}) |
    ({is_boolean(Type),
     print_literal("print_boolean("), expression(Expression),
     {print_literal(");"), print_newline}) |
    ({is_string(Type),
     print_literal("print_string("), expression(Expression),
     {print_literal(");"), print_newline}) |
    ({is_list(Type),
     print_literal("print_list("), expression(Expression),
     {print_literal(");"), print_newline}) |
    ({is_set(Type),
     print_literal("print_set("), expression(Expression),
     {print_literal(");"), print_newline}).

/* Processes expressions which can be used in to initialize C
   variables in C variable declarations. */
constant_expression(Expression) -->
    integer_literal(Expression) |
    real_literal(Expression) |
    boolean_literal(Expression).

/* Transforms MCL expression to equivalent C expressions. */
expression(Expression) -->
    expression_component(Expression) |
    type_conversion(Expression) |
    indexed_component(Expression) |
    slice(Expression) |
    selected_component(Expression) |
    random_literal(Expression) |
    string_literal(Expression) |
    list_literal(Expression) |
    set_literal(Expression) |
    integer_literal(Expression) |
    real_literal(Expression) |
    boolean_literal(Expression) |
    simple_var(Expression).

/* Generates a C identifier from a MCL identifier */
simple_var(VarName) -->
    {is_var_name(VarName)},
    identifier(VarName).

```

```

/* Generates a C function call which has the same functionality as a
MCL indexing operator. */
indexed_component(Name) -->
{is_index_component(Name),
 get_lhs(Name,Symbol),
 get_rhs(Name,Expression)},
(({is_string_expression(Symbol),
  print_literal("str_element(")},
 identifier(Symbol), {print_literal(", ")},
 expression(Expression),
 {print_literal(")")} |
 ({is_list_expression(Symbol),
  print_literal("list_element(")},
 identifier(Symbol), {print_literal(", ")},
 expression(Expression),
 {print_literal(")")})).

/* As all C components are flattened the left hand side of a MCL
selected component is not needed. */
selected_component(Name) -->
{is_selected_component(Name),
 get_rhs(Name,Expression)},
 expression(Expression).

/* Generates a C function call to create a string which is
equivalent to the MCL string literal. */
string_literal(Name) -->
{is_string_literal(Name),
 get_string_literal(Name,Literal),
 print_literal("create_string(", print_quote,
 print_literal(Literal), print_quote,
 print_literal(")")}).

/* Generates a C integer literal from a MCL integer literal. */
integer_literal(Name) -->
{is_integer_literal(Name),
 get_integer_literal(Name,Literal),
 print_integer(Literal)}.

/* Generates a C real literal from a MCL real literal. */
real_literal(Name) -->
{is_real_literal(Name),
 get_real_literal(Name,Literal),
 print_real(Literal)}.

/* Generates a C boolean literal from a MCL boolean literal. */
boolean_literal(Name) -->
{is_boolean_literal(Name),
 get_boolean_literal(Name,Literal),

```

```

    print_boolean(Literal)).

/* Processes unary and binary MCL expressions. */
expression_component(Expression) -->
    {is_expression(Expression)},
    (unary_expression(Expression) |
     binary_expression(Expression)).

/* Processes unary MCL expressions. */
unary_expression(Expression) -->
    {is_unary_expression(Expression),
     get_operator(Expression,Operator),
     get_rhs(Expression,RExpression)},
    (unary_plus_operator(Operator,RExpression) |
     unary_minus_operator(Operator,RExpression) |
     absolute_operator(Operator,RExpression) |
     length_operator(Operator,RExpression) |
     not_operator(Operator,RExpression)).

/* A C function call to a function which is functionally equivalent to
   the length operator is generated. */
length_operator(Operator,Expression) -->
    {is_length_operator(Operator)},
    (({is_string_expression(Expression),
       print_literal("str_len(")},
      expression(Expression),
      {print_literal(")")} |
     ({is_list_expression(Expression),
       print_literal("list_len(")},
      expression(Expression),
      {print_literal(")")} |
     ({is_set_expression(Expression),
       print_literal("set_size(")},
      expression(Expression),
      {print_literal(")"})))).

/* Processes binary MCL expressions. */
binary_expression(Expression) -->
    {is_binary_expression(Expression),
     get_operator(Expression,Operator),
     get_lhs(Expression,LExpression),
     get_rhs(Expression,RExpression)},
    (plus_operator(Operator,LExpression,RExpression) |
     minus_operator(Operator,LExpression,RExpression) |
     multiply_operator(Operator,LExpression,RExpression) |
     divide_operator(Operator,LExpression,RExpression) |
     exponentiation_operator(Operator,LExpression,RExpression) |
     mod_operator(Operator,LExpression,RExpression) |
     max_operator(Operator,LExpression,RExpression) |
     min_operator(Operator,LExpression,RExpression) |

```

```

equal_operator(Operator,LExpression,RExpression) |
not_equal_operator(Operator,LExpression,RExpression) |
greater_than_operator(Operator,LExpression,RExpression) |
less_than_operator(Operator,LExpression,RExpression) |
greater_than_equal_operator(Operator,LExpression,RExpression) |
less_than_equal_operator(Operator,LExpression,RExpression) |
and_operator(Operator,LExpression,RExpression) |
or_operator(Operator,LExpression,RExpression) |
xor_operator(Operator,LExpression,RExpression) |
with_operator(Operator,LExpression,RExpression) |
in_operator(Operator,LExpression,RExpression) |
notin_operator(Operator,LExpression,RExpression) |
less_operator(Operator,LExpression,RExpression) |
subset_operator(Operator,LExpression,RExpression)).

/* In the case of MCL strings, lists, and sets, this rule generates C
function calls to C functions which are functionally equivalent to
the plus operator. In the case of MCL reals and integers, this rule
generates Ada expressions using the plus operator. */
plus_operator(Operator,LExpression,RExpression) -->
{is_plus_operator(Operator)},
(({is_string_expression(LExpression),
  is_string_expression(RExpression),
  print_literal("str_concat(")},
  expression(LExpression),
  {print_literal(",")},
  expression(RExpression),
  {print_literal(")")})) |
({is_list_expression(LExpression),
  is_list_expression(RExpression),
  print_literal("list_concat(")},
  expression(LExpression),
  {print_literal(",")},
  expression(RExpression),
  {print_literal(")")})) |
({is_set_expression(LExpression),
  is_set_expression(RExpression),
  print_literal("set_union(")},
  expression(LExpression),
  {print_literal(",")},
  expression(RExpression),
  {print_literal(")")})) |
({print_literal("(")},
  expression(LExpression),
  {print_operator(Operator)},
  expression(RExpression),
  {print_literal(")"}))}.

/* Generates a C expression using the C mod operator rather than the
MCL mod operator. */

```

```

mod_operator(Operator, LExpression, RExpression) -->
    {is_mod_operator(Operator),
      print_literal("(")},
    expression(LExpression),
    {print_literal(" % ")},
    expression(RExpression),
    {print_literal(")")}

/* In the case of MCL strings, lists, and sets, generates a C function
   call to a C function which is functionally equivalent to the equal
   operator. In the case of MCL integers and reals, generates a C
   expression using the C equal operator rather than the MCL equal
   operator. */
equal_operator(Operator, LExpression, RExpression) -->
    {is_equal_operator(Operator)},
    (({is_string_expression(LExpression),
        is_string_expression(RExpression),
        print_literal("str_equal(")},
        expression(LExpression),
        {print_literal(", ")},
        expression(RExpression),
        {print_literal(")")} |
      ({is_list_expression(LExpression),
        is_list_expression(RExpression),
        print_literal("list_equal(")},
        expression(LExpression),
        {print_literal(", ")},
        expression(RExpression),
        {print_literal(")")} |
      ({is_set_expression(LExpression),
        is_set_expression(RExpression),
        print_literal("set_equal(")},
        expression(LExpression),
        {print_literal(", ")},
        expression(RExpression),
        {print_literal(")")} |
      {print_literal("(")},
        expression(LExpression),
        {print_literal(" == ")},
        expression(RExpression),
        {print_literal(")"})))).

/* In the case of MCL strings, lists, and sets, generates a C function
   call to a C function which is functionally equivalent to the not
   equal operator. In the case of MCL integers and reals, generates a C
   expression using the C not equal operator rather than the MCL not
   equal operator. */
not_equal_operator(Operator, LExpression, RExpression) -->
    {is_not_equal_operator(Operator)},
    (({is_string_expression(LExpression),

```



```

        is_string_expression(RExpression),
        print_literal("str_not_equal(")},
        expression(LExpression),
        {print_literal(",")},
        expression(RExpression),
        {print_literal(")")})) |
    ({is_list_expression(LExpression),
     is_list_expression(RExpression),
     print_literal("list_not_equal(")},
     expression(LExpression),
     {print_literal(",")},
     expression(RExpression),
     {print_literal(")")})) |
    ({is_set_expression(LExpression),
     is_set_expression(RExpression),
     print_literal("set_not_equal(")},
     expression(LExpression),
     {print_literal(",")},
     expression(RExpression),
     {print_literal(")")})) |
    ({print_literal("(")},
     expression(LExpression),
     {print_literal(" != ")},
     expression(RExpression),
     {print_literal(")"))}).

/* Generates a C expression using the C and operator rather than the
   MCL and operator. */
and_operator(Operator,LExpression,RExpression) -->
    {is_and_operator(Operator),
     print_literal("(")},
     expression(LExpression),
     {print_literal(" && ")},
     expression(RExpression),
     {print_literal(")")}.

/* Generates a C identifier from the name of the MachineChart
   component passed to the rule. */
identifier(Node) -->
    {get_node_string(Node,STRING),
     print_literal(STRING)}.

```

Appendix H

A Sampling of the C/JNX Library Used to Support MCL Expressions and Data Types

```
/*-----  
                                MCL Type Declarations  
  
    This header file contains the necessary definitions for all MCL types.  
    -----*/  
  
#define NULL 0  
  
/* Enumerated type declarations used for tagging list  
   or set types */  
  
typedef enum {integer_type,real_type,boolean_type,  
             string_type,list_type,set_type} MCLTypes;  
  
/* Generic list and set element declarations */  
  
typedef int *ListElement;  
typedef int *SetElement;  
  
/* List structure */  
  
struct LIST {  
    ListElement item;  
    struct LIST *next;  
};  
  
/* Set structure */  
  
struct SET {
```

```

    SetElement item;
    struct SET *next;
};

/* Declaration of MCL types */

typedef int MCLInteger;
typedef float MCLReal;
typedef enum {False, True} MCLBoolean;

typedef struct {
    int the_length;
    char *structure;
} MCLString;

typedef struct {
    MCLTypes contents;
    struct LIST structure;
} MCLList;

typedef struct {
    MCLTypes contents;
    struct SET structure;
} MCLSet;

/*-----
                                List Library

Contains procedures and functions which support MCL operations on
lists. The lists can be composed of any MCL type (integer, reals,
boolean, strings, sets, or other lists). The lists are implemented
as dynamic singly linked lists.
-----*/

#include "typedefs.h"

void print_list( MCLList value );
/* MCL Code: print value
   Outputs the list to standard I/O. */

MCLList copy_list( MCLList value );
/* Needed to support list assignment. */

MCLList create_list( ListElement value, MCLList tail );
/* Creates a list from a MCL list literal by recursively
   passing the elements of the list literal to the function. */

```

```

MCLList empty_list();
/* Creates an empty list. */

MCLInteger list_len( MCLList value );
/* MCL Code: #value
   Returns the length of the list. */

ListElement list_element( MCLList value, MCLInteger index );
/* MCL Code: value(index)
   Returns the list element at offset "index". An error
   results if "index" < 1 or if "index" > the length of
   the list. */

MCLList sub_list( MCLList value, MCLInteger low, MCLInteger high );
/* MCL Code: value(low .. high)
   Returns a sublist of the elements ranging from the offset "low"
   to the offset "high". An error results if "low" > "high", if
   "low" < 1, or if "high" > the length of the list. */

void modify_list_element( MCLList value, MCLInteger index,
                          ListElement element );
/* MCL Code: value(index) := element
   The location at offset "index" is assigned the value "element".
   An error results if "index" < 1, or if index > the length of
   the list */

void modify_sub_list( MCLList value, MCLInteger low,
                     MCLInteger high, MCLList sub_list );
/* MCL Code: value(low .. high) := sub_list
   The sublist consisting of the elements from "low" to "high"
   are replaced with the sublist passed to the function. This
   may result in expansion or shrinkage of the list. If "low"
   = "high" + 1 then the sublist is inserted into the original
   list at "high". If "low" = "high" + 1 and "high" = the
   length of the list then the sublist is appended to the
   original list. An error results if "low" < 1, "low" >
   "high" + 1, or "high" > the length of the list. */

MCLList list_concat( MCLList value1, MCLList value2 );
/* MCL Code: value1 + value2
   Returns a list which contains the first list concatenated to
   the second list. */

MCLList multiply_list( MCLList value, MCLInteger multiplier );
/* MCL Code: value * multiplier
   Returns a list which contains "multiplier" copies of the
   list "value" all concatenated together. */

```

```
MCLList append_list (MCLList value, ListElement element );
/* MCL Code: value with element
   Returns a list with "element" appended to list "value". */

MCLBoolean list_equal( MCLList value1, MCLList value2 );
/* MCL Code: value1 = value2
   Returns true if both lists are identical. */

MCLBoolean list_not_equal( MCLList value1, MCLList value2 );
/* MCL Code: value1 != value2
   Returns true if the lists are not identical. */

MCLBoolean list_in( ListElement element, MCLList value );
/* MCL Code: element in value
   Returns true if the element is in the list. */

MCLBoolean list_not_in( ListElement element, MCLList value );
/* MCL Code: element not in value
   Returns true if the element is not in the list. */
```

END

28·0|5·9|2

FIN