1

MicroD

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.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

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

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.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, S.R.C. 1970, c. C-30.

DESIGN AND IMPLEMENTATION OF ASCGEN,
A PROLOG TOOL FOR TRANSLATING GRAPHICAL
DESIGNS INTO ADA CODE

by

Kenneth William Van Der Loo, B.E.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
Faculty of Engineering
Carleton University
Ottawa, Ontario
April, 1987
(c) copyright
1987, Kenneth William Van Der Loo

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-46287-6

The undersigned hereby recommend to the Faculty of Graduate Studies and Research acceptance of the thesis, "DESIGN AND IMPLEMENTATION OF ASCGEN, A PROLOG TOOL FOR TRANSLATING GRAPHICAL DESIGNS INTO ADA CODE", submitted by Kenneth William Van Der Loo, B.E.Sc., in partial fulfillment of the requirements for the degree of Master of Engineering.

Thesis Supervisor

Chairman,

Department of Systems and
Computer Engineering

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
Ottawa, Ontario
April, 1987

ABSTRACT

The Carleton Embedded System Design Environment (CAEDE) Project (Carleton University, Ottawa, Canada) was initiated to investigate the application of an iconic design environment to the design of multitasking, embedded systems. This thesis covers the development of one of the CAEDE tools. The tool, an Ada Skeleton Code GENERator (ASCGEN), transforms a C-Prolog design data base, representing a graphical system design, into Skeleton Ada (S-Ada) code. ASCGEN was implemented in C-Prolog. The requirements analysis, requirements definition, design architecture and algorithms, implementation, testing and evaluation of ASCGEN are discussed. A translation walk-through illustrates the architectural and algorithmic choices made. A performance improvement case history illustrates optimizations applied to the prototype. The end result was a tool which produced almost compilable S-Ada code, including comments, at 600-700 lines per minute.

ACKNOWLEDGEMENTS

I wish to express my gratitude for my new-born son Tariq, who gave me inspiration, love, and smiles.

I also wish to express my appreciation to my wife Bushra, who gave me unfailing love, support and guidance.

I wish to thank Professor Ray J. A. Buhr for originally suggesting the thesis topic and for his guidance through the project.

Thanks are due, as well, to Professor Archie Bowen for his assistance and suggestions, and to the CAEDE project members, especially Brian Mondoux, and Gerald Karam.

CONTENTS

ACKNOWLEDGEMENTS.....	ii
CONTENTS.....	iii
LIST OF FIGURES.....	ix
TEXT GLOSSARY.....	xi
CAEDE-NOTATION GLOSSARY.....	ix
CHAPTER 1 INTRODUCTION.....	1-1
1.1 BACKGROUND.....	1-1
1.2 CONTRIBUTIONS.....	1-2
1.3 RELATED WORK.....	1-3
1.4 OUTLINE OF THESIS.....	1-7
CHAPTER 2 SOFTWARE REQUIREMENTS DEFINITION.....	2-1
2.1 INTRODUCTION.....	2-1
2.2 USER REQUIREMENTS ANALYSIS.....	2-2
2.2.1 GENERAL CONSTRAINTS.....	2-3
2.3 EXTERNAL INTERFACE REQUIREMENTS.....	2-4
2.4 SPECIFIC REQUIREMENTS.....	2-5

2.4.1	OUTPUT (S-ADA) REQUIREMENTS DEFINITION.....	2-5
2.4.2	ERROR HANDLING.....	2-8
CHAPTER 3	ASCGEN DESIGN AND IMPLEMENTATION.....	3-1
3.1	INTRODUCTION.....	3-1
3.2	PERSPECTIVE OF ASCGEN WITHIN CAEDE.....	3-2
3.3	ASCGEN ARCHITECTURE.....	3-5
3.3.1	OVERVIEW.....	3-5
3.3.2	PHASE DETAILS.....	3-9
3.3.2.1	Analysis Phase.....	3-10
3.3.2.2	Transformation Phase.....	3-12
3.3.2.3	Code Generation Phase.....	3-15
3.3.3	REJECTED DESIGN ALTERNATIVES.....	3-16
3.3.3.1	Bi-Directional Translation.....	3-16
3.3.3.2	DIANA as an Intermediate Representation.....	3-19
3.4	TRANSLATION WALK-THROUGH.....	3-20
3.4.1	ASCGEN INPUTS.....	3-23
3.4.2	ASCGEN OUTPUTS.....	3-24
3.4.3	ASCGEN PROCESSING.....	3-32
3.5	LEVEL OF IMPLEMENTATION.....	3-34
3.6	SUMMARY.....	3-35
CHAPTER 4	ASCGEN TESTING.....	4-1

4.1	INTRODUCTION.....	4-1
4.2	VALIDATION TESTING.....	4-2
4.3	IMPROVEMENT HISTORY.....	4-13
4.4	PERFORMANCE EVALUATION.....	4-20
4.5	SUMMARY.....	4-24
CHAPTER 5	CONCLUSIONS AND RECOMMENDATIONS.....	5-1
5.1	CONCLUSIONS.....	5-1
5.2	RECOMMENDATIONS FOR FURTHER WORK.....	5-4
APPENDIX A	PROLOG PDL SEMANTIC REQUIREMENTS	
DEFINITION.....		A-1
A.1	INTRODUCTION.....	A-1
A.1.1	AN OVERVIEW OF THE CAEDE-NOTATION.....	A-2
A.1.2	IDENTIFIERS.....	A-3
A.1.3	TEMPLATES AND INSTANCES.....	A-5
A.1.4	TEMPLATE USAGE.....	A-6
A.1.4.1	Package_family And Task_family.....	A-6
A.1.4.2	channel_family.....	A-7
A.1.5	ACTION_CHAINS.....	A-8
A.1.5.1	Action Chain Syntax.....	A-8
A.1.5.2	Interaction Semantics.....	A-10
A.1.5.3	Interaction-Chain Semantics.....	A-11
A.1.5.4	Creation-Chain Semantics.....	A-11

A.1.5.5	Data Flows.....	A-12
A.1.6	FORMAL PARAMETERS.....	A-12
APPENDIX B	SKELETON ADA (S-ADA) TARGET LANGUAGE	
DEFINITION.....		B-1
B.1	INTRODUCTION.....	B-1
B.1.1	SCOPE.....	B-1
B.1.2	EXTENT OF THE STANDARD.....	B-2
B.1.3	STRUCTURE OF THE S-ADA STANDARD.....	B-2
B.1.4	S-ADA DESIGN GOALS.....	B-3
B.1.5	S-ADA LANGUAGE SUMMARY.....	B-3
B.1.6	METHOD OF S-ADA DESCRIPTION AND SYNTAX NOTATION.....	B-4
B.2	DETAILED S-ADA LANGUAGE DEFINITION.....	B-5
APPENDIX C	FUNCTIONAL TEST DESIGN SPECIFICATION.....	C-1
C.1	DECLARATIONS.....	C-1
C.2	BODIES.....	C-2
C.3	COMMENTS.....	C-3
C.4	ACTION CHAIN SOURCES.....	C-3
C.5	ACTION CHAIN DESTINATIONS.....	C-3
C.6	ACTION CHAIN LINKS.....	C-4
C.7	PARAMETERS.....	C-5
C.8	TEMPLATE DECLARATIONS.....	C-5
C.9	CREATION CHAIN TARGETS.....	C-5

C.10	CREATION CHAIN SOURCES.....	C-6
C.11	CREATION CHAIN LINKS.....	C-6
C.12	OTHER REQUIREMENTS.....	C-6
APPENDIX D	ASCGEN-DEMO S-ADA CODE.....	D-1
APPENDIX E	ADDITIONS TO MAKE ASCGEN-DEMO COMPILE.....	E-1
APPENDIX F	SINGLE CHANNEL S-ADA CODE.....	F-1
APPENDIX G	CHANNEL INSTANCE S-ADA CODE.....	G-1
APPENDIX H	ASCGEN CODE LISTINGS.....	H-1
H.1	ACTION_CHAINS.....	H-1
H.2	CG_COMMENTS.....	H-9
H.3	CODEGEN_VAX.....	H-15
H.4	DECL_PARTS_3.9.....	H-17
H.5	DECLARATIONS_3.1.....	H-24
H.6	FORMAT_STUB.....	H-31
H.7	GENERIC_BDY.....	H-37
H.8	GENERIC_DECL.....	H-38
H.9	INSTANCES.....	H-39
H.10	PACK_STRUCT_7.1.....	H-59
H.11	STATEMENTS_5.1.....	H-63
H.12	SUBPGM_BODY_6.3.....	H-73

H.13	TASK_TYPE_DECL.....	H-77
H.14	TASK_TYPE_BDY.....	H-78
H.15	TASKS_9.1.....	H-79

LIST OF FIGURES

Figure 3.1	ASCGEN Translation Clause Files.....	3-4
Figure 3.2	ASCGEN Logical Translation Phases.....	3-8
Figure 3.3	CAEDE-notation Procedure Design for Translation.....	3-22
Figure 3.4	Prolog Fact Schemas Relevant to Figure 3.3 Design.....	3-25
Figure 3.5	Prolog Facts for Figure 3.3 Design.....	3-26
Figure 3.6	S-Ada Schema Related to Figure 3.3 Design.....	3-27
Figure 3.7	Target S-Ada produced for Figure 3.3 Design.....	3-29
Figure 3.8	Procedure Transformation Prolog Clauses.....	3-30
Figure 4.1	"ASCGEN-DEMO" CAEDE-notation Design.....	4-5
Figure 4.2	"SINGLE CHANNEL" CAEDE-notation Design.....	4-8
Figure 4.3	"CHANNEL INSTANCE" CAEDE-notation Design.....	4-11

Figure 4.4 ASCGEN Performance Improvements
Summary.....4-15

Figure 4.5 ASCGEN Performance Statistics Summary.....4-21

TEXT GLOSSARY

Action-chain

Another name for a combination of one or more graphical call primitives that represents a CAEDE-notation call.

Anonymous Variable

The anonymous variable (`_`) is used as an argument when the details of a matched term are not required.

ASCGEN

Ada Skeleton Code Generator.

AST

Abstract Syntax Tree.

Atom

One of the two kinds of "constants".

CAEDE

Carleton Embedded Systems Design Environment.

CAEDE-notation

The name for the graphical notation used for inputting designs via the CAEDE human interface.

Clause

In Prolog, this refers to either a fact or a rule.

Command

In Prolog, this refers to a headless clause starting with `"?-"` or `"::-"`. It requires Prolog to prove the command's truth or falsity. Is is also called a negative clause.

Constant

In Prolog, constants name specific objects or relationships. Constants are of two kinds: atoms (e.g. a, void123, 'Code-Generate', ?-, :-), and integers (e.g. 1, 93).

Fact

In Prolog, facts state arbitrary relationships between arguments. Facts are clauses without a body, and may have 0 to N arguments (e.g. a_0_argument_fact., owns(john, book)).

Functor

In Prolog, this refers to the name of a "structure".

Goal

The logical conclusion a Prolog interpreter attempts to prove from its facts and rules.

IDL

Item Descriptor List.

Logic Store

A name for the Prolog PDL generated from the CAEDE Picture Store and System Store data bases, for a particular design.

PDL

Program Design Language.

Picture Store

A CAEDE data base containing graphic information on a design.

Predicate

In Prolog, this refers to the name of a fact or rule.

Procedure

In Prolog, this refers to the collection of clauses for a given predicate.

Prolog PDL

The schema which defines the form of the Prolog Design Data Base input to ASCGEN.

Rule

In Prolog, a rule states the dependence of facts on other facts.

SDWA

A graphics-based Ada design methodology due to Buhr [Buhr 84].

Session Goal

The goal which C-Prolog constantly tries to prove. A side effect is that a user's commands are executed and goals are solved.

Structure

In Prolog, this refers to a single object which consists of a collection of other objects, called components (e.g. owns(john, book)). A structure is written by specifying its functor ("owns"), and its components surrounded by parenthesis ("(john, book)").

System Store

A CAEDE data base containing logical and supplementary information on a design.

Term

Prolog programs are built from terms. A term is either a constant, a variable or a structure. Each term is written as a sequence of characters.

Variables

In Prolog, variables look like atoms, but they begin with a capital letter or an underscore (e.g. X, Code_Generation, RESULT, _END). They have a scope of visibility that is restricted to the clause in which they appear. Refer also to "Anonymous Variable".

CAEDE-NOTATION GRAPHICS, GLOSSARY

CAEDE-NOTATION GRAPHICS GLOSSARY (continued)

CHAPTER 1 INTRODUCTION

1.1 BACKGROUND

The decreasing cost of graphics hardware and software has stimulated research and development of tools to support graphics-based software design methodologies. These design methodologies can simplify the conceptualization, specification, and implementation of large, and/or complex software systems. The Carleton Embedded Design Environment (CAEDE) was initiated at Carleton University as an experimental prototype of an iconic design environment that supports the CAEDE-notation graphical design methodology inspired by Ada [Buhr 84, 85, 85A]. This project was originally targeted for development of modular, multitasking, embedded systems, especially communications protocols.

This thesis contributes the design, implementation, and testing of one of the automated tools for CAEDE: an Ada Skeleton Code GENERator (ASCGEN). ASCGEN performs a translation of a Prolog representation of a CAEDE-notation design into Skeleton Ada (S-Ada) code.

1.2 CONTRIBUTIONS

The overall achievement of this thesis is the development of ASCGEN, a usable, prototype software tool to convert a graphics-based design representation into skeleton code. The input to ASCGEN is a CAEDE-specified Prolog Program Design Language (PDL), which represents the essential parts of a CAEDE-notation design. The target language is Skeleton Ada (S-Ada). ASCGEN was implemented in C-Prolog/1.4. The following outlines the contributions that were made in the development of ASCGEN:

A requirements analysis and requirements definition for ASCGEN was made. This produced a specification of the target language (S-Ada) for representing a CAEDE-notation design. This also produced a specification of the semantic constraints and interpretation of the Prolog PDL, to define correct translation into S-Ada. The latter specification could be used for the development of a consistency checking tool.

ASCGEN was then designed, implemented and tested, producing a useful tool. Additional code was added to the S-Ada produced from one of the test inputs to make it compilable. This demonstrated the correctness and completeness of the S-Ada code produced by ASCGEN.

Performance improvement efforts were applied to ASCGEN, resulting in an order of magnitude speed improvement and a similar space reduction, at run-time.

The development of ASCGEN contributed to the successful integration and debugging of CAEDE.

Experience was gained in ways to apply the Prolog language to the task of translating from a Prolog Design schema into a BNF language schema.

1.3 RELATED WORK

In this section, we discuss some research efforts that were relevant to the development of ASCGEN. We also include comments on the relationship of these works to ours.

Warren [Warren 77] describes a technique for implementing compilers using Logic Programming and Prolog. His report illustrates the design of a compiler to translate from a small Algol-like language to the machine language of a hypothetical one-accumulator computer.

The logical phases and pass algorithms used are traditional for compilers [Waite 84]. The compiler performs one pass/phase over the source code and intermediate structures. The definitions of the intermediate data structures are simultaneously specified and implemented in Prolog. The intermediate structures themselves are synthesized and retained through the remaining compilation processes in the form of parameters to Prolog clauses.

The need for storage management techniques is mentioned as a necessary feature of a full compiler implementation in order to avoid exhausting memory resources. Warren suggested these could be accomplished with programmer-managed extensions beyond those provided by basic Prolog.

The logical phases chosen for ASCGEN are similar to those in Warren's compiler. However, the inputs, outputs and algorithms are not similar. There are basic differences in the schema for the inputs, outputs and internal representation of the transformation products between the two translators. The input to ASCGEN is a set of facts, rather than a conventional, sequence of tokens conforming to a grammar. The output of ASCGEN is source Ada, rather than machine code. Internally, ASCGEN does not pass massive intermediate representations of the data (i.e. single, recursive tree parameters for Warrens implementation) from phase to phase, but makes use of the existing facts. These differences required different algorithms be used in ASCGEN than used in Warren's compiler.

Both Warren's and our implementation required careful utilization of the available process space in order to handle reasonably-sized designs. This constraint also affected the choice of architecture for ASCGEN.

Since CAEDE was intended as a designer's assistant, some attributes of program design languages were incorporated into the code produced by ASCGEN. The development of Ada was accompanied by research into methodologies for supporting the Ada design process, using Ada as the basis for a design language. Various extensions to Ada have been

developed, to create Ada Program Design Languages (PDLs) with varying degrees of sophistication:

- * A TRW PDL, described in [Hart 82], is an example of a simple syntax language which uses a limited set of Ada syntax and unstructured english to enhance Ada semantics. The syntax could not be rigorously enforced or verified by a checking tool;
- * An IBM PDL, described in [Waugh 80], is an example of an Ada full-syntax PDL which permits full checking, but needs a lot of coding before it is complete enough to be checked;
- * A middle way PDL, described in [Gabber 82], allows for transition between the simple and full-syntax style PDLs, as part of the stepwise refinement at the design level, and can be supported by a PDL checking tool.

The S-Ada Specification incorporates some features of Gabber's PDL. Instead of using an escape delimiter enclosing english comments, S-Ada defines a machine processable string (an Ada, non-terminal identifier), to mark points where the code generated is incomplete. Although S-Ada specifies some useful design information, it is not intended to be a definition of a PDL. Instead, the freedom to choose an Ada PDL was left to the user of the S-Ada code generated by ASCGEN.

Albrecht [Albrecht 80] describes an implementation of a source to source translator between Ada and Pascal, and from Pascal to Ada. The translator was written in Pascal. Each language required separate subsystems for each direction of translation (i.e. the Ada source

analyzer could not be used to synthesize Ada code). The approach used for performing a translation was:

- (a) to define subsets of each language between which there is a straightforward translation;
- (b) to translate each source program into its respective sub-language and then into a common representation by program transformations applied to tree structures;
- (c) to generate the target program from the common representation with a code generator/formater specific to the desired source.

A feature common to Albrecht's subsystems and the implementation of ASCGEN is that all synthesis and analysis tools performed translations in one direction only (none were bi-directional).

In general, the architecture and data representations of the Pascal/Ada translator were not similar to that used for ASCGEN. ASCGEN did not operate on tree-type, intermediate representations of programs, but interacted with the Prolog data base facts on a design.

Butler [Butler 84] describes the DIANA intermediate representation for use within Ada Program Development Environments. The DIANA representation is used for storing an Ada program's syntax and semantic information in an easily accessible form, for use by other Ada development tools. DIANA was examined as a possible intermediate representation in Prolog, to be passed between the translation phases of ASCGEN, in support of reversible translation. However, the requirement

for an intermediate representation was dropped when reversible translation was dropped from the proposed features for ASCGEN.

1.4 OUTLINE OF THESIS

Chapter Two covers the requirements analysis and definition of ASCGEN. This produced the "Prolog PDL Semantic Requirements Definition" (Appendix A) and the "Skeleton Ada (S-Ada) Target Language Definition" (Appendix B). The role and perspective of ASCGEN within the CAEDE project is described in detail. Design constraints predefined by CAEDE are identified and presented in more detail.

Chapter Three investigates the alternative program architectures and choices made in implementing ASCGEN. A tutorial walk-through of a simple design translation is also given.

Chapter Four discusses the testing and validation of ASCGEN. Functional tests and results are included. The code additions to make a sample S-Ada program compile (one of the functional test cases) are also discussed. The improvement history that resulted from performance testing is detailed. Performance statistics are presented for the final implementation of ASCGEN.

Chapter Five lists conclusions for this thesis effort. Possible directions for future work are also discussed.

CHAPTER 2 SOFTWARE REQUIREMENTS DEFINITION

2.1 INTRODUCTION

This chapter provides an overview of the issues, constraints and assumptions leading to the definition of the requirements for ASCGEN. ASCGEN is a prototype tool which translates a Prolog representation of a CAEDE-notation design (a high level, graphics based design), into Skeleton Ada (S-Ada) code, the target source language.

In the following sections, user requirements are analyzed and specified, and CAEDE project constraints and ASCGEN external interfaces identified. Issues and assumptions relevant to the translation of a CAEDE-notation design into S-Ada are also discussed. The major result of the requirements definition phase is a formal specification of the code to be produced (S-Ada, Appendix B).

2.2 USER REQUIREMENTS ANALYSIS

Users of ASCGEN may be categorized into two classes: those that operate CAEDE as a system design/development tool, and those that use the skeleton code for high-level design documentation for implementation of a system.

During an interactive design session with CAEDE, ASCGEN could assist a user in the following ways:

- (a) As a CAEDE-notation design is being refined, there is a need to verify its consistency against the CAEDE-notation methodology and syntax. This could be implemented by analyzing a design for consistency, and providing feedback in the form of error messages in a file. The capability to verify the consistency of the input design against design methodology syntax and semantics, would have application to more than Ada code generation;
- (b) When a CAEDE design session is completed, there is a need for an external representation of the design, in a consolidated form, rather than the multiple views and windows needed within CAEDE to view the information that constitutes a design. This could be implemented by generating annotated skeleton code containing this information.

During the implementation phase, ASCGEN could assist an implementor in the following ways:

- (a) Provide guidance on completing the Ada skeleton code (i.e. to make it compilable). It would therefore be desirable to flag incomplete code areas, where they occur;
- (b) Produce a design specification for implementation in a language other than Ada.

2.2.1 GENERAL CONSTRAINTS

The available alternatives for developing ASCGEN were limited by the following CAEDE project general constraints:

Policies:

- graphics paradigm design notation designated for CAEDE (CAEDE-notation).

ASCGEN role:

- invoked and terminated from iconic interface.
- no user interaction during code generation

Interfaces:

- interface to Logical Structure Manager module in CAEDE.
- Logical Store data base schema: Prolog Program Design Language (PDL).

Implementation hardware:

- Carleton SUN: graphics work station with limited disk swap space.

Implementation software:

- Berkeley BSD/4.2 Unix.
- C-Prolog/1.4 implementation language.
- must run co-resident with other CAEDE subsystems.

2.3 EXTERNAL INTERFACE REQUIREMENTS

The input interface to ASCGEN consists of:

- (a) a prespecified interface module, `c_spawn_cg`, which invokes the C-Prolog interpreter at the request of the Iconic User Interface. The module consults various files which constitute ASCGEN as well as the current CAEDE design Prolog Fact Data Base, starts the translation process, and terminates the interpreter when translation is complete;
- (b) a set of files, which contain Prolog Facts on the current CAEDE-notation design (the facts correspond to a pre-specified schema).

2.4 SPECIFIC REQUIREMENTS

2.4.1 OUTPUT (S-ADA) REQUIREMENTS DEFINITION

The definition of Skeleton Ada (S-Ada), which implements CAEDE-notation designs, is based on the SDWA graphics notation due to Buhr [Buhr 84]. The CAEDE-notation extends SDWA to include some new structures such as "channels", and "templates". There were several activities involved in the definition of S-Ada:

- * Verification that the Prolog Program Design Language (PDL) correctly represented the essence of CAEDE-notation designs;
- * Specification of the translation rules for translating from the CAEDE-notation graphics-based designs (and therefore from the corresponding Prolog PDL fact data base) into S-Ada;
- * Choosing suitable S-Ada code to implement all possible CAEDE-notation designs.

An analysis of the Prolog PDL revealed some limitations in the range of CAEDE-notation designs that the PDL could represent. Some problems were related to how well the translations of CAEDE data bases into inputs to ASCGEN were accomplished. In some cases, essential information present in the CAEDE database SYSTEM STORE (the primary CAEDE design data base) could not be correctly represented in the Prolog

PDL data base. For example, the Prolog PDL schema could not store enough information to represent Ada overloading contained in CAEDE-notation designs. Other problems were related to errors not caught by the CAEDE interface, and which would not be acceptable for translation into S-Ada.

Appendix A identifies the ambiguities in the Prolog PDL and adds additional semantic information to the PDL definition to facilitate translation into S-Ada. It contains a set of requirements which define when a CAEDE-notation design is syntactically and semantically complete enough for translation into S-Ada. This specification guided the implementation of ASCGEN translation rules in Prolog.

Appendix B contains the formal specification of Skeleton Ada (S-Ada). The following paragraphs analyze issues relevant to the development of that specification.

The most significant requirement of S-Ada was that it should implement all possible CAEDE-notation designs (i.e. it should support the design methodology, as implemented in CAEDE). The design notation supported by CAEDE contains structural information and call interactions that may be translated into Ada declarations, and sequence of statements respectively. However, the notation does not currently contain information to permit linear call organization within code (e.g. if n calls originate from an icon, there is no way to specify their order of occurrence in the S-Ada sequence of statements for that icon). There is also currently very little data structure translatable into other than comments.

A desirable feature of S-Ada was easy association of target code with the CAEDE-notation graphics design. Indentation was used to indicate the level of nesting of CAEDE-notation objects. Comments and visual markers were also specified to mark the code corresponding to a CAEDE-notation object. Visual markers are unique character patterns that herald the code for each graphics icon structure. For example, the string ">>>>>>>" is used in headers that precede the S-Ada code implementing the CAEDE-notation "channel" object.

It was desirable that S-Ada programs be as close to being compilable as possible, within the constraints of the completeness of the CAEDE-notation language. Support for compilability requires both correct syntax, and semantics. For example, S-Ada object declarations should be generated before references to them are made. The S-Ada specification defines the level of support of syntax while the associated semantic requirements are derived from the Ada language definition [ALRM 83]. We chose to use Ada-syntax, BNF-like non-terminals in the S-Ada specification at places where complete Ada code could not be generated from the corresponding Prolog facts. These non-terminals can support further manual or tool-assisted completion of code, and provide a means of expanding ASCGEN in future enhancements (when code can be substituted for particular non-terminals). The alternative would have been to generate an Ada full-syntax PDL. By only incorporating general Ada-syntax PDL concepts into S-Ada, we left room for the adoption (by an implementor) of more specific Ada PDLs when an S-Ada program is completed. Semantic comments to guide code completion are specified for S-Ada (for automatic generation by ASCGEN). The locations for embedding comment information (which would be input via the CAEDE interface) are also specified.

The need for configuration control was satisfied by the ASCGEN requirement to embed comments stating the version, source and target files for a CAEDE-notation design being translated, the date, and the version of ASCGEN used.

2.4.2 ERROR HANDLING

A desirable feature of ASCGEN is robustness, and graceful error recovery. If errors should occur when translating inconsistent designs, or internal errors occur for other reasons, ASCGEN should generate error messages and resume code generation in preference to fatally crashing. The errors that ASCGEN could encounter are detailed in Appendix A, which analyzes areas of weakness in the Prolog PDL semantics. In order to reduce the error handling requirements, we constrained the range of possible errors to handle. We assumed a CAEDE user would avoid making the specified errors in Appendix A until a fully implemented semantic checking tool was developed or another solution was devised.

CHAPTER 3 ASCGEN DESIGN AND IMPLEMENTATION

3.1 INTRODUCTION

This chapter discusses the issues and results of the Design and Implementation Phases of ASCGEN. The Implementation Phase bypassed the traditional "Detailed Design Phase" (associated with conventional language design methodologies) in order to take advantage of the power and flexibility of Prolog for both specification and implementation.

This chapter is organized as follows: Section 3.2 describes ASCGEN's sequence of operations within the context of the CAEDE system. Section 3.3 examines ASCGEN's architecture in detail. Subsection 3.3.1 provides an overview of the architecture. The next subsection (3.3.2) is devoted to a discussion of the three Phases around which ASCGEN was organized. The last architecture subsection (3.3.3) discusses some rejected design alternatives: incorporating a bi-directional translation capability and a DIANA intermediate representation into ASCGEN's architecture. The section on architecture is followed by Section 3.4 which contains a walk-through of the translation of a simple CAEDE-notation design structure into S-Ada. It recapitulates the previous concepts and illustrates some of the Prolog implementation strategies. The exposition is intended to be readable on its own. Section 3.5

describes the level of implementation of the ASCGEN prototype. This chapter is concluded with a summary of the results of the Design and Implementation Phases of ASCGEN (Section 5.6).

3.2 PERSPECTIVE OF ASCGEN WITHIN CAEDE

In this section, we present a system-level view of the process to translate a CAEDE-notation graphics design into Skeleton Ada. Our purpose is to identify the role of the CAEDE interfaces with ASCGEN, and to explain the system-level sequence of operations that was chosen for ASCGEN.

Translation of a previously entered CAEDE-notation design into S-Ada is fully automated. Code generation is initiated via a pick-menu option provided by the CAEDE human interface. When the code translation option is selected, the CAEDE system produces Prolog PDL files ("cg_fact_file" and "cg_info_fact_file"), corresponding to the current design, from the CAEDE "LOGIC STORE" and "PICTURE STORE" data bases. These data bases contain design information previously entered into the system in CAEDE-notation format. The file "cg_fact_file" contains Prolog facts which represent all aspects of the design except for supplementary textual comments, which are contained in the file "cg_info_fact_file". The CAEDE-to-ASCGEN software interface (defining the sequence described to this point), was an external constraint of this thesis. However, ASCGEN's algorithms are the subject of this thesis, and will be discussed in the following paragraphs.

The operation of ASCGEN consists of two major activities. The first activity is the initialization of the translator. Initialization consists of the invocation of the C-Prolog interpreter, and the consultation (or reading into the Prolog interpreter data base) of files containing facts and rules which characterize the ASCGEN translator. Figure 3.1 lists the various files consulted.

The second activity is the performance of the actual translation. The translation algorithm begins with the consultation of two files (`cg_fact_file` and `cg_info_fact_file`) comprising the Prolog design data base for the current CAEDE-notation design. When completed, the consultations are followed by a programmed command to translate the highest level CAEDE-notation structure, which encapsulates the design. The process of resolving this request performs the entire translation in a recursive fashion. The formatted S-Ada code produced is placed in the file "`cg_skeleton_code`" and a session log (which also lists any error messages), is placed in the file "`cg_output_file`". These and other related files would be located in the CAEDE design directory of the current design.

Figure 3.1 ASCGEN Translation Clause Files

FILE NAME	CLAUSES			FILE STATS		
	RULES	FACTS	TOTALS	CHARACTERS	WORDS	LINES
action_chains	10	1	11	6893	799	325
cg_comments	9	73	82	8043	618	248
decl_parts_3.9	18	0	18	8166	835	316
declarations_3.1	17	0	17	9538	848	298
format_stub	21	9	30	5888	651	233
generic_bdy	1	1	2	558	58	28
generic_decl	1	1	2	688	63	31
instances	44	7	51	28741	1958	876
pack_struct_7.1	4	0	4	3459	334	138
statements_5.1	27	4	31	11657	1121	431
subpgm_body_6.3	2	0	2	2843	288	77
subpgm_decl_6.1	19	3	22	4961	485	155
task_type_decl	1	1	2	851	88	36
task_type_bdy	1	1	2	553	58	29
tasks_9.1	4	0	4	3841	378	168
Totals	179	101	280	87872	8382	3381

3.3 ASCGEN ARCHITECTURE

3.3.1 OVERVIEW

We define the architecture of a program as the structure of that program in terms of its modules and submodules, and their interactions. The choice of architecture and its implementation were influenced by the types of algorithms required to perform the translation, and the modularity options available in C-Prolog. Each of these influences will be discussed in turn, in the following paragraphs.

The basic function of ASCGEN was to translate a CAEDE-notation design (defined by a recursive schema), into S-Ada (also defined by a recursive schema). The basic translation algorithm was to convert a CAEDE-notation construct into the corresponding S-Ada code fragment. This algorithm may be logically organized into 3 Phases, as shown in Figure 3.2:

- (a) Analysis Phase which analyzes a CAEDE-notation design,
- (b) Translation Phase which maps specific CAEDE-notation design constructs into S-Ada syntactic categories, and recursively into lower-level syntactic categories,

- (c) Code Generation Phase, which generates and formats the appropriate S-Ada code.

These phases provide a conceptual basis for ASCGEN's structure and translation algorithm, which are discussed in more detail in the next subsections.

We chose to use the S-Ada syntax as the basis for modularizing the structure of ASCGEN. In fact, the backbone of ASCGEN is an almost direct Prolog implementation of the S-Ada syntactic categories in the Transformation Phase. The following benefits resulted from this approach:

- (a) The recursive nature of the S-Ada specification could be directly implemented in the structure (and algorithms) of the Transformation Phase in a natural way.
- (b) A recursive framework was created within which to implement the three sequential phases.
- (c) The architecture was relatively stable because the target S-Ada specification (based on Ada syntactic categories) was stable.
- (d) The architecture promoted flexibility since it made provisions for incremental addition of code that implemented more S-Ada features.

Modularity of ASCGEN has been implemented in the following ways:

- (a) Logical and implementation modularity at the highest level has been achieved by partitioning predicates into files of related predicates (e.g. files of S-Ada syntactic category predicates organized by the corresponding section/chapter in the S-Ada Specification; refer to Figure 3.1). However, when the files are consulted by the Prolog interpreter during the initialization of ASCGEN, the files' modularizing effect is lost.
- (b) Prolog "procedures" implement functional modularity. Procedures are sets of Prolog facts and rules which share the same name and number of arguments. In addition to providing the only method for implementing true modularity in a C-Prolog program, procedures can implement the module and submodule concepts.

Figure 3.2 ASCGEN Logical Translation Phases

3.3.2 PHASE DETAILS

The basic algorithm selected for ASCGEN was to execute the Analysis, Transformation and Code Generation Phases in a sequential fashion when translating at an S-Ada syntactic category level, but within an overall recursive framework. For example, a CAEDE-notation "package" object will be sequentially translated in phases into an S-Ada "package" code fragment. But all objects internal to the "package" will in-turn be translated, in a recursive fashion, before the last tokens of S-Ada "package" code are emitted. The recursive nature of the CAEDE-notation designs and S-Ada specification made this linear-recursive algorithm a natural choice. Implementing the phases as structurally distinct algorithms was an alternative choice. The main reason to structure ASCGEN in this form would be to support bi-directional translation. This structuring option was dropped with the option for incorporating bi-directional translation. Details are provided in subsection "3.3.3.1 Bi-directional Translation".

In the following subsections, each of the logical phases of ASCGEN are discussed in more detail in terms of concepts, issues and implementation choices.

3.3.2.1 Analysis Phase

The Analysis Phase had two major functions to perform. The first function was to verify the consistency of the Prolog PDL representation of the CAEDE-notation design. The requirements analysis phase (previous chapter) identified the required checks to be made. However, the level of effort involved in implementing all of the checking was considered to be beyond the scope of this thesis. Therefore only some of the critical checks were included in ASCGEN with the assumption that all designs to be translated would conform to the consistency rules. The second function was to analyze the CAEDE-notation design in order to extract relevant information to be used by Transformation Phase predicates.

The significant result of the Analysis Phase is the extraction of information on the current CAEDE-notation object. This information may be obtained directly by searching for facts on an object or indirectly via other analysis goals which combine various facts to reach desired conclusions. A reference term is typically passed as a parameter to Transformation Phase predicates for use in obtaining further facts on the current CAEDE-notation structure. The "Item Descriptor List" (IDL) of a CAEDE-notation structure is used as this reference. The IDL contains information on the ancestry for the subject CAEDE-notation item. An example of an IDL is the first argument (i.e. {procedure, 'PROCEDURE_1', ['PACKAGE_TEMPLATE', 'MAIN_PACKAGE']}) of the component fact shown in Figure 3.5. The IDL acts as a composite data base key which can be used to obtain more facts on the current CAEDE-notation design structure, on the objects that own it and on those that it owns. It is an essential item to direct further recursive analysis of nested CAEDE-notation structures.

Another parameter passed is a stack of "environment" data structures which summarizes context information for the hierarchy of externals within which this CAEDE-notation object is nested. The environment external to this object is pushed onto the environment stack and the stack is passed as a parameter to goals that deal with the translation of its internals. This "environment stack" provides context information for generating code for CAEDE-notation objects and for calls on these objects.

An example use of the "environment" for generating code is the declaration of the internals of a CAEDE-notation "channel". Figure 4.2 (Chapter Four, ASCGEN Testing) shows a CAEDE-notation graphical test input that contains a channel with internals. The channel does not have a direct correspondence with any Ada object. All channels within the same CAEDE-notation parent (i.e. package or task) will end up being translated into S-Ada code within the sequence of statements of their parent. The internals of one channel will no longer be hidden from those of another because the virtual channel boundaries disappear. The "environment" term contains a record of the external view of each channel and these views are used to generate unique names for all internals.

The "environment" is also used for generating correct code for calls on CAEDE-notation objects. It assists in the tracing of a call from its source to its destination, and in the selection of the correct S-Ada name for the destination. Diagram 4.2 contains an example of a call interaction chain originating from within a package instance ("pkg_inst_1" located within channel "sing_chan_1") and targeted on a data area ("da_1" in the same channel). The tracing of a call is non-

trivial when the call interactions originate from within a template instance (as in our example). Within the instance, the source of the interaction is defined by facts on the internals of the template from which the instance was created. Outside of the instance, the interaction chain is defined by the externals of the instance itself. To do the analysis requires that the template/instance boundary be recognized and crossed correctly. The "environment" allows this transition to be made correctly because the external views were pushed onto the "environment stack" during the recursive analysis of a nested, CAEDE-notation design.

The selection of a correct S-Ada name for the target of an interaction chain is dependent on the environment of the target. The "environment stack" stores information on a target's environment permitting correct code to be generated.

3.3.2.2 Transformation Phase

The Transformation Phase converts the products of the Analysis Phase into inputs to the Code Generation Phase. It also implements the recursive features of ASCGEN's translation algorithm. The algorithmic behaviour is controlled by Transformation predicates designed to implement the S-Ada specification.

An example will illustrate the transformation algorithm. First, the highest level CAEDE-notation structure in a design is found, and an appropriate transformation rule applied to map it into an S-Ada syntactic category. The transformation rule will be chosen based on the

CAEDE-notation structure and related facts. The syntactic category clause will have body goals which perform further analysis, transformation or synthesis functions. If the current CAEDE-notation structure contains nested objects, then these objects will also be analyzed, and mapped into S-Ada syntactic categories.

A Transformation predicate implements an S-Ada specification as follows:

- (a) The predicate takes the name of the left-hand non-terminal symbol in an S-Ada production. If the S-Ada production has several alternatives, then there will be at least one predicate to implement each alternative. All of these predicates will have the same name. They will be distinguished from one another by their rule head parameters, and possibly by further body analysis goals included to ensure the correct alternative is selected;

*/

- (b) The rule body corresponds to one of the corresponding S-Ada alternatives defined by the right hand side. Recursion is supported by body goals that implement S-Ada non-terminals. Code Generation is supported by body goals corresponding to S-Ada terminal symbols. Analysis goals are also included to support any Transformation or Code Generation goals.

The S-Ada specification for the procedure alternative of "S_subprogram_specification" (Figure 3.6) and the corresponding Transformation predicate "subprogram_specification" (Figure 3.8) shows the close correlation between the specification and the Prolog implementation of a Transformation predicate. Note that a "function

subprogram" was not supported by CAEDE so only the "procedure" alternative of "S_subprogram_specification" was coded in Prolog.

The Prolog clauses that transform task, package and channel templates and instances into S-Ada were the most complex to implement. The interpretation of the code to generate for a template was assumed to depend on the context. We made the assumption that a template could conceivably map into "N" generics or task types, so considerable context testing code was needed. The channel has no corresponding structure in Ada, so it is transformed into a variety of pieces that are also context dependent. Figure 3.1 indicates more than 50 predicates (in ASCGEN files: "instances", "task_type_+", "generic_+") were developed to handle template and instance transformations. They were also the most difficult to code.

An alternative method considered for performing transformations, was to map between Prolog terms that implemented trees. The transformation would then involve a CAEDE-notation structure tree to S-Ada tree mapping. Experience with implementation showed that the process of synthesizing a CAEDE-notation structure, and transforming it into an S-Ada tree would progressively consume the Prolog stack resources. In an N-pass translator, the stack could grow with each pass, unless provisions were made to save and restore the translation products between passes in order to clear the stacks. This alternative was dropped in favour of the chosen method for several reasons. The tree-based approach was felt to be more complex and require more storage overhead, and would suffer a loss in speed as a result of the associated save and restore processing to control the storage requirements. The issues are more fully discussed in subsection "3.3.3.1 Bi-directional

Translation".

3.3.2.3 Code Generation Phase

The Code Generation Phase produces S-Ada code and comments as directed by Transformation Phase clauses that contain goals which produce output. This phase also manages the formatting of S-Ada terminals and comments.

The code and comment format conforms to that specified in the S-Ada Specification. Comments input during the CAEDE design phase (contained in the information facts for the design), are formatted within headers bordered with ascii characters chosen to match the associated CAEDE-notation structure (the comments for different icons have different borders). Other comments are automatically generated to guide an implementor in the completion of the S-Ada skeleton code.

The version of C-Prolog used for ASCGEN did not support complex string operations with built-in predicates. Page oriented operations (i.e. operations to do text filling) had to be programmed in a recursive fashion in general, with a great loss in efficiency. Consequently, the formatting operations required careful re-implementation and optimization in order to obtain reasonable performance. Details of these efforts can be found in "Chapter 4 ASCGEN Testing".

Code Generation is a non-reversible process in the current implementation of ASCGEN. A reversible alternative considered was to build an S-Ada code tree to be passed to a formatter for final code output. This approach would confine irreversibility to the final output phase. However, the alternative was rejected due to unnecessary, additional complexity, increased stack requirements and anticipated loss in performance.

3.3.3 REJECTED DESIGN ALTERNATIVES

3.3.3.1 Bi-Directional Translation

When designing ASCGEN, the feasibility of incorporating reversibility in the translation clauses was examined. It was assumed that this would be a desirable feature based on other related work in the CAEDE project directed towards automatically converting Ada programs into CAEDE-notation graphic descriptions. The major benefit of the reversibility feature would be reusability of the same translation clauses for translating from CAEDE-notation design facts into S-Ada or from S-Ada into CAEDE-notation facts. In addition, it was felt that the validation of translation in both directions could be achieved by validating one direction. It was concluded that implementing reversibility was not feasible within the scope of this thesis. The following paragraphs discuss the major problems.

Some of the clauses that implement the Analysis and Code Generation Phases are inherently irreversible because they involve irreversible operations, as explained in the following subparagraphs:

- (a) The Analysis Phase involves input of design facts as the task progresses. This interaction takes the form of resolution of design fact goals in some of the syntax analysis clauses. The flow of information is FROM the data base. The reverse process of CAEDE-notation fact synthesis cannot be accomplished by the resolution mechanism. Putting data base facts into the data base is only possible by ASSERTING facts. This is accomplished by a specific clause. Consequently, different clauses would be required for ANALYSIS and SYNTHESIS of CAEDE-notation facts.
- (b) Code generation involves writing characters to external streams. The reverse operation is reading from an external stream. Both I/O operations are irreversible in C-Prolog. Therefore, different clauses would be required for each of these operations.

The Transformation Phase (CAEDE-notation to S-Ada) could be designed for bi-directional execution provided that none of the irreversible clauses identified above are involved with the input and output data to be transformed. This requires that irreversibility be confined to two sets of clauses: one set for the data base or external data stream analysis functions and another set for the data base fact synthesis or external data stream synthesis phases. A further requirement is that input and output data be passed as Prolog terms between phases!

The practicality of incorporating a bi-directional translation capability into ASCGEN's predicates was limited by the following factors:

(a) Efficiency

In order to allow reversibility within a translation phase, all decisions made during one direction of execution of that phase must be able to be undone. This is a necessary requirement of a Prolog implementation to ensure the translation process is reversible. A negative consequence of reversibility is that a steadily increasing storage burden will be imposed as a translation progresses. The load consists of stack space, goal markers, variable instantiations and other storage representing the execution state reached. Performing reversible translation is like digging a hole with the following restriction: the hole cannot grow larger than the capacity of the bucket used to hold the dirt. In each case, the job has to be completed within the limits of available resources.

(b) Design Effort

Each sub-tree parameter requires a formal definition. All data to be transformed or required to guide the translation must be initially stored in the origin sub-tree data structures. The existing facts in the Prolog data base cannot be used nor can the flexible Prolog built-in predicates be used to access this data. Instead, a new set of predicates must be defined for operating on the sub-tree structure.

(c) Stability

The CAEDE Prolog PDL representation was subject to change. Changes would affect at least some of the intermediate representations and their access algorithms.

3.3.3.2 DIANA as an Intermediate Representation

Butler [Butler 84] described the DIANA intermediate representation for use within Ada Program Support Environments. The DIANA representation was intended to store an Ada program's syntax and semantic information in an easily accessible form, for use by other Ada development tools.

DIANA was examined for possible implementation as a Prolog intermediate representation, to be passed between phases of ASCGEN, in support of reversible translation. It was rejected for several reasons. Much of the information to be included in a DIANA tree is not producible from a CAEDE-notation design. An intermediate representation would also introduce a redundant representation of design data, already present in the Prolog PDL design representation, and require operators, as well as a translation step to convert the existing facts into the intermediate representation. The need for an intermediate representation disappeared when the requirement for supporting bi-directional translation was dropped.

3.4 TRANSLATION WALK-THROUGH

This section discusses the translation of a simple design example in order to demonstrate features of ASCGEN's architecture and translation algorithms. The design example to be translated (Figure 3.3) is a fragment extracted from the "ASCGEN DEMO" test design that is fully documented in Figure 4.1 and Appendix D. Enough information has been abstracted to provide a tutorial which is fully contained within this section.

The CAEDE-notation design of Figure 3.3 consists of a mixed graphic and textual portion and separate information facts. This design has the following components:

- (a) "invisible procedure" icon: "PROCEDURE_1" (i.e. a procedure declared within a package/task), and
- (b) a single "actual parameter" icon, of mode "in" : "DV21" (icon arrow is oriented along the iteration towards the target). The code to be generated from this design fragment will not depend on the interaction icon which is logically part of the icon from which the interaction originates. The separate information component consists of a textual comment: "P1: TEST COMMENT FOR INVISIBLE PROCEDURE", entered via the CAEDE Interactive interface, and stored separately from the graphics data base.

Each request for Skeleton-Ada code generation involves the creation and consultation of a Design data base of Prolog facts, implementing this schema, as input to ASCGEN.

Figure 3.3 CAEDE-notation Procedure Design for Translation

Text entered via the CAEDE human interface:
 "PR1: TEST COMMENT FOR INVISIBLE PROCEDURE"

3.4.1 ASCGEN INPUTS

The inputs to ASCGEN are transformations of the primary CAEDE data bases, expressed as Prolog facts. They do not contain physical graphical information, but only the essential logical design content of the primary CAEDE Design data bases.

Figure 3.4 contains the Prolog PDL schema for "invisible procedures" and "supplementary information", which define the facts relevant to the chosen CAEDE-notation design example. An important feature common to the schema is the "item descriptor list" (IDL) which acts as a composite data base key. An instance of the same schema, for example, a procedure fact, is distinguished from other facts within the same parent icon by CAEDE-notation-type and name, and by a "path identifier list" (a nesting hierarchy list), from all other objects of the same name declared in different contexts. We make use of the composite nature of the key to locally analyze and synthesize other keys to obtain facts about peers, children or ancestor objects when needed in a particular rule.

Figure 3.5 contains Prolog facts on the CAEDE-notation design. These facts are instances of the schema defined in Figure 3.4. The "item descriptor list" indicates this procedure is declared within "PACKAGE_TEMPLATE" which is declared within "main_package". Although "PACKAGE_TEMPLATE" will translate into an Ada generic package, it has no direct effect on the form of code generated for "PROCEDURE_1" since this procedure makes no calls outside of its parent.

3.4.2 ASCGEN OUTPUTS

The form of target code is defined by the S-Ada Specification, Appendix B, developed as part of this thesis. Figure 3.6 reproduces the portions of this specification relevant to the design example. Figure 3.7 contains the actual S-Ada code produced for the CAEDE-notation design. The code contains instances of the specifications shown in Figure 3.6.

Figure 3.4 Prolog Fact Schemas Relevant to Figure 3.3 Design

```

/*-----component([procedure,symbol,[...]],list)      */
component(
    [                                /* item-descriptor-list */
    procedure,                      /* SDWA type */
    (<id_ident>),                   /* procedure identifier */
    [...],                         /* item descriptor list (idl-list) */
    ],
    [                                /* component details */
    [                                /* parameter list */
    [<data_ident>,<data_node>,<data_type>], /* single parameter spec */
    ... more ...
    ]
    ]
).

/*-----information(item_descriptor_list,information_list) */
information(
    item-description-list,
    [                                /* information-list */
    line-of-text-atom,
    ...
    ]
).

```

Figure 3.5 Prolog Facts for Figure 3.3 Design

```

component(
  [procedure,'PROCEDURE_1',[ 'PACKAGE_TEMPLATE','main_package']],
  [
    [
      ['DV21','in','DATA_TYPE_21']
    ]
  ]
).

information(
  [procedure,'PROCEDURE_1',[ 'PACKAGE_TEMPLATE','main_package']],
  [
    'PRI: TEST COMMENT FOR INVISIBLE PROCEDURE',
    ''
  ]
).

```

Figure 3.6 S-Ada Schema Related to Figure 3.3 Design

-- 6.1 SKELETON SUBPROGRAM DECLARATIONS

S_subprogram_specification ::=

 procedure identifier [S_formal_part]
 ; function designator [S_formal_part]
 return type_mark

S_formal_part ::=

 (S_parameter_specification
 (; S_parameter_specification)
)

S_parameter_specification ::=

 *formal_parameter*_identifier : mode type_mark
 [:= (expression)]

mode ::= [in] ; in out ; out

Figure 3.6 S-Ada Schema (continued)

-- 6.3 SKELETON SUBPROGRAM BODIES

```

S_subprogram_body ::=

    -- CAEDE-notation does not currently support functions.

    S_HEADER.CG.CODE(
        ICON      => pack_ext_procedure -- no code strip
                  or procedure -- code strip
        CONTEXT   => Ada body code of parent
    )
    S_subprogram_specification is
        S_declarative_part
    begin
        S_sequence_of_statements
    [exception
        {
            (exception_handler)
            ((exception_handler)) ]
    end [ designator ] ;

```

-- 6.4 SKELETON SUBPROGRAM CALLS

```

S_formal_parameter ::= "parameter"_simple_name

```

Figure 3.7 Target S-Ada produced for Figure 3.3 Design

```
--  
--Spec. body for procedure: PROCEDURE_1  
--  
--  PR1: TEST COMMENT FOR INVISIBLE PROCEDURE  
--  
--  
procedure PROCEDURE_1( DV21 : in DATA_TYPE_21) is  
  -- make necessary declarations  
begin  
  -- FINISH body code  
end PROCEDURE_1 ;
```


Figure 3.8 Procedure Transformation Prolog Clauses

```

1: subprogram_body([procedure, NAME : IDL_TAIL ], PREFIX, ENV_STACK):-
2:   get_unique_key(KEY),
3:   parse_action_chains(
4:     KEY,
5:     [procedure, NAME : IDL_TAIL ],
6:     ENV_STACK
7:   ),
8:   cg_comments(major_header, procedure_body, [ procedure, NAME :
9:     IDL_TAIL ], PREFIX ),
10:  subprogram_specification([procedure, NAME : IDL_TAIL ], PREFIX),
11:  write_atom(' is'),newline,
12:  begin_block,
13:  write_atom('-- make necessary declarations'),newline,
14:  end_block,
15:  write_atom(begin),newline,
16:  begin_block,
17:  write_atom('-- FINISH body code'),newline,
18:  (sequence_of_statements(
19:    KEY,
20:    [procedure, NAME : IDL_TAIL],
21:    ENV_STACK)
22:  ; true ).
23:  end_block.
24:  write_atom('end '), write_atom(PREFIX), write_atom(NAME),write_atom(' ');
25:  newline, !.
26:
27: subprogram_specification([PROCEDURE_KIND,NAME : IDL_TAIL], PREFIX):-
28:   /* check that this is not a function specification */
29:   ((PROCEDURE_KIND = procedure) ; (PROCEDURE_KIND = pack_ext_procedure)),
30:   write_atom('procedure '), print(PREFIX), print(NAME),
31:   component([PROCEDURE_KIND,NAME : IDL_TAIL],
32:     [FORMAL_PARAMETER_SPEC_LISTS]),
33:   formal_part(FORMAL_PARAMETER_SPEC_LISTS).
34:
35:
36: formal_part([]):- !. /* no formal parameters implied by an empty list.*/
37: formal_part(PARAMETER_SPEC_LISTS):-
38:   write_atom('('),
39:   formal_parameter_spec(PARAMETER_SPEC_LISTS),
40:   write_atom(')').
41:
42:
43: formal_parameter_spec([(NAME,MODE,TYPE) : PARAMETER_SPEC_LISTS_TAIL]):-
44:   write_atom(' '),print(NAME),write_atom(' : '),print(MODE),write_atom(' '),

```

Figure 3.8 Procedure Transformation Prolog Clauses (continued)

```
45:    print(TYPE),
46:    next_formal_parameter_spec(PARAMETER_SPEC_LISTS_TAIL).
47:
48:
49: next_formal_parameter_spec([]). /* no more parameters termination condition */
50: next_formal_parameter_spec([NAME,MODE,TYPE] : PARAMETER_SPEC_LISTS_TAIL):-
51:    write_aton('; '),
52:    (text_node(temporary_comment) -> newline ; true),
53:    print(NAME),write_aton(' : '),print(MODE),write_aton(' '),
54:    print(TYPE), write_aton(' '),
55:    next_formal_parameter_spec(PARAMETER_SPEC_LISTS_TAIL).
```

3.4.3 ASCGEN PROCESSING

We will describe the head and body of the Prolog transformation rule "subprogram_body" (Figure 3.8) in order to highlight the structure and algorithms of ASCGEN. This rule is a typical example of those that implement the Transformation Phase in ASCGEN. Not shown are the rules which call these rules and some of the goals of the chosen rules.

The S-Ada "subprogram_body" specification (Figure 3.6) defines the transformation production corresponding to the example CAEDE-notation design. The implementation of this specification is the transformation rule "subprogram_body" contained in Figure 3.8. The rule head has several formal parameters to be matched against. The first parameter is an Item-Descriptor-List (IDL). The formal parameter specifies that for a match to occur, the IDL must be for a procedure. Only as much information as is needed for explicit use by this rule, is selected from the IDL with "IDL_TAIL" instantiated to the rest. The second parameter, "PREFIX" will be instantiated to a prefix string if there is the possibility that a naming conflict could occur with other CAEDE-notation objects. The third parameter, "ENV_STACK", is a stack of "environment" data structures, which provide context information to assist further analysis and synthesis. The IDL, "prefix" and "environment stack" have been discussed in more detail in a previous subsection (3.3.2.1 Analysis Phase).

Significant goals in the body of rule "subprogram_body" will now be examined.

- (a) Goal "get_unique_key" (line 2), obtains a globally unique key for use in the goal which follows it.
- (b) Goal "parse_action_chains" (line 3), synthesizes a code generation fact for each action chain sourced by procedure "NAME". We define an "action-chain" to be the source to destination sequence of 1 or more primitive interaction objects (i.e. the CAEDE-notation arrow joining a source and destination, composed of 1 interaction or several connected interaction objects). For this example, there are no interactions to be generated.
- (c) Goal "cg_comments" (line 8) produces a distinctive comment header for this procedure type object. This header consists of a border of characters unique to the CAEDE-notation graphic icon type (in this case the character '=' for procedure icons) and formatted comments if any were entered for this particular icon.
- (d) Goal "subprogram_specification" (line 10) calls the rule on line 27.
- (e) Goal "component" (line 31) performs a fact data base query to obtain information about procedure "PROCEDURE_1", using the IDL key. The schema that defines this fact is shown in Figure 3.4. The actual fact that this goal will match is shown in Figure 3.5.
- (f) Goal "formal_part" (line 33) generates code for any formal parameters that exist. There is a close correspondence between the "formal_part" rule and rules which it calls (listed after line 34), and the S-Ada specifications which they implement. These rules

actually provide an example of how an extended BNF production was implemented in Prolog.

Returning to rule "subprogram_body" (line 12):

- (a). Goal "begin_block" implements a margin tab function.
- (b). Of final note is the goal "sequence_of_statements" (line 18), which generates code from action-chain facts previously produced by rule "parse_action_chains" (line 3). For this example there are no action chains present.

3.5 LEVEL OF IMPLEMENTATION

ASCGEN was implemented to a level that allowed all features implied by the Prolog PDL definition to be translated. A small number of features planned for ASCGEN were not implemented, because the effort to implement them would not directly contribute to the research aims of this thesis. The following paragraphs detail the significant unimplemented features.

The requirements definition activity produced the requirements for a semantic checker to ensure that the Prolog facts on the CAEDE-notation design are consistent (Appendix A). This validation functionality is really part of the Analysis Phase of ASCGEN. However, the checker was not implemented since it was felt to be a separable program and not essential to the development of a prototype. CAEDE-notation designs were instead manually enforced to conform to consistency requirements.

The S-Ada specification defined syntactic category pseudo-terminals to be generated as a substitute for code that could not be generated automatically. However, the code to generate these terminals was not implemented since it could be added at a later date.

The version of the CAEDE-notation supported by CAEDE didn't include a paradigm for exceptions, so exception code generation was not supported.

3.6 SUMMARY

Significant details of design and implementation were exposed in an evaluation of the prototype, and by means of a CAEDE-notation design translation walk-through. The organization of the details in this summary mirrors that of the preceding sections.

ASCGEN was organized and implemented around the following phases:

- * Analysis: syntactic and semantic analysis of the CAEDE-notation structures;
- * Transformation: mapping of CAEDE-notation structures into relevant S-Ada syntactic categories;
- * Code Generation: from S-Ada syntactic categories.

The Three translation phases of ASCGEN were examined in detail. Inputs, outputs and implemented functionality were discussed (refer to the associated sections for details). Phases were executed sequentially, but within an overall recursive framework in the actual implementation of ASCGEN. This was felt to be the most natural algorithm once the requirement for reversibility in translation was dropped. A feasibility analysis on implementing a bi-directional translation capability and a DIANA intermediate representation into ASCGEN showed their disadvantages exceeded their benefits. The coding of the CAEDE-notation template transformation capability (Transformation Phase) was found to be the most difficult to implement.

A translation walk-through, for a simple CAEDE-notation design fragment, illustrated previously-discussed concepts in the design and implementation of ASCGEN.

The implementation of ASCGEN was very complete; any CAEDE-notation design could be translated into S-Ada code. Only a small number of non-critical features were left unimplemented.

CHAPTER 4 ASCGEN TESTING

4.1 INTRODUCTION

Testing of ASCGEN involved both Validation and Performance evaluation. Validation Testing involved Incremental Testing as the prototype was evolving, and a final series of Functional Tests on the completed prototype. One Functional Test (to evaluate the completeness and correctness of the code ASCGEN produced) consisted of adding code to an S-Ada program to make it compile. Performance Testing was used both to direct efforts aimed at improving the resource consumption of ASCGEN and to benchmark the performance of the final version of ASCGEN.

The "Validation Testing" section examines validation testing in more detail. The "Improvement History" section traces the performance evolution of ASCGEN, and the various modifications that supported it. The "Performance Evaluation" section investigates the performance of the final version of ASCGEN, when translating various inputs. The chapter is concluded with a summary.

4.2 VALIDATION TESTING

ASCGEN was validated in two phases: Incremental Testing of the prototype throughout the development period, and Functional Testing of the completed prototype. Incremental Testing supported the iterative development plan of ASCGEN. As features were implemented, test cases were designed and input to confirm that the associated analysis, transformation, and code synthesis were correct. There were quite a number of simple tests developed, each testing a particular feature. The debugging features of the C-Prolog interpreter were utilized to trace execution and debug predicates which did not generate human readable output.

Functional Testing was completed in two steps. The first step involved confirming that CAEDE-notation inputs were correctly translated. This confirmation was implemented visually by comparing the code produced with the input designs, and the appropriate specifications. The second step involved completing the code produced by ASCGEN from one of the test inputs to make it compilable.

The first step was accomplished with a set of test cases which collectively validated all required features. Appendix C highlights the functional test design specification to validate ASCGEN. It lists features to test organized by CAEDE-notation syntactic category. These features were derived from the Prolog PDL specification [Buhr 85A] and the S-Ada target specification (Appendix B). Figures 4.1 - 4.3 contain graphical CAEDE-notation test inputs for confirming the majority of features listed in Appendix C.

Figure 4.1 (an extension of an original design by G. M. Karam, CAEDE project), contains a test input for non-channel CAEDE-notation features. Appendix D contains the actual target S-Ada corresponding to the test input in figure 4.1. Figure 4.2, and 4.3 contain test inputs for validating the translation of single channel features, and channel instance features respectively. Appendices F and G contain the target S-Ada generated for the test inputs of figures 4.2 and 4.3 respectively.

The second step involved making the code produced from the design in Figure 4.1 (ASCGEN-DEMO) compilable. This test was performed in order to validate productions in the S-Ada specification and to obtain a measurement on the correctness and completeness of the code generated by ASCGEN. Appendix E contains a listing produced by the UNIX "diff" command. The listing shows the lines where tokens of code were added in order to allow successful compilation (using a validated Digital Equipment Corporation VAX/Ada compiler). All code modifications consisted of additions of code (30% increase in tokens of executable code; 75% of the additional tokens were for declarations). The success of the compilation effort showed that the code produced by ASCGEN for the ASCGEN-DEMO design was reasonably complete, and showed that both the syntax fragments produced and the ordering dependencies were entirely correct. The relatively high percentage of declaration tokens needed for compilability indicates that adding CAEDE support for declaration generation could make significant contributions to increasing the level of compilability in automatically generated code.

In both the Incremental and Validation Phases of testing, the S-Ada code was validated against the S-Ada specification, to ensure correctness. In particular, the Functional Testing phase verified that

the integrated version of ASCGEN correctly translated the range of CAEDE-notation designs implied by the input Prolog PDL specification.

Figure 4.1 "ASCGEN-DEMO" CAEDE-notation Design

directory:version - ascggen_demo:3

Figure 4.1 "ASCGEN-DEMO" CAEDE-notation Design (continued)

```
directory:version - ascgen_demo:3
```

Figure 4.1 "ASCGEN-DEMO" CAEDE-notation Design (continued)

TASK_TEMP

TASK_TEMPPL

PACKAGE_TE

Figure 4.2 "SINGLE CHANNEL" CAEDE-notation Design

directory version - chan_tst1 2x3

Figure 4.2 "SINGLE CHANNEL" CAEDE-notation Design (continued)

directory version - chan_tst1 2x 6

Figure 4.2 "SINGLE CHANNEL" CAEDE-notation Design (continued)

Figure 4.3 "CHANNEL INSTANCE" CAEDE-notation Design

Figure 4.3 "CHANNEL INSTANCE" CAEDE-notation Design (continued)

directory version - chan_inst1 2x.5

4.3 IMPROVEMENT HISTORY

In this section, we report on the increases in the performance of ASCGEN as a result of a series of tests and corrective design modifications. Trial use of early prototypes of ASCGEN revealed an inability to handle large designs. Since ASCGEN was still under development at the time, it was essential to determine what the problems were, and to correct them, in order to take advantage of the planned enhancements.

The "Remote Mailbox System" (RMS) design was chosen to benchmark ASCGEN performance, after each design modification. The details of RMS can be found in [Pearce 86]. RMS was adopted as a performance test case because:

- * it was available early in the performance testing cycle;
- * it was a large design. RMS was the first design that was even too large to translate with ASCGEN running as a stand-alone tool on a VAX, configured for five megabytes maximum virtual space. ASCGEN was originally intended to run co-resident with CAEDE on a SUN workstation, with about 1.5 megabytes maximum virtual space dedicated to ASCGEN;
- * it exercised enough ASCGEN features to make a study worthwhile.

The relevant information for the RMS test case is:

Design Name: RMS (Remote Mail System);
cg_fact_file: 1465 lines, 28K chars, 188 facts;
Test Environment: VAX/11-750, BSD/4.2 Unix, C-Prolog/1.4.

The test approach involved benchmarking the performance using the same CAEDE-notation design for test input, throughout the period of ASCGEN development and refinement. Figure 4.4 summarizes the improvement history statistics. The "Test Numbers" column defines specific test configurations, which will be elaborated on in the following discussion. The code modifications that preceded each test are discussed in general. Listings for the files that constitute ASCGEN are provided in Appendix H. Each step was executed several times in order to obtain a typical value for the metric being evaluated.

Figure 4.4 ASCGEN Performance Improvements Summary

TEST FOR STEP NO. (1)	SKELETON ADA CODE STATS (2)		PROLOG STACK STATISTICS (3)		CPU TIME (4)		
	CODE SIZE	GENERATION RATE (lines/min)	G: Global L: Local Assigned (Used)	RELATIVE TO TEST 3 (% Used)	INIT (sec)	XLAT (sec)	RELATIVE TO TEST 3 TOTAL (%)
1.	-	-	G: 2M L: 2M	OVERFLOW!	-	-	-
2.	8	-	G: 2M (26K) L: 2M (84K)	8.4 9.2	32	10	8.6
3.	49K 1247 lines	153	G: 2M (389K) L: 2M (913K)	100 100	32	456	100
4.	49K 1247 lines	156	G: 2M (21K) L: 2M (67K)	6.8 7.3	32	446	98
5.	49K 1247 lines	260	G: 1M (21K) L: 1M (67K)	6.8 7.3	32	255	59
6.	49K 1247 lines	322	G: 1M (21K) L: 1M (44K)	6.8 4.8	32	200	48
7.	27K 591 lines	724	G: 256K (40K) L: 128K (0.6K)	4.4 0.1	15	34	10

NOTES:

- [1] These tests benchmark performance improvements described in section "IMPROVEMENTS HISTORY".
- [2] Codegeneration statistics are only a general indicator of performance in this history log. The amount of whitespace and newlines was higher in the code produced by the version of ASCGEN in test 3 than in test 7 (28K versus 8K), giving a misleading impression. The most significant indicator for this case is the relative CPU time comparison since the code output is constant for these tests. Note also that GENERATION RATE = Lines/Total CPU time.
- [3] The Prolog Machine contains more dynamic data areas than the local and global stacks. The stacks are shown because they are the most significant run-time space users. Note that the term assigned refers to the static space "allocated" to the Prolog Interpreter. One may conclude that the runtime dynamic size never exceeded these limits when translation was successful. The term "used" only refers to the minimum stack not freed when code generation was completed. For a version of ASCGEN incorporating space recovery functionality, the runtime stack position will vary between the "Used" and "Allocated" limits.
- [4] The initialization time (INIT) includes the time for processing both ASCGEN predicates and the design facts. The translation time (XLAT) is the time to translate following INIT.

The details of each test are as follows:

Step 1

The results record the starting point for the improvement history comparison. The results show ASCGEN initially overflowed the virtual space available during translation. Attempts to reallocate the virtual space among the Prolog machine stacks could not overcome the problem (The overflow inevitably shifted to another stack). In order to isolate the problem, it was desirable to isolate the ASCGEN phase which was causing the error. The possible logical divisions were the Analysis Phase, the Translation Phase, and the Code Generation Phase. Since ASCGEN had been designed with these phases recursively executed, it was nearly impossible, in general to test phases in isolation. However, the translation of an arbitrary design could be traced interactively allowing the Prolog interpreter state to be measured where desired. The point at which the failure occurred during the translation of the RMS design, was determined, and then the interpreter states leading up to the failure were sampled. Although the information obtained was inconclusive, in that there was no obvious single predicate isolated, it did indicate the state of the various Prolog machine stacks leading up to the failure. This knowledge was applied to reconfiguring space allocated to the local and global stacks in an Ad Hoc manner. Unfortunately, this did not overcome the problem (The overflow inevitably shifted to another stack). Interactive tracing of the execution indicated the detonation point was shifting a great deal as well.

Step 2

When the formatting portion of the code synthesis phase was omitted in Step 2, the stack overflow error was eliminated, with an order of magnitude reduction in execution time. This test indicated a large proportion of time (> 90%) and space (greater than four megabytes) were consumed during the Code Generation Phase. To observe the trend in space consumption, the RMS test case was executed interactively using the Prolog debugger, in order to measure statistics at selected ASCGEN goals. Some simple test cases were also written to confirm the behavior of C-Prolog under varying search strategies. It was discovered that C-Prolog/1.4 would not release space held unless backtracking occurred undoing assignments for the current solution. Designing predicates with cuts to fix a solution at the current level or at several levels subordinate to the current predicate, had no effect on the recovery of space, unless backtracking occurred. This surprising result exposed a fundamental constraint of C-Prolog/1.4. We do not imply that this is a general limitation of all Prolog implementations, since there were no other implementations of Prolog available to experiment with. This constraint had to be accommodated in the further design of ASCGEN.

Step 3

Step 3 involved redesign of critical format synthesis predicates to internally backtrack to free space, while still satisfying the calling goal. This step was successful since it permitted a full translation of RMS to be achieved. The results of this step have

been used as the reference point for comparing the modifications which followed it.

Step 4

Step 4 involved further detailed analysis of space usage and a comprehensive and exhaustive application of the techniques pioneered in Step 3. The effort was extremely successful (Space was reduced to under 10% of that required for Step 3).

Step 5

The effort of Step 5 was directed towards improving the time performance. Further interactive testing revealed predicates which could be made more efficient. For example, some recursive character output operations were very inefficient. Since C-Prolog lacked built-in predicates for powerful string I/O operations, the needed functionality had been implemented in Prolog predicates which are not as efficient. Inefficient predicates were isolated and hand-optimized, resulting in a significant time reduction (time reduced to 59% of that required for Step 3).

Step 6

Step 6 involved further efforts to develop more efficient local algorithms, but with reduced effect.

Step 7

Based on experience gained, efforts were directed to making global, rather than local algorithmic changes to ASCGEN in order to overcome the inherent performance constraints. ASCGEN was modified to change the style of comment blocks, the algorithm for generating output lines, and the form of the ASCGEN translation rules. ASCGEN was preconsulted as well to speed the initialization phase. The complete version of ASCGEN, which contains significantly more functionality, was tested in this step. The test results show a further, overall improvement in time and space usage. (10% of the time, 5% of the Global Stack and 0.1% of the Local Stack required for Step 3).

4.4 PERFORMANCE EVALUATION

The translation performance, of the final version of the ASCGEN prototype, was characterized in several performance tests. The test inputs were the RMS design and the previously described functional test inputs (Figures 4.1, 4.2 and 4.3), translated with C-Prolog/1.4 on a Digital Equipment Corporation VAX/11-750 hosting BSD/4.2 Unix. Figure 4.5 summarizes the significant features of these designs, the translation times, and some performance metrics for comparing the performance for each input, in the following discussion. Unless otherwise stated, any specific data mentioned in the following discussion, may be found in Figure 4.5.

CAEIDE-NOT.	INPUT	OUTPUT STATISTICS (1)		TRANSLATION	OUTPUT RATE		TRANSLATION FACTORS	
NAME	FACTS	LINES	CHARS	WORDS	TIME (sec) (2)	lines/m	Kb/m (3)	n/Kfact (4)
								1/fact (5)
RMS	188	591	27576	1624	49	724	33.8	4.3
ascgen_demo	140	456	21449	1477	38.9	703	33.1	4.6
chan_1stk.2x	70	211	8916	511	13.8	673	28.5	4.5
chan_inst1.2x	58	133	5310	300	18.8	578	23.1	4.0
								2.29

NOTES:

- (1) Output statistics for file cg_skeleton_file containing S-Ada code.
- (2) Translation time is time from ASCGEN startup until termination (includes initialization time).
- (3) Rate refers to code output / translation time (kilobytes/minute).
- (4) Ratio compares translation time / total design facts (minutes/ 1000 facts).
- (5) Ratio compares lines of S-Ada code per SDWA design fact.

Figure 4.5 ASCGEN Performance Statistics Summary

The final performance testing had the following objectives:

- (a) To obtain some metrics on the speed of translation for comparison to other types of source translators;
- (b) To determine how different complexities of CAEDE-notation design inputs affected the speed of translation;
- (c) To determine if any relationship existed between the input and output parameters and the translation time. Desirable results would be some formulae for predicting the size of target code and the translation time, based on parameters of the input design.

ASCGEN typically produced S-Ada source, including comments, at a rate of approximately 700 lines per minute, dropping to 600 lines per minute with more complex CAEDE-notation input designs. The issue of how fast the translation rate is may be addressed in several ways. ASCGEN/CAEDE cannot currently be used in the code-compile-modify and recompile development cycle (i.e. the implementation phase of the software development cycle). ASCGEN is instead intended for generating a code skeleton for input into this cycle, once the CAEDE-notation design is completed. ASCGEN really only needs to be used once/design. We feel ASSCGEN's speed is more than adequate for this purpose. However, if incremental completion of a code skeleton is supported by ASCGEN/CAEDE in the future, this would put ASCGEN in the development loop, so fast translation would be desirable. The ASCGEN code generation rate is comparable to the rate of translation of source Ada for current compilers. In its current form, we feel ASCGEN's code generation rate to also be adequate for such use.

The results indicate a performance reduction when translating complex inputs. The CAEDE-notation designs are listed in increasing order of complexity from top to bottom in Figure 4.5. We define "complexity measure" to mean "the level of abstraction of CAEDE-notation structures, rather than the number of structures in a design". For example, a procedure is not complex because it has a 1:1 correspondence with an Ada construct. However, a channel instance will be complex because it is one of N instances of a channel template. It must be uniquely transformed into Ada primitives based on both its contents and context, which might place it amongst other instances of the same template. A speed reduction was anticipated when translating complex CAEDE-notation designs, based on the internal algorithms of ASCGEN. This trend is evident in the reduced output rate (lines/minute) as design complexity increases. However, the difference is not that great.

Some Translation Factors were derived from the translation results. They seemed independent of the type of design being translated. The time to translate a design was found to be relatively constant between 4.0 to 4.6 minutes per thousand facts input. The number of lines of code was found to be relatively constant, approximately 3 lines/fact, dropping slightly for more complex designs. Although these factors have not been exhaustively confirmed, we feel confident in using them for rule-of-thumb estimating.

4.5 SUMMARY

Validation Testing consisted of Incremental Testing during the development phase and Functional Testing when ASCGEN was completed. The Functional Testing was accomplished with a set of test cases, (Figures 4.1, 4.2 and 4.3 constituting the most significant ones), which collectively tested all required features. The skeleton code (S-Ada) generated was successfully validated against the S-Ada specification. One of the test inputs was made compilable to verify the correctness of the S-Ada specification itself and to measure the relative completeness of the code produced by ASCGEN. Compilability was accomplished through the addition of 30% more executable code tokens (75% of the tokens were declarations). The relatively high percentage of declaration tokens needed for compilability indicates that adding CAEDE support for the automatic generation of declarations could significantly increase the level of compilability. The success of the compilation effort showed that the code produced by ASCGEN for the ASCGEN-DEMO design was reasonably complete, and showed that both the code generated and its ordering were entirely correct.

The RMS design was used to bench-mark ASCGEN performance throughout a series of modifications to improve speed and resource usage. An order of magnitude speed improvement was obtained. The allocated virtual stack space (the maximum available for a translation) was reduced from five megabytes (and this still wasn't enough) to 384 kilobytes, allowing ASCGEN to co-reside with CAEDE on a SUN work-station. At the completion of an RMS translation, 5% of the Global Stack and 0.1% of the Local Stack were not recovered when compared to that for the first version of ASCGEN able to translate RMS.

A constraint of the implementation environment (C-Prolog/1.4) was discovered during the performance testing - that it required the designer to adopt a coding style that explicitly released resources, in order to perform large translations.

The final version of ASCGEN produced source S-Ada (code and comments) at a rate of 600-700 lines per minute on a VAX/11-750, C-Prolog/1.4. This rate dropped slightly when the complexity of a design increased. We feel this rate is more than adequate for the current use of ASCGEN as a tool to generate code for further, manual completion, and adequate for possible future use in the iterative design-compile-redesign cycle.

Some translation factors were derived to predict the translation time (approximately four minutes per thousand design facts), and the size of an S-Ada program (approximately three lines of S-Ada per design fact), for an arbitrary design input.

CHAPTER 5 CONCLUSIONS AND RECOMMENDATIONS

5.1 CONCLUSIONS

The successful analysis, design, implementation and testing of ASCGEN proved the feasibility of implementing translators in Prolog, for translating design facts into target source code in the style developed in this thesis.

An analysis of the software requirements for ASCGEN was performed, which clarified the semantics for mapping CAEDE-notation designs into S-Ada code (Appendix A), and which produced a specification for S-Ada (Appendix B).

The design process partitioned ASCGEN into the following phases:

- * Analysis: syntactic and semantic analysis of CAEDE-notation structures;
- * Transformation: mapping of CAEDE-notation structures into relevant S-Ada syntactic categories;

- * Code Generation: of formatted S-Ada syntactic categories.

ASCGEN's phases were executed in a sequential fashion within an overall recursive framework. This was felt to be the most natural algorithm when the requirement for reversibility in translation was dropped. A feasibility analysis on implementing a bi-directional translation capability and a DIANA intermediate representation into ASCGEN showed their costs were not offset by their benefits. The design of ASCGEN avoided the complexity, and resource overhead of incorporating reversibility.

A constraint of the implementation environment (C-Prolog/1.4) was discovered during the implementation/testing cycle - C-Prolog/1.4 required the designer to adopt a coding style that explicitly released resources, in order to perform large translations.

The final version of ASCGEN consisted of 280 clauses (101 facts, 179 rules, 88K code plus comments). The coding of the CAEDE-notation template transformation capability (Transformation Phase) was found to be the most difficult to implement, because of the context analysis required before code generation.

ASCGEN successfully translated the functional tests prepared for it. The skeleton code (S-Ada) generated was successfully validated against the S-Ada specification.

Ada code was added to the S-Ada code produced from the ASCGEN-DEMO test input to make it compilable. Compilation succeeded with the simple addition of code tokens (30% more executable code tokens; 75% of the

2

tokens were declarations) and without any other changes to the code produced by ASCGEN (i.e. without reordering the code). The relatively high percentage of declaration tokens needed for compilability indicates that adding CAEDE support for the automatic generation of declarations could significantly increase the level of compilability. The overall result of this compilation exercise was a clear demonstration of the correctness, correct ordering and completeness of the code produced by ASCGEN.

Performance improvements were achieved in both time and space consumption through common benchmark tests applied after each of a series of design modifications. An order of magnitude speed improvement was obtained. The allocated virtual stack space, necessary to permit translations of reasonably large designs, was reduced from five megabytes (and this still wasn't enough) to 384 kilobytes, allowing ASCGEN to co-reside with CAEDE on a SUN work-station.

The final version of ASCGEN produced source S-Ada (code and comments) at a rate of 600-700 lines per minute, dropping slightly with increasing complexity of design input.

Some translation performance factors were empirically derived by comparing performance for different classes of input designs. ASCGEN required approximately four minutes translation time per 1000 design facts, and produced approximately 3000 lines of S-Ada code per 1000 design facts.

5.2 RECOMMENDATIONS FOR FURTHER WORK

The most desirable expansion of ASCGEN would be to make the generated code more complete, preferably compilable. This would require enhancements to the graphical CAEDE-notation to permit more information to be input through the iconic interface (e.g. to support data structuring and to support algorithmic interaction code ordering). This would allow more complete declarative and sequence of statements code to be generated.

An alternative to designing totally at the CAEDE-notation level, would be to add supplementary information in the form of code strips, a feature previously suggested by others in the CAEDE project. We suggest implementing a mechanism for accumulating code strips via user interaction with ASCGEN, which would be aware of the non-terminal syntactic categories to be generated (during translation). A means to store the information that was input would be useful, in order to avoid the need to re-enter the data for each iteration of code generation. A tool could be developed for checking the syntax of the input code that completes the skeleton, to ensure the consistency of the supplementary code with the contents of other CAEDE data bases.

We recommend incorporating more of the features of Ada that support large program development (library units, subunits), into CAEDE and its graphical design notation. This would allow larger, or more complex designs to be handled by ASCGEN, since its limited resources could be sequentially applied to code generation of parts of a design, rather than the entire design.

Other versions of Prolog should be reviewed:

- (a) To assess their support for format and string operations, and their usage of stack space at run-time, to determine if they overcome the limitations of C-Prolog/1.4. Potential performance improvements may be immediately possible. In addition, design techniques might be refined to take advantage of any useful features discovered;
- (b) To assess the debugging environment, including the capability of built-in predicates to monitor run-time performance. Predicates which make available run-time statistics in a form that is directly useable by other predicates are desirable.

A useful feature to add to the CAEDE interface would be the generation of a message to the user estimating the time for code generation and the number of lines of code to be generated from the current design. The translation factors obtained from the test phase show this to be practical for ASCGEN.

Further work could also be directed to support alternate target languages.

We recommend implementing a tool to verify the consistency of the Prolog PDL on the basis of the "Prolog PDL Semantic Requirements Definition, Appendix A".

The above recommendations are based on experience gained during the development of ASCGEN in the 1984-1985 period. They do not necessarily reflect the state of further CAEDE releases since ASCGEN development was

CONCLUSIONS AND RECOMMENDATIONS

5-6

concluded by the author.

REFERENCES

[Albrecht 80]

Albrecht, P.F., Garrison, P.E., Heyerle, R.H., Ip, P., Krieg-Bruckner, B., "Source-to-Source Translation: Ada to Pascal and Pascal to Ada", *Sigplan Notices*, vol. 15, no. 11, pp. 183-193, (Nov. 1980).

[ALRM 83]

United States Department of Defence, Reference Manual for the Ada Programming Language - ANSI/MIL-STD-1815A-1983, U.S. Government Printing Office, Washington, D.C., (Feb. 17, 1983).

[Buhr 84]

R. J. A. Buhr, *System Design With Ada*, Prentice-Hall Inc., New Jersey, (1984).

[Buhr 85]

Buhr, R.J.A., Woodside, C.M., Karam, G.M., Van Der Loo, K.W., Lewis, G. D., *Experiments With Prolog Design Descriptions and Tools in CAEDE: A New Design Environment for Ada*, Eighth International Conference on Software Engineering, London, England, (Aug. 85).

[Buhr 85A]

Buhr, R.J.A., Hayes, C.J., Karam, G.M., Neron, E., Van Der Loo, K.W., "CAEDE 1.3 User's Guide", Report SCE-85-15, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, (1985).

[Butler 84]

K. J. Butler, "DIANA Past, Present and Future", pp. 3-22 in Ada Software Tools Interfaces Workshop, Bath 1983, Lecture Notes in Computer Science No. 180, ed. P. J. L. Wallis, Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, (1984).

[Clocksin 81]

W. F. Clocksin, C. S. Mellish, Programming in Prolog, Springer-Verlag, Berlin, Heidelberg, New York, (1981).

[Gabber 82]

Eran Gabber, "The Middle Way Approach for Ada Based PDL Syntax", Ada Letters, vol. II, no. 4, pp. 64-67, (???)

[Goos 83]

G. Goos, W. A. Wulf, A. Evans, K. J. Butler (Editors), DIANA, An Intermediate Language for Ada (Revised Version), Lecture Notes in Computer Science No. 161, Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, (1983).

REFERENCES

3

[Hart 82]

Hal Hart, "Ada for Design, An Approach for Transitioning Industry Software Developers", Ada Letters, vol. II, no. 1, pp. 1-12, (Jul-Aug. 1982).

[Luckham 1985]

David C. Luckham, Friedrich W. von Henke, "An Overview of Anna, a Specification Language for Ada", IEEE Software, vol. 2, no. 2, pp. 9-22, (Mar. 1985).

[MacLennan 83]

B. J. MacLennan, Principals of Programming Languages: Design, Evaluation, and Implementation, CBS College Printing, Holt, Rinehart and Winston, New York, (1983).

[Pearce 86]

T. Pearce, Adapting an Iconic Software Design Environment for Protocol Specification with Estelle, M. Eng. Thesis, Carleton University, Ottawa, Ontario, Canada, (1986).

[Pereira 81]

Fernando Pereira (Ed.), C-Prolog User's Manual - Version 1.4, SRI International, Menlo Park, California, (February 23, 1984).

REFERENCES

[Waite 84]

W. M. Waite, G. Goos, Compiler Construction, Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, (1984).

[Warren 77]

David H. D. Warren, "Logic Programming and Compiler Writing", Report No. 44, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, (Sept. 1977).

[Warren 77A]

David H. D. Warren, Luis M. Pereira, Fernando Pereira, "Prolog - The Language and its Implementation Compared with Lisp", Procs. ACM Symp. on AI and Programming Languages, pp. 109-115, Rochester, N.Y., (Aug. 1977).

[Waugh 80]

D. W. Waugh, "Ada as a Design Language", IBM Software Engineering Exchange, vol. 3, no. 1, pp. 8-12, (Oct. 1980).

APPENDIX A PROLOG PDL SEMANTIC REQUIREMENTS DEFINITION

A.1 INTRODUCTION

This appendix covers the issues and assumptions made in order to complete the syntactic and semantic meaning of the Prolog PDL schema which defines the design data base used by version 1.2 of ASCGEN, in the version 1.2 release of the CAEDE system.

CAEDE maintains several primary data bases for each CAEDE-notation design being developed: PICTURE STORE and SYSTEM STORE. The abstract CAEDE-notation design content is extracted from SYSTEM STORE and transformed into the LOGIC STORE data base, when required for consultation by various Prolog tools. The LOGIC STORE data base schema is defined by a Prolog based Program Design Language (PDL), described in [Hayes 84]. However, some syntactic and semantic ambiguities remain in that description.

This appendix identifies these ambiguities and adds additional semantic information to the definition of the Prolog PDL, to facilitate the translation. It constitutes a set of syntax and semantics requirements for correct CAEDE-notation designs. Checking conformance to these requirements could be done by the CAEDE user, by the ASCGEN translation tool, or by a separate checking tool. ASCGEN actually utilizes a combination of manual and internal checking. The problems addressed by this appendix can be used to guide improvements in the CAEDE Prolog PDL.

In the following discussions, the frame of reference is typically a concept expressed in terms of the graphic design language CAEDE-notation, supported by CAEDE. Where obscure, mapping of abstract CAEDE-notation syntax and semantics into the LOGIC STORE data base will be clarified as needed.

A.1.1 AN OVERVIEW OF THE CAEDE-NOTATION

The following definitions of CAEDE-notation structures are introduced in order to define terms of reference for discussions that follow:

task_family ::=

- single_task
- | task_instance
- | task_template

task_instance ::=

- static_task
- | dynamic_task
- | static_taskpool
- | dynamic_taskpool

package_family ::=

- single_package
- | package_instance
- | package_template

channel_family ::=

- single_channel
- | channel_instance
- | channel_template

A.1.2 IDENTIFIERS

Syntax checking is not sufficient to ensure that the structural uniqueness present in the primary CAEDE data base, SYSTEM STORE, will be preserved through the transformation into the Prolog FDL data base, LOGIC STORE.

When icons are entered as part of a CAEDE design session, user prompts are generated requesting supplementary textual data, typically identifiers which name this icon. CAEDE enforces the following syntax:

```
<identifier> ::=  
    <letter> ( [<under_line>] <letter_or_digit> )
```

The LOGIC STORE schema uses a composite key to distinguish facts, typically as follows:

```
[ <graphic_type>, <name>, [ <parent_name>,  
    <grandparent_name>, ... ] ]
```

In Ada, it is possible to concurrently use the same identifier to refer to distinct Ada syntactic objects (called overloading). Extra features of a program (beyond the name) are checked by an Ada compiler to determine which of the identically-named objects is intended in a particular statement.

Ada overloading cannot be supported by the CAEDE Prolog FDL in its present form. Although it is possible to give the same name to CAEDE-notation objects with the same parent in SYSTEM STORE, their uniqueness will be lost when they are transformed into the LOGIC STORE data base. The reason for this is that the composite key used to identify an object cannot distinguish between CAEDE-notation items with the same ancestry, name and type. Even, if the types were different, there will be problems in distinguishing the ancestry of their children (if they have any). In this case, it would not be clear to which parent a child item belongs, because their parents names, grandparents names, etc. would be identical, but the type for their parents (the distinguishing feature) is not present in the ancestry list.

For similar reasons, non-Ada CAEDE-notation constructs have restrictions on the choice of naming to prevent ambiguous translation from SYSTEM STORE to LOGIC STORE. For example, the names for all port plugs, connection points and guards associated with the same parent must be unique to avoid confusion.

The following rule summarises the requirements on a CAEDE-notation design to prevent ambiguous ancestry in LOGIC STORE: Icon labels must be unique for icons with the same parent and which are in the same set of graphic types. These sets are defined as follows:

(a) Set for Objects and Components:

```
{  
  channel,  
  channel_template,  
  task,  
  taskpool,  
  task_template,  
  task_entry,  
  package,  
  package_template,  
  procedure,  
  pack_ext_procedure,  
  vis_data_area,  
  inv_data_area,  
}
```

(b) Miscellaneous:

```
{tx_port_plug}  
{rx_port_plug}  
{connection points}  
{guard}
```

The content of SYSTEM STORE will be assumed to be constrained according to the following suggestions, to prevent ambiguous ancestry. This implies user discretion or other consistency checking be applied to SYSTEM STORE to ensure these assumptions are in fact correct. Such enforcement is beyond the scope of this thesis.

A.1.3 TEMPLATES AND INSTANCES

This section covers missing syntax and semantics for templates and instances which are not described in the Prolog based PDL definition [Hayes 84].

Templates are CAEDE-notation structures that have their origin in the concept of the Ada task type and generic.

We assume Package-templates and Task-templates to be abstractions of the Ada generic unit and task type respectively, rather than having a one-to-one correspondence with one another. This provides flexibility in semantics. The context of an instance will then define the Ada code to implement this CAEDE-notation structure. This assumption simplifies the CAEDE-notation semantic rules, but increases the amount of analysis that ASCGEN must perform in order to minimize the number of different Ada task types or generics needed to implement instances from a common template.

A template instance is the product of an instantiation. An instantiation creates a clone from a designated template. In a CAEDE-notation design session, an instance takes a name label as its only parameter.

There are three kinds of instantiations:

- (a) Static instantiations which produce static access instances:

```
static_task;  
static-taskpool;  
package_instance.
```

- (b) Dynamic instantiations which produce dynamic access instances:

```
dynamic_task;  
dynamic_taskpool.
```

- (c) Connection instantiations which create channel_instances from channel_templates.

A.1.4 TEMPLATE USAGE

A.1.4.1 Package family And Task family

The compatibility of usage of task_instances, package_instances, task_templates and package_templates must be ensured. The instances referred to are static_task, static_taskpool, dynamic_task, dynamic_taskpool and package_instances.

Valid usage occurs when the following conditions are met:

- (a) A template exists for each instance in the design.
- (b) The labels and number of connection points on all instances and corresponding templates match.
- (c) The type and access mode of interaction/connection point combinations on each instance must match their counterparts on the associated template.

It is expected that the addition or deletion of connection points will be permitted on a task_template or package_template icon frame until at least one instance of that template exists.

If all instances of a template are deleted, such editing should again be permitted.

Such editing should never be permitted on an instances' icon frame.

If these requirements are not enforced by the iconic interface or the LSM, then specific fact consistency must be checked to ensure correct template usage prior to code generation. This is however beyond the scope of the prototype.

A.1.4.2 channel family

The validity of usage of `single_channels` and `channel_instances` must be ensured. A port connection is made when two ports are linked by either a `single_channel` or a `channel_instance`. A design contains valid port connections when the following conditions are met:

- (a) All port pairs are interconnected by one channel, either a `single_channel` or a `channel_instance`.
- (b) A `channel_template` exists for each `channel_instance` in the design.
- (c) For each `rx_port_plug` on a port connector, there is an identically labelled `tx_port_plug` on the corresponding port.
- (d) For each `tx_port_plug` on a port connector, there is an identically labelled `rx_port_plug` on the corresponding port.
- (e) The type and access mode of interaction/port plug combinations on each port must match their counterparts on the associated port connector.

It is necessary that port plug addition, deletion, and editing not be allowed on a port connector once an instance of that template exists. Ports must be bare of port plugs before they may be connected by instantiation. If all port connections using that template are deleted then such editing should again be permitted. Editing of port labels and port plugs is forbidden on ports that are connected by instantiation.

For a port connected by a `single_channel`, a different approach is necessary. The addition and deletion of port plugs to ports is permitted until a connection is made. If the connection is deleted, such editing should again be permitted.

If these editing requirements are not enforced by the iconic interface or LSM consistency checking, then specific fact checking is required to ensure all connections are valid, before code generation.

A.1.5 ACTION CHAINS

Action_chains define the syntax and semantics for combinations of interaction and creation primitives.

A.1.5.1 Action Chain Syntax

action_chain ::=

interaction_chain
| creation_chain

interaction_chain ::=

initial_src-final_dest-interaction
| initial_src-interim_dest-interaction
(interaction-sub_chain)
interim_src-final_dest-interaction

creation_chain ::=

initial_creation_src-instance-creation_arrow
| initial_creation_src-interim_dest-interaction
(interaction-sub_chain)
interim_creation_src-instance-creation_arrow

interaction-sub_chain ::=

interim_src-sub_dest-interaction
(sub_src-sub_dest-interaction)
sub_src-interim_dest-interaction

```
initial_creation_src ::=
```

```
    main_line
```

```
    ! rx_port/plug
```

```
    ! pack_ext_procedure
```

```
    ! procedure
```

```
    ! task_entry
```

```
interim_creation_src ::=
```

```
    rx_port_plug
```

```
    ! package_instance-connection_point
```

```
    ! task_instance-connection_point
```

```
sub_src ::=
```

```
    single_package-connection_point
```

```
    ! single_task-connection_point
```

```
final_dest ::=
```

```
    single_package-pack_ext_procedure
```

```
    ! package_instance-pack_ext_procedure
```

```
    ! procedure
```

```
    ! entry
```

```
    ! task_instance-task_entry
```

```
    ! single_package-vis_data_area
```

```
    ! package_instance-vis_data_area
```

```

    inv_data_area

interim_dest ::=
    tx_port_plug
    | task_template-connection_point
    | package_template_connection_point

sub_dest ::=
    single_package-connection_point
    | single_task-connection_point

```

A.1.5.2. Interaction Semantics

The source of an interaction does not constrain the interaction-type or interaction-access-mode. However, the source of an interaction targetted on a connection point must be an internal of the icon to which the connection point is attached.

With the exception of the creation-chain interactions which must all be static type and wait mode interactions, interaction access mode, type and destination are valid as follows:

DESTINATION	TYPE	ACCESS MODE
task entry of: dynamic_task, dynamic_taskpool	dynamic	wait, no-wait, timed-wait
task entry of: static_task, static_taskpool	static	wait, no-wait, timed-wait

vis_data_area of: single_package, package_instance	static	wait
inv_data_area	static, dynamic	wait
tx_port_plug, connection_point	static, dynamic	wait, no-wait, timed-wait
pack_ext_procedure of: single_package, package_instance	static	wait
procedure	static	wait

A.1.5.3 Interaction-Chain Semantics

A legal interaction-chain is two or more legal interactions drawn as a chain, from an initial source to a final destination, and which are compatibly linked. A link occurs where two interactions join at a connection point. A link is compatible when the type and access modes of both interactions match, and the upstream interaction has no data flows associated with it.

A.1.5.4 Creation-Chain Semantics

A legal creation-chain consists of compatibly linked interactions and creation-arrow. Interactions are restricted to have the static type and wait mode.

A.1.5.5 Data Flows

CAEDE requires data flows, if any, to be associated with final-destination interactions. Therefore, data flows associated with anything other than a final_destination interaction will be ignored. Each data flow bound to an interaction is interpreted as a formal parameter specification.

A.1.6 FORMAL PARAMETERS

The formal parameter list for an entry, procedure or pack_ext_procedure graphic-type is defined by the in-bound final-destination interactions connected with it. Each data flow contributes one formal parameter. Hence these data flows must all be name unique. The formal parameters provided in the component facts of the subject graphic-types are assumed to follow these rules.

APPENDIX B SKELETON ADA (S-ADA) TARGET LANGUAGE DEFINITION

B.1 INTRODUCTION

This standard evolved as part of the Carleton University Embedded Computer Systems Design Environment (CAEDE) project. A major aspect of this project is the development of graphics-based tools to aid in the design of concurrent software system architectures. These designs are extensions of the original Software Design With Ada (SDWA) design methodology due to Buhr [Buhr 84]. CAEDE supports a graphics based design methodology and notation which will be referred to as CAEDE-notation. Since the graphics primitives used correspond to Ada language constructs, Ada was chosen as one of the possible implementation languages for these design architectures. This standard describes Skeleton-Ada (S-Ada). S-Ada is the specification of the form and content of code to be produced by an automatic code generator which will make the translation from a CAEDE-notation design input.

B.1.1 SCOPE

This standard specifies the form and meaning of S-Ada skeleton program units. These units implement the essential syntax and semantics in CAEDE-notation designs.

This standard is a supplement to the 'Reference Manual for the Ada Programming Language ANSI/MIL-STD-1815a-1983' and supercedes it for the purposes as noted herein.

This standard also relies on CAEDE 1.2 standards for graphics icons and related design data base contents.

B.1.2 EXTENT OF THE STANDARD

This standard specifies:

- (a) The form of the primitive skeleton code constructs which directly or by aggregate combination implement all CAEDE graphics objects. Certain high level CAEDE graphics objects will map into aggregates of these primitives (e.g. channels).
- (b) Restrictions on the organization of primitive skeleton code constructs within a compilation unit so that automatic code generation can be simplified.
- (c) Definitions for terminology which are not part of the Ada standard or CAEDE standards.

This standard does not specify:

- (a) The rules for translating a CAEDE-notation design data base into a skeleton compilation unit.
- (b) The means whereby the skeleton compilation unit is completed. (ie. adding the flesh to the skeleton) This includes maintaining the integrity of the skeleton code throughout the completion phase.

B.1.3 STRUCTURE OF THE S-ADA STANDARD

This document is divided and named in a manner like that of the Ada Reference Manual. S-Ada grammar rules are restrictions or special cases of some of the Ada grammar.

Each chapter or section contained herein defines productions which could be made by an automatic code generator.

B.1.4 S-ADA DESIGN GOALS

S-Ada grammar is intended to be the output specification of the form and content of code to be produced by a code generation tool. The grammar rules define the syntax of the target Ada source that can be automatically generated. S-Ada is also intended to give guidance to an implementor. This is accomplished by substituting Ada non-terminals as code when the code generator cannot produce compilable code.

B.1.5 S-ADA LANGUAGE SUMMARY

As its' name implies, S-Ada code is incomplete Ada code, consisting of partially specified:

- (a) packaging structures, such as packages, tasks, and subprograms;
- (b) data objects such as variables;
- (c) interactions such as data accesses, subprogram calls, and entry calls, and
- (d) Ada grammar non-terminals where essential Ada code could not be determined.

However, high-level program structure is fully specified:

- (a) object functional descriptions which implement system architectural semantics;
- (b) nesting of major and minor objects; and
- (c) object interactions and their functional descriptions can provide major algorithmic highlights.

These features are intended to capture the essential syntax and semantics in CAEDE-notation designs.

There are only a few restrictions imposed on the Ada code used to complete the skeleton.

B.1.6 METHOD OF S-ADA DESCRIPTION AND SYNTAX NOTATION

The original description in the Ada Reference Manual applies, with the following exceptions:

- f. Instead of using italicized prefix name(s) to add semantics to a syntactic category, prefix name(s) will be put in quotes, in the absence of a word processor/printer to generate italics.
- g. S-Ada syntactic category extensions:

Ada syntactic categories are typed in lower case. Their elaboration may be reviewed in the Ada Reference Manual.

`ada_syntactic_categories =>`

Any Ada syntactic categories that are not boldfaced and do not begin with upper case prefixes mark code to be provided by the code generator. Such Ada code shall be available either directly or by transformation from a CAEDE-notation design data base or be synthesized during code generation.

`S-Ada_syntactic_categories =`

Ada syntactic categories prefixed by "S_" are S-Ada non-terminals whose expansion will be defined in this document. The number of the expansions correspond to Ada syntactic categories in the Ada Reference Manual.

`boldfaced_S-Ada_terminals =>`

Boldfaced syntactic categories, reserved words, BNF metavariables, and special characters indicate S-Ada terminals. These terminals should be output in the exact form as shown in this document.

`CODE_STRIP =`

The form of this code may be derivable from the CAEDE design data base. A simple code fragment insertion from an information fact is one possible method.

`OTHER_LABELS_syntactic_category_COMMENTS =`

The form of comments is user defined. A restricted natural language description such as a PDL could be used to express the required information, but is left open to satisfy varying user requirements.

S_ or _S_ =

This label element is a mnemonic for SKELETON.
UPPER_CASE_PREFIX_OR_SUFFIXES =

These are special categories which are defined in this standard. For example:

```
CODE_STRIP( ICON; CONTEXT ) ::=
-- one or more lines of user-defined free formatted
  text consisting of code or comments
-- code generator will select appropriate strip on the
  basis of parameters listed.
```

B.2 DETAILED S-ADA LANGUAGE DEFINITION

3. SKELETON DECLARATIONS AND SKELETON TYPES

3.1 SKELETON DECLARATIONS

S_basic_declaration ::=

S_object_declaration	:	S_subprogram_declaration
S_package_declaration	:	S_task_declaration
S_generic_declaration	:	S_generic_instantiation
CHANNEL_declaration	:	DATA_AREA_declaration

CHANNEL_declaration ::=

```
S_HEADER.CG_CODE(
  ICON      => channel,
  CONTEXT   => Ada body code, channel declaration
)
(S_basic_declaration)
S_FOOTER.CG_CODE(
  ICON      => channel,
  CONTEXT   => Ada body code, channel declaration
)
```

DATA_AREA_declaration ::=

```

    S_HEADER.CG_CODE(
        ICON =>      vis_data_area
        CONTEXT =>   Ada declaration code
    )

```

```

    : S_HEADER.CG_CODE(
        ICON =>      inv_data_area
        CONTEXT =>   Ada body code
    )

```

3.2 SKELETON OBJECTS

S_object_declaration ::=

STATIC_TASK_INSTANTIATION_object_declaration

: STATIC_TASKPOOL_object_declaration

STATIC_TASK_INSTANTIATION_object_declaration ::=

```

    S_HEADER.CG_CODE(
        ICON      =>   static task instance
        CONTEXT   =>   Ada declaration code of parent
    )
    "task"_identifier : "task_type"_identifier

```

STATIC_TASKPOOL_object_declaration ::=

```

    S_HEADER.CG_CODE(
        ICON      =>   static taskpool
        CONTEXT   =>   Ada declaration code of parent
    )
    "task_pool"_identifier : array( <index_constraint> ) of
        "task_type"_identifier ;

```

3.9 SKELETON DECLARATIVE PARTS

S_declarative_part ::=

```

    {S_basic_declarative_item}
    {<basic_declarative_item>}
    {S_later_declarative_item}
    {<later_declarative_item>}

```

S_basic_declarative_item ::=

S_basic_declaration

```

    | S_HEADER_CODE(
        ICON =>      vis_data_area
        CONTEXT =>    Ada declaration code
    )

```

```

    | S_HEADER_CODE(
        ICON =>      inv_data_area
        CONTEXT =>    Ada body code
    )

```

```

    | S_HEADER_CODE(
        ICON =>      channel
        CONTEXT =>    Ada body code (channel declaration)
    )

```

S_later_declarative_item ::=

```

    S_body          | S_subprogram_declaration
    | S_package_declaration | S_task_declaration
    | S_generic_declaration | S_generic_instantiation

```

S_body ::= S_proper_body

S_proper_body ::=

```

    S_subprogram_body | S_package_body | S_task_body
    | CHANNEL_body

```

```

CHANNEL_body ::=

    S_HEADER_CG_CODE(
        ICON      => channel,
        CONTEXT   => Ada body code, channel body
    )
    (S_later_declarative_item)
    S_FOOTER_CG_CODE(
        ICON      = channel,
        CONTEXT   = Ada body code, channel body
    )

```

4. SKELETON NAMES AND EXPRESSIONS

4.1 SKELETON NAMES

```

S_name ::= S_simple_name

```

```

S_simple_name ::= "declaration"_identifier only

```

- a restriction for skeleton code only, no renaming_declarations are permitted.
- identifiers are directly mapped or transformed from a CAEDE-notation design data base or are synthesized during the code generation phase

4.1.3 SKELETON SELECTED COMPONENTS

```

S_selected_component ::= S_prefix.S_selector

```

```

S_prefix ::= S_name

```

```

S_selector ::= S_simple_name

```

5. SKELETON STATEMENTS

5.1 SKELETON SIMPLE AND COMPOUND STATEMENTS -
SEQUENCE OF STATEMENTS

S_sequence_of_statements ::=

```

        S_statement          -- alternative if code
        {S_statement}        -- to be generated
        {<statement>}

        : <statement>        -- alternative when no code.
        {<statement>}

```

S_statement ::=

```

        {<label>} S_simple_statement
        : {<label>} S_compound_statement
        : S_HEADER_CG_CODE(
            ICON      => vis_data_area or inv_data_area
            CONTEXT   => access code
        )
        : <statement>

```

S_simple_statement ::=

```

        S_procedure_call_statement
        : S_entry_call_statement
        : S_delay_statement
        : S_assignment_statement

```

S_compound_statement ::=

```

        S_accept_statement
        : S_select_statement

```


5.2 SKELETON ASSIGNMENT STATEMENTS

```

S_assignment_statement ::=

    "dynamic_task_instantiation"_CG_CODE

"dynamic_task_instantiation"_CG_CODE ::=

    S_HEADER_CG_CODE(
        ICON      => dynamic task instance
        CONTEXT    => instantiation code
    )
    "dynamic_task"_identifier : "access_type"_identifier :=
        new : "task_type"_identifier ;

```

6. SKELETON SUBPROGRAMS

6.1 SKELETON SUBPROGRAM DECLARATIONS

```

S_subprogram_declaration ::=

    S_HEADER_CG_CODE(
        ICON      => pack_ext_proc,
        CONTEXT    => Ada declaration of parent
    )
    S_subprogram_specification ;

    S_HEADER_CG_CODE(
        ICON      => pack_ext_proc,
        CONTEXT    => Ada body of parent
    )
    S_subprogram_specification ;

S_HEADER_CG_CODE(

    ICON ;      -- graphic type code representing

    CONTEXT ;   -- point in Ada code that this code will be
                -- positioned, as well as the purpose of
                -- this code

) ::=

```

```

BORDER_STRING_COMMENT( ICON )
CG_COMMENTS( ICON, CONTEXT )
[ CODE_STRIP( ICON, CONTEXT ) ]
ADA_SYNTACTIC_CATEGORY.CG_CODE( CONTEXT )
BORDER_STRING_COMMENT( ICON )

```

```

BORDER_STRING_COMMENT( ICON ) ::=

```

```

-- A comment border string intended to permit easy
-- association of a skeleton code fragment with the
-- CAEDE-notation design icon structure that it represents
-- char = function of ICON

```

```

CG_COMMENTS( ICON; CONTEXT ) ::=

```

```

-- comments appropriate to the icon and context

```

```

CODE_STRIP( ICON; CONTEXT ) ::=

```

```

-- One or more lines of user-defined free-formatted
-- text consisting of code or comments.
-- code generator will select appropriate strip on the
-- basis of parameters listed.

```

```

ADA_SYNTACTIC_CATEGORY.CG_CODE( ICON; CONTEXT ) ::=

```

```

-- uncompileable Ada syntactic category code with
-- supplementary terminal 'meta-variables' used to
-- mark points in code where further code completion is
-- required.
-- SYNTAX:

```

```

-- OPTIONAL:      [ <Ada_syntactic_category> ]
-- MANDATORY:     <Ada_syntactic_category>
-- ZERO OR MORE:  ( <Ada_syntactic_category> )

```

```

S_subprogram_specification ::=

```

```

    procedure identifier [ S_formal_part ]
    | function designator [ S_formal_part ]

```

```

        return type_mark

S_formal_part ::=
    (S_parameter_specification
     ( ; S_parameter_specification )

S_parameter_specification ::=
    "formal_parameter" _identifier_ mode type_mark
    [ := <expression> ]

mode ::= [ in ] | in out | out

```

6.3 SKELETON SUBPROGRAM BODIES

```

S_subprogram_body ::=
    -- CAEDE-notation does not currently support functions.

    S_HEADER_CODE(
        ICON      => pack_ext_procedure -- no code strip
                  or procedure -- code strip
        CONTEXT   => Ada body code of parent
    )
    S_subprogram_specification is
        S_declarative_part
    begin
        S_sequence_of_statements
    [exception
        <exception_handler>
        (<exception_handler>) ]
    end [ designator ] ;

```

6.4 SKELETON SUBPROGRAM CALLS

```

S_procedure_call_statement ::=
    "procedure"_name [ S_actual_parameter_part ] ;

S_actual_parameter_part ::=
    ( S_parameter_association {, S_parameter_association} )

S_parameter_association ::=
    S_formal_parameter => S_actual_parameter

S_formal_parameter ::= "parameter"_simple_name

S_actual_parameter ::=
    <actual_parameter>

```

7. SKELETON PACKAGES

7.1 SKELETON PACKAGE STRUCTURE

```

S_package_declaration ::=
    S_HEADER_CODE(
        ICON      = package or package instance
        CONTEXT   = package declaration
    )
    S_package_specification ;

S_package_specification ::=
    package identifier is
        { S_basic_declarative_item }
        { <basic_declarative_item> }
    [ <private>

```

```

    {<basic_declarative_item>} ]
end "package"_S_simple_name

```

S_package_body ::=

```

    S_HEADER_CG_CODE(
        ICON      => package or package instance
        CONTEXT   => package body
    )
    package body "package_S_simple_name" is
        S_declarative_part
    begin
        CODE_STRIP(
            ICON      => package_mainline,
            CONTEXT   => user code, comments
        )
        S_sequence_of_statements
    [ exception
        <exception_handler>
        {<exception_handler>} ]
    end "package"_S_simple_name ;

```

9. SKELETON TASKS

9.1 SKELETON TASK SPECIFICATIONS AND TASK BODIES

S_task_declaration ::=

```

    S_HEADER_CG_CODE(
        ICON      => static task
        CONTEXT   => static task declaration
    )
    S_task_specification ;

    "task_type_declaration"_S_HEADER_CG_CODE

    S_HEADER_CG_CODE(
        ICON      => none directly
        CONTEXT   => task type declaration derived
                    from dynamic tasks and all taskpools.
    )
    S_task_"type"_specification ;
    [ type "access_type"_identifier is access
      "task_type"_identifier ] ;

```

```
-- included if template used for dynamic task
-- instantiation.
```

```
S_task_specification ::=
```

```
task [type] identifier is
    {S_entry_declaration}
    {<entry_declaration>}
    {<representation_clause>}
end "task"_S_simple_name
```

```
S_task_body ::=
```

```
S_HEADER_CODE(
    ICON      => static task, else none for task types
    CONTEXT   => static task, else task type body
from dynamic tasks and all taskpools.
)
task body "task"_S_simple_name is
    S_declarative_part
begin
    [ CODE_STRIP(
        ICON      => task mainline,
        CONTEXT   => user code, comments
    ) ]
    loop          -- include if static task or task type
        S_sequence_of_statements
    end loop ;    -- include if static task or task type
[ exception
    <exception_handler>
    {<exception_handler>} ]
end "task"_S_simple_name ;
```

9.5 SKELETON ENTRIES, ENTRY CALLS AND ACCEPT STATEMENTS

S_entry_declaration ::=

```

S_HEADER_CG_CODE(
    ICON      = entry
    CONTEXT   = comments, indicates if entry guarded
)
entry identifier [ (<discrete_range>) ]
                  [ S_formal_part];

```

```

S_HEADER_CG_CODE(
    ICON      => timed_entry
    CONTEXT   = comments, indicates if entry guarded
)

```

S_entry_call_statement ::=

```

S_HEADER_CG_CODE(
    ICON      => initial source interaction
    CONTEXT   = comments on call, no code.
)
"entry"_name [S_actual_parameter_part]

```

S_accept_statement ::=

```

S_HEADER_CG_CODE(
    ICON      => entry
    CONTEXT   => entry accept header, no code.
)
accept
    "entry"_S_simple_name [ (<entry_index>) ]
                          [S_formal_part] [do -- rendezvous critical
                                          -- section
    S_sequence_of_statements
end    "entry"_S_simple_name ] ;

```

9.6 SKELETON DELAY STATEMENTS

S_delay_statement ::=

```

    delay CODE_STRIP(
        ICON      =, timed entry
        CONTEXT   = simple expression and
                    further sequence of
                    statements code.
    )
    -- if no code strip entered, generate this code and
    -- generate warning

    delay <simple_expression> ;

```

9.7 SKELETON SELECT STATEMENTS

S_select_statement ::=

```

    S_selective_wait
    | S_conditional_entry_call
    | S_timed_entry_call

```

9.7.1 SKELETON SELECTIVE WAIT

S_selective_wait ::=

```

    select
        S_select_alternative
    (or
        S_select_alternative)
    [ else
        <sequence_of_statements> ]
    end select ;

```

S_select_alternative ::=

```

    -- if code strip entered

    [
        S_HEADER_CODE(
            ICON      =, guard

```



```

        CONTEXT => cross-reference to guard label,
                    comments only, no code.
    )
    when CODE_STRIP(
        ICON      => guard,
        CONTEXT   => guard condition code
    )
    S_selective_wait_alternative

-- if no code strip entered, generate this code and
-- generate warning

: [
    S_HEADER_CG_CODE(
        ICON      => guard
        CONTEXT   => cross-reference to guard label,
                    comments only, no code.
    )
    when <condition> => ]
    S_selective_wait_alternative

S_selective_wait_alternative ::=

    S_accept_alternative
  | S_delay_alternative

S_accept_alternative ::=

    S_accept_statement
  [ <sequence_of_statements> ]

S_delay_alternative ::=

    S_delay_statement
  [ <sequence_of_statements> ]

```

9.7.2 SKELETON CONDITIONAL ENTRY CALLS

```

S_conditional_entry_call ::=

    select
        S_entry_call_statement
        [ <sequence_of_statements> ]
    else
        <sequence_of_statements>
    end select ;

```

9.7.3 TIMED ENTRY CALLS

```

S_timed_entry_call ::=

    select
        S_entry_call_statement
        [ <sequence_of_statements> ]
    or
        S_delay_alternative
    end select ;

```

10. SKELETON PROGRAM STRUCTURE AND COMPILATION ISSUES

10.1 SKELETON COMPILATION UNITS - SKELETON LIBRARY UNITS

The minimum implementation is a single skeleton compilation unit.

```

S_compilation_unit ::=

```

```

    context_clause          S_library_unit
  ! context_clause          S_secondary_unit

```

```

^S_library_unit ::=

```

```

    S_subprogram_declaration  ! S_package_declaration
    ! S_generic_declaration   ! S_generic_instantiation
    ! S_subprogram_body

```

```

S_secondary_unit ::= S_library_unit_body ! S_subunit

```

```

S_library_unit_body ::= S_subprogram_body ! S_package_body

```

10.2 SUBUNITS OF SKELETON COMPILATION UNITS

S_body_stub ::=

```
S_subprogram_specification is separate;  
! package body "package"_S_simple_name is separate;  
! task body "task"_S_simple_name is separate;
```

-- a graphics notation to mark those objects with
separate bodies has not yet been developed.

S_subunit ::=

```
separate ( "parent-unit"_S_name) S_proper_body
```

r

APPENDIX C FUNCTIONAL TEST DESIGN SPECIFICATION

This appendix contains a list of the features to test the functional capabilities of ASCGEN. The majority of them are tested by test cases:

```
ascgen_demo
chan_tst1.2x
chan_inst1.2x
```

The following features are organized by SDWA syntactic categories with SDWA inputs and S-Ada target code outputs.

C.1 DECLARATIONS

- visible data area:
 - package
 - generic
- invisible data area
- pack_ext procedure declaration in:
 - package
 - generic
 - with parameters for generic
- package declaration
- generic declaration
- generic instantiation
- task declaration
 - 0, 1 free entries
 - selective accept comments and entries
 - 2 selective entries
 - timeout_pseudo-entry comments
 - guard comments
 - 1 or more selective accepts
- task type declaration
 - entry with parameters must be tested
- static-task instance declaration
- static-task-pool declaration
- dynamic-task instance allocator declaration
- dynamic-task-pool access variable declarations
- within single channels or channel instances:

- invisible data area declarations
- package declarations
- generic instantiation
- task declaration
- static-task instance declaration
- static-task-pool declaration
- dynamic-task instance allocator declaration
- dynamic-task-pool access variable declarations

C.2 BODIES

- pack_ext procedure body in:
 - package
 - generic
 - with parameters to be tested for generic
- procedure body
- package body
- generic body
- task body
 - 0 or more free entries
 - selective accept and entries
 - 2 selective entries
 - timeout pseudo-entry comments, code
 - guard comments and code
 - 1, 2 selective accepts
- task type body
 - entry with parameters must be tested
- dynamic-task instance creation
- dynamic-task-pool creation
- within single channels or channel instances:
 - procedure body
 - package body
 - task body
 - dynamic-task instance creation
 - dynamic-task-pool creation

C.3 COMMENTS

Information facts for each of the items under DECLARATIONS should be formatted and included with the output code.

C.4 ACTION CHAIN SOURCES

Call code must be generated in the sequence of statements of the callee for each of the following sources and must conform to the required form for the DESTINATION and environment.

- originating within task or package
 - task-entry
 - procedure
 - pack-ext-procedure
 - mainline
- originating within task or package template
 - task-entry
 - procedure
 - pack-ext-procedure
 - mainline

C.5 ACTION CHAIN DESTINATIONS

Call code must be correctly generated for the following targets with and without parameters. Common code allows one test of action chains with data flows, and one without data flows for all except the generic and task type target code.

- visible data area:
 - package
 - generic
- invisible data area
- pack_ext procedure:
 - package
 - generic (with data flow)
- procedure
- wait, no-wait and timed wait calls on task-entry of:
 - task

- static-task instance
- static-task-pool instance
- dynamic-task instance
- dynamic-task-pool instance
- all of above destinations within single channels or channel instances.

C.6 ACTION CHAIN LINKS

- direct connection from source to all destinations:
 - within package or task
 - within single channel
 - within channel instance
- connection through 1 intermediate connector point:
 - through no instances
 - through 1 instance
- connection through 2 intermediate connector points:
 - through no instances
 - through 1 instance
 - through 2 instances
- single channel actions chains:
 - into channel internals
 - from within channel to outside channel
 - through channel from external source to external target
- channel instance actions chains:
 - into channel internals
 - from within channel to outside channel
 - through channel from external source to external target

C.7 PARAMETERS

- to all legal targets
 - 0
 - 1
 - greater than 1

C.8 TEMPLATE DECLARATIONS

- 1 package template
 - > 1 generic declaration -> n instances
 - > more than 1 generic declaration -> n instances
- 1 task template
 - > 1 task type declaration -> n instances
 - > more than 1 task type declaration -> n instances

C.9 CREATION CHAIN TARGETS

- dynamic task
- dynamic task-pool

C.10 CREATION CHAIN SOURCES

Correct sequence of statements code to be generated for each case of target (dyn task or dyn taskpool).

- task-entry
- procedure
- pack-ext-procedure
- mainline
- connector-point (.

C.11 CREATION CHAIN LINKS

- ~~direct~~ connection from creator to creation
- creation through 1 intermediate connector point:
 - on uniquely declared object
 - on template/instance boundary
- creation through 2 intermediate connector points:
 - through no instances
 - through 1 instance
 - through 2 instances

C.12 OTHER REQUIREMENTS

The form of code generated should conform to the S-Ada language definition.

APPENDIX D ASCGEN-DEMO S-ADA CODE

```

-----
--DESIGNER LOGIN NAME:      vandkw
--DESIGN DIRECTORY:        /e/vandkw/CAEDE_1.2/designs/ascgen_demo
--DESIGN VERSION NUMBER:    3
--TIME-DATE:               Mon Aug 25 22:24:41 1986
--CODEGENERATOR VERSION:    ASCGEN 1.2
-----

--Package Spec: main_package
package main_package is
end main_package;

--Package Body: main_package
package body main_package is

--Specs for Generic Packages from template: PACKAGE_TEMPLATE
--PACKAGE_TEMPLATE: test comment for package template
generic package PACKAGE_TEMPLATE_1 is

--Spec for visible procedure: PROC1
--PROC1: test comment for visble procedure.
procedure PROC1( DV18 : in out DATA_TYPE_18) ;

--Spec for visible procedure: PROC2
procedure PROC2( DV15 : in DATA_TYPE_15) ;
end PACKAGE_TEMPLATE_1 ;

--Specs for Generic Packages from template: PACKAGE_TEMPLATE_2

```

```

generic package PACKAGE_TEMPLATE_2_1 is

  -----
  --MAKE declarations defining data area: VDATA2
  --VDATA2:Test comment for vis data area in template.
  -----
  -- DATA AREA PARAMETERS( DV30 : in DATA_TYPE_30; )
end PACKAGE_TEMPLATE_2_1 ;

-----
--Specs for task types from template: TASK_TEMPLATE_2
-----
task type TASK_TEMPLATE_2_1 is
end TASK_TEMPLATE_2_1 ;
type TASK_TEMPLATE_2_1_ACCESS is access TASK_TEMPLATE_2_1 ;

-----
--Declaration for generic-instance: INST_PACK
--INST_PACK: test comment for instantiated package
-----
package INST_PACK is new PACKAGE_TEMPLATE_1 ;

-----
--Spec for task: TASK1
-----
task TASK1 is
  -----
  --FINISH declaration of entry: ENTRY_2
  --ENTRY_2: test comment for task entry
  -----
  entry ENTRY_2( DV14 : out DATA_TYPE_14 ) ;
end TASK1 ;

-----
--Spec for task: TASK2
-----
task TASK2 is
  -----
  --Begin select accept: ↓
  -----
  -----
  --FINISH declaration of entry: S_ENTRY_1
  -----
  --GUARD assigned is: G1
  entry S_ENTRY_1( DV19 : in DATA_TYPE_19; DV20 : in DATA_TYPE_20 ) ;
  -----

```

```

--FINISH declaration of entry: S_ENTRY_2
-----
entry S_ENTRY_2( DV16 : out DATA_TYPE_16; DV17 : out DATA_TYPE_17 ) ;
-----
--TIMEOUT pseudo-entry: 1
-----

-----
--End select accept: 1
-----

-----
--Begin select accept: 2
-----

-----
--FINISH declaration of entry: S_ENTRY_4
-----
entry S_ENTRY_4 ;
-----
--FINISH declaration of entry: S_ENTRY_3
-----
entry S_ENTRY_3 ;
-----
--End select accept: 2
-----

-----
--FINISH declaration of entry: ENTRY_5
-----
entry ENTRY_5 ;
end TASK2 ;

--////////////////////////////////////
--Declaration for generic-instance: INST2_PACK
--////////////////////////////////////
package INST2_PACK is new PACKAGE_TEMPLATE_2_1 ;

--////////////////////////////////////
--Declaration of Dynamic-task-access: DYN_TASK_2
--////////////////////////////////////
DYN_TASK_2 : TASK_TEMPLATE_2_1_ACCESS ;

--////////////////////////////////////
--Bodies for generic packages from template: PACKAGE_TEMPLATE
--////////////////////////////////////
package body PACKAGE_TEMPLATE_1 is
--////////////////////////////////////
--Specs for task types from template: TASK_TEMPLATE

```

[illegible]

```

-----
DYN_TASK : TASK_TEMPLATE_1_ACCESS ;

-----
--Declaration of static-task-instance: STAT_TASK
--STAT_TASK: test comment for static task
-----
STAT_TASK : TASK_TEMPLATE_2 ;

-----
--Package Spec: TEST_PACKAGE
--TPACK: test comment for package
-----
package TEST_PACKAGE is

-----
--Spec for visible procedure: SETUP
--SETUP:Test comment for package external procedure.
-----
procedure SETUP( DV2 : in out DATA_TYPE_2) ;

-----
--MAKE declarations defining data area: VISIBLE_DATA_1
--VDATA1: test comment for visible data area
--DATA AREA PARAMETERS( DATA_VALUE_1 : in DATA_TYPE_1; )
end TEST_PACKAGE ;

-----
--Bodies for Task Types from template:TASK_TEMPLATE
-----
task body TASK_TEMPLATE_1 is
begin
-----
--FINISH accept code for: ENTRY1
-----
accept ENTRY1 do
    TASK.ENTRY_2( DV14 => ) ;
end ENTRY1 ;
end TASK_TEMPLATE_1 ;

task body TASK_TEMPLATE_2 is
begin
-----
--FINISH accept code for: ENTRY1

```

```

    accept ENTRY1( DV3 : in out DATA_TYPE_3; DATA_VALUE_4 : in DATA_TYEPE_4 ;
                  DATA_VALUE_5 : out DATA_TYPE_5 ) do
        TASK.ENTRY_Z( DV14 => ) ;
    end ENTRY1 ;
end TASK_TEMPLATE_2 ;

```

```

-----
--Spec, body for procedure: PROCEDURE_1
--PR1: TEST COMMENT FOR INVISIBLE PROCEDURE
--
-----
procedure PROCEDURE_1( DV21 : in DATA_TYPE_21) is
    -- make necessary declacations
begin
    -- FINISH body code
end PROCEDURE_1 ;

-----
--Task body: TASK_A
-----
task body TASK_A is
begin
    -----
    --FINISH accept code for: ENTRY_A
    -----

    accept ENTRY_A( DV11 : in DATA_TYPE_11; DV12 : out DATA_TYPE_12 ) do
    end ENTRY_A ;
    -- FINISH call code for visible data area: VISIBLE_DATA_1, declared in
    -- package declaration for: TEST_PACKAGE
    TEST_PACKAGE.SETUP( DV2 => ) ;
end TASK_A ;

-----
--Package Body: TEST_PACKAGE
-----
package body TEST_PACKAGE is

    -----
    --Specs for task types from template: TASK_TEMPLATE
    --TASK_TEMPLATE: test comment for task template
    -----
    task type TASK_TEMPLATE_1_A is
        -----
        --FINISH declaration of entry: ENTRY1
        -----

```

[illegible]


```

    DYN_TASK := new TASK_TEMPLATE_1 ;
    -- FINISH call code for invisible data area: DATA_AR_10
    -- declared in parent body: TEST_PACKAGE
end TEST_PACKAGE;

```

```

-----
--Spec, body for visible procedure: PROC1
-----

```

```

procedure PROC1( DV18 : in out DATA_TYPE_19) is
    -- make necessary declarations
begin
    -- FINISH body code
    PROCEDURE_1( DV21 => ) ;
    TASK_A.ENTRY_A( DV11 => , DV12 => ) ;
end PROC1 ;

```

```

-----
--Spec, body for visible procedure: PROC2
-----

```

```

procedure PROC2( DV15 : in DATA_TYPE_15) is
    -- make necessary declarations
begin
    -- FINISH body code
    -- DYNAMIC TASKPOOL CREATION code!
    for I in DYN_POOL'RANGE loop
        DYN_POOL(I) := new TASK_TEMPLATE_1 ;
    end loop;
    select
        -- FINISH TASKPOOL call code!
        DYN_POOL( ).ENTRY1 ;
        -- FINISH any additional sequence of statements code.
    or
        delay -- FINISH delay statement code.
        -- FINISH any additional sequence of statements code.
    end select;
    select
        -- FINISH TASKPOOL call code!
        DYN_POOL( ).ENTRY1 ;
        -- FINISH any additional sequence of statements code.
    else
        -- FINISH alternative sequence of statements code.
    end select;
    PROCEDURE_1( DV21 => ) ;
    -- FINISH call code for invisible data area: DATA_AREA_2
    -- declared in parent body: PACKAGE_TEMPLATE
    select

```

```

        DYN_TASK.ENTRY1 ;
        -- FINISH any additional sequence of statements code.
    else
        -- FINISH alternative sequence of statements code.
    end select;
select
    STAT_TASK.ENTRY1( DV3 => , DATA_VALUE_4 => , DATA_VALUE_5 => ) ;
    -- FINISH any additional sequence of statements code.
or
    delay -- FINISH delay statement code.
    -- FINISH any additional sequence of statements code.
end select;
-- FINISH TASKPOOL call code'
DYN_POOL( ).ENTRY1 ;
end PROC2 ;
end PACKAGE_TEMPLATE_1 ;

--::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
--Bodies for generic packages from template: PACKAGE_TEMPLATE_2
--::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
package body PACKAGE_TEMPLATE_2_1 is
end PACKAGE_TEMPLATE_2_1 ;

--//////////////////////////////////////
--Bodies for Task Types from template:TASK_TEMPLATE_2
--//////////////////////////////////////
task body TASK_TEMPLATE_2_1 is
begin
end TASK_TEMPLATE_2_1 ;

--//////////////////////////////////////
--Task body: TASK
--//////////////////////////////////////
task body TASK1 is
begin

--FINISH accept code for: ENTRY_2
--ENTRY_2: test comment for task entry

accept ENTRY_2( DV14 : out DATA_TYPE_14) do
    -- DYNAMIC TASK CREATION code!
    DYN_TASK_2 := new TASK_TEMPLATE_2_1 ;
end ENTRY_2 ;

```

```

end TASK1;

-----
--Task body: TASK2
-----
task body TASK2 is
begin
  select -- sequence number 1
  -----
  --FINISH accept code for: S_ENTRY_1
  -----
  --FINISH guard: G1
  when =>
    accept S_ENTRY_1( DV19 : in DATA_TYPE_19; DV20 : in DATA_TYPE_20 ) do
    end S_ENTRY_1 ;
  or
  -----
  --FINISH accept code for: S_ENTRY_2
  -----
  accept S_ENTRY_2( DV16 : out DATA_TYPE_16; DV17 : out DATA_TYPE_17 ) do
  end S_ENTRY_2 ;
  or
  -----
  --FINISH TIMEOUT pseudo-entry: 1
  -----
  delay -- delay code
end select ; -- sequence number 1
select -- sequence number 2
  -----
  --FINISH accept code for: S_ENTRY_4
  -----
  accept S_ENTRY_4 do
    -- FINISH call code for visible data area: VDATA2, declared in package
    -- declaration for: INST2_PACK
  end S_ENTRY_4 ;
  or
  -----
  --FINISH accept code for: S_ENTRY_3
  -----
  accept S_ENTRY_3 do
  end S_ENTRY_3 ;
end select ; -- sequence number 2
  -----
  --FINISH accept code for: ENTRY_5
  -----
  accept ENTRY_5 do

```

```
        end ENTRY_5 ;  
        INST_PACK.PROC2( DV15 => ) ;  
    end TASK2 ;  
begin  
    TASK2.S_ENTRY_2( DV16 => , DV17 => ) ;  
    TASK2.S_ENTRY_1( DV19 => , DV20 => ) ;  
    INST_PACK.PROC1( DV18 => ) ;  
end main_package;
```

APPENDIX E ADDITIONS TO MAKE ASCGEN-DEMO COMPILE

12,43c12,43

```

<
<  subtype DATA_TYPE_1 is INTEGER;
<  subtype DATA_TYPE_2 is INTEGER;
<  subtype DATA_TYPE_3 is INTEGER;
<  subtype DATA_TYPE_4 is INTEGER;
<  subtype DATA_TYPE_5 is INTEGER;
<  subtype DATA_TYPE_6 is INTEGER;
<  subtype DATA_TYPE_7 is INTEGER;
<  subtype DATA_TYPE_8 is INTEGER;
<  subtype DATA_TYPE_9 is INTEGER;
<  subtype DATA_TYPE_10 is INTEGER;
<  subtype DATA_TYPE_11 is INTEGER;
<  subtype DATA_TYPE_12 is INTEGER;
<  subtype DATA_TYPE_13 is INTEGER;
<  subtype DATA_TYPE_14 is INTEGER;
<  subtype DATA_TYPE_15 is INTEGER;
<  subtype DATA_TYPE_16 is INTEGER;
<  subtype DATA_TYPE_17 is INTEGER;
<  subtype DATA_TYPE_18 is INTEGER;
<  subtype DATA_TYPE_19 is INTEGER;
<  subtype DATA_TYPE_20 is INTEGER;
<  subtype DATA_TYPE_21 is INTEGER;
<  ADV2 : DATA_TYPE_2 := 1;
<  ADV3 : DATA_TYPE_3 := 1;
<  ADV5 : DATA_TYPE_5 := 1;
<  ADV12 : DATA_TYPE_12 := 1;
<  ADV14 : DATA_TYPE_14 := 1;
<  ADV16 : DATA_TYPE_16 := 1;
<  ADV17 : DATA_TYPE_17 := 1;
<  ADV18 : DATA_TYPE_18 := 1;

```

206c167

```

<      MAX_DYN_POOL : constant INTEGER := 7 ;
>      MAX_DYN_POOL : constant INTEGER := ;

```

249c210

```

<      TASK1.ENTRY_Z( DV14 => ADV14 ) ;
>      TASK.ENTRY_Z( DV14 => ) ;

```

260c221

```

<      TASK1.ENTRY_Z( DV14 => ADV14 ) ;
>      TASK.ENTRY_Z( DV14 => ) ;

```

```

274d237
<      null;
285c245
<      accept ENTRY_A( DV11 : in DATA_TYPE_11; DV12 : out DATA_TYPE_12 ) do null;
>      accept ENTRY_A( DV11 : in DATA_TYPE_11; DV12 : out DATA_TYPE_12 ) do
289c249
<      TEST_PACKAGE.SETUP( DV2 => ADV2 );
>      TEST_PACKAGE.SETUP( DV2 => );
320,321c280,281
<      MAX_STATIC_POOL : constant INTEGER := 9 ;
<      STATIC_POOL : array( 1..MAX_STATIC_POOL ) of TASK_TEMPLATE_1_A ;
>      MAX_STATIC_POOL : constant INTEGER := ;
>      STATIC_POOL : array( 1..MAX_STATIC_POOL ) of TASK_TEMPLATE_1 ;
332c292
<      TASK1.ENTRY_Z( DV14 => ADV14 );
>      TASK.ENTRY_Z( DV14 => );
336a296
>
343d306
<      null;
347c307
<      STATIC_POOL( 7 ).ENTRY1 ;
>      STATIC_POOL( ).ENTRY1 ;
363,364c323,324
<      PROCEDURE_1( DV21 => 1 ) ;
<      TASK_A.ENTRY_A( DV11 => 1 , DV12 => ADV12 ) ;
>      PROCEDURE_1( DV21 => );
>      TASK_A.ENTRY_A( DV11 => , DV12 => );
380c340
<      DYN_POOL( 6 ).ENTRY1 ;
>      DYN_POOL( ).ENTRY1 ;
383c343
<      delay 1000.0;-- FINISH delay statement code.
>      delay -- FINISH delay statement code.
388c348
<      DYN_POOL( 5 ).ENTRY1 ;
>      DYN_POOL( ).ENTRY1 ;
392,394c352,354
<      null;
<      end select;
<      PROCEDURE_1( DV21 => 1 ) ;
>      end select;
>      PROCEDURE_1( DV21 => );
402d363
<      null;
405c363

```

```

<          STAT_TASK.ENTRY1( DV3 => ADV3 , DATA_VALUE_4 => 1, DATA_VALUE_5 => ADV5 ) ;
>          STAT_TASK.ENTRY1( DV3 =>      , DATA_VALUE_4 =>      , DATA_VALUE_5 =>      ) ;
408c366
<          delay 250.0; -- FINISH delay statement code.
>          delay -- FINISH delay statement code.
412c370
<          DYN_POOL( 4 ).ENTRY1 ;
>          DYN_POOL(      ).ENTRY1 ;
429d389
< null;
452d411
<          COUNT : INTEGER := 5;
459,460c415,416
<          when COUNT > 0 =>
<          accept S_ENTRY_1( DV19 : in DATA_TYPE_19; DV20 : in DATA_TYPE_20 ) do null;
>          when      =>
>          accept S_ENTRY_1( DV19 : in DATA_TYPE_19; DV20 : in DATA_TYPE_20 ) do
466c422
<          accept S_ENTRY_2( DV16 : out DATA_TYPE_16; DV17 : out DATA_TYPE_17 ) do null;
>          accept S_ENTRY_2( DV16 : out DATA_TYPE_16; DV17 : out DATA_TYPE_17 ) do
472c428
<          delay 10.0; -- delay code
>          delay -- delay code
478c434
<          accept S_ENTRY_4 do null;
>          accept S_ENTRY_4 do
486c442
<          accept S_ENTRY_3 do null;
>          accept S_ENTRY_3 do
492,494c448,450
<          accept ENTRY_5 do null;
<          end ENTRY_5 ;
<          INST_PACK.PROC2( DV15 => 1 ) ;
>          accept ENTRY_5 do
>          end ENTRY_5 ;
>          INST_PACK.PROC2( DV15 =>      ) ;
497,499c453,455
<          TASK2.S_ENTRY_2( DV16 => ADV16 , DV17 => ADV17 ) ;
<          TASK2.S_ENTRY_1( DV19 => 1 , DV20 => 1 ) ;
<          INST_PACK.PROC1( DV18 => ADV18 ) ;
>          TASK2.S_ENTRY_2( DV16 =>      , DV17 =>      ) ;
>          TASK2.S_ENTRY_1( DV19 =>      , DV20 =>      ) ;
>          INST_PACK.PROC1( DV18 =>      ) ;

```

APPENDIX F SINGLE CHANNEL S-ADA CODE

```

-----
--DESIGNER LOGIN NAME:      vandkw
--DESIGN DIRECTORY:        /e/vandkw/CAEDE_1.2/designs/chan_tst1.2:
--DESIGN VERSION NUMBER:    3
--TIME-DATE:               Mon Aug 25 20:34:41 1986
--CODEGENERATOR VERSION:    ASCGEN 1.2
-----

--|||||
--Package Spec: main_package
--|||||
package main_package is
end main_package;

--|||||
--Package Body: main_package
--|||||
package body main_package is

    --|||||
    --Specs for Generic Packages from template: pkg_temp_1
    --|||||
    generic package pkg_temp_1_1 is
    end pkg_temp_1_1;

    --|||||
    --Specs for task types from template: tsk_temp_1
    --|||||
    task type tsk_temp_1_1 is

        -----
        --FINISH declaration of entry: put
        -----

        entry put ;
    end tsk_temp_1_1 ;
    tsk_temp_1_1_ACCESS is access tsk_temp_1_1 ;
    task type tsk_temp_1_2 is

        -----
        --FINISH declaration of entry: put
        -----

        entry put( code : in char) ;

```


[illegible]

```
--=====
--Package Spec: sing_chan_1_pkg_4
--=====
package sing_chan_1_pkg_4 is

    =====
    --Spec for visible procedure: proc_41
    =====

    procedure proc_41 ;
end sing_chan_1_pkg_4 ;

-----
--Declaration of Static-taskpool-instance: sing_chan_1_st_pool_1
-----
-- SPECIFY number of tasks in pool
MAX_sing_chan_1_st_pool_1 : constant INTEGER := ;
sing_chan_1_st_pool_1 : array( 1..MAX_sing_chan_1_st_pool_1 ) of tsk_temp_1_2 ;

-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
--End of Single-Channel Declaration: sing_chan_1
--{>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

--=====
--Package Spec: pk_1
--=====
package pk_1 is
end pk_1 ;

--:::~::~::::::::::::::::::
--Bodies for generic packages from template: pkg_temp_1
--:::~::~::::::::::::::::::
package body pkg_temp_1_1 is
end pkg_temp_1_1 ;

--//////////////////////////////////////
--Bodies for Task Types from template:tsk_temp_1
--//////////////////////////////////////
task body tsk_temp_1_1 is
begin

    -----
    --FINISH accept code for: put
    -----

accept put do
end put ;
```

```
end tsk_temp_1_1 ;
```

```
task body tsk_temp_1_2 is
begin
```

```
-----
--FINISH accept code for: put
-----
```

```
accept put( code : in char) do
end put ;
end tsk_temp_1_2 ;
```

```
-----
--Package Body: pk_2
-----
package body pk_2 is
```

```
-----
--Spec, body for visible procedure: proc_21
-----
```

```
procedure proc_21 is
-- make necessary declarations
begin
-- FINISH body code
end proc_21 ;
end pk_2;
```

```
-----
--Single-Channel Body: sing_chan_1
-----
```

```
-----
--Spec, body for procedure: sing_chan_1_proc_1
-----
```

```
procedure sing_chan_1_proc_1 is
-- make necessary declarations
begin
-- FINISH body code
-- DYNAMIC TASKPOOL CREATION code!
for I in sing_chan_1_dyn_pool_1'RANGE loop
sing_chan_1_dyn_pool_1(I) := new tsk_temp_1_1 ;
end loop;
pk_2.proc_21 ;
-- FINISH TASKPOOL call code!
sing_chan_1_dyn_pool_1( ).put ;
sing_chan_1_tsk_inst_1.put ;
```

[illegible]

APPENDIX 6 CHANNEL INSTANCE S-ADA CODE

```

-----
--DESIGNER LOGIN NAME:      vandkw
--DESIGN DIRECTORY:         /e/vandkw/CAEDE_1.2/designs/chan_inst1.2x
--DESIGN VERSION NUMBER:    3
--TIME-DATE:                Mon Aug 25 19:24:51 1986
--CODEGENERATOR VERSION:    ASCGEN 1.2
-----

--Package Spec: main_package
package main_package is
end main_package;

--Package Body: main_package
package body main_package is

    --Specs for task types from template: tsk_temp_1
    task type tsk_temp_1_1 is
        -----
        --FINISH declaration of entry: put
        -----
        entry put ;
    end tsk_temp_1_1 ;
    tsk_temp_1_1_ACCESS is access tsk_temp_1_1 ;
    task type tsk_temp_1_2 is
        -----
        --FINISH declaration of entry: put
        -----
        entry put( letter : in char) ;
    end tsk_temp_1_2 ;
    tsk_temp_1_2_ACCESS is access tsk_temp_1_2 ;

--Package Spec: pk_1

```

```
package pk_1 is
end pk_1 ;
```

```

--Package Spec: pk_2
package pk_2 is

```

```

-----
--Spec for visible procedure: proc_21
-----
procedure proc_21 ;
end pk_2 ;

```

[illegible][illegible]

```

-----
--Declaration of static-task-instance: ch1_1_tsk_inst_1
-----
ch1_1_tsk_inst_1 : task temp_1_1 ;

```

[illegible][illegible][illegible][illegible]

```
begin
```

```
-----
--FINISH accept code for: put
-----
```

```
accept put do
end put ;
end tsk_temp_1_1 ;
```

```
task body tsk_temp_1_2 is
begin
```

```
-----
--FINISH accept code for: put
-----
```

```
accept put( letter : in char) do
end put ;
end tsk_temp_1_2 ;
```

```
-----
--Package Body: pk_1
-----
```

```
-----
package body pk_1 is
begin
```

```
chi_1_tsk_inst_1.put ;
pk_2.proc_21 ;
st_tsk_3.put( letter => ) ;
st_tsk_2.put ;
end pk_1;
```

```
-----
--Package Body: pk_2
-----
```

```
-----
package body pk_2 is
```

```
-----
--Spec, body for visible procedure: proc_21
-----
```

```
-----
procedure proc_21 is
```

```
    -- make necessary declarations
begin
```

```
    -- FINISH body code
end proc_21 ;
```

```
end pk_2;
```

```
-----
--Channel-Instance Body: chi_1
-----
```

APPENDIX G

[illegible]

—)))))))))))))))))))))))))))))))))

—End of Channel-Instance Body: ch1_1

```
end main_package;
```


APPENDIX H ASCGEN CODE LISTINGS

This Appendix lists the files that constitute ASCGEN, in alphabetical order. The root file that initializes ASCGEN is codegen_var. The execution of ASCGEN can be traced starting with this file.

H.1 ACTION CHAINS

```
get_unique_key(KEY):-  
    unique_key(KEY),  
    retract(unique_key(KEY)),  
    NEXT_KEY is KEY + 1,  
    asserta(unique_key(NEXT_KEY)),  
    !.
```

```
get_unique_key(1):-  
    asserta(unique_key(2)).
```

```
/*
```

```
*****
```

```
parse_action_chains(  
    KEY, a unique search key, to be used when  
        cg facts are searched for, in parameter,  
    ACTION_CHAIN_SOURCE_IDL, in parameter.  
    ENV_STACK, in parameter.  
).
```

DESCRIPTION:

This procedure is called to parse all action chains originating from the source idl. Codegeneration facts will be asserted for each interaction-chain or creation-chain syntactic category parsed. Error codegeneration facts will be asserted if parse errors occur. Completion of a syntactic category parse should result in backtracking to parse the next action chain until there are no more.

```
*****
```

```
*/
```

```
/* Parse potential interaction-chains and creation-chains */
```

```
parse_action_chains(
    KEY,
    ACTION_CHAIN_SOURCE_IDL,
    ENV_STACK
):-
    reference(
        [interaction : _],
        ACTION_CHAIN_SOURCE_IDL,
        DEST_IDL,
        DETAILS
    ),
    continue_parse_action_chains(
        KEY,
        DEST_IDL,
        DETAILS,
        ENV_STACK
    ).
```

```
/* Parse potential simple creation-chains */
```

```
parse_action_chains(
    KEY,
    CREATOR_IDL,
    -
):-
    reference(
        [created_by : _],
        CREATOR_IDL,
        CREATED_IDL,
        -
    ),
    asserta(creation_chain( KEY, CREATED_IDL)),
    fail.
```

```
parse_action_chains( _, _, _).
```

```
/* Rule to catch many different cases of channel related
node_node_interactions are required including:
```

1. INTERACTIONS DIRECTED TOWARDS CHANNELS:

- 1.1 Port CP to tx_port_plug;
- 1.2 Connecting port tx_port_plug to single channel rx_port plug;
- 1.3 Connecting port tx_port_plug to channel_instance rx_port_plug;

1.4 Connecting channel_instance rx_port_plug to channel_template
rx_port_plug;

2. INTERACTIONS DIRECTED FROM CHANNELS:

- 2.1 Channel_template internal CP to port_conn tx_port_plug;
- 2.2 Connecting channel_template tx_port_plug to channel_instance
tx_port_plug and:
- 2.3 Connecting channel_instance tx_port_plug to port rx_port_plug;
or
- 2.4 Single_channel internal CP to port_conn tx_port_plug and:
- 2.5 Connecting single_channel tx_port_plug to port rx_port_plug;

3. CHANNEL INTERNAL NODE-NODE INTERACTIONS

3.1 rx_port_plug to tx_port_plug interaction.

*/

/* Rule 1.1, 2.1 and all other connection point parse situations */

```
continue_parse_action_chains(
    KEY,
    [ connection_point, NAME : _ ],
    [ TYPE, ACCESS_MODE, _ ],
    [ _, environment( PREFIX, [ _, EXT_PARENT_NAME, EXT_PARENT_ANCESTRY ] )
      : ENV_STACK_TAIL ]
):-
    reference(
        [ interaction : _ ],
        [ connection_point, NAME, [ EXT_PARENT_NAME
          : EXT_PARENT_ANCESTRY ] ],
        DEST_IDL,
        [ TYPE, ACCESS_MODE, _ ]
    ),
    !,
    continue_parse_action_chains(
        KEY,
        DEST_IDL,
        [ TYPE, ACCESS_MODE, _ ],
        [ environment( PREFIX, [ _, EXT_PARENT_NAME, EXT_PARENT_ANCESTRY ] )
          : ENV_STACK_TAIL ]
    ).
```

/* Rule 2.2, 2.3, 2.5 */

```

continue_parse_action_chains(
    KEY,
    [ tx_port_plug, SRC_PLUG_NAME, port_conn, PORT_CONN_TYPE : _ ],
    [ TYPE, ACCESS_MODE, _ ],
    [
        _ ,
        /* single-channel or channel-instance environment */
        environment(PREFIX, CHANNEL_IDL)
        : ENV_STACK_TAIL
    ]
):-
    /* get correct port name to remove potential ambiguity */
    (
        /* port is user */
        reference(
            CHANNEL_IDL,
            [ port, PORT_NAME, PORT_CONN_TYPE, ANCESTRY ],
            _ ,
            [ ]
        )
        ; /* port is provider */
        reference(
            CHANNEL_IDL,
            _ ,
            [ port, PORT_NAME, PORT_CONN_TYPE, ANCESTRY ],
            [ ]
        )
    ),
    reference(
        [ interaction : _ ],
        [
            rx_port_plug,
            SRC_PLUG_NAME,
            PORT_NAME,
            PORT_CONN_TYPE,
            [ PORT_NAME : ANCESTRY ]
        ],
        DEST_IDL,
        [ TYPE, ACCESS_MODE, _ ]
    ),
    !,
    !,
    continue_parse_action_chains(
        KEY,
        DEST_IDL,
        [ TYPE, ACCESS_MODE, _ ],
        [ environment( PREFIX, CHANNEL_IDL ) : ENV_STACK_TAIL ]
    )

```

```

    ).

/* Rule 1.2 */
continue_parse_action_chains(
    KEY,
    [ tx_port_plug, SRC_PLUG_NAME, PORT_NAME, PORT_TYPE,
      [ _ : PORT_ANCESTRY]],
    [ TYPE, ACCESS_MODE, _],
    [
      /* environment external to single_channel or channel-instance */
      environment(PREFIX, PREV_ENV_IDL)
      : ENV_STACK_TAIL
    ]
):-
    /* get channel IDL */
    (
      /* port is user */
      reference(
        [channel, CH_NAME, CH_ANCESTRY],
        [ port, PORT_NAME, PORT_TYPE, PORT_ANCESTRY],
        _
      )
    ; /* port is provider */
      reference(
        [channel, CH_NAME, CH_ANCESTRY],
        [ port, PORT_NAME, PORT_TYPE, PORT_ANCESTRY],
        _
      )
    ),
    abstraction(
      [channel_access, CH_NAME, CH_ANCESTRY],
      [_, VAR_IF_SINGLE_CHANNEL]
    ),
    var( VAR_IF_SINGLE_CHANNEL ),
    reference(
      [ interaction : _],
      [
        rx_port_plug,
        SRC_PLUG_NAME,
        PORT_CONN_NAME,
        PORT_TYPE,
        [ PORT_CONN_NAME, CH_NAME : CH_ANCESTRY ]
      ],
    ),

```

```

        DEST_IDL,
        [ TYPE, ACCESS_MODE, _ ]
    ),
    catenate_prefix(PREFIX, CH_NAME, NEW_PREFIX),
    !,
    continue_parse_action_chains(
        KEY,
        DEST_IDL,
        [ TYPE, ACCESS_MODE, _ ],
        [
            environment( NEW_PREFIX, _ ), /* extra argument to carry
                NEW_PREFIX for interactions targetted on channel
                internals */
            environment( PREFIX, [ channel, CH_NAME, CH_ANCESTRY ] )
            ! ENV_STACK_TAIL
        ]
    ).

/* Rule 4.3, 1.4 */
continue_parse_action_chains(
    KEY,
    [ t:port_plug, SRC_PLUG_NAME, PORT_NAME, PORT_TYPE,
      [ _ ! PORT_ANCESTRY ] ],
    [ TYPE, ACCESS_MODE, _ ],
    [
        /* environment external to single_channel or channel-instance */
        environment(PREFIX, PREV_ENV_IDL)
        ! ENV_STACK_TAIL
    ]
):-
    /* get channel IDL */
    (
        /* port is user */
        reference(
            [channel, CH_NAME, CH_ANCESTRY],
            [ port, PORT_NAME, PORT_TYPE, PORT_ANCESTRY],
            _
        )
    )
    ; /* port is provider */
    reference(
        [channel, CH_NAME, CH_ANCESTRY],
        _
        [ port, PORT_NAME, PORT_TYPE, PORT_ANCESTRY],
        [ ]
    )

```

```

    ),
    abstraction(
        [channel_access, CH_NAME, CH_ANCESTRY],
        [_, TEMPLATE_NAME ]
    ),
    reference(
        [ interaction : _ ],
        [
            rx_port_plug,
            SRC_PLUG_NAME,
            PORT_CONN_NAME,
            PORT_TYPE,
            [ PORT_CONN_NAME, TEMPLATE_NAME, _ ]
        ],
        DEST_IDL,
        [ TYPE, ACCESS_MODE, _ ]
    ),
    catenate_prefix(PREFIX, CH_NAME, NEW_PREFIX),
    ',
    continue_parse_action_chains(
        KEY,
        DEST_IDL,
        [ TYPE, ACCESS_MODE, _ ],
        [
            environment( NEW_PREFIX, _), /* Extra argument to carry
                channel internal prefix for interactions targetted on
                channel internals. */
            environment(PREFIX, [ channel, CH_NAME, CH_ANCESTRY ])
            : ENV_STACK_TAIL
        ]
    ),
    ),

continue_parse_action_chains(
    KEY,
    [ connection_point, NAME : _ ],
    [ wait, static, _ ],
    [ _, environment( PREFIX, [ _, EXT_PARENT_NAME, EXT_PARENT_ANCESTRY ])
        : ENV_STACK_TAIL ]
):-
    reference(
        [ created_by : _ ],
        [ connection_point, NAME, [ EXT_PARENT_NAME
            : EXT_PARENT_ANCESTRY]],
        CREATED_IDL,
        -
    )

```

```
    ),
    asserta(creation_chain(KEY, CREATED_IDL)),
    ', fail.

/* DESTINATION IDL Rule */

continue_parse_action_chains(
    KEY,
    DEST_IDL,
    DETAILS,
    [ environment( PREFIX, _ ) : _ ]
):-
    asserta(interaction_chain(
        KEY,
        interaction_call(DEST_IDL, DETAILS, PREFIX)
    )
    ), !, fail.
```


H.2 CG COMMENTS

```

/*****
$ FILE_NAME:      cg_comments

$ PROJECT_NAME:   CADA Skeleton Ada Codegenerator

$ DESCRIPTION:    This file contains code which generates headers and
                  comment messages.

$ PROCEDURES IN FILE:
    border_chars,
    catenate_prefix,
    cg_comments,
    cg_comments_2,
    info_required,
    codegenerator_version,
    message,

$ HISTORY:

    date:          version:        by:          reason:
    -----
    15 Nov 84      CADA 1.1         K. Van Der Loo  freeze CADA 1.1
    5 Jan 85       CAEDE 1.X        "              "      mods to recover
space used in I/O: cg_comments_2.
    20 Jan 85      "              "              "      addition of different
border chars for each icon: border_chars, mods to cg_comments(major_header...
to control where info text is placed, addition of info_required,
codegenerator_version.
    13 June 85     ASCGEN/1.2X      "              "      addition of extra
parameter: PREFIX to handle renaming if objects nested
within channels.
    19 June 85     ASCGEN/1.2X      "              "      addition of procedure
catenate_prefix, and info required and message for single_channel_declaration,
single_channel_body.
    3 July 85      "              "              "      addition of package-
instantiation message.

*****/

cg_comments(design_header):-
    design_info(LOGIN_NAME,DESIGN_PATHNAME,

```

```

        VERSION_NUMBER, TIME_DATE),
    codegenerator_version(VERSION),
    begin_header(80,"*",":"),
    write_atom('DESIGNER LOGIN NAME: " ' & *),
    write_atom(LOGIN_NAME),
    newline,
    write_atom('DESIGN DIRECTORY:      '),
    write_atom(DESIGN_PATHNAME),
    newline,
    write_atom('DESIGN VERSION NUMBER:  '),
    write_atom(VERSION_NUMBER),
    newline,
    write_atom('TIME-DATE:              '),
    write_atom(TIME_DATE),
    newline,
    write_atom('CODEGENERATOR VERSION:  '),
    write_atom(VERSION),
    newline,
    end_header(80,"*",":"),
    !, fail.

cg_comments(W,X,Y,Z):-
    cg_comments_2(W,X,Y,Z).
cg_comments(_,_,_).

cg_comments_2(major_header,KIND,[ITEM_KIND,NAME : IDL_TAIL], PREFIX):-
    message(KIND,MESSAGE),
    border_chars(ITEM_KIND,LENGTH,HORIZ_CHAR,VERT_CHAR),
    newline,
    begin_header(LENGTH,HORIZ_CHAR,VERT_CHAR),
    write_atom(MESSAGE),
    write_atom(PREFIX), write_atom(NAME),
    newline,
    (((information([ITEM_KIND,NAME : IDL_TAIL],INFORMATION),
        info_required(KIND)) ->
        print_list(INFORMATION)
    );true
    ),
    end_header(LENGTH,HORIZ_CHAR,VERT_CHAR),
    !, fail.

cg_comments_2(major_footer,KIND,[ITEM_KIND,NAME : IDL_TAIL], PREFIX):-
    message(KIND,MESSAGE),
    border_chars(ITEM_KIND,LENGTH,HORIZ_CHAR,VERT_CHAR),
    newline,
    begin_header(LENGTH,HORIZ_CHAR,VERT_CHAR),

```

```

        write_atom(MESSAGE),
        write_atom(PREFIX), write_atom(NAME),
        newline,
        end_header(LENGTH, HORIZ_CHAR, VERT_CHAR),
        !, fail.

cg_comments_2(minor_footer, KIND, [_NAME : _], PREFIX):-
    message(KIND, MESSAGE),
    begin_header(60, "-", "!"),
    write_atom(MESSAGE),
    write_atom(PREFIX), write_atom(NAME),
    newline,
    end_header(60, "-", "!"),
    !, fail.

cg_comments_2(minor_header, KIND, [ITEM_KIND, NAME : IDL_TAIL], PREFIX):-
    message(KIND, MESSAGE),
    begin_header(60, "-", "!"),
    write_atom(MESSAGE),
    write_atom(PREFIX), write_atom(NAME),
    newline,
    ((information([ITEM_KIND, NAME : IDL_TAIL], INFORMATION) ->
        print_list(INFORMATION)
    ));true
),
    end_header(60, "-", "!"),
    !, fail.

cg_comments_2(simple_comment, KIND, [ITEM_KIND, NAME : IDL_TAIL], PREFIX):-
    message(KIND, MESSAGE),
    begin_comment,
    write_atom(MESSAGE),
    write_atom(PREFIX), write_atom(NAME),
    newline,
    ((information([ITEM_KIND, NAME : IDL_TAIL], INFORMATION) ->
        print_list(INFORMATION)
    ));true
),
    end_comment, !, fail.

border_chars(procedure, 60, "=", "!").
border_chars(pack_ext_procedure, 60, "=", "!").
border_chars(package, 60, "!", "!").
border_chars(task, 60, "/", "!").
border_chars(task_pool, 60, "#", "!").

```

```

border_chars(inv_data_area,60,"D","!").
border_chars(vis_data_area,60,"D","!").
border_chars(pack_ext_proc, 60, "P", "!").
border_chars(procedure, 60, "P", "!").
border_chars(task_entry, 60, "E", "!").
border_chars(timed_entry, 60, "T", "!").
border_chars(channel, 60, ">", "!").
border_chars(package_template, 60, ":", "!").
border_chars(task_template, 60, "/", "!").
border_chars(_,60,"?", "!").

```

```

codegenerator_version('ASCGEN 1.2').

```

```

info_required(pack_ext_procedure_declaration).
info_required(procedure_body).
info_required(package_declaration).
info_required(data_area_declaration).
info_required(single_task_declaration).
info_required(static_task_instantiation).
info_required(dynamic_task_declaration).
info_required(static_task_pool_instantiation).
info_required(dynamic_task_pool_declaration).
info_required(task_template_declaration).
info_required(package_instantiation).
info_required(package_template_declaration).
info_required(single_channel_declaration).
info_required(single_channel_body).
info_required(channel_instance_declaration).
info_required(end_channel_instance_declaration).
info_required(channel_instance_body).
info_required(end_channel_instance_body).

```

```

message(pack_ext_procedure_declaration,'Spec for visible procedure: ').
message(pack_ext_procedure_body,
    'Spec, body for visible procedure: ').
message(procedure_declaration,'Spec for procedure: ').
message(procedure_body,'Spec, body for procedure: ').
message(entry_declaration,'FINISH declaration of entry: ').
message(accept,'FINISH accept code for: ').
message(timed_entry_declaration,'TIMEOUT pseudo-entry: ').
message(timed_accept,'FINISH TIMEOUT pseudo-entry: ').
message(guard_declaration,'GUARD assigned is: ').
message(guard_code,'FINISH guard: ').
message(task_template_declaration,
    'Specs for task types from template: ').

```

```

message(single_task_declaration, 'Spec for task: ').
message(static_task_instantiation, 'Declaration of static-task-instance: ').
message(dynamic_task_declaration, 'Declaration of Dynamic-task-access: ').
message(static_task_pool_instantiation, 'Declaration of Static-taskpool-instance: ').
message(dynamic_task_pool_declaration, 'Declaration of Dynamic-taskpool-access: ').
message(task_body, 'Task body: ').
message(task_template_body,
      'Bodies for Task Types from template: ').
message(package_template_declaration,
      'Specs for Generic Packages from template: ').
message(package_declaration, 'Package Spec: ').
message(package_instantiation, 'Declaration for generic-instance: ').
message(package_body, 'Package Body: ').
message(package_template_body,
      'Bodies for generic packages from template: ').
message(single_channel_declaration, 'Single-Channel Declaration: ').
message(end_single_channel_declaration, 'End of Single-Channel Declaration: ').
message(single_channel_body, 'Single-Channel Body: ').
message(end_single_channel_body, 'End of Single-Channel Body: ').
message(channel_instance_declaration, 'Channel-Instance Declaration: ').
message(end_channel_instance_declaration,
      'End of channel-instance Declaration: ').
message(channel_instance_body, 'Channel-Instance Body: ').
message(end_channel_instance_body, 'End of Channel-Instance Body: ').
message(select_accept_declaration, 'Begin select accept: ').
message(end_select_accept_declaration, 'End select accept: ').
message(data_area_declaration, 'MAKE declarations defining data area: ').
message(data_area_access, 'ADD code for accessing data area: ').
message(call_actual_parameters, 'DECLARE actual parameters in call to: ').

```

```
/*
```

```
*****
```

NAME:

catenate_prefix(

 PREFIX an atom, either '' or '<name>_', in parameter.

 NAME, an atom to catenate to the end of PREFIX, in parameter.

 NEW_PREFIX, an atom, the catenation of the above parameters
 separated by '_', out parameter.

).

DESCRIPTION:

 This procedure catenates atoms separating them by underlines
 and appends a trailing underline to the result.

*/

catenate_prefix('', '', '').

catenate_prefix('', B, B_):-
 name(B, BS),
 append(BS, "_", B_S),
 name(B_, B_S),
 !.

catenate_prefix(A_, '', A_).

catenate_prefix(A_, B, A_B_):-
 name(A_, A_S),
 name(B, BS),
 append(A_S, BS, A_BS),
 append(A_BS, "_", A_B_S),
 name(A_B_, A_B_S),
 !.

H.3 CODEGEN VAX

/******

\$ FILE_NAME: codegen_vax

\$ PROJECT_NAME: CADA Skeleton Ada Codegenerator

\$ DESCRIPTION: This file contains the "mainline" for initializing and running the codegenerator. For portability, the pathname for files must reflect the actual location and must be complete since prolog may be invoked from many users directories and prolog only understands local directory file names or full pathnames.

\$ PROCEDURES IN FILE:
start,
consult command line

HISTORY:

date:	version:	by:	reason:
15 Nov 84	CADA 1.1	K. Van Der Loo	freeze CADA 1.1
5 Jan 85	CADA 1.X	" "	experimental: To
21 June 85	ASCGEN 1.2X	" "	modify code for this

establish I/O time.
version of the codegenerator.

*/

:- ['/e/vandkw/CAEDE_1.2/ASCGEN_1.2X/files_vax_1.2X'].
start:-

```
output_file(F),
tell(F),
(
display('
```

```
display('Initial statistics just before main spec invoked
```

```
statistics,
display('

'),
make_inst_externals_facts,
(cg_comments(design_header); true),
declaration([package,main_package,[], '' , []),
display('Main declaration completed.

'),
body([package,main_package,[], '' , []),
display('Main body completed.

'),
display('Completion statistics:

'),
statistics,
told
) ; display('failure during code generation ' ),
told.
```


H.4 DECL PARTS 3.9

```

/*****
$    FILE_NAME:      decl_parts_3.9

$    PROJECT_NAME:   CAEDE/1.2 ASCGEN Ada Skeleton Codegenerator

$    DESCRIPTION:    This file contains code to generate ada syntax
                    code as given by the procedure names.

$    PROCEDURES IN FILE:
                    basic_declarative_item,
                    body,
                    declarative_part,
                    later_declarative_item,

$    HISTORY:

        date:          version:          by:          reason:
        -----          -
        15 Nov 84       CADA 1.1          K. Van Der Loo  freeze CADA 1.1
        15 June 85      ASCGEN/1.2X       "            "    start addition of
code to handle channels and templates.
        21 June 85      "                "            "    addition of channel
body code.
        4 July 85       "                "            "    removal of non-instance
ids parameter from calls to get_instances.

-----
*/

declarative_part(
    [INT_TASK_KIND : INT_IDL_TAIL ],
    [
        environment(
            EXT_PREFIX,
            [ EXT_TASK_KIND : EXT_IDL_TAIL])
        : ENV_STACK
    ]
):-
    (
        ( INT_TASK_KIND = task ) ; ( INT_TASK_KIND = task_template )
    ),
    abstraction([task externals : EXT_IDL_TAIL],

```

```

        [ EXTERNAL_COMPONENT_IDLS ] ),
    abstraction([task_internals : INT_IDL_TAIL], [INTERNAL_COMPONENT_IDLS,
        INTERNAL_OBJECT_IDLS]),
    get_instances(
        INTERNAL_OBJECT_IDLS,
        [],
        CLASSIFIED_G_INSTANCES
    ),
/*--- generic declaration code will be put here' */
    generic_declarations(
        [INT_TASK_KIND : INT_IDL_TAIL],
        CLASSIFIED_G_INSTANCES
    ),
    basic_declarative_item(INTERNAL_COMPONENT_IDLS, ''),
    basic_declarative_item(INTERNAL_OBJECT_IDLS, '',
        CLASSIFIED_G_INSTANCES ),
/*--- generic body code will be put here' */
    generic_bodies(
        [INT_TASK_KIND : INT_IDL_TAIL],
        [
            environment(
                EXT_PREFIX,
                [ EXT_TASK_KIND : EXT_IDL_TAIL])
            : ENV_STACK
        ]
    ),
    later_declarative_item(
        INTERNAL_COMPONENT_IDLS,
        '',
        [
            environment(
                EXT_PREFIX,
                [ EXT_TASK_KIND : EXT_IDL_TAIL])
            : ENV_STACK
        ]
    ),
    later_declarative_item(
        INTERNAL_OBJECT_IDLS,
        '',
        [
            environment(
                EXT_PREFIX,
                [ EXT_TASK_KIND : EXT_IDL_TAIL])
            : ENV_STACK
        ]
    )

```

```

later_declarative_item(
    EXTERNAL_COMPONENT_IDLS,
    '',
    [
        environment(
            EXT_PREFIX,
            [ EXT_TASK_KIND : EXT_IDL_TAIL ]
        : ENV_STACK
    ]
).

declarative_part(
    [ INT_PACKAGE_KIND : INT_IDL_TAIL ],
    [ environment(
        EXT_PREFIX,
        [ EXT_PACKAGE_KIND : EXT_IDL_TAIL ]
    : ENV_STACK
    ]
):-
    (
        ( INT_PACKAGE_KIND = package )
    ;
        ( INT_PACKAGE_KIND = package_template )
    ),
    abstraction(
        [ package externals : EXT_IDL_TAIL ],
        [ EXTERNAL_COMPONENT_IDLS ]
    ),
    abstraction(
        [ package internals : INT_IDL_TAIL ],
        [ INTERNAL_COMPONENT_IDLS, INTERNAL_OBJECT_IDLS ]
    ),
    get_instances(
        INTERNAL_OBJECT_IDLS,
        [],
        CLASSIFIED_G_INSTANCES
    ),
    /*--- task type and generic declaration code will be put here! */
    generic_declarations(
        [ INT_PACKAGE_KIND : INT_IDL_TAIL ],
        CLASSIFIED_G_INSTANCES
    ),
    task_type_declarations(
        [ INT_PACKAGE_KIND : INT_IDL_TAIL ],
        CLASSIFIED_G_INSTANCES
    ),
    basic_declarative_item(INTERNAL_COMPONENT_IDLS, ''),

```

```

basic_declarative_item(INTERNAL_OBJECT_IDLS, '',
    CLASSIFIED_6_INSTANCES ),
/*----- generic body and task type body code will be put here */
generic_bodies(
    [ INT_PACKAGE_KIND : INT_IDL_TAIL ],
    [ environment(
        EXT_PREFIX,
        [EXT_PACKAGE_KIND : EXT_IDL_TAIL ] )
      : ENV_STACK ]
),
task_type_bodies(
    [ INT_PACKAGE_KIND : INT_IDL_TAIL ],
    [ environment(
        EXT_PREFIX,
        [EXT_PACKAGE_KIND : EXT_IDL_TAIL ] )
      : ENV_STACK ]
),
later_declarative_item(
    INTERNAL_COMPONENT_IDLS,
    '',
    [ environment(
        EXT_PREFIX,
        [EXT_PACKAGE_KIND : EXT_IDL_TAIL ] )
      : ENV_STACK ]
),
later_declarative_item(
    INTERNAL_OBJECT_IDLS ,
    '',
    [ environment(
        EXT_PREFIX,
        [EXT_PACKAGE_KIND : EXT_IDL_TAIL ] )
      : ENV_STACK ]
),
later_declarative_item(
    EXTERNAL_COMPONENT_IDLS,
    '',
    [ environment(
        EXT_PREFIX,
        [EXT_PACKAGE_KIND : EXT_IDL_TAIL ] )
      : ENV_STACK ]
),

basic_declarative_item([], _):-
    ' /* zero_or_more termination condition */

```

```

basic_declarative_item([IDL : IDL_LISTS_TAIL ], PREFIX):-
    /* Some items will not require basic declarations but the goal
       must succeed anyway. */
    declaration(IDL, PREFIX),
    basic_declarative_item(IDL_LISTS_TAIL, PREFIX),
    !.

basic_declarative_item([_ : IDL_LISTS_TAIL ], PREFIX):-
    basic_declarative_item(IDL_LISTS_TAIL, PREFIX),
    !.

basic_declarative_item([], _, _):-
    !. /* zero_or_more termination condition */

basic_declarative_item([IDL : IDL_LISTS_TAIL ], PREFIX, CGI):-
    /* Some items will not require basic declarations but the goal
       must succeed anyway. */
    declaration(IDL, PREFIX, CGI),
    basic_declarative_item(IDL_LISTS_TAIL, PREFIX, CGI),
    !.

basic_declarative_item([_ : IDL_LISTS_TAIL ], PREFIX, CGI):-
    basic_declarative_item(IDL_LISTS_TAIL, PREFIX, CGI),
    !.

later_declarative_item([], _, _):-
    !. /* zero_or_more termination condition */

later_declarative_item([ITEM_DESCRIPTOR_LIST : IDL_LISTS_TAIL ],
    PREFIX, ENV_STACK
):-
    body(ITEM_DESCRIPTOR_LIST, PREFIX, ENV_STACK),
    later_declarative_item(IDL_LISTS_TAIL, PREFIX, ENV_STACK),
    !.

later_declarative_item([_ : IDL_LISTS_TAIL ], PREFIX, ENV_STACK):-
    later_declarative_item(IDL_LISTS_TAIL, PREFIX, ENV_STACK),
    !.

/* Special handling of externals for generic body declarations */
body([pack_ext_procedure, NAME, [INST_NAME : IDL_TAIL ]], PREFIX, ENV_STACK):-
    abstraction([package_access, INST_NAME, IDL_TAIL ],[_ ,TEMPLATE_NAME]),
    atom(TEMPLATE_NAME),
    !.

```

```

    subprogram_body( [pack_ext_procedure, NAME, [TEMPLATE_NAME, _]], _,
        [environment(PREFIX, [pack_ext_procedure :
            [INST_NAME : IDL_TAIL]]) : ENV_STACK])).

body( [pack_ext_procedure : IDL_TAIL ], PREFIX, ENV_STACK):-
    subprogram_body( [pack_ext_procedure : IDL_TAIL ], _,
        [environment(PREFIX, [pack_ext_procedure : IDL_TAIL])
        : ENV_STACK])).

body( [procedure : IDL_TAIL ], PREFIX, ENV_STACK):-
    subprogram_body( [procedure : IDL_TAIL ], PREFIX,
        [environment( PREFIX, [procedure : IDL_TAIL])
        : ENV_STACK])).

body( [package : IDL_TAIL ], PREFIX, ENV_STACK):-
    !,
    abstraction([package_access : IDL_TAIL ],[_ ,TEMPLATE_NAME]),
    var(TEMPLATE_NAME), /* not an instantiation -> body */
    package_body(
        [package : IDL_TAIL ],
        [environment( PREFIX, [package : IDL_TAIL])
        : ENV_STACK]
    ).

body([task : IDL_TAIL ], PREFIX, ENV_STACK):-
    !,
    abstraction([task_access : IDL_TAIL ],[static,TEMPLATE_NAME]),
    var(TEMPLATE_NAME), /* not an instantiation, so it has a body */
    task_body(
        [task : IDL_TAIL ],
        [environment( PREFIX, [task : IDL_TAIL])
        : ENV_STACK]
    ).

body([channel, NAME : IDL_TAIL], PREFIX, ENV_STACK):-
    abstraction([channel_access, NAME : IDL_TAIL], [_ , TEMPLATE_NAME]),
    var(TEMPLATE_NAME).
    /* single channel body */
    cg_comments(major_header, single_channel_body, [channel,
        NAME : IDL_TAIL], PREFIX ),
    abstraction([channel_internals, NAME : IDL_TAIL ],
        [
            INTERNAL_COMPONENT_IDLS,
            INTERNAL_OBJECT_IDLS
        ]
    ),

```

```

catenate_prefix( PREFIX, NAME, NEW_PREFIX ),
begin_block,
later_declarative_item( INTERNAL_COMPONENT_IDLS, NEW_PREFIX,
    [ environment( PREFIX, [channel, NAME : IDL_TAIL] )
      : ENV_STACK ]
),
later_declarative_item( INTERNAL_OBJECT_IDLS, NEW_PREFIX,
    [ environment( PREFIX, [channel, NAME : IDL_TAIL] )
      : ENV_STACK ]
),
end_block,
cg_comments( major_header, end_single_channel_body, [channel,
    NAME : IDL_TAIL], PREFIX ),
newline,
'.

body([channel, NAME : IDL_TAIL], PREFIX, ENV_STACK):-
    abstraction([channel_access, NAME : IDL_TAIL], [_, TEMPLATE_NAME]),
    /* channel instance body */
    cg_comments( major_header, channel_instance_body, [channel,
        NAME : IDL_TAIL], PREFIX ),
    abstraction([channel_internals, TEMPLATE_NAME, []],
        [
            INTERNAL_COMPONENT_IDLS,
            INTERNAL_OBJECT_IDLS
        ]
    ),
    catenate_prefix( PREFIX, NAME, NEW_PREFIX ),
    begin_block,
    later_declarative_item( INTERNAL_COMPONENT_IDLS, NEW_PREFIX,
        [ environment( PREFIX, [channel, NAME : IDL_TAIL] )
          : ENV_STACK ]
    ),
    later_declarative_item( INTERNAL_OBJECT_IDLS, NEW_PREFIX,
        [ environment( PREFIX, [channel, NAME : IDL_TAIL] )
          : ENV_STACK ]
    ),
    end_block,
    cg_comments( major_footer, end_channel_instance_body, [channel,
        NAME : IDL_TAIL], PREFIX ),
    newline,
    '.

```

H.5 DECLARATIONS 3.1

```

/*****

```

```

$ FILE_NAME:      declarations_3.1

```

```

$ PROJECT_NAME:   CADA Skeleton Ada Codegenerator

```

```

$ DESCRIPTION:    This file contains code to generate the ada syntax
                  declarations for listed CAEDA objects.

```

```

$ PROCEDURES IN FILE:  declaration

```

```

$ HISTORY:

```

date:	version:	by:	reason:
15 Nov 84	CADA 1.1	K. Van Der Loo	freeze CADA 1.1.
21 Jan 85	CAEDE 1.X	" "	add cuts to recover
space.			
9 June 85	ASCGEN_1.2X	" "	add parameter Local_environment
prefix for channel internals renaming.			
15 June 85	ASCGEN_1.2X	" "	change number of
cg_comments parameters.			
23 Aug 86	ASCGEN_1.2X	" "	corrected comment bug
in timed entry declaration.			

```

*****
*/

```

```

declaration([task_entry,NAME : IDL_TAIL ],_):-
  cg_comments(minor_header,entry_declaration,
    [task_entry,NAME : IDL_TAIL ],''),
  /* Produce code for optional guard */
  (component([guard,GUARD_NAME,NAME : IDL_TAIL ],_) =>
    /* entry guarded */
    (
      begin_block,
      cg_comments(simple_comment,guard_declaration,
        [guard,GUARD_NAME,NAME : IDL_TAIL ],''),
      end_block
    ).; true /* entry unguarded */
  ),
  write_atom('entry '),write_atom(NAME),
  component([task_entry,NAME : IDL_TAIL ],

```



```

        [FORMAL_PARAMETER_SPECS_LIST]),
    formal_part(FORMAL_PARAMETER_SPECS_LIST),
    write_atom(' ;'),newline,'.

```

```

/* Special handling of externals for generic specifications: comments
associated with template while parameters with instance. */
declaration([vis_data_area, NAME, [INST_NAME:IDL_TAIL]], PREFIX):-
    abstraction([package_access, INST_NAME, IDL_TAIL ],[_ ,TEMPLATE_NAME]),
    atom(TEMPLATE_NAME),
    cg_comments(major_header,data_area_declaration,
        [vis_data_area, NAME, [TEMPLATE_NAME,_]], PREFIX),
    begin_block,
    write_atom('-- DATA AREA PARAMETERS('),
    data_area_parameters([vis_data_area , NAME, [INST_NAME:IDL_TAIL]]),
    write_atom(')'),
    newline,end_block,!.

```

```

declaration([vis_data_area : IDL_TAIL], PREFIX):-
    cg_comments(major_header,data_area_declaration,[vis_data_area :
        IDL_TAIL], PREFIX ),
    begin_block,
    write_atom('-- DATA AREA PARAMETERS('),
    data_area_parameters([vis_data_area : IDL_TAIL]),
    write_atom(')'),
    newline,end_block,!.

```

```

declaration([inv_data_area : IDL_TAIL], PREFIX):-
    cg_comments(major_header,data_area_declaration,[inv_data_area :
        IDL_TAIL], PREFIX ),
    begin_block,
    write_atom('-- DATA AREA PARAMETERS('),
    data_area_parameters([inv_data_area : IDL_TAIL]),
    write_atom(')'),
    newline,end_block,!.

```

```

/* Special handling of externals for generic specifications: comments
associated with template while parameters with instance. */
declaration([pack_ext_procedure, NAME,[INST_NAME:IDL_TAIL]], _):-
    abstraction([package_access, INST_NAME, IDL_TAIL ],[_ ,TEMPLATE_NAME]),
    atom(TEMPLATE_NAME),
    cg_comments(major_header,pack_ext_procedure_declaration,
        [pack_ext_procedure, NAME, [TEMPLATE_NAME,_]], ''),
    subprogram_specification([ pack_ext_procedure,NAME,
        [INST_NAME:IDL_TAIL]], ''),
    write_atom(' ;'),newline,!.

```

```

declaration([pack_ext_procedure : IDL_TAIL], _):-
    cg_comments(major_header, pack_ext_procedure_declaration,
        [pack_ext_procedure : IDL_TAIL], ''),
    subprogram_specification([ pack_ext_procedure : IDL_TAIL], ''),
    write_atom(' ;'), newline, !.

declaration([package, NAME : IDL_TAIL], PREFIX, CLASSIFIED_G_INSTANCES):-
    /* package instantiation */
    abstraction([package_access, NAME : IDL_TAIL], [_, TEMPLATE_NAME]),
    atomic(TEMPLATE_NAME),
    cg_comments(major_header, package_instantiation,
        [package, NAME : IDL_TAIL], PREFIX ),
    get_type_name(
        TEMPLATE_NAME,
        [package, NAME : IDL_TAIL],
        CLASSIFIED_G_INSTANCES,
        GENERIC_TYPE_NAME
    ),
    write_atom('package '), write_atom(PREFIX), write_atom(NAME),
    write_atom(' is new '), write_atom(GENERIC_TYPE_NAME), write_atom(' ;'),
    newline, !.

declaration([package, NAME : IDL_TAIL], PREFIX, _):-
    /* single package declaration */
    cg_comments(major_header, package_declaration, [package,
        NAME : IDL_TAIL], PREFIX ),
    package_specification([package, NAME : IDL_TAIL], PREFIX ),
    write_atom(' ;'), newline, !.

declaration([task, NAME : IDL_TAIL], PREFIX, CLASSIFIED_G_INSTANCES):-
    abstraction([task_access, NAME : IDL_TAIL], [dynamic, TEMPLATE_NAME]),
    /* a dynamic task instance access object declaration */
    cg_comments(major_header, dynamic_task_declaration,
        [task, NAME : IDL_TAIL], PREFIX ),
    get_type_name(
        TEMPLATE_NAME,
        [task, NAME : IDL_TAIL],
        CLASSIFIED_G_INSTANCES,
        TASK_TYPE_NAME
    ),
    write_atom(PREFIX), write_atom(NAME), write_atom(' : '),
    write_atom(TASK_TYPE_NAME), write_atom(' _ACCESS'), write_atom(' ;'),
    newline,
    asserta(

```

```

        creation_info(
            [ task, NAME : IDL_TAIL ],
            TASK_TYPE_NAME,
            PREFIX
        ):- !
    )
),
!.

declaration( [task, NAME : IDL_TAIL], PREFIX, CLASSIFIED_G_INSTANCES ):-
    abstraction([task_access, NAME : IDL_TAIL],[static, TEMPLATE_NAME]),
    atomic(TEMPLATE_NAME),
    /* a static instantiation */
    cg_comments(major_header, static_task_instantiation,
        [task, NAME : IDL_TAIL], PREFIX ),
    get_type_name(
        TEMPLATE_NAME,
        [task, NAME : IDL_TAIL],
        CLASSIFIED_G_INSTANCES,
        TASK_TYPE_NAME
    ),
    write_atom(PREFIX), write_atom(NAME), write_atom(' : '), write_atom(TASK_TYPE_NAME),
    write_atom(' ;'), newline, !.

declaration( [task, NAME : IDL_TAIL], PREFIX, _ ):-
    /* single task declaration */
    cg_comments(major_header, single_task_declaration,
        [task, NAME : IDL_TAIL], PREFIX ),
    task_specification([task, NAME : IDL_TAIL], PREFIX ),
    write_atom(' ;'), newline, !.

declaration([channel, NAME : IDL_TAIL], PREFIX, CGI ):-
    abstraction([channel_access, NAME : IDL_TAIL], [_, TEMPLATE_NAME]),
    var(TEMPLATE_NAME),
    /* single channel declaration */
    cg_comments(major_header, single_channel_declaration, [channel,
        NAME : IDL_TAIL], PREFIX ),
    abstraction([channel_internals, NAME : IDL_TAIL ],
        [
            INTERNAL_COMPONENT_IDLS ,
            INTERNAL_OBJECT_IDLS
        ]
    ),
    concatenate_prefix(PREFIX, NAME, NEW_PREFIX ),
    begin_block,

```

```

        basic_declarative_item( INTERNAL_COMPONENT_IDLS, NEW_PREFIX),
/*-----TEST CODE----- param CGI added*/
        basic_declarative_item( INTERNAL_OBJECT_IDLS, NEW_PREFIX, CGI ),
        end_block,
        cg_comments(major_footer, end_single_channel_declaration, [channel,
            NAME : IDL_TAIL], PREFIX ),

declaration([channel, NAME : IDL_TAIL], PREFIX, CGI):-
    abstraction([channel_access, NAME : IDL_TAIL], [_, TEMPLATE_NAME]),
    /* channel instance declaration */
    cg_comments(major_header, channel_instance_declaration, [channel,
        NAME : IDL_TAIL], PREFIX ),
    abstraction([channel_internals, TEMPLATE_NAME, {}],
        [
            INTERNAL_COMPONENT_IDLS ,
            INTERNAL_OBJECT_IDLS
        ]
    ),
    catenate_prefix( PREFIX, NAME, NEW_PREFIX ),
    begin_block,
    basic_declarative_item( INTERNAL_COMPONENT_IDLS, NEW_PREFIX),
/*-----TEST CODE----- param CGI added*/
    basic_declarative_item( INTERNAL_OBJECT_IDLS, NEW_PREFIX, CGI ),
    end_block,
    cg_comments(major_footer, end_channel_instance_declaration, [channel,
        NAME : _], PREFIX ),

declaration([select_accept, SEQUENCE_NUMBER : IDL_TAIL], _):-
    cg_comments(minor_header, select_accept_declaration,
        [select_accept, SEQUENCE_NUMBER : IDL_TAIL], ''),
    begin_block,
    component([select_accept, SEQUENCE_NUMBER : IDL_TAIL],
        [ENTRY_IDL_LISTS]),
    basic_declarative_item(ENTRY_IDL_LISTS, ''),
    end_block,
    cg_comments(minor_footer, end_select_accept_declaration,
        [_, SEQUENCE_NUMBER : _], ''),

declaration([task_pool, NAME : IDL_TAIL], PREFIX, CLASSIFIED_6_INSTANCES):-
    /* a static instantiation */
    abstraction([task_access, NAME : IDL_TAIL], [static, TEMPLATE_NAME]),
    cg_comments(major_header, static_task_pool_instantiation,
        [task_pool, NAME : IDL_TAIL], PREFIX),
    get_type_name(

```

3

of/de

3

MICRO

```

        TEMPLATE_NAME,
        [task_pool, NAME : IDL_TAIL],
        CLASSIFIED_G_INSTANCES,
        TASK_TYPE_NAME
    ),
    write_atom('— SPECIFY number of tasks in pool'),
    newline,
    write_atom('MAX_'),
    write_atom(PREFIX), write_atom(NAME),
    write_atom(' : constant INTEGER := ;'),
    newline,
    write_atom(PREFIX), write_atom(NAME),
    write_atom(' : array( 1..MAX_ ' ),
    write_atom(PREFIX), write_atom(NAME), write_atom(' ) of '),
    write_atom( TASK_TYPE_NAME), write_atom(' ;'),
    newline,!.

declaration([task_pool,NAME : IDL_TAIL], PREFIX, CLASSIFIED_G_INSTANCES):-
/* a dynamic task instance access object declaration */
abstraction([task_access,NAME : IDL_TAIL], [dynamic,TEMPLATE_NAME]),
cg_comments(major_header,dynamic_task_pool_declaration,
    [task_pool,NAME : IDL_TAIL], PREFIX),
get_type_name(
    TEMPLATE_NAME,
    [task_pool, NAME : IDL_TAIL],
    CLASSIFIED_G_INSTANCES,
    TASK_TYPE_NAME
),
write_atom('— SPECIFY number of tasks in pool'),
newline,
write_atom('MAX_'),
write_atom(PREFIX), write_atom(NAME),
write_atom(' : constant INTEGER := ;'),
newline,
write_atom(PREFIX), write_atom(NAME),
write_atom(' : array( 1..MAX_ ' ),
write_atom(PREFIX),write_atom(NAME),write_atom(' ) of '),
write_atom( TASK_TYPE_NAME), write_atom(' _ACCESS ;'),
newline,
asserta(
    (
        creation_info(
            [ task_pool, NAME : IDL_TAIL ],
            TASK_TYPE_NAME,
            PREFIX
        ):-

```

```

declaration([timed_entry, NAME : IDL_TAIL], _):-
    cg_comments(minor_header, timed_entry_declaration,
        [timed_entry, NAME : IDL_TAIL], ''),
    /* Produce code for optional guard */
    (component([guard, GUARD_NAME, NAME : IDL_TAIL], _) =>
        /* entry guarded */
        begin_block,
        cg_comments(simple_comment, guard_declaration,
            [guard, GUARD_NAME, NAME : IDL_TAIL], ''),
        end_block
    ) ; true /* entry unguarded */
), !,

```

H.6 FORMAT STUB

```

/*****
$   FILE_NAME:      format_stub
$   PROJECT_NAME:   CAEDE/1.2 ASCGEN Ada Skeleton Codegenerator
$   DESCRIPTION:    This file contains all page formatting facts and rules
                    to produce ada compatible code and comments. All characters,
                    string lists and atoms are printed using these procedures and
                    the current cursor position fact updated for each print
                    operation. Text mode states are continually upgraded as format
                    commands and print events occur.
$   PROCEDURES IN FILE:
        begin_block,
        begin_block_2,
        begin_comment,
        begin_comment_2,
        begin_header,
        convert_to_list,
        cursor_line_position,
        end_block,
        end_block_2,
        end_comment,
        end_comment_2,
        end_header,
        extract_token,
        indent_size,
        left_block_margin,
        left_header_margin,
        line_wrapped,
        newline,
        newline_2,
        page_width,
        portray,
        portray_2,
        print_horiz_border,
        print_list,
        print_string,
        repeat_put,
        reset,
        status,
        text_mode,
*****/
```



```

write_line,
write_token,

```

\$ HISTORY:

date:	version:	by:	reason:
15 Nov 84	CADA 1.1	K. Van Der Loo	freeze CADA 1.1
5 Jan 85	CAEDE 1.X	"	mods to and
			addition of portray_2, begin_comment_2, end_comment_2, begin_block_2,
			end_block_2, newline_2 for space efficiency improvements.
6 Jan 85	CAEDE 1.X	"	more efficient
			repeat_put procedure.
6 Jan 85	CAEDE 1.X	"	changed
			convert_to_list subgoal atom to atomic to handle numbers.
8 Jan 85	CAEDE 1.X	"	corrections to
			print_list: change print_string to print; add space to separate sublists.
8 Jan 85	"	"	mod to print_string
			to test output performance improvement.
12 Jan 85	"	"	added more print_list
			clauses: for newline, space-paragraph delimiters
20 Jan 85	"	"	addition of
			print_horiz_border, in substitution for repeat_put in begin_header, end_header
3 April 85	"	"	addition of extra
			print_list procedure to handle lists of atoms expected as info facts.
18 April 85	"	"	correction of print_list
			procedure that converts atoms to lists -> name(atom,list) will cause crash
			if non-atom in atom field. Precede with atomic test.
15 June 85	ASCGEN_1.2X	"	changes to portray_2
			and newline_2 to remove line_wrap feature for testing, to allow faster output.
			Code changes are noted in place with replaced code commented out.
23 Aug 86	ASCGEN_1.2X	"	corrected bug in
			print_list: lists were being printed instead of atoms. Last fact changed.

```

*****
*/

```

```

print_list([' : LIST_TAIL]):-
    newline,
    !,
    print_list(LIST_TAIL).
print_list([SENTENCE : LIST_TAIL]):-
    atomic(SENTENCE),
    write_atom( SENTENCE),
    newline,
    !,

```

```

        print_list(LIST_TAIL).
print_list([]).
print_list([[]: SUBLIST_TAIL):-          /* newline required */
    !,
    newline,
    print_list(SUBLIST_TAIL).
/* Original formatter did line fill operation with column 1 line space
indicating paragraph break. This is not used now. I assume the user formats
the text as he sees fit. A better implementation of Prolog might make this
operation efficient'
print_list([[]32 : SUBLIST_TAIL] : LIST_TAIL):-  /* When space is first
character on line, then a new paragraph is being started */
    !,
    newline,
    print_string(SUBLIST_TAIL),
    write_atom(' '),
    print_list(LIST_TAIL).
There are some other rules needed to call string put operations.
*/

/*
print_string(ASCII_STRING):-
    name(ATOM, ASCII_STRING),
    write(ATOM).
*/
print_string([]).
print_string([ASCII_CODE : ASCII_STRING_TAIL ]:-
    put(ASCII_CODE),
    print_string(ASCII_STRING_TAIL).

/* FORMAT CONSTANTS */

page_width(80).

indent_size(4).

o /* INITIAL FORMAT STATE */

line_of_code.

indent_line.    /* No indent initially */

left_block_margin(0).

print_horiz_border(LENGTH, [ASCII_CHAR]):- /* Learning border fact generator */
    repeat_put(LENGTH, ASCII_CHAR, BORDER_LIST),

```

```

name(BORDER_ATOM, BORDER_LIST),
asserta((print_horiz_border(LENGTH, [ASCII_CHAR])):-
    write_atom(BORDER_ATOM, '!'),
    write_atom(BORDER_ATOM),
    fail. /* for stack-space recovery */
print_horiz_border(_, _).

repeat_put(0, _, []):- !. /* for stack-space recovery */
repeat_put(NUMBER_OF_TIMES, ASCII_CHAR, [ASCII_CHAR : SUB_BORDER]):-
    TIMES_LEFT is NUMBER_OF_TIMES - 1,
    repeat_put(TIMES_LEFT, ASCII_CHAR, SUB_BORDER).

/* If line already started, newline */
newline:-
    retract(line_already_started),
    nl,
    !.

/* else if blank code newline */
newline:-
    line_of_code,
    nl,
    !.

/* else blank indented comment line */
newline:-
    indent_line,
    write('---'),
    nl,
    !.

/* else blank comment line, column 0 */
newline:-
    write('---'),
    nl,
    !.

/* If line already started */
write_atom(ATOM):-
    line_already_started,
    write(ATOM),
    !.

/* else first atom of code */

```

```
write_atom(ATOM):-
    line_of_code,
    indent_line,
    write(ATOM),
    assert(line_already_started),
    !.

/* else first comment atom */
write_atom(ATOM):-
    indent_line,
    write('—'),
    write(ATOM),
    assert(line_already_started).

begin_comment:-
    retract(line_of_code).

end_comment:-
    asserta(line_of_code).

begin_header(HEADER_WIDTH, HORIZONTAL_BORDER_CHAR, _):-
    retract(line_of_code),
    print_horiz_border(HEADER_WIDTH, HORIZONTAL_BORDER_CHAR),
    newline,
    !.

end_header(HEADER_WIDTH, HORIZONTAL_BORDER_CHAR, _):-
    print_horiz_border(HEADER_WIDTH, HORIZONTAL_BORDER_CHAR),
    newline,
    asserta(line_of_code),
    !.

begin_block:-
    left_block_margin(MARGIN),
    indent_size(TAB),
    NEW_MARGIN is MARGIN + TAB,
    asserta(left_block_margin(NEW_MARGIN)),
    asserta((indent_line:- tab(NEW_MARGIN),)),
    !.

end_block:-
    retract(left_block_margin(_)),
    retract((indent_line:- tab(NEW_MARGIN),)),
    !.
```

```
convert_to_list(TEXT_ITEM, TEXT_ITEM_LIST) :-  
    atomic(TEXT_ITEM),  
    name(TEXT_ITEM, TEXT_ITEM_LIST).  
/* already a list */  
convert_to_list(TEXT_ITEM_LIST, TEXT_ITEM_LIST).
```

H.7 GENERIC BODY

```

generic_bodies( PARENT_INT_IDL, ENV_STACK ):-
    cg_comments_description(
        generic,
        PARENT_INT_IDL,
        TEMPLATE_NAME
    ),
    cg_comments(
        major_header,
        package_template_body,
        [package_template, TEMPLATE_NAME, _],
        ''
    ),
    type_body_description(
        PARENT_INT_IDL,
        TEMPLATE_NAME,
        INSTANCE_EXT_IDL,
        GENERIC_TYPE_NAME
    ),
    generic_body(
        GENERIC_TYPE_NAME,
        [ package_template, TEMPLATE_NAME, _],
        [ environment('', INSTANCE_EXT_IDL ) : ENV_STACK ]
    ),
    newline, fail.

generic_bodies( _, _). /* catchall when no more generic bodies
    required. */

```


H.9 INSTANCES

make_inst_externals_facts:-

```

    object( [ task, NAME, ANCESTRY], _),
    abstraction( [task_access, NAME, ANCESTRY], [_, TEMPLATE_NAME]),
    atom( TEMPLATE_NAME ),
    abstraction( [ task_externals, TEMPLATE_NAME, []],
        [ TEMPLATE_EXT_IDLS ] ),
    duplicate_idls(
        TEMPLATE_EXT_IDLS,
        [ NAME : ANCESTRY ],
        INSTANCE_EXT_IDLS
    ),
    asserta(
        abstraction( [ task_externals, NAME, ANCESTRY ],
            [ INSTANCE_EXT_IDLS ] )
    ),
    fail.

```

make_inst_externals_facts:-

```

    object( [ task_pool, NAME, ANCESTRY], _),
    abstraction( [task_access, NAME, ANCESTRY], [_, TEMPLATE_NAME]),
    atom( TEMPLATE_NAME ),
    abstraction( [ task_externals, TEMPLATE_NAME, []],
        [ TEMPLATE_EXT_IDLS ] ),
    duplicate_idls(
        TEMPLATE_EXT_IDLS,
        [ NAME : ANCESTRY ],
        INSTANCE_EXT_IDLS
    ),
    asserta(
        abstraction( [ task_externals, NAME, ANCESTRY ],
            [ INSTANCE_EXT_IDLS ] )
    ),
    fail.

```

make_inst_externals_facts:-

```

    object( [ package, NAME, ANCESTRY], _),
    abstraction( [package_access, NAME, ANCESTRY], [_, TEMPLATE_NAME]),
    atom( TEMPLATE_NAME ),
    abstraction( [ package_externals, TEMPLATE_NAME, []],
        [ TEMPLATE_EXT_IDLS ] ),
    duplicate_idls(
        TEMPLATE_EXT_IDLS,
        [ NAME : ANCESTRY ],

```



```

        INSTANCE_EXT_IDLS
    ),
    asserta(
        abstraction( [ package_externals, NAME, ANCESTRY ],
            [ INSTANCE_EXT_IDLS ] )
    ),
    fail.

```

```
make_inst_externals_facts.
```

```
duplicate_idls( [], _, []):- !.
```

```

duplicate_idls(
    [[ KIND, NAME, _ ] : TEMPLATE_EXT_IDLS_TAIL ],
    INSTANCE_ANCESTRY,
    [[ KIND, NAME, INSTANCE_ANCESTRY] : INSTANCE_EXT_IDLS_TAIL]
):-
    duplicate_idls(
        TEMPLATE_EXT_IDLS_TAIL,
        INSTANCE_ANCESTRY,
        INSTANCE_EXT_IDLS_TAIL
    ).

```

```
append( [], LIST, LIST).
```

```
append( [ X : L1 ], L2, [ X : L3 ]):- append( L1, L2, L3 ).
```

```
/*
```

```
*****
```

```
NAME:
```

```

get_prefix(
    partially assembled prefix list, formed by catenating
        more-local environment prefixes, in param
    local environment prefix list, a list of remaining atoms
        which name channels with the left list elements
        being nested within the right list elements, in
        param.
    assembled prefix atom, out parameter.
).

```

```
DESCRIPTION:
```

This procedure takes the LOCAL_ENVIRONMENT_PREFIX list, which is a list of channel names in reverse order of nesting and converts it into an atom prefix with each element separated from the next by an underline. The list order is reversed so that the prefix reads "outer-channel_.. innermost-channel_".

```
*****
*/
```

```
get_prefix( [], [], '' ):- '.
```

```
get_prefix( L, [], A ):-
    name( A, L ),
    !.
```

```
get_prefix( AL, [ B : BT ], CA ):-
    name( B, BL ),
    append( BL, [95], BBL ),
    append( BBL, AL, BBAL ),
    get_prefix( BBAL, BT, CA ),
    !.
```

```
/****** Add null completion code******/
```

```
/*
```

```
*****
```

NAME:

```
get_instances(
    list of object internals,
    local_environment_prefix, a list to designate ancestry when classifying
    channel internals. Channel internals of interest could
    be either task instances, package instances or channel
    or channel instances, in parameter.
    classified_g_instances structure
).
```

DESCRIPTION:

This procedure recursively filters instances from a list of IDL's and classifies these instances by template name and type in the structure classified_g_instances. The filtrate, non-instance IDL's are returned for further processing to the caller.

```
*****
*/
```

```
get_instances( [], _, _):-
```

```
get_instances( [ IDL : REMAINING_IDLS ],
    LOCAL_ENVIRONMENT_PREFIX,
    classified_g_instances( G_TEMPLATE_CATEGORY )
):-
    is_instance( IDL, LOCAL_ENVIRONMENT_PREFIX,
        classified_g_instances( G_TEMPLATE_CATEGORY )
    ),
    get_instances( REMAINING_IDLS, LOCAL_ENVIRONMENT_PREFIX,
        classified_g_instances( G_TEMPLATE_CATEGORY )
    ),
    !.
```

```
get_instances( [ _ : OTHER_IDLS ],
    LOCAL_ENVIRONMENT_PREFIX,
    classified_g_instances( G_TEMPLATE_CATEGORY )
):-
    get_instances( OTHER_IDLS, LOCAL_ENVIRONMENT_PREFIX,
        classified_g_instances( G_TEMPLATE_CATEGORY )
    ),
    !.
```

```
/*
```

```
*****
```

NAME:

```
is_instance(
    object internal IDL in parameter.
    local_environment_prefix, a list to designate ancestry when classifying
        channel internals. Channel internals of interest could
        be either task instances, package instances or channel
        or channel instances, in parameter.
    classified_g_instances structure, in out parameter.
).
```

DESCRIPTION:

This procedure succeeds if the IDL is an instance and will categorize the instance in the classified_g_instances structure. This structure will associate instances by the required Ada task

type or generic name and will identify the outbound interface for each of them.

*/

```
is_instance( [ channel, NAME : IDL_TAIL ],
             LOCAL_ENVIRONMENT_PREFIX, CLASSIFIED_G_INSTANCES
):-
    /* This processes g_channel_instances */
    abstraction( [ channel_access, NAME : IDL_TAIL ], [_, TEMPLATE_NAME ]),
    atom( TEMPLATE_NAME ),
    abstraction( [ channel_internals, TEMPLATE_NAME. [] ],
                [_, OBJECT_IDLS_LIST ]),
    append( LOCAL_ENVIRONMENT_PREFIX, [ NAME ], NEW_LOCAL_ENVIRONMENT_PREFIX ),
    get_instances( OBJECT_IDLS_LIST, NEW_LOCAL_ENVIRONMENT_PREFIX,
                  CLASSIFIED_G_INSTANCES ),
    !.
```

```
is_instance( [ channel, NAME : IDL_TAIL ], LOCAL_ENVIRONMENT_PREFIX,
             CLASSIFIED_G_INSTANCES
):-
    /* This processes single channels */
    abstraction( [ channel_internals, NAME : IDL_TAIL ],
                [_, OBJECT_IDLS_LIST ]),
    append( LOCAL_ENVIRONMENT_PREFIX, [ NAME ], NEW_LOCAL_ENVIRONMENT_PREFIX ),
    get_instances( OBJECT_IDLS_LIST, NEW_LOCAL_ENVIRONMENT_PREFIX,
                  CLASSIFIED_G_INSTANCES ),
    !.
```

/* This processes all g_package_instances, sorting instances
by TEMPLATE_NAME.

*/

```
is_instance( [package, NAME : IDL_TAIL ], LOCAL_ENVIRONMENT_PREFIX,
             classified_g_instances( G_TEMPLATE_CATEGORY )
):-
    abstraction( [package_access, NAME : IDL_TAIL ], [_, TEMPLATE_NAME ]),
    atom( TEMPLATE_NAME ),
    locate_g_template( TEMPLATE_NAME, [package, NAME : IDL_TAIL ],
                      LOCAL_ENVIRONMENT_PREFIX, G_TEMPLATE_CATEGORY ),
    !.
```

/* This processes all other g_task_instances, sorting instances
by TEMPLATE_NAME.

*/

```

is_instance( [GRAPHIC_TYPE, NAME : IDL_TAIL ], LOCAL_ENVIRONMENT_PREFIX,
             classified_g_instances( G_TEMPLATE_CATEGORY )
):-
    abstraction( [task_access, NAME : IDL_TAIL ], [_, TEMPLATE_NAME ]),
    !,
    atom( TEMPLATE_NAME ),
    locate_g_template( TEMPLATE_NAME, [GRAPHIC_TYPE, NAME : IDL_TAIL ],
                      LOCAL_ENVIRONMENT_PREFIX, G_TEMPLATE_CATEGORY ),

/*
*****

NAME:

locate_g_template(
    template_name in parameter,
    IDL;
    local_environment_prefix, a list to designate ancestry when classifying
        channel internals. Channel internals of interest could
        be either task instances, package instances or channel
        or channel instances; in parameter.
    g_template_category structure for this template_name, in or
        out parameter dependent on rule.
).

DESCRIPTION:

IF TEMPLATE_NAME is already categorized in this template category
    structure, ie. the rule head matches existing instantiations
    in the goal parameters, then classify this g_instance
    OR the goal parameters are uninstantiated, so add a new
    g_template_category element to the classified_g_instances
    structure by instantiation, and classify this g_instance.

ELSE recursively search the g_template_category list for this
    TEMPLATE_NAME.

*****
*/

locate_g_template( TEMPLATE_NAME, IDL, LOCAL_ENVIRONMENT_PREFIX,
                  g_template_category(
                      TEMPLATE_NAME,
                      G_TYPE_CATEGORY,

```

```

    )
):-
    check_for_action_chains(IDL, ANY_OUTBOUND_ACTION_CHAINS),
    get_all externals_with_parameters( IDL, INBOUND_INTERFACE ),
    classify_g_instance(
        IDL,
        LOCAL_ENVIRONMENT_PREFIX,
        1,
        instance_profile(
            INBOUND_INTERFACE,
            ANY_OUTBOUND_ACTION_CHAINS
        ),
        G_TYPE_CATEGORY
    ),
    !.

locate_g_template( TEMPLATE_NAME, IDL, LOCAL_ENVIRONMENT_PREFIX,
    g_template_category(
        -,
        -,
        OTHER_G_TEMPLATE_CATEGORIES
    )
):-
    locate_g_template( TEMPLATE_NAME, IDL, LOCAL_ENVIRONMENT_PREFIX,
        OTHER_G_TEMPLATE_CATEGORIES
    ),
    !.

```

/*

NAME:

check_for_action_chains(

IDL to be examined for outbound action chains. in parameter,
boolean result of search, out parameter.

).

DESCRIPTION:

This procedure checks an instance for any outbound
action_chains. If any are found this means the instance must have
an ada type declaration exclusively for itself. This is the reason
for this check.

*/

```
check_for_action_chains(
    [_, NAME : IDL_TAIL ],
    any_outbound_action_chains( yes )
):-
    component(
        [connection_point, _ : [NAME : IDL_TAIL ]],
```

```
),
!.,
check_for_action_chains(
    [_,
    any_outbound_action_chains( no )
):-
```

/*

NAME:

```
classify_g_instance(
    IDL to be classified, in parameter,
    local_environment_prefix, a list to designate ancestry when classifying
        channel internals. Channel internals of interest could
        be either task instances, package instances or channel
        or channel instances, in parameter.
    type_name_suffix sequence number corresponding to this type,
        in parameter,
    instance_profile(
        inbound_interface: list of all access-point formal
            parameters, in parameter.
        any_outbound_action_chains( X ), yes or no, in parameter
    ),
    g_type_category structure, in out parameter
```

7.

DESCRIPTION:

IF a new type category is needed, a new g_type_category is
instantiated along with the next type_suffix sequence number

and the instance as the identical_g_instance list head.

ELSE IF this instance matches with the first instance on the current type category list, then this instance is recursively appended to the end of the list. A match occurs if the instances could be represented by the same Ada task type or generic unit. Two instances of the same template, which could be a task or task_pool, or a package. MATCH when:

- (1) Have same template; -- already guaranteed by this rule;
- (2) Have same real nesting level; -- instances to be compared by this rule might be nested within channels which are not Ada objects, hence are not really nested.
- (3) Have same instance profile.

ELSE IF there is no match, then the rule fails and the remaining type categories are recursively searched.

*/

```

classify_g_instance(
    IDL,
    LOCAL_ENVIRONMENT_PREFIX,
    TYPE_NAME_SUFFIX,
    INSTANCE_PROFILE,
    g_type_category(
        TYPE_NAME_SUFFIX,
        INSTANCE_PROFILE,
        INSTANTIATED_IF_DYNAMIC_ACCESS,
        identical_g_instance(
            IDL,
            LOCAL_ENVIRONMENT_PREFIX,
        ),
    ),
)
):-
    check_for_dynamic_access( IDL, INSTANTIATED_IF_DYNAMIC_ACCESS ),

```

```

classify_g_instance(
    IDL,
    LOCAL_ENVIRONMENT_PREFIX,
    -
    INSTANCE_PROFILE,
    g_type_category(
        -

```



```

        INSTANCE_PROFILE,
        INSTANTIATED_IF_DYNAMIC_ACCESS,
        identical_g_instance(
            -',
            -',
            OTHER_IDENTICAL_G_INSTANCES
        ),
    -
)

):-
    check_for_dynamic_access( IDL, INSTANTIATED_IF_DYNAMIC_ACCESS ),
    add_g_instance( IDL, LOCAL_ENVIRONMENT_PREFIX,
        OTHER_IDENTICAL_G_INSTANCES ),
    !.

classify_g_instance(
    IDL,
    LOCAL_ENVIRONMENT_PREFIX,
    TYPE_NAME_SUFFIX,
    INSTANCE_PROFILE,
    g_type_category(
        -',
        -',
        -',
        -',
        OTHER_G_TYPE_CATEGORIES
    )
):-
    NEXT_TYPE_NAME_SUFFIX is TYPE_NAME_SUFFIX + 1,
    classify_g_instance(
        IDL,
        LOCAL_ENVIRONMENT_PREFIX,
        NEXT_TYPE_NAME_SUFFIX,
        INSTANCE_PROFILE,
        OTHER_G_TYPE_CATEGORIES
    ),
    !.

/*
*****

NAME:

check_for_dynamic_access(
    IDL,

```

instantiated to -> some_dynamic_access_instances or left
uninstantiated.

).

DESCRIPTION:

The second argument is instantiated if IDL is a dynamic
access task or task_pool.

```
*****
*/
check_for_dynamic_access(
    [ _, NAME : IDL_TAIL ],
    some_dynamic_access_instances
):-
    abstraction( [task_access, NAME : IDL_TAIL ], [ dynamic, _ ] ),
    !.

check_for_dynamic_access( _, _ ):-
    !.

/*
*****
```

NAME:

```
get_all_externals_with_parameters(
    instance IDL, in parameter
    inbound_interface, a list of either pack_ext_procedures or
    task_entries in standard order with their parameters in
    standard order as well, out parameter.
).
```

DESCRIPTION:

get_all_externals_with_parameters returns with all of this
instances' access points, either pack_ext_procedure or task_entries
along with their parameters, each sorted in standard order.

```
*****
*/
```

```
get_all_externals_with_parameters(
    [ task, TASK_NAME, ANCESTRY],
    INBOUND_INTERFACE
):-
```

```

        setof(
            access_point(NAME, SORTED_PARAMETERS),
            sorted_parameters(
                [ task_entry, NAME, [TASK_NAME : ANCESTRY ]],
                SORTED_PARAMETERS
            ),
            INBOUND_INTERFACE
        ),
        !.

get_all_externals_with_parameters(
    [ task_pool, TASKPOOL_NAME, ANCESTRY],
    INBOUND_INTERFACE
):-
    setof(
        access_point(NAME, SORTED_PARAMETERS),
        sorted_parameters(
            [ task_entry, NAME, [TASKPOOL_NAME : ANCESTRY ]],
            SORTED_PARAMETERS
        ),
        INBOUND_INTERFACE
    ),
    !.

get_all_externals_with_parameters(
    [ package, PACKAGE_NAME, ANCESTRY ],
    INBOUND_INTERFACE
):-
    setof(
        access_point(NAME, SORTED_PARAMETERS),
        sorted_parameters(
            [pack_ext_procedure, NAME, [PACKAGE_NAME : ANCESTRY ]],
            SORTED_PARAMETERS
        ),
        INBOUND_INTERFACE
    ),
    !.

get_all_externals_with_parameters( _, []).    /* Catch-all if no externals */

/*
*****

```

NAME:

```
sorted_parameters(
    IDL, for either pack_ext_procedure or entry for which
        parameters must be obtained and sorted, in parameter.
    list of formal parameters, sorted in standard order, out
        parameter
).
```

DESCRIPTION:

sorted_parameters returns with the parameters sorted in the standard order.

```
*****
*/

sorted_parameters(
    IDL,
    SORTED_PARAMETERS
):-
    component( IDL, [ PARAMETERS ] ),
    sort( PARAMETERS, SORTED_PARAMETERS ).

/*
*****
```

NAME:

```
add_g_instance(
    IDL, in parameter,
    local_environment_prefix list for this instance, in parameter,
    identical_g_instance list, in out parameter
).
```

DESCRIPTION:

Add_g_instance adds the instance unique identification to the end of the type category list recursively.

```
*****
*/

add_g_instance(
    IDL,
    LOCAL_ENVIRONMENT_PREFIX,
```

```

        identical_g_instance(
            IDL,
            LOCAL_ENVIRONMENT_PREFIX,
            -
        )
    ):-
        !.

add_g_instance(
    IDL,
    LOCAL_ENVIRONMENT_PREFIX,
    identical_g_instance(
        -
        -
        OTHER_IDENTICAL_G_INSTANCES
    )
):-
    add_g_instance(
        IDL,
        LOCAL_ENVIRONMENT_PREFIX,
        OTHER_IDENTICAL_G_INSTANCES
    ),
    !.

/*
*****

NAME:

get_type_name(
    template_name, either a package or task templated name,
                    in parameter.
    instance_idl, in parameter.
    classified_g_instances structure, in parameter.
    type_name, out parameter.
).

DESCRIPTION:

This procedure finds a predetermined type name extension for the
given idl, preclassified in the structure. The search key is the
TEMPLATE NAME initially, after which a linear search is made to
find which type name suffix the idl was classified under.

*****

```

```

*/
get_type_name(
    TEMPLATE_NAME,
    IDL,
    classified_g_instances( G_TEMPLATE_CATEGORY ),
    TYPE_NAME
):-
    match_template_name(
        TEMPLATE_NAME,
        G_TEMPLATE_CATEGORY,
        MATCHING_TEMPLATE_TYPE_CATEGORY
    ),
    match_idl(
        IDL,
        MATCHING_TEMPLATE_TYPE_CATEGORY,
        TYPE_SUFFIX
    ),
    catenate_atoms( TEMPLATE_NAME, '_', R ),
    catenate_atoms( R, TYPE_SUFFIX, TYPE_NAME ).

just_get_type_name(
    PARENT_INT_IDL,
    KIND,
    package_template,
    TEMPLATE_NAME,
    IDL,
    classified_g_instances( G_TEMPLATE_CATEGORY ),
    TYPE_NAME
):-
    !,
    match_template_name(
        TEMPLATE_NAME,
        G_TEMPLATE_CATEGORY,
        MATCHING_TEMPLATE_TYPE_CATEGORY
    ),
    object([package_template, TEMPLATE_NAME : _], _),
    assert(
        cg_comments_description(
            generic,
            PARENT_INT_IDL,
            TEMPLATE_NAME
        )
    ),
    cg_comments(major_header, KIND,
        [package_template, TEMPLATE_NAME : _], '' ),
    match_first_idl(

```

```

        IDL,
        MATCHING_TEMPLATE_TYPE_CATEGORY,
        TYPE_SUFFIX
    ),
    catenate_atoms( TEMPLATE_NAME, '_', R ),
    catenate_atoms( R, TYPE_SUFFIX, TYPE_NAME ).

just_get_type_name(
    PARENT_INT_IDL,
    KIND,
    task_template,
    TEMPLATE_NAME,
    IDL,
    classified_g_instances( G_TEMPLATE_CATEGORY ),
    TYPE_NAME
):-
    1,
    match_template_name(
        TEMPLATE_NAME,
        G_TEMPLATE_CATEGORY,
        MATCHING_TEMPLATE_TYPE_CATEGORY
    ),
    object([task_template, TEMPLATE_NAME : _], _),
    assert(
        cg_comments_description(
            task_type,
            PARENT_INT_IDL,
            TEMPLATE_NAME
        )
    ),
    cg_comments(major_header, KIND,
        [task_template, TEMPLATE_NAME : _], '' ),
    match_first_idl(
        IDL,
        MATCHING_TEMPLATE_TYPE_CATEGORY,
        TYPE_SUFFIX
    ),
    catenate_atoms( TEMPLATE_NAME, '_', R ),
    catenate_atoms( R, TYPE_SUFFIX, TYPE_NAME ).

match_template_name(
    _
    G_TEMPLATE_CATEGORY,
    _
):-
    var( G_TEMPLATE_CATEGORY ),

```

', fail.

```
match_template_name(
    TEMPLATE_NAME,
    g_template_category(
        TEMPLATE_NAME,
        G_TYPE_CATEGORY,
        -
    ),
    G_TYPE_CATEGORY
).
```

```
match_template_name(
    TEMPLATE_NAME,
    g_template_category(
        -
        -
        OTHER_G_TEMPLATE_CATEGORIES
    ),
    G_TYPE_CATEGORY
):-
    match_template_name(
        TEMPLATE_NAME,
        OTHER_G_TEMPLATE_CATEGORIES,
        G_TYPE_CATEGORY
    ).
```

```
match_idl(
    IDL,
    g_type_category(
        TYPE_SUFFIX,
        -
        -
        IDENTICAL_G_INSTANCE,
        -
    ),
    TYPE_SUFFIX
):-
    is_present(IDL, IDENTICAL_G_INSTANCE),
    '.
```

```
match_idl(
    IDL,
    g_type_category(
        -
    ),
```



```

        -'
        -'
        -'
        OTHER_G_TYPE_CATEGORY
    ),
    TYPE_SUFFIX
):-
    match_idl(
        IDL,
        OTHER_G_TYPE_CATEGORY,
        TYPE_SUFFIX
    ).

match_first_idl(
    IDL,
    g_type_category(
        TYPE_SUFFIX,
        -'
        -'
        IDENTICAL_G_INSTANCE,
        -
    ),
    TYPE_SUFFIX
):-
    var( IDENTICAL_G_INSTANCE ),
    !, fail.

match_first_idl(
    IDL,
    g_type_category(
        TYPE_SUFFIX,
        -'
        -'
        identical_g_instance(
            IDL,
            -'
            -
        ),
        -
    ),
    TYPE_SUFFIX
).

match_first_idl(
    IDL,
    g_type_category(

```

```

        _',
        _',
        _',
        _',
        OTHER_G_TYPE_CATEGORY
    ),
    TYPE_SUFFIX
):-
    match_first_idl(
        IDL,
        OTHER_G_TYPE_CATEGORY,
        TYPE_SUFFIX
    ).

is_present(_, V):-
    var( V ),
    !, fail.

is_present(
    IDL,
    identical_g_instance(
        IDL,
        _',
        _'
    )
):-
    !.

is_present(
    IDL,
    identical_g_instance(
        _',
        _',
        OTHER_IDENTICAL_G_INSTANCE
    )
):-
    is_present(
        IDL,
        OTHER_IDENTICAL_G_INSTANCE
    ).

catenate_atoms( '', B, B ).

catenate_atoms( A, '', A ).

catenate_atoms( A, B, AB ):-

```

```
name( A, AL ),  
name( B, BL ),  
append( AL, BL, ABL ),  
name( AB, ABL ).
```

H.10 PACK STRUCT 7.1

```

/*****
$      FILE_NAME:      pack_struct_7.1

$      PROJECT_NAME:   CAEDE/1.2 ASCGEN Ada Skeleton Codegenerator

$      DESCRIPTION:    This file contains code to generate ada syntax code.

$      PROCEDURES IN FILE:
                package_specification,
                package_body.

$      HISTORY:

                date:      version:      by:      reason:
                -----
                15 Nov 84   CADA 1.1      K. Van Der Loo   freeze CADA 1,1
                15 June 85  ASCGEN/1.2X   "              "      add
PREFIX parameter to cg_comments goals, and
package_specification and body.

*****/

```

```
generic_specification([package, NAME : IDL_TAIL], GENERIC_NAME):-
    write_atom('generic '), write_atom('package '),
    write_atom(GENERIC_NAME), write_atom(' is'), newline,
    begin_block,
    abstraction([package_externals, NAME : IDL_TAIL],
        [ EXTERNAL_COMPONENT_IDLS ] ),
    basic_declarative_item(EXTERNAL_COMPONENT_IDLS, ''),
    end_block,
    write_atom('end '), write_atom(GENERIC_NAME),
    !.

/* package specs occur in package and generic package declarations */
package_specification([package, NAME : IDL_TAIL], PREFIX):-
    write_atom('package '),
    write_atom(PREFIX ), write_atom(NAME), write_atom(' is'), newline,
    begin_block,
    abstraction([package_externals, NAME : IDL_TAIL],
        [ EXTERNAL_COMPONENT_IDLS ] ),
```

```

    basic_declarative_item(EXTERNAL_COMPONENT_IDLS, ''),
    end_block,
    write_atom('end '), write_atom(PREFIX), write_atom(NAME).

```

/* New package body rule */

```

generic_body(
    GENERIC_NAME,
    [INT_PACKAGE_KIND, INT_NAME : INT_IDL_TAIL ],
    [ environment( EXT_PREFIX, EXT_IDL ) : ENV_STACK_TAIL ]
):-
    get_unique_key(KEY),
    parse_action_chains(
        KEY,
        [main_line, INT_NAME : INT_IDL_TAIL],
        [
            environment(
                "",
                [INT_PACKAGE_KIND, INT_NAME : INT_IDL_TAIL ]
            ),
            environment( EXT_PREFIX, EXT_IDL )
            : ENV_STACK_TAIL
        ]
    ),
    write_atom('package body '),
    write_atom(GENERIC_NAME), write_atom(' is'), newline,
    begin_block,
    (
        declarative_part(
            [INT_PACKAGE_KIND, INT_NAME : INT_IDL_TAIL ],
            [ environment( EXT_PREFIX, EXT_IDL ) : ENV_STACK_TAIL ]
        )
    ); true
),
end_block,
/* code to complete optional package_mainline */
sequence_of_statements(
    KEY,
    [INT_PACKAGE_KIND, INT_NAME : INT_IDL_TAIL ],
    ),
write_atom('end '),
write_atom(GENERIC_NAME), write_atom(' ;'), newline,
!

```

```

package_body(
    [INT_PACKAGE_KIND, INT_NAME : INT_IDL_TAIL ],
    [ environment(
        EXT_PREFIX,
        [ EXT_PACKAGE_KIND, EXT_NAME : EXT_IDL_TAIL ]
    )
    : ENV_STACK_TAIL ]
):-
    get_unique_key(KEY),
    parse_action_chains(
        KEY,
        [main_line, INT_NAME : INT_IDL_TAIL],
        [
            environment(
                "",
                [INT_PACKAGE_KIND, INT_NAME : INT_IDL_TAIL ]
            ),
            environment( EXT_PREFIX,
                [EXT_PACKAGE_KIND, EXT_NAME : EXT_IDL_TAIL ])
            : ENV_STACK_TAIL ]
        ),
    cg_comments(major_header, package_body,
        [ EXT_PACKAGE_KIND, EXT_NAME : EXT_IDL_TAIL ], EXT_PREFIX ),
    write_atom('package body '),
    write_atom(EXT_PREFIX), write_atom(EXT_NAME), write_atom(' is'), newline,
    begin_block,
    (
        declarative_part(
            [INT_PACKAGE_KIND, INT_NAME : INT_IDL_TAIL ],
            [ environment(
                EXT_PREFIX,
                [EXT_PACKAGE_KIND, EXT_NAME : EXT_IDL_TAIL ]
            )
            : ENV_STACK_TAIL ]
        )
    ); true
),
end_block,
/* code to complete optional package_mainline */
sequence_of_statements(
    KEY,
    [INT_PACKAGE_KIND, INT_NAME : INT_IDL_TAIL ],
),

```

```
write_atom('end '),  
write_atom(EXT_PREFIX), write_atom(EXT_NAME), write_atom(';'), newline,  
!.
```

H.11 STATEMENTS 5.1

```

/*
/*****

$    FILE_NAME:      seq_of_statements_5.1

$    PROJECT_NAME:   CAEDE/1.2 ASCGEN Ada Skeleton Codegenerator

$    DESCRIPTION:    This file contains procedures to generate ada
                    syntax code.

$    PROCEDURES IN FILE:
        accepts,
        interaction_call,
        generate_calls,
        more_accepts,
        sequence_of_statements,
        statement,

$    HISTORY:

        date:          version:          by:          reason:
        -----
        15 Nov 84      CADA 1.1          K. Van Der Loo  freeze CADA 1.1
        15 June 85     ASCGEN/1.2X       "             "      add local
environment parameter to cg_comments calls.
        5 July 85      "                 "             "      correction to
package code.

*****/

sequence_of_statements(
    KEY,
    [ task : INT_IDL_TAIL ],
    [
        INT_ENV,
        environment(
            EXT_PREFIX,
            [ EXT_TASK_KIND : EXT_IDL_TAIL ]
        ),
        ENV_TAIL
    ]
):-
    abstraction(

```



```

        [task_externals : EXT_IDL_TAIL ],
        [ EXTERNAL_COMPONENT_IDLS ]),
accepts(
    EXTERNAL_COMPONENT_IDLS,
    [ task : INT_IDL_TAIL ],
    [
        INT_ENV,
        environment(
            EXT_PREFIX,
            [ EXT_TASK_KIND : EXT_IDL_TAIL ])
        : ENV_TAIL ]
),
task_allocators( KEY),
generate_calls( KEY),'.

sequence_of_statements(
    KEY,
    [ task_template : INT_IDL_TAIL ],
    [
        INT_ENV,
        environment(
            EXT_PREFIX,
            [ EXT_TASK_KIND : EXT_IDL_TAIL ]
        )
        : ENV_TAIL
    ]
):-
    abstraction(
        [task_externals : EXT_IDL_TAIL ],
        [ EXTERNAL_COMPONENT_IDLS ]),
    accepts(
        EXTERNAL_COMPONENT_IDLS,
        [ task_template : INT_IDL_TAIL ],
        [
            INT_ENV,
            environment(
                EXT_PREFIX,
                [ EXT_TASK_KIND : EXT_IDL_TAIL ])
            : ENV_TAIL ]
    ),
    task_allocators(KEY),
    generate_calls( KEY),'.

sequence_of_statements(
    KEY,

```

```

[package : INT_IDL_TAIL ],
:-
    component([main_line : INT_IDL_TAIL ],_),
    write_atom('begin'),newline, begin_block,
    task_allocators(KEY),
    generate_calls(KEY),
    end_block,!.

/* termination condition if no mainline in package. */
sequence_of_statements(_, [package : _ ], _):- !.

sequence_of_statements(
    KEY,
    [package_template : INT_IDL_TAIL ],
):-
    component([main_line : INT_IDL_TAIL ],_),
    write_atom('begin'),newline, begin_block,
    task_allocators(KEY),
    generate_calls(KEY),
    end_block,!.

/* termination condition if no mainline in package. */
sequence_of_statements(_, [package_template : _ ], _):- !.

sequence_of_statements(KEY, _, _):-
    task_allocators(KEY),
    generate_calls(KEY),
    !.

accepts([], _, _).

accepts(
    [IDL : IDL_TAIL ],
    TASK_KIND_IDL,
    ENV_STACK
):-
    statement(
        IDL,
        TASK_KIND_IDL,
        ENV_STACK
    ),
    accepts(
        IDL_TAIL,

```

```

TASK_KIND_IDL,
ENV_STACK
).

statement(
  [select_accept,SEQUENCE_NUMBER : EXT_IDL_TAIL],
  PARENT_IDL,
  ENV_STACK
):-
  component([select_accept,SEQUENCE_NUMBER : EXT_IDL_TAIL],
    [[ ENTRY_IDL : IDL_LISTS_TAIL ]]),
  write_atom('select -- sequence number '),write_atom(SEQUENCE_NUMBER),
  newline, begin_block,
  statement(
    ENTRY_IDL,
    PARENT_IDL,
    ENV_STACK
  ),
  more_accepts(
    IDL_LISTS_TAIL,
    PARENT_IDL,
    ENV_STACK
  ),
  write_atom('end select ; -- sequence number '),
  write_atom(SEQUENCE_NUMBER),newline.

statement(
  [task_entry, EXT_NAME : EXT_IDL_TAIL ],
  [_, INT_PARENT_NAME, ANCESTRY ],
  ENV_STACK
):-
  cg_comments(minor_header,accept, [task_entry, EXT_NAME : EXT_IDL_TAIL ], ''),
  /* Produce code for optional guard */
  (
    component([guard,GUARD_NAME, EXT_NAME : EXT_IDL_TAIL ],_) ->
      /* entry guarded */
      (
        cg_comments(simple_comment,guard_code,
          [guard,GUARD_NAME, EXT_NAME : EXT_IDL_TAIL ], ''),
        write_atom('when =>'),
        newline
      ); true /* entry unguarded */
    ),
  write_atom('accept '),write_atom( EXT_NAME),
  component([task_entry, EXT_NAME : EXT_IDL_TAIL ],

```

```

    ( [FORMAL_PARAMETER_SPECS_LIST],
    formal_part(FORMAL_PARAMETER_SPECS_LIST), write_atom(' do'), newline,
    begin_block,
    get_unique_key(KEY),
    parse_action_chains(
        KEY,
        [ task_entry, EXT_NAME, [ INT_PARENT_NAME::ANCESTRY ],
        ENV_STACK
    ),
    sequence_of_statements(KEY, [task_entry : _], _),
    end_block,
    write_atom('end '), write_atom(EXT_NAME), write_atom(' ;'), newline.

statement(
    [timed_entry, EXT_NAME : EXT_IDL_TAIL ],
    _
):-
    cg_comments(minor_header, timed_accept,
        [timed_entry, EXT_NAME : EXT_IDL_TAIL ], ''),
    /* Produce code for optional guard */
    (
        component([guard, GUARD_NAME, EXT_NAME : EXT_IDL_TAIL ], _) ->
            /* entry guarded */
            (
                cg_comments(simple_comment, guard_code,
                    [guard, GUARD_NAME, EXT_NAME : EXT_IDL_TAIL ], ''),
                write_atom('when      =>'),
                newline
            ); true /* entry unguarded */
    ),
    write_atom('delay -- delay code'),
    newline.

statement(_, _, _). /* catchall */

more_accepts([], _, _):- end_block.

more_accepts(
    [ ENTRY_IDL : ENTRY_IDLS_TAIL ],
    PARENT_IDL,
    ENV_STACK
):-
    end_block,
    write_atom('or '), newline,
    begin_block,

```

```

statement(
    ENTRY_IDL,
    PARENT_IDL,
    ENV_STACK
),
more_accepts(
    ENTRY_IDLS_TAIL,
    PARENT_IDL,
    ENV_STACK
).

/* dynamic task instantiations */
task_allocators( KEY ):-
    creation_chain( KEY, [ task, NAME : IDL_TAIL ] ),
    creation_info([ task, NAME : IDL_TAIL ], TASK_TYPE_NAME, PREFIX),
    write_atom('-- DYNAMIC TASK CREATION code!'), newline,
    write_atom(PREFIX), write_atom(NAME), write_atom(' := new '),
    write_atom(TASK_TYPE_NAME), write_atom(' ;'),
    newline, fail.

/* dynamic task_pool instantiations */
task_allocators( KEY ):-
    creation_chain( KEY, [ task_pool, NAME : IDL_TAIL ] ),
    creation_info([ task_pool, NAME : IDL_TAIL ], TASK_TYPE_NAME, PREFIX),
    write_atom('-- DYNAMIC TASKPOOL CREATION code!'), newline,
    write_atom('for I in '), write_atom(PREFIX), write_atom(NAME),
    write_atom(''RANGE loop'),
    newline,
    begin_block,
    write_atom(PREFIX), write_atom(NAME), write_atom(' (I) := new '),
    write_atom( TASK_TYPE_NAME ), write_atom(' ;'),
    newline,
    end_block,
    write_atom('end loop;'),
    newline, fail.

task_allocators(_).
/* procedure generate calls will succeed always. It will backtrack to find
   all interaction-chains for the given source object or component IDL */

generate_calls(KEY):-
    interaction_chain(KEY, ACTION),
    call(ACTION).

generate_calls(_). /* Termination condition when no calls or no more

```

calls found */

/* This procedure is incomplete with respect to comment generation for code */

```
interaction_call([procedure, NAME : IDL_TAIL], [wait, static, _], PREFIX):-
    write_atom( PREFIX), write_atom(NAME),
    component([procedure, NAME : IDL_TAIL],
    [ FORMAL_PARAMETER_SPEC_LISTS ]),
    actual_part(FORMAL_PARAMETER_SPEC_LISTS),
    newline,
    !,fail.
```

```
interaction_call(
    [pack_ext_procedure, NAME, [PARENT_NAME : LINEAGE_TAIL ]],
    [wait, static, _],
    PREFIX
):-
    write_atom( PREFIX), write_atom(PARENT_NAME), write_atom('.'), write_atom(NAME),
    component([pack_ext_procedure, NAME, [PARENT_NAME : LINEAGE_TAIL]],
    [ FORMAL_PARAMETER_SPEC_LISTS ]),
    actual_part(FORMAL_PARAMETER_SPEC_LISTS),
    newline,
    !,fail.
```

/* Rule for task_pool entry calls */

```
interaction_call([task_entry,NAME, [PARENT_NAME : LINEAGE_TAIL]], [wait, _, _],
    PREFIX
):-
    object([task_pool, PARENT_NAME, LINEAGE_TAIL], _),
    /*
    abstraction([task_access, PARENT_NAME, LINEAGE_TAIL], [static, _]), */
    component([task_entry, NAME, [PARENT_NAME : LINEAGE_TAIL]],
    [ FORMAL_PARAMETER_SPEC_LISTS ]),
    write_atom('— FINISH TASKPOOL call code'),
    newline,
    write_atom( PREFIX), write_atom(PARENT_NAME), write_atom('('),
    write_atom(NAME),
    actual_part(FORMAL_PARAMETER_SPEC_LISTS),
    newline,
    !,fail.
```

```
interaction_call([task_entry,NAME, [PARENT_NAME : LINEAGE_TAIL]], [wait, _, _],
    PREFIX
):-
    component([task_entry, NAME, [PARENT_NAME : LINEAGE_TAIL]],
    [ FORMAL_PARAMETER_SPEC_LISTS ]),
```

```

write_atom(PREFIX), write_atom(PARENT_NAME),
write_atom('.'), write_atom(NAME),
actual_part(FORMAL_PARAMETER_SPEC_LISTS),
newline,
!, fail.

/* Rule for calls to static task_pools */
interaction_call([task_entry, NAME, [PARENT_NAME : LINEAGE_TAIL]],
[no_wait, _, _], PREFIX
):-
    object([task_pool, PARENT_NAME, LINEAGE_TAIL], _),
/*
    abstraction([task_access, PARENT_NAME, LINEAGE_TAIL], [static, _]),*/
    component([task_entry, NAME, [PARENT_NAME : LINEAGE_TAIL]],
        [ FORMAL_PARAMETER_SPEC_LISTS ]),
    write_atom('select'), newline,
    begin_block,
    write_atom('-- FINISH TASKPOOL call code!'),
    newline,
    write_atom(PREFIX), write_atom(PARENT_NAME), write_atom(' ( ' ),
    write_atom(NAME),
    actual_part(FORMAL_PARAMETER_SPEC_LISTS),
    newline,
    write_atom('-- FINISH any additional sequence of statements code.'),
    newline, end_block,
    write_atom('else'), newline, begin_block,
    write_atom('-- FINISH alternative sequence of statements code.'),
    newline, end_block,
    write_atom('end select;'), newline,
    !, fail.

interaction_call([task_entry, NAME, [PARENT_NAME : LINEAGE_TAIL]],
[no_wait, _, _], PREFIX
):-
    component([task_entry, NAME, [PARENT_NAME : LINEAGE_TAIL]],
        [ FORMAL_PARAMETER_SPEC_LISTS ]),
    write_atom('select'), newline,
    begin_block, write_atom(PREFIX), write_atom(PARENT_NAME),
    write_atom('.'), write_atom(NAME),
    actual_part(FORMAL_PARAMETER_SPEC_LISTS),
    newline,
    write_atom('-- FINISH any additional sequence of statements code.'),
    newline, end_block,
    write_atom('else'), newline, begin_block,
    write_atom('-- FINISH alternative sequence of statements code.'),
    newline, end_block,
    write_atom('end select;'), newline.

```

```

!,fail.

/* Rule for calls to static task_pools */
interaction_call([task_entry,NAME, [PARENT_NAME : LINEAGE_TAIL]],
    [timed_wait,_,_], PREFIX
):-
    object([task_pool, PARENT_NAME, LINEAGE_TAIL], _),
/*
    abstraction([task_access, PARENT_NAME, LINEAGE_TAIL], (static, _)),*/
    component([task_entry, NAME, [PARENT_NAME : LINEAGE_TAIL]],
        [ FORMAL_PARAMETER_SPEC_LISTS ]),
    write_atom('select'),newline,
    begin_block,
    write_atom('-- FINISH TASKPOOL call code'),
    newline,
    write_atom(PREFIX), write_atom(PARENT_NAME),write_atom(' ( ).'),
    write_atom(NAME),
    actual_part(FORMAL_PARAMETER_SPEC_LISTS),
    newline,
    write_atom('-- FINISH any additional sequence of statements code.'),
    newline,end_block,
    write_atom('or'), newline, begin_block,
    write_atom('delay  -- FINISH delay statement code.'), newline,
    write_atom('-- FINISH any additional sequence of statements code.'),
    newline, end_block,
    write_atom('end select;'), newline,
    !,fail.

interaction_call([task_entry,NAME, [PARENT_NAME : LINEAGE_TAIL]],
    [timed_wait,_,_], PREFIX
):-
    component([task_entry, NAME, [PARENT_NAME : LINEAGE_TAIL]],
        [ FORMAL_PARAMETER_SPEC_LISTS ]),
    write_atom('select'),newline,
    begin_block, write_atom(PARENT_NAME), write_atom(' '), write_atom(NAME),
    actual_part(FORMAL_PARAMETER_SPEC_LISTS),
    newline,
    write_atom('-- FINISH any additional sequence of statements code.'),
    newline,end_block,
    write_atom('or'), newline, begin_block,
    write_atom('delay  -- FINISH delay statement code.'), newline,
    write_atom('-- FINISH any additional sequence of statements code.'),
    newline, end_block,
    write_atom('end select;'), newline,
    !,fail.

interaction_call([vis_data_area, NAME, [PARENT_NAME : _ ]],

```



```
[wait, static, _], PREFIX
):-
    write_atom('— FINISH call code for visible data area: '), write_atom(NAME),
    write_atom(' declared in package declaration for: '), write_atom(PREFIX),
    write_atom(PARENT_NAME), newline,
    !, fail.

interaction_call([inv_data_area, NAME, [PARENT_NAME : _ ]],
    [wait, static, _], PREFIX
):-
    write_atom('— FINISH call code for invisible data area: '), write_atom(NAME),
    write_atom(' declared in parent body: '), write_atom(PREFIX),
    write_atom(PARENT_NAME), newline,
    !, fail.

interaction_call(, , _):- /* Error message when no link to supported components or objects */
    write_atom('— INTERACTION chain TERMINATED on UNRECOGNIZED destination or'),
    write_atom(' INCONSISTENT interaction TYPES linked. NO codegeneration'),
    newline,
    !, fail.
```

H.12 SUBPGM BODY 6.3

```

/*****
$   FILE_NAME:      subpgm_decl_6.1

$   PROJECT_NAME:   CAEDE/1.2 ASCGEN Ada Skeleton Codegenerator

$   DESCRIPTION:    This file contains procedures to generate ada syntax
                    code.

$   PROCEDURES IN FILE:
        actual_parameter_spec,
        actual_part,
        data_area_parameters,
        formal_part,
        next_actual_parameter_spec,
        next_formal_parameter_spec,
        subprogram_spec,
        traceback_call,

$   HISTORY:

        date:      version:      by:      reason:
        -----
        15 Nov 84   CADA 1.1       K. Van Der Loo  freeze CADA 1.1
        21 Jan 85   CAEDE 1.X      "      "      remove begin_block,
end_block code from formal_parameter_spec,next_formal_parameter_spec.
        13 June 85   ASCGEN/1.2X   "      "      simplify data area parameters
to check final-destination interactions only for data area parameters.
        5 July 85   CAEDE_1.2x    "      "      add prefix parameter.

*****/

subprogram_specification([PROCEDURE_KIND,NAME : IDL_TAIL], PREFIX):-
/* check that this is not a function specification */
((PROCEDURE_KIND = procedure) ; (PROCEDURE_KIND = pack_ext_procedure)),
write_atom('procedure '), print(PREFIX), print(NAME),
component([PROCEDURE_KIND,NAME : IDL_TAIL],
(FORMAL_PARAMETER_SPEC_LISTS)),
formal_part(FORMAL_PARAMETER_SPEC_LISTS).

```

```

formal_part([]):- ' /* no formal parameters implied by an empty list. */

formal_part(PARAMETER_SPEC_LISTS):-
    write_atom('('),
    formal_parameter_spec(PARAMETER_SPEC_LISTS),
    write_atom(')').

formal_parameter_spec([[NAME,MODE,TYPE] : PARAMETER_SPEC_LISTS_TAIL]):-
    write_atom(' '),print(NAME),write_atom(' : '),print(MODE),write_atom(' '),print(TYPE),
    next_formal_parameter_spec(PARAMETER_SPEC_LISTS_TAIL).

next_formal_parameter_spec([]). /* no more parameters termination condition */

next_formal_parameter_spec([[NAME,MODE,TYPE] : PARAMETER_SPEC_LISTS_TAIL]):-
    write_atom('; '),
    (text_mode(temporary_comment) -> newline ; true),
    print(NAME),write_atom(' : '),print(MODE),write_atom(' '),
    print(TYPE), write_atom(' '),
    next_formal_parameter_spec(PARAMETER_SPEC_LISTS_TAIL).

/* TEMPORARY CODE to print all data flow specs associated with the given
   data area. */

data_area_parameters(DESTINATION_IDL):-
    reference([interaction : _],SOURCE_IDL,DESTINATION_IDL,
              [_,_,DATA_FLOW_LIST]),
    data_area_parameter_spec(DATA_FLOW_LIST).

data_area_parameters(_).

/* TEMPORARY CODE to print data area parameters */
data_area_parameter_spec([]):- ' ,fail.

data_area_parameter_spec([[NAME,MODE,TYPE] : PARAMETER_SPEC_LISTS_TAIL]):-
    write_atom(' '),print(NAME),write_atom(' : '),print(MODE),write_atom(' '),print(TYPE),
    write_atom('; '),
    !,
    data_area_parameter_spec(PARAMETER_SPEC_LISTS_TAIL).

/* TEMPORARY CODE to print data area parameters */
traceback_call(DESTINATION_IDL):-

```

```

reference([interaction : _], [pack_ext_procedure : _], DESTINATION_IDL,
          [_,_, DATA_FLOW_LIST ]),
data_area_parameter_spec(DATA_FLOW_LIST),
!,fail.

traceback_call(DESTINATION_IDL):-
reference([interaction : _], [procedure : _], DESTINATION_IDL,
          [_,_, DATA_FLOW_LIST ]),
data_area_parameter_spec(DATA_FLOW_LIST),
!,fail.

traceback_call(DESTINATION_IDL):-
reference([interaction : _], [task_entry : _], DESTINATION_IDL,
          [_,_, DATA_FLOW_LIST ]),
data_area_parameter_spec(DATA_FLOW_LIST),
!,fail.

traceback_call(DESTINATION_IDL):-
reference([interaction : _], [timed_entry : _], DESTINATION_IDL,
          [_,_, DATA_FLOW_LIST ]),
data_area_parameter_spec(DATA_FLOW_LIST),
!,fail.

traceback_call(DESTINATION_IDL):-
reference([interaction : _], [main_line : _], DESTINATION_IDL,
          [_,_, DATA_FLOW_LIST ]),
data_area_parameter_spec(DATA_FLOW_LIST),
!,fail.

traceback_call(DESTINATION_IDL):-
reference([interaction : _], [hardware : _], DESTINATION_IDL,
          [_,_, DATA_FLOW_LIST ]),
data_area_parameter_spec(DATA_FLOW_LIST),
!,fail.

traceback_call(DESTINATION_IDL):-
reference([interaction : _], SOURCE_IDL, DESTINATION_IDL,
          [_,_, DATA_FLOW_LIST ]),
data_area_parameter_spec(DATA_FLOW_LIST),
!,
traceback_call(SOURCE_IDL). /* This goal will fail when a legitimate
source is recognized. It's assumed that all interactions have
legitimate sources, however incomplete interactions will be
traced back to the first break without complaint */

actual_parameters([]):- /* no actual parameters implied by an empty list.*/

```

```
write_atom(' ;').

actual_part(PARAMETER_SPEC_LISTS):-
    write_atom('('),
    actual_parameter_spec(PARAMETER_SPEC_LISTS),
    write_atom(' ;').

actual_parameter_spec([([NAME,MODE,TYPE] : PARAMETER_SPEC_LISTS_TAIL):-
    write_atom(' '),print(NAME),write_atom(' => '),
    next_actual_parameter_spec(PARAMETER_SPEC_LISTS_TAIL).

next_actual_parameter_spec([]). /* no more parameters termination condition */

next_actual_parameter_spec([([NAME,MODE,TYPE] : PARAMETER_SPEC_LISTS_TAIL):-
    write_atom(' '),print(NAME),write_atom(' => '),
    next_actual_parameter_spec(PARAMETER_SPEC_LISTS_TAIL).
```

H.13 TASK TYPE DECL

```
task_type_declarations( PARENT_INT_IDL, CLASSIFIED_G_INSTANCES ):-
```

```

just_get_type_name(
    PARENT_INT_IDL,
    task_template_declaration,
    task_template,
    TEMPLATE_NAME,
    [TASK_KIND : IDL_TAIL], /* This will be
        instantiated to the first identical instance
        for this type category */
    CLASSIFIED_6_INSTANCES,
    TASK_TYPE_NAME
),
(TASK_KIND = task ; TASK_KIND = task_pool),
task_type_specification(
    [ TASK_KIND : IDL_TAIL],
    TASK_TYPE_NAME
),
write_atom(' ;'),
newline,
write_atom('type '),
write_atom(TASK_TYPE_NAME),
write_atom('_ACCESS'),
write_atom(' is access '), write_atom(TASK_TYPE_NAME), write_atom(' ;'),
newline,
assertz(
    type_body_description(
        PARENT_INT_IDL,
        TEMPLATE_NAME,
        [TASK_KIND : IDL_TAIL],
        TASK_TYPE_NAME
    )
),
fail.

```

```
task_type_declarations(, ). /* catchall when no more task type
    declarations required. */
```

H.14 TASK TYPE BDY

```

task_type_bodies( PARENT_INT_IDL, ENV_STACK ):-
    cg_comments_description(
        task_type,
        PARENT_INT_IDL,
        TEMPLATE_NAME
    ),
    cg_comments(
        major_header,
        task_template_body,
        [task_template, TEMPLATE_NAME, _],
        ''
    ),
    type_body_description(
        PARENT_INT_IDL,
        TEMPLATE_NAME,
        INSTANCE_EXT_IDL,
        TASK_TYPE_NAME
    ),
    task_type_body(
        TASK_TYPE_NAME,
        [task_template, TEMPLATE_NAME, _],
        [environment( '', INSTANCE_EXT_IDL ) : ENV_STACK ]
    ),
    newline, fail.

task_type_bodies( _, _ ) /* catchall when no more task type bodies
    required. */

```

H.15 TASKS 9.1

```

/*****

```

```

$ FILE_NAME:      task_specs_bodies_9.1

$ PROJECT_NAME:   CAEDE/1.2 ASCGEN Ada Skeleton Codegenerator

$ DESCRIPTION:    This file contains code to generate ada syntax code.

$ PROCEDURES IN FILE:
    task_specification,
    task_body.

```

```

$ HISTORY:

```

date:	version:	by:	reason:
15 Nov 84	CADA 1.1	K. Van Der Loo	freeze CADA 1.1
15 June 85	ASCGEN/1.2X	" "	add PREFIX

```

to various calls

```

```

-----
*/

```

```

task_type_specification(
    [TASK_OR_TASKPOOL, EXT_NAME : EXT_IDL_TAIL],
    TASK_TYPE_NAME
):-
    write_atom('task type '),
    write_atom(TASK_TYPE_NAME), write_atom(' is'), newline,
    begin_block,
    abstraction([task externals, EXT_NAME : EXT_IDL_TAIL],
        [ EXTERNAL_COMPONENTS_LISTS ]),
    basic_declarative_item(EXTERNAL_COMPONENTS_LISTS, ''),
    end_block,
    write_atom('end '), write_atom(TASK_TYPE_NAME),
    !.

```

```

task_specification(
    [TASK_OR_TASKPOOL, EXT_NAME : EXT_IDL_TAIL],
    PREFIX
):-
    write_atom('task '),
    write_atom(PREFIX), write_atom(EXT_NAME), write_atom(' is'), newline,

```



```

begin_block,
abstraction([task externals, EXT_NAME : EXT_IDL_TAIL],
            [ EXTERNAL_COMPONENTS_LISTS ]),
basic_declarative_item(EXTERNAL_COMPONENTS_LISTS, ''),
end_block,
write_atom('end '), write_atom(PREFIX), write_atom(EXT_NAME).

```

```

task_type_body(
    TASK_TYPE_NAME,
    [ INT_TASK_KIND, INT_NAME : INT_IDL_TAIL ],
    [ environment( EXT_PREFIX, [ EXT_TASK_KIND, EXT_NAME : EXT_IDL_TAIL ])
      : ENV_STACK ]
):-
    get_unique_key(KEY),
    parse_action_chains(
        KEY,
        [main_line, INT_NAME : INT_IDL_TAIL],
        [
            environment(
                '',
                [INT_TASK_KIND, INT_NAME : INT_IDL_TAIL ]
            ),
            environment(
                EXT_PREFIX,
                [ EXT_TASK_KIND, EXT_NAME : EXT_IDL_TAIL ]
            )
            : ENV_STACK
        ]
    ),
    write_atom('task body '), write_atom( TASK_TYPE_NAME ), write_atom(' is'),
    newline,
    begin_block,
    ((declarative_part(
        [ INT_TASK_KIND, INT_NAME, []],
        [environment(
            EXT_PREFIX,
            [EXT_TASK_KIND, EXT_NAME : EXT_IDL_TAIL ])
          : ENV_STACK ]
    ))
    ; true
    ),
    end_block,
    write_atom(begin), newline,
    begin_block,
    sequence_of_statements(
        KEY,

```

```

        [ INT_TASK_KIND, INT_NAME : INT_IDL_TAIL ],
        [
            environment(
                "",
                [INT_TASK_KIND, INT_NAME : INT_IDL_TAIL ]
            ),
            environment(
                EXT_PREFIX,
                [ EXT_TASK_KIND, EXT_NAME : EXT_IDL_TAIL ]
            )
        ] ENV_STACK
    ],
end_block,
write_atom('end '), write_atom(TASK_TYPE_NAME), write_atom(' ;'),
newline,
!.

task_body(
    [ INT_TASK_KIND, INT_NAME : INT_IDL_TAIL ],
    [ environment( EXT_PREFIX, [ EXT_TASK_KIND, EXT_NAME : EXT_IDL_TAIL ])
      : ENV_STACK ]
):-
    /* called for static non_instantiated tasks or task type body
       generation */
    get_unique_key(KEY),
    parse_action_chains(
        KEY,
        [main_line, INT_NAME : INT_IDL_TAIL],
        [
            environment(
                "",
                [INT_TASK_KIND, INT_NAME : INT_IDL_TAIL ]
            ),
            environment(
                EXT_PREFIX,
                [ EXT_TASK_KIND, EXT_NAME : EXT_IDL_TAIL ]
            )
        ] ENV_STACK
    ),
    cg_comments(major_header, task_body, [ EXT_TASK_KIND,
        EXT_NAME : EXT_IDL_TAIL ], EXT_PREFIX ),
    write_atom('task body '), write_atom( EXT_PREFIX ),
    write_atom( EXT_NAME ), write_atom(' is!'),
    newline,

```

```
begin_block,
((declarative_part(
  [ INT_TASK_KIND, INT_NAME : INT_IDL_TAIL],
  [environment(
    EXT_PREFIX,
    [EXT_TASK_KIND, EXT_NAME : EXT_IDL_TAIL ]
  ] ENV_STACK ]
))
; true
),
end_block,
write_atom(begin),newline,
begin_block,
sequence_of_statements(
  KEY,
  [ INT_TASK_KIND, INT_NAME : INT_IDL_TAIL],
  [
    environment(
      "",
      [INT_TASK_KIND, INT_NAME : INT_IDL_TAIL ]
    ),
    environment(
      EXT_PREFIX,
      [ EXT_TASK_KIND, EXT_NAME : EXT_IDL_TAIL ]
    )
  ] ENV_STACK
],
end_block,
write_atom('end '), write_atom(EXT_PREFIX), write_atom( EXT_NAME),
write_atom(' ;'), newline,
!).
```

END

06-07-89

FIN