1

**PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT**

PRECISIONSM RESOLUTION TARGETS

Practical Automatic Code Generation

by

ANDREW J. FORREST

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

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6

© Copyright
1996 Andrew J. Forrest

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa (Ontario)
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395 rue Wellington
Ottawa (Ontario)
K1A 0N4

Author's Acknowledgement

Author's Acknowledgement

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

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

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

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

ISBN 0-612-19491-4

Canada

Name Andrew Forrest

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided

0984

U-M-I

SUBJECT TERM

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
General 0289
Ancient 0290
Linguistics 0291
Modern 0291
Literature 0401
General 0294
Classical 0295
Comparative 0297
Medieval 0298
Modern 0316
African 0591
American 0305
Asian 0352
Canadian (English) 0355
Canadian (French) 0593
English 0311
Germanic 0312
Latin American 0315
Middle Eastern 0313
Romance 0314
Slavic and East European

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion 0318
General 0321
Biblical Studies 0319
Clergy 0320
History of 0322
Philosophy of 0469
Theology

SOCIAL SCIENCES

American Studies 0323
Anthropology 0324
Archaeology 0326
Cultural 0327
Physical 0310
Business Administration 0272
General 0770
Accounting 0454
Banking 0338
Management 0385
Marketing 0501
Canadian Studies 0503
Economics 0505
General 0508
Agricultural 0509
Commerce Business 0510
Finance 0511
History 0358
Labor 0366
Theory 0351
Folklore 0578
Geography
Gerontology
History
General

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0337
United States 0585
History of Science 0398
Law
Political Science
General 0615
International Law and Relations 0616
Public Administration 0617
Recreation 0814
Social Work 0452
Sociology
General 0626
Criminology and Penology 0627
Demography 0938
Ethnic and Racial Studies 0631
Individual and Family Studies 0628
Industrial and Labor Relations 0629
Public and Social Welfare 0630
Social Structure and Development 0700
Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy
Animal Culture and Nutrition 0475
Animal Pathology 0476
Food Science and Technology 0359
Forestry and Wildlife 0478
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0817
Range Management 0777
Wood Technology 0746
Biology
General 0306
Anatomy 0287
Biostatistics 0308
Botany 0309
Cell 0379
Ecology 0329
Entomology 0353
Genetics 0369
Limnology 0793
Microbiology 0410
Molecular 0307
Neuroscience 0317
Oceanography 0416
Physiology 0433
Radiation 0821
Veterinary Science 0778
Zoology 0472
Biophysics
General 0786
Medical 0760
EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Polymology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
General 0566
Audiology 0300
Chemotherapy 0992
Dentistry 0567
Education 0350
Hospital Management 0769
Human Development 0758
Immunology 0982
Medicine and Surgery 0564
Mental Health 0347
Nursing 0569
Nutrition 0570
Obstetrics and Gynecology 0380
Occupational Health and Therapy 0354
Ophthalmology 0381
Pathology 0571
Pharmacology 0419
Pharmacy 0572
Physical Therapy 0382
Public Health 0573
Radiology 0574
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences
Chemistry
General 0485
Agricultural 0749
Analytical 0486
Biochemistry 0487
Inorganic 0488
Nuclear 0738
Organic 0490
Pharmaceutical 0491
Physical 0494
Polymer 0495
Radiation 0754
Mathematics 0405
Physics
General 0605
Acoustics 0986
Astronomy and Astrophysics 0606
Atmospheric Science 0608
Atomic 0748
Electronics and Electricity 0607
Elementary Particles and High Energy 0798
Fluid and Plasma 0759
Molecular 0609
Nuclear 0610
Optics 0752
Radiation 0756
Solid State 0611
Statistics 0463
Applied Sciences
Applied Mechanics 0346
Computer Science 0984

Engineering
General 0537
Aerospace 0538
Agricultural 0539
Automotive 0540
Biomedical 0541
Chemical 0542
Civil 0543
Electronics and Electrical 0544
Heat and Thermodynamics 0348
Hydraulic 0545
Industrial 0546
Marine 0547
Materials Science 0794
Mechanical 0548
Metallurgy 0743
Mining 0551
Nuclear 0552
Packaging 0549
Petroleum 0765
Sanitary and Municipal System Science 0554
System Science 0790
Geotechnical 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

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

Practical Automatic Code Generation

submitted by Andrew J. Forrest, P. Eng. in partial fulfillment of the requirements for the degree of Master of Engineering.

Chair.
Department of Systems and
Computer Engineering

Dr. Gerald Karam,
Thesis Supervisor

Department of Systems and Computer Engineering
Carleton University
April 8, 1996

Abstract

This thesis examines the development of features for an Automatic Code Generation facility that enhance its practicality. Although the potential for automatic generation of high-level computer language source code from design representations (Automatic Code Generation) has been demonstrated, it is not in wide-spread use; the reasons for this revolve around the practicality of the code generators. By addressing this aspect of Automatic Code Generation directly, such facilities can be made more useable, thereby facilitating the realization of benefits such as improved software quality and developer productivity.

A set of requirements is developed in this work that describes the necessary elements of a practical Automatic Code Generator and is based on both the principal objections to Automatic Code Generation and the goals of implementing quality software. An existing design environment with code generation (*TimeBench*) is then extended by providing a code generator targeted for the C programming language and the Unix operating system with a number of new features (*User Annotations*) that address these requirements. Finally, evaluation criteria for assessing code generator practicality are developed and applied in a case study.

Acknowledgements

I would like to thank Can Aysan, Greg Franks and Alex Hubbard for their patience and for helping me with understanding TimeBench. Without their support I would no doubt have not progressed as quickly with this research.

Next, I wish to thank Prof. Gerald Karam for his practical and insightful advice for my own research, his guidance with respect to the literature of related fields of study and ultimately, for teaching me the true value of a master's degree.

I am also indebted to my family and friends for their unflagging encouragement and understanding during the course of this research. Your patience and support are greatly appreciated.

Finally, I would like to thank the Natural Science and Engineering Research Council (NSERC) and the Department of Systems and Computer Engineering at Carleton University for their help through partial funding of this research.

Table of Contents

Abstract	iii
Acknowledgements	iv
Table of Contents	v
List of Tables	vii
List of Figures	viii
 CHAPTER I: Introduction	 1
1.1 Motivation	1
1.2 Research Objectives	4
1.3 Thesis Organization	5
 CHAPTER II: Background	 7
2.1 Related Fields of Study	7
2.2 MachineCharts: A Design Notation	12
2.3 A Brief Introduction to MetaTranslation	17
2.4 MetaGen	18
2.5 TimeBench: A Framework for Software Tool Exploration	20
2.6 TimeBench/MetaGen Architecture	20
2.7 The TimeBench Design Tree	22
2.8 Some Commercial Offerings	24
 CHAPTER III: A Step Towards Usable Automatic Code Generation	 26
3.1 Defining the Problem	26
3.2 Requirements for a Practical Automatic Code Generator	31
3.2.1 Principal Objections to Automatic Code Generation	32
3.2.2 Aspects of Software Quality	34
3.2.3 Detailed Requirements Analysis	37
3.3 Design of a Practical Automatic Code Generator	43
3.3.1 Techniques for Achieving Software Quality.	46
3.3.2 Design of Features	50
3.3.3 New Automatic Code Generator Features	54
3.4 Implementation of New Features	57
3.4.1 Details of Additional Features	57
3.4.2 Example of Adding an Annotation	69

CHAPTER IV: Evaluation Criteria	73
4.1 Introduction	73
4.2 From Requirements to Evaluation Criteria	73
CHAPTER V: Case Study: Communicating Stacks of Modules	82
5.1 The Case Study Design Model	82
5.2 Annotating the Design	95
5.2.1 Partitioning the Implementation	95
5.2.2 Automatic ADT Implementation Selection	95
5.2.3 Incorporating Foreign Source Code	98
5.2.4 Incorporating Foreign Object Code	99
5.3 Code Generator Evaluations	101
CHAPTER VI: Conclusion	107
6.1 Research Summary	107
6.2 Future Research	109
APPENDIX A: MetaGen Specification Language	118
APPENDIX B: C/JNX Generated Code	122
APPENDIX C: C/Unix Generated Code	145
APPENDIX D: IPC Library for Unix	170

List of Tables

1	Level of Abstraction Supported by Various Programming Technologies	2
2	Classes of Information Required to Implement a Design	29
3	Factors Affecting Translation Strategy	29
4	Descriptions of the Characteristics of Quality Software	35
5	Requirements for a Practical Automatic Code Generator.....	37
6	Features added to TimeBench/MetaGen	55
7	Requirements Traceability Matrix	56
8	List of Current Pragmas.....	62
9	Evaluation Criteria for a Practical Automatic Code Generator.....	74
10	Comparison of Original and New Code Generators	102
11	Degree of Improvement by Evaluation Criterion.....	108
12	Some Areas for Improvement in the C/Unix Code Generator	109

List of Figures

1	The Producer/Consumer MachineChart.....	14
2	The Internal Design of the Buffer Component.....	16
3	The Agenda of the Buffer Component's Engine	16
4	The Agenda of the Tx Gate's Engine.....	17
5	The Declarations within the BufferStore DataStore.....	17
6	A Translation Rule Expressed in MetaGen Specification Language	19
7	Architectural Overview of TimeBench's Code Generation Facility.....	22
8	Partial Design Tree for the Producer/Consumer Design (Schematic).....	24
9	Inadequate Transformation Process.....	44
10	Augmented Transformation Process.....	45
11	Architecture of Improved TimeBench Code Generation Facility.....	53
12	Example ADT Usage in MCL.....	58
13	Appearance of the File Name Annotation.....	60
14	Schematic Representation of Filename Assignment to Design Components.....	61
15	Example of the \$include Pragma (Before).....	63
16	Implementation of Push Button next - No Foreign Source Included.....	63
17	Example of the \$include Pragma (After).....	64
18	Foreign Source to be Included in the Implementation of next.	64
19	Implementation of Push Button next - Foreign Source Included.	65
20	Specifying Task Priorities.....	66
21	Example of the \$stub Pragma (Before)	68
22	Implementation of Push Button sink - Before Stubbing.	68
23	Example of the \$stub Pragma (After)	69
24	Code for enclosingFileName	71
25	Stack of Communicating Components (Top-Level MachineChart)	84
26	The Src Box (MachineChart).....	85
27	The S_task Actor (MachineChart).....	86
28	The Agenda of S_task's Engine	86
29	The Contents of Datastore I (MCL code).....	87
30	The Agenda of the S_d Gate Button's Engine (MCL code).....	87
31	The courier1 Actor (MachineChart).....	87
32	The Agenda of Courier1's Engine (MCL code).....	88
33	The Comm1 Box (MachineChart).....	89
34	The Comm1 Actor (MachineChart)	90
35	The Altbit_1 Abstract Controller Machine (MachineChart).....	91

List of Figures (continued)

36	The States and Actions of the ACM Altbit_1 (MCL code).....	91
37	The Agenda of Perform1's Engine (MCL code).....	92
38	The Agenda of S1_c1's Engine (MCL code)	93
39	The Timer Actor (MachineChart).....	94
40	The Courier3 Actor (MachineChart)	94
41	s_task - Fragment of Source Generated by CUnix_codegen.....	96
42	d_task - Fragment of Source Generated by CUnix_codegen.....	97
43	Agenda for Push Button next.....	99
44	generate_next.c - Foreign Source to be Included in the Implementation.	99
45	next_Button - Source Fragment Generated by CUnix_codegen.	99
46	Agenda for Push Button sink.....	100
47	sink_Button stub- Source Fragment Generated by CUnix_codegen.....	100
48	Makefile - Fragment Generated by CUnix_codegen.....	101
49	absorb.c - Source for Foreign Object Code.	101
50	Configuration of Conventional Implementation Performance Tuning.....	112
51	Possible Configuration of ADT Implementation Selection based on Predicted Performance	113
52	commdemo.c - Source Generated by CJNX_codegen for Case Study.	122
53	commdemo_main.c - Source Generated by CUnix_codegen for Case Study.	145
54	comm1 - Source Generated by CJNX_codegen for Case Study.	146
55	source - Source Generated by CUnix_codegen for Case Study.....	159
56	dest - Source Generated by CUnix_codegen for Case Study.....	163
57	courier3 - Source Generated by CUnix_codegen for Case Study.	167
58	Makefile - Source Generated by CUnix_codegen for Case Study.....	168
59	ipc.h - Interface for the Inter-Process Communication Library	170
60	ipclib.c - Body of the InterProcess Communication Library	171

1.1 Motivation

Since the beginning of computing, there has been keen interest and anticipation in having computers assist in the performance of intelligent activities – having them extend the power of the human mind. Nowhere is the need for this assistance greater or more appropriate than in the area of software development itself, for no sooner had the field been established than it fell into a crisis (the *software problem*) over development productivity and software quality [23]. Productivity in software development is thought to trail what it should (or could) be, while software quality has also been disappointing. It is thought that this is because *manual* software development has many drawbacks. Chief among them are: informal or ad hoc procedures – processes which are poorly understood; and transformations that are difficult to characterize, reproduce and examine [48]. The consequences of these flaws reveal themselves through errors appearing in the finished (and intermediate) product(s) of software development and the perceived poor productivity of software developers.

It is only natural that the idea of applying the strengths of computers to the software problem should arise and fascinate workers in the field of software engineering; by automating as many phases or aspects of software development as possible, productivity and quality gains should be achieved. As for what to automate, a practical approach is to make evolutionary changes to what we do today. For example, it is highly unlikely that a programmer today would seriously consider embarking on a large project by coding machine language in hexadecimal. In fact, it is unlikely that he or she would even entertain the thought of using an assembly language at all for the entire project. Without knowing

more about the requirements of this hypothetical project, it can be assumed that the first choice for expressing the program would be what is known as a *high level language* (HLL). This represents the use of technology to automate the final translation from a program expressed in a high-level human-readable language to a binary representation suitable for execution on a processor.

This trend toward the high-level language and beyond is shown in table 1 which depicts a progression of software development technology increasing the level of abstraction that can be maintained by the programmer and in shortening the time required to create an executable program. The logical next step in the progression from hexadecimal to mnemonics, from mnemonics to HLL is to dispense with *coding* entirely and stop after specifying the design of a software system and possibly some implementation constraints.¹ This incremental, but significant change extends the support for automation backwards from compilation to implementation, thereby enabling software developers to shift their attention to higher levels of abstraction – or equivalently – earlier phases of the software life-cycle.

TABLE 1. Level of Abstraction Supported by Various Programming Technologies

Programming Technology	Conceptual Innovation Supported
Machine Language (hexadecimal)	Negligible
Machine Language (mnemonic)	Symbolic Representation
High Level Language	Abstract Data Types, Structured Flow, Object Orientation
Automatic Programming	Specify the Design, Specify the target constraints and have the implementation produced automatically.

1. Until such time as this technology is in wide-spread use, it is unrealistic to expect that coding will disappear completely. The sheer amount and value of legacy code in the world today suggests a hybrid scenario at best.

The specific field that addresses automating the implementation activity, *Automatic Programming* (AP), was born shortly after computing itself [4, 43] but has so far failed to have wide-spread success [50]. The successes that have been enjoyed in AP have generally been associated with systems which served a narrow, well-understood application domain. Nevertheless, systems built with automatic programming technology promise to be more easily maintained, more re-usable, more quickly developed and to possess higher quality [41]. These accomplishments are thought to be possible because automatic implementation could produce a rigorous transformation from design to implementation, one that is repeatable and can, in principle, be proved correct. A goal of automatic code generation technology is to enable developers to test, maintain and reuse at the design level, all of which is expected to improve the productivity of software developers and the quality of their products.

The productivity of software developers can be enhanced by automatic code generation in several ways: automating the implementation phase can shorten development time; formal transformation can lessen the testing burden [54]; and, perhaps most importantly, facilitating the reuse of software development artifacts (designs and their representations) eliminates the need to perform certain activities. Software quality improvements, which are thought to be an even more compelling motivation for this work [23], are also possible because automatic code generation facilitates software reuse [23] – arguably the most significant contributor to achieving gains in software quality [23]. Other ways that automatic code generation contributes to software quality include: permitting rapid production of alternative implementations which can be individually evaluated; enabling a shift in resources from later development activities to earlier ones wherein the introduction of undetected flaws is more costly and more difficult to overcome.

1.2 Research Objectives

The problem this work seeks to solve is to make automatic code generation a viable alternative to manual implementation for software systems because, despite all their promise, automatic code generators are not in wide-spread use. This unfortunate situation is the result of existing generators falling short in a number of areas: efficiency of the generated code; integrability of the generator or generated code with existing code or development procedures including configuration management, testing and code inspections; the breadth of scope of the application domain; and, ease of use. These shortcomings can be considered jointly in terms of the effects that they have on the practicality of automatic code generators.

In this work, the requirements for a practical automatic code generator are derived and presented in detail. The derivation of these requirements takes two phases: how implementation should be automated; and, how automatic implementation should be made practical. The latter phase receives the most attention in this work drawing on a number of sources. Once the requirements are known, a working implementation of a design tool that incorporates features addressing these requirements is produced. In this work the design tool that is produced is the extension of an existing design tool with automatic code generation capability – TimeBench [8]. The additional features are designed to meet the requirements identified and improve the practicality of using TimeBench for automatic code generation. Among the features added to TimeBench are: a code generator targeted for the C programming language and the Unix operating system; an interprocess communication library for the new C/Unix target; a number of code generation pragmas to be included in the design language; the ability to suppress code generation for certain design components selectively (used to model existing environmental components in a design); the ability to selectively include explicit user code expressed in the target language; the

ability to control the mapping of generated source to files and the names of these files; the ability to direct an intelligent incremental compilation of only the sources affected by a change; and, the ability to edit and integrate externally provided source code.

In addition to deriving requirements for, and producing a model of a practical automatic code generator, a set of evaluation and cataloguing criteria have been developed to assess automatic code generator capability and practicality. These criteria are derived largely from the requirements, but stand alone as a method suitable for automatic code generator assessment. To illustrate the value of the additions to TimeBench made as part of this work, a case study is undertaken wherein a medium-sized problem, representative of the kind at which TimeBench is aimed, is implemented. The case study problem is implemented with both the prior TimeBench and the one with a more practical automatic code generator. The code generator and the implementations are evaluated using the criteria mentioned earlier.

1.3 Thesis Organization

In chapter 2, background information on a number of topics is provided. Among the subjects covered are several fields of study related to this work: *program transformation*; *code optimization*; *automatic generation of code generators and optimizers*, *visual programming*; and, *automatic programming*. In addition to these topics, the *MachineChart* design notation, *metatranslation*, and the *TimeBench* design tool are all introduced with some emphasis on the internal representation of MachineCharts within TimeBench. Some commercial products which attempt to address the goals of this work are reviewed briefly at the close of chapter 2.

A more complete and formal problem statement appears in the early part of chapter 3 along with a detailed list and discussion of the requirements for a practical code gen-

Reference [43] contributes greatly in the area of automatic programming by introducing the idea of "cliches" which are idiomatic collections of code. This is a grouping that exists above the level of individual language statements and can include things such as algorithms and data structures. Unfortunately, decisions concerning implementation are left to the programmer in an unspecified way in [44] — which is exactly the problem we seek to solve.

2.2 MachineCharts: A Design Notation

MachineCharts [9, 10, 11] are chiefly concerned with the architectural details of a real-time system's design. Structural and temporal features of the design such as encapsulating components, parallelism and communication paths figure prominently in its visual vocabulary. Entities in a design are represented visually by either nodes or arcs of various kinds. Nodes are composed of further components which may be *internal* in which case their existence and identity is not known outside the containing node's boundaries (depicted by drawing the internal component wholly within and not touching the enclosing node's boundaries), or they may be *interface* components in which case their existence and identity is known outside the containing node (depicted by drawing the interface component within, but touching, the containing node's boundaries). Arcs serve to connect nodes together to indicate interactions between components. Detailed behaviour can be specified for a MachineChart design by either of two mechanisms: *Abstract Controller Machines* (ACMs) or *Engines*. ACMs are used to encapsulate *Finite State Machines* (FSMs) whereas engines have their behaviour specified by an agenda, the form of which is not dictated by the MachineChart design notation.

TimeBench, since it provides an animated interpretive execution of designs, dictates that the agenda be expressed in a design language which is called *MachineChart Language* (MCL). MCL is a high level, expressive but economic and loosely typed lan-

In this research, an existing software design tool called *TimeBench* [8] with code generation capability [48] is modified to enhance its practicality. In order to better appreciate the contribution of this research, it is useful to consider some background material which helps to place the technique and results in a broader context. Among the background elements that are considered in this chapter are: related fields of study [4, 15, 29], a design notation known as *MachineCharts* [9, 10, 11], a concept known as *metatranslation*, the TimeBench software design tool itself and its current code generator generation facility, MetaGen [47, 48].

2.1 Related Fields of Study

The topic of practical automatic code generation [2, 3] draws on several distinct, but related areas. For the most part, these areas are characterized by systems which translate a finite input (source) into a finite output (target). Mathematically, the general case can be described by the *Transformation Mapping*:

$$f:S \rightarrow T \quad (\text{EQ 1})$$

where f is the function of some transformation system that takes inputs from S and maps them onto outputs from T . Examples of these kinds of systems are found in the fields of: *program transformation; code optimization; automatic generation of code generators and optimizers, visual programming; and, automatic programming.*

Program Transformation is concerned with changing programs from a source representation to a target representation either through *translation* or *refinement*. In the nota-

tion of equation 1, S is the set of programs expressed in a language L_S while T is the set of programs expressed in language L_T . Both source and target representations can be different high level language source codes (i.e. $L_S \neq L_T$) in which case the transformation is usually referred to as a translation and is typically motivated by the desire to reuse software. An example of this case is the translation of a program written in Pascal to one written in Ada or vice versa. Alternatively, program transformation can also be concerned with the *refinement* of a source representation ($L_S \sim L_T$). This refinement usually takes the form of either an optimization of some kind, a specialization of a general representation or an expansion of some complex computation in terms of more basic elements. Subexpression elimination at the source level, instantiation of a generic or template type and pre-processing C++ to obtain C are, respectively, examples of this kind of transformation. The term *concretization* is sometimes used to refer to the production of a specific target representation out of a set of possible target representations, all of which are associated with a particular source representation. An example of *concretization* is the inclusion of source by the C pre-processor in many C programs where the inclusion is conditional upon the host or target operating system.

Comments on the problems faced by source to source translators are made in [36]. Basically, source to source translation is useful only where (1) efficiency and/or maintenance are of no concern, and/or (2) hand editing is acceptable. This is probably because *implementing spreads knowledge*; that is, the idiomatic use of a programming language obscures the design goal or higher-level construction that is being implemented. This makes a rote translation (at a low level) of one programming language to another language prone to producing code that is not idiomatic in the target language and is thus difficult to understand. This problem is magnified by the syntactic and/or semantic distance between the source and target languages.

Promise appears in [32] where general purpose programs (*generic programs*) are transformed (*concretized*) into multiple special purpose programs where the criteria for concretization may be given as memory, time or reliability. The approach is close to our needs but the level from which the transformations are applied is too low. It relies on heavily annotated source programs which often exceed the target program in size thus negating much of the advantage of automatic high-level language code generation. Because the source and target of the translation are "at the same level" of detail, program transformation has little to offer in *implementing* designs.

Code optimization has as its goal the improvement of code quality through transformation without affecting code correctness. That is, optimization seeks to transform a specification into another representation (the implementation perhaps) without violating the meaning of the specification and in a way that improves on some measure of code quality as compared with a naive or straight-forward transformation. The most popular measures of code quality are size and execution speed, although others are possible.

In general, optimization techniques (perhaps necessarily) focus on the translation problem at a very low level. Even so called "global" techniques really act in a highly local manner when effecting their optimizations. One possible application of these techniques is to use some of the global measurements to decide between implementation alternatives. For purposes of implementing systems from designs, most code optimization techniques are too narrowly focussed.

Automatic Generation of Code Generators and Optimizers is concerned with the creation of translators from descriptions of the source and object forms. Since it is generally easier to describe a grammar for both the source and object forms than it is to hand craft a translator for it, interest in automatic generation of these translators is high. The

chief drawback of automatically generated translators is their inferior execution performance compared with hand-crafted translators. This is becoming less and less important [50] as well as less and less true [21].

Attributed grammars are applied in [46] to the generation of *back ends*, which for the purposes of this work includes both code generators and optimizers. The principles of data flow analysis and optimizing transformations are applied here. Other papers are devoted to the automatic generation of back ends, however they seldom address optimization at the level of design alternative (i.e. choice of data representation/algorithms). The extensibility of these approaches is unknown.

Visual Programming is concerned with the creation of programs through the manipulation of graphical representations of implementation components. It is in a sense a very high level programming activity. However, because of the great degree of abstraction, generality suffers and the expressiveness of the visual languages seems to be lagging. Although visual programming bears a superficial resemblance to this work, it differs strongly in that the visual representation in this work is a *design* and not a program. Time-Bench is arguably a visual programming environment when coupled with its code-generation facilities however, as further compilation, linking and debugging may be necessary via separate tools, it is not a visual programming environment in the strictest sense.

Automatic Programming (AP) [4], largely a pursuit of the Artificial Intelligence (AI) community, is the field to which this research most closely relates; this relationship exists because both seek to solve the same problems — although not necessarily by the same means. AP systems can be classified by the choices made in their designs. The main characteristics of AP system implementations are: *specification method*; *target language*; *problem area*; and, *method of operation*.

The specification method of an automatic programming system can be *Formal*, *Specification by Example* or *Natural Language*. Formal specifications support the precise and unambiguous expression of requirements or design. They also tend to be *complete*. This makes them amenable to automatic proof of correctness. Unfortunately, this benefit does not come without a cost. Formal specification methods also tend to be difficult to use and frequently require effort comparable to manually implementing a system. *Specification by Example* tries to overcome the unwieldiness of formal specification techniques by using a more natural approach. The key drawback of this approach is that the resulting specifications are rarely complete which can lead to inappropriate program behaviour when encountering conditions not present in the specification examples. *Natural Language* specification is even more difficult to manage as it can lead to specifications which are both incomplete and inconsistent.

Target Language describes the language of the automatically generated program source code. AP systems developed in AI tend, of course, to have AI languages (e.g. LISP) as their targets.

Problem Area is again a characteristic where the requirements for TimeBench and the features of classical AP systems diverge. For classical AP systems, the problem domain tends to be very narrow, especially for systems which permit incomplete specification (e.g. by example or through natural language).

The final characteristic of interest in placing AP systems is that of the *method of operation* chosen to implement the translation. Classical AP systems use theorem-proving, program-transformation, knowledge engineering, traditional problem solving or a combination of these.

Reference [43] contributes greatly in the area of automatic programming by introducing the idea of "cliches" which are idiomatic collections of code. This is a grouping that exists above the level of individual language statements and can include things such as algorithms and data structures. Unfortunately, decisions concerning implementation are left to the programmer in an unspecified way in [44] — which is exactly the problem we seek to solve.

2.2 MachineCharts: A Design Notation

MachineCharts [9, 10, 11] are chiefly concerned with the architectural details of a real-time system's design. Structural and temporal features of the design such as encapsulating components, parallelism and communication paths figure prominently in its visual vocabulary. Entities in a design are represented visually by either nodes or arcs of various kinds. Nodes are composed of further components which may be *internal* in which case their existence and identity is not known outside the containing node's boundaries (depicted by drawing the internal component wholly within and not touching the enclosing node's boundaries), or they may be *interface* components in which case their existence and identity is known outside the containing node (depicted by drawing the interface component within, but touching, the containing node's boundaries). Arcs serve to connect nodes together to indicate interactions between components. Detailed behaviour can be specified for a MachineChart design by either of two mechanisms: *Abstract Controller Machines* (ACMs) or *Engines*. ACMs are used to encapsulate *Finite State Machines* (FSMs) whereas engines have their behaviour specified by an agenda, the form of which is not dictated by the MachineChart design notation.

TimeBench, since it provides an animated interpretive execution of designs, dictates that the agenda be expressed in a design language which is called *MachineChart Language* (MCL). MCL is a high level, expressive but economic and loosely typed lan-

guage well suited for the expression of fine-grained design behaviour. Having a language such as MCL for expressing designs is a considerable advantage because the alternative — using an existing high-level computer language as a design language — has many disadvantages. Chief among these disadvantages is the loss of flexibility in terms of target [37]. By expressing the design in one high-level target language, automatic code generation for another high-level target language is complicated. Another significant disadvantage to using a conventional high-level language as a design language is that it may invite premature decision-making with regard to abstract data types [53]. Commitments like the representation of a particular abstract data type are the kinds of decisions which should be deferred as long as possible, and perhaps even made by an optimizing implementor. For a complete description of MCL the reader is referred to [14].

To make the discussion of MachineCharts concrete, a simple design which will serve for further explanation appears in Figure 1. This design is an instance of the familiar real-time software paradigm of the producer/consumer. In this scenario, the producer creates a stream of integers which is buffered by the buffer component before being made available to the consumer.

FIGURE 1. The Producer/Consumer MachineChart

The various design components which appear in Figure 1 include:

- *actors* (or *robots*) which appear as parallelograms (to emphasize their parallel execution semantics), represent autonomous control entities in the design. *Producer*, *Consumer* and *Buffer* are examples of actors;
- *gate buttons* (e.g. *Tx* and *Rx*) are present on the interface edges of actor *Buffer* and serve as opportunities for interaction;
- *push fingers* interconnect components. They originate on *arc tails* on the interface of an entity and terminate on a particular gate button. They represent a control interaction and an opportunity for data flow;
- *data flows* (when present) appear along push fingers and indicate the direction and number of data items that are exchanged in the course of some rendezvous. There are two dataflows in this figure, both named *val*;
- *waiting places* (e.g. *Full* and *Empty*) are denoted by a small solid black dot placed adjacent to a gate. This is used to indicate that attempts by the originator of the push finger to rendezvous at this gate may be blocked;

- *choice frames* (e.g. *ch*) are arcs which connect several of a component's gate buttons. When *activated*, they indicate which of the components gates are to be considered as eligible for a rendezvous attempt. From an implementation analogy, gates connected by the same choice frame correspond to entries in the same selective accept statement in Ada or to the file numbers in a set given to the Unix system call *select*; and,
- *arc tails* are nodes which reside on the surface of a component and indicate a place for an outward push finger to originate.

Designs are recursively decomposed yielding a tree structure where a given subtree is rooted at its enclosing node. For example, the actor *Buffer* in Figure 1 is in turn represented by a MachineChart which appears in Figure 2. This chart introduces some additional kinds of nodes:

- *push buttons* (e.g. *Tx* and *Rx*) correspond to some computation. Input to the computation can take the form of parameters passed via dataflows along an incoming push finger. Output is provided by dataflows in the reverse direction. The actual computation to be performed is specified by the button's engine;
- *engines* (e.g. *main*) are denoted by a box with diagonal lines connecting its corners to each other. The engine is the design element with the agenda and represents the source of autonomous action;
- *pull fingers* (e.g. *accept*) also interconnect components and are complementary to push fingers. Pull fingers connect engines with choice frames by terminating on a connection point. Agenda pull on a pull finger to activate the corresponding choice frame;
- *data stores* (e.g. *BufferStore*) provide a node that contains data declarations in MCL for the various agenda and data flows in scope; and,
- *connection points* (e.g. *accept*) provide a node which connects pull fingers to choice frames.

FIGURE 2. The Internal Design of the Buffer Component

The MCL code associated with the agenda of *Buffer's* engine *main* is shown in Figure 3. In this code we see the initialization required for *Buffer* and then the steady-state behaviour which is to continue pulling the *accept* pull finger. The *accept* pull finger is attached to the choice frame *ch* which, along with the guards *Full* and *Empty*, controls the acceptance of rendezvous at either of the two gates *Rx* and *Tx*.

FIGURE 3. The Agenda of the Buffer Component's Engine

```

-- Actor Buffer
engine main is
begin
  Empty := true;
  Full  := false;
  head  := 0;
  tail  := 0;
  values:= [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ];
  loop
    pull ch;
  end loop;
end;

```

The gate buttons have their own engines and agenda which relate to depositing the value into the buffer, removing it from the buffer and ensuring that the guards *Full* and *Empty* are always correct. As both of the agenda are very similar, only that of the *Tx* gate is shown and it appears in Figure 4. The agendas of the gate buttons are special because

they show the use of code within a *critical region*. Gate buttons have two regions in their code: the critical region and the non-critical region. In this particular design, the gate buttons have all their code within the critical region and none without.

FIGURE 4. The Agenda of the Tx Gate's Engine

```
-- Gate Tx
engine Tx is
begin CRITICAL_REGION
  x      := values(head);
  head   := (head + 1) mod 10;
  Full   := false;
  Empty  := (head = tail);
end CRITICAL_REGION;
begin
end;
```

The data referred to within the agendas shown here are declared within the MCL specification for the data store *BufferStore*. These declarations are shown in Figure 5 and provide the basics needed to implement a small ring buffer of integers. Note that the buffer is designed as a list of integers deferring the commitment to an implementation representation until code generation is performed.

FIGURE 5. The Declarations within the BufferStore DataStore

```
store BufferStore is
  head   : integer;
  tail   : integer;
  values : list of integer;
end;
```

2.3 A Brief Introduction to MetaTranslation

Metatranslation is a process whereby a translator is itself produced through the transformation (or translation) of an input, in this case usually a set of rules. The transformation is performed by a processor called the *metatranslator*. Metatranslation is motivated

by several needs, but the relevant one for this work is the need to overcome a lack of flexibility found in procedural translators.

Procedural translators have the desirable property that they can be highly automatic, however they tend to be inflexible. Once a commitment to using procedural translators has been made, the lost flexibility can be regained in a large part by adopting a metatranslation approach for generating the procedural translator. Recalling equation 1, where f was a fixed mapping from sources to targets, the explicit addition of some translation rules from R adds flexibility and yields the *Naive Implementation Mapping*:

$$f_N: R \times S \rightarrow T \quad (\text{Eq 2})$$

This naive mapping may not be very efficient as the computational effort in evaluating the mapping for a fixed $r \in R$ and many different $s_i \in S$ is excessive. This leads to the form that metatranslation actually takes in practice: a generator of translators. The process of metatranslation is depicted by the function m in the *Metatranslator Mapping*:

$$m: R \rightarrow (S \rightarrow T) \quad (\text{Eq 3})$$

M partially evaluates the function f yielding a mapping from designs (elements of S) to implementations (elements of T). Thus the translation rules form the specification for a code generator which itself is a translator. The translation rules are “once-removed” from the actual translation of the input to the code generator. This is the technique with which code generators for the TimeBench tool are generated.

2.4 MetaGen

The particular metatranslator used in the TimeBench case is called *MetaGen* and is written in Prolog [52]. MetaGen takes input in a Prolog-like language called *MetaGen Specification Language* (MSL), described in detail in Appendix A. The translators pro-

duced by MG are procedural and so are highly automatic. The usual drawback of inflexibility in a procedural translator is overcome by the level of indirection that using a metatranslation approach provides. MSL provides a compact and easily modified mechanism for specifying translators.

A fragment of a translation rule expressed in MetaGen Specification Language is shown in Figure 6. This rule operates on a list of robot nodes to generate code bodies for each robot and any code required for its children. Rules operate by evaluating clauses until either a clause returns true or no more clauses exist. For example, the rule shown in Figure 6 executes by first testing for an empty list; extracting a node from the (non-empty) list, obtaining that node's children, generating the code required for each of the children, freeing the list of children, generating the task body for the extracted robot node and then finally performing some tail recursion to process the rest of the robot node list. The basic syntactic elements to note are: that variable names conform to the Prolog requirement of beginning with an upper case letter; the head of a rule is separated from the body by the symbol "-->"; alternate clauses are separated from each other by the symbol "|"; sequential clauses are separated by commas; the rule is terminated with a period; clauses may be collected by parenthesizing them; and built-in predicates that correspond to the inspection layer of TimeBench are enclosed in a pair of braces.

FIGURE 6. A Translation Rule Expressed in MetaGen Specification Language

```
task_body_list(RobotNodes) -->
  {is_empty(RobotNodes)} |
  ({extract_element(RobotNodes, RobotNode, RTail),
   get_all_children(RobotNode, ChildList)},
   generate_bodies(ChildList),
   {free_list(ChildList)},
   task_body(RobotNode),
   task_body_list(RTail)).
```

2.5 TimeBench: A Framework for Software Tool Exploration

TimeBench is a software system for the design of Real-Time (software) Systems [8]. TimeBench supports a design approach known as *Architectural Design* through its use of a combination of a design notation known as Machine Charts and a program design language known as Machine Chart Language (MCL). By using TimeBench, a real-time system designer can document a design, check it for consistency and correctness, "execute" it, and even evaluate its performance. The design can then be transformed into an implementation for a particular target system. MetaGen currently permits the documented design to be transformed into a high level language source code such as Ada [25] or C.

TimeBench/MetaGen fits most closely with the formal specification technique of automatic programming because the design is checked for consistency and completeness before translation. (Note that formal specification does not require that the form of the program be specified, only that its behaviour be completely specified.) TimeBench's range of applications is that of real-time and embedded systems which is arguably a much broader problem area than those of simple queueing problems, data-processing systems, etc.

2.6 TimeBench/MetaGen Architecture

The code generation facility of TimeBench consists of a set of targeted code generators each of which is produced by the metatranslator *MetaGen* operating on a set of translation rules. A set of translation rules is required for each target that is supported. The metatranslator operates on the translation rules to produce a code generator which is expressed in the same programming language as that in which the rest of the TimeBench tool is written: Objective-C [18]. In this way, several code generators may reside in the tool concurrently. Each code generator is capable of transforming a design specification into a corresponding implementation. The code generators rely on a common layer in TimeBench which allows

inspection of the design tree representation of a particular design. The implementations in turn rely on corresponding support libraries, the existence of which is assumed by the translation rules.

The complete configuration is shown in Figure 7. The TimeBench tool itself is depicted as a large rectangle with heavy borders in the bottom centre of the figure and has its basic composition shown by sub-dividing the rectangle into the *inspector library*, *design editor*, *design interpreter*, *performance analyzer* and a series of *code generators* each of which is denoted by the letter G (for generator) and subscripted with its target. The ellipses indicate the possibility of additional generators being present. MetaGen is shown as a rectangle with heavy borders appearing above TimeBench which places it adjacent to the code generator components that it produces. The various input and output elements, *design specifications* (S_i), *translation rules* (R_j), *implementations* (T_{ij}) and *target libraries* (L_j) are depicted as rectangles with rounded corners. Arrows are used to show the general flow of information in this system: translation rules are processed by MetaGen to become code generator components of TimeBench; design specifications are processed by the code generators to become implementations; and, target libraries are provided to complete the implementations. Most of the input/output elements are shown replicated and offset slightly so as to overlap and give the impression of a stack of items. To emphasize cardinality relationships, the labels of these elements are subscripted consistently; e.g. a set of translation rules R_j relies upon a target library L_j .

FIGURE 7. Architectural Overview of TimeBench's Code Generation Facility

2.7 The TimeBench Design Tree

In this section, the representation of a design within TimeBench is described. TimeBench is an object-oriented application and consequently, designs are represented using object-oriented software components. The reader is referred to Rumbaugh et al [45], Booch [7] and Jacobson et al [26] for a background in object oriented terminology and techniques.

Designs are represented by trees of objects. There are four abstract classes of design objects found in TimeBench design trees:

- *machinechart objects* represent structure in the design and correspond visually to nodes in the notation;
- *abstract controller machine objects* pertain to the representation and specification of ACMs including their states and behaviour. These objects appear in sub-trees which are rooted at instances of the MachineChart Objects which correspond to the entire ACM;
- *machinechart language objects* are used to contain statements in MCL which specify the behaviour of engines, the contents of datastores and so on in the design. Consequently, these objects appear in sub-trees rooted by instances of MachineChart Objects which correspond to engines or data stores; and,
- *arc objects* are used to connect MachineChart Objects to each other and also to connect ACM Objects to each other.

A portion of the TimeBench design tree for the producer/consumer design of Figure 1 is shown schematically in Figure 8. The objects in the tree are represented by rectangles with rounded edges. The dashed arcs indicate the existence of elided material. Note the children of the tree node *Actor Buffer* and how they are depicted in Figures 2 and 8.

FIGURE 8. Partial Design Tree for the Producer/Consumer Design (Schematic)

2.8 Some Commercial Offerings

In this section a few representative commercial code generators are discussed.

ObjecTime [37, 53] is a tool that supports the ROOM method [49] which combines object oriented analysis and design techniques with a real-time application domain. Some experience with *ObjecTime* has been obtained through [37, 53]. This experience shows that it can be very difficult within the design tool to model existing environmental components with which the generated system must ultimately co-exist. *ObjecTime* provides an opportunity to generate code but requires that detailed implementation behaviour be specified by fragments of C++, the implementation language. This is a serious deficiency as it greatly restricts the retargetability of the generated implementation. In addition it requires

too much knowledge about the implementation *and* the generator for a programmer to write a good fragment.

SDT - SDL Design Tool [5] is a mature design product which supports the use of the System Design Language (SDL) standard for capturing the designs of reactive systems. In addition to the graphical SDL editor and message sequence chart (MSC) facility in the SDT tool, code generators are available to create implementations of the designs. Unfortunately the nature of SDL is such that the SDT-generated implementations (like those of ObjecTime) are quite restricted in their form — they end up being finite state machines (FSMs).

TeamWork from Cadre [12] is also a mature commercial product which supports design capture and implementation generation. Typically only skeleton implementations are available unless the designer wishes to produce a number of fragments for each component, each corresponding to a particular source location in the skeleton (e.g. before the declarations, after the declarations, before the executable statements, within the exception handling region etc.). Again, the fragments are expressed in the target language and consequently suffer the same drawbacks described earlier for ObjecTime.

A Step Towards Usable Automatic Code Generation

This chapter contains a detailed examination of the problem addressed by this work, followed by a thorough analysis of the requirements a solution must satisfy, a discussion of the proposed solution's design and concludes with a description of the solution's implementation.

3.1 Defining the Problem

The problem this work seeks to solve is to make automatic code generation a viable alternative to manual implementation for software systems. The technology for transforming designs to implementations already exists, but is not in widespread use. This is, in the author's opinion, not due to ignorance or lack of interest, but rather because the existing approaches are impractical. Therefore, the main aim of this work is to address how current approaches fail to be practical or usable. To do so, we need to examine the implementation process to provide some context for the decisions made in the design and implementation of a solution. Several questions must be considered:

- "What differentiates a good implementation from a bad one?";
- "Given a design, what additional information is required to implement the design?";
- "How can this additional information be best expressed?"; and,
- "How can an automatic code generator make use of this information?".

Before proceeding any further, it is helpful to define what is meant by *design* and *implementation*:

Design: A design is an abstract entity which is a strategy or plan for creating a working software entity (the *implementation*) which satisfies some goals (the *requirements*).

Implementation: a working software entity which conforms to a *design* and therefore ultimately satisfies the original *requirements* upon which the design is based.

It should be noted that it is natural for there to be many possible implementations that satisfy a specific design [47] this is because designs deal mostly with the overall structure or architecture of a software program and ideally omit any details not considered relevant to the satisfaction of the requirements. These omissions represent degrees of freedom for the implementation where additional information is required to produce a concrete instance. Some of these areas of freedom are used up by constraints placed on the implementation through the choice of implementation technology, while others are consumed by *non-functional* requirements as described in [47]. The determination of relevancy with respect to the content of a design can be made by examining the requirements or, by examining the degree or level of detail involved. This last decision criterion (the level of detail) is mentioned, because it has been suggested that the boundary between design and implementation is a personal one, depending on the capability of the designer [46]. What this work seeks to do is to raise this boundary for all practitioners by codifying the implementation transformation without sacrificing the flexibility and implementation efficiency that manual implementation provides.

As an implementation, the design is *under-determined*. In this sense, the design is a projection of, or serves as an equivalence class for a set of implementations. It is the particular implementation chosen from the set of possible ones that determines the overall quality and satisfaction that a software system provides.

How to Transform a Design to an Implementation. This issue is not addressed in detail by this work except to note that evidence suggests that a procedural transformation approach is worthwhile [43] and that metatranslation can buy back the needed flexibility [47, 48]. Approaches such as providing a design interpreter in the target environment are eschewed completely.

Differentiating between Good and Bad Implementations. Good implementations are distinguished from bad ones through an aggregate measure of how well the implementation meets its requirements (the design) and how well it measures along scales corresponding to the non-functional requirements or constraints. Therefore, assuming all implementations satisfy their explicit requirements, the goodness or badness of an implementation is almost entirely determined by the choices made during its implementation. These choices are possible (indeed, necessary) only because there are areas of the implementation where no explicit requirements exist. Factors affecting these choices are described in the next section.

The Additional Information Required to Implement a Design. As mentioned above, designs by themselves are inadequate for automatic translation into implementations. Either the non-functional requirements or the decisions regarding them need to be captured to complete the implementation process. In [24], the kinds of information that affect translation (implementation) are divided into three classes: *Translation strategy* information which specifies which of several alternative strategies is to be utilized where choice exists; *External Interface* information for characterizing a boundary between the design and its environment; and *Configuration Information* which determines which vari-

ant of each component is appropriate for building a particular version of the system. These classes of information are summarized in Table 2.

TABLE 2. Classes of Information Required to Implement a Design

Information Class	Description
Translation Strategy	Where more than one alternative exists for translating a given design element, some non-functional requirement may be used to influence the selection.
External Interface	When generating systems that are "connected" to their environments, it is advantageous to have an explicit means to describe the boundary.
Configuration	As a matter of practical software engineering, the control of revisions is a necessary capability.

Examples of information that are considered to be factors affecting translation strategy are shown in table Table 3. Although any of these items could be specified explicitly in the system's requirements, they are generally examples of the non-functional requirements which need to be considered during implementation.

TABLE 3. Factors Affecting Translation Strategy

Free Items	Description
Data Type Efficiency	Efficient data type implementations may or may not be an explicit part of the user requirements. Controlling space/time efficiency may however confer competitive advantage on developers. Currently many approaches rely on hand-coded implementations, sometimes several different implementations of the same type. Implementation limits like: maximum integer size
Hardware Target	The choice of hardware target brings with it a number of constraints which can influence an implementor. (e.g. how many physical processors are present? What should be the task to processor mapping?)
Corporate Standards	Existing code from other development methods may need to be accommodated.
Error Handling	What level of debugging code should be generated?

TABLE 3. Factors Affecting Translation Strategy (Continued)

Free Items	Description
Performance	Are there critical performance requirements which arise from earlier partitioning of the design?
Code-size	How large may the program be? What is the trade-off between code and data size?
Reliability/Security	What degree of reliability is warranted? What need for security exists?

As for the *External Interface* class of additional information, it is rarely true in practical systems that the software is isolated¹. At a minimum, an application's environment usually contains various hardware devices which may not warrant modelling in the design. Device drivers, interrupt handlers or legacy code may all be present in the environment too. Other examples of "environmental" factors include code produced by hand, standard system libraries and highly time-critical/device-dependent functions.

Finally, configuration information is highly desirable in a large system. Although it doesn't affect translation directly, a practical automatic code generator should address this issue by ensuring that it can inter-operate with some existing (and usually mature) configuration management system.

Expressing the Additional Information. This issue concerns how the implementor can express the additional information or capture the decisions made that are required for an automatic code generator to supply a good implementation. The technique used here should be simple, preserve the re-targetability of the code generator and lend itself to ease of maintenance (for the designer/implementor). The term *Annotation* is used through-

1. If 70% of software development activity is maintenance, tools like this will be slow to penetrate unless they accommodate interoperating with legacy systems.

out the remainder of this thesis to refer to the expression of an additional item of information required for producing a good implementation.

Using the Additional Information in an Automatic Code Generator. This matter is a practical one for the implementor of the code generator and not really of concern to its user. The determination of how to make use of any additional information is influenced heavily by how the annotations are expressed and the current architecture of TimeBench. A general mechanism of providing *inspectors* for the code generator to use will suffice.

3.2 Requirements for a Practical Automatic Code Generator

In this section, the requirements for a hypothetical practical automatic code generator are presented. Sources for these requirements are named and some are discussed in detail. Some generalizations about the requirements are presented before discussing each of the requirements in detail.

Broadly speaking, a practical automatic code generator needs to produce quality software, practically. In terms of specific requirements however, this is not very helpful. Fortunately, there are several appropriate sources for requirements pertaining to producing quality software and producing it practically: commercial requirements statements; suggestions found in published research; as well as complaints and deficiencies observed in existing approaches [48]. In addition, some requirements can be derived from taking a broad perspective and considering the deep analogy between modern automatic code generation systems and the modern high level language compiler.

3.2.1 Principal Objections to Automatic Code Generation

In this section the broad categories of the principal objections designers and implementors have with automatic code generation are listed and examined. These objections frequently need to be addressed by a number of different requirements and solutions.

There is a deep analogy between the code generator described in this work and a high level language compiler. In fact, compilers for high level languages were initially referred to as automatic programming systems [34]. Initially met with resistance, compilers for high level languages have become ubiquitous. This analogy is useful to keep in mind when considering the sources of issues, the mechanisms for their resolution and the evaluation of the solutions. The original objections raised against high level language compilers serve, by analogy, as a first source of requirements for an automatic code generator:

- efficiency of the generated code;
- correctness of the generated code; and,
- flexibility of the generating tool.

In general, most current code generators address the "hard" requirements such as correctness; practicality is distinguished by flexibility and efficiency in these cases.

Efficiency refers primarily to the efficiency of the generated code [16, 21] either in terms of code size, data size or execution speed. The fear is that automatically generated code will necessarily be too slow or require too much memory due to the generality of the generation mechanism. Although the prevalence of this flaw may be over-rated [31], it's a valid concern and one that can be addressed in a number of ways. It is recognized that it is usually only a small part of a program that is time critical. Techniques that permit intervention in these areas or provide some other form of assistance in this area are becoming

more common [21] and are certainly practical. This concern is an historical echo of those voiced over high level language compilers at the time of their introduction [34].

"Taken to the extreme, these arguments can be used "to motivate the usage of assembly languages rather than high level languages." [31].

A secondary concern is the efficiency of the generating tool itself. Ironically, this is only a concern because the speed of generation is great enough to permit alternate implementations to be created and evaluated — something that would be unlikely to be done in an environment where implementation proceeded manually. Also it is noted that efficiency of implementations is often in tension with their understandability, portability, modifiability and even reliability.

Correctness of the generated code is an issue because of debugging considerations. When the generated system is subject to verification and validation, it is important to have a high degree of confidence that deviations are due to design or requirements errors and not because of flaws in the generation process. [54]

Flexibility can mean many things when discussing automatic code generation. Many of the requirements discussed here are specific examples of flexibility requirements or contribute to flexibility in various ways. Flexibility requirements can arise from things like the desirable ability to postpone implementation decisions such as choice of programming language [31]. Some objections from [48] which relate to flexibility are:

- canned library implementations are not always appropriate for every application;
- richly typed designs are inadequately accommodated;
- directing the eventual compilation and linking of the generated code is difficult; and,
- literals and magic numbers may need to be translated for a particular target.

Software Quality

Achieving quality software requires attention throughout the life cycle and across all the development products and processes [20, 23, 42, 55]. Although no implementation technique, regardless of how perfect, can resurrect lost opportunity to achieve a quality outcome, implementation quality itself is a factor in overall software quality. Definitions of Software Quality and how it is achieved with respect to code generation appear in the next section.

3.2.2 Aspects of Software Quality

The conventional ideas concerning software quality can be used to form an opinion about the suitability of a particular code generation strategy such as the one described in this work. However, there are some important differences to keep in mind. Again, the analogy with high-level language compilers is useful in keeping perspective when considering software quality attributes that relate directly to the textual expression of a program.

Software quality is something that spans the entire life cycle of software development [23] and all of the various development products such as documents, reviews, prototypes and so on. In this section, the focus is on software quality as it relates to implementation, the interaction of implementation with the phases before and after it (namely, design and testing respectively) and with the work products (1) generated code and (2) design document(s). Before proceeding, it is prudent to define what is meant by software quality. The definition adopted in this work is that of Glass [23].

According to Glass, software quality is a vector with seven components. These components measure various aspects or attributes of the quality of software. The aspects chosen by Glass appear in Table 4 with a brief description.

TABLE 4. Descriptions of the Characteristics of Quality Software

Quality Aspect	Description
Portability	The ability of the software to be transferred easily from one computer to another for the purposes of execution
Reliability	The ability of the software to satisfy its requirements without error
Efficiency	The ability of the software to perform with minimum use of computer resources such as internal memory, external memory and machine cycles
Human Engineering	The ability of the software to be easily understood and used by human users.
Testability	The ability of the software to be easily verified by execution.
Understandability	The ability of the software to be read by a software maintainer.
Modifiability	The ability of the software to be revised by a software maintainer.

Portability and Reliability warrant special comment in this section:

Portability. The quality characteristic embodied in portability is somewhat special in that it may or may not be important depending on the requirements for the software being produced. If it is important to a particular product, one means of obtaining it more readily is to rely on standardization or an insulating layer of software that provides an abstract environment in which the software can run. Automatic code generation with multiple platform targets is analogous to this abstract insulating environment.

Reliability. Perhaps the most universal of quality attributes, reliability requires a rigorous approach to requirements capture and analysis. Once performed, the only remaining opportunity to achieve reliable software is to minimize errors during subsequent devel-

opment. Having a (relatively) small set of well-used translation rules can contribute greatly to minimizing these errors [31, 54].

Also, two characteristics of software not covered by Glass's vector but which affect software quality are:

Modularity. Modularity has several aspects which relate to desirability of a particular implementation. One factor is readability. Although modularity leads to code that is initially difficult to read [23], once a modules' functions are understood, readability is enhanced. Another factor affected by modularity is the ability to manage code with regard to configuration management and compilation dependencies. A small localized change should "not lead to that[sic] the complete system has to be regenerated or re-compiled." [31]

Consistency Between Design and Implementation. A practical tool must address as many of an implementor's needs as possible so that they may resist the temptation to *polish* the generated code. Polishing leads to implementations which are no longer consistent with the design. If the design should change, a newly generated implementation would then require the merging of the manual changes made to the previous implementation. Implementors polish the generated code for one of two reasons: (1) the generated code fails to satisfy some efficiency concern, and (2) the generated implementation lacks some form of expressivity. An example of (2) is the inability of a generator to take advantage of some native target feature such as exception handling in Ada [25].

3.2.3 Detailed Requirements Analysis

In this section each of the requirements is identified and discussed in detail. The requirements assembled for this work appear summarized in Table 5. Each requirement is given a unique identifier of the form *R-n* (where *n* is a number) in order to permit efficient reference to it. In addition, an indication is provided in the table for whether the requirement relates to efficiency, correctness, and/or flexibility concerns.

TABLE 5. Requirements for a Practical Automatic Code Generator

Id	Efficiency	Correctness	Flexibility	Requirement Summary
R-1		•		Generated code should be correct
R-2	•			Generated code should be efficient
R-3			•	Generated code should be readable
R-4	•	•	•	Generated code should easily interface with existing software
R-5	•		•	Code generator should provide control over static software structure
R-6	•		•	Generated code should be able to use native target features
R-7	•	•	•	Code generator should support software re-use.
R-8	•		•	Code generator should accommodate compiler/linker directives
R-9	•		•	Code generator should be re-targetable
R-10	•		•	Code generator should support partial/incremental generation
R-11	•	•	•	Code generator should be implementation-language neutral

TABLE 5. Requirements for a Practical Automatic Code Generator (Continued)

Id	Efficiency	Correctness	Flexibility	Requirement Summary
R-12			•	Special implementation coding requirements
R-13	•		•	User guidance should be optional not required
R-14	•	•	•	Expressive power
R-15	•	•	•	Should integrate with a design tool that supports High Level Design
R-16	•	•	•	Generated code should be debuggable.
R-17	•		•	Regeneration should be preferable to manual changes.

R-1: Correctness. This requirement means that the automatic code generator must generate implementations that behave as specified in the design. Relaxing this requirement would make the problem trivial to solve and so it is meaningless to consider systems that do not attempt to implement designs "correctly"; i.e., preserving the semantic content of a design and honouring that design as an implementation requirement. This requirement is not meant to address *formal* correctness (see "Future Research" on page 109 for a discussion of this).

R-2: Efficiency. As discussed above, efficiency refers primarily to the space/time efficiency of the implementation that is generated. A practical automatic code generator must consider producing efficient implementations or a number of undesirable things may occur. Designers may anticipate the implementation and design accordingly [31], or implementors may change the implementation manually compromising the leverage of

working at a higher conceptual level (design) possibly wasting effort, risking correctness and sacrificing design and implementation consistency (see requirement R-17).

When considering efficiencies of the development process itself however, other requirements come into play such as those which support reusability (R-7) and those which support partial/incremental implementation (R-10).

R-3: Readability. Readability refers to the attribute of the generated program source text that facilitates understanding of it. Aside from typographic considerations, if the expression of the program is idiomatic and follows a consistent structure, it is readable [39]. Readability is only an issue if someone wishes to read the code. Although it would be ideal if nobody needed to look at the generated code (this would certainly simplify translation), this is not a realistic possibility [31]. For a number of reasons, readable code will remain a requirement in the near future. Existing debugging tools operate at the implementation level. Even if a design tool has the capability to verify and validate a design, the implementation still requires the same treatment. Until tools support these activities at the level of the design, tracing bugs back to the design is facilitated by having the generated code be readable and by having its structure map easily back to that of the design [24, 28]. This requirement, since it relates to verifying the correctness of the implementation, takes precedence over any concerns about efficiency. An additional reason for requiring readability is for those users who generate code as part of a larger system where safety and or correctness concerns cannot be satisfied solely through confidence in the generator. These users may still require source code to be inspected (even if their compiler is not provably correct) [37].

Readability is in tension with other desirable qualities in software engineering. Efficiency has already been mentioned but for occasional readers of code, another may be

modularity. Modularity leads to code that is initially difficult to read [23]. However, once a modules' functions are understood, readability is enhanced. Thus for users who need to read the generated code rarely (and are therefore repeating the "initial" case), modularity may compromise readability.

R-4: Ease of Interfacing with Existing Software. Rarely are the software systems that could benefit most from automatic code generation new ones [37, 53]. Rather, most software work is performed maintaining existing systems. Even without legacy code considerations, interfacing with existing software is highly desirable because it facilitates re-use [38, 57] through a software component paradigm. Thus, a practical automatic code generator should accommodate interfacing the generated implementation to existing software at both the source and object levels.

R-5: Control of Static Software Structure. By requiring that a practical automatic code generator provide the implementor with control over the static structure of the software, configuration control, partial/incremental generation of code and reading the generated code are all made easier [19, 51].

R-6: Use of Native Target Features. If a target environment possesses a somewhat unique and valuable technology that is not appropriate to include in a design, it should still be available to users of the code generator. An example is the exception-handling capability of Ada. A practical automatic code generator will provide some means for implementors to take advantage of such features without requiring them to resort to polishing the generated code by hand.

R-7: Explicit Support for Reuse. Given the importance of reuse in improving software development productivity and quality [23, 41], a realistic automatic code generation facility must explicitly acknowledge and support reuse. Re-use of existing code

whether in libraries or supplied in the design document should be possible through the code generator.

R-8: Compiler/Linker Directives. Compilers, linkers and other *down-stream* tools may require certain option settings to achieve various implementation goals such as optimization, preparation for symbolic debugging or producing a sharable object code. These directives are properly part of the implementation and consequently should be accommodated by the code generator.

R-9: Retargetability. In order to facilitate re-use of a design in various target environments and to achieve the desirable quality of portability in the generated software, a practical automatic code generator should be retargetable. This is even more important once partial and incremental generation is supported as the likelihood that re-usable components will be developed (and their value) increases dramatically.

R-10: Partial/Incremental Generation. Partial generation (selected components of the system are generated) [28] is a requirement because of several reasons. The most important is perhaps that changes as a result of correcting defects in the design should not cascade into large re-generations and re-compiles as this could be impractical for systems of any significant size. Another reason is that strict adherence to a water-fall model of development is not always possible (or desirable) so in order to support some unit testing in parallel with design activity for instance, partial generation is required. A final, and no less significant, reason is that the design may contain elements that are present only to model existing components and therefore don't require generation. These modelled com-

ponents may represent existing software in the environment of the design and may be present only to permit simulation of the design [53].

Partial generation also has a beneficial interaction with configuration management by isolating and localizing changes. "For large systems, separate compilation is a practical necessity." [24]. A small localized change should "not lead to that[sic] the complete system has to be regenerated or re-compiled." [31]

R-11: Target Neutrality. Again, to facilitate re-use and portability, a practical automatic code generator will attempt to be as independent of the target¹ as possible. This is evidenced by the generator being re-targetable, by accommodating implementation hints (annotations) in a target-independent manner. As much as possible, target language independence should be pursued too [28].

R-12: Administrability. If the code generated by the generator is to fit in with existing software in the runtime environment, it must also do so in the development process itself. Administrative requirements such as coding standards should be accommodated where possible.

R-13: Autonomy. The code generator should operate automatically without *requiring* intervention from the implementor. This is because of two reasons: interaction is difficult to document, configure and track; and, if the generator can meet all other requirements without needing interactive capabilities, then it need not complicate matters.

R-14: Expressive Power. Expressive power is a property of the interface between the user (through the design representation) and the automatic code generator. It relates to

1. For the purposes of this work, the term *target* refers to the hardware and software (e.g. operating system, supported languages, windowing systems, etc.) environment in which the generated implementation is expected to execute.

the ability to specify things such as performance and time constraints (e.g. minimum/maximum), concurrency, communication, synchronization and data format [28].

R-15: Tool Integration. A practical automatic code generator should integrate smoothly into an existing development tool set [37]. Ideally, it would fit into a suite that could assist with high-level design; decomposition and detailed design; verify behaviour of the design, and create implementations that can be run on the target hardware [37]. The goal is to create living designs [53].

R-16: Debuggability. This requirement relates to the ability of the generator to produce code that is no more difficult to debug than that produced by manual implementation. Fixing of bugs and of compiling errors must be as easy as possible to avoid hand polishing and the design/implementation consistency issues described immediately below.

R-17: Consistency of Design and Implementation. Keeping a design in sync with an implementation is a challenge that requires discipline in manual implementation. When automatic code generation is used, it is simple in principle. The problem arises when the code generator is not practical in some area and manual intervention is performed. A practical automatic code generator will minimize the need for such intervention and ideally eliminate it entirely. The code generator may address this requirement by other features which promote the consistency.

3.3 Design of a Practical Automatic Code Generator

Many of the requirements identified in the previous section are satisfied in part by the selection of TimeBench/MetaGen as a base for this work. However, some requirements which are particularly important for creating a practical code generator remain unsatisfied. This research focuses on designing and implementing capabilities for TimeBench, MetaGen and its code generators to more fully meet these requirements. Most of

these failings can be traced to inadequate support for specifying and using the information identified earlier (see "Defining the Problem" on page 26) as necessary for mapping a design to an implementation. This is the situation as shown in Figure 9 which depicts an inadequate transformation process. In this inadequate process, either the additional information that is required to produce a good implementation is absent, or worse, it is encoded in the translator which compromises the translator's generality and obscures the implementation decisions.

FIGURE 9. Inadequate Transformation Process

The preferred case is shown in Figure 10 where the transformation process is augmented and the additional implementation knowledge, which is shown in the shaded area, is made explicit. The three components in the shaded area represent candidates for expression through some means such as user annotations. In any case, they represent Implementation Knowledge and Decisions and so should be explicitly stated but separate from the design proper.

FIGURE 10. Augmented Transformation Process

In the design and implementation of features that address the requirements identified above, some guiding principles are followed:

- User intervention (through annotation) is optional — the code generator should not require the user to annotate a design in order to produce an implementation.
- Implementation is still a distinct phase and so the tool should separate it from design.

While forming a design that will address the requirements of the previous section, it is useful to keep the following techniques for achieving software quality in mind. These techniques help influence the exact choice of or form taken by a proposed feature for the code generator.

3.3.1 Techniques for Achieving Software Quality.

According to Glass, technique number 0 is Re-use. The use of existing design and implementation components affects quality beneficially in many ways. Opportunities to facilitate reuse for the user are noted throughout this section.

Techniques for Portability: when required, portability is vital. Opportunities to exploit standard programming languages, operating system interfaces, to isolate hardware dependencies, language variations and timing dependencies are sought during the design of the practical automatic code generator.

Techniques for Achieving Efficiency: the first observation that can be made about automatic code generation from designs is that it avoids the need for a *design interpreter* that is hosted on the target system such as the one proposed in [28]. The second observation is that efficiency should be a design goal rather than an implementation afterthought: "the place to start achieving efficiency in software is during design. If the effort is postponed to the coding phase, it may well be too late to achieve the desired efficiency." [23]. In [31] their "experience is that the efficiency [of automatically generated code] will be better than generally believed. Furthermore, it is usually only small parts of a program that are time critical.". Taken to the extreme, these arguments [against automatic code generation] can be used "to motivate the usage of assembly languages rather than high level languages." [31]. Returning to [23]: "It is important first to choose the optimum form of such an algorithm, and second to code it as tightly as possible". Continuing with: "Only after all design efforts for optimization have been utilized should coding solutions be employed.". And finally observing that efficiency is in tension with understandability: "Software quality is dramatically improved by the use of the highest level language possible. Techniques for coding more efficiently often involve moving in the opposite direction."

Unfortunately, performance cannot simply be added in at the end of a development project. What is needed is a straightforward understanding of the implementation process so that the design may proceed with an eye toward efficiency; for example, most high-level language programmers are aware of space/time trade-offs involved in the implementation of a sparse case statement and a series of IF/ELSEIF statements without being directly aware of the actual form of their implementation. Nevertheless, the code generator must attempt to be sufficiently flexible in satisfying efficiency concerns that it doesn't lead to programming rules which can be "a pain" [53].

Turning over the implementation of software to an automatic code generator can focus an implementor's energies on efficiency during design as well as, given a suitable generator, automate the selection of efficient algorithms and data structures.

As for efficiency in development, this can be facilitated in two ways: (1) by supporting a small granularity of source code generation, compilation dependencies need not cause small design changes to cascade into overwhelming re-compiles; and (2) by reducing the amount or changing the kind of testing required. Although it seems somewhat overly-optimistic to state that "when the synthesis of application software is performed automatically by means of appropriate tools, the conformance of the implementation can be guaranteed without extensive testing" [31], it does appear that there are some savings to be had here.

Techniques to Achieve Understandability: there is a standard trade-off in natural language between the effort involved in expressing one's self and in understanding another [40]. This same trade-off exists in the expression of software programs. Mechanisms such as naming conventions; code simplicity; structured techniques and modularity; and, the clarification of logic intent through indentation and white space are all employed in a con-

sistent manner in good implementations to promote understandability. Since an automatic code generator can make re-use of its abilities, it can be made to write code in a highly understandable way and thereby overcome this standard trade-off by assuming more of the responsibility for effectively communicating the implementation while amortizing the effort of creating the generator over many applications.

There is another aspect to understandability worth recognizing here and that is that the mapping between implementation and design should be somewhat obvious to the implementor. In this way, the implementor can readily determine the context in which the generated code fits. Ideally nobody should need to look at generated code (this simplifies translation), however this is not yet realistic. For one thing, debugging tools typically operate at the implementation level. This necessitates the ability to trace bugs back to the design which is where the changes should be made. Until debugging is provided at the design level, some awareness of the translation rules will be necessary.

If an automatic code generator can promote generation of an implementation without the need for hand polishing, then understandability of the implementation will be enhanced because the documentation (the design) will be consistent with code.

As noted above in the section on efficiency, understandability is in tension with efficiency. Drawing on the automatic code generator as compiler analogy, many debuggers will not permit the debugging of optimized code because the control flow and local data lifetimes are confusing. Since the object code of the automatic code generator is in fact high-level language source code, we can see that efforts to make it understandable can have a negative impact on efficiency.

Testability: derives largely from having well-formed measurable and decidable requirements.

Modifiability: efficient maintenance has two key contributors: understandability and modifiability. Once a software program is understood, it can be modified. Single-point control, high-order languages, standards and structured coding techniques all contribute to understandability. Named constants, data hiding, low-coupling and high-cohesion facilitate correct and easy modification. In the realm of automatic code generation, however we wish to encourage the implementor to make changes in the annotations or design model rather than the source code. To do this effectively in the presence of existing software (in both source and object forms) and partial/incremental generation, the code generator must possess the capability of generating the entire system including all of the hand-coded additions that may be necessary or present already.

Another technique that promotes modifiability and that can be accommodated by an automatic code generator is to design anticipating change and include *stubs* in the system that represent future functionality that can reasonably be foreseen but is not currently required.

Because understandability is a precursor to modifiability, the conflict between efficiency and understandability applies to efficiency and modifiability also.

Reliability: Most universal of these software quality attributes, reliability focuses on minimizing errors during development and, to a lesser extent, during the execution of the software. By far the biggest influence on software reliability is at the front end of the development process as most software errors are introduced during requirements analysis and design: "by the time the coding phase is reached, the most important opportunities for eliminating devastating errors have passed." [23]. Therefore, reliability during development is fostered by rigorous approaches to determining and capturing the requirements and by having a correct design approach. Reliability during implementing the software is

achieved by taking advantage of error-inhibiting implementation techniques such as structured programming in a high-level language.

Although some [54] anticipate high reliability software from automatic code generators, this is unlikely for two reasons. The first reason is noted above, that reliability is a strong function of the earlier development phases. The second reason is that a realistic scenario for the use of a practical automatic code generator involves integrating the generated implementation with existing software thereby introducing a weak link and compromising any reliability claims that the generator can make.

3.3.2 Design of Features

In this section, the approach for designing a new feature for a practical automatic code generator based on TimeBench/MetaGen is described. The issues involved are identified and described.

User Annotations: in order to help overcome the shortcomings of automatic code generation, a facility has been added to TimeBench to allow a designer to provide hints to the code generator. These hints are known as *User Annotations*. The introduction of a particular form of user annotation requires several decisions. The first is to decide if a proposed feature is amenable to being treated by the annotation mechanism. Usually this decision is made by recognizing that some information is required from the user of the code generator to properly implement the feature. The next decision is regarding what form this additional user-supplied information should take (i.e. the user interface to this information). Typically, the point in the internal representation of a design where this additional information should be stored needs to be determined too. Modifications to the translation rules are needed to take advantage of the additional information and the layer in TimeBench which allows code generators to inspect and manipulate design models needs

to be augmented. Finally, the target libraries may also need extensions to permit full utilization of the new technique.

With this in mind, the process of automatic code generation can now be extended in a general way to accommodate the additional information required to produce good implementations. Recalling equation 2, the *Naive Implementation Mapping*, and extending it by adding in the *annotations* (A), yields the *Annotated Implementation Mapping*:

$$f_A: R \times S \times A \rightarrow T \quad (\text{EQ 4})$$

This mapping suffers the same kind of computational disadvantages that equation 2 did however and consequently, the remedy is the same — improve the mapping by introducing metatranslation. The result is the *Ideal Implementation Mapping*:

$$m_A: R \rightarrow (S \times A \rightarrow T) \quad (\text{EQ 5})$$

Using this new mapping as the fundamental approach to improving the code generation facilities of TimeBench/MetaGen suggests an extended architecture for the tool which is depicted in Figure 11. The additional elements in this architecture are shaded to differentiate them from the original elements. The original architecture was described in “TimeBench/MetaGen Architecture” on page 20 so only the changes and additions are described here.

Starting from the left of the diagram, the *annotations* (A_i) are depicted by the rectangles with rounded corners just below the design specifications (S_i). Although the annotations are independent from the design specification in principle, they are highly correlated in practice and therefore share the same subscript as the design specifications. At the top of the diagram one set of translation rules has been shaded to indicate the source for a new code generator which serves as the basis for implementing and evaluating the

contributions of this work. At the top right of the diagram are the examples of *environmental code* ($E_{i,j}$). This code represents existing software which must be accommodated or special code that is to be included in the generated source. It is subscripted to indicate that it is related to both the design and the target. The environmental code combines with the target libraries and implementations to form the entire generated system.

The final addition of note, as alluded to above, is the creation of a code generator targeted for the C programming language and the UNIX operating system. This addition is necessary for two main reasons: to provide a mechanism to act on the information provided by the annotations when generating code; and, to implement those features that address practical automatic code generator requirements but are not possible, or desirable, to implement through annotations. The bulk of the work in this project was done on this component. Although some modifications were necessary to the inspector layer, these were straight-forward and are not highlighted here.

FIGURE 11. Architecture of Improved TimeBench Code Generation Facility

There are two issues when designing a mechanism to annotate a design: *reference* and *extent*, i.e. to what does the annotation refer, and how can this be indicated; *repetition* and *maintenance*, the same implementation information should be useable in several designs. The annotations this work proposes address these issues by implementing annotations in a couple of different ways as the particular situation warrants. The two basic ways that annotations are implemented in this work are: as a text string that is treated as an attribute of a MachineChart object; and, as a *pragma*, or directive that appears within the

MCL code of an agenda. In the former case, the reference and extent of the annotation is determined by the structure inherent in the MachineChart model itself. That is, it is clear from the hierarchical MachineChart model *what* and *how much* is being referenced. In the latter case, the scope of influence is local, however annotations may use *indirection* to point to a single instance of an implementation component. For example, several annotations may reference the same foreign source code for inclusion in the generated implementation. In this way, re-use within designs is achieved in that components may be referenced in more than one annotation and therefore do not have to be duplicated; and, re-use across designs is achieved by adding annotations to other designs that point to the original implementation component.

The general technique of annotating the design through associating additional text with the relevant MachineChart entities avoids cluttering the visual vocabulary of MachineCharts and hides the implementation hints under normal circumstances. The pragmas appear as comments in MachineChart Language and are therefore transparent to the design verification and simulation components of TimeBench. Further advantages include avoiding the necessity of changing the definition of MCL, the ability to view all parts simultaneously, and ease of manipulation.

3.3.3 New Automatic Code Generator Features

In this section, the new features that have been designed and implemented for code generation in TimeBench/MetaGen are presented. Features are numbered *F-n* (where *n* is an integer) to permit ready reference to them. A detailed description of each feature from the

perspective of a user is provided. Additionally, a discussion of what requirements are addressed by which features and why is provided. A summary tracing requirements to features appears in Table 6.

TABLE 6. Features added to TimeBench/MetaGen

Identifier	Requirements Addressed	Feature Description
F-1	R-2, R-13, R-14, R-15, R-17	Optimal Abstract Data Type (ADT) Implementation Selection
F-2	R-2, R-8, R-13, R-16	Support for specifying directives for downstream tools
F-3	R-3, R-4, R-5, R-7, R-10, R-12, R-13	User-guided partitioning of design into source modules
F-4	R-3, R-4, R-6, R-7, R-9, R-11, R-12, R-13, R-16, R-17	Inclusion of literal target language source code/
F-5	R-14	Task Priorities
F-6	R-7, R-15	Component Stubbing

The new features implemented by this work include [48]:

- automatic or user-guided selection among functionally equivalent Abstract Data Type (ADT) implementations;
- additional flexibility in specifying directives for other *downstream* development tools such as the high-level language compiler;
- user-guided partitioning of the generated implementation into separate source files to facilitate configuration management and partial/incremental development;
- ability to incorporate additional software expressed in the target language into the generated implementation;

- ability to incorporate additional software expressed in the target object code into the generated implementation; and,
- ability to use native capabilities of the target environment.

A matrix that traces which features address which requirements appears in Table 7. This is a restatement of Table 6 but organized to permit ready identification of unaddressed requirements. The right-most column of the table records whether one or more features address a particular requirement (i.e. the requirement is *met*). Note that none of the new features addresses requirement R-1 (Correctness) directly although it is considered to be met given the earlier discussion of it.

TABLE 7. Requirements Traceability Matrix

Requirement	Feature						Met
	F-1	F-2	F-3	F-4	F-5	F-6	
R-1							•
R-2	•	•			•		•
R-3			•	•			•
R-4			•	•			•
R-5			•				•
R-6				•			•
R-7			•	•		•	•
R-8		•					•
R-9				•			•
R-10			•				•
R-11				•			•
R-12			•	•			•
R-13	•	•	•	•			•
R-14	•				•		•
R-15	•					•	•
R-16		•		•			•
R-17	•			•			•

3.4 Implementation of New Features

This section describes the implementation of the new features developed in this work. The implementation details of each feature as well as its capability are discussed broadly. A summary of the activities involved in implementing a feature based on a user annotation is presented and one particular user annotation is singled out with its implementation is examined in detail.

3.4.1 Details of Additional Features

F-1: Abstract Data Type Implementation Selection: A frequent complaint heard about high level language compilers in their early days was that the object code that was generated was not very efficient in terms of both time and space. The same complaint can be levelled at code generators but at a correspondingly higher level of granularity. It is generally accepted that the first and most fruitful approach to optimizing a high level language source program is for the user to choose an appropriate algorithm. The selection of algorithms is intimately tied to the data structures chosen for the problem being solved. The selection of the data structure representation and the attendant algorithms is uniquely an implementation issue. The matter should not concern the designer. A designer may specify the use of an abstract data type merely by indicating the kind of type (list, set, tree, etc.), whereas the implementor must choose an implementation for this abstract data type. It is in this area that we can offer perhaps the single greatest improvement in usability of the automatic code generator [17].

With an appropriate library of implementations for each abstract data type available in MCL and a corresponding *cost function*, the transformed design can have the appropriate ADT implementations selected and incorporated in the overall system implementation without requiring guidance or intervention from the user. The analysis that is

required to perform this selection involves recording the (static) number and type of operations which are performed on each instance of an abstract data type, computing the cost function and selecting an implementation. For example, the MCL code fragment in Figure 12 shows the declaration of the sets *A* and *B*. One or both of these sets are tested for emptiness, used in a union and have elements added to them. Based on the presence and relative (static) frequency of certain operations (e.g. membership testing, union, addition), a cost function can be evaluated. The cost function maps a vector of operation occurrences to an indication of which implementation to choose. Other work stops at the suggestion that "While translator tools should always provide a default translation strategy for each SDL construct, it should also be possible explicitly to state which translation strategy is to be adopted in a particular instance." [24]. This work improves on this by providing an intelligent choice among equivalent ADT implementations.

FIGURE 12. Example ADT Usage in MCL

```
store ADT_definition is
  A,B    : set of integer;
end;

-- Actor Manipulate_ADT
engine main is
begin
  A := (); B := ();
  if #A = 0 then
    A := B + (2, 3, 4);
    B := { 5, 6, 7 }
    A := A + B;
  endif;
end;
```

F-2: Specifying Directives for Down-stream Tools: occasionally there is a need to supply parameters to *down-stream* tools such as the high-level language compiler for the implementation's source language. This can be achieved by having the code generator produce a command or control file that directs the operation of the down-stream tool and

by providing some mechanism in the control file for the augmentation of its options and directives. This is done for the C/Unix code generator in this work by having it produce a *makefile* for each design that is implemented. The makefile that is produced includes files called *Makefile.preamble* and *Makefile.postamble* which are initially empty but may be edited by the user to include additional macro definitions for options (e.g., COPTS which specifies options such as “include symbolic debugging information in executable” for the C compiler) or additional rules respectively.

F-3: Generating Separate Source Files: this feature permits the designer to optionally specify a filename for certain individual design components. The design components for which these optional filenames are permitted are those that have an *internal* view. Thus *Boxes*, *Robots*, *Buttons* and so on are permitted to have filenames associated with them while *Pull Fingers* and *Install Arcs* (among others) are not. There is a default file name associated with the entire design and, should a component not have a filename assigned to it, it is grouped with its parent component. In this way, every component has a file name: either an explicit file name provided for it by the designer; a parent that has an explicit file name provided for it; or the default file name associated with the design's root component. The filename itself is provided in an annotation which appears on the same panel that is used to provide the component with a name. This is shown in Figure 13 where a portion of the lower left corner of the TimeBench window is overlaid with the *edit label* panel for the actor *courier3*.

FIGURE 13. Appearance of the File Name Annotation.

A schematic depiction of the association of filenames and design components is shown in Figure 14. This figure shows three layers of a design. Each layer contains one or more views corresponding to the various components which inhabit the layers. Layers are identified with the notation L_n where n is the layer number and component views are identified with the notation CV_n where n is the (hierarchically composed) component number. Components with explicit filenames associated with them are depicted by a medium gray shading and by the filename being shown within quotation marks adjacent to the component's view. Thus components CV_1 , $CV_{1.3}$ and $CV_{1.4}$ have filenames "ESystem", "Info-Panel" and "POTest" respectively while component $CV_{1.1}$ inherits its parent's name (ESystem) as does $CV_{1.1.1}$. Component $CV_{1.4.1}$ recursively inherits its parent's name (POTest).

This feature permits the tool user to group whole areas of related components into separate source files with relative ease. This can have important benefits for the partial generation of systems (throw away everything in file "XYZ" for example), for configuration control; and for compartmentalizing source code changes thereby lessening the re-compilation burden.

FIGURE 14. Schematic Representation of Filename Assignment to Design Components

F-4: Inclusion of Foreign Source: A usable code generator should allow the user to contribute user-written code to the implementation. As high level language compilers are obliged to provide information on interfacing assembly language routines to their generated code, so must the translators generated with MetaGen provide a mechanism for the inclusion of user-written source in the implementation. In our solution, this is achieved by providing a pragma in the Machine Chart Language. This pragma is one of several that

were added to the language to support the various extensions required by this work. The basic format of a pragma is:

```
--$pragmaname optional_parameters
```

where the first two characters are dashes followed immediately by a dollar sign and the name of the pragma. If the pragma requires or permits parameters, then they appear after some white-space following the pragma name. Since the pragma begins with two dashes, it is equivalent to a comment as far as the definition of MCL is concerned. A list of currently implemented pragmas appears in Table 8.

TABLE 8. List of Current Pragmas

Name	Parameter(s)	Description
include	file name base	Causes code generator to textually include a source file in the generated code at the point at which the pragma appears. "File name base" is augmented by the target name in order to produce the name of the source file to include and preserve portability of the annotation.
optimize	goal (one of space/time)	Directs the code generator to optimize the code generated for the MCL fragment in which this pragma appears. The goal of the optimization is either space (memory) or time and can be pursued either through ADT implementation selection and/or directives to the High-level language compiler.
stub	file name base	Causes the code generator to not generate code for the MCL fragment in which this pragma appears. "File name base" is augmented by the target name in order to produce the name of an object file which is to be linked with the application and to preserve portability of the annotation.

For inclusion of foreign source code, the pragma is `--$include <filenamebase>` where `<filenamebase>` is replaced by the root name of the user-supplied source code. As an example of the use of this annotation, consider Figure 15 where the MCL code for the agenda of the Push Button *next* is shown (the "before" picture). This MCL code, when

transformed into the target's implementation language might appear as shown in Figure 16.

FIGURE 15. Example of the \$include Pragma (Before)

FIGURE 16. Implementation of Push Button *next* - No Foreign Source Included.

```

1  /* --- Begin Button Body:next --- */
2
3  void next_Button()
4  {
5      print_string(create_string("About to execute nothing.));
6      print_newline();
7      print_string(create_string("Done executing it.));
8      print_newline();
9  }
10 /* -- End Button Body:next --- */

```

When the foreign source inclusion annotation is applied to the agenda for the *next* push button, the MCL code appears as in Figure 17 (the "after" picture). In this case, the foreign source code (Figure 18) is inserted in the generated implementation at the point where the pragma appeared and ends up in the implementation as shown in Figure 19.

FIGURE 17. Example of the \$include Pragma (After)

FIGURE 18. Foreign Source to be Included in the Implementation of *next*.

```

1  {
2      /*
3       * This is the "generate_next.c" source file. This source
4       * will be included in the implementation generated for
5       * the button "next".
6       */
7
8      printf("next_Button called\n");
9  }

```

FIGURE 19. Implementation of Push Button *next* - Foreign Source Included.

```

1  /*      Begin Button Body:next --- */
2
3  void next_Button()
4  {
5      print_string(create_string("About to execute
generate_next.c. "));
6      print_newline();
7      {
8          /*
9              * This is the "generate_next.c" source file. This
10             * source will be included in the implementation
11             * generated for the button "next".
12             */
13
14             printf("next_Button called\n");
15         }
16         print_string(create_string("Done executing it. "));
17         print_newline();
18     }
19  /* --- End Button Body:next --- */

```

This facility has a subtle mechanism to enable the design to stay re-targetable without changes. When the translator is generating code for the design component whose MCL contains the include pragma, a prefix is attached to the filename to indicate the target. In this way, the MCL can remain constant (i.e. no changes need be made to the MCL to retarget the implementation). An additional point to note is that if the file is not found, then a comment reflecting this is injected into the generated source at the point of inclusion instead of causing an error. This permits the user-written code to be absent for targets which don't require it. In this way, native features of a target environment which have no counter-part in another environment may be used. One example of such a native feature is exception handling in Ada.

The importance of providing this foreign source inclusion capability through the indirection of an include directive is that the MCL may remain implementation neutral;

i.e., using a High-Level Language as a design language hampers re-targetability [31] because the high-level language chosen for design is usually that of the implementation target and other targets may support different languages thus necessitating *translation* of the design fragments. As well, foreign source inclusion creates the possibility for the generator to create a complete example of the implementation including the connection between automatically-generated code and legacy code without the need for manual polishing each time the generator is re-run. This is very important for keeping the design and implementation in synchronization [31].

F-5: Specifying Task Priorities: MachineCharts provide for robots to have differing priorities. The priority execution model is a straightforward one of round-robin within a priority level with preemptive execution of the highest priority ready-to-run robot. The emulation of these scheduling semantics by the target library, while possible, is beyond the scope of this work; however, the code generator can perform simple priority translations to preserve priority ordering between the design and the implementation. For example, in MachineCharts, priorities run from 1 to 255 with 1 being the highest, while in Unix, the most readily adjusted corresponding parameter of a process is its *nice* factor which runs from 0 to 20 with 0 being the highest. The task priority is specified for an actor in its *edit label* panel, an example of which is reproduced in Figure 20 where it can be seen that the actor *courier3* is assigned a priority of zero.

FIGURE 20. Specifying Task Priorities.

<input type="button" value="Done"/>	
Name :	<i>courier3</i>
File Name :	<i>courier3</i>
Actor Priority :	0

F-6: Component Stubbing: a usable code generator must provide the facility to link the object code from the generated source with existing foreign object code. Exam

ples of foreign object code include interfaces to real-time executives, tool kits for user interfaces, libraries of numerical methods, etc. A desirable aspect of linking with foreign object code however is that type checking in the design should be preserved where possible [37]. The traditional K&R model of separate compilation for the C programming language provided no enforcement of type checking across separately compiled modules thereby permitting various kinds of errors. In our solution, the foreign code is represented by a full-fledged Machine Chart design component, complete with typing and behaviour specified. When generating the implementation however, it would be redundant to generate source code for the pre-existing foreign object code. This feature aids the tool user by allowing the specification of design components for which there should be no code generation performed. This feature is the homologue of F-5: Inclusion of Foreign Source.

Through the use of an additional pragma, the *stub* pragma, the code generator can be directed to reference an object but not generate code for it. This allows the design consistency checker and design interpreter to operate on the design successfully and, without changing the design, generate an implementation that links correctly. An example of the application of this directive appears in Figure 21 and Figure 23, wherein the MCL code for the agenda of the push button *sink* is shown both before and after the inclusion of the *\$stub* pragma. In the former case, the generated implementation for the Push Button *sink* might appear as in Figure 22 while in the latter case, no implementation is generated at all.

FIGURE 21. Example of the \$stub Pragma (Before)

FIGURE 22. Implementation of Push Button *sink* - Before Stubbing.

```

1  /* --- Begin Button Body:sink --- */
2
3  void next_Button()
4  {
5      print_string(create_string("Push Button sink was called.));
6      print_newline();
7      print_string(MCL_string_element(data, i));
8      print_newline();
9      print_string(create_string("Push Button sink finished.));
10     print_newline();
11 }
12 /* --- End Button Body:sink --- */

```


FIGURE 23. Example of the \$Stub Pragma (After)

3.4.2 Example of Adding an Annotation

In this section, the steps in adding support for a particular annotation are identified and illustrated with an example. Although each feature and its implementation is unique, each one that is based on user annotations follows a similar development process:

- identify appropriate requirement to address with a feature based on user-annotation;
- devise a technique to address the identified requirement;
- determine what information will be provided by the annotation;
- determine where in the design representation to best store the information;
- determine how to present the annotation to the tool user;

- modify the translation rules to act on the additional information provided by the annotation;
- modify the inspector layer of TimeBench to provide code generators with access to the annotation's information; and,
- modify target libraries to support any new code that may be generated.

Implementation of F-3: Generating Separate Source Files

Several requirements for a practical code generator could be partially addressed by a capability of the automatic code generator to produce output in separate source files. Since it would be desirable for these separate source files to have meaningful (and predictable) names, it would seem that this capability is a good candidate for the user-annotation mechanism with the annotation providing the files' names. An intelligent default value for this annotation mechanism should be chosen so that in the degenerate case where no annotations are provided by the user, the entire implementation source code that is generated is placed in a single file. To deal with the problem of *reference* and *extent*, as described earlier in "Design of Features" on page 50, this feature should possess a *natural* mapping from the structure of the design to that of the location of implementation components so that: the reference and extent of the annotation would seem reasonable to the tool-user; and, there is not an unnecessary burden placed on the code generator when trying to duplicate in the implementation technology the scoping rules of entities in the design. To satisfy this guideline, the correspondence between the internal view of a MachineChart element *M* (possibly decomposed through several levels) and the placing of implementation components within a file whose name is associated with *M* would appear to be the *natural* mapping desired. To state the function of this feature concisely then,

- A MachineChart Component will have its implementation (if generated) appear in a file whose name is either: directly associated with the component; or, recursively, that of its parent component. The root component of the design uses the user-supplied design name as a default file name.

The decision to associate an optional file name with a component that has an internal view suggests that the annotation be placed on the same user-interface panel that the tool-user uses to name the component. The code generator can extract this information from the design tree via an inspector called *get_enclosing_file_name*, that takes a design tree node as input and returns the file name associated with the node. This inspector, as most do, is implemented by invoking an Objective-C method called *enclosingFileName* on the node. The code for the basic *enclosingFileName* method appears in Figure 24.

FIGURE 24. Code for *enclosingFileName*

```
/*
 * returns the enclosing file name for the receiver
 *
 * if self has no particular file name, forward message to parent
 */

-(STR)enclosingFileName
{
    STR    fileName;

    fileName = [[self toolcomponent] getFileName];

    if (fileName && strlen(fileName)) {
        return fileName;
    } else {
        return [parent enclosingFileName];
    }
}
```

This method obtains the file name (if any) from the component naming user-interface panel of the design model component internal to TimeBench (the *toolcomponent*) associated with the design model component in the design tree. Then, if the file name is

not null, it is returned otherwise the message is forwarded to the node's parent. Thus, this message may be relayed up the model hierarchy until it reaches the root node of the design tree wherein the name of the design model is returned as a file name.

One complication in implementing this feature is the need to translate from a truly hierarchical representation (the design) to one that is more flat (the implementation source code) and could be better described as a network in one (or two at most) layer(s). Referring again to Figure 14, CV1.4 and its descendants form the internals of a component which appears within the CV1 view and similarly CV1.3 also has an external boundary depicted in the CV1 view. Because these two sub-trees of the design both have different file names in this example, and they both have components visible on the same view (CV1), there is the potential for interaction *across* file name boundaries. Thus the first step in generating code for such a system is to partition the design tree based on the enclosing file in which a component's code will reside. Once this is done, interactions between the partitions can be identified, and the necessary interface declarations may be generated.

Conventional Wisdom: "Many ideas have passed into the computer culture and are taken as gospel by software developers" — SOFTWARE THAT WORKS

4.1 Introduction

The purpose of this chapter is to derive and present a set of evaluation criteria for determining the degree of practicality of an automatic code generator. The criteria presented draw heavily from the requirements presented earlier in chapter II and from accepted measures of software quality [1, 6, 13, 20]. The criteria themselves are expressed in terms of narrative elements and ratings elements as inspired by [30], [33] and [56]. This set of criteria is applied in the subsequent chapter to evaluate the work described in this thesis.

The over-riding theme of this chapter is that a code generator can be considered to have a high degree of practicality when it:

- 1) produces software with a high degree of quality, and
- 2) successfully addresses the requirements for code generators presented earlier.

4.2 From Requirements to Evaluation Criteria

In this section, the requirements for a practical automatic code generator are discussed with a view to creating a set of evaluation criteria for practical automatic code generators. The approach taken in creating the criteria is taken from [30]; however, all the evaluation criteria developed here are classed as *rating* criteria. Rating criteria are appropriate when the degree of support for some feature or the extent to which an implementa-

tion satisfactorily meets some requirement needs to be expressed. An example of a rating criteria might be "Level of support for re-use" with possible values of "A", "B", "C" and "D" corresponding to "at design as well as source and object levels", "at object-code and source levels", "at object-code level only", and "none" respectively. The rating criteria are summarized in table Table 9 where they are numbered and briefly described.

TABLE 9. Evaluation Criteria for a Practical Automatic Code Generator

Id	Criterion Summary
EC-1	Attention to correctness of generated code
EC-2	Attention to efficiency of generated code
EC-3	Attention to readability of generated code
EC-4	Ease with which generated software interfaces with existing code
EC-5	Degree of user control over static software structure
EC-6	Ease with which generated code can use native target features
EC-7	Degree to which the code generator supports software re-use
EC-8	Accommodation of compiler/linker directives
EC-9	Re-targetability of code generator
EC-10	Extent of support for partial/incremental generation
EC-11	Implementation-language neutrality
EC-12	Degree to which generator operates automatically
EC-13	Should integrate with a design tool that supports High Level Design
EC-14	Generated code should be debuggable
EC-15	Support for Portable Implementations

Before the interpretations of these criteria are described, it is worth expanding on the observation of the previous chapter that software quality attributes are often in tension with each other. The obverse is also true — namely that some criterion overlap others and that efforts to achieve improvement in one area may pay dividends in another. The result is a set in which each criterion may be independent, at odds with, or reinforcing of other criteria. In particular, as will be seen shortly, criteria EC-4, EC-5, EC-7 and EC-10 all benefit

from the ability to produce an implementation that is not monolithic, but rather decomposable. Also, the requirements R-12 (Administrability) and R-14 (Expressive Power) have not been mapped to evaluation criteria here as they are both quite broad in scope and are therefore not amenable to a simple rating criterion.

EC-1: Correctness. This evaluation criterion deals with the degree to which the automatic code generator addresses correctness of the generated implementation. A rating of level C indicates no specific support or attention is paid to correctness; level B indicates that the translator has been proven to be correct; level A indicates that the translator can assess a design for correctness. Note that for ratings of level B and above to be meaningful, the design must be expressed in an appropriate formalism.

EC-2: Efficiency. The efficiency criterion assesses the extent of support and attention paid by the translator to concerns of implementation efficiency whether they be memory consumption or execution time related. A rating of level D indicates that no control over space or time consumption is provided to the user; level C allows the user to select among alternatives; level B permits specification of alternatives, i.e. the user may provide coded implementations and level A provides for the tool to select the appropriate implementation alternative (possibly from user-supplied implementations) given only the users preference for optimization goal.

EC-3: Readability. The rating for readability gauges the scope of support for the automatic code generator to produce readable implementations. A rating of level D means that the implementations are not human-readable; level C means that the implementations are human-readable; level B means that there is specific attention paid to readable implementations by providing the user with some way to specify readability attributes; and level A indicates that the translator is completely *open* with respect to read-

ability, providing the user with both a selection of readability options and the means to specify custom readability properties. An example of a level A feature would be allowing the user to select among several variable naming conventions or to provide their own.

EC-4: Ease of Interface. An important yardstick in the practicality of an automatic code generator is the ease with which the generated implementations can be made to interface to existing external entities. Sources of these existing entities abound: code produced by hand; legacy systems; interrupt handlers; device drivers; standard system libraries; and highly time-critical and/or device-dependent functions are just some examples of environmental entities that would be useful to interact with. A rating of level D indicates that generated systems are monolithic and do not interact outside a set environment; level C indicates that interaction is possible but requires specific support from the developer of the translator; level B indicates that the user is capable of (indirectly) specifying the interaction and interfacing¹; level A indicates that the user is capable of *directly* specifying the interface [37] as opposed to relying on an intermediate transaction centre to manage interfaces [24].

EC-5: Control Over Static Structure. This criterion rates the support provided by the automatic code generator for controlling the static structure of the generated implementation [19]. Because there is a conceptual gap between a design description and its actual implementation as hardware and software components [24], there is a need either to understand the mapping of structure from one to the other or a need to control that mapping. This capability also has ramifications with respect to configuration control (see EC-10: Partial/Incremental Generation Support) which is an absolute necessity for systems of any appreciable size [24], reasonable life expectancy or quality goals. A level D rating is

1. One possible way to manage interfaces between generated code and externally-provided code is to create a piece of "glue" code that serves as a transaction centre, marshalling messages and their parameters for exchange between the two types of software.

given to generators that provide no control over the static structure of the generated implementation; level C is awarded to those that have some documented structure that produces separate components from the design; level B means that the user may specify the exact structure (to a reasonable granularity) while level A means that not only may the user specify the exact structure, but the user could provide hints or goals for the partitioning of the system which could then be acted upon by the generator.

EC-6: Ease of Using Native Target Features. This rating is used to assess how easily native features of the target environment can be exploited in the implementation process. Level D indicates no native target features can be employed; level C indicates that a preset palette of special target features may be available for use; level B indicates the user can select or specify native features to be applied; while level A indicates that the generator will assist the user in re-targeting native features where possible. An example of a feature that rates a level A is the mapping of native operating system inter-process communication semantics from one target to another.

EC-7: Re-use Support. Provisions for supporting software re-use are measured by this criterion. Although design re-use is the ultimate goal here, re-use at source and object levels may be essential for the practical realization of design-level re-use [22]. A rating of level D corresponds to an absence of any specific support for re-use; level C indicates that object-code may be re-used; level B indicates that re-use may be performed at the source-code level and level A means that re-use of development products is possible at the design stage. In this criterion, higher level ratings are inclusive in that systems with a certain rating in this criterion meet the decision criteria of all lower ratings too.

EC-8: "Downstream" Tool Support. This criterion is used to capture the support the automatic code generator has for interacting with other software development

tools such as compilers, linkers, profilers and source-level debuggers. A level C rating means that no specific support is provided; level B means that the generator is aware of and can influence the operation of downstream tools by setting certain options for these tools; level A means that the user can be involved in the selection and specification of the downstream tools' options.

EC-9: Generator Re-targetability. An important characteristic of practical code generators is the ability to readily switch targets during implementation or for the purposes of re-implementation. If the target of a generator is fixed and cannot be changed, then the generator is rated level D; if the generator produces a standard programming language and assumes a portable standard environment such (e.g. POSIX), then level C is attained [28]; level B is warranted in the case that the code-generator developer is able to provide generators for specific targets with some degree of ease; while level A is attained when re-targeting the generator is a user operation.

EC-10: Partial/Incremental Generation Support. In conjunction with the ability to control the static structure of the generated software (see EC-5: Control over static structure), this criterion is an important one for systems of any realistic size, life-expectancy or quality aspirations. Partial generation is important as an aid to rapid implementation of corrective action or rapid generation of alternatives for evaluation. "For large systems, separate compilation is a practical necessity." [24] "A local change in, for example, a process graph should not lead to that[sic] the complete system has to be regenerated or re-compiled." [31] Incremental generation is closely related and, depending on how it is implemented, may be considered a special case of partial generation. It has the additional benefit of facilitating rapid prototyping if required by the development plan. Generators that provide no specific capability in this area rate level D; the ability to generate part of an implementation rates level C; level B is reserved for generators that can perform a *smart*

generation that produces the minimum source necessary after a design or annotation change (analogous to *make*); and level A is reserved for generators that can provide incremental generation that does not require the design to be complete.

EC-11: Implementation-language neutrality. An important characteristic of a practical automatic code generator is its independence from the generated implementation's target language. This is related to re-targetability of the generator but goes further as the independence is a measure of coupling between the design and the implementation especially where the user is permitted to provide literal source code in the implementation language or to incorporate existing software in the implementation. If a generator requires fragments written in the target language in order to fully specify the design, then an undesirable coupling has been made. Although re-targetability suffers, a more serious problem is premature implementation decisions such as those that might be required if the target language is strongly typed. A high-level design language is the ideal solution here. Consequently, this aspect of generator quality warrants a separate evaluation criterion. A rating of level D demonstrates that the generator is incapable of handling any inputs expressed in the implementation language and is therefore trivially neutral; level C generators permit the user to supply either design or annotation information expressed in the implementation language for inclusion in the generated implementation; level B generators are able to provide a design language that avoids premature implementation decisions, but still are able to accommodate annotations coded in the implementation language; and finally, level A generators are capable of transforming the annotations from one implementation language to another if implementing for other targets is desired [37].

EC-12: Degree of Autonomy of Translator. This measure of generator practicality pertains to the way the software designer interacts with the generator. If the generator requires constant guidance from the designer to proceed or depends upon the designer

to resolve problems encountered during generation, then it rates level D. Generators that are capable of operating without intervention rate level C; those that still require initial guidance to generate their best implementations are graded at level B; and, generators that can select appropriate defaults for these implementation decisions without input from the designer rate level A. An iterative or interactive approach is unrealistic for projects of moderate or large size, or with more than a small staff.

EC-13: Integration with High-level Design Tool. A practical code generator will be integrated with a high-level design tool. This avoids any translation between the high-level design as expressed in a design tool and the form of input acceptable to the code generator. A rating of level D is given to code generators that are stand-alone and do not have a design tool integrated with them; level C is for generators that stand-alone yet have their input in a standardized form such as SDL [5]; level B is attained by generators that are integrated with a design tool; and level A is award to generators whose design tool offers design checking and simulation.

EC-14: Generated Code can be Debugged. A practical automatic code generator should allow the designer to "close the loop" by producing an implementation which can be instrumented, observed and profiled. A rating of level D means that the generated implementation can not be debugged with the ease associated with most modern software development (e.g. source-level debug); level C means that the code can be debugged but there is difficulty in tracing back to the design elements that are involved; level B indicates that tracing is feasible but a strong temptation to patch the generated code rather than the

design still exists; and, level A which means that debugging can be done directly from the design level or that fixing the design is the path of least resistance for the designer.

EC-15: Support for Portable Implementations. This quality characteristic requires a rating criterion for evaluating code generators. Level D automatic code generators are those that support only one target; level C's support multiple (but a fixed selection of) targets; vendor support for re-targeting the generator rates level B; and level A is given when re-targeting the generator can be a user activity.

Case Study: Communicating Stacks of Modules

This chapter, which presents a case study in the application of the improved automatic code generator, is divided into three main parts: an introduction to the design which is used to demonstrate and evaluate both the original TimeBench code generator and the improved one; a section which looks in detail at the application of the new code generator features to the implementation of this design; and, a concluding section wherein the evaluation criteria of the previous chapter are applied to both code generators.

5.1 The Case Study Design Model

The case study selected for this thesis involves a design example which is part of the standard TimeBench distribution and which models two nodes in a point-to-point communication system. Each node is designed as a layered stack of components implementing the *Alternating-Bit Protocol* (ABP) (see the Estelle model DIS9074). From a hierarchical decomposition viewpoint, there are four levels of detail in the design model. This design is a good candidate for evaluating this work because it is of non-trivial size, represents a realistic application of the tool and is likely to be familiar to the audience for this work and therefore permits the reader to focus more easily on the main ideas of this work instead of the details of an unfamiliar application domain. The selection of this design is not meant to be an endorsement of its quality for any other purpose.

The top-level MachineChart of the case study's design model, which is shown in Figure 25, depicts six major components: boxes *src*, *dst*, *comm1* and *comm2*; and, actors

courier3 and *courier4*. These components are arranged so that *src* and *dst* can communicate (virtually) with each other by sending data to, and receiving data from, their corresponding communication boxes: *comm1* and *comm2* respectively. The communication boxes implement the ABP and have their protocol data units relayed between themselves with the help of two actors: *courier3* which relays data from *comm2* to *comm1*; and *courier4* which relays data in the opposite direction¹. There is symmetry in this design between the boxes on the left and right sides of the MachineChart as well as between the two actors.

1. A *courier* is an autonomous component that continually tries to fetch data items from a source and, once successful, tries to send them to a destination. It can be thought of as a buffer of at least one data item. Its purpose is to permit data exchange between two other components without requiring either one to *block*.

FIGURE 25. Stack of Communicating Components (Top-Level MachineChart)

Next, the *src* box is decomposed into a lower level MachineChart that shows its internal components (Figure 26). The *src* component consists of an actor (*s_task*), a courier actor, an *init* engine (for the box) and its interface button *s_in*. The engine shown in this MachineChart takes care of any initialization required by the box. The autonomous behaviour of the *src* box comes from the actor *s_task*. This actor is responsible for creating messages for, and sinking those that arrive from the *dst* component.

FIGURE 26. The Src Box (MachineChart)

A further decomposition of the *src* box leads to decompositions of the *s_task* actor (Figure 27) and the *courier1* actor (Figure 31). From its MachineChart, it can be seen that the *s_task* actor has two gate buttons (*s_u* and *s_d*) as well as a data store (*i*) and an engine. It can also be seen from this chart, that there is no connection from the *src* engine to the *s_u* gate button. This means that this component can only send data and not receive it. This simplification has been made to reduce the amount of generated code in this case study and can be made without losing anything of the case's illustrative power because the *dst* component contains the opposite behaviour.

FIGURE 27. The *S_task* Actor (MachineChart)

The behaviour of *s_task* is specified by the MCL code (Figure 28) in the agenda of its engine, *src*. This code shows that *s_task* enters an infinite loop of pulling the *s_dn* pull finger and then changing the value of the index *i*, always being careful to keep it in the range of 1 to size.

FIGURE 28. The Agenda of *S_task*'s Engine

```

-- Box src
engine src is
begin
  i := 1;
  size := #data;
  --
  -- Infinite loop version
  --
  loop
    pull s_dn;
    i := i mod size;
    i := i + 1;
  end loop;
end;

```

OF / DE

2

PRECISIONSM RESOLUTION TARGETS

A look inside the data store *i* (Figure 29) and the engine for the gate button *s_d* (Figure 30) reveals the significance of this agenda: the index *i* is used to step through the characters in the string named *data* such that, every time there is a rendezvous with the gate button *s_d*, it returns a new character in the parameter *sdu*.

FIGURE 29. The Contents of Datastore I (MCL code)

```
store i is
  i      : integer;
  data   : string; --      := "This is a test";
  size   : integer;
end;
```

FIGURE 30. The Agenda of the S_d Gate Button's Engine (MCL code)

```
engine s_d is
begin critical_region
  sdu := data(i);
end critical_region;
begin
end;
```

The final component of the *src* box that permits decomposition is the *courier1* actor, shown in Figure 31. In this MachineChart, two data stores (*pdu* and *sdu*) are shown as well as the engine which provides *courier1* with its behaviour.

FIGURE 31. The courier1 Actor (MachineChart)

The MCL code for this courier component is shown in Figure 32 to show concretely how the behaviour of a courier, described earlier, is specified in MCL. Here it is seen that: the courier loops forever, pushing the *rx_sdu* finger so as to rendezvous with *v_task*'s gate button, *s_d*; accepts the data from *sdu* and transfers it to *pdu*; and finally pushes the *tx_pdu* finger so as to rendezvous with an actor within the *comm1* component.

FIGURE 32. The Agenda of Courier1's Engine (MCL code)

```

- Box src
engine courier1 is
begin
  loop
    push rx_sdu;
    pdu := sdu;
    push tx_pdu;
  end loop;
end;

```

With the exception of the transmit/receive asymmetry mention earlier, the *dst* component decomposes identically to the *src* one and is not shown here for this reason.

The most complex components of the design are found in the boxes *comm1* and *comm2*. Since they are completely symmetrical in this design, only *comm1* is examined here. The core component of the *comm1* box, as shown in Figure 33, is the *comm1* actor with five gate buttons: *s1_c1*, *s2_c1*, *p1_c1*, *p1_c2* and *tol*. Also shown are: four components (*cl_in*, *s_in*, *cl_dn* and *cl_up*) on the interface of the *comm1* box; an actor called *timer1*; a datastore called *cl_store*; an actor called *courier2*; and the box's initialization engine, *init*. The *comm1* actor, with help from the *timer1* actor, provides the central point of control for the *comm1* box. To understand its role in this component, it is necessary to examine its internal view as shown in Figure 34.

FIGURE 33. The Comm1 Box (MachineChart)

The *comm1* actor consists primarily of a button (*perform1*) that acts upon requests from either the interface components in this actor, or from an *Abstract Controller Machine* (ACM) named *albit_1* that implements the state transitions of the ABP communication protocol.

FIGURE 34. The Comm1 Actor (MachineChart)

The control aspect of the ABP communication protocol can easily be described as a finite state machine and therefore naturally maps to an ACM MachineChart component. In this case, the ACM *altbit_1*, as shown in Figure 35 contains two operational states (and an initialization state) called *estab* and *ack_wait*. Transitions between states, and transitions that originate and terminate on the same state are shown in this MachineChart. Each transition represents the occurrence of some event and is associated with an action, possible pre-conditions, inputs and outputs. The events and actions associated with this particular ACM are shown in Figure 36.

FIGURE 35. The Altbit_1 Abstract Controller Machine (MachineChart)

Alternating Bit, from ESTELLE spec DIS9074

FIGURE 36. The States and Actions of the ACM Altbit_1 (MCL code)

```

ACM altbit_1 is
  Type EVENT_TYPE is ("init", "receive_request", "send_request",
    "timeout", "data_response" );
  Type ACTION_TYPE is ( "initialize", "ack_received",
    "data_received", "data_request", "receive_response" );
end;

```

To give an overview of the *comm1* actor's behaviour, it is useful to examine the MCL code for the agenda of the *perform1* button's engine (Figure 37). In this code it can be seen that each of the actions specified by the ACM *altbit_1* are implemented by *perform1*. The events arise primarily from rendezvous with the gate buttons. When such a rendezvous occurs, the datastore *ACMData* is updated by the gate button involved with the rendezvous so as to indicate the new event, data may be buffered in the *comm* datastore and then a finger is pushed to activate the *perform1* button.

FIGURE 37. The Agenda of Perform1's Engine (MCL code)

```

button perform1
engine perform 1 is
begin
  push doNothing;

  case anAction is
  when altbit_1.initialize =>
    send_seq := 0;
    recv_seq := 0;
    -- print "Comm1: initialized.", newline;
  when altbit_1.DATA_request =>
    pduDn_data := sduDn;
    pduDn_seq := send_seq;
    pduDn_type := 0,
    pduDnW := true;-- enable NDATA_request for data.
    -- print "comm1: data request.", newline;
    push start;
  when altbit_1.data_received =>
    sduUp := pduUp_data;
    pduDn_type := 1;
    pduDn_seq := pduUp_seq;
    pduDnW := true;-- enable NDATA_request for ack.
    sduUpW := true;-- enable upgoing data.
    recv_seq := ( 1 + recv_seq );
    -- print "comm1: data received.", newline;
  when altbit_1.ack_received =>
    push stop;
    send_seq := ( 1 + send_seq );
    sduDnW := true;-- allow more data.
    pduUpW := true;-- allow more acks.
    -- print "comm1: ack received", newline;
  when altbit_1.RECEIVE_RESPONSE =>
    pduUpW := true;
    -- print "comm1: receive response", newline;
  when others =>
    print "comm1:perform1: bad action >", anAction, newline;
  end case;
end;

```

A representative example of this appears in Figure 38, where the MCL code in the agenda for the engine of the gate button *s/_c/* shows what happens when the *src* box tries to transmit an *sdu* to the *comm1* box. In this case, the waiting place *sduDnW* is activated thereby preventing any more transmission requests while this one is underway, a *send_request* is recorded as the new event to be processed, the incoming *sdu* is copied to the

This can be seen in the generated implementation, by first noting the contrast in the *s_task* and *d_task* MCL code and then examining the generated implementations. The *s_task* MCL code extracts an element from its local *data* (Figure 28) which is of the MCL type *String*, while the *d_task* MCL code inserts an element into its local *data*. Consequently, the code for the *src* box (appearing in the files named *source* in Appendix C and below in Figure 41) contains declarations and uses of the variable *data* which correspond to the *imp4* implementation of the MCL type *String*, while the code for the *dst* box (appearing in the files named *dest*, also in Appendix C and below in Figure 42) contains declarations and uses of the its variable *data* which correspond to the *imp2* implementation (contrast the lines in bold typeface in Figure 41 with those in Figure 42).

FIGURE 41. *s_task* - Fragment of Source Generated by CUnix_codegen.

```

51  ...
52  /* --- Begin s_task_type Task Body --- */
53
54  void s_task_type()
55  {
56      /*-- local_gate_object_list --*/
57      /*-- gate_formal_parm_declaration_list --*/
58      MCLString sdu;
59      MCLString sduin;
60      /*-- object_declaration_list --*/
61      MCLInteger i;
62      imp4_String data;
63      MCLInteger size;
64      int event;
65
66      MCLBoolean timeExpired;
67
68
69      /* - local_declarations --*/
70
71      /*-- init_gates --*/
72      int s_d_gate_caller, s_d_gate = ipcPullInit("s_d_gN");
73      s_d_parms_type s_d_parms;
74      int s_u_gate_caller, s_u_gate = ipcPullInit("s_u_gN");
75      s_u_parms_type s_u_parms;
76
77      /*-- task_call_declarations --*/

```

FIGURE 39. The Timer Actor (MachineChart)

The final top-level components that permit decomposition in this case study's design model are the *courier3* and *courier4* actors. Again, perfect symmetry is present in these components, and they represent the standard form of courier component, so only the MachineChart for *courier3* (Figure 40) is included here.

FIGURE 40. The Courier3 Actor (MachineChart)

The *courier3* MachineChart shows an engine and a data store (*pdu*). This is the canonical courier configuration as discussed earlier in this section and therefore doesn't warrant any special discussion here.

5.2 Annotating the Design

In this section, the application of the code generator features that were developed in this work and presented in chapter II is described. These features were applied in the implementing of the case study's design model. The reader is referred to chapter III for examples which show the appearance of particular annotations, and to Appendix C for the code that is actually generated by the C/Unix code generator.

5.2.1 Partitioning the Implementation

For this case study, file names have been provided for each of the six top-level design components. This leads to each of these components' implementations being placed in separate files. In fact, because of the need for an interface specification and implementation body for each of the implementation components, and because of the convention followed in the C language, two files are created for each component: a *.h* file and a *.c* file that correspond to the interface and implementation respectively. A main program is generated in a file with the name *<design>_main.c* where *<design>* is the name of the design being implemented.

5.2.2 Automatic ADT Implementation Selection

A typical trade-off in the implementation of an Abstract Data Type (ADT) is whether adding an element to the data structure is to be more efficient than locating an existing element within the data structure. For example, appending a record to the end of a sequential file makes addition efficient, but searching inefficient. In light of this and for the purposes of this case study, the decision function that selects among implementations of the MCL String type has been tuned to select one implementation (*imp4*) if the number of element additions exceeds the number of element look-ups; another implementation (*imp2*) if the reverse is true; and, if the numbers are equal, a third implementation (*MCL*) is chosen.

in the program at link time. The effect of this pragma can be verified by noting the absence of any generated code for the push button *sink* in the source code *dest*. Instead, a comment appears in Figure 47 on line 20 of *dest.c* showing that a stub appears for the button *sink* and that the foreign object file is called *absorb.o* (see Figure 56 in Appendix C for full context).

FIGURE 46. Agenda for Push Button *sink*.

```

1  -- Push Button sink
2  engine main is
3  begin
4  --$stub absorb
5  end;
```

FIGURE 47. *sink_Button* stub- Source Fragment Generated by CUnix_codegen.

```

...
19
20  /** stub for sink: absorb.o **/
21
22
...
```

Additionally, the file name *absorb.o* appears in the generated Makefile (Figure 48) in order to direct the linker to include this foreign object code in the image of the implementation program. For completeness, the source code corresponding to the object file *absorb.o* appears in Figure 49.

```

78
79      * -- declare_frames -- *
80      id_get s_dn_frame;
81      int s_dn_frame_rate;
82
83      /*-- init_frames --*
84      FD_ZERO(&s_dn_frame);
85
86      /*-- agenda_node --*
87      i = 1;
88      size = imp4_str_len(data);
89      for ( ; ; ) {
90          s_d_gate_caller = ipcPull(s_d_gate, (void *)&s_d_parms,
sizeof(s_d_parms_type));
91          sdu = imp4_str_element(data, i);
92          s_d_parms.sdu = sdu;
93          ipcEndCrit(s_d_gate_caller, (void *)&s_d_parms,
sizeof(s_d_parms_type));
94          next_Button();
95          i = (i % size);
96          i = (i + 1);
97      }
98  }
99
100  * --- End s_task_type Task Body --- */
...

```

FIGURE 42. d_task - Fragment of Source Generated by CUnix_codegen.

```

...
32
33  /* --- Begin d_task_type Task Body --- */
34
35  void d_task_type()
36  {
37      /*-- local_gate_object_list --*/
38      /*-- gate_formal_parm_declaration_list --*/
39      MCLString sdu;
40      /*-- object_declaration_list --*/
41      MCLInteger i;
42      imp2_String data;
43      MCLInteger size;
44      int event;
45
46      MCLBoolean timeExpired;
47
48
49      /*-- local_declarations --*/
50
51      /*-- init_gates --*/
52      int d_d_gate_caller, d_d_gate = ipcPullInit("d_d_gn");

```

```

53     d_d_parms_type d_d_parms;
54     int d_u_gate_caller, d_u_gate = ipcPullInit("d_u_gN");
55     d_u_parms_type d_u_parms;
56
57     /* -- task_call_declarations -- */
58
59     /*-- declare_frames --*/
60     fd_set d_dn_frame;
61     int d_dn_flagGate;
62
63     /*-- init_frames --*/
64     FD_ZERO(&d_dn_frame);
65
66     /*-- agenda_code --*/
67     i = 1;
68     size = imp2_str_len(data);
69     for ( ; ; ) {
70         d_d_gate_caller = ipcPull(d_d_gate, (void *)&d_d_parms,
sizeof(d_d_parms_type));
71         imp2_modify_str_element(data, i, sdu);
72         d_d_parms.sdu = sdu;
73         ipcEndCrit(d_d_gate_caller, (void *)&d_d_parms,
sizeof(d_d_parms_type));
74         sink_Button(i);
75         i = (i % size);
76         i = (i + 1);
77     }
78 }
79
...

```

5.2.3 Incorporating Foreign Source Code

The design of the *s_task* robot has the gate *s_d* calling the button *next*. The agenda code for this button, Figure 43, contains an annotation on line 4 to include foreign source code, supplied by an external source file called *generate_next.c* (Figure 44). The result of generating an implementation for the *next* button appears in Figure 45. Note the included source, lines 25 to 33, is indented according to the level in effect when the pragma was encountered.

FIGURE 43. Agenda for Push Button *next*.

```

1  -- Push Button next
2  engine main is
3  begin
4      --$include generate_next.c
5  end;

```

FIGURE 44. *generate_next.c* - Foreign Source to be Included in the Implementation.

```

1  {
2      /*
3       * This is the "generate_next.c" source file. This source
4       * will be included in the implementation generated for
5       * the button "next".
6       */
7
8      printf("next_Button called\n");
9  }

```

FIGURE 45. *next_Button* - Source Fragment Generated by CUnix_codegen.

```

...
10 /* --- Begin Button Body:next --- */
11
12 void next_Button()
13 {
14     /* -- Push Button next */
15     (
16         /*
17          * This is the "generate_next.c" source file. This source
18          * will be included in the implementation generated for
19          * the button "next".
20          */
21
22         printf("next_Button called\n");
23     )
24 }
25 /* --- End Button Body:next --- */
26
...

```

5.2.4 Incorporating Foreign Object Code

The design of the *d_task* robot has the gate *d_d* calling the button *sink*. The agenda for the *sink* button, Figure 46, includes an annotation that indicates no code should be generated for this button and that the externally supplied object file called *absorb.o* is to be included

in the program at link time. The effect of this pragma can be verified by noting the absence of any generated code for the push button *sink* in the source code *dest*. Instead, a comment appears in Figure 47 on line 20 of *dest.c* showing that a stub appears for the button *sink* and that the foreign object file is called *absorb.o* (see Figure 56 in Appendix C for full context).

FIGURE 46. Agenda for Push Button *sink*.

```

1  -- Push Button sink
2  engine main is
3  begin
4  --$stub absorb
5  end;
```

FIGURE 47. *sink_Button* stub- Source Fragment Generated by CUnix_codegen.

```

...
19
20  /** stub for sink: absorb.o **/
21
22
...
```

Additionally, the file name *absorb.o* appears in the generated Makefile (Figure 48) in order to direct the linker to include this foreign object code in the image of the implementation program. For completeness, the source code corresponding to the object file *absorb.o* appears in Figure 49.

FIGURE 48. Makefile - Fragment Generated by CUnix_codegen.

```

...
1  include Makefile.preamble
2
3  INCLUDES = ../MetaTranslator/CLibrary
4  LIBS      = BoolLib.o IntLib.o ListLib.o RealLib.o SetLib.o StrLib.o
5  CC        = cc $(CCOPT)
6  OBJECTS = courier3.o courier4.o comm1.o source.o dest.o absorb.o
           comm2.o
7
8  all:  commdemo
9
10
11  commdemo:commdemo_main.c courier3.c courier3.o courier4.c
           courier4.o comm1.c comm1.o source.c source.o dest.c dest.o
           absorb.o comm2.c comm2.o
12      $(CC) -I$(INCLUDES) -o commdemo $(OBJECTS) $(LIBS)
           commdemo_main.c ipclib.o
13
...

```

FIGURE 49. absorb.c - Source for Foreign Object Code.

```

1  /*
2   * absorb.c
3   *
4   * Source for object code to be included in
5   * implementation.
6   */
7
8  void sink_Button()
9  {
10     /* The sink_Button will be provided by this code. */
11
12     printf("sink_Button called\n");
13 }

```

5.3 Code Generator Evaluations

In this, the closing section of this chapter, the evaluation criteria developed in this work and presented in chapter IV are applied to both the original (C/JNX) and new (C/Unix) code generators. To see the definitions of each rating, please refer to Table 9, "Eval-

uation Criteria for a Practical Automatic Code Generator" in chapter IV. Where the ratings differ or other points of interest arise, a detailed discussion is presented.

TABLE 10. Comparison of Original and New Code Generators

Id	Brief Criterion Name	Original	New
EC-1	Correctness	C	C
EC-2	Efficiency	D	A
EC-3	Readability	C	B
EC-4	Ease of Interface with Existing Code	D	B
EC-5	User Control of Static Structure	D	B
EC-6	Native Target Features	D	B
EC-7	Re-use Support use.	D	A
EC-8	Directives	C	B
EC-9	Re-targetability	B	B
EC-10	Partial/Incremental generation	D	C
EC-11	Implementation-language neutrality	D	B
EC-12	Autonomy of Generator	C	A
EC-13	Integration with HLD Tool	A	A
EC-14	Debug Support	B	A
EC-15	Portability Support	B	B

By applying the evaluation criteria of the previous chapter, we can see that:

EC-1: Correctness. Neither code generator provides any specific support or attention to correctness. In order to do so, the metatranslator itself would first have to be designed and implemented in such a way as to produce code generators that were demonstrably correct and which could assess TimeBench design trees for correctness too. One

complicating factor is the ability to include foreign source or foreign object code in the generated implementations as this may compromise correctness.

EC-2: Efficiency. The C/JNX code generator provides a single implementation for each abstract data type found in the design. Accordingly, it is assigned a rating of level D for this evaluation criterion. The C/Unix code generator on the other hand, provides for the automatic selection of appropriate abstract data type implementations as well as pragmas to direct the optimization goals of the code generator. Because of this, it is given a level A rating in this evaluation.

EC-3: Readability. Both code generators produce high-level language source with appropriate use of white-space and indentation to express the implementations and therefore rate at least a level C. The new code generator extends this capability by allowing the inclusion of externally-generated source code in the implementation. It also allows the partitioning of the implementation into separate source files. Since both the additional code and the increased modularity of the implementation could have a beneficial effect on the readability of the generated code, a level B is warranted in this case.

EC-4: Ease of Interface. The original code generator produces a monolithic implementation which is incapable of interfacing in a dynamic way to the target environment. Although the existence of a runtime library of sorts is relied upon by the code generator, there is no meaningful way for the design to indicate interfaces to existing software; consequently, a rating of level D is warranted. The C/Unix code generator delivers the possibility of specifying interfaces however and therefore rates at least a level B rating; how-

ever, it does not provide a means to specify the *exact* existing interface and therefore cannot avoid the manual creation of some "glue" code.

EC-5: Control Over Static Structure. Since its generated implementation is monolithic and its mapping from design component to source component is fixed, a rating of level D is given to the C/JNX generator. The C/Unix code generator does not permit the user to hint as to the general structure of the generated implementation or its design to source component mapping. The same can be said about allowing the user to precisely specify these things; however, the new code generator does allow some direct user control over both of these elements of the static implementation structure and, as a result, earns a rating of level B.

EC-6: Ease of Using Native Target Features. Although the original code generator does (necessarily) employ some native target features, the use of these features is fixed statically by the code generator and has a one-to-one correspondence with design elements. Therefore, the designer cannot expect the freedom of selection of these native target features and consequently, a rating of level D must be given to the C/JNX code generator. Level B is achieved by the new code generator because it permits the inclusion of externally-generated source code in the implementation and therefore provides an escape mechanism that accommodates the desire to use native target features.

EC-7: Re-use Support. Because the C/JNX code generator does not permit the partial generation of an implementation, nor does it allow easy interface to externally-generated components, no meaningful means of re-use at either the object or source-code

levels is possible, negating any practical case for design re-use. Both of these shortcomings are corrected by the C/Unix code generator giving it a level A rating.

EC-8: "Downstream" Tool Support. The original generator rates level C here as no specific support is provided. The new generator (rated at level B on this criterion) is slightly better as it generates a *makefile* for its downstream tools and therefore controls their operation somewhat more directly.

EC-9: Generator Re-targetability. The metagenerator architecture employed in the creation of code generators for TimeBench permits relatively easy re-targeting. The generators (both rated at level B in this area) cannot be re-targeted without a considerable amount of knowledge regarding the internal structure of the TimeBench tool however and therefore this is not seen as a feasible user activity.

EC-10: Partial/Incremental Generation Support. As already mentioned, the original code generator (rating level D here) produces monolithic implementations with no possibility of partial or incremental generation. Level C is attained by the new code generator as it can be directed to "stub-out" some of the implementation components.

EC-11: Implementation-language neutrality. As the original code generator is incapable of handling any inputs expressed in the implementation language, it is trivially neutral to the implementation language and therefore rates level D. The new generator is able to provide a mechanism that is able to accommodate annotations coded in the implementation language via the indirect `$include` pragma, rating a level B evaluation.

EC-12: Degree of Autonomy of Translator. A level C rating is given to the C/JNX generator in this area because it requires no interaction with the designer to generate an implementation. Designers can rely on the C/Unix code generator (which rates level A

here) to generate its best implementation without interaction from the designers because of its automatic ADT-implementation selection mechanism.

EC-13: Integration with High-level Design Tool. Both generators are integrated with the same high-level design tool (TimeBench) which possesses design checking and a simulator.

EC-14: Generated Code can be Debugged. Since the new code generator allows the creation of complete implementations, even in the presence of externally-generated code, it presents no impediment to the designer correcting problems at the design level and therefore warrants a rating of level A in this area. On the other hand, the original code generator is extremely likely to be used in a setting where some kind of post-generation code polishing is performed (either manually or by a "home-grown" tool). This increases the likelihood that there would be a temptation to change the implementation rather than the design when trying out solutions during debugging. Consequently, the C/JNX code generator can be rated no higher than level B on this criterion.

EC-15: Support for Portable Implementations. Both generators (each rated level B for this criterion) are capable of producing implementations in standard high-level languages for portable standard operating systems¹ and being re-targeted with reasonable ease by the developer of the tool.

1. Although the C/JNX target (JNX + MS-DOS) is not an official standard, MS-DOS is sufficiently popular that this generator meets the spirit if not the letter of this criterion.

6.1 Research Summary

Although automatic programming systems exist, they are seldom practical in one or more important ways. To overcome this, attention must be paid to the information and techniques required for implementing designs as well as to practical considerations for convenience in the use of such systems. To this end, this work pursued several activities: exploration into various techniques for automatic code generation; the choice of a technique (metaprogramming and procedural transformation) and existing tool (TimeBench/MetaGen) that applies the technique; the examination of shortcomings in the application of the automatic programming techniques; and a programme to address these shortcomings by extending the tool (User Annotations).

Specifically, this work provides a list of seventeen requirements that an automatic programming system should fulfill in order to be practical. To satisfy these requirements, six new features were designed to extend TimeBench so as to improve its practicality for use as an automatic programming system. These features: automated selection of optimal Abstract Data Type implementations; control over tools used in later development phases; user-guided partitioning of generated implementations; inclusion of user-supplied literal source code in the generated implementations; specification of task priorities; and, incorporation of externally-provided object code in generated implementations, were implemented in a code generator targeted for the C language and the Unix operating system.

Additionally, to support the presentation of this work, fifteen criteria were derived for evaluating the features developed in this work, a case study was undertaken and the

evaluation criteria were applied to both the new C/Unix code generator and the previously existing C/JNX code generator. The evaluation reveals progress in all but four of the fifteen evaluation criteria with the most significant improvement occurring in the areas of efficiency and re-use. This suggests some success in addressing issues of practicality with automatic code generating systems, but also confirms that this is a difficult problem with no single or trivial solution. The outcome of the evaluation is reproduced here in Table 11 for the reader's convenience with the right-most column showing the number of levels of improvement achieved.

TABLE 11. Degree of Improvement by Evaluation Criterion

Id	Brief Criterion Name	Original	New	Improvement (levels)
EC-1	Correctness	C	C	0
EC-2	Efficiency	D	A	+3
EC-3	Readability	C	B	+1
EC-4	Ease of Interface with Existing Code	D	B	+2
EC-5	User Control of Static Structure	D	B	+2
EC-6	Native Target Features	D	B	+2
EC-7	Re-use Support use.	D	A	+3
EC-8	Directives	C	B	+1
EC-9	Re-targetability	B	B	0
EC-10	Partial/Incremental generation	D	C	+1
EC-11	Implementation-language neutrality	D	B	+2
EC-12	Autonomy of Generator	C	A	+2
EC-13	Integration with HLD Tool	A	A	0
EC-14	Debug Support	B	A	+1
EC-15	Portability Support	B	B	0

Through the development of the requirements, features and evaluation criteria for practical automatic code generation, this work demonstrates capabilities that facilitate the practical use of automatic code generation as an implementation technology. It is hoped that this contribution will help to allow at least some of the many benefits of automatic code generation to be realized.

6.2 Future Research

A starting point for determining future work in this area is to note those aspects of the new code generator that failed to achieve the highest possible rating(s) during evaluation in the previous chapter. These are reproduced below in Table 12 for the reader's convenience. While these may be the least capable areas of the new code generator, other

TABLE 12. Some Areas for Improvement in the C/Unix Code Generator

Id	Brief Criterion Name	Original	New
EC-1	Correctness	C	C
EC-3	Readability	C	B
EC-4	Ease of Interface with Existing Code	D	B
EC-5	User Control of Static Structure	D	B
EC-6	Native Target Features	D	B
EC-8	Directives	C	B
EC-9	Re-targetability	B	B
EC-10	Partial/Incremental generation	D	C
EC-11	Implementation-language neutrality	D	B
EC-15	Portability Support	B	B

areas such as support for reuse should not be ignored, as continued improvement in such areas can be expected to yield considerable benefits and result in higher expectations

which will in turn lead to a re-scaling of the evaluation criterion for this aspect of code generator quality.

An area that could extend the practicality of this work is to add annotations to help the code generator in the selection of alternative concurrency mechanisms. Much like the way the Abstract Data Type implementations are selected, code generator analysis could help guide the selection of light-weight processes (or *threads*) for some Actors and full processes for others. Similarly, the communication mechanism could be selected from among shared memory, semaphores and sockets for example where the target technology permits.

The TimeBench/MetaGen platform inherently accommodates some scalability through the nature of the hierarchically composable design notation. Coupling this with the new abilities to "stub-out" some design components and include "foreign" object code, a design can be scaled upwards till further permitting several groups to concentrate on different areas of a design and to combine their generated implementations with those of the others by treating them as foreign objects. Nevertheless, a more rigorous investigation of scalability issues could be a fruitful line of inquiry.

Some of the difficulties encountered in implementing the improvements in this work also suggest areas for future investigation. One such problem, which could be called the *Naming Problem*, relates to the differences found in the scope and linkage of design components versus implementation components. Typical design models involve a graphical lexicon, while most implementation technologies depend upon a textual form of expression. These two different modes are fundamentally different when it comes to component visibility: in the former, components possess identity primarily due to their two-dimensional spatial locations (in addition to their names); and in the latter, a strict linear

view is taken that requires component names to "have been seen" by the point at which a reference is desired. When transforming from a hierarchical higher-dimension name space to a flat lower-dimension one, some technique for avoiding name-space *collisions* is absolutely necessary.

Although trivial automatic code generators could solve this problem with a simple serializing technique, this is an inadequate approach for more sophisticated generators because it is desirable to support partial implementation (generating source code for some region of the design) and to handle smart regeneration of the implementation where only the semantically and structurally significant design changes result in new implementation components being generated. The naming of these components should be repeatable to avoid unnecessary re-compilation and to facilitate re-use. Merely moving the location of a component for aesthetic reasons in a MachineChart should not result in the recreation of its corresponding implementation components, nor necessarily should the cutting and pasting of design components.

Another fruitful possibility for future work is found in choosing the optimum implementation for Abstract Data Types. The current approach for manual performance tuning (Figure 50) is to compile, link and execute the program of interest. The execution is performed with the aid of a *profiler* which is a tool that measures the amount of a resource, in this case time, consumed by particular regions of the program. A step involving manual intervention is required to complete the feed-back loop and improve the implementation's performance. In fact, such a technique is required for the development of both the library of ADT implementations used by this work's code generator and for the cost function that is used to select amongst them.

FIGURE 50. Configuration of Conventional Implementation Performance Tuning

The current implementation of this work performs a *static* analysis of the design to select an implementation; by gathering statistics on the relative frequency of various operations performed on instances of ADTs, a better choice of ADT implementation can be made. This amounts to profiling the implementation in a predictive manner [27, 35] and performing semantic-preserving transformations to the overall system implementation. An exciting prospect for future work is to allow this static analysis to be augmented or even replaced by a *dynamic* analysis provided by the design interpreter. While verifying the execution of the design, the above statistics could be collected to provide an additional picture of the usage of the ADT instances which could contribute to a greater degree of accuracy in the selection of the ADT implementation (Figure 51).

FIGURE 51. Possible Configuration of ADT Implementation Selection based on Predicted Performance

REFERENCES

- [1] Arthur, Lowell Jay. 1992. *Improving software quality: An insider's guide to TQM*. New York, New York: John Wiley and Sons, Inc.
- [2] Balzer, R.M., T.E. Cheatham and C. Green. 1983. "Software technology in the 1990's: using a new paradigm," *Computer*. vol. 16, no. 11, pp 39-45.
- [3] Balzer, R.M. 1985. "A 15 Year perspective on automatic programming," *IEEE transactions on software engineering*. vol SE-11, pp 1257-1267.
- [4] Barstow, David et al. 1982. "Automatic programming," *The handbook of artificial intelligence*. Edited by Avron Barr, Edward A. Feigenbaum. Vol. 2. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc.
- [5] Belina, Ferenc, J.P. Hubaux and Gert Nilsson. 1987. "SDL design tool," *SDL Forum* (3rd: 1987: Hague Netherlands) SDL '87. Edited by R. Saracco and P.A.J. Tilanus. pp 75-81. New York, New York: Elsevier Science Publishing Company, Inc.
- [6] Blum, Bruce I. 1992. *Software engineering: A holistic view*. New York, New York: Oxford University Press.
- [7] Booch, G. 1991. *Object-oriented design*. Redwood City, California: Benjamin/Cummings.
- [8] Buhr, R.J.A., G.M. Karam, C.M. Woodside, R. Casselman, G.Franks, H.J. Scott and D. Bailey. 1991. "TimeBench: A CAD tool for real-time system design," *Proceedings of IEEE/ACM 2nd intl. symposium on environments and tools for Ada (SETA2)*. Washington, D.C.
- [9] Buhr, R.J.A. and R.S. Casselman. 1992. "Architectures with pictures," *OOPSLA '92*. pp 466-83. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc.
- [10] Buhr, R.J.A. 1984. *System design with Ada*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.
- [11] Buhr, R.J.A. 1990. *Practical visual techniques in system design (with applications to Ada)*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.
- [12] Cabral, G., H. E. Dunsmore, S. Stratton, D. M. Zage and W. M. Zage. 1988. "An Evaluation of Teamwork," SERC-TR-16-P, West Lafayette, Indiana, Department of Computer Sciences, Purdue University.
- [13] Card, David N. and Robert L. Glass. 1990. *Measuring software design quality*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.
- [14] Casselman, R., G.M. Karam and R.J.A. Buhr. 1989. *TimeBench: A short description of the MachineCharts textual support language*. SCE-89-25, Department of Systems and Computer Engineering, Ottawa, Ontario, Carleton University.

- [15] Chang, Shi-Kuo. 1990. *Principles of visual programming systems*. Edited by Shi-Kuo Chang. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.
- [16] Cloutier, J.J.B. and R.E. Seivora. 1993. "Automated assistance to the design of embedded real-time software systems using SDL," *SDL Forum* (6th: 1993: Hague Netherlands) SDL '93. Edited by Ove Færgemand and Amadeo Sarma. pp 361-74. Amsterdam: Elsevier Science Publishing B.V.
- [17] Cohen, Donald and Neil Campbell, "Automating relational operations on data structures," *IEEE Software*, vol. 10, no. 3, pp. 53-60, May, 1993.
- [18] Cox, Brad J. 1986. *Object oriented programming - an evolutionary approach*. New York, New York: Addison-Wesley Publishing Company, Inc.
- [19] Diaz-Herrera, Jorge L., "The Importance of static structures in software construction," *IEEE Software*, vol. 10, no. 3, pp. 75-87, May, 1993.
- [20] Dyer, Michael. 1992. *The cleanroom approach to quality software development*. New York, New York: John Wiley and Sons, Inc.
- [21] Englehart, M. 1994. "High quality automatic code generation for control applications," *Proceedings of the IEEE/IFAC joint symposium on computer-aided control system design*. p 363-7.
- [22] Fafchamps, Danielle, "Organizational factors and reuse," *IEEE Software*, vol. 11, no. 5, pp. 31-41, September, 1994.
- [23] Glass, Robert L. 1992. *Building quality software*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.
- [24] Gorman, Joe and Ulrick Johansen. 1991. "Engineering the implementation of SDL specifications," *SDL Forum* (5th: 1991: Hague Netherlands) SDL '91. Edited by Ove Færgemand and Rick Reed. pp 367-78. Amsterdam: Elsevier Science Publishing B.V.
- [25] Ichbiah, J.D., J.G.P.P Barnes, R.J. Firth and M. Woodger. 1986. *Rationale for the design of the Ada programming language*. U.S. Government Ada Joint Program Office.
- [26] Jacobson, I., M. Christerson, P. Jonsson and G. Overgaard. 1992. *Object-oriented software engineering - a use case driven approach*. Don Mills, Ontario: Addison-Wesley Publishing Company, Inc.
- [27] Jain, Raj. 1991. *The Art of computer systems performance analysis: Techniques for experimental design, measurement, simulation and modeling*. New York, New York: John Wiley & Sons.
- [28] Johansen, U., Elab Vefsnmo and Eirik Vefsnmo. 1987. "Automatic program generation of SDL-specifications: Principles and solutions," *SDL Forum* (3rd: 1987: Hague Netherlands) SDL '87. Edited by R. Saracco and P.A.J. Tilanus. pp 349-60. New York, New York: Elsevier Science Publishing Company, Inc.
- [29] Kaiser, Gail E. 1990. *Knowledge engineering*. Edited by Hojjat Adeli. Vol. 2, *Applications*. New York, New York: McGraw-Hill Publishing Company.

- [30] Karam, Gerald M. and Ronald S. Casselman. 1993. "A Cataloging framework for software development methods," *IEEE computer* (Feb). pp 34-46.
- [31] Kariisson, Jan and Lennart Månsson. 1987. "Using SDL as specification and design language and Ada as implementation language," *SDL Forum* (3rd: 1987: Hague Netherlands) SDL '87. Edited by R. Saracco and P.A.J. Tilanus. pp 339-48. New York, New York: Elsevier Science Publishing Company, Inc.
- [32] Kasyanov, V.N. 1990. "Tools and techniques of annotated programming," *3rd International workshop on compiler compilers*, Edited by D. Hammer. pp 117-31. New York, New York: Springer-Verlag.
- [33] Kelly, J.C. 1987. "A Comparison of Four Design Methods for Real-Time Systems," *Proceedings of the Ninth International Conference on Software Engineering*, IEEE CS Press.
- [34] Kingsbury, M.A. 1962. "Operating experience with COBOL in a service bureau," *The Computer Journal*, vol 5.
- [35] Lazowska, Edward D. , John Zahorjan, G. Scott Graham, Kenneth C. Sevcik. 1984. *Quantitative System Performance - Computer System Analysis Using Queueing Network Models*. Englewood Cliffs, New Jersey: Prentice-Hall Inc.
- [36] Moynihan, V.D. and P.J.L. Wallis. 1991. "The Design and Implementation of a high-level language converter," *Software - practice and experience*. vol 21, pp. 391-400.
- [37] Mussar, Gary. Dec. 1992. Applying ObjecTime in Falcon. Ottawa: Bell-Northern Research (presentation slides).
- [38] O'Connor, James, Catharine Mansour, Jerri Turner-Harris and Grady Campbell, "Reuse in command-and-control systems," *IEEE Software*, vol. 11, no. 5, pp. 70-9, September, 1994.
- [39] Oualline, Steve. 1992. *C elements of style: the programmers guide to developing well written C and C++ programs*. San Mateo, California: M&T Publishing, Inc.
- [40] Pinker, Steven. 1994. *The Language instinct*. New York, New York: William Morrow and Company, Inc.
- [41] Prieto-Díaz, Rubén, "Status report: software reusability," *IEEE Software*, vol. 10, no. 3, pp. 61-66, May, 1993.
- [42] Pressman, R.S. 1987. *Software engineering: A Practitioner's approach*, 2nd Ed. Toronto, Ontario: McGraw-Hill, Inc.
- [43] Rich, Charles and Richard C. Waters. 1988. "Automatic programming: Myths and prospects," *Computer*, vol. 15, pp. 40-51.
- [44] _____. 1990. *The Programmer's apprentice*. Reading, Massachusetts.: Addison-Wesley Publishing Company; New York, New York: ACM PRESS.
- [45] Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, William Lorenzen. 1991. *Object-oriented modeling and design*. Englewood Cliffs, New Jersey: Prentice-Hall Inc.

- [46] Sassa, 1990. "Rie and Jun: Towards the generation of all compiler phases." *3rd International workshop on compiler compilers*, Edited by D. Hammer, pp 56-70. New York, New York: Springer-Verlag
- [47] Scott, H.J. and Gerald M. Karam. 1991. *Meta-code generation in a design environment*. SCE-91-25. Dept. of Systems and Computer Engineering, Carleton University.
- [48] Scott, H.J. 1991. *Automatic Code Generation for Real Time Systems*. M.Eng. Thesis, Department of Systems and Computer Engineering, Carleton University.
- [49] Selic, Bran, Garth Gullekson, Paul T. Ward. 1994. *Real-Time Object-Oriented Modeling*. New York, New York: John Wiley and Sons, Inc.
- [50] Setliff, Dorothy, Elaine Kant and Tom Cain. "Practical software synthesis." *IEEE Software*, vol. 10, no. 3, pp. 6-9, May, 1993.
- [51] Smart, J. 1994. "Packaging policies for automatic code generation," *Ada user*, vol. 15, no. 1, pp 34-41.
- [52] Sterling, L. and E. Shapiro. 1987. *The Art of prolog: Advanced programming techniques*. Cambridge, Massachusetts: The MIT Press.
- [53] Swanson, Douglas, René Tio and Gary Mussar. 1992. *Falcon Software Design in ObjecTime*. Ottawa: Bell-Northern Research (working paper).
- [54] Venemans, P.H.A. and R.A. Beukers. 1989. "An Experiment with algebraic reduction," *SDL Forum* (4th: 1989: Hague Netherlands) SDL '89. Edited by Ove Færgemand and Maria Manuela Marques, pp 325-34. Amsterdam: Elsevier Science Publishing B.V.
- [55] Ward, Michael. 1990. *Software that works*. San Diego, California: Academic Press, Inc.
- [56] Wasserman, A.I. "Characteristics of Software Development Methodologies," *Information Systems Design Methodologies Proc*. Elsevier Science Publishing B.V. North Holland, 1983 pp 37-62
- [57] Weide, Bruce W., William F. Ogden and Muralid Sitaraman, "Recasting algorithms to encourage reuse," *IEEE Software*, vol. 11, no. 5, pp. 80-, September, 1994.

```

124         MCLString sduin;
125     ) s_u_parms;
126
127 /* ----- courier1_type Task Declaration ----- */
128
129 void courier1_type();
130 task_parameter_block courier1_parameters = (courier1_type, 1000, 3,
131     20, 1);
132 task_id courier1;
133
134 /* ----- Package Specification : dst----- */
135
136 void d_in(MCLString sdu);
137
138
139 /* ----- Package Body Declarations : dst----- */
140
141 /* ----- d_task_type Task Declaration ----- */
142
143 void d_task_type();
144 task_parameter_block d_task_parameters = (d_task_type, 1000, 3, 20,
145     1);
146 task_id d_task;
147 mailbox_id d_d, d_u;
148 typedef enum (d_d_event, d_u_event) d_task_events;
149 typedef struct (
150     MCLString sdu;
151     ) d_d_parms;
152
153 /* ----- courier5_type Task Declaration ----- */
154
155 void courier5_type();
156 task_parameter_block courier5_parameters = (courier5_type, 1000, 3,
157     20, 1);
158 task_id courier5;
159
160 /* ----- Package Specification : comm2----- */
161
162 void c2_in(MCLString sdu);
163
164 void c2_up(MCLInteger t_u2, MCLInteger s_u2, MCLString d_u2);
165
166 void c2_dn(MCLInteger *formal_t_d2, MCLInteger *formal_s_d2,
167     MCLString *formal_d_d2);
168
169 /* ----- Package Body Declarations : comm2----- */
170 MCLString d_dn2;

```

another rule in the specification or can refer to a *built-in rule* in which case the expression is enclosed in braces. Logical relationships between the expressions in a rule body are constructed using the *and* and *or* operators; *expr1* , *expr2* means "the value of *expr1* and the value of *expr2*"; *expr1* | *expr2* means "the value of *expr1* or the value of *expr2*". An alternative to using the *or* operator is to specify two rules with identical heads and with bodies corresponding to *expr1* and *expr2*. The *or* operator has higher precedence than the *and* operator but precedence can be forced through the use of parentheses; otherwise, rules are evaluated one expression at a time, in source order, with short-circuit semantics. A final note: there is a special rule with the head *root* (*RootNode*) that must be present in a specification and that serves as an entry point for the generated code generator.

MSL Grammar

rule	⇒	head --> body .
fact	⇒	head .
head	⇒	symbol symbol (formal_arg_list)
body	⇒	expression expression_list
expression	⇒	symbol symbol (actual_arg_list) (expression_list) (expression_list) expression op expression
expression_list	⇒	expression expression , expression_list
formal_arg_list	⇒	formal_arg formal_arg , formal_arg_list
actual_arg_list	⇒	actual_arg actual_arg , actual_arg_list
formal_arg	⇒	mode variable variable

actual_arg	⇒	number string variable
op	⇒	' '
mode	⇒	+ -
symbol	⇒	lower_case_alpha lower_case_alpha alpha_numerics
variable	⇒	upper_case_alpha _ upper_case_alpha alpha_numerics _ alpha_numerics
number	⇒	numeral_list + numeral_list - numeral_list
string	⇒	" symbol_list "
alpha_numerics	⇒	lower_case_alpha upper_case_alpha numeral _ lower_case_alpha alpha_numerics upper_case_alpha alpha_numerics _ alpha_numerics
numeral_list	⇒	numeral numeral numeral_list
symbol_list	⇒	lower_case_alpha upper_case_alpha numeral symbol lower_case_alpha symbol_list upper_case_alpha symbol_list numeral symbol_list symbol symbol_list
lower_case_alpha	⇒	a b c d e f g h i j k l m n o p q r s t u v w x y z
upper_case_alpha	⇒	A B C D E F G H I J K L M N O P

		Q R S T U V W X
		Y Z
numeral	⇒	0 1 2 3 4 5 6 7
		8 9
symbol	⇒	! @ # \$ % & *
		() _ ~ + = ' '
		" () [] ; :
		' " , . - > ?

This appendix contains the source code generated by the original TimeBench code generator for the design considered in the case study. This code is monolithic and constitutes a single source file.

FIGURE 52. commdemo.c - Source Generated by CJNX_codegen for Case Study.

```

1  #include <stdio.h>
2  #include <jnx.h>
3  #include <jnx-dos.h>
4  #include <math.h>
5
6  #include "typedefs.h"
7
8  #include "ListLib.h"
9  #include "SetLib.h"
10 #include "StrLib.h"
11 #include "IntLib.h"
12 #include "RealLib.h"
13 #include "BoolLib.h"
14
15 semaphore_id timing_semaphore;
16
17 /* ----- courier3_type Task Declaration ----- */
18
19 void courier3_type();
20 task_parameter_block courier3_parameters = {courier3_type, 1000, 3,
21 20, 1};
21 task_id courier3;
22
23 /* ----- courier4_type Task Declaration ----- */
24
25 void courier4_type();
26 task_parameter_block courier4_parameters = {courier4_type, 1000, 3,
27 20, 1};
28 task_id courier4;
29
30 /* ----- Package Specification : comm1----- */
31
32 void c1_in(MCLString sdu);
33
34 void c1_up(MCLInteger t_u1, MCLInteger s_u1, MCLString d_u1);

```

```

34
35 void cl_dn(MCLInteger *formal_t_dl, MCLInteger *formal_s_dl,
MCLString *formal_d_dl);
36
37
38 /* ----- Package Body Declarations : comm1 ----- */
39
40 MCLString d_dn1;
41
42 MCLInteger t_dn1;
43
44 MCLInteger s_dn1;
45
46 MCLString d_up1;
47
48 MCLInteger t_up1;
49
50 MCLInteger s_up1;
51
52 /* ----- ACM Specification : altbit_1----- */
53
54 typedef enum (init, receive_request, send_request, timeout,
data_response) altbit_1_EVENT_TYPE;
55 typedef enum (altbit_1_IMPLICIT_ERROR, initialize, ack_received,
data_received, data_request, receive_response)
altbit_1_ACTION_TYPE;
56 void altbit_1_decode_event(altbit_1_EVENT_TYPE cur_event_var,
altbit_1_ACTION_TYPE *cur_action_var, MCLInteger typ, MCLInteger
rcvd_seq, MCLInteger sent_seq);
57
58 void perform1();
59
60 /* ----- comm1_type Task Declaration ----- */
61
62 void comm1_type();
63 task_parameter_block comm1_parameters = (comm1_type, 1000, 3, 20,
1);
64 task_id comm1;
65 mailbox_id p1_cl, p2_cl, s2_cl, s1_cl, tol;
66 typedef enum (p1_cl_event, p2_cl_event, s2_cl_event, s1_cl_event,
tol_event) comm1_events;
67 typedef struct {
68     MCLInteger s_dl;
69     MCLInteger t_dl;
70     MCLString d_dl;
71 } p1_cl_parms;
72 typedef struct {
73     MCLInteger t_ul;
74     MCLInteger s_ul;
75     MCLString d_ul;
76 } p2_cl_parms;

```



```

77 typedef struct {
78     MCLString sdulup;
79     } s2_cl_parms;
80 typedef struct {
81     MCLString sdu;
82     } s1_cl_parms;
83 MCLBoolean pdudnw;
84 MCLBoolean pduupw;
85 MCLBoolean sduupw;
86 MCLBoolean sdudnw;
87
88 void controll();
89
90 /* ----- timer1_type Task Declaration ----- */
91
92 void timer1_type();
93 task_parameter_block timer1_parameters = (timer1_type, 1000, 3, 20,
94     1);
95 task_id timer1;
96 mailbox_id onl, offl;
97 flag_id recvtimer1_timer;
98 typedef enum (onl_event, offl_event, recvtimer1_event)
99     timer1_events;
100
101 /* ----- courier2_type Task Declaration ----- */
102 void courier2_type();
103 task_parameter_block courier2_parameters = (courier2_type, 1000, 3,
104     20, 1);
105 task_id courier2;
106
107 /* ----- Package Specification : src----- */
108 void s_in(MCLString sdu);
109
110 /* ----- Package Body Declarations : src----- */
111
112 /* ----- s_task_type Task Declaration ----- */
113 void s_task_type();
114 task_parameter_block s_task_parameters = (s_task_type, 1000, 3, 20,
115     1);
116 task_id s_task;
117 mailbox_id s_d, s_u;
118 typedef enum (s_d_event, s_u_event) s_task_events;
119 typedef struct {
120     MCLString sdu;
121     } s_d_parms;
122 typedef struct {

```

```

124         MCLString  sduin;
125     ) s_u_parms;
126
127 /* ----- courier1_type Task Declaration ----- */
128
129 void courier1_type();
130 task_parameter_block courier1_parameters = {courier1_type, 1000, 3,
131 20, 1};
132 task_id courier1;
133
134 /* ----- Package Specification : dst----- */
135
136 void d_in(MCLString  sdu);
137
138
139 /* ----- Package Body Declarations : dst----- */
140
141 /* ----- d_task_type Task Declaration ----- */
142
143 void d_task_type();
144 task_parameter_block d_task_parameters = {d_task_type, 1000, 3, 20,
145 1};
146 task_id d_task;
147 mailbox_id d_d, d_u;
148 typedef enum {d_d_event, d_u_event} d_task_events;
149 typedef struct {
150     MCLString  sdu;
151     ) d_d_parms;
152
153 /* ----- courier5_type Task Declaration ----- */
154
155 void courier5_type();
156 task_parameter_block courier5_parameters = {courier5_type, 1000, 3,
157 20, 1};
158 task_id courier5;
159
160 /* ----- Package Specification : comm2----- */
161
162 void c2_in(MCLString  sdu);
163
164 void c2_up(MCLInteger  t_u2, MCLInteger  s_u2, MCLString  d_u2);
165
166 void c2_dn(MCLInteger  *formal_t_d2, MCLInteger  *formal_s_d2,
167 MCLString  *formal_d_d2);
168
169 /* ----- Package Body Declarations : comm2----- */
170 MCLString  d_dn2;

```

```

171
172 MCLInteger t_dn2;
173
174 MCLInteger s_dn2;
175
176 MCLString d_up2;
177
178 MCLInteger t_up2;
179
180 MCLInteger s_up2;
181
182 /* ----- ACM Specification : altbit_2----- */
183
184 typedef enum (init, receive_request, send_request, timeout,
data_response) altbit_2_EVENT_TYPE;
185 typedef enum (altbit_2_IMPLICIT_ERROR, initialize, ack_received,
data_received, data_request, receive_response)
altbit_2_ACTION_TYPE;
186 void altbit_2_decode_event(altbit_2_EVENT_TYPE cur_event_var,
altbit_2_ACTION_TYPE *cur_action_var, MCLInteger typ, MCLInteger
rcvd_seq, MCLInteger sent_seq);
187
188 void perform2();
189
190 /* ----- comm2_type Task Declaration ----- */
191
192 void comm2_type();
193 task_parameter_block comm2_parameters = (comm2_type, 1000, 3, 20,
1);
194 task_id comm2;
195 mailbox_id p1_c2, p2_c2, s2_c2, s1_c2, to2;
196 typedef enum (p1_c2_event, p2_c2_event, s2_c2_event, s1_c2_event,
to2_event) comm2_events;
197 typedef struct (
198     MCLInteger s_d2;
199     MCLInteger t_d2;
200     MCLString d_d2;
201 ) p1_c2_parms;
202 typedef struct (
203     MCLInteger t_u2;
204     MCLInteger s_u2;
205     MCLString d_u2;
206 ) p2_c2_parms;
207 typedef struct (
208     MCLString sdulup;
209 ) s2_c2_parms;
210 typedef struct (
211     MCLString sdu;
212 ) s1_c2_parms;
213 MCLBoolean pdudnw;
214 MCLBoolean pduupw;

```

```

215 MCLBoolean sduupw;
216 MCLBoolean sdudnw;
217
218 void control2();
219
220 /* ----- timer2_type Task Declaration ----- */
221
222 void timer2_type();
223 task_parameter_block timer2_parameters = (timer2_type, 1000, 3, 20,
224 1);
225 task_id timer2;
226 mailbox_id on2, off2;
227 flag_id recvtimer2_timer;
228 typedef enum (on2_event, off2_event, recvtimer2_event)
229 timer2_events;
230
231 /* ----- courier6_type Task Declaration ----- */
232
233 void courier6_type();
234 task_parameter_block courier6_parameters = (courier6_type, 1000, 3,
235 20, 1);
236 task_id courier6;
237
238 /* ----- courier3_type Task Body ----- */
239
240 void courier3_type()
241 {
242     MCLInteger typ;
243     MCLInteger seq;
244     MCLString data;
245     for ( , ; ) {
246         c2_dn(&typ, &seq, &data);
247         c1_up(typ, seq, data);
248     }
249 }
250
251 /* ----- courier4_type Task Body ----- */
252
253 void courier4_type()
254 {
255     MCLInteger typ;
256     MCLInteger seq;
257     MCLString data;
258     MCLInteger count;

```

```

766     if (altbit_2_cur_state_var == init) {
767         if ((cur_event_var == init)) {
768             altbit_2_cur_state_var = estab;
769             *cur_action_var = initialize;
770             return;
771         }
772     } else if (altbit_2_cur_state_var == estab) {
773         if ((cur_event_var == send_request)) {
774             altbit_2_cur_state_var = ack_wait;
775             *cur_action_var = data_request;
776             return;
777         }
778         if ((cur_event_var == receive_request)) {
779             altbit_2_cur_state_var = estab;
780             *cur_action_var = receive_response;
781             return;
782         }
783         if ((cur_event_var == data_response)) {
784             if (((typ == 1) && (rcvd_seq == sent_seq))) {
785                 altbit_2_cur_state_var = estab;
786                 *cur_action_var = data_received;
787                 return;
788             }
789         }
790     } else if (altbit_2_cur_state_var == ack_wait) {
791         if ((cur_event_var == receive_request)) {
792             altbit_2_cur_state_var = ack_wait;
793             *cur_action_var = receive_response;
794             return;
795         }
796         if ((cur_event_var == data_response)) {
797             if (((typ == 1) && (rcvd_seq != sent_seq))) {
798                 altbit_2_cur_state_var = ack_wait;
799                 *cur_action_var = data_received;
800                 return;
801             }
802         }
803         if ((cur_event_var == data_response)) {
804             if (((typ == 1) && (rcvd_seq == sent_seq))) {
805                 altbit_2_cur_state_var = estab;
806                 *cur_action_var = ack_received;
807                 return;
808             }
809         }
810         if ((cur_event_var == timeout)) {
811             altbit_2_cur_state_var = ack_wait;
812             *cur_action_var = data_request;
813             return;
814         }
815     }
816     cur_action_var = altbit_2_IMPLICIT_ERROR;

```

```

313 /* ----- ACM Body : altbit_1----- */
314
315 typedef enum (init, estab, ack_wait) altbit_1_STATE_TYPE;
316 altbit_1_STATE_TYPE altbit_1_cur_state_var = init;
317
318 void altbit_1_decode_event(altbit_1_EVENT_TYPE cur_event_var,
    altbit_1_ACTION_TYPE *cur_action_var, MCLInteger typ, MCLInteger
    rcvd_seq, MCLInteger sent_seq)
319 {
320     if (altbit_1_cur_state_var == init) {
321         if ((cur_event_var == init)) {
322             altbit_1_cur_state_var = estab;
323             *cur_action_var = initialize;
324             return;
325         }
326     } else if (altbit_1_cur_state_var == estab) {
327         if ((cur_event_var == send_request)) {
328             altbit_1_cur_state_var = ack_wait;
329             *cur_action_var = data_request;
330             return;
331         }
332         if ((cur_event_var == receive_request)) {
333             altbit_1_cur_state_var = estab;
334             *cur_action_var = receive_response;
335             return;
336         }
337         if ((cur_event_var == data_response)) {
338             if (((typ == 1) && (rcvd_seq == sent_seq))) {
339                 altbit_1_cur_state_var = estab;
340                 *cur_action_var = data_received;
341                 return;
342             }
343         }
344     } else if (altbit_1_cur_state_var == ack_wait) {
345         if ((cur_event_var == receive_request)) {
346             altbit_1_cur_state_var = ack_wait;
347             *cur_action_var = receive_response;
348             return;
349         }
350         if ((cur_event_var == data_response)) {
351             if (((typ == 1) && (rcvd_seq != sent_seq))) {
352                 altbit_1_cur_state_var = ack_wait;
353                 *cur_action_var = data_received;
354                 return;
355             }
356         }
357         if ((cur_event_var == data_response)) {
358             if (((typ == 1) && (rcvd_seq == sent_seq))) {
359                 altbit_1_cur_state_var = estab;
360                 *cur_action_var = ack_received;
361                 return;

```

```

362         )
363     )
364     if ((cur_event_var == timeout)) {
365         altbit_1_cur_state_var = ack_wait;
366         *cur_action_var = data_request;
367         return;
368     )
369 )
370 cur_action_var = altbit_1_IMPLICIT_ERROR;
371 )
372
373
374 void perform1()
375 {
376     altbit_1_decode_event(anEvent, &anAction, pduup_type, pduup_seq,
377     send_seq);
378     if ((anaction == initialize)) {
379         send_seq = 0;
380         rcv_seq = 0;
381     } else if ((anaction == data_request)) {
382         pdudn_data = copy_string(sdudn);
383         pdudn_seq = send_seq;
384         pdudn_type = 0;
385         pdudnw = True;
386         send_mail(onl, NULL);
387     } else if ((anaction == data_received)) {
388         sduup = copy_string(pduup_data);
389         pdudn_type = 1;
390         pdudn_seq = pduup_seq;
391         pdudnw = True;
392         sduupw = True;
393         rcv_seq = (1 + rcv_seq);
394     } else if ((anaction == ack_received)) {
395         send_mail(offl, NULL);
396         send_seq = (1 + send_seq);
397         sdudnw = True;
398         pduupw = True;
399     } else if ((anaction == receive_response)) {
400         pduupw = True;
401     } else {
402         print_string(create_string("comm1: Bogus action >"));
403     } else if ((anaction == /* NULL */
404 )
405
406 /* ----- comm1_type Task Body ----- */
407 void comm1_type()
408 {
409     MCLInteger s_d1;
410
411     MCLInteger t_d1;

```

```

412
413     MCLString  d_dl;
414
415     MCLInteger  t_ul;
416
417     MCLInteger  s_ul;
418
419     MCLString  d_ul;
420
421     MCLString  sdulup;
422
423     MCLString  sdu;
424
425     MCLInteger  send_seq;
426
427     MCLString  pdudn_data;
428
429     MCLInteger  pdudn_type;
430
431     MCLInteger  pdudn_seq;
432
433     MCLInteger  recv_seq;
434
435     MCLInteger  pduup_type;
436
437     MCLInteger  pduup_seq;
438
439     MCLString  pduup_data;
440
441     MCLString  sduup;
442
443     MCLString  sdudn;
444
445     altbit_1_event_type anevent;
446
447     altbit_1_action_type anaction;
448
449     mail_id mail;
450
451     comm1_events event;
452
453     sduupw = False;
454     pdudnw = False;
455     sdudnw = True;
456     pduupw = True;
457     perform1();
458     for ( ; ; ) {
459         if (sdudnw) {
460             unmask_mailbox(s1_c1);
461         } else {
462             mask_mailbox(s1_c1);

```



```

463     )
464     if (sduupw) {
465         unmask_mailbox(s2_cl);
466     } else {
467         mask_mailbox(s2_cl);
468     }
469     if (pduupw) {
470         unmask_mailbox(p2_cl);
471     } else {
472         mask_mailbox(p2_cl);
473     }
474     if (pdudnw) {
475         unmask_mailbox(p1_cl);
476     } else {
477         mask_mailbox(p1_cl);
478     }
479     event = await_any_mail(&mail);
480     switch(event) {
481         case s1_cl_event:
482             sdu = ((s1_cl_parms *) mail)->sdu;
483             sdudnw = False;
484             anevent = send_request;
485             sdudn = copy_string(sdu);
486             perform1();
487             reply(s1_cl);
488             break;
489         case s2_cl_event:
490             sduupw = False;
491             anevent = receive_request;
492             perform1();
493             sdulup = copy_string(sduup);
494             ((s2_cl_parms *) mail)->sdulup = sdulup;
495             reply(s2_cl);
496             break;
497         case tol_event:
498             reply(tol);
499             anevent = timeout;
500             perform1();
501             break;
502         case p2_cl_event:
503             t_ul = ((p2_cl_parms *) mail)->t_ul;
504             s_ul = ((p2_cl_parms *) mail)->s_ul;
505             d_ul = ((p2_cl_parms *) mail)->d_ul;
506             pduupw = False;
507             pduup_type = t_ul;
508             pduup_seq = s_ul;
509             pduup_data = copy_string(d_ul);
510             anevent = data_response;
511             perform1();
512             reply(p2_cl);
513             break;

```

```

514         case pl_ci_event:
515             pdudnw = False;
516             d_dl = copy_string(pdudn_data);
517             t_dl = pdudn_type;
518             s_dl = pdudn_seq;
519             ((pl_ci_parms *) mail)->s_dl = s_dl;
520             ((pl_ci_parms *) mail)->t_dl = t_dl;
521             ((pl_ci_parms *) mail)->d_dl = d_dl;
522             reply(pl_ci);
523             break;
524     )
525 )
526 )
527
528 void controll()
529 {
530     timeout_parms timeout_mail;
531
532     if (onoroff) {
533         onoroff = False;
534         send_mail(timeout, &timeout_mail);
535     } else {
536         /* NULL */
537     }
538 }
539
540 /* ----- timer1_type Task Body ----- */
541
542 void timer1_type()
543 {
544     MCLBoolean onoroff;
545
546     mail_id mail;
547
548     timer1_events event;
549
550     recvtimer1_timer = create_flag(timer1, recvtimer1_event, false);
551     onoroff = False;
552     for ( ; ; ) {
553         signal_flag_after_time(10.000000, recvtimer1_timer);
554         event = await_any_event(&mail);
555         switch(event) {
556         case off1_event:
557             /* NULL */
558             reply(off1);
559             break;
560         case on1_event:
561             /* NULL */
562             reply(on1);
563             break;
564         case recvtimer1_event:

```

```

565     controll();
566     break;
567     )
568     reset_timed_flag(recvtimer1_timer);
569     )
570 )
571
572 /* ----- courier2_type Task Body ----- */
573
574 void courier2_type()
575 {
576     MCLString sdu;
577
578     MCLString pdu;
579
580     s2_cl_parms s2_cl_mail;
581
582     for ( ; ; ) {
583         send_mail(s2_cl, &s2_cl_mail);
584         sdu = s2_cl_mail.sdulup;
585         pdu = copy_string(sdu);
586         s_in(sdu);
587     }
588 )
589
590 void initialize_comm1()
591 {
592     /* NULL */
593
594 /* ----- PACKAGE BODY : src----- */
595
596 void s_in(MCLString sdu)
597 {
598     s_u_parms s_u_mail;
599
600     s_u_mail.sduin = sdu;
601     send_mail(s_u, &s_u_mail);
602 }
603
604 /* ----- s_task_type Task Body ----- */
605
606 void s_task_type()
607 {
608     MCLString sdu;
609
610     MCLString sduin;
611
612     MCLInteger i;
613
614     MCLString data;
615

```

```

616     MCLInteger size;
617
618     mail_id mail;
619
620     s_task_events event;
621
622     s_d_parms s_d_mail;
623
624     i = 1;
625     size = str_len(data);
626     for ( ; ; ) {
627         await_mail(s_d, &mail);
628         sdu = str_element(data, i);
629         ((s_d_parms *) mail)->sdu = sdu;
630         reply(s_d);
631         i = (i % size);
632         i = (i + 1);
633     }
634     print_string(create_string("Src terminates"));
635     print_newline();
636 }
637
638 /* ----- courier1_type Task Body ----- */
639
640 void courier1_type()
641 {
642     MCLString sdu;
643
644     MCLString pdu;
645
646     s_d_parms s_d_mail;
647
648     for ( ; ; ) {
649         send_mail(s_d, &s_d_mail);
650         sdu = s_d_mail.sdu;
651         pdu = copy_string(sdu);
652         cl_in(sdu);
653     }
654 }
655
656 void initialize_src()
657 {
658     /* NULL */
659
660     /* ----- PACKAGE BODY : dst----- */
661
662     void d_in(MCLString sdu)
663     {
664         /* NULL */
665
666         /* ----- d_task_type Task Body ----- */

```

```

667
668 void d_task_type()
669 {
670     MCLString sdu;
671
672     MCLInteger i;
673
674     MCLString data;
675
676     MCLInteger size;
677
678     mail_id mail;
679
680     d_task_events event;
681
682     d_d_parms d_d_mail;
683
684     i = 1;
685     size = str_len(data);
686     for ( ; ; ) {
687         await_mail(d_d, &mail);
688         sdu = str_element(data, i);
689         ((d_d_parms *) mail)->sdu = sdu;
690         reply(d_d);
691         i = (i % size);
692         i = (i + 1);
693     }
694     print_string(create_string("Src terminates"));
695     print_newline();
696 }
697
698 /* ----- courier5_type Task Body ----- */
699
700 void courier5_type()
701 {
702     MCLString sdu;
703
704     MCLString pdu;
705
706     d_d_parms d_d_mail;
707
708     for ( ; ; ) {
709         send_mail(d_d, &d_d_mail);
710         sdu = d_d_mail.sdu;
711         pdu = copy_string(sdu);
712         c2_in(sdu);
713     }
714 }
715
716 void initialize_dst()
717 {

```

```

718  /* NULL */)
719
720  /* ----- PACKAGE BODY : comm2----- */
721
722  void c2_in(MCLString sdu)
723  {
724      sl_c2_parms sl_c2_mail;
725
726      sl_c2_mail.sdu = sdu;
727      send_mail(sl_c2, &sl_c2_mail);
728  }
729
730  void c2_up(MCLInteger t_u2, MCLInteger s_u2, MCLString d_u2)
731  {
732      p2_c2_parms p2_c2_mail;
733
734      p2_c2_mail.t_u2 = t_u2;
735      p2_c2_mail.s_u2 = s_u2;
736      p2_c2_mail.d_u2 = d_u2;
737      send_mail(p2_c2, &p2_c2_mail);
738  }
739
740  void c2_dn(MCLInteger *formal_t_d2, MCLInteger *formal_s_d2,
MCLString *formal_d_d2)
741  {
742      MCLInteger t_d2;
743
744      MCLInteger s_d2;
745
746      MCLString d_d2;
747
748      pl_c2_parms pl_c2_mail;
749
750      send_mail(pl_c2, &pl_c2_mail);
751      d_dn2 = pl_c2_mail.d_d2;
752      t_dn2 = pl_c2_mail.t_d2;
753      s_dn2 = pl_c2_mail.s_d2;
754      *formal_t_d2 = t_d2;
755      *formal_s_d2 = s_d2;
756      *formal_d_d2 = d_d2;
757  }
758
759  /* ----- ACM Body : altbit_2----- */
760
761  typedef enum {init, estab, ack_wait} altbit_2_STATE_TYPE;
762  altbit_2_STATE_TYPE altbit_2_cur_state_var = init;
763
764  void altbit_2_decode_event(altbit_2_EVENT_TYPE cur_event_var,
altbit_2_ACTION_TYPE *cur_action_var, MCLInteger typ, MCLInteger
rcvd_seq, MCLInteger sent_seq)
765  {

```

```

766     if (altbit_2_cur_state_var == init) {
767         if ((cur_event_var == init)) {
768             altbit_2_cur_state_var = estab;
769             *cur_action_var = initialize;
770             return;
771         }
772     } else if (altbit_2_cur_state_var == estab) {
773         if ((cur_event_var == send_request)) {
774             altbit_2_cur_state_var = ack_wait;
775             *cur_action_var = data_request;
776             return;
777         }
778         if ((cur_event_var == receive_request)) {
779             altbit_2_cur_state_var = estab;
780             *cur_action_var = receive_response;
781             return;
782         }
783         if ((cur_event_var == data_response)) {
784             if (((typ == 1) && (rcvd_seq == sent_seq))) {
785                 altbit_2_cur_state_var = estab;
786                 *cur_action_var = data_received;
787                 return;
788             }
789         }
790     } else if (altbit_2_cur_state_var == ack_wait) {
791         if ((cur_event_var == receive_request)) {
792             altbit_2_cur_state_var = ack_wait;
793             *cur_action_var = receive_response;
794             return;
795         }
796         if ((cur_event_var == data_response)) {
797             if (((typ == 1) && (rcvd_seq != sent_seq))) {
798                 altbit_2_cur_state_var = ack_wait;
799                 *cur_action_var = data_received;
800                 return;
801             }
802         }
803         if ((cur_event_var == data_response)) {
804             if (((typ == 1) && (rcvd_seq == sent_seq))) {
805                 altbit_2_cur_state_var = estab;
806                 *cur_action_var = ack_received;
807                 return;
808             }
809         }
810         if ((cur_event_var == timeout)) {
811             altbit_2_cur_state_var = ack_wait;
812             *cur_action_var = data_request;
813             return;
814         }
815     }
816     cur_action_var = altbit_2_IMPLICIT_ERROR;

```

```

817 )
818
819
820 void perform2()
821 (
822     altbit_2_decode_event(anEvent, &anAction, pduup_type, pduup_seq,
823     send_seq);
824     if ((anaction == initialize)) {
825         send_seq = 0;
826         recv_seq = 0;
827     } else if ((anaction == data_request)) {
828         pdudn_data = copy_string(sdudn);
829         pdudn_seq = send_seq;
830         pdudn_type = 0;
831         pdudnw = True;
832         send_mail(on2, NULL);
833     } else if ((anaction == data_received)) {
834         sduup = copy_string(pduup_data);
835         pdudn_type = 1;
836         pdudn_seq = pduup_seq;
837         pdudnw = True;
838         sduupw = True;
839         recv_seq = (1 + recv_seq);
840     } else if ((anaction == ack_received)) {
841         send_mail(off2, NULL);
842         send_seq = (1 + send_seq);
843         sdudnw = True;
844         pduupw = True;
845     } else if ((anaction == receive_response)) {
846         pduupw = True;
847     } else {
848         print_string(create_string("comm1: Bogus action >"));
849     } else if ((anaction == /* NULL */)
850 )
851 /* ----- comm2_type Task Body ----- */
852
853 void comm2_type()
854 (
855     MCLInteger s_d2;
856
857     MCLInteger t_d2;
858
859     MCLString d_d2;
860
861     MCLInteger t_u2;
862
863     MCLInteger s_u2;
864
865     MCLString d_u2;
866

```



```
867 MCLString sdulup;
868
869 MCLString sdu;
870
871 MCLInteger send_seq;
872
873 MCLString pdudn_data;
874
875 MCLInteger pdudn_type;
876
877 MCLInteger pdudn_seq;
878
879 MCLInteger recv_seq;
880
881 MCLInteger pduup_type;
882
883 MCLInteger pduup_seq;
884
885 MCLString pduup_data;
886
887 MCLString sduup;
888
889 MCLString sdudn;
890
891 altbit_2_event_type anevent;
892
893 altbit_2_action_type anaction;
894
895 mail_id mail;
896
897 comm2_events event;
898
899 sduupw = False;
900 pdudnw = False;
901 sdudnw = True;
902 pduupw = True;
903 perform2();
904 for ( ; ; ) {
905     if (sduupw) {
906         unmask_mailbox(s2_c2);
907     } else {
908         mask_mailbox(s2_c2);
909     }
910     if (sdudnw) {
911         unmask_mailbox(s1_c2);
912     } else {
913         mask_mailbox(s1_c2);
914     }
915     if (pduupw) {
916         unmask_mailbox(p2_c2);
917     } else {
```

```

918 mask_mailbox(p2_c2);
919     )
920     if (pdudnw) {
921 unmask_mailbox(p1_c2);
922     } else {
923 mask_mailbox(p1_c2);
924     }
925     event = await_any_mail(&mail);
926     switch(event) {
927 case to2_event:
928     reply(to2);
929     anevent = timeout;
930     perform2();
931     break;
932 case s2_c2_event:
933     sduupw = False;
934     anevent = receive_request;
935     perform2();
936     sdulup = copy_string(sduup);
937     ((s2_c2_parms *) mail)->sdulup = sdulup;
938     reply(s2_c2);
939     break;
940 case s1_c2_event:
941     sdu = ((s1_c2_parms *) mail)->sdu;
942     sdudnw = False;
943     anevent = send_request;
944     sdudn = copy_string(sdu);
945     perform2();
946     reply(s1_c2);
947     break;
948 case p2_c2_event:
949     t_u2 = ((p2_c2_parms *) mail)->t_u2;
950     s_u2 = ((p2_c2_parms *) mail)->s_u2;
951     d_u2 = ((p2_c2_parms *) mail)->d_u2;
952     pduupw = False;
953     pduup_type = t_u2;
954     pduup_seq = s_u2;
955     pduup_data = copy_string(d_u2);
956     anevent = data_response;
957     perform2();
958     reply(p2_c2);
959     break;
960 case p1_c2_event:
961     pdudnw = False;
962     d_d2 = copy_string(pdudn_data);
963     t_d2 = pdudn_type;
964     s_d2 = pdudn_seq;
965     ((p1_c2_parms *) mail)->s_d2 = s_d2;
966     ((p1_c2_parms *) mail)->t_d2 = t_d2;
967     ((p1_c2_parms *) mail)->d_d2 = d_d2;
968     reply(p1_c2);

```

```

969     break;
970     )
971 )
972 )
973
974 void control2()
975 {
976     timeout_parms timeout_mail;
977
978     if (onoroff) {
979         onoroff = False;
980         send_mail(timeout, &timeout_mail);
981     } else {
982         /* NULL */
983     }
984 }
985
986 /* ----- timer2_type Task Body ----- */
987
988 void timer2_type()
989 {
990     MCLBoolean onoroff;
991
992     mail_id mail;
993
994     timer2_events event;
995
996     recvtimer2_timer = create_flag(timer2, recvtimer2_event, false);
997     onoroff = False;
998     for ( ; ; ) {
999         signal_flag_after_time(10.000000, recvtimer2_timer);
1000         event = await_any_event(&mail);
1001         switch(event) {
1002         case on2_event:
1003             /* NULL */
1004             reply(on2);
1005             break;
1006         case off2_event:
1007             /* NULL */
1008             reply(off2);
1009             break;
1010         case recvtimer2_event:
1011             control2();
1012             break;
1013         }
1014         reset_timed_flag(recvtimer2_timer);
1015     }
1016 }
1017
1018 /* ----- courier6_type Task Body ----- */
1019

```

```

1020 void courier6_type()
1021 (
1022     MCLString sdu;
1023
1024     MCLString pdu;
1025
1026     s2_c2_parms s2_c2_mail;
1027
1028     for ( ; ; ) (
1029         send_mail(s2_c2, &s2_c2_mail);
1030         sdu = s2_c2_mail.sdulup;
1031         pdu = copy_string(sdu);
1032         d_in(sdu);
1033     )
1034 )
1035
1036 void initialize_comm2()
1037 (
1038     /* NULL */
1039
1040 void main()
1041 (
1042     char *keystroke;
1043
1044     initialize_multitasking(true, 50, 10, 10000, 0, 0, 1);
1045     initialize_DOS_package();
1046     timing_semaphore = create_non_counting_semaphore(false);
1047     courier3 = create_task(&courier3_parameters);
1048     courier4 = create_task(&courier4_parameters);
1049     initialize_comm1();
1050     comm1 = create_task(&comm1_parameters);
1051     p1_c1 = create_mailbox(comm1, p1_c1_event, false);
1052     p2_c1 = create_mailbox(comm1, p2_c1_event, false);
1053     s2_c1 = create_mailbox(comm1, s2_c1_event, false);
1054     s1_c1 = create_mailbox(comm1, s1_c1_event, false);
1055     tol = create_mailbox(comm1, tol_event, false);
1056     timer1 = create_task(&timer1_parameters);
1057     on1 = create_mailbox(timer1, on1_event, false);
1058     off1 = create_mailbox(timer1, off1_event, false);
1059     courier2 = create_task(&courier2_parameters);
1060     initialize_src();
1061     s_task = create_task(&s_task_parameters);
1062     s_d = create_mailbox(s_task, s_d_event, false);
1063     s_u = create_mailbox(s_task, s_u_event, false);
1064     courier1 = create_task(&courier1_parameters);
1065     initialize_dst();
1066     d_task = create_task(&d_task_parameters);
1067     d_d = create_mailbox(d_task, d_d_event, false);
1068     d_u = create_mailbox(d_task, d_u_event, false);
1069     courier5 = create_task(&courier5_parameters);
1070     initialize_comm2();

```

```
1071     comm2 = create_task(&comm2_parameters);
1072     p1_c2 = create_mailbox(comm2, p1_c2_event, false);
1073     p2_c2 = create_mailbox(comm2, p2_c2_event, false);
1074     s2_c2 = create_mailbox(comm2, s2_c2_event, false);
1075     sl_c2 = create_mailbox(comm2, sl_c2_event, false);
1076     to2 = create_mailbox(comm2, to2_event, false);
1077     timer2 = create_task(&timer2_parameters);
1078     on2 = create_mailbox(timer2, on2_event, false);
1079     off2 = create_mailbox(timer2, off2_event, false);
1080     courier6 = create_task(&courier6_parameters);
1081     schedule_tasks();
1082     for( ; ;) {
1083         if ( Pgetch(keystroke)) break;
1084     }
1085     terminate_DOS_package();
1086     terminate_multitasking();
1087 }
```

This appendix contains the majority of the code generated by the improved TimeBench code generator for the design considered in the case study. Certain source codes are omitted on the grounds that they differ from those included here by only the names of the variables and functions. Source codes generated for the *src* box, *comm1* box, *courier3* actor and their respective internals are included here along with the project's main routine (*commdemo_main*) and the project's *makefile*.

FIGURE 53. *commdemo_main.c* - Source Generated by CUnix_codegen for Case Study.

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/socket.h>
4  #include <math.h>
5
6  #include "typedefs.h"
7
8  #include "ipclib.h"
9
10 #include "ListLib.h"
11 #include "SetLib.h"
12 #include "StrLib.h"
13 #include "IntLib.h"
14 #include "RealLib.h"
15 #include "BoolLib.h"
16
17 #include "courier3.h"
18 #include "courier4.h"
19 #include "comm1.h"
20 #include "source.h"
21 #include "dest.h"
22 #include "comm2.h"
23
24 void main()
25 {
26     initialize_comm1();
27     initialize_src();
28     initialize_dst();
```

```

29     initialize_comm2();
30     createProcess(comm1_type, 20);
31     createProcess(timer1_type, 20);
32     createProcess(courier2_type, 20);
33     createProcess(s_task_type, 20);
34     createProcess(courier1_type, 20);
35     createProcess(d_task_type, 20);
36     createProcess(courier5_type, 20);
37     createProcess(comm2_type, 20);
38     createProcess(timer2_type, 20);
39     createProcess(courier6_type, 20);
40     createProcess(courier3_type, 20);
41     createProcess(courier4_type, 20);
42 }

```

FIGURE 54. comm1 - Source Generated by CJNX_codegen for Case Study.

Interface: (comm1.h)

```

1  /* ----Begin ACM Specification : altbit_1--- */
2
3  typedef enum
4  (
5      altbit_1_init_event,
6      altbit_1_receive_request_event,
7      altbit_1_send_request_event,
8      altbit_1_timeout_event,
9      altbit_1_data_response_event
10 ) altbit_1_event_type;
11
12 typedef enum
13 (
14     altbit_1_IMPLICIT_ERROR, altbit_1_initialize_action,
15     altbit_1_ack_received_action,
16     altbit_1_data_received_action,
17     altbit_1_data_request_action,
18     altbit_1_receive_response_action
19 ) altbit_1_action_type;
20
21 typedef enum
22 (
23     altbit_1_init_state,
24     altbit_1_estab_state,
25     altbit_1_ack_wait_state
26 ) altbit_1_state_type;
27
28 void altbit_1_decode_event(altbit_1_event_type cur_event_var,
29 altbit_1_action_type *cur_action_var, MCLInteger typ, MCLInteger
30 rcvd_seq, MCLInteger sent_seq);
31
32 /* ----End ACM Specification : altbit_1--- */

```

```

31 MCLInteger send_seq;
32 MCLString pdudn_data;
33 MCLInteger pdudn_type;
34 MCLInteger pdudn_seq;
35 MCLInteger recv_seq;
36 MCLInteger pduup_type;
37 MCLInteger pduup_seq;
38 MCLString pduup_data;
39 MCLString sduup;
40 MCLString sdudn;
41 altbit_1_event_type anevent;
42 altbit_1_action_type anaction;
43 MCLBoolean onoroff;
44 MCLString sdu;
45 MCLString pdu;
46 MCLString d_dnl;
47 MCLInteger t_dnl;
48 MCLInteger s_dnl;
49 MCLString d_upl;
50 MCLInteger t_upl;
51 MCLInteger s_upl;
52 void perform1_Button();
53
54 void controll1_Button();
55
56 void cl_in_Button(MCLString sdu);
57
58 void cl_up_Button(MCLInteger t_ul, MCLInteger s_ul, MCLString d_ul);
59
60 void cl_dn_Button(MCLInteger *formal_t_dl, MCLInteger *formal_s_dl,
MCLString *formal_d_dl);
61
62 /* ---Begin comm1_type Task Declaration --- */
63
64 extern void comm1_type();
65 extern task_id comm1;
66
67 /* Mailbox declaration list */
68
69 #define p1_cl_gN*p1_cl_gN"
70 #define p2_cl_gN*p2_cl_gN"
71 #define s2_cl_gN*s2_cl_gN"
72 #define s1_cl_gN*s1_cl_gN"
73 #define tol_gN*tol_gN"
74
75 /* Flag declaration list */
76
77 /* Mailbox Event declaration list */
78 typedef enum {
79     comm1_p1_cl_event,
80     comm1_p2_cl_event,

```



```

81  comm1_s2_cl_event,
82  comm1_s1_cl_event,
83  comm1_tol_event
84  /* Flag Event declaration list */
85
86  ) comm1_events;
87
88  /* Mailbox Parameter declarations */
89  typedef struct {
90      MCLInteger s_d1;
91      MCLInteger t_d1;
92      MCLString d_d1;
93  } p1_cl_parms_type;
94
95  typedef struct {
96      MCLInteger t_u1;
97      MCLInteger s_u1;
98      MCLString d_u1;
99  } p2_cl_parms_type;
100
101  typedef struct {
102      MCLString sdulup;
103  } s2_cl_parms_type;
104
105  typedef struct {
106      MCLString sdu;
107  } s1_cl_parms_type;
108
109  typedef struct {
110  } tol_parms_type;
111
112  /* Flag Parameter declarations */
113  typedef struct {
114  } recvcomm1_parms_type;
115
116  MCLBoolean pdudnw;
117  MCLBoolean pduupw;
118  MCLBoolean sduupw;
119  MCLBoolean sdudnw;
120
121  /* --- End comm1_type Task Declaration --- */
122
123  /* ---Begin timer1_type Task Declaration --- */
124
125  extern void timer1_type();
126  extern task_id timer1;
127
128  /* Mailbox declaration list */
129
130  #define on1_gN"on1_gN"
131  #define off1_gN"off1_gN"

```

```

132
133 /* Flag declaration list */
134 #define recvtimer1_flagNode"recvtimer1_flagNode"
135 #define recvtimer1_flagReset"recvtimer1_flagReset"
136
137 /* Mailbox Event declaration list */
138 typedef enum (
139     timer1_onl_event,
140     timer1_offl_event
141 /* Flag Event declaration list */
142 ,
143     timer1_recvtimer1_event
144 ) timer1_events;
145
146 /* Mailbox Parameter declarations */
147 typedef struct (
148     ) onl_parms_type;
149
150 typedef struct (
151     ) offl_parms_type;
152
153 /* Flag Parameter declarations */
154 typedef struct (
155     ) recvtimer1_parms_type;
156
157
158 /* --- End timer1_type Task Declaration --- */
159
160 /* ---Begin courier2_type Task Declaration --- */
161
162 extern void courier2_type();
163 extern task_id courier2;
164
165 /* --- End courier2_type Task Declaration --- */
166
167 /* --- Begin Package Specification: comm1--- */
168
169 /* --- Pkg Spec 1 --- */
170
171 /* --- Begin Package Body Declarations : comm1--- */
172
173 /* --- End Package Specification: comm1--- */
174
175
176

```

Implementation: (comm1.c)

```

1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/socket.h>

```

```

4  #include <math.h>
5
6  #include "typedefs.h"
7
8  #include "ipclib.h"
9
10 #include "ListLib.h"
11 #include "SetLib.h"
12 #include "StrLib.h"
13 #include "IntLib.h"
14 #include "RealLib.h"
15 #include "BoolLib.h"
16
17 #include "source.h"
18 #include "comm1.h"
19
20 /* --- Begin ACM Body : altbit_1--- */
21
22 altbit_1_state_type altbit_1_cur_state_var = altbit_1_init_state;
23
24 void altbit_1_decode_event(altbit_1_event_type cur_event_var,
25 altbit_1_action_type *cur_action_var, MCLInteger typ, MCLInteger
26 rcvd_seq, MCLInteger sent_seq)
27 {
28     if (altbit_1_cur_state_var == altbit_1_init_state) {
29         if ((cur_event_var == altbit_1_init_event)) {
30             altbit_1_cur_state_var = altbit_1_estab_state;
31             *cur_action_var = altbit_1_initialize_action;
32             return;
33         }
34     } else if (altbit_1_cur_state_var == altbit_1_estab_state) {
35         if ((cur_event_var == altbit_1_send_request_event)) {
36             altbit_1_cur_state_var = altbit_1_ack_wait_state;
37             *cur_action_var = altbit_1_data_request_action;
38             return;
39         }
40         if ((cur_event_var == altbit_1_receive_request_event)) {
41             altbit_1_cur_state_var = altbit_1_estab_state;
42             *cur_action_var = altbit_1_receive_response_action;
43             return;
44         }
45         if ((cur_event_var == altbit_1_data_response_event)) {
46             if (((typ == 1) && (rcvd_seq == sent_seq))) {
47                 altbit_1_cur_state_var = altbit_1_estab_state;
48                 *cur_action_var = altbit_1_data_received_action;
49                 return;
50             }
51         }
52     } else if (altbit_1_cur_state_var == altbit_1_ack_wait_state) {
53         if ((cur_event_var == altbit_1_receive_request_event)) {
54             altbit_1_cur_state_var = altbit_1_ack_wait_state;

```

```

53         *cur_action_var = altbit_1_receive_response_action;
54         return;
55     )
56     if ((cur_event_var == altbit_1_data_response_event)) (
57         if (((typ == 1) && (rcvd_seq != sent_seq))) (
58             altbit_1_cur_state_var = altbit_1_ack_wait_state;
59             *cur_action_var = altbit_1_data_received_action;
60             return;
61         )
62     )
63     if ((cur_event_var == altbit_1_data_response_event)) (
64         if (((typ == 1) && (rcvd_seq == sent_seq))) (
65             altbit_1_cur_state_var = altbit_1_estab_state;
66             *cur_action_var = altbit_1_ack_received_action;
67             return;
68         )
69     )
70     if ((cur_event_var == altbit_1_timeout_event)) (
71         altbit_1_cur_state_var = altbit_1_ack_wait_state;
72         *cur_action_var = altbit_1_data_request_action;
73         return;
74     )
75 )
76 cur_action_var = altbit_1_IMPLICIT_ERROR;
77 )
78
79
80 /* --- End ACM Body : altbit_1--- */
81
82
83 /* --- Begin Button Body:performl --- */
84
85 void performl_Button()
86 {
87     altbit_1_decode_event(anevent, &anaction, pduup_type, pduup_seq,
88     send_seq);
89     /* -- print "Comml: initialized.", newline; */
90     /* -- enable NDATA_request for data. */
91     /* -- print "comml: data request.", newline; */
92     /* -- enable NDATA_request for ack. */
93     /* -- enable upgoing data. */
94     /* -- print "comml: data received.", newline; */
95     /* -- allow more data. */
96     /* -- allow more acks. */
97     /* -- print "comml: ack received", newline; */
98     /* -- print "comml: receive response", newline; */
99     if ((anaction == altbit_1_initialize_action)) (
100         send_seq = 0;
101         rcv_seq = 0;
102     ) else if ((anaction == altbit_1_data_request_action)) (
103         pduup_data = MCLcopy_string(sdudn);

```

```

103     pdudn_seq = send_seq;
104     pdudn_type = 0;
105     pdudnw = True;
106     ipcPush(onl_gN, NULL, 0);
107 ) else if ((anaction == altbit_1_data_received_action)) (
108     sduup = MCLcopy_string(pduup_data);
109     pdudn_type = 1;
110     pdudn_seq = pduup_seq;
111     pdudnw = True;
112     sduupw = True;
113     recv_seq = (1 + recv_seq);
114 ) else if ((anaction == altbit_1_ack_received_action)) (
115     ipcPush(offl_gN, NULL, 0);
116     send_seq = (1 + send_seq);
117     sdudnw = True;
118     pduupw = True;
119 ) else if ((anaction == altbit_1_receive_response_action)) (
120     pduupw = True;
121 ) else (
122     MCLprint_string(create_string("comml: Bogus action >"));
123 )
124 )
125
126 /* --- End Button Body:perform1 --- */
127
128
129 /* --- Begin Button Body:control1 --- */
130
131 void control1_Button()
132 {
133     /* -- print "comml control called..." */
134     /* -- print "Timeout taken!"; */
135     /* -- else */
136     /* -- print "Timeout ignored!"; */
137     if (oncroff) {
138         oncroff = False;
139         ipcJab(tol_gN, NULL, 0);
140     }
141     /* -- print newline; */
142 }
143
144 /* --- End Button Body:control1 --- */
145
146
147 /* --- Begin Button Body:cl_in --- */
148
149 void cl_in_Button(MCLString sdu)
150 {
151     sl_cl_parms_type sl_cl_parms;
152
153     sl_cl_parms.sdu = sdu;

```

```

154     ipcPush(s1_cl_gN, (void *)&s1_cl_parms, sizeof(s1_cl_parms));
155 }
156
157 /* --- End Button Body:cl_in --- */
158
159
160 /* --- Begin Button Body:cl_up --- */
161
162 void cl_up_Button(MCLInteger t_ul, MCLInteger s_ul, MCLString
d_ul)
163 {
164     p2_cl_parms_type p2_cl_parms;
165
166     p2_cl_parms.t_ul = t_ul;
167     p2_cl_parms.s_ul = s_ul;
168     p2_cl_parms.d_ul = d_ul;
169     ipcPush(p2_cl_gN, (void *)&p2_cl_parms, sizeof(p2_cl_parms));
170 }
171
172 /* --- End Button Body:cl_up --- */
173
174
175 /* --- Begin Button Body:cl_dn --- */
176
177 void cl_dn_Button(MCLInteger *formal_t_d1, MCLInteger
*formal_s_d1, MCLString *formal_d_d1)
178 {
179     MCLInteger t_d1;
180
181     MCLInteger s_d1;
182
183     MCLString d_d1;
184
185     p1_cl_parms_type p1_cl_parms;
186
187     ipcPush(p1_cl_gN, (void *)&p1_cl_parms, sizeof(p1_cl_parms));
188     d_d1 = p1_cl_parms.d_d1;
189     t_d1 = p1_cl_parms.t_d1;
190     s_d1 = p1_cl_parms.s_d1;
191     *formal_t_d1 = t_d1;
192     *formal_s_d1 = s_d1;
193     *formal_d_d1 = d_d1;
194 }
195
196 /* --- End Button Body:cl_dn --- */
197
198
199 /* --- Begin comm1_type Task Body --- */
200
201 void comm1_type()
202 {

```

```

203  /*-- local_gate_object_list --*/
204  /*-- gate_formal_parm_declaration_list --*/
205  MCLInteger  s_dl;
206  MCLInteger  t_dl;
207  MCLString  d_dl;
208  MCLInteger  t_ul;
209  MCLInteger  s_ul;
210  MCLString  d_ul;
211  MCLString  sdulup;
212  MCLString  sdu;
213  /*-- object_declaration_list --*/
214  MCLInteger send_seq;
215  MCLString  pdudn_data;
216  MCLInteger pdudn_type;
217  MCLInteger pdudn_seq;
218  MCLInteger recv_seq;
219  MCLInteger pduup_type;
220  MCLInteger pduup_seq;
221  MCLString  pduup_data;
222  MCLString  sduup;
223  MCLString  sdudn;
224  altbit_1_event_type anevent;
225  altbit_1_action_type anaction;
226  int         event;
227
228  MCLBoolean timeExpired;
229
230
231  /*-- local_declarations --*/
232
233  /*-- init_gates --*/
234  int p1_cl_gate_caller, p1_cl_gate = ipcPullInit("p1_cl_gN");
235  p1_cl_parms_type p1_cl_parms;
236  int p2_cl_gate_caller, p2_cl_gate = ipcPullInit("p2_cl_gN");
237  p2_cl_parms_type p2_cl_parms;
238  int s2_cl_gate_caller, s2_cl_gate = ipcPullInit("s2_cl_gN");
239  s2_cl_parms_type s2_cl_parms;
240  int s1_cl_gate_caller, s1_cl_gate = ipcPullInit("s1_cl_gN");
241  s1_cl_parms_type s1_cl_parms;
242  int tol_gate_caller, tol_gate = ipcPullInit("tol_gN");
243  tol_parms_type tol_parms;
244
245  /*-- task_call_declarations --*/
246
247  /*-- declare_frames --*/
248  fd_set recvcomml_frame;
249  int     recvcomml_flagGate;
250
251  /*-- init_frames --*/
252  FD_ZERO(&recvcomml_frame);
253  FD_SET(tol_gate, &recvcomml_frame);

```

```

254
255     /*-- agenda_code --*/
256     sduupw = False;
257     pdudnw = False;
258     sdudnw = True;
259     pduupw = True;
260     performl_Button();
261     for ( ; ; ) {
262         if (sdudnw) {
263             FD_SET(s1_cl_gate, &recvcomml_frame);
264         } else {
265             FD_CLR(s1_cl_gate, &recvcomml_frame);
266         }
267         if (sduupw) {
268             FD_SET(s2_cl_gate, &recvcomml_frame);
269         } else {
270             FD_CLR(s2_cl_gate, &recvcomml_frame);
271         }
272         if (pduupw) {
273             FD_SET(p2_cl_gate, &recvcomml_frame);
274         } else {
275             FD_CLR(p2_cl_gate, &recvcomml_frame);
276         }
277         if (pdudnw) {
278             FD_SET(p1_cl_gate, &recvcomml_frame);
279         } else {
280             FD_CLR(p1_cl_gate, &recvcomml_frame);
281         }
282         event = ipcSelect(&recvcomml_frame, (float) 0.0);
283         for (; event ; ) {
284             if (FD_ISSET(s1_cl_gate, &recvcomml_frame)) {
285                 s1_cl_gate_caller = ipcPull(s1_cl_gate, (void
286 *)&s1_cl_parms, sizeof(s1_cl_parms_type));
287                 sdu = s1_cl_parms.sdu;
288                 sdudnw = False;
289                 anevent = altbit_l_send_request_event;
290                 sdudn = MCLcopy_string(sdu);
291                 performl_Button();
292                 ipcEndCrit(s1_cl_gate_caller, (void *)&s1_cl_parms,
293 sizeof(s1_cl_parms_type));
294                 event--;
295                 FD_CLR(s1_cl_gate, &recvcomml_frame);
296             }
297             if (FD_ISSET(s2_cl_gate, &recvcomml_frame)) {
298                 s2_cl_gate_caller = ipcPull(s2_cl_gate, (void
299 *)&s2_cl_parms, sizeof(s2_cl_parms_type));
300                 sduupw = False;
301                 anevent = altbit_l_receive_request_event;
302                 performl_Button();
303                 sdulup = MCLcopy_string(sduup);
304                 s2_cl_parms.sdulup = sdulup;

```



```

302         ipcEndCrit(s2_cl_gate_caller, (void *)&s2_cl_parms,
sizeof(s2_cl_parms_type));
303         event--;
304         FD_CLR(s2_cl_gate, &recvcomm1_frame);
305     }
306     if (FD_ISSET(tol_gate, &recvcomm1_frame)) {
307         tol_gate_caller = ipcPull(tol_gate, (void *)&tol_parms,
sizeof(tol_parms_type));
308         ipcEndCrit(tol_gate_caller, (void *)&tol_parms,
sizeof(tol_parms_type));
309         anevent = altbit_1_timeout_event;
310         perform1_Button();
311         event--;
312         FD_CLR(tol_gate, &recvcomm1_frame);
313     }
314     if (FD_ISSET(p2_cl_gate, &recvcomm1_frame)) {
315         p2_cl_gate_caller = ipcPull(p2_cl_gate, (void
*)&p2_cl_parms, sizeof(p2_cl_parms_type));
316         t_ul = p2_cl_parms.t_ul;
317         s_ul = p2_cl_parms.s_ul;
318         d_ul = p2_cl_parms.d_ul;
319         pduupw = False;
320         pduup_type = t_ul;
321         pduup_seq = s_ul;
322         pduup_data = MCLcopy_string(d_ul);
323         anevent = altbit_1_data_response_event;
324         perform1_Button();
325         ipcEndCrit(p2_cl_gate_caller, (void *)&p2_cl_parms,
sizeof(p2_cl_parms_type));
326         event--;
327         FD_CLR(p2_cl_gate, &recvcomm1_frame);
328     }
329     if (FD_ISSET(p1_cl_gate, &recvcomm1_frame)) {
330         p1_cl_gate_caller = ipcPull(p1_cl_gate, (void
*)&p1_cl_parms, sizeof(p1_cl_parms_type));
331         pdudnw = False;
332         d_d1 = MCLcopy_string(pdudn_data);
333         t_d1 = pdudn_type;
334         s_d1 = pdudn_seq;
335         p1_cl_parms.s_d1 = s_d1;
336         p1_cl_parms.t_d1 = t_d1;
337         p1_cl_parms.d_d1 = d_d1;
338         ipcEndCrit(p1_cl_gate_caller, (void *)&p1_cl_parms,
sizeof(p1_cl_parms_type));
339         event--;
340         FD_CLR(p1_cl_gate, &recvcomm1_frame);
341     }
342 }
343 )
344 )
345

```

```

346 /* --- End comm1_type Task Body --- */
347
348
349 /* --- Begin timer1_type Task Body --- */
350
351 void timer1_type()
352 {
353     /*-- local_gate_object_list --*/
354     /*-- gate_formal_parm_declaration_list --*/
355     /*-- object_declaration_list --*/
356     MCLBoolean onoroff;
357     int         event;
358
359     MCLBoolean timeExpired;
360
361
362     /*-- local_declarations --*/
363
364     /*-- init_gates --*/
365     int onl_gate_caller, onl_gate = ipcPullInit("onl_gN");
366     onl_parms_type onl_parms;
367     int offl_gate_caller, offl_gate = ipcPullInit("offl_gN");
368     offl_parms_type offl_parms;
369
370     /*-- task_call_declarations --*/
371
372     /*-- declare_frames --*/
373     fd_set recvtimer1_frame;
374     int     recvtimer1_flagGate;
375
376     /*-- init_frames --*/
377     FD_ZERO(&recvtimer1_frame);
378     FD_SET(offl_gate, &recvtimer1_frame);
379     FD_SET(onl_gate, &recvtimer1_frame);
380
381     /*-- agenda_code --*/
382     onoroff = False;
383     for ( ; ; ) {
384         timeExpired = ((event = ipcSelect(&recvtimer1_frame, (float)
10.000000)) == 0);
385         for ( ; event ; ) {
386             if (FD_ISSET(offl_gate, &recvtimer1_frame)) {
387                 offl_gate_caller = ipcPull(offl_gate, (void
*)&offl_parms, sizeof(offl_parms_type));
388                 /* NULL */
389                 ipcEndCrit(offl_gate_caller, (void *)&offl_parms,
sizeof(offl_parms_type));
390                 event--;
391                 FD_CLR(offl_gate, &recvtimer1_frame);
392             }
393             if (FD_ISSET(onl_gate, &recvtimer1_frame)) {

```

```

394         onl_gate_caller = ipcPull(onl_gate, (void *)&onl_parms,
sizeof(onl_parms_type));
395         /* NULL */
396         ipcEndCrit(onl_gate_caller, (void *)&onl_parms,
sizeof(onl_parms_type));
397         event--;
398         FD_CLR(onl_gate, &recvtimer1_frame);
399     }
400 }
401     if (timeExpired) {
402         timeExpired = False;
403         controll_Button();
404     }
405 }
406 }
407
408 /* --- End timer1_type Task Body --- */
409
410
411 /* --- Begin courier2_type Task Body --- */
412
413 void courier2_type()
414 {
415     /*-- local_gate_object_list --*/
416     /*-- gate_formal_parm_declaration_list --*/
417     /*-- object_declaration_list --*/
418     MCLString sdu;
419     MCLString pdu;
420     int event;
421
422     MCLBoolean timeExpired;
423
424
425     /*-- local_declarations --*/
426
427     /*-- init_gates --*/
428
429     /*-- task_call_declarations --*/
430     s2_c1_parms_type s2_c1_parms;
431
432
433     /*-- declare_frames --*/
434
435     /*-- init_frames --*/
436
437     /*-- agenda_code --*/
438     for ( ; ; ) {
439         ipcPush(s2_c1_gN, (void *)&s2_c1_parms, sizeof(s2_c1_parms));
440         sdu = s2_c1_parms.sdulup;
441         pdu = MCLcopy_string(sdu);
442         s_in_Button(sdu);

```

```

443     }
444 )
445
446 /* --- End courier2_type Task Body --- */
447
448 /* --- Begin PACKAGE BODY : comm1--- */
449
450 void initialize_comm1()
451 {
452     /* -- Box comm1 */
453 }
454 /* --- Pkg Body 1 --- */

```

FIGURE 55. source - Source Generated by CUnix_codegen for Case Study.

Interface: (source.h)

```

1  MCLInteger i;
2  imp4_String data;
3  MCLInteger size;
4  MCLString sdu;
5  MCLString pdu;
6  void next_Button();
7
8  void s_in_Button(MCLString sdu);
9
10 /* ---Begin s_task_type Task Declaration --- */
11
12 extern void s_task_type();
13 extern task_id s_task;
14
15 /* Mailbox declaration list */
16
17 #define s_d_gN"s_d_gN"
18 #define s_u_gN"s_u_gN"
19
20 /* Flag declaration list */
21
22 /* Mailbox Event declaration list */
23 typedef enum {
24     s_task_s_d_event,
25     s_task_s_u_event
26     /* Flag Event declaration list */
27
28     ) s_task_events;
29
30 /* Mailbox Parameter declarations */
31 typedef struct {
32     MCLString sdu;
33     ) s_d_parms_type;
34

```

```

35 typedef struct {
36     MCLString sduin;
37 } s_u_parms_type;
38
39 /* Flag Parameter declarations */
40 typedef struct {
41     ) s_dn_parms_type;
42
43
44 /* --- End s_task_type Task Declaration --- */
45
46 /* ---Begin courier1_type Task Declaration --- */
47
48 extern void courier1_type();
49 extern task_id courier1;
50
51 /* --- End courier1_type Task Declaration --- */
52
53 /* --- Begin Package Specification: src--- */
54
55 /* --- Pkg Spec 1 --- */
56
57 /* --- Begin Package Body Declarations : src--- */
58
59 /* --- End Package Specification: src--- */
60
61

```

Implementation: (source.c)

```

1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/socket.h>
4  #include <math.h>
5
6  #include "typedefs.h"
7
8  #include "ipclib.h"
9
10 #include "ListLib.h"
11 #include "SetLib.h"
12 #include "StrLib.h"
13 #include "IntLib.h"
14 #include "RealLib.h"
15 #include "BoolLib.h"
16
17 #include "comm1.h"
18 #include "source.h"
19
20 /* --- Begin Button Body:next --- */
21

```

```

22 void next_Button()
23 {
24     /* -- Push Button next */
25     {
26         /*
27          * This is the "generate_next.c" source file. This source
28          * will be included in the implementation generated for
29          * the button "next".
30          */
31
32         printf("next_Button called\n");
33     }
34 }
35 /* --- End Button Body:next --- */
36
37
38 /* --- Begin Button Body:s_in --- */
39
40 void s_in_Button(MCLString sdu)
41 {
42     s_u_parms_type s_u_parms;
43
44     /* -- Push Button s_in */
45     s_u_parms.sduin = sdu;
46     ipcPush(s_u_gN, (void *)&s_u_parms, sizeof(s_u_parms));
47 }
48
49 /* --- End Button Body:s_in --- */
50
51
52 /* --- Begin s_task_type Task Body --- */
53
54 void s_task_type()
55 {
56     /*-- local_gate_object_list --*/
57     /*-- gate_formal_parm_declaration_list --*/
58     MCLString sdu;
59     MCLString sduin;
60     /*-- object_declaration_list --*/
61     MCLInteger i;
62     imp4_String data;
63     MCLInteger size;
64     int event;
65
66     MCLBoolean timeExpired;
67
68
69     /*-- local_declarations --*/
70
71     /*-- init_gates --*/
72     int s_d_gate_caller, s_d_gate = ipcPullInit("s_d_gN");

```

```

73     s_d_parms_type s_d_parms;
74     int s_u_gate_caller, s_u_gate = ipcPullInit("s_u_gN");
75     s_u_parms_type s_u_parms;
76
77     /*-- task_call_declarations --*/
78
79     /*-- declare_frames --*/
80     fd_set s_dn_frame;
81     int     s_dn_flagGate;
82
83     /*-- init_frames --*/
84     FD_ZERO(&s_dn_frame);
85
86     /*-- agenda_code --*/
87     i = 1;
88     size = imp4_str_len(data);
89     for ( ; ; ) {
90         s_d_gate_caller = ipcPull(s_d_gate, (void *)&s_d_parms,
sizeof(s_d_parms_type));
91         sdu = imp4_str_element(data, i);
92         s_d_parms.sdu = sdu;
93         ipcEndCrit(s_d_gate_caller, (void *)&s_d_parms,
sizeof(s_d_parms_type));
94         next_Button();
95         i = (i % size);
96         i = (i + 1);
97     }
98 }
99
100 /* --- End s_task_type Task Body --- */
101
102
103 /* --- Begin courier1_type Task Body --- */
104
105 void courier1_type()
106 {
107     /*-- local_gate_object_list --*/
108     /*-- gate_formal_parm_declaration_list --*/
109     /*-- object_declaration_list --*/
110     MCLString sdu;
111     MCLString pdu;
112     int       event;
113
114     MCLBoolean timeExpired;
115
116
117     /*-- local_declarations --*/
118
119     /*-- init_gates --*/
120
121     /*-- task_call_declarations --*/

```

```

122     s_d_parms_type s_d_parms;
123
124
125     /*-- declare_frames --*/
126
127     /*-- init_frames --*/
128
129     /*-- agenda_code --*/
130     for ( ; ; ) {
131         ipcPush(s_d_gN, (void *)&s_d_parms, sizeof(s_d_parms));
132         sdu = s_d_parms.sdu;
133         pdu = MCLcopy_string(sdu);
134         cl_in_Button(sdu);
135     }
136 }
137
138 /* --- End courier1_type Task Body --- */
139
140 /* --- Begin PACKAGE BODY : src--- */
141
142 void initialize_src()
143 {
144     /* -- Box src */
145 }
146 /* --- Pkg Body 1 --- */

```

FIGURE 56. dest - Source Generated by CUnix_codegen for Case Study.

Interface: (dest.h)

```

1  MCLInteger i;
2  imp2_String data;
3  MCLInteger size;
4  MCLString sdu;
5  MCLString pdu;
6  void sink_Button(MCLInteger index);
7
8  void d_in_Button(MCLString sdu);
9
10 /* ---Begin d_task_type Task Declaration --- */
11
12 extern void d_task_type();
13 extern task_id d_task;
14
15 /* Mailbox declaration list */
16
17 #define d_d_gN*d_d_gN"
18 #define d_u_gN*d_u_gN"
19
20 /* Flag declaration list */
21

```



```

22  /* Mailbox Event declaration list */
23  typedef enum {
24      d_task_d_d_event,
25      d_task_d_u_event
26      /* Flag Event declaration list */
27
28      } d_task_events;
29
30  /* Mailbox Parameter declarations */
31  typedef struct {
32      MCLString sdu;
33      } d_d_parms_type;
34
35  typedef struct {
36      } d_u_parms_type;
37
38  /* Flag Parameter declarations */
39  typedef struct {
40      } d_dn_parms_type;
41
42
43  /* --- End d_task_type Task Declaration --- */
44
45  /* ---Begin courier5_type Task Declaration --- */
46
47  extern void courier5_type();
48  extern task_id courier5;
49
50  /* --- End courier5_type Task Declaration --- */
51
52  /* --- Begin Package Specification: dst--- */
53
54  /* --- Pkg Spec 1 --- */
55
56  /* --- Begin Package Body Declarations : dst--- */
57
58  /* --- End Package Specification: dst--- */
59
60

```

Implementation: (dest.c)

```

1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/socket.h>
4  #include <math.h>
5
6  #include "typedefs.h"
7
8  #include "ipclib.h"
9

```

```

10  #include "ListLib.h"
11  #include "SetLib.h"
12  #include "StrLib.h"
13  #include "IntLib.h"
14  #include "RealLib.h"
15  #include "BoolLib.h"
16
17  #include "comm2.h"
18  #include "dest.h"
19
20  /** stub: absorb **/
21
22
23  /* --- Begin Button Body:d_in --- */
24
25  void d_in_Button(MCLString sdu)
26  {
27      /* -- Push Button d_in */
28  }
29
30  /* --- End Button Body:d_in --- */
31
32
33  /* --- Begin d_task_type Task Body --- */
34
35  void d_task_type()
36  {
37      /*-- local_gate_object_list --*/
38      /*-- gate_formal_parm_declaration_list --*/
39      MCLString sdu;
40      /*-- object_declaration_list --*/
41      MCLInteger i;
42      imp2_String data;
43      MCLInteger size;
44      int event;
45
46      MCLBoolean timeExpired;
47
48
49      /*-- local_declarations --*/
50
51      /*-- init_gates --*/
52      int d_d_gate_caller, d_d_gate = ipcPullInit("d_d_gN");
53      d_d_parms_type d_d_parms;
54      int d_u_gate_caller, d_u_gate = ipcPullInit("d_u_gN");
55      d_u_parms_type d_u_parms;
56
57      /*-- task_call_declarations --*/
58
59      /*-- declare_frames --*/
60      fd_set d_dn_frame;

```

```

61     int      d_dn_flagGate;
62
63     /* -- init_frames -- */
64     FD_ZERO(&d_dn_frame);
65
66     /*-- agenda_code --*/
67     i = 1;
68     size = imp2_str_len(data);
69     for ( ; ; ) {
70         d_d_gate_caller = ipcPull(d_d_gate, (void *)&d_d_parms,
sizeof(d_d_parms_type));
71         imp2_modify_str_element(data, i, sdu);
72         d_d_parms.sdu = sdu;
73         ipcEndCrit(d_d_gate_caller, (void *)&d_d_parms,
sizeof(d_d_parms_type));
74         sink_Button(i);
75         i = (i % size);
76         i = (i + 1);
77     }
78 }
79
80 /* --- End d_task_type Task Body --- */
81
82
83 /* --- Begin courier5_type Task Body --- */
84
85 void courier5_type()
86 {
87     /*-- local_gate_object_list --*/
88     /*-- gate_formal_parm_declaration_list --*/
89     /*-- object_declaration_list --*/
90     MCLString sdu;
91     MCLString pdu;
92     int      event;
93
94     MCLBoolean timeExpired;
95
96
97     /*-- local_declarations --*/
98
99     /*-- init_gates --*/
100
101     /*-- task_call_declarations --*/
102     d_d_parms_type d_d_parms;
103
104
105     /*-- declare_frames --*/
106
107     /*-- init_frames --*/
108
109     /*-- agenda_code --*/

```

```

110     for ( ; ; ) {
111         ipcPush(d_d_gN, (void *)&d_d_parms, sizeof(d_d_parms));
112         sdu = d_d_parms.sdu;
113         pdu = MCLcopy_string(sdu);
114         c2_in_Button(sdu);
115     }
116 }
117
118 /* --- End courier5_type Task Body --- */
119
120 /* --- Begin PACKAGE BODY : dst--- */
121
122 void initialize_dst()
123 {
124     /* -- Box src */
125 }
126 /* --- Pkg Body 1 --- */
127
128

```

FIGURE 57. courier3 - Source Generated by CUnix_codegen for Case Study.

Interface: (courier3.h)

```

1  MCLInteger typ;
2  MCLInteger seq;
3  MCLString data;
4
5  /* ---Begin courier3_type Task Declaration --- */
6
7  extern void courier3_type();
8  extern task_id courier3;
9
10 /* --- End courier3_type Task Declaration --- */

```

Implementation: (courier3.c)

```

1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/socket.h>
4  #include <math.h>
5
6  #include "typedefs.h"
7
8  #include "ipclib.h"
9
10 #include "ListLib.h"
11 #include "SetLib.h"
12 #include "StrLib.h"
13 #include "IntLib.h"

```

```

14  #include "RealLib.h"
15  #include "BoolLib.h"
16
17  #include "comm1.h"
18  #include "comm2.h"
19  #include "courier3.h"
20
21  /* --- Begin courier3_type Task Body --- */
22
23  void courier3_type()
24  {
25      /*-- local_gate_object_list --*/
26      /*-- gate_formal_parm_declaration_list --*/
27      /*-- object_declaration_list --*/
28      MCLInteger typ;
29      MCLInteger seq;
30      MCLString  data;
31      int        event;
32
33      MCLBoolean timeExpired;
34
35
36      /*-- local_declarations --*/
37
38      /*-- init_gates --*/
39
40      /*-- task_call_declarations --*/
41
42      /*-- declare_frames --*/
43
44      /*-- init_frames --*/
45
46      /*-- agenda_code --*/
47      for ( ; ; ) {
48          c2_dn_Button(&typ, &seq, &data);
49          c1_up_Button(typ, seq, data);
50      }
51  }
52
53  /* --- End courier3_type Task Body --- */

```

FIGURE 50. Makefile - Source Generated by CUnix_codegen for Case Study.

```

1  include Makefile.preamble
2
3  INCLUDES = ../MetaTranslator/CLibrary
4  LIBS      = BoolLib.o IntLib.o ListLib.o RealLib.o SetLib.o StrLib.o
5  CC        = cc $(CCOPT)
6  OBJECTS   = courier3.o courier4.o comm1.o source.o dest.o absorb.o
              comm2.o
7

```

```
8  all:  commdemo
9
10
11  commdemo:commdemo_main.c courier3.c courier3.o courier4.c
    courier4.o comm1.c comm1.o source.c source.o dest.c dest.o comm2.c
    comm2.o
12      $(CC) -I$(INCLUDES) -o commdemo $(OBJECTS) $(LIBS)
    commdemo_main.c ipclib.o
13
14  courier3.o:courier3.c courier3.h comm1.h comm2.h
15      $(CC) -I$(INCLUDES) -c courier3.c
16
17
18  courier4.o:courier4.c courier4.h comm1.h comm2.h
19      $(CC) -I$(INCLUDES) -c courier4.c
20
21
22  comm1.o:comm1.c comm1.h source.h
23      $(CC) -I$(INCLUDES) -c comm1.c
24
25
26  source.o:source.c source.h comm1.h
27      $(CC) -I$(INCLUDES) -c source.c
28
29
30  dest.o:dest.c dest.h comm2.h
31      $(CC) -I$(INCLUDES) -c dest.c
32
33
34  comm2.o:comm2.c comm2.h dest.h
35      $(CC) -I$(INCLUDES) -c comm2.c
36
37  include Makefile.postamble
38
39
40
```

This appendix contains the source code for and a brief description of a simple library of subroutines that implement inter-process communication (IPC) primitives with the semantics of MachineCharts on top of facilities found in the Unix operating system.

This library provides routines that perform the *Push*, *Jab*, *Push with Time Out* operations as well as the *Pull* and *Pull with Time Out* operations. As an implementation constraint, the engines that invoke either *ipcPull* or *ipcPullWTO* need to first invoke *ipcPullInit* in order to initialize their communication port. Most of the routines use the convention of taking an *address* which is expressed as an alphanumeric string, a pointer to void which serves to point to a struct of parameters and an integer *parmsize* that indicates the size in bytes of the parameter struct.

When an engine needs to activate a choice frame, it first invokes *ipcSelect* which determines which of the eligible gates is ready for rendezvous. Once *ipcSelect* returns, the engine can invoke *ipcPull* with a specific gate addressed. This slight departure from the MachineChart semantics is required for implementation on Unix.

FIGURE 59. ipc.h - Interface for the Inter-Process Communication Library

```
1      /*
2      *  ipclib.h
3      */
4
5      typedef int task_id;
6
7      extern int ipcPush      (char    *addr,
8                             void    *parms,
9                             int     parmsize);
10
```

```

11  extern int ipcJab      (char  *addr,
12                          void  *parms,
13                          int    parmsize);
14
15  extern int ipcPushWTO (char  *addr,
16                          void  *parms,
17                          int    parmsize,
18                          float  timeout);
19
20  extern int ipcPullInit(char  *finger);
21
22  extern int ipcPull     (int    finger,
23                          char  *parms,
24                          int    parmsize);
25
26  extern int ipcPullWTO (int    finger,
27                          char  *parms,
28                          int    parmsize,
29                          float  timeout);
30
31  extern int ipcEndCrit (int    finger,
32                          char  *parms,
33                          int    parmsize);
34
35  extern int ipcSelect (fd_set      *set,
36                       float       timeout);
37
38  extern int createProcess(void Root(),int NiceVal);
39
40  extern void signal_flag_after_time(float timeout,
41      char  *flagToSignal,
42      char  *flagForReset);
43
44  extern void reset_timed_flag(char *flagForReset);

```

FIGURE 60. ipclib.c - Body of the InterProcess Communication Library

```

1  /*
2  *  ipclib.c - InterProcess Communication library.
3  *
4  *  maps machine chart rendezvous semantics onto sockets.
5  *
6  *  Copyright 1994, Andrew Forrest
7  *
8  */
9
10 #include <stdio.h>
11 #include <sys/types.h>

```



```

12  #include <sys/socket.h>
13  #include <sys/errno.h>
14  #include <sys/time.h>
15  #include <sys/resource.h>
16  #include "ipclib.h"
17
18  /*
19   * forward declarations:
20   */
21
22  static int _push (char          *addr,
23                   void          *parms,
24                   int            parmsize,
25                   struct timeval *blocktime);
26
27  static int _pull (int          finger,
28                   char          *parms,
29                   int            parmsize,
30                   struct timeval *blocktime);
31
32  /*
33   * externals:
34   */
35
36  extern int errno;
37
38  /*
39   * constants
40   */
41
42  #define INTERVAL 20000L /* 20 ms (expressed in usec) */
43
44  /*
45   * ipcPush - primitive for blocking "push"
46   */
47
48  int ipcPush (char*   addr,
49              void*   parms,
50              int      parmsize)
51  {
52  #ifdef DEBUG
53      printf("ipcPush: (%s)\n", addr);
54  #endif
55      return _push(addr, parms, parmsize, NULL);
56  }
57
58  /*
59   * ipcJab - primitive for non-blocking "push"
60   */
61
62  int ipcJab (char*   addr,

```

```

63         void*    parms,
64         int       parmsize)
65     {
66         struct timeval tv = { 0L, 0L };
67
68         #ifdef DEBUG
69             printf("ipcJab: (%s)\n", addr);
70         #endif
71         return _push(addr, parms, parmsize, &tv);
72     }
73
74     /*
75     * ipcPushWTO - primitive for "push with t/o"
76     */
77
78     int ipcPushWTO (char*    addr,
79                    void*    parms,
80                    int       parmsize,
81                    float     timeout)
82     {
83         struct timeval tv;
84
85         tv.tv_sec = (long int) timeout;
86         tv.tv_usec = (long int) ((timeout - tv.tv_sec) * 1000000L);
87
88         #ifdef DEBUG
89             printf("ipcPushWTO: (%s) (%f)\n", addr, timeout);
90         #endif
91         return _push(addr, parms, parmsize, &tv);
92     }
93
94
95     /*
96     * _push - internal implementation of "push", "jab" and
97     * "push with time-out"
98     */
99
100    static int _push (addr, parms, parmsize, blocktime)
101        char*    addr;
102        void*    parms;
103        int       parmsize;
104        struct timeval* blocktime; /* NULL = 4ever, 0 = jab */
105    {
106        struct sockaddrst_listener;
107        int          skt, oks, okr, rcv, flags = 0, ret = 0;
108
109        if ((skt = socket (PF_UNIX, SOCK_STREAM, 0)) < -1) {
110            perror("push-socket:");
111            exit (errno);
112        }
113    }

```

```

114     skt_listener.sa_family = AF_UNIX;
115     strcpy (skt_listener.sa_data, addr);
116
117     #ifdef DEBUG
118     printf("_push: attempting connect on skt %2d\n", skt);
119     #endif
120
121     do (
122         if ((rcv = connect(skt, &skt_listener, sizeof(skt_listener)))
123 == -1) (
124         switch (errno) (
125             case ECONNREFUSED :      /* too many others there already */
126             case ENOENT :           /* got here first */
127                 if (blocktime != NULL) (
128                     if (blocktime->tv_sec <= 0 && blocktime->tv_usec <= 0) (
129                         fprintf (stderr, "[%5d] push timed out\n", getpid ());
130                         ret = -1;
131                         goto leave;
132                     ) else (
133                         usleep (INTERVAL);      /* sleep INTERVAL microseconds */
134
135                         blocktime->tv_usec -= INTERVAL;
136                         while (blocktime->tv_usec < 0 && blocktime->tv_sec > 0) (
137                             blocktime->tv_sec -= 1;
138                             blocktime->tv_usec += (1000000L);
139                         )
140                     )
141                 )
142                 break;
143             default:
144                 perror("push-connect:");
145                 exit (errno);
146             )
147         )
148         ) while (rcv == -1);
149
150     #ifdef DEBUG
151     printf("_push: attempting send on skt %2d\n", skt);
152     #endif
153
154     if ((oks = send(skt, (char *)parms, parmsize, flags)) == -1) (
155         perror("push-send:");
156         exit (errno);
157     )
158
159     #ifdef DEBUG
160     printf("_push: attempting recv on skt %2d\n", skt);
161     #endif
162
163     if ((okr = recv (skt, (char *)parms, parmsize, flags)) == -1) (

```

```

164     perror("push-recv:");
165     exit (errno);
166 )
167
168 leave:
169     close (skt);
170     errno = 0;
171     return ret;
172 }
173
174 struct sockaddr skt_listener;
175
176 /*
177  * ipcPullInit - initializes a "pull-finger" for subsequent use.
178  */
179
180 int ipcPullInit(char *finger)
181 {
182     int skt = socket (PF_UNIX, SOCK_STREAM, 0);
183     int okb, okl;
184
185     skt_listener.sa_family = AF_UNIX;
186     strcpy (skt_listener.sa_data, finger);
187     okb = bind (skt, &skt_listener, sizeof (skt_listener));
188     okl = listen (skt, 1);
189 #ifdef DEBUG
190     printf("ipcPullInit: \"%s\" s-%2d, b%-2d, l%-2d, err-%2d\n",
191           finger, skt, okb, okl, errno);
192 #endif
193     if (errno) perror ("ipcPullInit");
194     return skt;
195 }
196
197 /*
198  * ipcPull
199  */
200
201 int ipcPull (int          finger,
202             char          *parms,
203             int           parmsize)
204 {
205     return _pull(finger, parms, parmsize, NULL);
206 }
207
208 /*
209  * ipcPullWTO
210  */
211
212 int ipcPullWTO (int      finger,
213               char      *parms,
214               int      parmsize,

```

```

215             float    timeout)
216  {
217      struct timeval tv =
218          { (long int) timeout,
219            (long int) ((long int)(timeout * 1000000L) % 1000000L)
220          };
221
222      return _pull(finger, parms, parmsize, &tv);
223  }
224
225  /*
226   * _pull - internal implementation of "pull", "pullWTO" and
227   * non-blocking pull
228   */
229
230  int _pull(finger, parms, parmsize, blocktime)
231      int          finger;
232      char*        parms;
233      int          parmsize;
234      struct timeval* blocktime;
235  {
236      struct sockaddr    skt_caller;
237      int skt_caller_len = sizeof(skt_caller);
238      int reply_socket;
239      int r1, r2;
240      fd_set sockets;
241
242      if (blocktime != NULL) {
243          FD_ZERO(&sockets);
244          FD_SET (finger, &sockets);
245      #ifdef DEBUG
246          printf("_pull: attempting select on skt %2d\n", finger);
247      #endif
248          select (FD_SETSIZE, &sockets, NULL, NULL, blocktime);
249          if (!FD_ISSET(finger, &sockets)) {
250              fprintf(stderr, "[%5d] pull timed out\n", getpid());
251              reply_socket = -1;
252              goto leave;
253          }
254      }
255
256      #ifdef DEBUG
257          printf("_pull: attempting accept on skt %2d\n", finger);
258      #endif
259
260      if ((reply_socket = accept(finger, &skt_caller, &skt_call-
261      er_len)) == -1) {
262          perror("pull-accept:");
263          exit (errno);
264      }

```

```

265  #ifdef DEBUG
266  printf("_pull: attempting recv on skt %2d\n", reply_socket);
267  #endif
268
269      if ((r1 = recv (reply_socket, parms, parmsize, 0)) == -1) {
270          perror("pull-recv:");
271          exit (errno);
272      }
273
274      leave:
275      return reply_socket;
276  }
277
278  /*
279   * ipcEndCrit - ends the rendezvous
280   */
281
282  int ipcEndCrit(int    reply_socket,
283                 char *parms,
284                 int    parmsize)
285  {
286      #ifdef DEBUG
287      printf("ipcEndCrit: attempting send on skt %2d\n", reply_socket);
288      #endif
289      if (send (reply_socket, parms, parmsize, 0) == -1) {
290          perror("ipcEndCritical-send:");
291          exit (errno);
292      }
293
294      close (reply_socket);
295  }
296
297  /*
298   * ipcSelect - await writes to a set of file descriptors
299   */
300
301  int ipcSelect(set, timeout)
302      fd_set      *set;
303      float       timeout;
304  {
305      long width;
306      struct timeval tv;
307
308      tv.tv_sec = (long int) timeout;
309      tv.tv_usec = (((long int)timeout) * 1000000L) % 1000000L;
310
311      /* wait for a write to any of a set of file descriptors */
312
313      width = FD_SETSIZE; /* get the width of the fd table */
314
315      #ifdef DEBUG

```

```

316 printf("ipcSelect: attempting select (%d) (%ld . %ld)\n", width,
317         tv.tv_sec, tv.tv_usec);
318 #endif
319
320     if (timeout == 0) {
321         return select((int)width, set, NULL, NULL, NULL);
322     } else {
323         return select((int)width, set, NULL, NULL, &tv);
324     }
325 }
326
327 /*
328  * createProcess - forks off a process.
329  */
330
331 int createProcess (Root, NiceVal)
332     void (*Root)();
333     int NiceVal;
334 {
335     int retCode;
336
337     #ifdef DEBUG
338     printf("createProcess\n");
339     #endif
340
341     retCode = fork();
342
343     if (retCode == -1) { /* grave problem */
344         perror("createProcess:");
345         exit (errno);
346     } else if (retCode == 0) { /* Child executes proc */
347         setpriority(PRIO_PROCESS, 0, NiceVal);
348         Root ();
349         exit (0);
350     } /* Parent returns */
351
352     #ifdef DEBUG
353     fprintf(stderr, "kill %d\n", retCode);
354     #endif
355     return retCode;
356 }
357
358 /*
359  * signal_flag_after_time - forks off a process that listens
360  * on flagForReset until timeout has expired. If timeout
361  * expires, then it pushes flagToSignal, otherwise it just dies.
362  * It is assumed that flagToSignal and flagForReset have been
363  * initialized elsewhere.
364  *
365
366 void signal_flag_after_time(float timeout,

```

```
367     char *flagToSignal,
368     char *flagForReset)
369 {
370     /* createProcess(); */
371 }
372
373 /*
374  * reset_timed_flag - pushes the flagForReset, thereby killing
375  * the process which is serving as a timer before it can timeout
376  * and signal flagToSignal (see signal_flag_after_time).
377  */
378
379 void reset_timed_flag(char *flagForReset)
380 {
381     /* */
382 }
```


END

1 8 0 6 - 9 7

FIN