A MODEL-CONTINUOUS SPECIFICATION AND DESIGN METHODOLOGY FOR EMBEDDED MULTIPROCESSOR SIGNAL PROCESSING SYSTEMS

A Dissertation
Presented to
The Academic Faculty

By

Randall Scott Janka

In Partial Fulfillment
Of the Requirements for the Degree
Doctor of Philosophy in Electrical and Computer Engineering

Georgia Institute of Technology
December, 1999

Copyright © 1999 by Randall S. Janka

A MODEL-CONTINUOUS SPECIFICATION AND DESIGN METHODOLOGY FOR EMBEDDED MULTIPROCESSOR SIGNAL PROCESSING SYSTEMS

Approved:

____

Linda M. Wills, Chairman

____

Mark A. Richards

____

Sudhakar Yalamanchili

Date Approved _________________________

Dedication

"But as for me, I trust in You, O LORD, I say, 'You are my God.'
My times are in your hand…."
–Psalm 31:14-15a

To our awesome God
who engineered this wondrous creation
and our salvation through his son Jesus Christ;
And to my beloved wife
Beverly
and our three incredible children
Alexandra, Bethany, and Victoria

Acknowledgements

This is the section that is written last and that I have been looking forward to the most. It should be written last, because it is only at the end of such a long road can one appreciate all who have helped him travel down that road all the way to its end. My road is long, having begun sixteen years ago, so I owe a debt of gratitude to many.

It has been an absolute blessing to be a part of Georgia Tech, especially being able to work full-time for the Georgia Tech Research Institute (GTRI). There are two people at Georgia Tech who have been key in my completing this dissertation and for whom I am extremely grateful. The first is my advisor, Dr. Linda Wills, who has been phenomenal in helping me pull order out of chaos. Besides providing excellent mentoring in how to develop my thesis, she has been incredibly supportive and encouraging. Finding editorial comments with time stamps of one or three in the morning says a lot about her dedication. Her enthusiasm and professionalism are exemplary. The other is my boss, Dr. Mark Richards, who hired me into GTRI and made sure that I got into their special program to complete my dissertation by giving me a year off half-time while receiving full-time compensation. He has been an advocate, mentor, supporter, and has also been patient with me while I finished this dissertation. His technical acumen, integrity, and concern for his people are remarkable.

There are others at Georgia Tech who have been most helpful. I am thankful for both Dr. Dale Ray and Dr. David Hertling for their oversight. Dr. Vijay Madisetti has provided some invaluable literature references and search directions, for which I am thankful. I am very grateful for those who have served on my committee, which includes Dr. Henry Owen, Dr. Mark Richards, Dr. Linda Wills, and Dr. Sudhakar Yalamanchili. They have been a great source of insight and support. Thanks also to Marilou Mycko and Glenna Thomas who provide amazing administrative support, often under great pressure.

There are others at GTRI who have been quite helpful while completing my dissertation. They include my lab director, Dr. Bob Trebits of SEAL and my division director Guy Morris of RSD. Janice Rogers of HQ has been creative and flexible in providing coverage for dissertation completion as well as conferences. Others at GTRI who have been extremely helpful over the last couple of years include support staff like Melanie Price, Karen Everson and her computer support staff. Fellow GTRI research colleagues who have been both helpful and just plain encouraging include Dr. Byron Keel, Dr. Chris Barnes, Dr. Bill Holm, and especially Bill Marshall, my fellow traveler on the long road with whom I've had the pleasure of commiserating evenings, late at night, and weekends. Congratulations, Dr. Marshall–and thanks for all your help beating the network into submission so I could do all my performance modeling simulations.

A critical element of my research included the integration of commercial products into my methodology. I have had the pleasure of dealing directly with the developers of bleeding edge state-of-the-art COTS multiprocessing hardware and software that have been so integral to my research. I have been most fortunate in being able to have received gratis the hardware and software that was the best of breed as well as what I specified. I am extremely grateful to Mercury Computer Systems not just for the privilege of being able to work there for two years, but especially for their generous contributions to GTRI of hardware and software to support my research and our project work. Thanks to Barry Isenstein, Arlan Pool, Dave Toms, Karen Lauro, and the customer support team. It has been a privilege and pleasure to have worked directly and indirectly with the technical staff at The MathWorks over the years, especially recently on this dissertation. I am grateful for their software donations and for the interchanges with Dr. Don Orofino and Rick Drohan. I am especially thankful for Stuart McGarrity for his help with the SAR Simulink modeling. Viewlogic has been generous in both their software donations and assistance, and I thank Binoy Yohannan and especially Mark Hepburn for his help with the performance modeling using eArchitect. Thanks, also, to Christopher Robbins and Carl Ecklund of MCCI for PGM ACT, and to W. Bernard Schaming and William Lundgren of Lockheed Martin ATL for GEDAE.

It has been a real pleasure working in the middleware arena, co-chairing the VSIPL Forum and peripherally supporting the MPI/RT Forum. Those whose insights and contributions to VSIPL have helped make it a genuine computation middleware standard include Dr. David Schwartz, Randall Judd, Dr. James Lebak, Dr. Sharon Sacco, Steve Paavola, and James Kenny. I am especially grateful for Dan Campbell of GTRI who has so ably stepped in and helped me with the Test Suite so I could co-chair the VSIPL Forum and for the times I had to occasionally retreat because of my dissertation.

Jesus said, “the first shall be last.” I close these acknowledgements with expressing the deepest of gratitude to my wife, Beverly, who has been a radiant woman of God through the tough times of completing my doctorate. Her belief in me, encouragement, joyful disposition (especially her singing), and never complaining even during crunch times is a testimony to her heart and faith. My three daughters have also paid a price for their daddy to finish this dissertation. While Alexandra, Bethany, and Victoria made it clear they missed their daddy, they have persevered and been patient. Their notes, prayers, hugs, and kisses kept me going on more than one occasion.

Most of all I want to thank God who has been so incredibly faithful to His promises and to me (Psalm 37:3-6). I knew that He would have to intervene in His own sovereign way for me to complete this dissertation. I decided after my twins were born, since I was a disciple of Jesus, a husband, and a father, that finishing my graduate work would require some unconventional creativity on God’s part. It seemed that finishing my doctorate and most recently my dissertation would have to be part of a full-time job since there would be no other way that I could be the disciple, husband, or father Jesus calls men to be. So God moved me to Atlanta to work for GTRI. Yet even with this blessing, it has been a challenge. I thank God for the inspiration, example, faithfulness, conviction, prayers and love of the awesome Atlanta Church of Christ and especially Sid Howell and Wayne Sisco for being “brothers who have stuck closer than a friend” (Proverbs 18:24). Thank you Father, for your faithfulness and for holding my hand.

“If the LORD delights in a man's way, He makes his steps firm;
though he stumble, he will not fall, for the LORD upholds him with His hand.”

–Psalm 37:23-24

Table of Contents

1 INTRODUCTION 1

1.1 THE BASIC PROBLEM ... 1

1.2 A SOLUTION TO THE PROBLEM ... 5

1.3 CONTRIBUTIONS ... 6

1.3.1 Novel Methodology ... 6

1.3.2 Validation of Methodology with Complex Application ... 7

1.3.3 Intrinsic and Extrinsic Assessment of Frameworks ... 8

1.4 DISSERTATION ORGANIZATION ... 9

1.5 CONVERGENCE OF RESEARCH THREADS ... 10

2 PROBLEM BACKGROUND 11

2.1 BACKGROUND ... 11

2.1.1 Application & Technology Domains ... 11

2.1.2 Current Design Practice in Our Domain ... 16

2.1.3 Specification and Design Methodology (SDM) ... 24

2.1.4 Implementation Frameworks ... 27

2.2 THE DOMAIN-SPECIFIC PROBLEM ... 34

2.2.1 Traditional SDM ... 34

2.2.2 Unified SDM of RASSP ... 35

2.3 A DOMAIN-SPECIFIC SOLUTION ... 35

3 SYSTEM REQUIREMENTS AND INTRINSIC SDM ASSESSMENT 37

3.1 SYSTEM REQUIREMENTS SPECIFICATION DOMAINS ... 38

3.2 DOMAIN-RELEVANT MODELS OF COMPUTATION ... 39

3.2.1 Definition of MoC ... 39

3.2.2 Domain-Relevant MOCs ... 39

3.3 BEST MOCS FOR DOMAIN-SPECIFIC SPECIFICATION AXES 43
3.3.1 ADoI-based Simplifications... 44
3.3.2 Computational MoC... 45
3.3.3 Communication & Control MoC.. 46
3.3.4 Constraints MoC 46
3.4 IMPLICIT FRAMEWORK MOCS 47
3.4.1 Ptolemy.. 48
3.4.2 Khoros 48
3.4.3 RTExpress.. 48
3.4.4 MATLAB DSP Workstation 48
3.4.5 PGM Autocoding Toolset.. 49
3.4.6 GEDAE... 49
3.4.7 RIPPEN.. 49
3.4.8 Talaris & Associated Frameworks... 49
3.5 COMPARING THE MONOLITHIC FRAMEWORKS 50
4 QUANTIFIED EXTRINSIC SDM ASSESSMENT 51
4.1 A UNIFIED SPECIFICATION-MODELING METHODOLOGY EVALUATION FRAMEWORK 52
4.1.1 Attributes 54
4.1.2 Sub-attributes 56
4.1.3 Metrics.. 58
4.2 QUANTIFICATION OF SARKAR BASIS 58
4.2.1 Quantifying the Language Support Attribute.. 61
4.2.2 Quantifying the Complexity Control Attribute... 67
4.2.3 Quantifying the Model-Continuity Attribute 71
4.2.4 Quantification of the Attributes and a Methodology.................................... 76
4.3 USING QUANTIFIED BASIS TO CHARACTERIZE CASE SDM FRAMEWORKS........... 78
4.3.1 Language Support Attributes 78
4.3.2 Complexity Control Attributes 80

4.3.3 Model Continuity Attributes ... *80*
4.3.4 Summary ... *82*
4.4 CONCLUSION ... 85

5 EXTENDING GAJSKI'S SER METHODOLOGY 86

5.1 BACKGROUND ... 86
5.2 PARALLELS BETWEEN GAJSKI'S SER AND OUR ADOI ... 87
5.3 EXTENDING GAJSKI'S SER TO OUR ADOI ... 93

6 THE MAGIC SPECIFICATION AND DESIGN METHODOLOGY 95

6.1 OVERVIEW OF THE MAGIC METHODOLOGY ... 95
6.2 ESTABLISHING MODEL CONTINUITY ... 98
6.3 "RULES"–THE STEPS OF THE MAGIC SDM ... 101
6.3.1 Tabulate Requirements ... *101*
6.3.2 Capture Non-Constraint Requirements in an Executable Model ... *101*
6.3.3 Build Executable Workbook with Requirements ... *102*
6.3.4 Gather Benchmarks for Tokens ... *102*
6.3.5 Explore Alternative Architectures and Technologies ... *102*
6.3.6 Make Design Decisions ... *103*
6.3.7 Create Implementation Specification ... *103*
6.4 "TOOLS"–THE FRAMEWORKS INTEGRATED INTO THE MAGIC SDM ... 104
6.4.1 DSP Workstation ... *104*
6.4.2 Excel and MATLAB Excel Link ... *107*
6.4.3 eArchitect ... *108*
6.5 MODEL CONTINUITY VIA MIDDLEWARE ... 109
6.5.1 VSIPL: Computational Middleware ... *110*
6.5.2 MPI: Communications Middleware ... *111*
6.5.3 Using VSIPL & MPI for Model Continuity ... *114*

7 CASE STUDY: VALIDATING THE MAGIC SDM USING A SAR PROCESSOR APPLICATION 120

7.1 RASSP SAR BENCHMARK OVERVIEW ... 121

7.1.1 Application Domain for the RASSP SAR Benchmark ... *121*
7.1.2 SAR Processing Overview and Assumptions ... *122*
7.2 TABULATE REQUIREMENTS ... 126
7.3 CAPTURE NON-CONSTRAINT REQUIREMENTS IN AN EXECUTABLE MODEL ... 129
7.3.1 Non-Parallel Pipelined Model ... *129*
7.3.2 Parallel Pipelined Model ... *133*
7.4 BUILD EXECUTABLE WORKBOOK WITH REQUIREMENTS ... 137
7.5 GATHER BENCHMARKS FOR TOKENS ... 138
7.6 EXPLORE ALTERNATIVE ARCHITECTURES AND TECHNOLOGIES ... 138
7.6.1 VSIPL Code Generation ... *139*
7.6.2 MPI Code Generation ... *141*
7.6.3 Latency Estimation ... *141*
7.7 MAKE DESIGN DECISIONS ... 152
7.8 CREATE IMPLEMENTATION SPECIFICATION ... 154
7.9 DIFFICULTIES ENCOUNTERED AND OVERCOME ... 154
7.9.1 Limitations of Simulink ... *154*
7.9.2 Limitations of eArchitect ... *156*
7.10 CONCLUSION ... 160

8 MAGIC QUANTIFICATION AND CONCLUSION 162

8.1 MODEL CONTINUITY IN THE MAGIC SDM ... 162
8.2 SARKAR QUANTIFICATION OF MAGIC SDM ... 167
8.2.1 Language Attribute ... *167*
8.2.2 Complexity Control Attribute ... *169*
8.2.3 Model Continuity Attribute ... *169*
8.2.4 Attribute Aggregate Values ... *173*
8.3 SUMMARY ... 176
8.4 DIRECTIONS FOR FURTHER RESEARCH ... 177
8.4.1 Applied Framework Research ... *177*
8.4.2 Basic Methodological Research ... *177*

APPENDIX A: DETAILS OF VSIPL AND MPI MIDDLEWARE 179

A.1 VSIPL DETAILS ... 179

A.1.1 VSIPL Fundamentals ... 179

A.1.2 VSIPL Concepts ... 181

A.2 MPI: COMMUNICATIONS MIDDLEWARE ... 187

A.2.1 Standardization and Functionality ... 188

A.2.2 Basic Theory of Operation ... 189

APPENDIX B: DETAILS OF CASE STUDY 192

B.1 SIMULINK DETAILS ... 192

B.1.1 Simulink Start-up and Initializing MATLAB Workspace ... 192

B.1.2 Addressing Matrices as Vectors ... 193

B.1.3 Stripping Off Previous SAR Image Frame ... 194

B.1.4 Executing the Specification and Flushing its Queue ... 194

B.1.5 Scatter/Gather with Demux/Mux ... 195

B.2 VSIPL CODE GENERATION SUBTLETIES ... 196

B.3 EARCHITECT DETAILS ... 197

B.3.1 STARTING UP EARCHITECT ... 197

B.3.2 HARDWARE MODEL LAYERS ... 199

B.3.3 SOFTWARE EDITOR GUI DETAILS ... 201

B.3.4 SCATTER/GATHER DETAILS ... 201

B.3.5 CODING THE PROCESSES OTHER THAN FOR RANGE PROCESSING ... 202

B.3.6 SETTING UP EARCHITECT FOR SIMULATION ... 204

B.3.7 RUNNING SIMULATIONS IN EARCHITECT ... 205

REFERENCES 212

List of Tables

Table 2-1. Basic characteristics of COTS MP codesign frameworks. 34

Table 3-1. Summary of models of computation (MoCs) with example frameworks and/or languages 43

Table 3-2. Models of computation implicit in frameworks of §2.1.4. 51

Table 4-1. Language support sub-attributes spreadsheet. 79

Table 4-2. Complexity Control Sub-attributes spreadsheet. 80

Table 4-3. Model Continuity Sub-attributes spreadsheet. 81

Table 4-4. Raw values for attribute integrations. 82

Table 4-5. Normalized attribute values for the CASE SDMs. 82

Table 5-1. Extending SER design representation and abstraction levels to our ADoI (Board Level). 88

Table 5-2. Language support for conceptual model characteristics of embedded systems. 89

Table 5-3. System design tasks. 91

Table 6-1. Technologies in our application domain. 108

Table 6-2. Summary of transformation rules for code generator. 118

Table 7-1. Summary of system requirements and constraints. 126

Table 7-2. Environmental non-performance characteristics of processor boards. 128

Table 7-3. Example of system parameters portion of Token Quantification worksheet. 142

Table 7-4. Example of token quantification and non-IPC latency computation (`nPrange`=4, `nPazimuth`=8). 144

Table 7-5. Token Summary for performance modeling, including latency estimates without accounting for IPC. 146

Table 7-6. Ruling out architectures that do not meet scalability requirement (in black). 146

Table 7-7. Latencies of SAR processor architectures accounting for IPC. (Architectures that do not meet scalability requirement are shaded.) 151

Table 7-8. Assessing our design options, optimizing on minimal board count, N_{boards}. . 153

Table 8-1. Normalized language attribute values and the improvement with the MAGIC SDM. 167

Table 8-2. Language support sub-attributes spreadsheet including MAGIC SDM. 168

Table 8-3. Normalized complexity control attribute values and the improvement with the MAGIC SDM. 169

Table 8-4. Normalized model continuity attribute values and the improvement with the MAGIC SDM. 170

Table 8-5. Complexity control sub-attributes spreadsheet including MAGIC SDM...... 171

Table 8-6. Model continuity sub-attributes spreadsheet including MAGIC SDM. 172

Table 8-7. Raw attributes for CASE SDMs vis à vis Ideal and MAGIC SDMs............ 173

Table 8-8. Normalized attributes for CASE SDMs vis à vis Ideal and MAGIC SDMs. 173

Table 8-9. Normalized model continuity attribute values and the improvement with the MAGIC SDM. 176

Table B-1. Subtle VSIPL code generation equivalents... 197

List of Figures

Figure 1-1. Cost of specification and design errors increase throughout process (after Figure 1.5 in [1]). 1

Figure 1-2. Basic flow of information needed to support model continuity. 4

Figure 1-3. Convergence of research threads leading to our MAGIC SDM. 10

Figure 2-1. Simplified functional block diagram of "The Processor" in a generic radar system. 12

Figure 2-2. Simplified signal flow and system states representative of the application domain of The Processor of Figure 2-1; the shading shows how such a design might be mapped to an embedded MP target. 14

Figure 2-3. Potential architecture to implement The Processor of Figure 2-1. 16

Figure 2-4. Fundamental perspective of layering software on hardware. 19

Figure 2-5. Expanded model of canonical MP hardware/software codesign layers. 19

Figure 2-6. Canonical model as implemented with recent and current COTS vendor MPs. 20

Figure 2-7. Framework codesign model decoupling computation from configuration. 21

Figure 2-8. Model of Figure 2-6 utilizing standardized middleware. 23

Figure 2-9. Model of Figure 2-7 utilizing standardized middleware. 24

Figure 4-1. Taxonomy of Sarkar's unified reactive-systems specification methodology attributes (branches) and sub-attributes (leaves). 54

Figure 4-2. Attributes of Sarkar basis. 55

Figure 4-3. Sub-attributes of the language support attribute (in blue). 61

Figure 4-4. Sub-attributes of the complexity control attribute (in blue). 67

Figure 4-5. Sub-attributes of the model continuity attribute (in blue). 72

Figure 4-6. Graphical representation of Sarkar basis attributes. 76

Figure 4-7. Plot of normalized attribute values for the CASE SDMs in Table 4-5. 83

Figure 4-8. Plot of CASE SDMs and Ideal SDM in 3-tuple space. 84

Figure 5-1. Gajski et al.'s three classes and scopes of design methodology.................... 91

Figure 5-2.Extending Gajski's SER (from Figure 5-1) to our ADoI. 94

Figure 6-1. Simplified diagram of the MAGIC specification and design flow................ 97

Figure 6-2. How model continuity is currently lacking in CASE SDMs......................... 99

Figure 6-3. Establishing model continuity between an executable specification model and a design specification model. 100

Figure 6-4. Simple Simulink model to illustrate code generation................................. 119

Figure 7-1. SAR block diagram with SAR image processor highlighted in blue. 123

Figure 7-2. SAR image formation algorithm flow. 125

Figure 7-3. Simulink model of single threaded version of our SAR processor. 129

Figure 7-4. The model for video-to-baseband conversion. 130

Figure 7-5. Range processing block of our Simulink SAR model................................ 131

Figure 7-6. Azimuth processing block of our Simulink SAR model. 132

Figure 7-7. Parsing the SAR processor into separate processes. 133

Figure 7-8. Parallelized SAR processing Simulink model where `nPrange`=4 and `nPazimuth`=8. 135

Figure 7-9. Cornerturn required when parallelizing range and azimuth processing....... 136

Figure 7-10. Token Quantification worksheet from executable workbook (`nPrange`=4, `nPazimuth`=8).. 143

Figure 7-11. First order estimation of system latencies, based on middleware token delays in the absence of performance modeling.. 147

Figure 7-12. Hardware model used for all of our performance models. 148

Figure 7-13. The software model of our performance model. 149

Figure 7-14. The range process with a template used in all of the blocks. 150

Figure 7-15. Latencies for SAR processing architectures based on performance modeling simulations. 152

Figure 7-16. Mapping window for the SAR processing performance model. 158

Figure 7-17. Scatter/send code that shows our flexible mapping. 158

Figure 7-18. Local variables for range processing. 159

Figure 8-1. How model continuity is currently lacking in CASE SDMs (Figure 6-2). 163

Figure 8-2. Establishing model continuity between an executable specification model and a design specification model (Figure 6-3). 164

Figure 8-3. Establishing model continuity between an executable specification model and a design specification model. 166

Figure 8-4. Plot of normalized attribute values for the CASE SDMs vis à vis the Ideal and MAGIC SDMs from Table 8-8. 174

Figure 8-5. Plot of SDMs in 3-D attribute-space, which shows MAGIC's improvement over CASE SMDs moving towards Ideal SDM. 175

Figure A-1. Data space characteristics and interrelationship. 184

Figure A-2. VSIPL application flow. 187

Figure B-1. Data input block. 193

Figure B-2. Formatting the SAR image for output. 194

Figure B-3. Setting the model execution parameters. 195

Figure B-4. Demux dialog box used for scattering the data for range processing. 195

Figure B-5. Mux dialog box used for gathering range results for the cornerturn. 196

Figure B-6. Project window at the beginning of an eArchitect session. 198

Figure B-7. Opening the supplemental Mercury library to support our MAGIC SDM. 198

Figure B-8. Hardware model of Mercury MCH6 motherboard. 199

Figure B-9. Hardware model of Mercury PPC daughtercard. 200

Figure B-10. One of two compute nodes (CE) on daughtercard. 201

Figure B-11. Contents of a software block's process that performs a delay. 202

Figure B-12. Local variables for FIR processing. 203

Figure B-13. Local variables for the cornerturn. 203

Figure B-14. Local variables for azimuth processing. 204

Figure B-15. Local variables for display processing. 204

Figure B-16. Parameters of the input data source. 205

Figure B-17. Dialog window to set parameters for simulation. 206

Figure B-18. Tool for accessing simulation runs for analysis. 206

Figure B-19. Analysis tools for visualization of simulation data. 207

Figure B-20. After loading our simulation run we see the bottom-line latency. 207

Figure B-21. Tools options in the Analysis Control Panel. 208

Figure B-22. Simulation time-line showing the exact latency of the simulation is 1.207 seconds. 209

List of Acronyms

ACL	Application Configuration Language
ACT	Autocoding Toolset
ACT	Autocoding Toolset
ADC	Analog-to-digital converter
ADoI	Application domain of interest
API	Application programming interface
BM-2	Benchmark-2
CAD	Computer-aided design
CASE	Computer-aided system engineering
CE	Compute Element
CFD	Control flow diagram
CFSM	Communicating finite state machine
COTS	Commercial-off-the-Shelf
CPS	Composite program-states
DAC	Digital-to-analog converter
DARPA	Defense Advanced Research Projects Agency
DCDF	Directed-control dataflow
DE	Discrete event
DF	Dataflow
DFD	Dataflow diagram
DFT	Discrete Fourier transform
DSP	Digital signal processing
DSPW	DSP Workstation (MATLAB/Simulink/Stateflow framework)
FFT	Fast Fourier transform
FLOPS	Floating-point operations

FLOPS/s	Floating-point operations per second
FP('s)	Function Point(s)
FSM	Finite state machine
GE	Graphical editor
GEDAE	Graphical Entry, Distributed Application Environment
GFLOPS/s	Gigaflops per second
GPP	General-purpose processor
GUI	Graphical user interface
IPC	Interprocessor communication
ISPME	Integrated Specification and Performance Modeling Environment
LAN	Local area network
MAGIC	Methodology Applying Generation, Integration, and Continuity
MB/s	Megabytes per second
MCCI	Management Communications and Control, Inc.
MFLOPS	Million floating-point operations
MFLOPS/s	Million floating-point operations per second
MIPS	Million instructions per second
MoC	Model of computation
MP	Multiprocessor
MPI	Message Passing Interface
MPI/RT	Message Passing Interface/Real-Time
MSTI	MPI Software Technology, Inc.
NOW	Network of workstations
OO	Object-oriented
PMS	Processors/Memories/Switches
PS	Program-states
PSM	Programming-state machine
PSS	Program-substates
QoS	Quality-of-service

RASSP	Rapid prototyping of Application Specific Signal Processors
RIPPEN	Real-time Interactive Programming and Processing Environment
RISC	Reduced Instruction Set Computing
RTOS	Real-time operating system
RTW	Real-Time Workshop
RTWG	RASSP Taxonomy Working Group
S/R	Synchronous/reactive
SAN	System-area network
SAR	Synthetic aperture radar
SBC	Single board computer
SPC	Scalable parallel computer
SDE	Software development environment
SDF	Synchronous dataflow
SDL	Specification and Description Language
SDM	Specification and design methodology
SDS	Software development specification
SDE	Software development environment
SER	Specify-Explore-Refine
SLDL	System Level Design Language
SoC	System-on-a-chip
SPMD	Single-program multiple-data
SREM	Software Requirements Engineering Methodology
SRS	System requirements specification
STAP	Space-time adaptive processing
SWAP	Size, weight, and power
TBD	To be defined
TMW	The MathWorks
UI	User interface
VP	Virtual prototype/prototyping

VSIPL	Vector/Signal/Image Processing Library
WAN	Wide area network

Summary

The process of designing large real-time embedded signal processing systems is plagued by a lack of coherent specification and design methodology. A canonical waterfall design process is commonly used to specify, design, and implement these systems with commercial-off-the-shelf (COTS) multiprocessing (MP) hardware and software. Powerful frameworks exist for each individual phase of this canonical design process, but no single methodology exists which enables these frameworks to work together coherently, i.e., allowing the output of a framework used in one phase to be consumed by a different framework used in the next phase.

This lack of coherence usually leads to design errors that are not caught until well in to the implementation phase. Since the cost of redesign increases as the design moves through these three stages, redesign is the most expensive if not performed until the implementation phase, thus making the current incoherent methodology costly. This dissertation shows how designs targeting COTS MP technologies can be improved by providing a coherent coupling between these frameworks, a quality known as "model continuity."

We have developed a new specification and design methodology (SDM) which accomplishes the requirements specification, design exploration, and implementation of COTS MP-based signal processing systems by using powerful commercial frameworks that are intelligently integrated into a single domain-specific SDM. Our integration establishes model continuity by using autogenerated computation (VSIPL) and communication (MPI) standards-based middleware. We have dubbed our new SDM MAGIC, an acronym for "Methodology Applying Generation, Integration, and Continuity."

To measure improvement, we have developed an analytical means of measuring SDMs in our domain by quantifying Sarkar's unified basis for evaluating specification-

modeling methodologies. We measured computer-aided system engineering (CASE) SDMs capable of generating real-time code and our own MAGIC SDM, and found the MAGIC SDM was much closer to ideal than the CASE SDMs. We have also validated the MAGIC SDM and demonstrated its efficacy with a real-world benchmark. In so doing we also demonstrated that the MAGIC SDM is clearly superior to both VHDL virtual prototyping and the CASE-based SDMs that must commit to an implementation technology *before* performing design analysis. We also consider further research directions.

Chapter 1

Introduction

1.1 The Basic Problem

The process of designing large real-time embedded signal processing systems is plagued by a lack of coherent specification and design methodology. A canonical waterfall design process is commonly used to specify, design, and implement these systems with commercial-off-the-shelf (COTS) multiprocessing (MP) hardware and software. Powerful frameworks exist for each individual phase of this canonical design process, but no single methodology exists which enables these frameworks to work together coherently, i.e., allowing the output of a framework used in one phase to be consumed by a different framework used in the next phase.

This lack of coherence usually leads to design errors that are not caught until well in to the implementation phase. Since the cost of redesign increases as the design moves through these three stages (see Figure 1-1), redesign is the most expensive if not performed until the implementation phase, thus making the current incoherent methodology costly. This dissertation shows how designs targeting COTS MP technologies can be improved by providing a coherent coupling between these frameworks, a quality known as "model continuity."

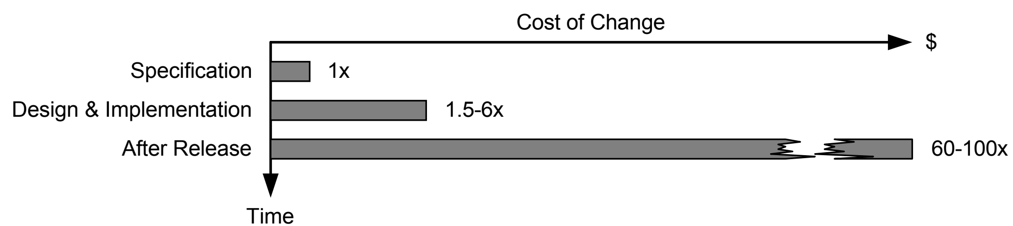

Figure 1-1. Cost of specification and design errors increase throughout process (after Figure 1.5 in [1]).

The lack of model continuity has a variety of negative impacts, as can be seen in the following scenario. Suppose a signal processor is to be implemented with COTS MP hardware and software. Algorithms are developed, modeled, and specified in some pseudocode, perhaps in MATLAB. System constraints (e.g., size, weight, and power) for the processor are specified by a system engineer. In this traditional methodology, there is no way to use this pseudocode in the design analysis phase. The specification is partially executable; the behavioral part of the specification (signal processing algorithms) is typically written in MATLAB, which is executable. However, this partial specification model cannot be used in the design analysis phase (which means there is no model continuity). Due to the absence of system-level design and analysis tools that have a model of the overall system behavior, only a few low-precision calculations can be made, which are based on published specifications of competing vendors' interprocessor bandwidths and algorithm benchmarks, and adjusted based on experience. A vendor is chosen and design begins. Even though margins have been included via some heuristic rules of thumb, it is only when the detailed design is complete that it is seen that the hardware chosen cannot meet system throughput requirements. Unfortunately, the allotted chassis space is already full with a technology that will not meet specifications because the low-precision analysis was unable to predict complex interconnectivity between compute elements used to implement the dataflow model. Despite the project schedule being at great risk, the design process must start over. The engineering staff will have to make up the lost time and only hope they will be successful with the next iteration.

If on the other hand, the requirements model was executable, and the requirements model along with non-performance constraints could have been passed to a design tool framework, then the system engineer would have been equipped to consider and evaluate alternative architectures and implementation technologies before implementation proceeded. The system engineer could have made sure that alternatives would at least satisfy requirements, then achieve a near-optimal design solution. This could have been accomplished *before* committing the design prematurely to a particular technology that could not satisfy requirements and constraints, despite its promising

specifications because the complexity of the design hid subtle technology limitations. Instead, the system engineer would be able to specify the technology, software processes, hardware configuration, and a software-to-hardware mapping.

This unnecessary and costly redesign could have been avoided if only model continuity and the right integration of tools had been present in our engineers' specification and design methodology. Important system information revealed by having an executable specification would not have been lost in the design phase, such as accounting for nondeterministic interprocessor communication and assuring that candidate architectures satisfy non-performance constraints. Similarly, important information revealed in the design phase would have been leveraged in the implementation phase, such as software-to-hardware mapping as well as software functions and parameter arguments. The necessary flow of information is illustrated in Figure 1-2.

Unfortunately, the previous unfortunate design scenario is not uncommon. In recent years both market forces and technological requirements have been driving the design process to limit hardware options to COTS hardware. In the radar signal processing domain, this means using RISC[1]-based and DSP[2]-based multiprocessor boards with high-speed interprocessor bandwidth and C language support. Despite limiting the design space to a finite number of hardware options, the design process has still been challenging, given compressing development cycles that increase software development productivity requirements, implicitly requiring that software be portable, so that previous design and development efforts can be reused. This productivity and portability must be achievable without an appreciable loss of system performance.

---

[1] RISC–Reduced Instruction Set Computing architecture, e.g., the superscalar PowerPC from the Motorola/IBM/Apple consortium.

[2] DSP–Digital Signal Processor, e.g., microprocessors tuned for 1-D signal processing numeric computation such as the SHARC (super Harvard architecture) from Analog Devices.

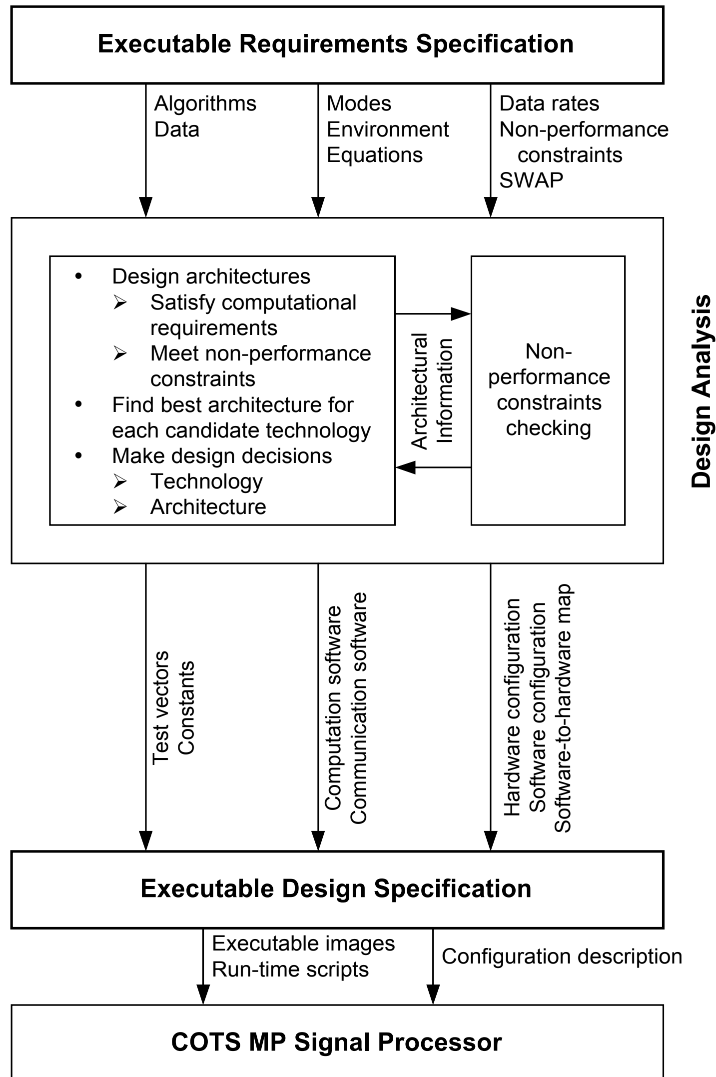

Figure 1-2. Basic flow of information needed to support model continuity.

A partial response to this design challenge of real-time multiprocessor digital signal processing systems has been the development of different frameworks of tools, such as GEDAE, RIPPEN, and PGM ACT, to provide computer-aided system engineering (CASE) support for system implementation. In particular, these frameworks offer code generation that reduces the complexity of system configuration and communication coding, a quality known as "complexity control." Yet no one single framework or one single language can cover the entire design process. Powerful implementation tools can generate deployable application code, but are weak in capturing requirements and are difficult to use in exploring architecture design alternatives. Some

languages, such as MATLAB, are powerful in capturing computational requirements, but do not readily lend themselves to being used for deployed implementations.

1.2 A Solution to the Problem

A single domain-specific specification and design methodology (SDM) is needed which accomplishes the following:

- Leverages existing frameworks that in and of themselves are inadequate for providing a complete SDM by extending and integrating them into a single SDM.
- Uses the right tool at the right time, by using tools whose granularity and utility are matched to the appropriate phase of the specification and design process.
- Establishes model continuity to maximize executability and minimize loss of system design detail.

The requirements specification, design exploration, and implementation of COTS MP-based signal processing systems can be accomplished using powerful commercial frameworks that are intelligently integrated into a single domain-specific SDM. The integration makes use of the model continuity provided by middleware and supported by the frameworks.

Existing deployable implementation frameworks alone cannot fill this need. They fall short of being able to provide a complete specification and design methodological framework because they are biased toward a given COTS hardware/software target. However, these existing frameworks can be leveraged as enabling technologies. By integrating them in the right way, a model-continuous specification and design methodology can be specified, prototyped, and demonstrated to show a measured improvement over just using a single implementation framework.

Clearly, a methodology that consumes the pseudocode and system constraints at an architectural level early in the design process and passes it along to later stages would have spared the engineers in the hypothetical scenario (§1.1). Performance modeling design tools and implementation frameworks do exist but require modification to support such a methodology. Standardized computation and communication middleware (a software layer in between the application code and the COTS MP hardware and software)

is just now arriving that was intended to support portability, and as such can be used as a channel for model continuity.

This thesis presents an "integrated model-continuous" SDM, in which the requirements specification is converted to an executable model, which is translated into computation and communication middleware (industry standard API[3]s) that is a priori well-characterized on different COTS MP vendors' targets. Computation and communication middleware benchmark data is then used in a performance modeling framework that can provide high fidelity simulation of the MP data traffic during an architectural tradeoff evaluation phase. Thus, before committing to a particular COTS MP target, there is a high degree of confidence that the architecture can meet requirements because the steady state compute element code has been obtained by translating the pseudocode and simulating its execution. After arriving at an architecture that is known to satisfy requirements, the computation and communication middleware can be consumed by an implementation framework in developing the application software for the detailed design. This dissertation specifies and develops a methodology that achieves high measured levels of model continuity and complexity control that are specific to our particular application and technology domains.

1.3 Contributions

This dissertation has made a number of specific and concrete contributions to the domain of hardware/software codesign, targeting embedded real-time COTS MP systems that are used to implement radar signal processors and other similarly demanding systems.

1.3.1 Novel Methodology

In Chapter 6 we present a new methodology for the specification and design of COTS MP-based digital systems used for real-time embedded signal processing. It overcomes the methodological shortcomings of the state-of-the-art CASE frameworks by integrating them with tools that can capture requirements and explore alternate

---

[3] API–application programming interface.

architectures *before* committing to a specific vendor target. This methodology provides model continuity by using computation and communication middleware generated by the requirements specification tool in the design analysis and implementation frameworks, as well as an "executable workbook" that links specification and design.

The methodology begins with translating the natural language requirements specification into an executable requirements specification. Information iterates between the executable requirements specification and the design analysis framework. When design analysis is complete, the executable requirements specification and design analysis framework provide the inputs necessary for creating an executable design specification.

We have prototyped this middleware code generation but have not yet fully automated it, due to inaccessibility of the internals of the COTS frameworks employed. Vendors are beginning to support middleware and are heading towards supporting middleware code generation. We have dubbed it the "MAGIC" specification and design methodology (SDM), for "Methodology Applying Generation, Integration, and Continuity," which emphasizes the leveraging of code generation at both specification and implementation levels, integration of tool frameworks, and establishment and maintenance of model continuity.

1.3.2 Validation of Methodology with Complex Application

In Chapter 7 we show how we have validated that the MAGIC SDM can be used to accomplish the specification and design of a system representative of our application domain of interest (ADoI). We chose the RASSP[4] SAR[5] benchmark since it will be a level playing field on which to assess how our MAGIC SDM performs compared to the two main types of SDMs used with COTS MP technologies in our ADoI. The first type is virtual prototyping (VP), which is the specification and design of a digital system using an executable language such as VHDL. Virtual prototyping was found to be quite unwieldy for larger more complex applications like those found in this ADoI, because

[4] RASSP–Rapid prototyping of Application Specific Signal Processors, a DARPA program.
[5] SAR–synthetic aperture radar.

simulation runtimes were painfully long, and only those activities near the beginning of the hardware initialization cycle could be explored. For example, in the virtual prototyping of RASSP SAR, only the first 150 milliseconds of a 3-second frame could be simulated [2, 3]. The second type is using deployable CASE frameworks, which have some model continuity and complexity control, but require the developer to commit to a hardware target *before* starting the design phase, the reverse of what the specification and design process should do.

We validate the MAGIC SDM empirically by showing the following claims are true:

1) The MAGIC SDM works as postulated, which means the rules can be followed and the tools work–especially in providing model continuity. This is indicated in this chapter by a at the beginning of the paragraph.

2) The MAGIC SDM can simulate complex system performance for whatever period is necessary. (It is able to simulate at least 20 times longer than a comparable VP simulation on the SAR benchmark.) This enables the designer to obtain a high fidelity assessment of how well a candidate architecture and technology will do in meeting latency requirements.

3) The MAGIC SDM provides the framework to evaluate competitive technologies *prior to implementation*, which the CASE SDMs cannot do at all.

1.3.3 Intrinsic and Extrinsic Assessment of Frameworks

In this research, we have developed assessment techniques to characterize COTS MP CASE frameworks both intrinsically and extrinsically. By "intrinsic" we mean the basic structure and operation characterizing a CASE framework. In Chapter 3 we specified the different types of requirements pertinent to this application domain and discerned the best models of computation to capture these requirements. We have identified the underlying model(s) of computation intrinsic to the frameworks and compared them to the optimum. All the CASE frameworks have been found wanting, and the optimum models of computation provide guidance for the best types of frameworks to integrate into a new methodology.

"Extrinsic" refers to qualities we can ascribe to how a framework is used in specification and design modeling. We have a detailed quantitative means of extrinsic

assessment. In Chapter 2 we present how we have developed an analytical basis of comparison by quantifying a well regarded but qualitative unified methodology basis by Sarkar [4, 5]. This enabled us to compare the frameworks extrinsically, as well as visually in a 3-D methodology attribute space. This also provides the basis for clearly demonstrating methodological improvement provided by our new methodology, achieving a 20%–90% improvement over CASE SDMs in Sarkar space as shown in Chapter 8.

1.4 Dissertation Organization

This dissertation begins by reviewing existing specification and design methodologies as related to COTS MP technologies in Chapter 2. There are two pertinent unified COTS MP design methodologies discussed in this chapter. They are the VHDL-based virtual prototyping approach, which is based on a single model of computation, and the approach using software-oriented monolithic CASE frameworks.

Chapter 3 and Chapter 2 provide ways to assess these CASE SDM frameworks. Chapter 3 first considers an intrinsic means of assessing them by considering their implicit models of computation (MoCs) as weighed against the MoCs an ideal framework would possess. The tools capable of generating deployable (real-time) code are then assessed extrinsically in Chapter 2 by using a quantified unified specification and design methodology framework.

The weaknesses and shortcomings revealed by intrinsic and extrinsic assessment are addressed by adapting a single-board SDM recently developed by Gajski, which is discussed in Chapter 5. Extending this SDM from a single-board domain to our multiple-board domain of COTS MP leads to our new SDM. By leveraging middleware specific to the COTS MP domain and integrating the right frameworks, we are able to achieve both model continuity and a level of complexity control, which is currently lacking in any existing single framework-based SDM. Integrating different frameworks and exploiting a new COTS MP computational middleware leads to our new MAGIC integrated model-continuous SDM, which is described in Chapter 6. A moderately complex synthetic aperture radar (SAR) benchmarking application is used to demonstrate MAGIC and

assess MAGIC vis à vis other SDM frameworks. This SAR case study is presented in Chapter 7.

The dissertation concludes by summarizing the research and identifying directions for further research in Chapter 8.

1.5 Convergence of Research Threads

This research is in an area that is an overlap of digital systems design, signal processing, and computer engineering. Different threads of research, design, and development converge in this research, contributed by key individuals and projects. Complementing the list of references at the end of this dissertation is a "genealogy" of this research shown in Figure 1-3.

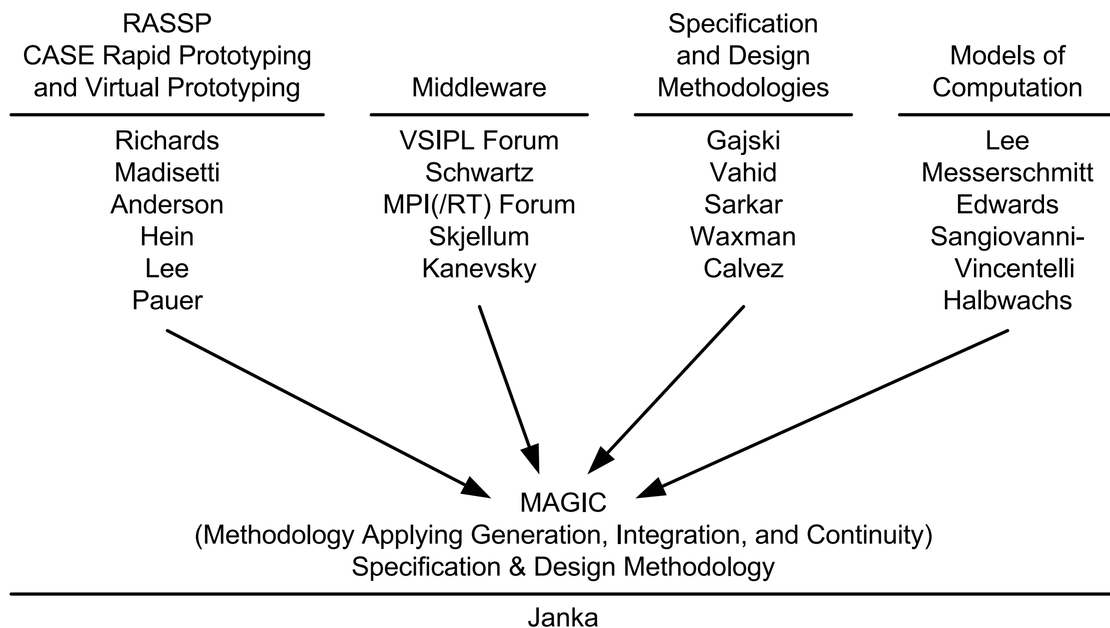

Figure 1-3. Convergence of research threads leading to our MAGIC SDM.[6]

[6] Rapid Prototyping and RASSP references: [3, 6-14]. Middleware references: [15-20]. Specification and design methodology references: [4, 21-25]. Models of computation references: [13, 26-28]. MAGIC-related references: [29-43].

Chapter 2

Problem Background

In this chapter we develop the motivation for assessing existing SDMs and for specifying a significantly improved SDM. We discuss what characterizes our application and technology domains in order to match the optimum methodology to them. We also review the recent movement to transition from custom hardware to COTS hardware and the impact that has had on the design process, especially the creation of certain CASE tool frameworks that are rapid prototyping tools in the least and actual SDM frameworks at best. Appreciating the problem area in general and the specific approaches taken by others will help us to move in the right direction toward developing a better SDM.

2.1 Background

Domain specificity is critical in the development of a sound specification and design methodology. This section reviews what characterizes our application and technology domains. The rest of the section deals with current engineering practice and its foundation and current evolution.

2.1.1 Application & Technology Domains

The application domain chosen as the focus of this research is the class of deployed systems performing signal processing for radar, including range Doppler radar, synthetic aperture radar (SAR), and space-time adaptive processing (STAP) radar. These are applications that are vector-oriented, either in one or two dimensions, typically requiring a large proportion of spectral operations (usually FFTs) and an increasing amount of linear algebra computation (e.g., QR solvers in STAP processing). The processing tends to be pipelined and distributed, with stages of the processing distributed

over multiple processors. The sheer volume of data moving through this processing demands a high-speed datapath whose bandwidth is complementary to the processors' throughput and local memory bandwidth.

These systems are described as "complex" because of the high volume of streaming signal data coming into the system and the high computational throughput required to process the data. Systems that meet this level of complexity include both commercial and defense airborne radar, shipborne radar and sonar, and automatic target recognition. These systems have multiple sensor inputs with individual data rates on the order of 20-100 MB/s (megabytes per second) and throughput requirements on the order of tens of GFLOPS/s (gigaflops per second) range, which require tens to hundreds of processors to handle the load [44] [45]. A functional block diagram of this type of system, which would lend itself to a COTS MP implementation, is shown in Figure 2-1. Note that often, "The Processor" can be decomposed into "the signal processor" and "the data processor." The signal processor typically computes data-independent algorithms invariantly, passing its results to the data processor. The data processor performs data-dependent computations, making decisions and maintaining a history of the processing.

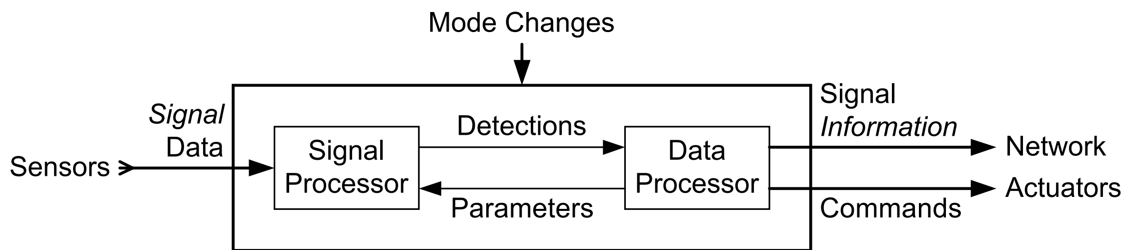

Figure 2-1. Simplified functional block diagram of "The Processor" in a generic radar system.

The Signal Processor is primarily data transformational in nature. Arriving at the input are large volumes of signal *data* (targets, noise, clutter, etc.) whose rates are known a priori and are typically constant. The signal data is processed, producing a much lower volume of signal *information* (detections and signal characterization). The output must be produced in a timely fashion such that input data is not lost. The Processor is a reactive system with real-time constraints, which means that it has a timeline that must be met.

Typically, a real-time controller governs the data transformation process, making the system as a whole reactive, where the data transformation's processing parameters (or "modes") change slowly if at all with respect to the data rate of the input. Further constraining the signal processor is that it must be deployed in an embedded system, which means hardware will be limited to those technologies which can satisfy non-performance constraints such as size, weight, and power limits. A simplified representative example showing signal paths and system states being mapped onto an embedded processing target is shown in Figure 2-2. A methodology must be able to specify and guide the system design considering these characteristics.

For reactive systems with data transformation cores that are embedded, the limitations of size, weight, and power ("SWAP") must be considered. This will drive specification, vendor selection, and design, possibly requiring reevaluation of the requirements. Other non-real-time requirements include reliability and testability. These characteristics are important for assuring ongoing performance of the system, but do not enter the execution requirements or analysis. Structurally, the system is under form factor constraints that restrict the hardware option envelope to those COTS technologies possessing minimal SWAP. Deployment constraints such as these limit the number of potential COTS targets to those products that can satisfy both robust form factor requirements along with scaleable multiprocessor boards that can deliver high computational throughput balanced by commensurate interprocessor bandwidth.

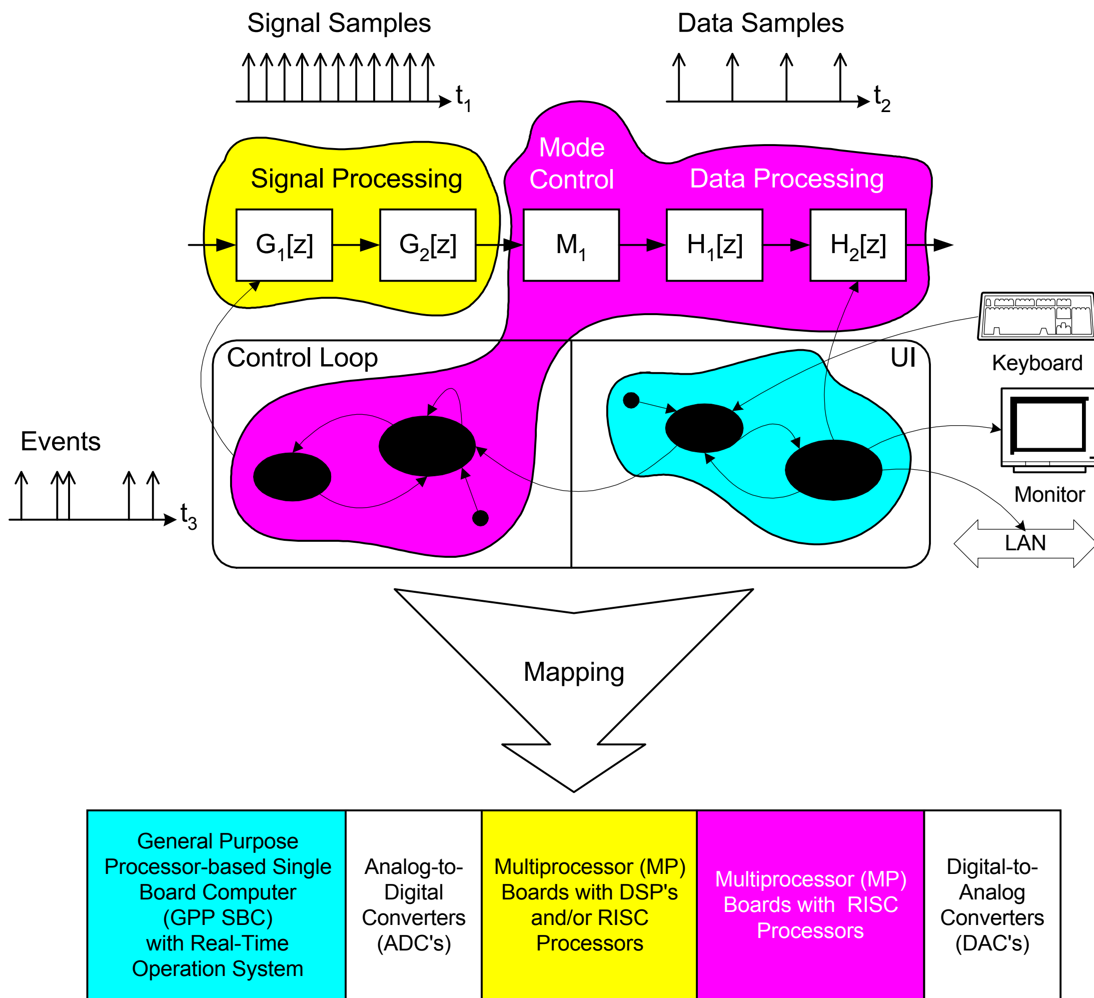

Figure 2-2. Simplified signal flow and system states representative of the application domain of The Processor of Figure 2-1; the shading shows how such a design might be mapped to an embedded MP target.

This hardware constraint will also affect the software development, and therefore the methodology to specify and design the signal processing system. Functionally, the COTS MP hardware world is not very dissimilar from the distributed heterogeneous network of servers and workstations. The computing environment consists of multiple microprocessors in a distributed shared memory architecture. Heterogeneity exists in the form of using DSPs and/or RISCs for vector-intensive computing and RISCs for more scalar-oriented computing. Instead of a LAN or WAN, networking occurs over a “SAN”

(system-area network). Paradigms from the distributed network computing community can be exploited in this domain as well.

Depending on density and processing requirements, implementations include DSP and/or RISC devices for the signal processing, and probably RISC devices for the data processing. Since the signal and data processor are data-driven processes, they will perform best with small real-time operating system kernels. To coordinate the two processors and provide system-wide real-time reactivity and network (LAN or WAN) connectivity, a general purpose processor (GPP) single board computer (SBC) system controller running a more robust real-time operating system (RTOS) is typically employed. Getting data in and out of The Processor will require analog-to-digital conversion (ADC) and digital-to-analog conversion (DAC) boards. Due to the volume of data, it can be advantageous to have dedicated parallel data paths between the conversion boards and the multiprocessor (MP) boards. Control/data and power can be provided over an industry-standard system bus such as VME or PCI. High-speed MP interconnectivity can be provided by industry standard vendor SANs such as RACEway, Myrinet, or SKYchannel. A simple architecture that shows a potential implementation of Figure 2-1 and Figure 2-2 is shown in Figure 2-3.

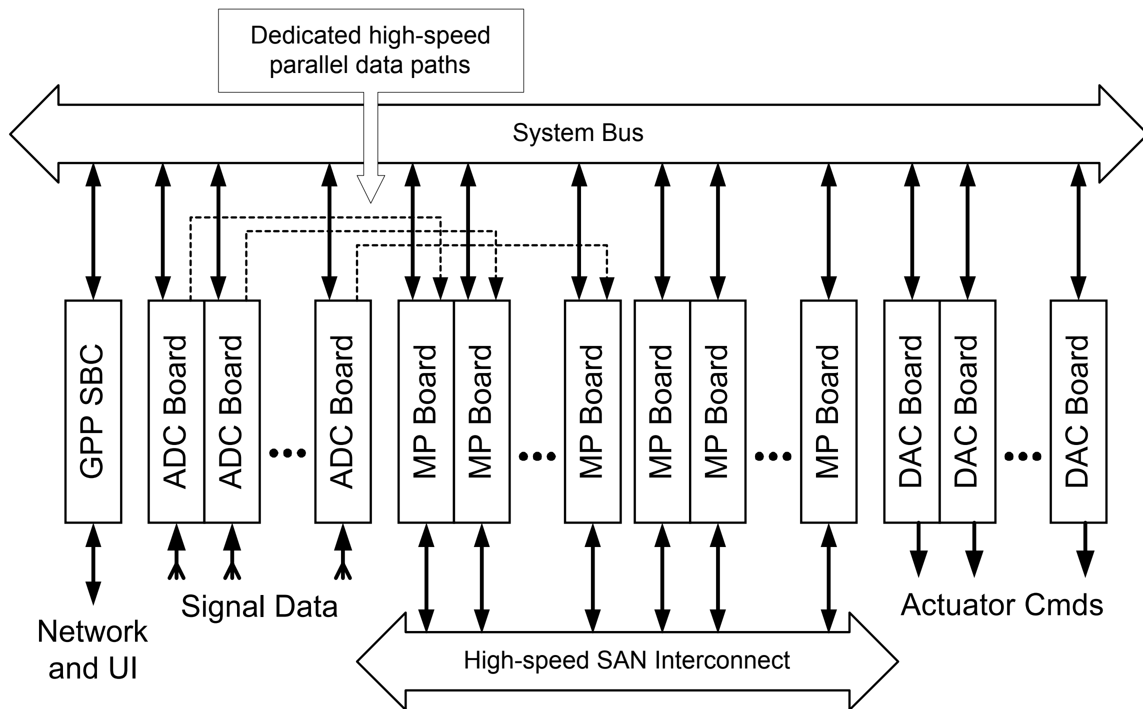

Figure 2-3. Potential architecture to implement The Processor of Figure 2-1.

2.1.2 Current Design Practice in Our Domain

The development of design methodology using COTS MP technologies in embedded real-time signal processing systems will be reviewed. This will show where existing frameworks have come from as well as identify the need for these powerful tools and where they fit into COTS MP codesign. More importantly, it will show that there is much left to do in developing a coherent specification and design methodology for this application and technology domain, our “application domain of interest,” or ADoI.

2.1.2.1 Recent and Ongoing Engineering Practice

Before DARPA commissioned the RASSP (“rapid prototyping of application specific signal processors”) program, the basic methodology had been–and to a large degree continues to be–to sketch a block diagram of an architecture to meet some specification written in a syntactically and semantically ambiguous language such as English. Throughput requirements are divided by the FLOPS/s or MIPS rating to

determine the number of processors (floating or fixed point, respectively) required, then adjusted depending on interprocessor bandwidths and margins desired. Hardware design and development proceed in parallel with algorithm specification and coding. Integration will then *hopefully* proceed without too much reiteration of hardware or software design. This ad hoc heuristic methodology will only deliver a reasonable product if the processor count is low.

The push for requiring COTS technology grew steadily in the late 1980s to early 1990s. DARPA responded by commissioning the RASSP program in the summer of 1993 in an attempt to formalize and automate specification and design shortcomings such as those introduced in the hypothetical scenario in §1.1 and discussed at greater length in §2.1.2.2–§2.1.2.3. A number of elemental methodological improvements, as well as CASE tools for implementation, have come from this program, but systemic process improvement is still needed [3].

2.1.2.2 Traditional Specification and Design

The basic traditional design flow is to first clearly specify system requirements. This leads to system-level design exploration and architecture, which is then implemented in the detailed design. The level of detail is lowest at the beginning of this flow, and it is here that making changes is the least expensive. A good design methodology will catch errors early in the cycle. This is best supported by being able to pass a model (preferably executable) from the requirements phase to the design phase, then on to the implementation phase. This allows information gained from each phase to be included in the model, so that it is not lost. Such information is fed back to preceding phases ("back-annotation") and fed forward in following design phases. This is called "model continuity." It does not currently exist in the real-time embedded COTS MP design domain.

Specification typically consists of a document written in English that details data rates and algorithms. This document is the system requirements specification (SRS). Sometimes the algorithms are provided in pseudocode, which is often in the MATLAB

language. Automation or CASE support for this phase is seldom used; consequently, the SRS cannot be executed to detect conflicts or errors.

Design of a multiprocessor system typically comes from coarsely estimating the number of processors needed and estimating interprocessor bandwidth. The designer then peruses data sheets to select vendors for multiprocessing boards, host computer, and RTOS. A software development specification (SDS) is drawn up, depending upon the hardware target to be employed. Again, without a formal methodology and accompanying tools, the SDS is verified against the SRS by hand.

Implementation consists of translating the pseudocode into algorithms, typically using hand-optimized algorithm libraries from the vendor. Data flow is sketched out and then hardcoded into the C code with vendor-specific communication library routines. The system configuration is hardcoded into the C code and header files. The design is then written using rudimentary tools such as a C compiler and source-level debugger (textual and maybe graphical) or in some cases just "printf's". A minority of developers will have access to system analysis tools to observe the data flow. Verification of the prototype or production code against the SDS is difficult in the least.

2.1.2.3 Maturation of COTS MP Codesign Methodology

Evolution beyond this traditional design process has been underway as the use of COTS technologies has become the rule rather than the exception. The integration of hardware and software design has also evolved into a process called "codesign" to emphasize the interrelation of these two design processes into a single process.

Currently, analysis of algorithm throughput and I/O data rate requirements lead to early selection of a vendor's embedded real-time MP DSP target. Accompanying such a target is the vendor's own proprietary real-time kernel running on each processor, as well as a software development environment (SDE) that usually includes an ANSI C compiler along with vendor-proprietary computation and communication libraries. Consequently, the codesign problem becomes one of "layering the application (software) on the target (hardware)" as illustrated in Figure 2-4.

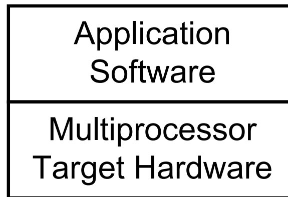

Figure 2-4. Fundamental perspective of layering software on hardware.

This simplified figure can be expanded to show in more detail what is included in these two layers, as shown in Figure 2-5.

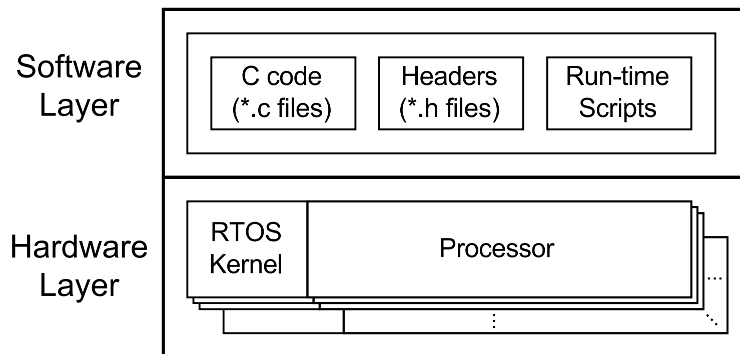

Figure 2-5. Expanded model of canonical MP hardware/software codesign layers.

This diagram should emphasize that it is the software portion of the system that is most flexible–and where the lion's share of design and development lies. While the hardware configuration is under the developer's control, once the target hardware is defined and selected, the arduous task is that of developing the software processes, synchronizing those processes and data movement, configuring the hardware, distributing the images at run-time, and finally ensuring that the real-time requirements are satisfied. In many recent and existing design environments, the model of Figure 2-6 is used.

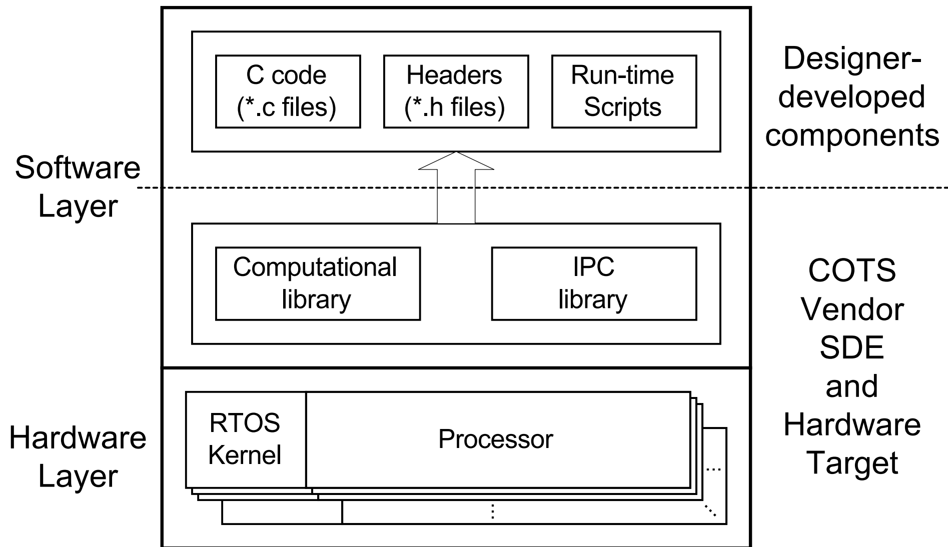

Figure 2-6. Canonical model as implemented with recent and current COTS vendor MPs.

This is a simple and primitive environment. All hardware configuration lies in the C code and header files, making it extremely difficult to develop, maintain, and extend. Computational performance is obtained through the use of optimized (typically hand-assembled) algorithm libraries. Maximal use of the vendor's high-speed network is achieved through the use of a vendor-specific interprocessor communication (IPC) library. Both of these libraries tend to have vendor-specific application programming interfaces (API's), meaning that software designed and developed for a given vendor's hardware target is *not* portable. More importantly the application software is not readily interoperable with other frameworks or vendor platforms unless the implementation engineers are familiar with the original vendor's unique API. This lack of interoperability reduces model continuity.

The development of COTS MP codesign tools in the last few years has led to the implementation frameworks described in §2.1.4.2. The codesign model then becomes as shown in Figure 2-7. Significant in this model is that the configuration model is lifted out of the C code and header files and captured by the model through a GUI and/or text files. This model is used by the framework to generate IPC-specific data movement and process synchronization code that is embedded in the application executable images for

deployment on the hardware target. Optimization is achieved through the framework's use of the vendor's optimized computational and communication libraries. MATLAB code is either compiled or translated by hand into vendor computational calls using their vendor-specific C language API. Portability is achieved *iff* the framework supports an alternate vendor's hardware target and SDE with libraries.

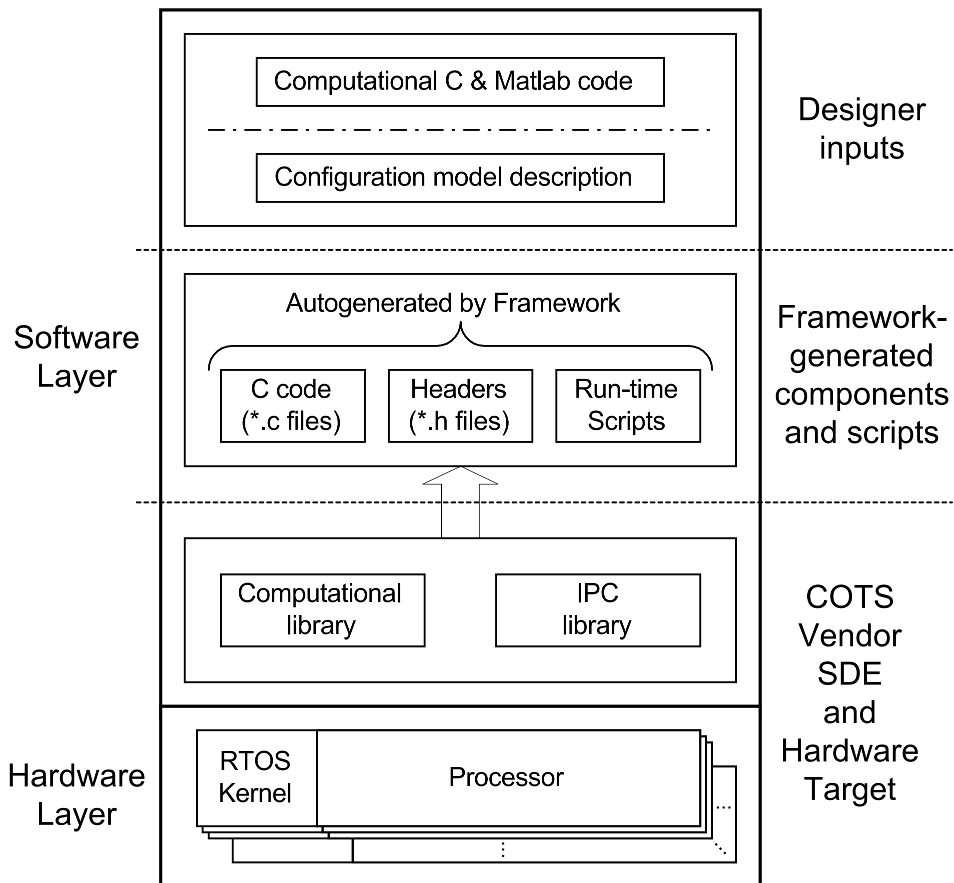

Figure 2-7. Framework codesign model decoupling computation from configuration.

Recent efforts have been underway to standardize the computational and communication libraries in order to make software more portable. With support from DARPA, the Navy, and several other academic and commercial organizations (including vendors), the Vector/Signal/Image Processing Library (VSIPL™) Forum has developed a standard object-based computational API primarily for real-time embedded MP COTS vendors. These vendors are active participants in the VSIPL Forum and momentum is

behind this computational middleware becoming the standard for embedded real-time COTS MP computation [15, 16].

The de facto standard for message passing in the parallel programming community has been the Message Passing Interface (MPI) [18, 46, 47]. Recent high-performance implementations of MPI for some COTS MP hardware targets (utilizing the vendors' own high-performance IPC libraries) have made it a viable soft real-time standard [48]. Also, an effort complementary to VSIPL has led to developing a hard real-time implementation of MPI, known as MPI/RT [17, 20, 49, 50]. MPI/RT could become a standard middleware for communication and control for real-time COTS MP systems.

Specification and design methodological frameworks would greatly benefit by middleware for computation and communication being established as a real standard. Becoming a real standard means being adopted by the whole community of both users and vendors, not just the sanction of official bodies such as ANSI and/or IEEE. Given standardized middleware for computation and communication, target code generated by an implementation tool would be readily ported from one COTS vendor's target to another's. The non-framework design model version (Figure 2-6) is illustrated in Figure 2-8 while the framework design model version (Figure 2-7) is illustrated in Figure 2-9.

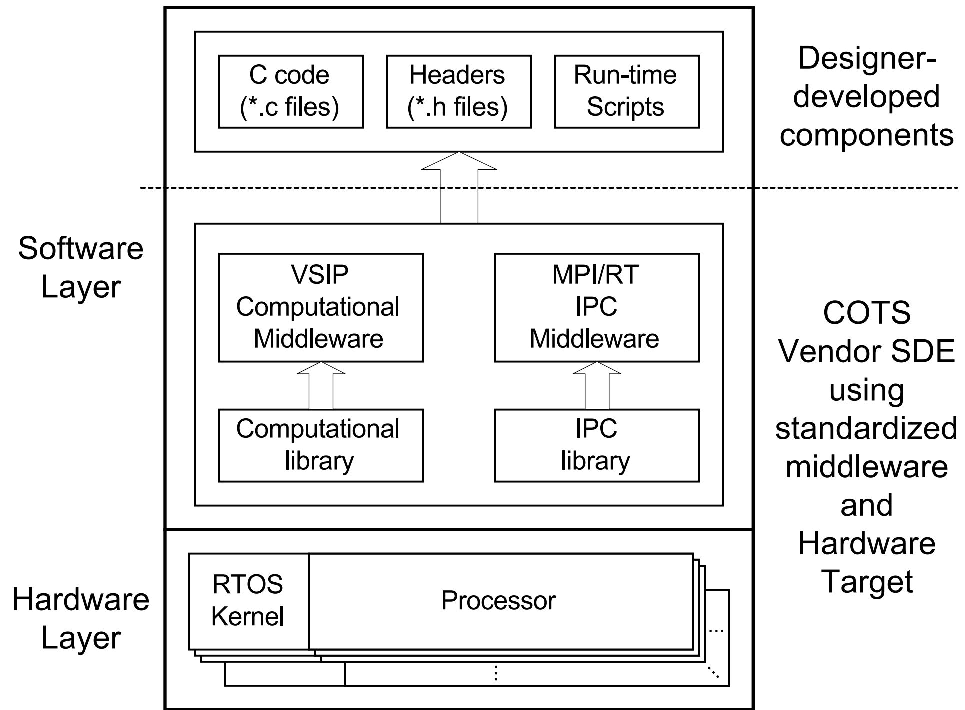

Figure 2-8. Model of Figure 2-6 utilizing standardized middleware.

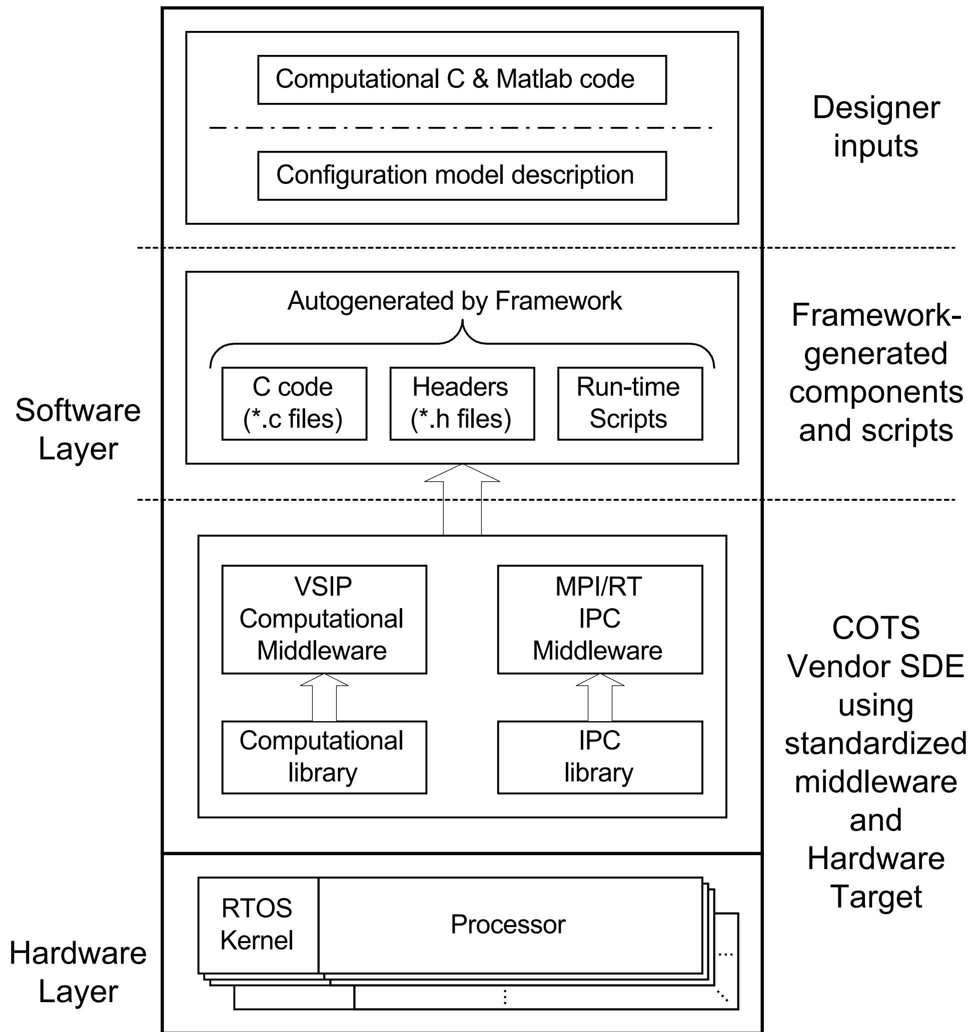

Figure 2-9. Model of Figure 2-7 utilizing standardized middleware.

2.1.3 Specification and Design Methodology (SDM)

The hardware/software codesign problem under consideration actually has very little "hardware design" since the hardware is COTS. There is a challenging system problem with which to grapple, including the specification, design, development, and verification. Though complexity is reduced by using COTS multiprocessing hardware, there are still fundamental methodological issues, such as specification expression and encapsulation, exploring and optimizing the hardware options and configuration, developing robust and optimum software, and verification. There is a large software engineering component of

this codesign whose complexity must be managed by imposing on it the discipline that nature imposes on the hardware engineering process [51, 52]. It is therefore desired to apply a rigorous methodological approach to both the specification as well as the design and development of these large digital signal processing systems.

The design process of the system of interest can be broken down into three fundamental phases:

1) *Specification*. Formulate and document the requirements and constraints of the intended system into a behavioral representation.

2) *Design*. Consider and evaluate possible implementation strategies for achieving the behavioral specification, then decide on a structural representation of hardware and mapping of algorithms and data flow to processors.

3) *Implementation*. Translate the chosen design into a physical representation of hardware and software. This includes writing the computational and communication software, creating configuration files, and writing the run-time scripts for loading and running the executable images.

How these steps are accomplished is defined by a "methodology," which until recently would depend on whether it was hardware, software, or a combination of the two.

Simply put, a methodology consists of "tools and rules." It is supported by a toolbox containing a variety of tools to help the system designer do his work, all the way from requirements specification through design and on into implementation and production [25]. The set of specific tasks, the particular order in which they are executed, and a set of computer-aided design (CAD) tools to be used during the execution of each task forms a methodology [22]. More specifically, a methodology is a coherent set of methods and tools to develop, maintain, and analyze a system at a given stage in its specification and/or design [4]. The tools are important, but only insofar as they can adequately support the relevant method. Conversely, a method or methodology without good tools is not very useful. This research is concerned with both. There has been a growing academic interest in methodologies for this application domain [22, 25, 53-56] complemented by industry's CAD tool development to assist in the design and development of such systems (surveyed in §2.1.4). It can be seen that there is a strong

interrelationship between methodology and tool(s): Methodologies infer tool requirements, and tools, at least in part, implicitly infer methodologies.

In designing such complex systems, achieving correct functionality is far more important–and difficult–than minimizing board count or program-memory size [21, 22]. Clear and correct encapsulation of requirements in the specification phase has been clearly shown to drastically reduce design and development errors. Also, the earlier such errors are detected, the less expensive they are to correct [22, 24, 25, 57-59].

A good specification methodology will effectively capture the requirements of the system, preferably in a format that is clearly understood by both the specifier and designer. A language and CAD tool that can process that language provides the best environment in which to verify that requirements are not in conflict with one another and that those requirements are indeed what the specifier desires. Satisfying this element of the methodology should naturally lead to a complementary design environment where the specification can be more directly (thus less erroneously) translated into an implementation.

Since system design is the process of implementing functionality in hardware and software, this functionality should be clearly and unambiguously defined. The natural way to achieve the needed precision is to think of the system as an integration of simpler subsystems. There are a number of methods to do this; these methods are called *specification models*. A model is a formal system consisting of objects and rules used to describe a system's characteristics, and its purpose is to provide an abstracted view of the system. Useful models will possess the following qualities:

1) *Formal*–They should contain no ambiguity and be machine executable and manipulable.
2) *Complete*–They should describe the whole system.
3) *Comprehensible*–They should be readily accessible to both specifiers and designers.
4) *Modifiable*–They should be easy to modify since change is inevitable to requirements and design.

5) *Natural aid*–They should help and not hinder the designer's understanding of the system.

Given a verified set of requirements, design of the system can then proceed to transforming the specification model into an architecture, which serves to define the model's structural implementation. Specification models and design architectures are conceptual and implementation views, respectively. Specification models describe what the system is to do, while design architectures describe how it will be built. The transformation from specification model to design architecture and implementation constitutes the design process. Ideally, the specification can be translated into the design implementation space using a framework of CAD tools, which can be searched for the best implementation. This process constitutes a system design methodology and involves system partitioning, design quality estimation, and specification refinement.

2.1.4 Implementation Frameworks

Methodologies for large software development projects have been developed over the last twenty years (see Chapter 7 of [25] for a survey) and much has been written recently on the integration of hardware and software design ("codesign") of embedded digital systems. However, most of these embedded systems are of smaller scale, e.g., controller or telecommunications applications requiring at most a single processor and perhaps programmable logic, and the associated codesign methodologies reflect this [24, 55, 56, 58, 60-71]. There have been other investigations into codesign methodology for the complex COTS multiprocessor embedded signal processing systems of interest in this paper. Most notable of these has been the DARPA RASSP program [3, 6, 8, 11, 63, 72-77].

Driven by the market and by well-funded DARPA efforts, a number of design tool frameworks have begun to evolve over the last few years that have a spectrum of effectiveness in specifying and designing hardware/software signal processing systems using COTS MP hardware. These are surveyed in the following sections (§2.1.4.1–§2.1.4.2) by giving an overview of the tool and then describing the specification and design methodologies inferred and supported. Many of the observations and comments

made in the survey are not reported in the literature, but come from our own personal experience with certain frameworks including MATLAB-related tools, RIPPEN, Talaris, and PeakWare for RACE.

2.1.4.1 Frameworks Generating Non-deployable Software

Non-deployable frameworks are worth noting because of their high profile and usefulness in researching this area. They do not generally provide for generating real-time deployable target software, but they do provide a rich environment for the specification and design of signal and image processing systems. Ptolemy is a research framework and very flexible, but typically does not generate deployable code for COTS MP targets, but rather for smaller embedded systems. Khoros's niche is developing application software for parallel and distributed application software on large parallel machines or networks of workstations, as well as providing a groupware framework for a team of developers. RTExpress can generate autocode for COTS MP targets, but it is not high quality code for a real-time embedded target. The DSP Workstation can generate autocoding for real-time uniprocessor targets but not COTS MP targets. These frameworks are illustrative of specification and design methodologies for this application domain. The Ptolemy and MATLAB-related frameworks are discussed in further detail below.

Ptolemy

Ptolemy is a software environment developed at the University of California at Berkeley. Since it is a flexible academic research framework, it is not as stable as a commercially supported design framework. Also, it does not support generating high-performance communication source code. It is a powerful research framework that supports heterogeneous system simulation and design using several different models of computation, each implemented in a separate domain. The core of Ptolemy is a software infrastructure ("the Ptolemy kernel") upon which specialized design environments ("domains") can be built. Domains can operate in one of two modes:

- *Simulation*–A scheduler invokes code segments in an order appropriate to the model of computation.
- *Code generation*–Code segments in an arbitrary language are stitched together to produce one or more programs that implement the specified function.

The kernel is made up of a family of C++ classes. It is this use of object-oriented (OO) technology that permits domains to interact with one another with their internals encapsulated. Ptolemy also supports heterogeneity, which when combined with the OO nature of Ptolemy, provides a rich research laboratory to test and explore multiple design methodologies. It now supports (among other extensions in v0.7) data flow oriented graphical programming for signal processing and synthesis environments for embedded software [78]. Mixing discrete-event models with data flow has been modeled with Ptolemy [79], an important capability in describing the reactive controller and data transformation paradigm typical with COTS MP signal processing systems.

One application of Ptolemy in the COTS MP domain was to use the Ptolemy kernel to develop an architectural trade tool. This tool gives the user an easy way to map algorithmic functions onto an architecture of COTS RISC and DSP MPs and simulate its performance for a quickly measured estimate of design quality [14]. Ptolemy has also been used to automate code generation and executable image creation using a graphical interface [80]. Another effort has been to use Ptolemy to cosynthesize data flow and control flow for a COTS MP target [81]. These successes illustrate what a framework can do, but closer investigation of these applications of Ptolemy has shown that despite possessing a powerful framework, model continuity is still lacking.

MATLAB-related

MATLAB is a commercial product and framework from The MathWorks. It is the de facto lingua franca of signal processing algorithm developers, and is ubiquitous in the signal processing community. Two recent developments make MATLAB of interest in this discussion. One is the development of the MATLAB Compiler, that metacompiles the MATLAB code (“m-files”) into C code for which there exist two libraries, one for workstations and one for the hardware of a COTS MP vendor. This means that MATLAB

can be used to capture the functional (algorithmic) requirements. A DARPA program required The MathWorks and Mercury Computer Systems (the COTS MP industry leader) to integrate a framework that could also provide the ability to capture behavioral specifications, then generate and compile the code necessary to deploy in an embedded MP target [32, 82].

The other development is the addition of Stateflow (a Statecharts [83, 84] variant) and a code generator to Simulink and the Real-Time Workshop (RTW), respectively. Simulink is a tool originally designed for simulation of MATLAB-defined signal processing algorithms that at latest revision is a respectable rapid prototyping tool for DSP. RTW is effectively a C code generation framework for Simulink. These three frameworks together are referred to as the DSP Workstation (DSPW). This framework does not yet support large COTS MP targets (though it does support smaller scale single board multiple-DSP targets), but it is converging on COTS MP technology and therefore deserves close attention [85].

An interesting related effort is a DARPA-funded framework under development known as RTExpress, which is most aggressively seeking to use MATLAB as a specification language. It uses source MATLAB code and the MATLAB Compiler along with MPI to rapidly prototype the application software to run on a real-time embedded multiprocessing target. These targets include multiple COTS MP and parallel processing workstation technologies [86].

2.1.4.2 Frameworks Generating Deployable Software

The following frameworks are of the greatest interest since they are capable of generating high quality real-time application code for COTS MP targets by leveraging the communication and computational libraries of the hardware vendors. They each support multiple targets, i.e., COTS MPs from at least two different vendors. They are actually implementation frameworks used in the design process after requirements specification and early design exploration. They have powerful code generation capabilities, thus greatly speeding up the implementation iteration phase. These tools also better support software reuse by adding to built-in libraries or building up custom libraries.

RIPPEN

Orincon is a research and development company that has fielded a number of multiprocessor-based real-time embedded sonar systems for the Navy and for DARPA. They have gone on to embed their domain knowledge in a framework known as RIPPEN (which stands for Real-time Interactive Programming and Processing Environment). It is a graphical programming environment for developing signal processing systems and supports a number of workstation and embedded COTS MP platforms [87].

RIPPEN supplies a library of software building blocks ("processing tools") that can be linked together in a dataflow diagram (DFD) to build a system. A processing tool can generate, store, modify, or fuse data. For functionality not included, custom blocks can be written, e.g., in C or C++. Orincon has also just recently added initial support for integrating MATLAB code using what they call a "MATLAB Bridge" [88]. Processing blocks are selected via a GUI and "connected" with the GUI. A "processing system" is a connection of processing tools. There are three processor control modes, depending on the level of control intervention by the tool required. They are in order of tool involvement: parallel, pipeline, and independent (pure data-driven). These three modes trade off throughput and ease of synchronization. The framework runs on a number of workstations and COTS MP embedded targets [89, 90].

GEDAE

Another framework of interest is GEDAE, an acronym for Graphical Entry, Distributed Application Environment. GEDAE supports both graphical software development and autocoding for execution on workstations and/or embedded COTS MP hardware. The workstation development environment hosts the framework, which supports the specification capture and system design, including mapping and code generation, compiling, and targeting of the executables. One of the powerful features is the visibility the framework provides into how the application is running on the target of choice.

Specification capture is done using the Graphical Editor (GE), selecting processing functions from the library, which is mostly populated with DSP functions. The

library is extensible to allow creation of new functions or modifications of existing functions. GEDAE provides support for efficient parallel processing through the GE. Designs can be validated by simulation through the same UI used to capture requirements and construct candidate designs. The UI allows the designer visibility into both the hardware domain (execution tracing and memory usage) and the software domain (textual and visual data plotting) in rich display options. Embedded code generation is computationally optimal through GEDAE's invocation of vendor libraries. Run-time support is provided through a schedule that GEDAE constructs as well as running a GEDAE Run-Time Kernel resident on each embedded node. Designer interaction is well supported for partitioning and scheduling. It supports a number of workstations and COTS MP targets as well [91].

PGM Autocoding Toolset

The Autocoding Toolset (ACT) is a design environment developed by Management Communications and Control, Inc., (MCCI) and has reported measured productivity increases up to ten-fold [72]. MCCI is now moving ACT from a research project into a commercial product. The framework basically allows the user to capture functional requirements in a GUI that uses the Processing Graph Method, which is a mature modeling technique for describing and analyzing SDFs for signal processing applications [92-97].

The most recent version of the ACT framework supports application specification, hardware target configuration, automatic code generation using vendor-specific optimized libraries, and makefile generation. Run-time support includes a distributed run-time system that governs execution using a data-driven paradigm with efficiencies comparable to that of using a fixed schedule, and external control and data interfaces. Extensions to the framework under development include quality measurement tools for software and hardware. A graph translation tool will create partition behavior models that can be executed independent of the target. Also, a hardware/software cosimulator under development will enable the designer to estimate the quality of

designs, particularly in the data movement domain, including bottlenecks, blockages, and latencies [98-100].

Talaris & PeakWare for RACE

Developed under DARPA sponsorship by Mercury Computer Systems, Talaris is a framework for application configuration, rather than hardware/software codesign specification and design of COTS MP signal processing systems. Talaris provides an extensible framework for tools used to accomplish the codesign. Currently it is limited to Solaris (2.5) and Windows NT (4.0) workstations and Mercury embedded targets running their proprietary MC/OS real-time multicomputing operating system.

The Talaris framework is of interest for two reasons. First, it provides a way to describe and capture the specification of complex applications (100's of processes and processors). Second, it provides a middleware where a COTS MP system could be specified and designed by third party tools since it is defined by an open specification, the Application Configuration Language (ACL) [101]. The framework itself has three subsystems [102]:

- *Talaris Core*–holds the application configuration data, allows access by multiple tools, and supports dynamic editing of a configuration object.
- *Talaris editing tools*–used by the designer to view and modify configuration data.
- *Talaris target tools*–consume configuration specification data to produce application executable images.

Mercury has developed tools to operate within the Talaris framework, such as mapping software components to hardware components, and connect their ports for IPC. Once software is configured and mapped onto the hardware, a generator tool creates a "launch kit" which the launch tool can use to load and run a multicomputing application. It should be noted that the framework does not load the embedded target with a scheduler. That is up to the application to perform, which allows for very efficient processing and communication.

As a framework, it provides other tools with the middleware services needed to provide specification and design support. One such tool is PeakWare for RACE, a CASE

framework developed by a COTS MP integrator (MATRA S&I) with a lot of experience using RACE hardware. They layered it on top of Talaris and now provide a CASE framework that accommodates specification capture to a perfunctory degree, as well as partitioning and autocoding very efficient code to a great degree [31, 33, 34, 103, 104].

2.1.4.3 Summary

The basic pertinent characteristics of these frameworks are summarized in Table 2-1. We summarize the capability of the code generators (deployable or not), the user interface (graphical and/or textual), and support for MATLAB.

Table 2-1. Basic characteristics of COTS MP codesign frameworks.

Framework	Code Generation Capability		User Interface		MATLAB Support
	Non-deployable	Deployable	Textual	Graphical	
Ptolemy	✓			✓	
DSPW	✓		✓	✓	✓
RTExpress	✓		✓	✓	✓
RIPPEN		✓		✓	✓
GEDAE		✓		✓	
PGM ACT		✓		✓	
Talaris		✓	✓	✓	
PW4R		✓	✓	✓	

2.2 The Domain-Specific Problem

As seen in the preceding section, the challenges of hardware software codesign in this ADoI are many. There have certainly been some good responses to these challenges. However, methodological problems remain and will be succinctly noted here to provide a foundational context for describing our new methodology.

2.2.1 Traditional SDM

Basically, the traditional SDM is “loose” (limited rules and tools) at best and prone to error. Specifications are written in a natural language (e.g., English) that is not

executable. Tools are typically limited to compiler, debugger, and profiler. Consequently, model continuity does not exist and complexity control is minimal.

2.2.2 Unified SDM of RASSP

While the traditional SDM is loose, the RASSP-inspired SDMs are tight–or unified–to a fault. These SDMs can be grouped into two classes, which are the virtual prototyping and software CASE methodologies. Virtual prototyping is the specification and design of a digital system using a language that is executable. VHDL was the language chosen in RASSP, and while it is a good language for low-level digital system design (e.g., interface circuits and boards), it is quite unwieldy for larger more complex applications like those found in this ADoI. Simulation runtimes are painfully long, and only those activities near the beginning of the hardware initialization cycle (the first 150 ms) can be explored [2, 3]. While model continuity is strong, the methodology is poorly matched to codesigns based on COTS MP targets with primarily application software to be developed. One must go through the entire design process and commit to a vendor target long before being able to assess if any requirements are satisfied.

CASE frameworks such as GEDAE and ACT (cf. §2.1.4.2) have some model continuity and complexity control, but require the developer to commit to a hardware target *before* starting the design phase. The developer then must implement alternative architectures to see if they satisfy constraints. Granted, the automatic rapid prototyping via code generation reduces the time of these iterations vis à vis traditional code development, but it is still the reverse of how it should be, and may explain in part why the community has yet to embrace these frameworks. These frameworks are powerful and have a certain SDM capability, but they also have intrinsic limitations such as requiring premature hardware commitment that need to be overcome to truly be used to their full potential.

2.3 A Domain-Specific Solution

A methodology that overcomes the shortcomings of the above will be cognizant of what is specific to both the application and technology domains. This application

domain of vector-oriented data transformation will provide an environment that conveniently captures this type of behavioral requirements in a framework that is well matched to the algorithms and pseudocode used to describe such algorithms. The tools will support design exploration of architectures comprised of potential COTS MP technologies without biasing the designer. In a domain that is moving towards open standards middleware for computation and communication, the frameworks will be open and extensible to accommodate such adoption.

The methodology will provide the right tools at the right time, working at a level of structural granularity that is appropriate for the given phase of the design process. This means that requirements capture is done at a behavioral level, but also accounts for the non-performance constraints, which are so important in embedded system design. It also means that design alternatives will be examined before committing to a particular vendor. Only when an optimum software design, hardware design, and software-to-hardware mapping ("system integration") is determined will the detailed design be done.

These methodological rules must be supported with the right mix of tool frameworks that can be directly applied or altered to be used, each at the right time. These tools' characteristics are identified and then integrated in a manner that overcomes the two main SDM challenges of model continuity and complexity control. These tools will be identified in the following chapter by characterizing both requirements and intrinsic structure of the CASE frameworks to capture such requirements and then provide a design environment to satisfy the requirements.

Chapter 3

System Requirements and Intrinsic SDM Assessment

Existing design tools as discussed in the previous chapter are extremely powerful, especially for the system implementation phase of system partitioning and flexibly mapping and remapping software to hardware. C code generation is the key feature of most of these tools, and enables them to take advantage of specific vendors' computational and interprocessor communication (IPC) libraries to achieve maximal performance. Code generation and mapping facilities are effective next generation features in this specific application domain of hardware/software codesign. But powerful design tools that do not support a sound specification and design methodology cannot guarantee a correct implementation that satisfies system requirements.

While these tools quite effectively support the rapid prototyping of complex embedded real-time multiprocessor signal processing systems, they should also be leveraged to support sound specification-modeling to whatever degree possible. This includes the effective capture of specification requirements and translation into an appropriate system architecture. Just as a filter should be matched to an expected waveform, a CASE framework should be intrinsically well matched to the types of specification requirements to be captured as well as the COTS MP hardware and software to be used in architecture exploration. That is, the innate "model of computation" (MoC) of the framework should have strong similarities to the requirements and design components. In this chapter we carefully consider our ADoI, which will allow us to identify the types of requirements to be specified and the best models of computation (MOCs) to capture these requirements in an executable model. This will allow us to assess intrinsically how well the CASE frameworks introduced in the last chapter are matched to our ADoI.

3.1 System Requirements Specification Domains

Domain specificity is a very important consideration in any specification and design effort, especially in the development of frameworks and tools to support such methodologies [4, 105]. In this particular application domain, we prescribe three specification domains of interest: 1) computation, 2) communication and control, and 3) constraints.

Computational requirements define *what* operations the signal processor must perform on the signal data. These are the algorithms, the "number-crunching" data transformational operations. These requirements may include some or all of the following: 1) data conversion; 2) lightweight vector and/or matrix operations; and 3) heavyweight functions such as discrete Fourier transforms and linear equation solvers. Of course in any computational domain (data transformational or reactive), for a system to perform correctly, the algorithms must be correctly specified and implemented.

Communication and control requirements specify *when* the signal processor tasks must be accomplished. These are the modal requirements, which are basically the states in which the processor may be, given data results or operator inputs. These requirements may include some or all of the following: 1) mode definition, timing and transition rules; 2) data transfer specification and synchronization; and 3) exception handling. It is clear in the reactive real-time application domain that the right answer at the wrong time is wrong.

Constraint requirements are the non-functional requirements that entail *how* the signal processor interfaces to its environment. These interfaces are both systemic (i.e., within the context of the larger system the signal processor serves) and environmental (the input and output boundaries). These requirements may include one or both of the following: 1) "SWAP" (size, weight, and power) and 2) "illities" (such as testability, reliability, maintainability, etc.).

3.2 Domain-Relevant Models of Computation

The concept of a model of computation (MoC) is defined and a classification of those MOCs relevant to our ADoI is developed. The MOCs that best match the three specification domains are then discerned.

3.2.1 Definition of MoC

A language is a set of symbols, rules for combining the symbols (syntax), and rules for interpreting combinations of symbols (semantics). There are two type of semantics, denotational and operational. Denotational semantics give the meaning of a language in terms of relations. Operational semantics give meaning of a language in terms of actions taken by some abstract machine, and is typically closer to the implementation. A semantic model, or "model of computation" (MoC), underlies the language and is defined by its features. Semantic features include what relations are possible in denotational semantics and how such an abstract machine behaves in operational semantics. Other features include communication style, behavior aggregation to create more complex compositions, and how hierarchy abstracts such compositions. It should be noted that a language and a MoC are not necessarily synonymous; this will be addressed in §3.4 [27].

Lee and Sangiovanni-Vincentelli have developed a denotational basis, or a "meta model," which allows certain properties of models of computation (MOCs) relevant to embedded system specification to be compared [13]. It is very abstract, but essentially provides a useful basis for comparing MOCs. The fundamental entity is the "event," which is a value/tag pair, where tags are typically used to denote temporal behavior. A "signal" is a set of events, an aggregation that is abstract. A "process" is a relation on signals and is expressed as sets of n-tuples of signals. A particular MoC is characterized by the order it imposes on tags and the characteristics of processes in the model [27].

3.2.2 Domain-Relevant MOCs

While many MOCs exist, it is useful to develop a classification of those MOCs relevant to our ADoI. The following come primarily from Berman [106], Edwards et al.

[27], and the System Level Design Language (SLDL) Forum [107]. An overview of these MOCs follows.

3.2.2.1 Discrete Event (DE)

In the discrete-event (DE) model, events usually carry a totally ordered time stamp indicating the time at which the event occurs. A DE simulator usually maintains a global event queue that sorts events by time stamp. DE modeling can be expensive due to the time-consuming task of sorting time stamps. Consequently, DE simulation is most efficient for large systems with large, frequently idle or autonomously operating sections. The advantage of the DE approach is that only changes in the system need to be propagated rather than updating the entire system. The disadvantage is that it relies on a global notion of one or more event queues, making it difficult to map the semantic model efficiently onto specific implementations. Also, such totally ordered time requires a global clock, which is very expensive to implement in a heterogeneous multiprocessor system such as is targeted by the ADoI. Examples of industry and academic frameworks include Verilog, VHDL, Cadence's BONeS, Mil3's OpNet, and The Math Works' Simulink.

3.2.2.2 Communicating Finite State Machines (CFSMs)

Traditional finite state machines (FSMs) represent a system as a set of input symbols, a set of output symbols, a finite set of states (with a defined initial state), an output function that maps inputs and states to outputs, and a set of state transitions. They are good for modeling sequential behavior, but are impractical for representing concurrency due to the "state explosion problem." A triple exponential reduction in complexity can be achieved by applying Harel's complexity reduction of finite automata:

- Hierarchy–A state can be represented as an enclosed FSM ("or" states), compactly describing the notion of preemption which is fundamental in embedded control applications.
- Concurrency–At least two states can be active at the same time ("and" states).
- Nondeterminism–Not completely specifying functionality; which is not necessarily erroneous, but is actually a rather powerful abstraction.

One of the more common extended FSM models is called StateCharts in which different cooperating state machines are synchronized through global clocks [83, 84]. Examples of industry and academic frameworks include StateCharts (over 20 variants), CFSMs, SDL Process Networks, and The Math Works' Stateflow (integrated with Simulink).

3.2.2.3 Synchronous/Reactive (S/R) MOCs

In a synchronous MoC, all events are synchronous. This means that all signals have events with identical tags, which are totally ordered and globally available. Unlike the DE MoC, every signal in a system has an event at every clock tick. The synchronous MoC is useful for "cycle-based" simulators, where processing all events at a given clock tick constitutes a "cycle." Cycle-based models have been applied effectively at the system level in certain signal processing applications. Examples of industry and academic frameworks are found in the more general "synchronous/reactive" (S/R) MOD, which is embodied in "synchronous languages" such as Esterel, Lustre, Signal, and Argos. These languages use textual and/or graphical description techniques, and can support other MOCs by implicitly supporting dataflow (Lustre and Signal) and hierarchical FSMs (Argos) [28].

The S/R languages describe systems as a set of synchronized modules executing concurrently, which communicate through signals that are either present or not in each clock tick. The presence of a signal is considered an event, often possessing a value, which is usually as an integer. Important to note for our ADoI is that the modules are reactive in the sense that they only perform computation and produce output events in clock ticks with at least one input event.

3.2.2.4 Dataflow (DF) Process Networks

A dataflow (DF) program is specified by a directed graph where the nodes (vertices), called "actors", represent computations and the arcs (edges) represent FIFO channels. These channels queue data values, encapsulated in objects called "tokens," which are passed from the output of one node to the input of a different node. A key

requirement of the computation to be performed by an actor is that it be "functional," which means that each output value of a computation is determined solely by the input value(s) of the computation.

Each process in a dataflow graph (DFG) is decomposed into a sequence of firings, which are atomic computations. Each firing consumes and produces tokens. A major objective in many signal processing environments is to statically schedule (i.e., at compile-time) the actor firings such that an efficient interleaved implementation of the concurrent MoC is achieved. This implementation is accomplished by organizing the firings into a list for a uniprocessor target or a set of lists for a multiprocessor target. Many variants of dataflow process network MOCs have been defined to handle different types of models and mappings, typically trading expressiveness for formal properties. They include Karp and Miller's computation graphs [92], Lee and Messerschmitt's synchronous DFGs [12], Kaplan et al.'s processing graph method (PGM) [93], and Granular Lucid [108].

Of particular interest is synchronous dataflow (SDF) [12], which requires processes to consume and produce a fixed number of tokens for each firing. The SDF MoC has the useful property that a finite static schedule can always be found that will return the graph to its original state, where state in this context is defined as the number of tokens on each arc. This property allows for extremely efficient implementations [109].

Graph specification is typically graphical and hierarchical, where an actor is encapsulating another directed graph. The nodes in the graph can be language primitives or subprograms specified in a language such as C, C++, Fortran, or MATLAB. It should be noted that this is actually mixing two MOCs where dataflow serves as a coordination language for the subprograms written in an imperative language, which is to say that models of computation may be mixed if such a hybridization of the MoC is a better model. Examples of industry and academic frameworks include Ptolemy, Khoros, COSSAP, SPW, and MATLAB.

3.2.2.5 Formal Notations and Hybrids

There are languages and/or frameworks that possess a formal notation and can be used effectively in the hardware/software specification and design domain, yet do not possess a clearly definable MoC. One interesting formal notation is the one that underlies a new system-on-a-chip (SOC) specification and design framework by Improv Systems, Inc., (a spin-off of Cadence) called Notation. Its MoC can be denoted as directed-control dataflow (DCDF) [110-112]. Other codesign notations include VSPEC, which is a Larch interface language for VHDL that allows specification of non-functional performance constraints such as power consumption, etc. [113], and also Talaris and ACL (cf. §2.1.4.2).

3.2.2.6 Summary

These MOCs are listed in Table 3-1, accompanied by their acronyms and frameworks and/or languages that possess the MoC.

Table 3-1. Summary of models of computation (MoCs) with example frameworks and/or languages

Model of Computation	Acronym	Frameworks and/or Languages with MoC
Discrete Event	DE	Verilog, VHDL, Cadence's BONeS, Mil3's OpNet, and The Math Works' Simulink
Communicating Finite State Machines	CFSM	StateCharts (over 20 variants), CFSMs, SDL Process Networks, and The Math Works' Stateflow (integrated with Simulink)
Synchronous/Reactive	S/R	Esterel, Lustre, Signal, and Argos
Dataflow Process Networks	DF	Synchronous DFGs, PGM, Granular Lucid, Ptolemy, Khoros, COSSAP, SPW, and MATLAB
Formal Notations and Hybrids	None	Notation, VSPEC, Talaris, and ACL

3.3 Best MOCs for Domain-Specific Specification Axes

Different MOCs have been developed, both in domain (e.g., DF) and in variants within a single domain (e.g., SDF), to best model the system being specified and/or designed. As the specification and design of embedded RT systems has evolved, so has the number of MOCs increased, depending on different nuances that have had to be specified, modeled, and designed. Experience has shown that using the best MoC leads to

the best design. In considering the specification and design of signal processors in our ADoI, there are specific classes of attributes that have been observed that must be accounted for. These are our "three domains of specification" as discussed above in §3.1.

3.3.1 ADoI-based Simplifications

Edwards et al. [27] assert that many MOCs have been defined not just because of the immaturity of the field, but also due to fundamental differences, i.e., the best model is a function of the design. The heterogeneous nature of most embedded systems makes multiple MOCs a necessity. In fact, in the system-on-a-chip (SOC) application domain, many MOCs are built by combining three largely orthogonal aspects: sequential behavior, concurrency, and communication. Our aspects are slightly different, due to our ADoI, e.g., communication and control can be combined because the same communication techniques based on distributed shared memory are used for initiating state transition as for data movement. Also, the specification of non-performance constraints such as SWAP are critical in an embedded real-time multiprocessor-based signal processing system, and should be adequately specified and accommodated for in the design from the outset, though how this is accomplished is a framework design issue.

Investigating the appropriate MOCs for specification and design requires understanding what simplifications may be implicit, if any, in our ADoI. These simplifications should both identify the best MOCs for the different domains of specification and constrain them, making their integration simpler. We determined that these domain-specific simplifications are:

3.3.1.1 Minimally Reactive

Signal processors in our ADoI typically possess at least two states, an "outer loop" and an "inner loop." The outer loop does initialization at the beginning of the processor's mission, such as memory allocation, coefficient generation (e.g., FFT twiddle factors), and process synchronization start-up. Its execution time is seldom tightly bounded. The inner loop is the repetitive part of the processor, where the data transformation execution of the streaming signal input data must be fast enough to keep

up with the data. It is sufficient to note that the number of states is small, probably only two.

3.3.1.2 Synchronous

In DF terms, the nature of the signal processor is that the number of tokens for a firing is fixed. E.g., an actor will require at least one vector of data, and perhaps arguments about how the vector(s) should be offset and strided, as well as the length of the vector(s). While the values may not always be the same, depending on the mode of the processing, the number and types of arguments will remain the same, with their upper bounds determined a priori and memory allocated accordingly. This will impact the MoC in that dataflow MOCs can be considered synchronous, with the powerful implication being that a finite static schedule can always be found. This allows for extremely efficient implementations, and is essential for algorithm partitioning in parallel implementations.

3.3.1.3 Deterministic Memory and Process Requirements

Signal processors must be designed with a finite amount of processors and memory, requiring a priori determination of processor and memory requirements. Dynamic memory allocation and process spawning are also performance inhibiting. Most of the S/R languages are static in that they cannot allocate additional memory nor spawn additional processes during run-time, which makes them leading candidates for implementation of memory-bound and time-critical embedded applications, since their behavior can be extensively analyzed at compile time. This static property also makes a synchronous program finite-state, therefore making formal verification viable.

3.3.2 Computational MoC

The best MoC for the computational specification domain will most appropriately encapsulate algorithms in a specification, and it will also effectively support distributed and parallel processing in the system design. Potentially relevant MOCs are either S/R with a DF flavor (cf. §3.2.2.3), or, conversely, DF with a S/R flavor. However, in light of the simplifications observed in §3.3.1, the best MoC is the DF MoC domain in general,

and the SDF MoC variant in particular. It should therefore come as no surprise to consider just how many specification and design tools have been developed based on this MoC. It should also be noted that the DF MoC as related to program representation, usually should (and does for implementation reasons) support a mixed grain DF model. This means that actors may be fine (atomic) or coarse (encapsulating another SDF or imperative language-based model) [109].

3.3.3 Communication & Control MoC

The best MoC for the communication and control specification domain will be able to most adequately support the definition and verification of ordering discrete events, e.g., data rates, algorithm completion constraints, and/or data transfer events. Good candidates include the DE, CFSM, and S/R MOCs. The discrete nature of the signal processing in our ADoI makes the DE MoC good for modeling specifications and design analysis, but not for implementation since it requires a global clock for the system. For a smaller system (uniprocessor, single board), this would make sense, but for large multiprocessor signal processing systems this is expensive and impractical. The CFSM is a good candidate due to its ability to support multiple states as is present in our ADoI. The S/R is also a good candidate, especially with the static constraint. Typically such systems embed the state-based control function management to a single board computer, decoupled from the multiprocessing signal processor boards by interconnection, language, library, and operating system. So, practically speaking, a controller may be interrupt-driven, but would communicate with computational processes through shared memory. This means that either the CFSM or S/R MoC could be used effectively in specifying, designing, and implementing the controller, while the DE MoC would be limited to specification and design.

3.3.4 Constraints MoC

The best MoC for constraints specification domain is from the formal notation and hybrid domain (§3.2.2.5), since such MOCs are the only ones that are capable of expressing (and therefore specifying) non-functional parameters. Interesting semantics

showing promise include VSPEC [107, 113-115] and ViewPoints [116-119]. Implementation may be possible with the Talaris framework, either explicitly or by extension. However, the exact MoC that would be best is an open issue and under investigation.

3.4 Implicit Framework MOCs

As introduced in §2.1.4, sophisticated CASE tools have begun to appear to assist in the specification and design of large DSP systems relevant to our ADoI. Implicit in each of these tools and/or frameworks are MOCs that affect both their specification ability and their design assistance. It was previously noted (§3.2.1) that a language and its underlying MoC are not necessarily the same thing. A given MoC can lead to the implementation of more than one language; e.g., the SDF has led to the implementation of a number of the graphical and textual languages described in §3.2.2.4. Conversely, there are hardware design languages (HDLs) such as VHDL and Verilog that support more than one MoC [106, 107].

We have evaluated each of the specification and design frameworks relevant to our ADoI to ascertain their implicit MOCs. We have done this by studying the frameworks' documentation and theory of operation. We have varying degrees of personal experience with most of these frameworks through either using them or being trained in their use. These are tabulated in Table 3-2. They are called "monolithic" in that the frameworks allow the specifier and/or the designer to describe, modify, and maintain a model of the codesign within the framework throughout the specification and codesign process. This is actually a limited type of model continuity, however these frameworks usually do not allow specification models to be executed, to test the specification for requirements in accuracy and conflicts. Also, the design model must actually be implemented before the design model can be verified. And the greatest shortcoming is that the target hardware and software must be committed to *before* implementing the design model. This is premature, since the design phase is where candidate technologies should be investigated, considered, analyzed, and then decided upon.

It should also be noticed that none of the frameworks handle the "illites" or SWAP. This deficiency indicates that monolithic frameworks are currently not adequate for complete system specification, though they can be very effective in system design.

3.4.1 Ptolemy

As a large research framework, Ptolemy supports a number of MOCs. Different objects can be described at different levels of abstraction using different MOCs, then integrated hierarchically. In the computational domain, Ptolemy uses the static SDF MoC. In the communication and control domain, Ptolemy uses DF, DE, or S/R MOCs via hierarchical FSMs.

3.4.2 Khoros

The central component of the Khoros framework is its visual programming language canvas, Cantata, which dynamically schedules glyphs, dispatching them as processes. The scheduler is event-driven, not data-driven. Consequently, Khoros uses a SDF computational MoC and a S/R communication and control MoC.

3.4.3 RTExpress

RTExpress has most aggressively sought to use MATLAB as a specification language and then use it to design and implement the application software on a real-time embedded multiprocessing target. RTExpress uses a static SDF MoC for computation and implicitly uses a CFSM MoC for communication and control.

3.4.4 MATLAB DSP Workstation

The MATLAB DSP Workstation (DSPW) is an integration of the familiar MATLAB analysis environment and the maturing Simulink simulation environment, along with a state-oriented StateChart type of Simulink subset called Stateflow. The computational MoC is SDF. The Simulink simulation environment is a discrete event simulation environment, which can be used for rapid prototyping; it possesses a DE MoC. Stateflow is a StateChart variant that can be used to create state-based controllers; it uses a CFSM MoC.

3.4.5 PGM Autocoding Toolset

The Autocoding Toolset (ACT) uses a Process Graph Method (PGM) canvas for system specification and possesses dataflow MOCs for both computation (SDF) and communication and control (DF). However, preemption and priority facilities are included to meet hard real-time requirements, so the MoC for communication and control can also be characterized as S/R.

3.4.6 GEDAE

GEDAE also uses a graphical canvas for specification, and also uses a static SDF computational MoC; generating a fixed schedule after specification and partitioning is complete. It is data-driven and therefore also uses a DF MoC for communication and control. While the schedule generation process maximizes the use of static scheduling to minimize overhead, GEDAE does preserve dynamic behavior where necessary. Consequently, GEDAE also possess a S/R MoC for communication and control.

3.4.7 RIPPEN

Conceptually, RIPPEN has some strong similarities to GEDAE. It uses a graphical canvas as well for specification. This graphical specification will lead to a data-driven implementation, hence using a static SDF computational MoC. RIPPEN offers different modes for run-time; it possesses both a DF and S/R MoC for communication and control.

3.4.8 Talaris & Associated Frameworks

Talaris is a framework that works at a configuration level, not strictly at the specification level. Its MOCs are governed by the languages that are used to develop software components, which are used in the application configuration. Currently, only C is supported, though C++ is beginning to be supported as well. Libraries available for an application developer for signal processing and IPC infer that there is flexibility as to the MOCs for computation as well as for communication and control.

3.5 Comparing the Monolithic Frameworks

The monolithic frameworks and their MOCs described above are succinctly delineated below in Table 3-2. Not surprisingly, the SDF dominates the computation specification axis. Dataflow (DF), and the more general MoC of SDF, can be used for communication and control, by treating control signals as data tokens. Real-time support can be maintained by allowing enough execution margin, though this form of open loop reactivity does not necessarily deterministically guarantee correct performance. Only the MATLAB DSP Workstation supports a more rigorous communication and control MoC, but unfortunately it does not yet support autocoding for the multiprocessor targets required to implement the signal processing systems in our ADoI.

The absence of support for the constraints axis could be accounted for at least two ways. One is that MOCs do not yet exist to model this aspect of a signal processing system, which is plausible given the immaturity of this field. Another more pragmatic explanation is that it is just not that important to the frameworks in this ADoI. Still another possibility is in between these two. That is, these requirements can be specified at a system level and are therefore not nearly as difficult to account for at the design level, making the other two specification domains more critical for a specification and/or design framework to cover. They still must be covered, not necessarily at a granular level, but at the hardware/software level. We will show how the MAGIC SDM uses the optimum MoC at the right time during the specification and design process by integrating frameworks of tools possessing the optimum MoC(s).

Table 3-2. Models of computation implicit in frameworks of §2.1.4.

Framework	***Models of Computation***		
	Computation	**Communication and Control**	**Constraints**
Ptolemy	SDF	DF, DE, and/or S/R	None
Khoros	SDF	S/R	None
RTExpress	SDF	CFSM	None
MATLAB DSP Workstation	SDF	DF (MATLAB) DE (Simulink) CFSM (Stateflow)	None
PGM Autocoding Toolset	SDF	DF and/or S/R	None
GEDAE	SDF	DF and/or S/R	None
RIPPEN	SDF	DF and/or S/R	None
Talaris	SDF	DF and/or S/R	None
Optimum	***SDF***	***CFSM or S/R***	***Hybrid and/or formal notation***

Chapter 4

Quantified Extrinsic SDM Assessment

Considering the intrinsic structure underlying a CASE framework being used as an SDM is valuable, but it unfortunately lacks precision. We develop a quantitative basis to analytically determine an ideal SDM and evaluate how a SDM CASE framework measures up against that ideal in this chapter. We then use it to characterize the deployable SDM CASE frameworks from §2.1.4.2. Sarkar [4] has produced the only well-delineated means by which to compare methodologies (e.g., SDM CASE frameworks). It is especially capable in clearly characterizing model continuity and complexity control. It is also structured in a way that lends itself well to quantification for more explicit and exacting characterization. Intended to enable a user to qualitatively compare reactive-system specification-modeling methodologies, we have been able to adapt it to quantitatively compare CASE SDMs.

4.1 A Unified Specification-Modeling Methodology Evaluation Framework

Sarkar has developed a unified basis for evaluating any specification-modeling methodology relevant to reactive system design [4], including his own proposed methodology, the Integrated Specification and Performance Modeling Environment (ISPME) [5]. Sarkar actually refers to his "unified basis" as a "unified framework," but to avoid confusion with our dominant use of "framework" (integrated suite of specification and design tools), we will refer to Sarkar's "unified framework" as a "unified basis" in the rest of this dissertation. This is quite apropos since Sarkar's unified basis is used as a means of comparison, especially with regard to three different attributes.

This unified basis is useful in evaluating potential methodologies and tools. A specification-modeling methodology is a coherent set of methods and tools to develop,

maintain, and analyze the specification (or, "specification model") of a given system. Sarkar states that a method consists of the three following items:

1) *Underlying model*–used to conceptualize and comprehend the system requirements and/or design.
2) *Set of languages*–provides notations to express the system requirements and/or design.
3) *Set of techniques*–needed to develop complete specification from preliminary concepts. It can be as rough as a set of loosely specified guidelines all the way up to a complete specification.

Sarkar's assertion is that the tools are important, but only to the end that they adequately implement and/or support the methods. We agree with this assertion, in that excellent CASE tools exist, but a better "method" or methodology is needed to avoid a premature technology commitment which would be very expensive to correct, as well as other specification and design errors late in the process.

Further, Sarkar shows there are three key requirements of a specification-modeling methodology, which we interpret for our methodology:

1) *Language support*. The methodology should be supported by languages that are appropriate for specifying the requirements of the system. In other words, the languages and their intrinsic models of computation should be well matched to describing the system in its domain.
2) *Complexity control*. The methodology should provide assistance in controlling the complexity of specifying the system. In other words, the specifier and designer should work only at a level of detail that is necessary to specify and design the system.
3) *Model continuity*. The methodology should support the usefulness of the specification model throughout the design and implementation phases. In other words, it is best to carry an executable specification model into the design phase, and for those two models to be carried on into the implementation phases.

The necessary attributes of Sarkar's unified specification modeling methodology basis are shown in Figure 4-1. The three major attributes are discussed in §4.1.1, which discusses the language support attribute, complexity control attribute, and model continuity attribute. These attributes are broken down into their constituent attributes and quantum sub-attributes. The branches of Figure 4-1 provide a visual aid for conveniently

organizing the sub-attributes, which are the fundamental elements that characterize a methodology.

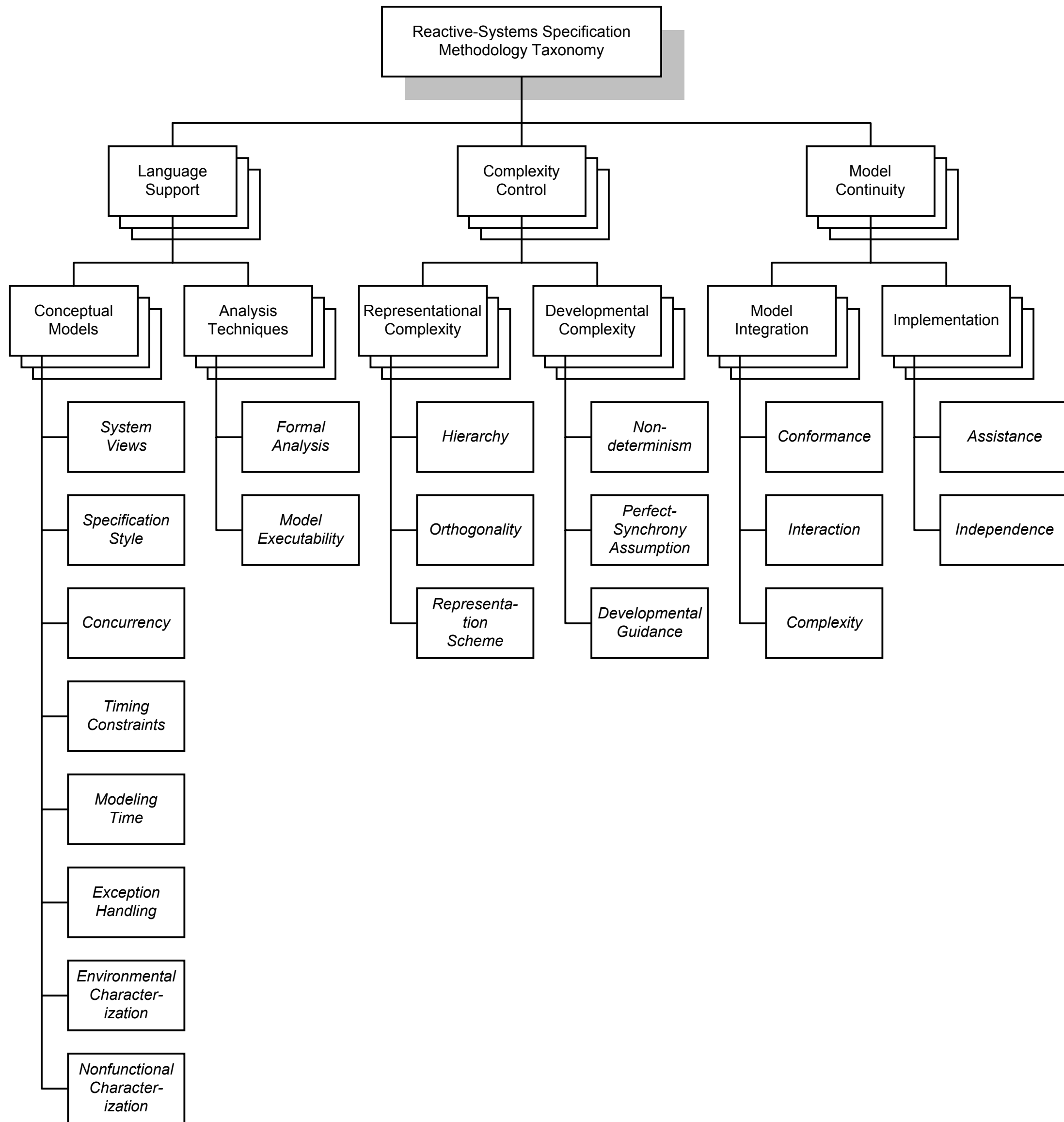

Figure 4-1. Taxonomy of Sarkar's unified reactive-systems specification methodology attributes (branches) and sub-attributes (leaves).

4.1.1 Attributes

The attributes are the branches from Figure 4-1 and are reproduced in part from Figure 4-2 below with the text in blue for ready reference.

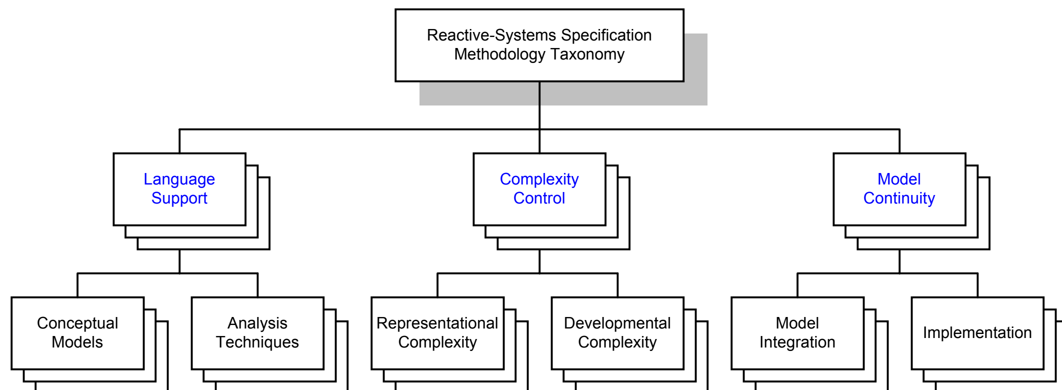

Figure 4-2. Attributes of Sarkar basis.

4.1.1.1 Language Support

The language support attribute represents the set of languages used to support the methodology. The purpose of a specification-modeling language (or set of languages) is to unambiguously express the desired functionality of the system. A specification language is defined by the conceptual models it offers the specifier to express these characteristics. A specification language typically offers just one conceptual model for a given characteristic, based on its targeted domain of application [120] [59]. In addition to providing conceptual models, the specification language should also support the facility to analyze the specification. The two most important language characteristics are that the language is based on a sound mathematical formalism and that the specification is executable [4].

4.1.1.2 Complexity Control

The complexity control attribute represents the methodology's ability to control complexity. A main requirement of any design methodology is to be able to control the complexity of the design process. There are two dimensions of complexity in non-trivial systems, representational and developmental. Representational complexity deals with developing a specification that is understandable, i.e., concise and decomposable into simpler components. Developmental complexity refers to developing the specification in an organized and productive manner, i.e., step-wise and incrementally refined.

4.1.1.3 Model Continuity

The model continuity attribute is the distinguishing attribute that makes the methodology a specification methodology as well as a design methodology. A specification modeling methodology should be more focused towards developing and maintaining a specification model instead of a proposed implementation. The significant effort involved in developing and debugging a model of the system under design will be wasted without maintaining model continuity. The relationships between models created in different model spaces must be maintained such that the models can interact in a controlled manner and may be utilized concurrently throughout the design work. Model continuity can be broken down into three different sub-problems. The first is model integration, which is to insure compatibility between the specification model and models developed during the design and implementation phases. The second is implementation assistance, which is automating the development of an implementation from a specification. The last sub-problem is implementation independence, which involves developing a specification free from implementation bias. This is important because it allows the specifier to focus on describing the behavior of the system, not its potential implementation, and leaves open the design space to the designer's creativity.

4.1.2 Sub-attributes

The sub-attributes are the quantum components that comprise the attributes. They are quantified in §4.2, but a qualitative and succinct description of the sub-attribute leaves in Figure 4-1 follows:

- System views–How the specification model is described, e.g., control-flow diagram, dataflow diagram, or datatype definitions.
- Specification style–Model-oriented (e.g., state-machines) or property-oriented (e.g., black box description).
- Concurrency–How concurrent behaviors communicate and synchronize.
- Timing constraints–Specified directly (data rates, execution throughput, etc.) or indirectly (implied through language constructs, e.g., Statecharts).
- Modeling time–Able to explicitly specify time in a metric (e.g., seconds) or not.

- Exception handling–Specified textually or graphically, if at all.
- Environmental characterization–How the interface(s) to the system's environment is modeled, either with an explicit model (perhaps using the same language as the specification) or a set of properties (e.g., frequencies, timings, etc.).
- Nonfunctional characterization–How the non-functional constraints (SWAP and "illities," e.g., reliability, maintainability, testability etc.) are expressed.
- Formal analysis–The degree (if any) to which the specification can be analyzed formally (e.g., finite-state machines, PGM, process algebras, etc.).
- Model executability–The degree to which the specification model can be executed.
- Hierarchy–Ability to compose and decompose multiple levels of abstraction.
- Orthogonality–Ability to describe two behaviors independently of each other.
- Representation scheme–Graphical and/or textual description of model.
- Nondeterminism–Ability to defer complete specification until necessary, including being able to detect and resolve nondeterminism.
- Perfect-synchrony assumption–When system reactivity bandwidth is much higher than input bandwidth; e.g., SAR image processing time is less than the time it takes to acquire a frame. This allows for a more concise specification.
- Developmental guidance–How next step in specification and/or design is determined, such as top-down, bottom-up, or middle-out.
- Conformance–How well different models are checked against one another.
- Interaction–Maintaining visibility of the specification model during design and implementation, such that back annotation can be supported.
- Complexity–How well system details are kept commensurate with the point in the specification and design process, reflected in the support of hierarchical tools.
- Assistance–Synthesis capability of tool frameworks, e.g., efficient code generation using high performance middleware.
- Independence–Measure of being devoid of implementation bias, which occurs when specifying externally unobservable properties of the system under specification, thus restricting the designer. An example is specifying a certain interprocessor bandwidth which "hardwires" a backplane technology (RACEway, SKYchannel, or Myrinet) *before* the design process has adequately explored potential architectures.

4.1.3 Metrics

We quantitatively extract our metrics from Sarkar's methodological basis by letting each attribute become a feature vector (similar to the multi-axis taxonomy of the RASSP methodology [11] [121]), each orthogonal to the others (similar to the Design Cube of Ecker et al. [122]). The magnitude of each vector can assume a discrete value depending on the scope of the attribute. As an example, the system view attribute has a scope of three since three different types of views, activity, behavior, and entity can describe it. If a certain methodology only provides dataflow diagrams (activity view), then the system view would be worth a value of "one" out of a possible "three," assuming each view was worth "one" (equal weighting).

Other attributes have a less obvious quantification; e.g., the hierarchy attribute in the complexity control attributes group is "limited" (the specification model cannot be readily decomposed) or "supported" (the specification model can be decomposed in multiple ways). This attribute may also have a null value, as with the Software Requirements Engineering Methodology (SREM).

We will now quantify each of the sub-attributes (e.g., system views), then form the three aggregate attribute values for language support, complexity control, and model continuity. Having quantified the three attributes, we will show how to use them as a 3-tuple to view a methodology in attribute space.

4.2 Quantification of Sarkar Basis

It is a challenge to assess a framework, especially quantitatively. Possessing Sarkar's basis is a starting point, but to then convert it from a qualitative basis to an analytical basis requires an approach that is rational and also usable. The software engineering domain has experienced the same difficulty in defining quantifiable metrics. The approach we have developed of using integer values counted within a complete methodological scope is similar to Function Point (FP) counting which has established itself as a valid quantitative analysis approach [1, 123-128].

We comment now on notation and naming. Some liberty is being taken with the notions of vectors and set theory in binding an analytical quantification notation upon a

qualitative basis. Regardless, it should provide an initial vehicle that is useful for examining methodologies and frameworks more quantitatively and less subjectively and qualitatively.

Some comment on naming variables is also in order. While variable names are unique among sub-attributes within an attribute, some attributes have sub-attributes with the same variable name. An attempt was made to keep variable names simple as a single alphabetic character. The character used is underlined in the sub-attribute's description in the defining statement. Not stated explicitly in every sub-attribute description is the trivial case of zero for non-coverage of an element.

Quantifying Sarkar's unified specification modeling basis [4] involves integrating the quantification of the three attribute supports of the basis:

1) Language
2) Complexity control
3) Model-continuity

Each attribute has a set of sub-attributes, which is composed of a set of elements. Each sub-attribute is denoted by defining it as a variable:

$$A_s \equiv \text{Sub-attribute}$$

Each sub-attribute is characterized by a set of elements, each of which can assume discrete or a spectrum of values, depending on the element. Some elements are composite values, which will be discussed later. A sub-attribute could be described by a set of elements:

$$A_{s_i} \in \left\{ A_{s_1}, A_{s_2}, \ldots, A_{s_{Ni}} \right\}$$

where for example, $A_{s_1} \in \{0,1,\ldots,L\}$ if it is a discrete element or $A_{s_1} = [0, L]$ if it has a continuum of values. Most elements are discrete assuming binary values of 1 or 0, depending on whether the element is covered or not, respectively. It is a practical difficulty with this initial quantification to determine a continuum on some broader spectrum of sub-attributes. These elements will typically be quantified to a set of discrete trinary values of 2, 1, or 0, depending on whether an element has support that is full,

limited, or absent, respectively, where the limited case falls between supported and none. In the absence of hard quantifiable information, the values are assumed to be distributed uniformly and are orthogonal to one another. The sub-attribute is denoted as a vector and defined as the union of possible coverage elements:

$$\mathbf{A_s} \equiv \bigcup_i A_{s_i}$$

The magnitude of a sub-attributes vector is evaluated by summing the magnitude of its component values. The more elements a sub-attribute possesses and the greater the coverage of each element, the larger the magnitude the sub-attribute will have.

$$\begin{aligned} |\mathbf{A_s}| &\equiv \sum_i^M |A_{s_i}| \\ &\in \{0,1,2,\ldots,M\} \end{aligned}$$

For example, for the first sub-attribute discussed, the system sub-attribute of the language support attribute has three potential element coverages, which are activity, behavior, and entity. Its magnitude would be evaluated as follows:

$$\begin{aligned} \mathbf{V} &\equiv \bigcup_i V_i \\ &= V_1 \cup V_2 \cup V_3 \\ |\mathbf{V}| &\equiv \sum_i |V_i| \\ &= |V_1| + |V_2| + |V_3| \text{ where } V_i \in [0,1] \end{aligned}$$

Assume a methodology provides coverage of only modeling activity and entity elements. Then |**V**| would be evaluated as follows:

$$\begin{aligned} |\mathbf{V}| &= 1+0+1 \text{ for coverage of activity and entity elements,} \\ &\qquad\qquad \text{but not the behavior element} \\ &= 2 \\ |\mathbf{V}| &< |\mathbf{V}|_{ideal} ; |\mathbf{V}|_{ideal} = 3 \end{aligned}$$

Methodologies can be compared on an attribute basis or on a complete magnitude basis. The sub-attributes of each attribute are first quantified, then summed up into a composite attribute, leading to an integrated methodology quantification. In quantifying the sub-attributes, the elements of each sub-attribute are identified (including succinct

comments pertinent to the application domain of interest) and quantified, and then integrated into a sub-attribute vector and magnitude. The ideal value (which is usually the maximum) is also noted. This value typically agrees with Sarkar's, but comment is made as to why this is so for the application domain of interest.

4.2.1 Quantifying the Language Support Attribute

The language support attribute portion of Figure 4-1 is reproduced below in Figure 4-3 with a slight modification in layout and with the sub-attributes highlighted in blue text. They are grouped into conceptual model sub-attributes and analysis technique sub-attributes.

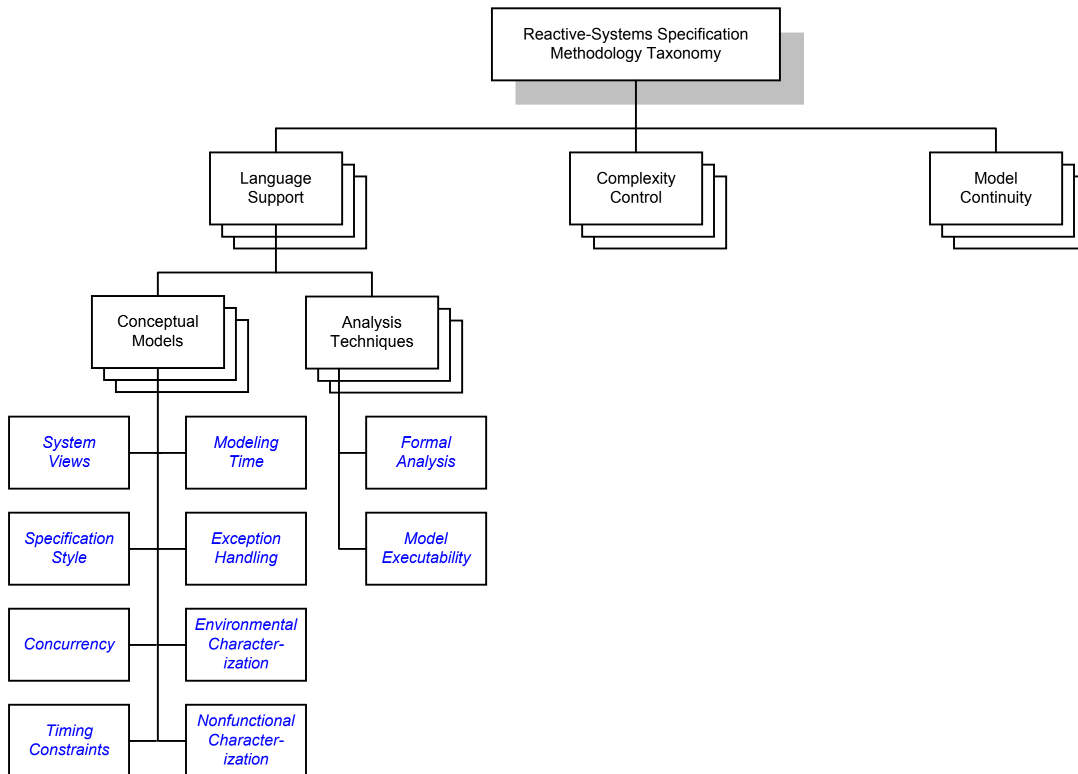

Figure 4-3. Sub-attributes of the language support attribute (in blue).

4.2.1.1 System views

There are three different yet complementary system views, denoted by the following sub-attribute elements:

1) A ≡ Activity–Data flow

2) B ≡ Behavior–Control flow

3) E ≡ Entity–Datatypes

A methodology will provide support for up to three different system views, with each view weighted equally at unity. The discrete range of values for each view is binary (0 or 1), depending on whether or not the view is provided. Ideally, a methodology will provide all three at the same time. The methodology should support at least one language and at least one view. In fact, to be useful it must support at least one language and one view. This causes a known bias, which we will accept and note. Define the system view sub-attribute as $V \equiv$ system view comprised of the element set, $V_i \in \{A,B,E\}$, then:

$$\mathbf{V} \equiv \bigcup_i V_i$$

$$|\mathbf{V}| \equiv \sum_i |V_i| \in \{0,1,2,3\} \text{ ; equal weight}$$

$$|\mathbf{V}|_{ideal} = 3 \text{ ; all three views}$$

4.2.1.2 Specification style

A methodology will have at least one of two specification styles, denoted by the following sub-attribute elements:

1) M ≡ Model-oriented–described with state-machines, processes, or sets (easier to understand)

2) P ≡ Property-oriented–described as a "black box," i.e., in terms of what is directly observable at the interface of system to its environment (less implementation-dependent).

Ideally, a methodology will use both specification styles. Define the system view sub-attribute as $S \equiv$ specification style comprised of the element set, $S_i \in \{M,P\}$, then:

$$\mathbf{S} \equiv \bigcup_i S_i$$

$$|\mathbf{S}| \equiv \sum_i |S_i| \in \{0,1,2\} \text{ ; equal weight}$$

$$|\mathbf{S}|_{ideal} = 2 \text{ ; both styles}$$

4.2.1.3 Concurrency

There are two complementary sub-attribute elements essential for concurrent behaviors to cooperate with each other:

1) C ≡ Communication–Either through shared memory buffers (SMBs) or via a message-passing paradigm.

2) S ≡ Synchronization–Either through system control statements such as fork-join or communication channels such as flags and/or semaphores using SMBs or system calls.

Ideally, a methodology will have both elements. Define C ≡ concurrency comprised of the element set, $C_i \in \{C,S\}$, then:

$$\mathbf{C} \equiv \bigcup_i C_i$$
$$|\mathbf{C}| \equiv \sum_i |C_i| \in \{0,1,2\} \text{ ; equal weight}$$
$$|\mathbf{C}|_{ideal} = 2 \text{ ; both elements}$$

4.2.1.4 Timing constraints

There are two timing constraints which are mutually exclusive, denoted by the following sub-attribute elements:

1) D ≡ Direct–Inter-event delays, data rates, execution time constraints (ideal; simple and flexible)

2) I ≡ Indirect–Implied through language constructs (e.g., Statecharts)

Since direct is preferred over indirect, direct is weighted over indirect, so we assign D=2 and I=1. Ideally, a methodology will support the direct timing constraint, but at the least support the indirect. Define the timing constraint sub-attribute as T ≡ timing constraints comprised of the element set, $T_i \in \{D,I\}$, then:

$$\mathbf{T} \equiv \bigcup_i T_i$$
$$|\mathbf{T}| \equiv \sum_i |T_i| \in \{1,2\}$$
$$|\mathbf{T}|_{ideal} = 2 \text{ ; direct}$$

4.2.1.5 Modeling time

There is one single sub-attribute element for modeling time that is binary; the methodology either does (t = 1) or does not (t = 0) support the explicit expression of time in the specification modeling. Ideally, a methodology will support modeling time. Define the timing constraint sub-attribute as $t \equiv$ modeling time comprised of the element set, $t_i \in$ {t}, then:

$$\mathbf{t} = t_i$$
$$|\mathbf{t}| = t_i \in \{0,1\} \text{ ; equal weight}$$
$$|\mathbf{t}|_{ideal} = 1 \text{ ; does model time}$$

4.2.1.6 Exception handling

There are two sub-attribute elements for a methodology's ability to describe how the system is to handle exceptions such as numerical traps, if at all. It is denoted by the following sub-attribute elements:

1. T ≡ Textual–Via a textual language such as Ada.
2. G ≡ Graphical–Via a visual environment such as Statecharts.

A methodology will support one or both. Ideally, a methodology will support both. The methodology should support at least one. Define the exception handling sub-attribute as $H \equiv$ exception handling comprised of the element set, $H_i \in$ {T,G}, then:

$$\mathbf{H} \equiv \bigcup_i H_i$$
$$|\mathbf{H}| \equiv \sum_i |H_i| \in \{0,1,2,\} \text{ ;equal weight}$$
$$|\mathbf{H}|_{ideal} = 2 \text{ ; both elements}$$

4.2.1.7 Environmental characterization

There are two sub-attribute elements for describing how the interface(s) to the system's environment is(are) modeled, either with an explicit model or a set of properties. The following denotes them:

1) M ≡ Model–Separate entity specified as model, perhaps using the same language as the specification.

2) P ≡ Property–Set of hints about operational conditions (e.g., data rates, workloads, timings, etc., as well as volume, power, heat, and weight).

A methodology will support at least one, though both are unlikely. Ideally, a methodology will support the property characterization, hence it is weighted over the model characterization, hence we assign M = 1, P = 2, and both = 3. The methodology should support at least one. Define the environmental characterization sub-attribute as E ≡ environmental characterization comprised of the element set, $E_i \in \{M,P\}$, then:

$$\mathbf{E} \equiv \bigcup_i E_i$$

$$|\mathbf{E}| \equiv \sum_i |E_i| \in \{0,1,2,3\}$$

$$|\mathbf{E}|_{ideal} = 3 \text{ ; both elements}$$

4.2.1.8 Nonfunctional characterization

There is one single sub-attribute element for describing how well a methodology covers nonfunctional characterization, such as reliability, maintainability, testability, etc. Ideally, a methodology will support as full a spectrum as possible. Define the nonfunctional characterization sub-attribute as N ≡ nonfunctional characterization comprised of the element set, $N_i \in \{0,L,E\}$, where N can assume uniform integer values for extent of coverage:

1) N = 0. None.
2) N = L = 1. Limited coverage.
3) N = E = 2. Extensive coverage (ideal).

Its vector composition, magnitude, and ideal magnitude are given below:

$$\mathbf{N} = N_i$$

$$|\mathbf{N}| = N_i \in \{0,1,2\} \text{ ; equal weight}$$

$$|\mathbf{N}|_{ideal} = 2 \text{ ; extensive support}$$

4.2.1.9 Formal analysis

The value of formal analysis is currently under debate [1, 129, 130]. It is valuable to be able to completely analyze a model formally. However, those who understand these

modeling and analysis techniques are few, and the majority is resistant, despite its potential value. Consequently, the sub-attribute element associated with a methodology's support for formal analysis will set formal support as the maximum value, with full semiformal support being just one degree less. Define the formal analysis sub-attribute as $A \equiv$ formal analysis comprised of the element set, $A_i \in \{0,L,S,F\}$. Again, in the absence of hard quantifiable attributes, a uniform distribution will be assumed and discrete values assigned, where A can assume the following uniform integer values:

1) $A = 0$. None.
2) $A = L = 1$. Limited–no formal support, but some semiformal analysis support; e.g., can analyze control and data flow diagrams (CFDs, DFDs, etc.).
3) $A = S = 2$. Supported–full semiformal analysis support, but lacks implementation independence, unambiguousness, and precision of process algebras (e.g., CSP). Includes PGM, extended FSMs as used and extended in SDL and Statecharts, and ECS and META-IV language in VDM.
4) $A = F = 3$. Formal–e.g., the process-algebra used in LOTOS.

Ideally, the methodology fully supports formal analysis techniques, then:

$$\mathbf{A} = A_i$$
$$|\mathbf{A}| \equiv A_i \in \{0,1,2,3\} \text{ ; equal weight}$$
$$|\mathbf{A}|_{ideal} = 3 \text{ ; formal analysis}$$

4.2.1.10 Model executability

There is one single sub-attribute element for describing how well a methodology can execute a specification. Ideally, a methodology will fully support model executability. Define the model executability sub-attribute as $M \equiv$ model executability comprised of the element set, $M_i \in \{0,L,S\}$, where M can assume uniform integer values for extent of coverage:

1) $M = 0$. None.
2) $M = L = 1$. Limited–Executability of the specification can be done, but within the scope of the methodology.
3) $M = S = 2$. Supported–Methodology supports direct execution of specification (ideal).

Its vector composition, magnitude, and ideal magnitude are given below:

$$\mathbf{M} = M_i$$
$$|\mathbf{M}| = M_i \in \{0,1,2\} \text{ ; equal weight}$$
$$|\mathbf{M}|_{ideal} = 2 \text{ ; supported}$$

4.2.2 Quantifying the Complexity Control Attribute

The complexity control attribute portion of Figure 4-1 is reproduced below in Figure 4-4 with the sub-attributes highlighted in blue text. They are grouped into representational complexity sub-attributes and developmental complexity sub-attributes.

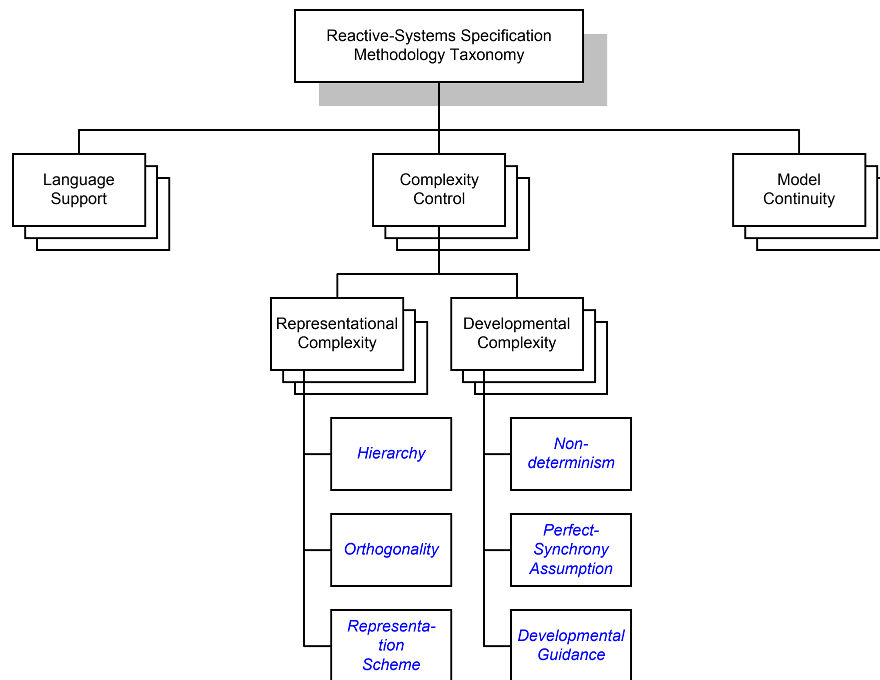

Figure 4-4. Sub-attributes of the complexity control attribute (in blue).

4.2.2.1 Hierarchy

There is one single sub-attribute element for describing how well a methodology can hierarchically decompose a specification. Ideally, a methodology will fully support specification hierarchy as described below. Define the hierarchy sub-attribute as $H \equiv$ hierarchical support comprised of the element set, $H_i \in \{0,L,S\}$, where H can assume uniform integer values for extent of coverage:

1) $H = 0$. None (unlikely).

2) $H = \text{L} = 1$. Limited–Cannot readily decompose spec.

3) $H = \text{S} = 2$. Supported–Methodology supports multiple levels of specification decomposition (ideal).

It is thus described:

$$\mathbf{H} = H_i$$
$$|\mathbf{H}| = H_i \in \{0,1,2\} \text{ ; equal weight}$$
$$|\mathbf{H}|_{ideal} = 2 \text{ ; supported}$$

4.2.2.2 Orthogonality

There is one single sub-attribute element for describing how well a methodology allows a specification's behaviors to be described "orthogonally," which means independently of one another. An example is being able to describe how one process can distribute streaming data to different processes–and describe how the receiving process is to operate on that data. Ideally, a methodology will fully support orthogonality as described below. Define the orthogonality sub-attribute as $O \equiv$ orthogonality comprised of the element set, $O_i \in \{0,\text{L},\text{S}\}$, where O can assume uniform integer values for the extent of coverage:

1) $O = 0$. None (unlikely).

2) $O = \text{L} = 1$. Limited–Cannot readily describe two behaviors independently of one another.

3) $O = \text{S} = 2$. Supported–Can describe two behaviors independently of one another (ideal).

Its quantization is below:

$$\mathbf{O} = O_i$$
$$|\mathbf{O}| = O_i \in \{0,1,2\} \text{ ; equal weight}$$
$$|\mathbf{O}|_{ideal} = 2 \text{ ; supported}$$

4.2.2.3 Representation scheme

There are two sub-attribute elements for a methodology's representation scheme, denoted by the following sub-attribute elements:

1. T ≡ Textual–Non-visual, e.g., VDM and LOTOS.
2. G ≡ Graphical–Visual formalism, e.g., Petri nets, PGM, Statecharts, etc.

A methodology will support one or both. Ideally, a methodology will support both. The methodology should support at least one; in fact, to be useful it must support at least one. Define the representation scheme sub-attribute as R ≡ representation scheme comprised of the element set, $R_i \in \{T,G\}$, then:

$$\mathbf{R} \equiv \bigcup_i R_i$$

$$|\mathbf{R}| \equiv \sum_i |R_i| \in \{1,2\}; \text{ equal weight}$$

$$|\mathbf{R}|_{ideal} = 2; \text{ both elements}$$

4.2.2.4 Nondeterminism

There is one single sub-attribute element for describing how well a methodology can accommodate nondeterminism within a specification. Ideally, a methodology will fully support expressing nondeterminism as described below. Define the nondeterminism sub-attribute as D ≡ nondeterminism support comprised of the element set, $D_i \in \{0,L,S\}$, where D can assume uniform integer values for extent of coverage:

1) $D = 0$. None (unlikely).
2) $D = L = 1$. Limited–Cannot incorporate nondeterminism into specification in a controlled manner.
3) $D = S = 2$. Supported–Can incorporate nondeterminism into specification in a controlled manner, also allowing detection and resolution of nondeterminism during specification (ideal).

Its vector composition, magnitude, and ideal magnitude are given below:

$$\mathbf{D} = D_i$$

$$|\mathbf{D}| = D_i \in \{0,1,2\}; \text{ equal weight}$$

$$|\mathbf{D}|_{ideal} = 2; \text{ supported}$$

4.2.2.5 Perfect-synchrony assumption

The perfect-synchrony hypothesis implies that a reactive system produces its outputs synchronously with its inputs, which in practical terms means that outputs are produced relatively instantaneously after the inputs occur. In the application domain under consideration, this could refer to the reactive controller part of the specification, as the data transformation part of the large DSP system is best represented as an SDF structure. Instantaneous would mean that a radar dwell frame is processed and passed along the pipeline before the next frame has arrived, which would mean that data does not pile up and double buffers do not overflow, causing data to get "dropped on the floor."

For the reliability of the reactive part of the system, this assumption must be made, recognizing that fault conditions could occur should processing "fall behind." Given the perfect synchrony assumption, specification languages can be divided into two types:

1) A = Asynchronous–Time advances implicitly as in concurrent languages such as Ada, SREM, et al.
2) S = Synchronous–Time advances iff explicitly specified, as in Statecharts.

Let the synchrony sub-attribute be defined a $S \equiv$ synchronism support, and is comprised of the element set, $S_i \in \{A,S\}$, where S can assume uniform integer values for extent of coverage. Ideally, the methodology fully supports both models, then:

$$\mathbf{S} \equiv \bigcup_i S_i$$

$$|\mathbf{S}| \equiv \sum_i |S_i| \in \{0,1,2\}$$

$$|\mathbf{S}|_{ideal} = 2\text{ ; both elements}$$

4.2.2.6 Developmental guidance

There are three design paradigms by which a methodology can guide the designer from specification through system design, denoted by the following sub-attribute elements:

1) B ≡ Bottom-up–Identify the primitives, then combine upward into subsystems, and eventually into the system.
2) T ≡ Top-down–Decompose the specification into smaller and more-detailed components downward into units which are then integrated upward into subsystems, etc.
3) M ≡ Middle-out–Combination of bottom-up and top-down.

A methodology will support up to the three different design paradigms, weighted equally at unity and are binary (0 or 1). Ideally, a methodology will support all three, and the methodology should support at least one paradigm. Define the developmental guidance sub-attribute as G ≡ developmental guidance comprised of the element set, $G_i \in \{B,T,M\}$, then:

$$\mathbf{G} \equiv \bigcup_i G_i$$

$$|\mathbf{G}| \equiv \sum_i |G_i| \in \{0,1,2,3\} \text{ ;equal weight}$$

$$|\mathbf{G}|_{ideal} = 3 \text{ ; all three design paradigms}$$

4.2.3 Quantifying the Model-Continuity Attribute

The model continuity attribute portion of Figure 4-1 is reproduced below in Figure 4-5 with the sub-attributes highlighted in blue text. They are grouped into model integration sub-attributes and implementation sub-attributes.

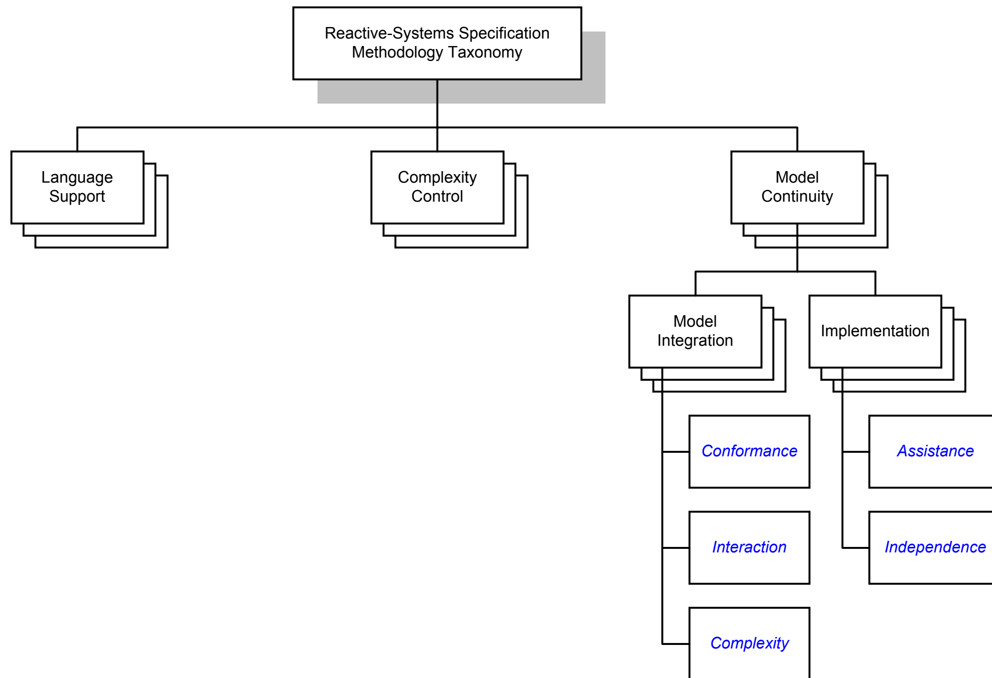

Figure 4-5. Sub-attributes of the model continuity attribute (in blue).

4.2.3.1 Conformance

There are two dimensions in which specification models will relate to one another as they are integrated:

1) V = Vertical–Different levels of abstraction; e.g., between algorithmic-level and hardware-mapping-level models.

2) H = Horizontal–Different modeling domains; e.g., between the functional-level and behavioral-level models.

This directionality applies to each of the model integration sub-attributes, which are conformance, interaction, and complexity.

The conformance sub-attribute identifies how well a methodology checks conformance among models. There are two means within a methodology which will be weighted equally (binary) and denoted by:

1) S = Simulation.

2) A = Analysis.

Sarkar has defined two levels (beyond the trivial "none") of conformance, leading to the following quantization of the elements:

3) 0 = None.
4) 1 = L. Limited–Support either vertical or horizontal conformance, but not both very well.
5) 2 = S. Supported–Provides simulation-based and/or analysis-based support model conformance in both directions (ideal).

So, define $C \equiv$ conformance sub-attribute, which is a composite of the direction and basis of the conformance checking which are defined by:

$$
\begin{aligned}
C_D &\equiv \text{Direction} \\
C_{D_i} &\in \{\text{H}, \text{V}\} \\
\mathbf{C}_D &\equiv \bigcup_i C_{D_i} \\
|\mathbf{C}_D| &\equiv \sum_i C_{D_I} \in \{0,1,2\} \text{ ; equal weight} \\
C_B &\equiv \text{Basis} \\
C_{B_i} &\in \{\text{A}, \text{S}\} \\
\mathbf{C}_B &\equiv \bigcup_i C_{B_i} \\
|\mathbf{C}_B| &\equiv \sum_i C_{B_I} \in \{0,1,2\} \text{ ; equal weight}
\end{aligned}
$$

Quantifying the aggregation of conformance direction and performance is given by the following expressions:

$$
\begin{aligned}
C &\in C_D \cup C_B \\
C_i &\in \bigcup_i C_{D_i} \cup \bigcup_i C_{B_i} \\
\mathbf{C} &\equiv \mathbf{C}_D + \mathbf{C}_B \\
|\mathbf{C}| &= \sum_i C_i \\
&= \sum_i C_{D_i} + \sum_i C_{B_i} \\
&\in \{0,1,2,3,4\} \text{ ; equal weight} \\
|\mathbf{C}|_{ideal} &= 4 \text{ ; both bases in both directions}
\end{aligned}
$$

4.2.3.2 Interaction

This sub-attribute describes how well the specification model remains visible during design and implementation, feeding back relevant systemic details back into the specification model. This requires interaction and information flow, ideally in both directions, both vertically and horizontally. Define $I \equiv$ model interaction where each direction of interaction can have one of the two following values (besides none):

1) U = Unidirectional–Information only flows in one direction between models.
2) B = Bi-directional–Information flows in both directions.

Its vector composition, magnitude, and ideal magnitude are given below:

$$I_i \in I_H \bigcup I_V$$

$$\text{where } \mathrm{I}_{\mathrm{H}_\mathrm{i}} \in \{H_U, H_B\} \text{ and } \mathrm{I}_{\mathrm{V}_\mathrm{i}} \in \{V_U, V_B\}$$

$$\mathbf{I} = \mathbf{I}_H + \mathbf{I}_V$$

$$|\mathbf{I}| = |\mathbf{I}_H| + |\mathbf{I}_V| \in \{0,1,2,3,4\}$$

$$\text{where } |\mathbf{I}_H| \in \{0,1,2\} \text{ and } |\mathbf{I}_V| \in \{0,1,2\} \text{ ; equal weight}$$

$$|\mathbf{I}|_{ideal} = 4 \text{ ; information flow is bidirectional in both model directions}$$

4.2.3.3 Complexity

This sub-attribute describes how well complexity is controlled by a methodology, which is primarily through hierarchical representations. Without complexity control the other two model integration sub-attributes are greatly weakened. Define P $\equiv$ complexity control scheme comprised of the element set, $P_i \in \{\text{H,V}\}$ where the direction complexity values assume binary values of 1 and 0, depending on whether or not the methodology allows incremental expansion in that direction, then:

$$\mathbf{P} \equiv \bigcup_i P_i$$

$$|\mathbf{P}| \equiv \sum_i |P_i| = P_V + P_H \in \{0,1,2\} \text{ ;equal weight}$$

$$|\mathbf{P}|_{ideal} = 2 \text{ ; both directions}$$

4.2.3.4 Implementation assistance

There is one single sub-attribute element for describing how well a methodology provides assistance in converting the specification into an implementation. Ideally, a methodology will fully support implementation assistance as described below. Define the implementation assistance sub-attribute as $A \equiv$ implementation assistance support comprised of the element set, $A_i \in \{0,L,S\}$, where A can assume uniform integer values for extent of coverage:

6) $A = 0$. None (unlikely).
7) $A = L = 1$. Limited–Inefficient synthesis or implementation is strictly based on specification; both lead to suboptimal implementations.
8) $A = S = 2$. Supported–Able to produce complete implementation with some degree of optimality (ideal).

Its vector composition, magnitude, and ideal magnitude are given below:

$$\mathbf{A} = A_i$$
$$|\mathbf{A}| = A_i \in \{0,1,2\} \text{ ; equal weight}$$
$$|\mathbf{A}|_{ideal} = 2 \text{ ; supported}$$

4.2.3.5 Implementation independence

There is one single sub-attribute element for describing how well a methodology avoids implementation bias, where such bias occurs if the specification methodology specifies externally unobservable properties of the system it specifies. Ideally, a methodology is implementation independent if it lacks implementation bias. Define the implementation independence sub-attribute as $N \equiv$ implementation independence which is comprised of the element set, $N_i \in \{0,L,S\}$, where N can assume uniform integer values for extent of coverage:

1) $N = 0$. None (unlikely).
2) $N = L = 1$. Limited–Specification has some measure of implementation bias, which means that some externally unobservable properties are being specified.
3) $N = S = 2$. Supported–Specification is without bias (ideal):
   a) Specifier can focus strictly on behavior (not implementation) of system.

b) Avoids placing unnecessary restrictions on designer freedom.

Its vector composition, magnitude, and ideal magnitude are given below:

$$\mathbf{N} = N_i$$
$$|\mathbf{N}| = N_i \in \{0,1,2\} \text{ ; equal weight}$$
$$|\mathbf{N}|_{ideal} = 2 \text{ ; supported}$$

4.2.4 Quantification of the Attributes and a Methodology

The quantified sub-attributes must be combined into their respective attributes. The attributes can be used to compare methodologies vis à vis the individual attributes or combined to see how the methodologies compare overall in attribute space.

4.2.4.1 Integrating Attribute Quantifications

The attributes established by Sarkar and used to develop a unified basis for evaluating reactive-system design specification-modeling methodologies is a composition of three distinct attributes. One could submit that these attributes are indeed orthogonal in that each is independent of the other and uniquely quantified with regard only to its sub-attributes. These three attributes can then be viewed in 3-tuple attribute space as shown in the Figure 4-6 below, oriented similarly to how MATLAB plots in three dimensions.

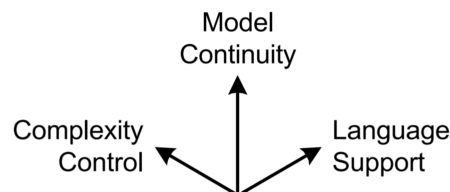

Figure 4-6. Graphical representation of Sarkar basis attributes.

The value of the axes is determined by evaluating the sub-attributes, which is essentially an integration of discrete values, i.e., summing up elemental coverage. This is described in the following integrations, beginning with the language support attribute:

$$\begin{aligned} Language_Support &= Conceptual_Models \\ &\quad + Analysis_Techniques \end{aligned}$$

where

$$Conceptual_Models = |\mathbf{V}| + |\mathbf{S}| + |\mathbf{C}| + |\mathbf{T}| + |\mathbf{t}| + |\mathbf{H}| + |\mathbf{E}| + |\mathbf{N}|$$
$$Analysis_Techniques = |\mathbf{A}| + |\mathbf{M}|$$

Therefore,

$$Language_Support = |\mathbf{V}| + |\mathbf{S}| + |\mathbf{C}| + |\mathbf{T}| + |\mathbf{t}| + |\mathbf{H}| + |\mathbf{E}| + |\mathbf{N}| + |\mathbf{A}| + |\mathbf{M}|$$

The complexity control is similarly obtained:

$$\begin{aligned} Complexity_Control &= Representational_Complexity \\ &\quad + Developmental_Complexity \end{aligned}$$
$$Representational_Complexity = |\mathbf{H}| + |\mathbf{O}| + |\mathbf{R}|$$
$$Developmental_Complexity = |\mathbf{D}| + |\mathbf{S}| + |\mathbf{G}|$$
$$\therefore Complexity_Control = |\mathbf{H}| + |\mathbf{O}| + |\mathbf{R}| + |\mathbf{D}| + |\mathbf{S}| + |\mathbf{G}|$$

And the model continuity is also obtained thusly:

$$Model_Continuity = Model_Integration + Implementation$$
$$Model_Integration = |\mathbf{C}| + |\mathbf{I}| + |\mathbf{P}|$$
$$Implementation = |\mathbf{A}| + |\mathbf{N}|$$
$$\therefore Model_Continuity = |\mathbf{C}| + |\mathbf{I}| + |\mathbf{P}| + |\mathbf{A}| + |\mathbf{N}|$$

These attributes can be computed in a manner similar to FPs, wherein the one who evaluates the methodology must have domain knowledge of both the application and the candidate methodology for the evaluation and quantification to be valuable. It is proposed that each axis be normalized to the ideal, meaning that each attribute's quantification will fall in the range [0,1], where 0 is "none" for each sub-attribute (unlikely) and 1 is ideal coverage by the methodology of each sub-attribute. Each attribute integration will be divided by the ideal integration to accomplish this normalization.

4.2.4.2 Methodology Quantification Aggregation

Once the each attribute is quantified and normalized, a methodology can be plotted in the cube's space by making a composite of these values ("aggregation") into a 3-tuple. This is conceptually similar to Gajski and Kuhn's Y-chart [131] and Ecker's design cube [122] with regard to geometry, but more closely aligned with the RASSP taxonomy [121] with regard to quantification of sub-attributes. The actual geometry is not so significant as is the relative position methodologies have to one another (between the line from (0,0,0) to (1,1,1)) and to the axes (where a methodology fails or is ideal with respect to a specific attribute). The frameworks surveyed in §2.1.4.2 have been quantifiably measured by these metrics and will be plotted.

4.3 Using Quantified Basis to Characterize CASE SDM Frameworks

Each attribute of the quantified basis is computed by analyzing the deployable CASE SDMs with regard to each sub-attribute. The DSPW is similarly computed for comparison later as part of the MAGIC SDM. An Excel spreadsheet was created to most easily document the sub-attribute quantization, using a worksheet for each attribute. A fourth worksheet was created to summarize the attribute values as well as to export the quantization into MATLAB using the Excel Link hooks.

The ideal value is included, computed from the above quantification development (§4.2). The ideal is used to normalize the individual attribute quantifications. These are plotted in 3-tuple space and in different combinations of 2-tuple space.

4.3.1 Language Support Attributes

The spreadsheet that captures the quantification of the language support attributes appears below in Table 4-1.

Table 4-1. Language support sub-attributes spreadsheet.

Sub-attribute	Sub-attribute components		Element Set	**Ideal**	DSPW	RIPPEN	ACT	GEDAE	PW4R
	Element(s)	Component							
System views	$V_i \in \{A,B,E\}$	$\lvert \mathbf{V} \rvert = A+B+E$	$\lvert \mathbf{V} \rvert \in \{0,1,2,3\}$	**3**	2	1	1	2	1
	A	Activity--data flow	{0,1}	1	1	1	1	1	1
	B	Behavior--control flow	{0,1}	1	1	0	0	1	0
	E	Entity--datatypes	{0,1}	1	0	0	0	0	0
Specification style	$S_i \in \{M,P\}$	$\lvert \mathbf{S} \rvert = M+P$	$\lvert \mathbf{S} \rvert \in \{0,1,2\}$	**2**	1	1	1	1	1
	M	Model--states, processes, or sets (easier to understand)	{0,1}	1	1	1	1	1	1
	P	Property--"black box" (less implementation-dependent)	{0,1}	1	0	0	0	0	0
Concurrency	$C_i \in \{C,S\}$	$\lvert \mathbf{C} \rvert = C+S$	$\lvert \mathbf{C} \rvert \in \{0,1,2\}$	**2**	2	2	1	2	2
	C	Communication (SMB's and/or MP paradigms)	{0,1}	1	1	1	0	1	1
	S	Synchronization (system control statements and/or comm channels)	{0,1}	1	1	1	1	1	1
Timing constraints	$T_i \in \{D,I\}$	$\lvert \mathbf{T} \rvert = D+I$	$\lvert \mathbf{T} \rvert \in \{0,1,2\}$	**2**	1	1	1	2	1
	D	Direct--inter-event delays, data rates, etc. (ideal--simple & flexible)	{0,2}	2	0	0	0	2	0
	I	Indirect--implied through lang constructs (e.g., Statecharts)	{0,1}	0	1	1	1	0	1
Modeling time	t	$\lvert \mathbf{t} \rvert = t$	$\lvert \mathbf{t} \rvert \in \{0,1\}$	**1**	1	0	0	1	0
		t=0. Does not support explicit expression of time.				0	0		0
		t=1. Does support explicit expression of time.		1	1			1	
Exeption handling	$H_i \in \{T,G\}$	$\lvert \mathbf{H} \rvert = T+G$	$\lvert \mathbf{H} \rvert \in \{0,1,2\}$	**2**	1	1	0	1	0
	T	Textual--e.g., language like Ada	{0,1}	1	0	0	0	0	0
	G	Graphical--e.g., visual environment like Statecharts	{0,1}	1	1	1	0	1	0
Environmental characterization	$E_i \in \{M,P\}$	$\lvert \mathbf{E} \rvert = M+P$	$\lvert \mathbf{E} \rvert \in \{0,1,2,3\}$	**3**	1	1	1	1	0
	M	Model--spec environment using same spec lang	{0,1}	1	1	1	1	1	0
	P	Property--set of hints about operational conditions (incl. SWAP)	{0,2}	2	0	0	0	0	0
Nonfunctional characterization	N	$\lvert \mathbf{N} \rvert = N$	$\lvert \mathbf{N} \rvert \in \{0,1,2\}$	**2**	0	0	0	0	0
		N=0=None.			0	0	0	0	0
		N=L=1. Limited coverage (only one illity).							
		N=E=2. Extensive coverage (more than one).		2					
Formal Analysis	A	$\lvert \mathbf{A} \rvert = A$	$\lvert \mathbf{A} \rvert \in \{0,1,2,3\}$	**3**	2	1	2	1	1
		A=0=None.							
		A=I_{lim}=1. Limited informal support--e.g., CFD's, DFD's, etc.				1		1	1
		A=I_{sup}=2. Full informally support--i.e., lacks implementation independence, unambiguousness, & precision of process algebras.			2		2		
		A=F=3. Formal support--e.g., process algebra like LOTOS.		3					
Model Executability	M	$\lvert \mathbf{M} \rvert = M$	$\lvert \mathbf{M} \rvert \in \{0,1,2\}$	**2**	2	1	1	2	1
		M=0=None.							
		M=L=1. Limited support--spec executable, but w/in scope of methodology.				1	1		1
		M=S=2. Supported fully; supports direct execution of spec.		2	2			2	
Total				**22**	13	9	8	13	7
Normalized Total				**1.00**	0.59	0.41	0.36	0.59	0.32

4.3.2 Complexity Control Attributes

The spreadsheet that captures the quantification of the complexity control attributes appears below in Table 4-2.

Table 4-2. Complexity Control Sub-attributes spreadsheet.

Sub-attribute	Sub-attribute components		Element Set	**Ideal**	DSPW	RIPPEN	ACT	GEDAE	PW4R
	Element(s)	Component							
Hierarchy	*H*	|**H**| =*H*	|**H**|∈{0,1,2}	**2**	2	2	1	2	1
		H =0=None.							
		H =L=1. Limited--cannot readily decompose spec.					1		1
		H =S=2. Supported--supports multiple levels of spec decomposition.		2	2	2		2	
Orthogonality	*O*	|**O**| =*O*	|**O**|∈{0,1,2}	**2**	2	2	2	2	2
		O =0=None.							
		O =L=1. Limited--cannot readily describe two behaviors independently of one another.							
		O =S=2. Supported--Can readily describe two behaviors independently of one another.		2	2	2	2	2	2
Representation	R_i∈{*T*,*G*}	|**R**| =*T*+*G*	|**R**|∈{0,1,2}	**2**	2	1	1	2	2
	T	Textual--Non-visual, e.g., ACL, Matlab, etc.	{0,1}	1	1	0	0	1	1
	G	Graphical--Visual formalism, e.g., GEDAE, ACT, RIPPEN, etc.	{0,1}	1	1	1	1	1	1
Non-determinism	*D*	|**D**| =*D*	|**D**|∈{0,1,2}	**2**	1	1	2	1	1
		D =0=None.							
		D =L=1. Limited--cannot incorporate non-determinism into spec in a controlled manner.			1	1		1	1
		D =S=2. Supported--Can incorporate non-determinism into specification in a controlled manner, also allowing detection & resolution of non-determinism during specification.		**2**			2		
Perfect-synchrony assumption	S_i∈{A,S}	|**S**| =A+S	|**S**|∈{0,1,2}	**2**	1	1	1	1	0
	A	Asynchronous--Time advances implicitly as in concurrent languages such as Ada, SREM, etc.	{0,1}	1	0	0	0	0	0
	S	Synchronous--Time advances *iff* explicitly spec'd, as in Statecharts.	{0,1}	1	1	1	1	1	0
Developmental guidance	G_i∈{*B*,T,M}	|**G**| =*B*+T+M	|**G**|∈{0,1,2,3}	**3**	3	3	2	3	3
	B	Bottom-up--Identify primitives, then combine upwards into subsystems which combine eventually into the system.	{0,1}	1	1	1	1	1	1
	T	Top-down--Decompose the spec into smaller and more-detailed components downward into components, which are then integrated upward into subsystems, etc.	{0,1}	1	1	1	1	1	1
	M	Middle-out--Combination of *B* and *T*, leveraging reuse.	{0,1}	1	1	1	0	1	1
Total				**13**	11	10	9	11	9
Normalized Total				**1.00**	0.85	0.77	0.69	0.85	0.69

4.3.3 Model Continuity Attributes

The spreadsheet that captures the quantification of the model continuity attributes appears below in Table 4-3.

Table 4-3. Model Continuity Sub-attributes spreadsheet.

Sub-attribute	Sub-attribute components		Element Set	**Ideal**	DSPW	RIPPEN	ACT	GEDAE	PW4R
	Element(s)	Component							
Conformance	$C_i \in C_H \cup C_V$	$\lvert \mathbf{C} \rvert = \lvert \mathbf{C}_H \rvert + \lvert \mathbf{C}_V \rvert$	$\lvert \mathbf{C} \rvert \in \{0,1,2,3,4\}$	**4**	4	3	3	4	1
Horizontal	$C_H \in \{A, S\}$	$\lvert \mathbf{C}_H \rvert = A + S$	$\lvert \mathbf{C}_H \rvert \in \{0,1,2\}$	**2**	2	1	1	2	0
		H=Horizontal--Different modeling domains; e.g., between the functional-level and behavioral-level models.							
	A	*A*=Analysis.	{0,1}	1	1	0	0	1	0
	S	*S*=Simulation.	{0,1}	1	1	1	1	1	0
Vertical	$C_V \in \{A, S\}$	$\lvert \mathbf{C}_V \rvert = A + S$	$\lvert \mathbf{C}_V \rvert \in \{0,1,2\}$	**2**	2	2	2	2	1
		V=Vertical--Different levels of abstraction; e.g., between algorithmic-level and hardware-mapping-level models.							
	A	*A*=Analysis.	{0,1}	1	1	1	1	1	0
	S	*S*=Simulation.	{0,1}	1	1	1	1	1	1
Interaction	$I_i \in \{I_H, I_V\}$	$\lvert \mathbf{I} \rvert = I_H + I_V$	$\lvert \mathbf{I} \rvert \in \{0,1,2,3,4\}$	**4**	4	2	2	3	1
Horizontal	I_H	I_H=0=None.				0	0		0
		I_H=U=1. Unidirectional--Information only flows in one direction between models.						1	
		I_H=B=2. Bidirectional--Information flows in both directions between models.		**2**	2				
Vertical	I_V	I_V=0=None.							
		I_V=U=1. Unidirectional--Information only flows in one direction between models.							1
		I_V=B=2. Bidirectional--Information flows in both directions between models.		**2**	2	2	2	2	
Complexity	$P_i \in \{H, V\}$	$\lvert \mathbf{P} \rvert$ =H+V	$\lvert \mathbf{P} \rvert \in \{0,1,2\}$	**2**	2	1	1	2	1
	H	*H*=Horizontal--Different modeling domains; e.g., between the functional-level and behavioral-level models.	{0,1}	1	1	0	0	1	0
	V	*V*=Vertical--Different levels of abstraction; e.g., between algorithmic-level and hardware-mapping-level models.	{0,1}	1	1	1	1	1	1
Implementation Assistance	*A*	$\lvert \mathbf{A} \rvert = A$	$\lvert \mathbf{A} \rvert \in \{0,1,2\}$	**2**	0	1	1	2	1
		A=0=None.			0				
		A=L=1. Limited--Inefficient synthesis or implementation is strictly based on specification; both lead to suboptimal implementations.				1	1		1
		A=S=2. Supported--Able to produce complete implementation with some degree of optimality.		2				2	
Implementation Independence	*N*	$\lvert \mathbf{N} \rvert = N$	$\lvert \mathbf{N} \rvert \in \{0,1,2\}$	**2**	0	1	1	1	1
		N=0=None.			0				
		N=L=1. Limited--Spec has some measure of implementation bias, i.e., specs some externally unobservable properties.				1	1	1	1
		A=S=2. Supported--Spec is w/o bias: a) Specifier can focus strictly on behavior (not implementation) of system. b) Avoids placing unnecessary restrictions on designer freedoms.		2					
Total				**14**	10	8	8	12	5
Normalized Total				**1.00**	0.71	0.57	0.57	0.86	0.36

4.3.4 Summary

The total raw values of the three SDM attributes are tabulated below in Table 4-4. Dividing them by the ideal value for each of the attributes normalizes the attributes' total raw values. These normalized values are tabulated in Table 4-5 and plotted one at a time in the bar graph of Figure 4-7. These values are also plotted in 3-tuple space (Figure 4-8) to illustrate how they compare to one another and against the ideal SDM.

Table 4-4. Raw values for attribute integrations.

Raw	**Attributes**		
	Language	Complexity Control	Model Continuity
Ideal	**22**	**13**	**14**
DSPW	13	11	10
RIPPEN	9	10	8
ACT	8	9	8
GEDAE	13	11	12
PW4R	7	9	5

Table 4-5. Normalized attribute values for the CASE SDMs.

Normalized	**Attributes**		
	Language	Complexity Control	Model Continuity
Ideal	**1.00**	**1.00**	**1.00**
DSPW	0.59	0.85	0.71
RIPPEN	0.41	0.77	0.57
ACT	0.36	0.69	0.57
GEDAE	0.59	0.85	0.86
PW4R	0.32	0.69	0.36

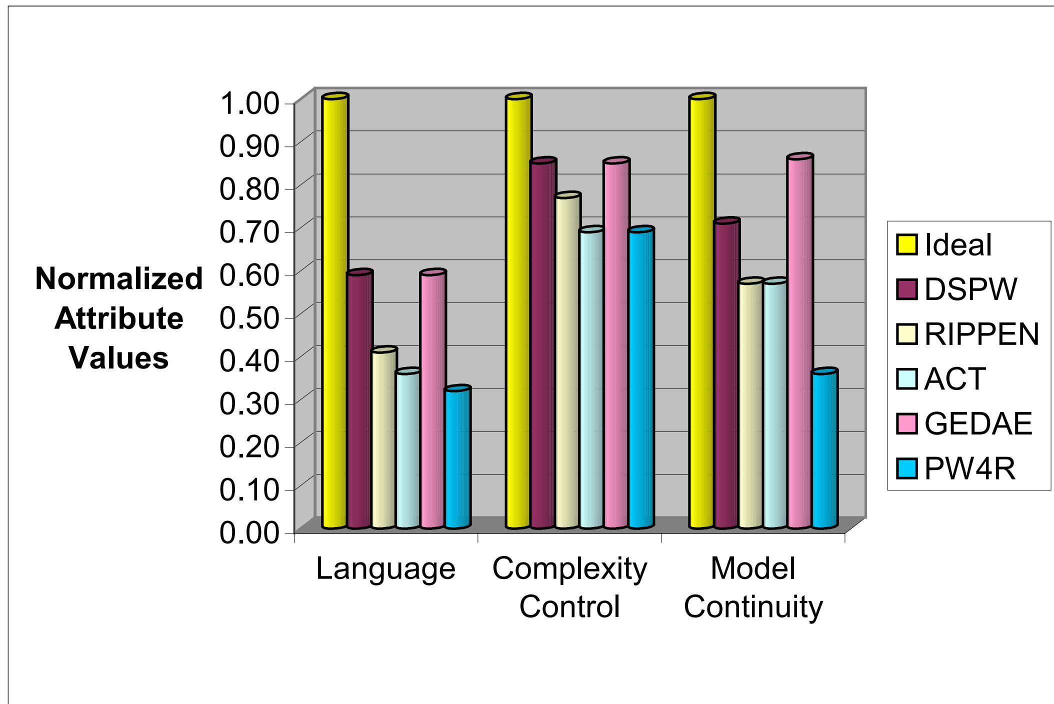

Figure 4-7. Plot of normalized attribute values for the CASE SDMs in Table 4-5.

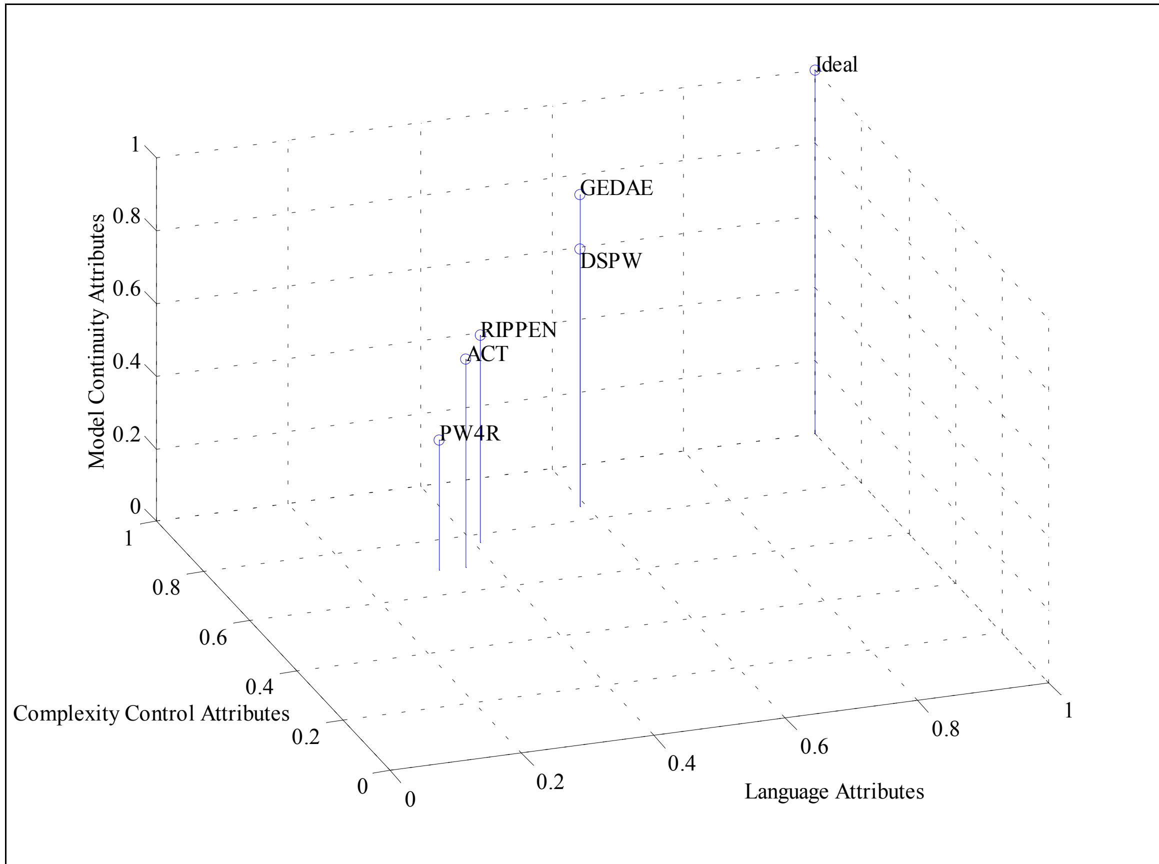

Figure 4-8. Plot of CASE SDMs and Ideal SDM in 3-tuple space.

These two figures provide a good view of how the CASE SDMs in general are strong in complexity control and fair in language support attributes, with model continuity somewhere in between. It is important to note that the model continuity attribute is an aggregate of five sub-attributes. Implementation independence is a financially and temporally (vis à vis schedule time) weighty sub-attribute, but in our initial Sarkar basis quantization it is no weightier than the other sub-attributes because we are opting to not weight sub-attributes in the absence of data to guide this. We feel that the model continuity appears better than it really is because the CASE SDMs have an implicit bias toward system implementation technologies on which they can generate implementation code. This can be costly because they require a premature acquisition of hardware and software to perform design exploration.

4.4 Conclusion

A good set of metrics should make clear the delineation between methodological attributes as to what is “good” and why. Reexamination and revision will no doubt be necessary in order to better calibrate what the sensitivity of certain sub-attributes are, how to better scale defined element set values in order to clarify methodological quality differences, among other issues. For example, we are particularly interested in model continuity, and perhaps it should be weighted. However, in this initial quantification of the Sarkar basis, we believe it is best to not weight sub-attribute variables or attribute variables, but to let them stand as they are. Weighting the sub-attributes will require empirical data to determine reasonable weights. Such empirical data comes from the basis being used. This takes time, and since we have developed the first quantification of the Sarkar basis, empirical data is absent.

The scoring scheme used in quantifying a methodology clearly infers that “more is better.” This is true because each of the sub-attributes came from assessing a wide range of reactive specification and design methodologies. This is not to say that Microsoft Word is better at editing a file with C code because it has so many more features than Multi-Edit. If we delineated the features required for a code editor then measured how well Microsoft Word and Multi-Edit measured up against those requirements, Multi-Edit would come out on top since the framework was tuned to editing code, not creating documents. Multi-Edit is a much smaller application than Microsoft Word, but it is a much better code editor. Similarly we are using a framework tuned to reactive specification and design methodologies and measuring SDMs in the COTS MP technology domain against this framework. The better they score against this framework, the better they serve as a specification and design methodology for reactive systems.

Chapter 5

Extending Gajski's SER Methodology

Two types of analysis have revealed how CASE SDMs fall short. First, implicit model of computation evaluation has shown this. Second, explicit quantification via the Sarkar basis has also shown this, and with specificity. Sarkar's basis allows us to evaluate the CASE SDMs, but primarily from a software perspective. This chapter describes a methodology from the real-time *embedded digital system design domain* by Gajski et al., which provides a very useful foundation on which to build a new SDM to overcome the shortcomings identified. This SDM is of great interest because its design objects and non-performance constraints have strong parallels with those in our ADoI. Gajski's ADoI is real-time controllers implemented with single-board uniprocessor-based systems, which will require us to extend and adapt certain key aspects of their methodology in developing our own MAGIC SDM in the next chapter.

5.1 Background

Dissatisfaction with existing reactive domain specific SDMs is what drove Sarkar to develop his own ISPME methodology, and in the process developed his unified basis for evaluating specification-modeling methodologies relevant to reactive system design. Sarkar's unified basis was specific to the domain of reactive systems, yet not necessarily embedded.

Like Sarkar, Gajski et al. also reviewed and surveyed software methodologies but have also surveyed hardware methodologies, summarizing them into two classes. This survey inspired them to develop a hardware/software codesign methodology appropriate to their application domain, along with associated system design language with supporting tools [22, 23, 58, 59, 67]. Gajski et al. had gone through a process similar to

Sarkar's, yet from a much narrower domain of small-scale reactive and embedded digital systems with at most a single processor. It is the close matching of Gajski et al.'s domain to that of COTS MP-based codesign that makes their methodology so interesting. Their methodology will be reviewed and extended to our ADoI.

A brief overview of the evolution of digital design will be presented which shows the rationale driving the methodology proposed by Gajski et al. dubbed "Specify-Explore-Refine" (SER). The first class of methodology is "capture-and-simulate," which has been in use for the past 25 years by ASIC and system houses. The system requirements are described inexactly in English, then translated into a block diagram that serves as a preliminary (and incomplete) specification. The specification is then translated into a digital design, decomposing the design into logic circuits that are eventually captured by schematic CAD tools. This encapsulation of the design is used to lay out and manufacture the design. This methodology gives way to the second class of methodology, called "describe-and-synthesize." Starting with system requirements in English, the system specification is encapsulated in an implementation-independent form, such as Boolean equations, FSM diagrams, or a hardware language like VHDL. CAD tools can then generate the specific designs (usually with human guidance).

5.2 Parallels between Gajski's SER and Our ADoI

While this is an improvement, there is still room for error should the specification be incomplete or ambiguous, as a natural language like English can be. Hence, Gajski et al. have developed a methodology that is similar to describe-and-synthesize, except that the requirements specification is captured in an executable language called SpecCharts, which Gajski et al. developed. The methodology also raises the level of abstraction in an attempt to achieve higher productivity similar to the earlier "PMS" (Processors/Memories/Switches) hierarchy of McFarland, et al. [132] The design objects we use are logical extensions of those considered by Gajski and McFarland. See Table 5-1 (after Figure 1 in [22] and similar to Table 1 in [132]) for how we have extended this table to include the COTS MP board-level hardware (in blue italics) used in our ADoI.

Table 5-1. Extending SER design representation and abstraction levels to our ADoI (Board Level).

Levels	Behavioral Forms	Structural Components	Physical Objects
Transistor	Differential equations, current-voltage diagrams.	Transistors, resistors, capacitors.	Analog and digital cells.
Gate	Boolean equations, finite-state machines (FSMs).	Gates, flip-flops.	Modules, units.
Register	Algorithms, flowcharts, instruction sets, generalized FSMs.	Adders, comparators, registers, counters, register files, queues.	Microchips, ASICs.
Processor	Executable specification(s), programs.	Processors, controllers, memories, ASICs.	PCB's, MCM's.
Board	***Executable specification(s), programs.***	***Components, connections, ports***	***SBC's, MP boards, I/O boards, high-speed interconnections***

To be useful, there must be a good match between the system design methodology and the system under specification and design. The scope of embedded systems considered by Gajski et al. ([22, 67]) includes a bus controller, microwave-transmitter controller, telephone answering machine, and other systems containing at most a single microprocessor and usually a number of FPGAs. While these systems are at a lower level of digital design than the ADoI in this research, the characteristics common to these types of applications are still pertinent. These characteristics are tabulated in Table 5-2 (after Figure 3.22 in [21] and explained below), and include how effective a representative spectrum of current system design languages are in modeling embedded systems concisely and precisely [133] [134]. These characteristics are those essential to the application and technology domains of Gajski et al. They are also a subset of those in the Sarkar taxonomy of Figure 4-6.

Table 5-2. Language support for conceptual model characteristics of embedded systems.

Language	Embedded System Features					
	State Transitions	Behavioral Hierarchy	Con-currency	Program Constructs	Exceptions	Behavioral Completion
VHDL	○	◎	●	●	○	●
Verilog	○	●	●	●	●	●
HardwareC	○	◎	●	●	○	●
CSP	○	●	●	●	○	●
Statecharts	●	●	●	○	●	○
SDL	●	◎	●	○	○	●
Silage	–	–	●	–	–	–
Esterel	○	●	●	●	●	●
SpecCharts	●	●	●	●	●	●
Coverage Key	● Feature fully supported ◎ Feature partially supported ○ Feature not supported – Not applicable					

Embedded systems are intrinsically state-based and change from mode to mode as driven by external events. These mode changes, or s*tate transitions*, are accounted for by Sarkar in the orthogonality attribute, which is being able to describe two behaviors independent of one another. *Behavioral hierarchy* is the feature necessary to decompose large complex behaviors into smaller sub-behaviors, which can be either sequential or concurrent. It is accounted for in Sarkar's hierarchy attribute. *Concurrency* is the feature describing behaviors executing at the same time. It is an essential feature in complex reactive systems and must be accommodated by the specification model to reduce complexity; hence, it is accounted for by both Gajski and Sarkar. *Program constructs* is the feature that describes the degree to which mathematical expressions can be represented, as well as programmatic control flow paradigms such as branching and iteration. This feature is implicitly encapsulated by Sarkar's representation scheme attribute. *Exceptions* are those events requiring immediate system response, and are common to embedded reactive systems, being explicitly called out by both Gajski and Sarkar. *Behavioral completion* is the condition when a behavior completes, and notifies the system so that the system controller can utilize the behavior's resources for another

task(s). Sarkar's methodology basis does not explicitly cover this attribute, though it is implicitly covered in his sub-attribute of concurrency within the language support attribute.

Both capture-and-simulate and describe-and-synthesize methodologies are hardware-oriented. Gajski et al. have acknowledged the well-established software methodologies and have raised the abstraction level for hardware design to microprocessors, memories, and buses in developing their "Specify-Explore-Refine" (SER) methodology, which is illustrated in Figure 5-1 (after Figure 2 in [22]). The SER methodology is composed of three clearly defined tasks on three classes of functional objects and are summarized in Table 5-3 (after Figure 5 of [22]). The *specify* phase involves the capture of the system requirements using an executable language. This language should be able to accurately and completely capture the requirements, be easy to understand, and be able to interface to CAD tools in order to support modeling and analysis. Gajski et al. have developed SpecCharts as the language best matched to their methodology.

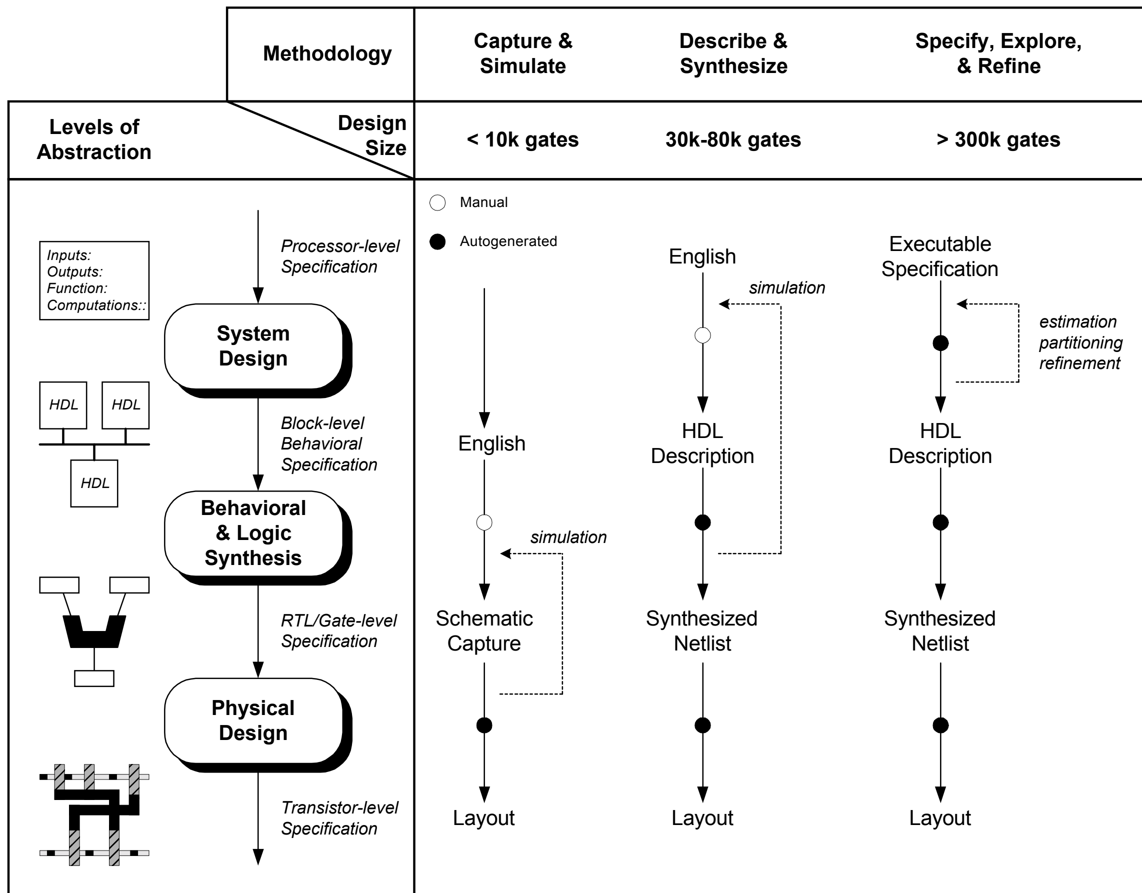

Figure 5-1. Gajski et al.'s three classes and scopes of design methodology.

Table 5-3. System design tasks.

Functional Objects	**Specification** *(e.g., VHDL)*	**Exploration**		**Refinement**
		Allocation	**Partitioning**	
Variables	Signals, variables	Memories	Variables to memories	Address assignment
Behaviors	Processes, procedures	Processors	Behaviors to processors	Interfacing
Channels	Global signals, ports, port maps	Buses	Channels to buses	Arbitration and protocols

The *explore* phase consists of different mappings of the system functionality to different hardware and software components in an attempt to best satisfy design constraints. This involves performing two tasks for each of the functional objects:

- *Allocation*–Adding system components to the design which are those shown at the Processor level in Table 5-1 for the systems of interest in this paper. Note that the

designer may specify appropriate constraints or parameters necessary to characterize each allocated component (e.g., bus bandwidth, processor throughput, etc.)

- *Partitioning*–Mapping functional objects into the allocated software and hardware components. Various closeness criteria can be used to determine optimal object clustering (cf. Chapter 6 in [21]). Common criteria for behaviors include interconnection, communication, sequentiality, and hardware shareability. Common criteria for variables and channels include sequential access, common accessors, and width similarity. Partitioning pertinent to the systems of interest in this research involve which parts of the processing are best suited to which processor types (vectors to DSPs and data to RISCs), how to parse the algorithms (pipelining and parallelization), and perhaps assigning state control to the host/system processor.

Each different allocation of system components and each different partition will produce one candidate system implementation. These implementations comprise the design solution space that for COTS hardware-based design is finite. Quality metrics are needed to evaluate partitioning option to best search the design solution space. Comparing the candidate designs' metrics to the given requirements leads to the optimal design. Though the design space is finite, these systems can be complex. But being able to link specification language based design exploration to the specification requirements allows designers to find the best (as the designer defines it) solutions.

The *refine* phase translates the explore phase decisions into updates in the system specification. Refinement migrates the design from a pure functional spec toward a structural implementation. E.g., behaviors must be added to maintain correct functionality, while defined behaviors may need to be distributed over multiple processors. This requires variables such as data vectors and matrices to be mapped into shared memory buffers and communication protocols to be established, such as defining and assigning semaphores for process synchronization. After the refinement, the specification will look much like the block diagram a chief architectural guru might have sketched in the traditional approach. However, there are two significant differences:

1) *Optimal*. The refined specification was obtained via a thorough and organized solution space search.

2) *Consistent and Complete*. The refined specification was derived formally from the original specifications and is therefore more likely to be consistent and complete with respect to the original specifications. By doing this at the beginning of the design cycle, the need for expensive and time-consuming design iterations is eliminated.

5.3 Extending Gajski's SER to Our ADoI

There is a natural extension of Gajski's SER to our ADoI. Table 5-1 shows there are differences in Gajski's ADoI and ours with respect to target technologies. But it also illustrates the strong similarity in design objects as extended for a board-level SER. Also, the nature of the constraints is the same, including embedded SWAP and a real-time paradigm. We can extending Gajski's SER to the COTS MP domain as illustrated in Figure 5-2, with the "Board-level SER" complement of the Gajski SER diagram in blue.

In the next chapter we develop a methodology (a tool or combination of tools and a set of rules) to allow a designer to capture system requirements and then search and explore system-level design alternatives to discern which different technologies and architectures are able to satisfy the system requirements. Refinement of the system-level design will be done by iterating through the search and explore phase. System-level design exploration can be done by employing performance modeling [5, 135-138] similar to a Ptolemy-based architectural trade tool [14]. Emphasis will be on creating model continuity to maximize the Sarkar unified basis metrics.

In the next chapter we also discuss the choice of the frameworks we use to accomplish each stage in the design process, along with a novel technique for effectively integrating the frameworks. Suffice to say, the most significant inspiration of the Gajski SER methodology is the focus on maximizing specification capture and design correctness vis à vis saving a gate–or even a processing node–or line of code here or there. As complexity increases, so must the methodology's ability to capture specifications in an executable model and provide model continuity between frameworks integrated to accomplish each stage in the design process. We present this methodology in the following chapter.

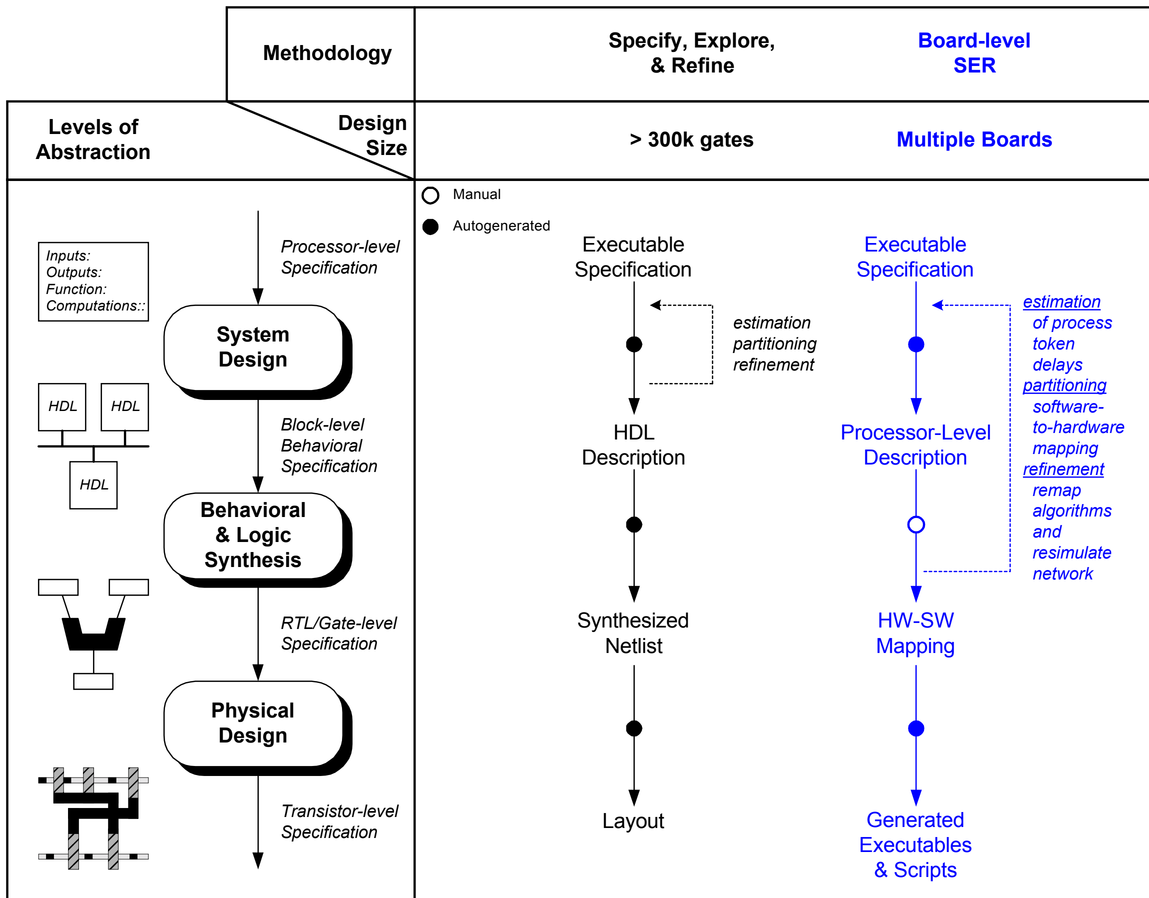

Figure 5-2.Extending Gajski's SER (from Figure 5-1) to our ADoI.

Chapter 6

The MAGIC Specification and Design Methodology

We now describe our new specification and design methodology, the MAGIC SDM. In this chapter we lay out the "tools and rules" of this methodology by first establishing the rules, and then discuss the framework characteristics that will effectively support the rules. We then describe the DSP rapid prototyping and performance modeling tools useful for specification and for design according to our MAGIC SDM. We also provide an overview of VSIPL and MPI middleware that provides model continuity. We then lay out the transformation rules to generate the middleware code from the DSP rapid prototyping graphical environment that we use as a specification framework.

6.1 Overview of the MAGIC Methodology

Any SDM will start with some human language text requirements specification document. The goal of SDMs is to go from this inexact document to a design and implementation in a manner that minimizes propagation of specification and/or design errors. We do this with an integration of tools guided by sound rules to capture the requirements in a format to make sure there are no conflicts or absence of requirements, then proceed on through a vendor-independent design phase. Without first committing to a vendor, alternate architectures can be considered and an optimum one decided upon. We then take code generated from specification and software-to-hardware mapping determined from design to provide inputs to an implementation framework.

The starting point for specification and design in our ADoI is the set of computation requirements. These are algorithms and data "specified" by MATLAB code, including different scenarios of inputs and their associated outputs. The MATLAB code serves well as an input to a framework that can use it to create an executable

specification. The scenarios will provide valuable inputs for the generation of test data to be used downstream in the implementation phase. Communication and control requirements typically refer to data I/O rates as well as the signal processor modes and the control signals that determine the processor's mode (state). Processors in our ADoI have few states; often there are two: one state for initialization and setup ("outer loop") and one state for steady state data transformation ("inner loop"). These modes must be defined and described, preferably in an executable model. Constraints include SWAP, latencies, reliability, and other "illities," which are usually tabulated. It would be useful to have these data encapsulated in a fashion that allows us to include their verification during the specification and design iterations. Recalling how we are extending Gajski's SER to our ADoI with a "Board-level SER" in Figure 5-2, we now redraw it as our new MAGIC SDM, as shown in a simplified diagram of the specification and design flow in Figure 6-1. We will expand this diagram in the following sections.

Our executable specification will be encapsulated in a framework that is capable of generating middleware that can be used to valuate tokens in a performance modeler core to the design phase. This allows design exploration within a given technology and among multiple technologies. Thus, the designer is free to explore different technologies, verifying that a certain technology can satisfy requirements before purchasing expensive multiprocessing hardware and software, and prototyping the application software. The designer uses a performance modeling framework to accomplish this, arriving at an architecture with a given technology that is optimum, typically with respect to the number of compute elements ("CEs"). The architecture with the minimum number of CEs that still satisfy non-performance constraints, such as SWAP and reliability, will be the design candidate for that technology. This is repeated for the other technologies.

We will make a design decision on which technology is to be used for implementation. This decision is based on monetary cost, prior investment in spares, familiarity with the software development tools for a particular vendor, etc. The middleware generated by the specification framework can be used in the implementation phase for steady-state software. The software-to-hardware map developed in the design

phase will also be used in the implementation for developing configuration files and as a specification for writing the communication code. Some tweaking of the architecture may be required, eased by the use of the deployable CASE SDMs, which are excellent for rapid prototyping because of their mapping tools and code generation facilities. Each of these phases is laid out in more detail in the following sections.

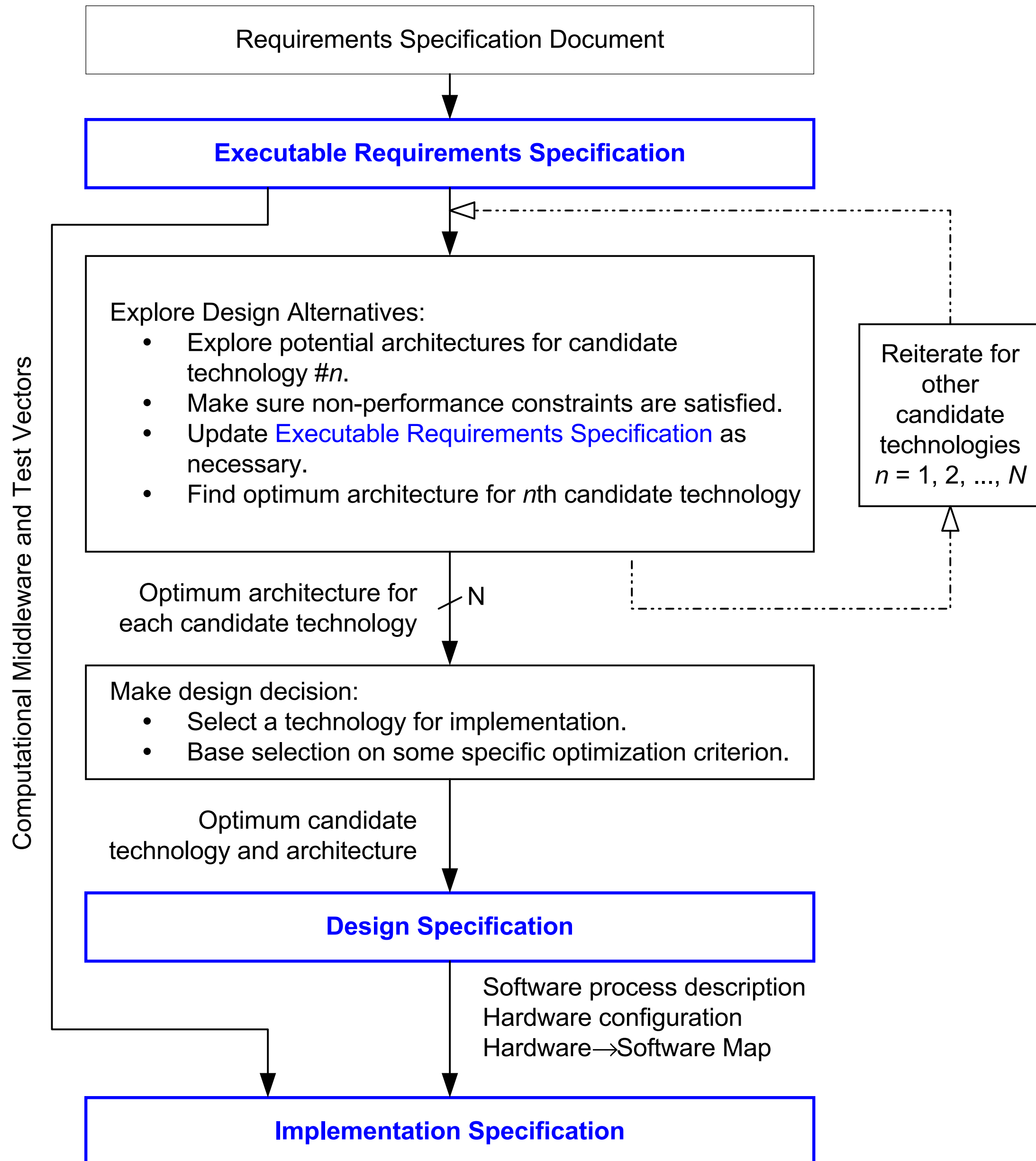

Figure 6-1. Simplified diagram of the MAGIC specification and design flow.

6.2 Establishing Model Continuity

While Figure 6-1 describes the flow of the methodology, we want to show how our MAGIC SDM establishes model continuity. We first illustrate how model continuity is missing in today's COTS MP methodologies in Figure 6-2. Currently, constants such as filter coefficients can be passed from MATLAB .m files into a CASE SDM or a simpler vendor software development environment, but that is the only link from the requirements specification and design specification to the implementation phase in the whole design process. Not having an executable requirements model and a channel for passing it to the design analysis phase leads to *model discontinuity*, which is the total absence or minimal presence of model continuity. Model discontinuity requires that a design specification be drafted in a natural language, specifying the following:

- Software processes
- Hardware configuration
- Software-to-hardware mapping

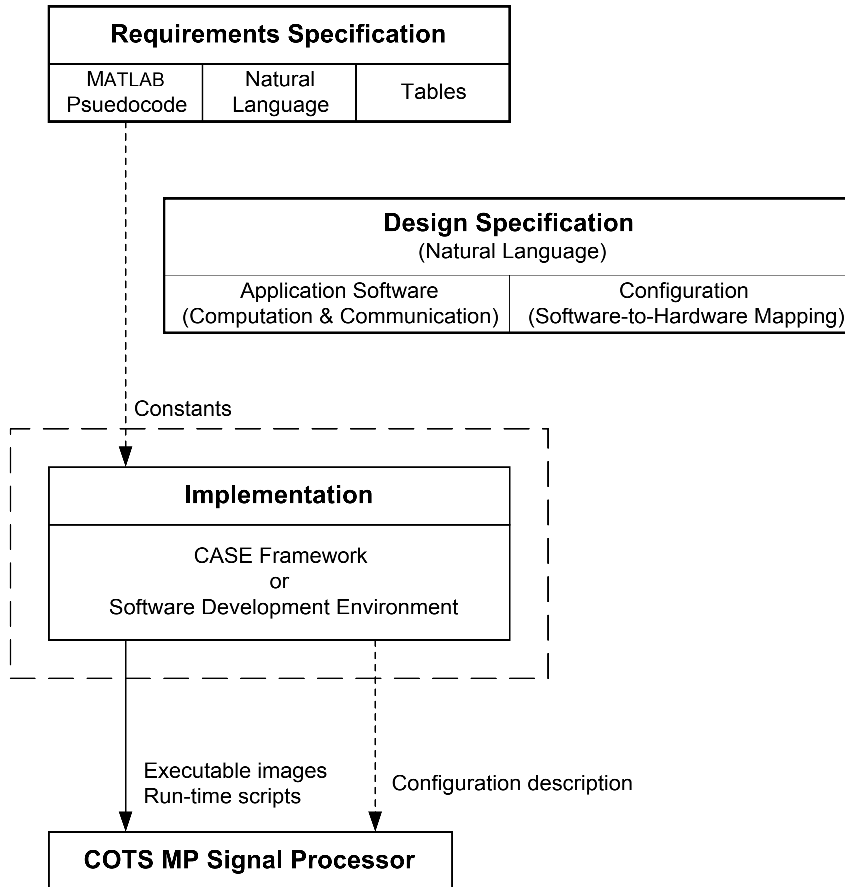

Figure 6-2. How model continuity is currently lacking in CASE SDMs.

Our MAGIC SDM specifies the use of tools and rules to establish model continuity. We present this in generic terms as shown in Figure 6-3, deferring the specification of tools and how we established continuity between them to Chapter 7 and summarized in Figure 8-3.

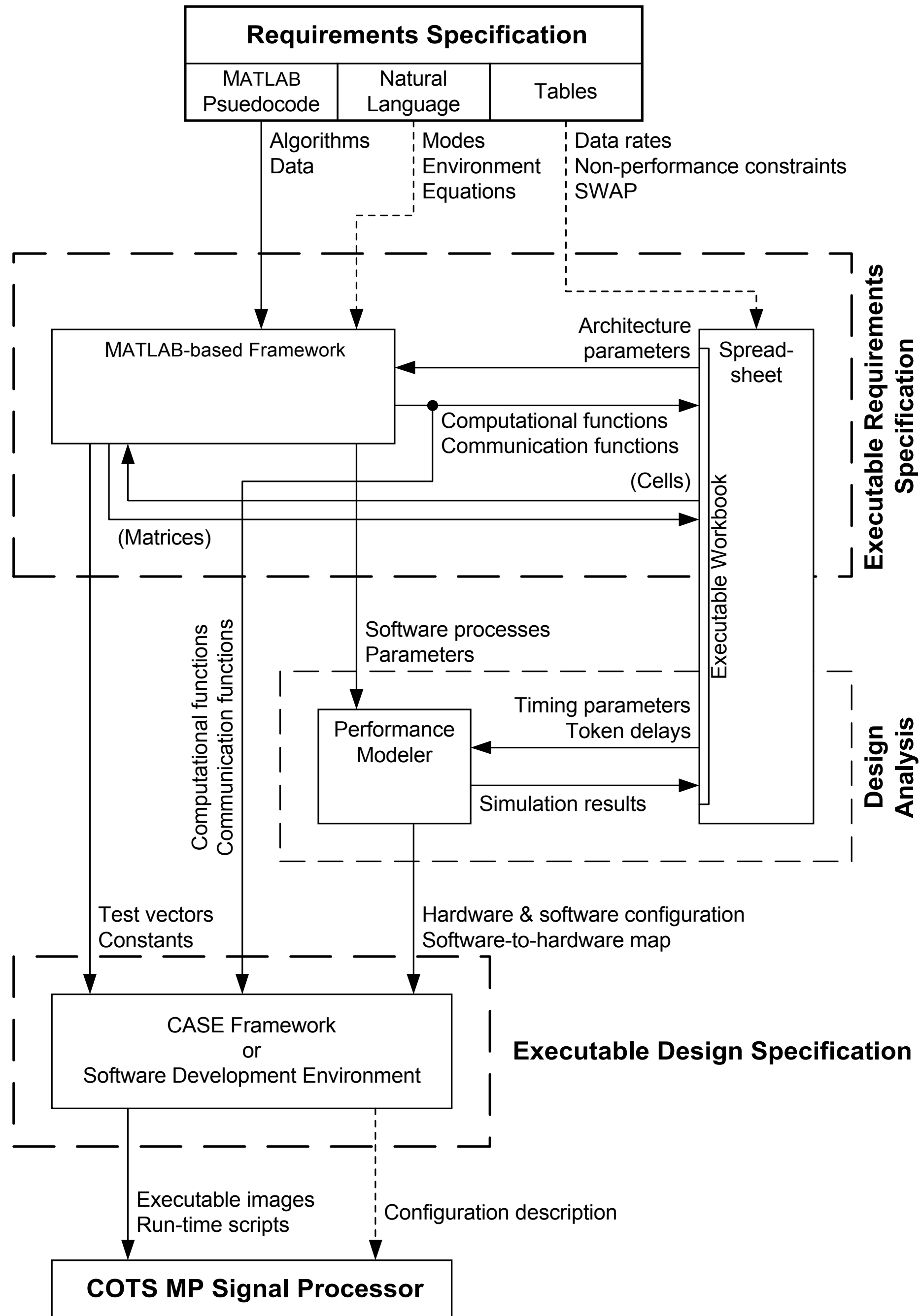

Figure 6-3. Establishing model continuity between an executable specification model and a design specification model.

6.3 "Rules"–The Steps of the MAGIC SDM

In this section we lay out the specification and design rules of the MAGIC SDM. We assume that a natural language (e.g., English) requirements specification document exists that contains the system requirements, interfaces, data rates, etc. We do not assume that the algorithms have been coded in MATLAB, though it would be very unusual for them not to be. In the following sections, we describe how to take the natural language requirements specification document and convert it into a MAGIC specification and design flow.

These steps are intended to provide a summary, while the details are deferred to the next chapter where the MAGIC SDM is used in a case study. In the following sections (§6.3.1–§6.3.7), the MAGIC SDM rule is succinctly stated in italics, followed by some brief commentary as needed.

6.3.1 Tabulate Requirements

Identify and cull details of the requirements from the requirements specification document.

The natural language requirements specification document will by its nature contain excessive verbiage. It must be sifted to extract all the requirements specifics. In particular, computational requirements are specified by algorithms and their complexity is given in operations/time. Communication and control requirements are similarly identified along with their data rates and state/input/output, respectively. Constraint requirements are tabulated with their dimensions such as size, weight, and power, as well as "illities" such as reliability (MTBF), etc.

6.3.2 Capture Non-Constraint Requirements in an Executable Model

Describe computation, communication, and control requirements in an executable model.

The non-constraint requirements are characterized by computational algorithms and communication and control state information. These requirements readily lend themselves to being captured in an executable model. The algorithms are best described

in a SDF MoC-based framework and/or language. The communication and data pathing and control mode states are described in a DE and/or S/R MoC-based framework and/or language that supports parallelization, including scalability if possible. The framework must have the ability to generate middleware for computation and communication.

6.3.3 Build Executable Workbook with Requirements

Put all the requirements into a tabular form to facilitate computational manipulation, e.g., in a worksheet/workbook environment such as Excel.

Individual worksheets are used for different requirements and for summaries. This is a very natural matching of a computational tool to the type of data to be operated upon. Spreadsheet-type tools provide a tabular format canvas which is most suitable for small discrete data definitions such as is characterized by requirements data.

6.3.4 Gather Benchmarks for Tokens

Gather benchmarks of the middleware functions that are likely to be used in design and implementation and enter them into the executable workbook.

The executable specification framework will generate middleware code for computation and communication for the mode(s)'s steady state execution. Benchmarks are employed to valuate the token delays used in the performance modeling for system design analysis and trade-offs. Vendors typically provide benchmarks for their libraries, whether computation or communication. Since these benchmarks are typically obtained under the most favorable conditions, care must be exercised to note under what conditions the benchmarks were obtained.

6.3.5 Explore Alternative Architectures and Technologies

Use performance modeling to explore potential architectures for a given technology, then determine the best architecture for that technology. Repeat as necessary for other candidate technologies.

Starting with one of the technologies under consideration, the designer will use the executable specification as a basis for investigating potential architectures. The

executable specification is used to assure that candidate architectures still yield the correct results within specified time constraints. Outputs of this phase include test vectors that can be used in the implementation phase for verification, as well as the middleware and the data sizing. The data sizing allows the benchmarks to be indexed and the correct value determined for that particular architecture. This applies to both types of middleware, for computation and for communication. Some scaling or interpolation will probably be needed. These benchmarks are used to compute token delays for each processor in the network, then simulated with a performance modeling framework. After iterating through some candidate architectures, an optimum architecture is arrived at, where optimality is determined a priori.

After arriving at the optimum architecture for a given technology, repeat the above for alternative technologies. Remember that no hardware has yet been purchased.

6.3.6 Make Design Decisions

Decide which technology and architecture to use in implementing the signal processor.

Given a number of architectures and technologies, make decisions as to which technology to use as well as which architecture to implement using the chosen technology. This can be made based on any (possibly weighted) combination of considerations including monetary cost, spares, software reuse, specification and design framework investment, and other considerations.

6.3.7 Create Implementation Specification

Pass along architectural details to the system implementation specification based on the design exploration.

The design search phase of §6.3.5 produces the following implementation specification items that can be consumed by a CASE implementation framework:

- A hardware configuration
- Generated middleware for computation and communication
- A software-to-hardware map

- Test vectors for verification

Having selected an architecture that meets or exceeds specifications with the above specifications already obtained, these specifications are consequently passed along to the implementation framework or environment.

6.4 "Tools"–The Frameworks Integrated into the MAGIC SDM

We have chosen the following frameworks to integrate into the MAGIC SDM. We did not choose them because they are perfect; almost all frameworks targeting complex systems are more accurately described not as "frameworks" but as "frame*works-in-progress*." We have chosen the frameworks described in this section for the following reasons:

- Appropriate–They met the criteria of Table 3-2 in §3.5.
- COTS–They are commercially available, stable, and supported.
- Available–We could obtain them and use them in our case study (Chapter 7).

For requirements capture and modeling we have chosen Excel, as well as the DSP Workstation (DSPW) and Excel Link from The MathWorks. For design exploration we have chosen eArchitect from Viewlogic. Characterization and features driving the selection of The MathWorks and Viewlogic frameworks are given in the following sections where we discuss these frameworks.

6.4.1 DSP Workstation

MATLAB's importance has been stated previously. It is the de facto lingua franca of algorithm developers, including radar signal processing system analysts. It is common to find MATLAB used as pseudocode for individual algorithms or even whole systems. It is therefore imperative to have a framework that can execute such pseudocode if it is to be used in an executable specification. DSP Workstation (DSPW) from The MathWorks consists of three integrated frameworks, which are discussed in the following sections (§6.4.1.1–§6.4.1.4). There are a few competitive frameworks (Mathtools' MATCOM, Mango's Math-Link Accelerator™, et al.), but they lacked the comprehensiveness that DSPW possesses (cf. Table 3-2).

6.4.1.1 MATLAB

Ubiquitous in its use for algorithm development, MATLAB is useful for concretely describing algorithms, hence encapsulating computational requirements. Its rich matrix-oriented language is expressive and its data visualization is exceptionally powerful for allowing results obtained within any of the frameworks to be displayed [139, 140]. Possessing a dataflow MoC, it satisfies the optimal preference described in §3.3.2.

6.4.1.2 Simulink and the DSP Blockset

Earlier versions of Simulink were strictly dynamic simulation frameworks, seeming to have only the vendor's name in common with MATLAB. The latest version (version 3.x, which is called "Simulink 3") is a dramatic transformation, now becoming a viable rapid prototyping framework strongly tied to MATLAB, allowing MATLAB expressions to be used explicitly in Simulink blocks [141, 142]. Simulink itself has a DE MoC, which makes it appropriate as a good MoC for capturing communications and control requirements as shown in §3.3.3.

Earlier incarnations of Simulink were more oriented toward the modeling and simulation of control systems. Simulink 3 has made Simulink much more open and extensible, and with the DSP Blockset it has become a viable rapid prototyping environment for DSP applications. The DSP Blockset is a collection of block (Simulink elements) libraries designed specifically for DSP applications, including the following key features:

- Frame-based operations
- Matrix support
- Classical, adaptive and multirate filtering
- Linear algebra
- Real-time code generation capability (cf. §6.4.1.4)

The frame-based paradigm is critical since it is oriented toward implementation on most any processor-based signal processing system. Most real-time DSP systems optimize throughput rates by processing signal data in a batch mode where the batch is

referred to as a “frame” [143]. The frame is a logical temporal subset of the incoming signal stream. In radar terms, a “dwell” is the data associated with the return of a group of pulses that are processed together. In application terms it could be the entire dwell, a collection of dwells, or subsets of either, distributed to multiple processors for parallel processing. Being frame-based, the DSP Blockset forces the specifier to be careful in expressing the format of the data with respect to time. This is easy to lose track of in MATLAB.

The DSP Blockset also maps well into the VSIPL computational middleware. Simulink allows sinking data to the MATLAB workspace to allow the specifier to check Simulink outputs with MATLAB results. While Simulink and the DSP Blockset forces the specifier to be more structured in describing the signal flow and processing, it does not bias the design. Consequently, Simulink in general and the DSP Blockset in particular will be our canvas for capturing the signal dataflow and processing.

6.4.1.3 Stateflow

Stateflow is a Statecharts [83, 84] variant that is tightly integrated with both MATLAB and Simulink. Stateflow complements Simulink by providing a framework and canvas for designing state-based control systems by providing the following key features [144]:

- GUI-based modeling and simulation of complex reactive systems
- Seamlessly integrates event-driven behavior within Simulink’s discrete-time simulation environment
- Uses FSM theory, Statechart formalisms, and flow diagram notation
- Supports hierarchy, parallelism, junctions, and history
- Performs runtime checks for transition conflicts, cyclic problems, state consistency, and data range checking

Stateflow has a CFSM MoC, and by being integrated with Simulink and its DE MoC, provides a very effective means of capturing and modeling communication and control requirements (§3.3.3).

6.4.1.4 Real-Time Workshop (RTW)

RTW is the C code generation facility complementing Simulink and Stateflow. There is another C code generation capability within the MATLAB family, the MATLAB Compiler, which translates MATLAB code into C code. It is designed for streamlining the execution of MATLAB-based simulations on workstations, and is not appropriate for fixed memory real-time implementations. C code for Stateflow requires the Stateflow Coder to support code generation and seamless integration with RTW. Relevant features of RTW's code generation include the following [145]:

- Support for discrete-time and event-driven systems
- Customizable through modifications to the Target Language Compiler (TLC)
- Makefiles are customizable through the RTW control framework API
- Generates concise, readable, and portable C code

Another reason RTW was chosen is that The MathWorks is moving towards adding VSIPL computational middleware support to RTW, an effort we are supporting.

6.4.2 Excel and MATLAB Excel Link

Excel provides the framework needed for requirements tabulation and analysis (cf. §6.3.2–§6.3.4). The Excel spreadsheet is a commodity productivity application familiar to all, and in the same way that many analysis and design frameworks provide support for MATLAB, tabular data-oriented frameworks provide support for Excel. One good example is the number of reliability software products such as the following:

- Relex 7 (Relex Software Corp.; [146])
- PROACT (Reliability Center, Inc.; [147])
- The Reliability & Maintenance Analyst (Espinoza Consulting; [148])

Each of these provides some kind of link to Excel and/or Access, both of which belong to the Microsoft Office framework and can exchange data between one another.

Excel Link is a facility rather than a framework. It is a channel to allow Excel to copy data into MATLAB and execute on it in MATLAB while remaining in Excel [149]. Many requirements are easily "captured" in a spreadsheet, and depending on the

sophistication of the computation required to iterate between requirements modeling and design analysis, Excel or MATLAB may be required. Excel Link allows the specifier to remain in a single framework. The MAGIC SDM integrates frameworks as necessary to achieve model continuous specification and design before committing to a given hardware vendor. However, it is also important to minimize bouncing between frameworks, which could lead to confusion and careless mistakes, the very thing the MAGIC SDM is striving to avoid.

6.4.3 eArchitect

Performance modeling was chosen for design exploration and analysis since it supports architectural trade-off analysis without prematurely committing to a given vendor's hardware and software. The COTS performance modeling framework that is best matched to our ADoI will provide support for the technologies most likely to be used for implementing the signal processor. Leading vendors in this technology space include those listed in Table 6-1, which notes their open standards interconnection technology and processors they support.

Table 6-1. Technologies in our application domain.

Vendor	Interconnection	Processors Supported
CSPI	Myrinet	i860, SHARC, PPC
Mercury Computer Systems	RACE, RACE++	i860, SHARC, PPC, and Altivec
SKY Computers	SKYchannel	i860, SHARC, PPC, and Altivec

There are few performance multiprocessing modeling frameworks available commercially. We are only aware of one that supports VME and at least two of the above interconnection technologies, and that is eArchitect from Viewlogic.

Viewlogic's eArchitect is an architectural prototyping tool that supports hardware software codesign by providing canvases for hardware and for software design. The framework provides a hardware canvas for connecting elements in the hardware library, including SHARCs, PPC603s, VME, PCI, RACEway, and Myrinet. The software canvas

is where the software is modeled, typically as a block diagram representing communication between various software tasks. As the specification-design cycle iterates, more detail can be added to the software flow diagram, either graphically or textually. The designer only needs to estimate the clock cycles consumed by a processor to complete the design prototype. The codesign can be then be simulated after which powerful visualization tools allow the design to be examined as to individual hardware and software timelines, hot spot analysis, latency and utilization analysis, and other characteristics as well [150-152].

6.5 Model Continuity via Middleware

Model continuity will be achieved in large part through the use of middleware for computation and communication. Open standards-based middleware supports computation and communication software portability, which means that middleware written for one vendor's hardware should run on another vendor's platform. Consequently, middleware code that constitutes the inner-loop software implementation can be used for different vendors' platforms for design analysis in performance modeling. Critical to making the use of middleware a strong thread of model continuity is the autogeneration of middleware code, since automating the generation of software by a framework that is correct in specification reduces the chance of error in the design and implementation.

A code generator such as the RTW of DSPW that could generate middleware for computation using VSIPL, MPI for communication, and/or MPI/RT for communication and control will produce code for both design and implementation. The generated middleware can be used to quantify process delays in the performance model framework and as the core for signal processing implementation application software. An overview of VSIPL and MPI is presented, after which we show how we use it for model continuity.

Our reasons for choosing VSIPL and MPI are very similar to our reasons for choosing the frameworks discussed above in §6.4. They are stated here in order of importance with the most important reason stated first:

- Acceptable performance–These middlewares deliver high-performance because they are tightly integrated with the vendors' computation and communication libraries.
- Standards-based–Since all the COTS MP vendors in our ADoI space support these middlewares and actively participate in their standardization processes, frameworks that generate VSIPL and MPI code will be consumable by all of the hardware vendors' SDEs considered in the design phase.
- COTS–They are now becoming commercially available and therefore stable and supported.
- Available–We could obtain them and/or benchmarks of their performance for use in our case study (Chapter 7).

6.5.1 VSIPL: Computational Middleware

VSIPL is an API supporting portability for COTS users of real-time embedded multicomputers that has been produced by a national forum of government, academia, and industry participants. VSIPL is computational middleware, which also supports interoperability with interprocessor communication (IPC) middleware such as MPI and MPI/RT. The VSIPL Forum is nearing completion of the API, a prototype reference library, and a test suite to verify API compliance. Commercial implementations are just now becoming available (Fall of 1999). Earnest consideration by various defense programs is underway and early adoption has begun.

The VSIPL API standard provides hundreds of functions to the application software developer to support computation on scalars, vectors, or dense rectangular arrays. The v1.0 API specification document lays out the categories of the functionality in the following way:

- Support functions
  - Object creation and interaction
  - Memory management
- Basic scalar operations
- Random number generation
- Basic vector and elementwise operations

- Signal processing
    - FFT operations
    - Correlation and convolution
    - Windowing
    - Filtering
- Linear algebra
    - Basic matrix and vector operations
    - Linear system solvers

Canonical development of embedded signal processing applications using COTS multiprocessing hardware and software typically consists of partitioning the code into two portions. One portion is the "outer loop" where the setup and cleanup functions are executed, typically memory allocation and coefficient generation, such as FFT twiddle factors and window coefficients. The other portion is the "inner loop" where the time-critical repetitive streaming data transformation functions lie. A VSIPL application will be built similarly, with the outer loop executing heavyweight system functions that allocate memory when creating blocks and parameterized accessors called views. The block creation is substantial in both memory and execution time due to requiring system support. The view object handles take up very little memory, but is still a heavyweight function with respect to time because it also requires system support. This is discussed in further detail in §A.1 and illustrated in **Error! Reference source not found.**.

6.5.2 MPI: Communications Middleware

There have been a number of approaches to accomplishing parallel processing, a topic of breadth and depth that is beyond the scope of our discussion. Suffice to say, out of the plethora of approaches (hardware and/or software) grew an approach that has gained growing support and become a standard. Rather than trying to develop a special language (such as HPF, High Performance Fortran) and concomitant compiler, a library of functions was specified to achieve parallelism by message passing, explicitly transmitting data from one process to another. Message passing is a powerful and very general method of expressing parallelism and can be used to create extremely efficient

parallel software applications. It has become the most widely used method of programming many types of parallel computers [47, 153-155].

Message passing is especially popular on scalable parallel computers (SPCs) with distributed memory, and on Networks of Workstations (NOWs). There have been many variations over the last ten to fifteen years, with each variation helping to crystallize what is core and critical to the message passing paradigm. About five years ago a consortium known as the Message-Passing Interface (MPI) Forum formed to define both the syntax and semantics of a standard core of library routines that would be useful to a wide range of users and efficiently implementable on a wide range of computers. The MPI Forum was made up of over 80 people from 40 organizations of vendors, users, and researchers. Their goals included portability but not at the expense of performance, including heterogeneous platforms, and multiple language bindings, including C and Fortran [18, 156].

High-performance implementations of MPI are now available. The leading vendor is MPI Software Technology, Inc. (MSTI) who provides high-performance implementations of MPI under the commercial trademark MPI/PRO for NOWs and SPCs, including two of the three leading COTS MP vendors in our technology space (RACEway and Myrinet). There is another standards effort underway to specify a real-time version of MPI with a guaranteed quality-of-service (QoS) called MPI/RT. Non-QoS beta versions of MPI/RT are just now beginning to appear.

The MPI standard includes the following characteristics, features, and functionality:

- Point-to-point communication
- Collective operations
- Process groups
- Communication domains
- Process topologies
- Environmental management and inquiry
- Profiling interface

- Bindings for Fortran and C

The MPI standard does *not* specify:

- Explicit shared-memory operations
- Operations requiring OS support not standard during standardization
- Program construction tools
- Debugging facilities
- Explicit support for threads
- Support for task management
- I/O functions

For the applications in our ADoI, the parallel programming model will be single-program multiple-data (SPMD). In strict MPI terms, the executable images are identical, with the process having to identify itself and branch accordingly to operate on the data as a function of its process rank. This model as applied to our ADoI has the same computational code, but operates on different tiles of the data square. Consequently, while VSIPL is the computational middleware and MPI is the communication middleware, the application software is actually a set of MPI programs. Communication and control are accomplished by the MPI middleware, determining what processes operate on what and when. The processing itself is accomplished by VSIPL middleware. The two fundamental functions that accomplish the actual message passing are `MPI_Send`, which sends a message to a designated process, and `MPI_Recv`, which receives a message from a process. Their prototypes and other basic MPI details are in §A.2.

The two most important reasons for choosing VSIPL and MPI are acceptable performance and that they were standards-based. If these middlewares could not deliver performance commensurate with the vendors' native computational and communications libraries, they would not be as useful and therefore less acceptable. However, preliminary VSIPL benchmarks recently released by one COTS MP vendor shows computational throughput achieving as much as 98% of the throughput (MFLOPs/s) of their native algorithm library. MPI benchmarks released by one commercial MPI vendor show

bandwidths within 5% of the RACE theoretical maximum for large block sizes, which is very close to that achieved by the vendor's own native communication library.

Being standards-based is the other key characteristic of these middlewares. The participation of researchers, implementers, and users to form and support these standards goes a long way towards assuring their adoption. VSIPL and MPI being official standards and becoming de facto standards means that code generated within the MAGIC SDM can be used to estimate communication and computation token delays in performance modeling for multiple vendors' platforms. The generated code can also be used as the inner-loop computational code in the implementation. This strengthens the thread of continuity from specification to design (token delays) and implementation (inner-loop code).

6.5.3 Using VSIPL & MPI for Model Continuity

We introduced our interest in the autogeneration of middleware code in §6.5, where we stated that a code generator such as the RTW of DSPW that could generate middleware for computation using VSIPL, MPI for communication, and/or MPI/RT for communication and control would be able to produce code for both design and implementation. This generated middleware can be used to quantify process delays in the performance model framework and can also be the inner-loop code for the signal processing implementation application software.

6.5.3.1 Code Generation Prototype

Currently the RTW of Simulink generates C or Ada code optimized for a single thread and fully commented. All Simulink blocks are converted to code except for MATLAB function blocks, which must be written as C MEX "S-functions" (user-supplied Simulink block whose behavior is defined by C code) in order to be integrated into a RTW code build. RTW allows different types of code output:

- C code
- Ada code
- Real-time program

- High-performance stand-alone simulation

We are primarily interested in the first type of code output, since C is still the language of choice for COTS MP vendors. It contains the system equations and initialization functions for the Simulink model, which can be used in nonreal-time simulations or for real-time applications. The real-time program option targets specific lower-end single board DSP and controller products. It is adequate for our purposes that the functions invoked by the RTW be VSIPL and MPI functions.

6.5.3.2 Mapping the DSP Blockset to VSIPL

There is a strong correlation between the functionality provided by Simulink's DSP Blockset and VSIPL. This is not a surprise since each targets the core functionality used by DSP analysts and software developers. This common functionality includes the following [15, 157]:

- Complex exponential
- Contiguous copy
- Convolution
- Correlation
- Cumulative sum
- Matrix scaling
- Matrix sum
- Submatrix
- Toeplitz
- Matrix Multiplication
- Matrix Product
- Transpose
- Cholesky Factorization
- Cholesky Solver
- LU Factorization
- LU Solver
- QR Factorization
- QR Solver
- Histogram
- Maximum
- Mean
- Minimum
- Window Function
- FFT
- Magnitude FFT

Other functionality shared by both is not as explicit, but obtainable by correct use of VSIPL functions, e.g., "flip" functionality in MATLAB and Simulink is achieved by traversing backwards through a VSIPL vector. Another example is that VSIPL uses the

same function call for forward and inverse FFTs with a flag passed in to control the direction of the FFT. This less obvious shared functionality is:

- Autocorrelation
- Difference
- Flip
- Matrix Constant
- Zero pad
- IFFT

6.5.3.3 Mapping Simulink to MPI

The applications in our domain tend to be computation-bound, not communications-bound; hence, the use of COTS MP hardware and the parallelization of the software. In this prototype of the MAGIC SDM, a simple point-to-point scatter-gather model is used. Multiple single `MPI_Send`s are used for a process to distribute its data (scatter) to parallelized processes. When a process collects interim results from parallelized processes (gather), it iterates through multiple `MPI_Recv`s. More complex multiprocessor models of communication exist that are supported by MPI and vendor-specific APIs. Since they provide incremental performance improvements, we will not consider them in this initial prototype of the MAGIC SDM.

6.5.3.4 Prototype Code Generation

Our focus is on the steady-state inner-loop application software, since it is the real-time code whose throughput requirements drive the codesign. It is this code that one of our MAGIC SDM frameworks generates. The outer-loop VSIPL setup code creates blocks and attaches views. The outer-loop MPI setup code does initialization and finalization. This code is not needed until implementation and is best left to that phase after the architecture and technology have been determined.

To generate the steady-state inner-loop middleware-based C code from Simulink, the DSP Blockset is translated into VSIPL or MPI function calls with the arguments determined by the parameters contained in the Simulink blocks. Basically, Simulink "boxes" are transformed into VSIPL computational function calls, while the "arrows" are transformed into MPI communication function calls.

We now describe the specific rules for transforming a Simulink diagram into VSIPL and MPI functions. These transformation rules are summarized in Table 6-2. For now we will assume two levels, a top system-level canvas and the boxes of the system-level canvas, which have their own individual process-level canvas. The top system-level canvas with the diagram of boxes and arrows is a system-level diagram where all boxes will be processes (one process per processor). Arrowheads are `MPI_Recv` functions and "arrowtails" (tails of arrows) are functions. There are two exceptions, the first and the last box. The first box has no arrowhead, hence no `MPI_Recv`. Similarly, the last box has no arrowtail, hence no `MPI_Send`. These two boxes refer to data input and output, respectively.

Except for these two end exceptions, all the middleware of the processes is generated in the same way. An individual process is defined by a top-level box, incoming arrowhead(s), and outgoing arrowtail(s). At the top level, all boxes are processes, composed of the following:

1) At least one `MPI_Recv` (one/arrowhead)
2) At least one VSIPL call (we assume no trivial processes)
3) At least one `MPI_Send` (one/arrowtail)

The `MPI_Recv` returns an array ("buffer") of data which maps into a VSIPL "block," which is "admitted" and "bound," after which the VSIPL "view" (handle) is used in the VSIPL function calls. For a single thread of Simulink blocks, the VSIPL functions may be executed "in-place," which means the output argument is the same as one of the input arguments. Heavyweight functions such as the FFT and linear solvers are exceptions and must be done "out-of-place." The code generator takes this into consideration, generating a token identical to the input, but appended with a character to differentiate it from the input. Each box on the process-level canvas corresponds to a VSIPL function as delineated in §6.5.3.2. Arguments of the Simulink blocks map into the VSIPL function arguments. The last VSIPL output is "released," then mapped into a buffer for the `MPI_Send`.

There are two exceptions regarding boxes of the top system-level diagram. They are the DSP Blockset mux (gather) and demux (scatter) blocks. The code generator must look ahead to where the block's arrowtail is going to know what the output argument will be. Consequently, if the arrow arrives at a demux, then the code generator knows to find all the destinations and generate the appropriate number of `MPI_Send`s as needed. Similarly, if the arrow arrives at a mux, then the code generator must generate the appropriate number of `MPI_Recv`s at the input of the box on the other side of the mux.

Table 6-2. Summary of transformation rules for code generator.

Level	Transformation Rule
Top	1) Parse top-level system diagram into processes.
	2) Pull demux into preceding process.
	3) Pull mux into succeeding process.
Individual processes	1) Translate arrowhead/input to `MPI_Recv`.
	2) Bind and admit `MPI_Recv` buffer into VSIPL block. Generate view name based on input name and datatype. Use view in proceeding VSIPL functions. (Note: VSIPL view will have to be created in outer-loop code.)
	3) Translate DSP Blockset block to corresponding VSIPL function call. Map block arguments into VSIPL arguments. (Note: Create heavy-weight objects such as filter and FFT objects in outer-loop code.)
	4) Use in-place arguments, except for heavyweight functions. Generate output views for heavyweight functions based on input argument(s).
	5) Release block to `MPI_Send` buffer.
	6) Translate output/arrowtail to `MPI_Send`.

A simple example of how to apply these rules (summarized in Table 6-2) uses the process box that has been opened as illustrated in Figure 6-4. We will assume we have done the top-level transformation and now we are doing the process-level middleware generation.

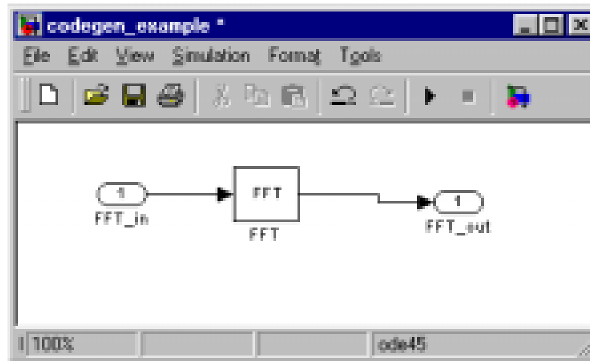

Figure 6-4. Simple Simulink model to illustrate code generation.

Simulink creates an input node inside the process model that maps to an arrowhead in the top-level canvas, which is mapped into the input node “`1`” and conversely so for the output node also labeled “`1`”. We re-label the nodes to reflect the name of the process, naming them `FFT_in` and `FFT_out`. We dragged the FFT icon over from the Simulink DSP Blockset. We follow the code generator’s transformation rules as follows:

1) Create `MPI_Recv` call with `FFT_in` as the buffer name and its size, datatype, and other parameters determined by what the designer enters into the dialog box.

2) Create VSIPL bind and admit calls:

   `vsip_cvrebind_f` –Bind MPI buffer.
   `vsip_cvadmit_f`–Admit MPI buffer; get view handle.

3) Create FFT call:

   `vsip_fcfftop_f`–Real in, complex out FFT done out-of-place; use view handle.

4) See (3) preceding.

5) Create VSIPL release call:

   vsip_cvrelease_f–Release view handle’s block to MPI buffer.

6) Create `MPI_Send` call with `FFT_out` as the buffer name, etc., as in (1) above.

Further details about the use of VSIPL and MPI are found in §B.2.

Chapter 7

Case Study:
Validating the MAGIC SDM Using a SAR Processor Application

In this chapter we validate that the MAGIC SDM can be used to accomplish the specification and design of a system representative of our ADoI. We choose the RASSP SAR benchmark (cf. §7.1) since it will be a level playing field on which to assess how our MAGIC SDM performs compared to the two main types of SDMs used with COTS MP technologies in our ADoI (cf. §2.2.2). The first type is virtual prototyping (VP), which is the specification and design of a digital system using an executable language such as VHDL. Virtual prototyping was found to be quite unwieldy for larger more complex applications like those found in this ADoI, because simulation runtimes were painfully long, and only those activities near the beginning of the hardware initialization cycle could be explored. For example, in the virtual prototyping of RASSP SAR, only the first 150 milliseconds of a 3-second frame could be simulated [2, 3]. The second type of SDM are deployable CASE frameworks (cf. §2.1.4.2), which have some model continuity and complexity control, but require the developer to commit to a hardware target *before* starting the design phase, the reverse of what the specification and design process should do.

We validate the MAGIC SDM empirically by showing the following claims are true:

1) The MAGIC SDM works as postulated, which means the rules can be followed and the tools work–especially in providing model continuity. This is indicated in this chapter by a "☞" at the beginning of the paragraph.

2) The MAGIC SDM can simulate complex system performance for whatever period is necessary. (It is able to simulate at least 20 times longer than a comparable VP simulation on the SAR benchmark.) This enables the designer to obtain a high

fidelity assessment of how well a candidate architecture and technology will do in meeting latency requirements.

3) The MAGIC SDM provides the framework to evaluate competitive technologies *prior to implementation*, which the CASE SDMs cannot do at all.

These are shown in this chapter by applying the MAGIC SDM to a real-world application of moderate complexity. This allows us to refine our MAGIC SDM rules and exercise our tools with a domain-representative and realistic application. We begin our case study by introducing the SAR processing benchmark. We then report on how we followed each of the MAGIC SDM rules and the efficacy of the tools we chose for the MAGIC SDM. Finally we report on the difficulties we encountered and how we responded to them.

7.1 RASSP SAR Benchmark Overview

The application we chose is the SAR benchmark used to evaluate competing RASSP methodologies [3]. The COTS MP technologies in our ADoI are often the technology of choice for implementation of SAR image processors, so the SAR benchmark is representative of the ADoI. Also, since it is the benchmark used in the RASSP program, artifacts exist that make this benchmark tractable for an individual researcher working in the public domain.

7.1.1 Application Domain for the RASSP SAR Benchmark

The SAR benchmark was a design exercise undertaken as a vehicle to assess performance of a RASSP-developed system. Application areas for these benchmarks were intended to present realistic challenges to RASSP as well as being of interest to a broad community of users. The application chosen for the first series of benchmarks was that of synthetic aperture radar (SAR). SAR is an important tool for the collection of high-resolution, all-weather image data and has application to tactical military systems as well as civilian systems for remote sensing. SAR can also be used to identify man-made objects in the ground or in the air. Such object identification typically requires SAR processing to be performed in real time by means of an embedded signal processor. The substantial computational throughput and memory requirements associated with image

formation processing alone make SAR a good application vehicle for use in benchmarking the RASSP design process. The eventual host for the SAR processor that could form images in real time was to be on board an uninhabited air vehicle (UAV). In order to develop and demonstrate the processor, radar data collected from the MIT Lincoln Laboratory Advanced Detection Technology Sensor (ADTS) was used. The ADTS is a Ka-band SAR sensor with on board data recording system, but had no existing real-time processor at the outset of the benchmarking program [158].

The requirements were published and made available to the public domain in a variety of formats, formally in [159] and informally in [2, 3, 6, 8-10, 72, 160-165]. The MIT Lincoln Laboratory RASSP web site[7] has been a rich source of relevant material, including a C-based executable specification and real-world data [164]. Corresponding to that is a MATLAB version of the executable specification that we obtained, both code and data [166].

The context of the radar system in which the SAR image processor was to operate is described in detail in these documents. In this dissertation we are not interested in SAR per se, but in its processing requirements, such as throughput and latency. Our interest is in the data format imposed by the ADTS to which our SAR processor must interface, and in the SWAP constraints imposed by the Amber UAV on the SAR processor. These requirements will be addressed in detail in the following two sections, §7.1.2 and §7.2.

7.1.2 SAR Processing Overview and Assumptions

Figure 7-1 shows a block diagram of the ADTS SAR processing system. The SAR processing to be accomplished by our processor as developed using the MAGIC SDM is shown in blue in Figure 7-1 (after Figure 5 in [158]). The post-processing is shown in green. After azimuth de-sampling and A/D conversion, data is recorded on tape for processing on the ground. In order to provide real-time data to the SAR image processor, the A/D output data will intercepted, buffered, and transmitted serially over a fiber optic link to the SAR processor. The input data frame is one of up to three

[7] RASSP Benchmarking Home Page (http://www.ll.mit.edu/llrassp/).

“polarizations,” where a polarization refers to the combination of transmitted and received electromagnetic wave polarizations during the data collection, e.g., “HH” (horizontal transmit, horizontal receive), “HV” (horizontal transmit, vertical receive), or “VV.” The data received for any given polarization will have 512 pulses, and each pulse is made up of 4064 real samples. The 4064 real samples are actually 2032 complex pairs in even/odd format. These real samples of a given polarization are what stream into the video-to-I/Q stage.

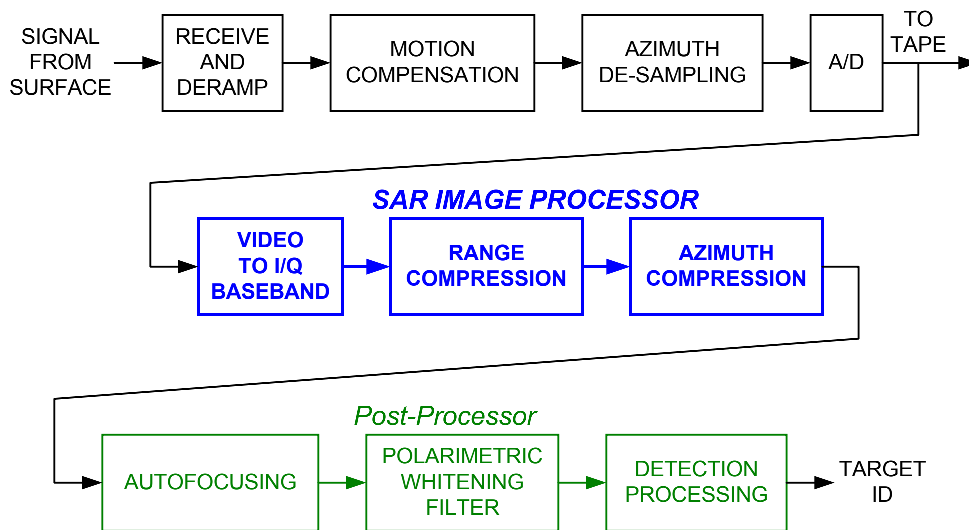

Figure 7-1. SAR block diagram with SAR image processor highlighted in blue.

The SAR image processing flow to be implemented by our processor is shown in Figure 7-2 (after Figure 3 in [167]). We describe this processing flow now, describing a few simplifications we had to make in order to make the case study tractable. The core processing is in blue. The inner-loop steady-state code is in bold, while the outer-loop setup code is shaded with a yellow background. There is some fix-to-float conversion and packet decoding that must be done with the A/D data. Our executable specification written in MATLAB included one full frame (512 pulses) of one polarization already decoded and stored in a MATLAB data file as a “data square,” i.e., a matrix of samples by pulses where rows 1 to 512 of column 1 are the returns of pulse 1, etc. In this format, we can bring it directly into the DSPW as we capture requirements and explore design alternatives with the MAGIC SDM. We therefore work under the assumption that our

processor's input data has already been so "prepared," which is reasonable since this data preparation accounts for only 3% of the baseline throughput as shown in Table 4 in [158]. We shall allow more than adequate margin to account for this.

We also assume certain constants, such as the video-to-baseband FIR filter order (N=8), which is reasonable since the only filter size used in the SAR benchmarking effort ended up being N=8. We assumed that the processor would always process whole frames (512 pulses) and not single strips (a single pulse). Constraints satisfied for 512 frames will more than satisfy a processor of single pulses since 512>>1. Making these assumptions meant doing extensive editing of the MATLAB model since it was created to be a flexible analysis framework. We require an executable specification that clearly captures our assumed fixed requirements, so we had to take care that when we streamlined it the correct processing was still accomplished. The edited code can be found in the Appendices. The MATLAB model became the key input to our specification because it is an executable specification verified with test data, and as such was the encapsulation of our computational requirements.

The fact that the polarization would be one of three possibilities only affects the post-processing software, which would integrate the different polarizations. It is therefore outside of our specification and design domain. With our simplifying assumptions of processing one polarization at a time of a full 512-pulse frame, we find ourselves operating under the typical model of two states, setup (highlighted in yellow in Figure 7-2) and steady-state (bold blue in Figure 7-2). This makes our SAR image processing system characterized as single real-time state, embedded, and data transformational. Hence, we will not have to capture multi-mode requirements. We will therefore not require the use of Stateflow in our requirements capture, just Simulink.

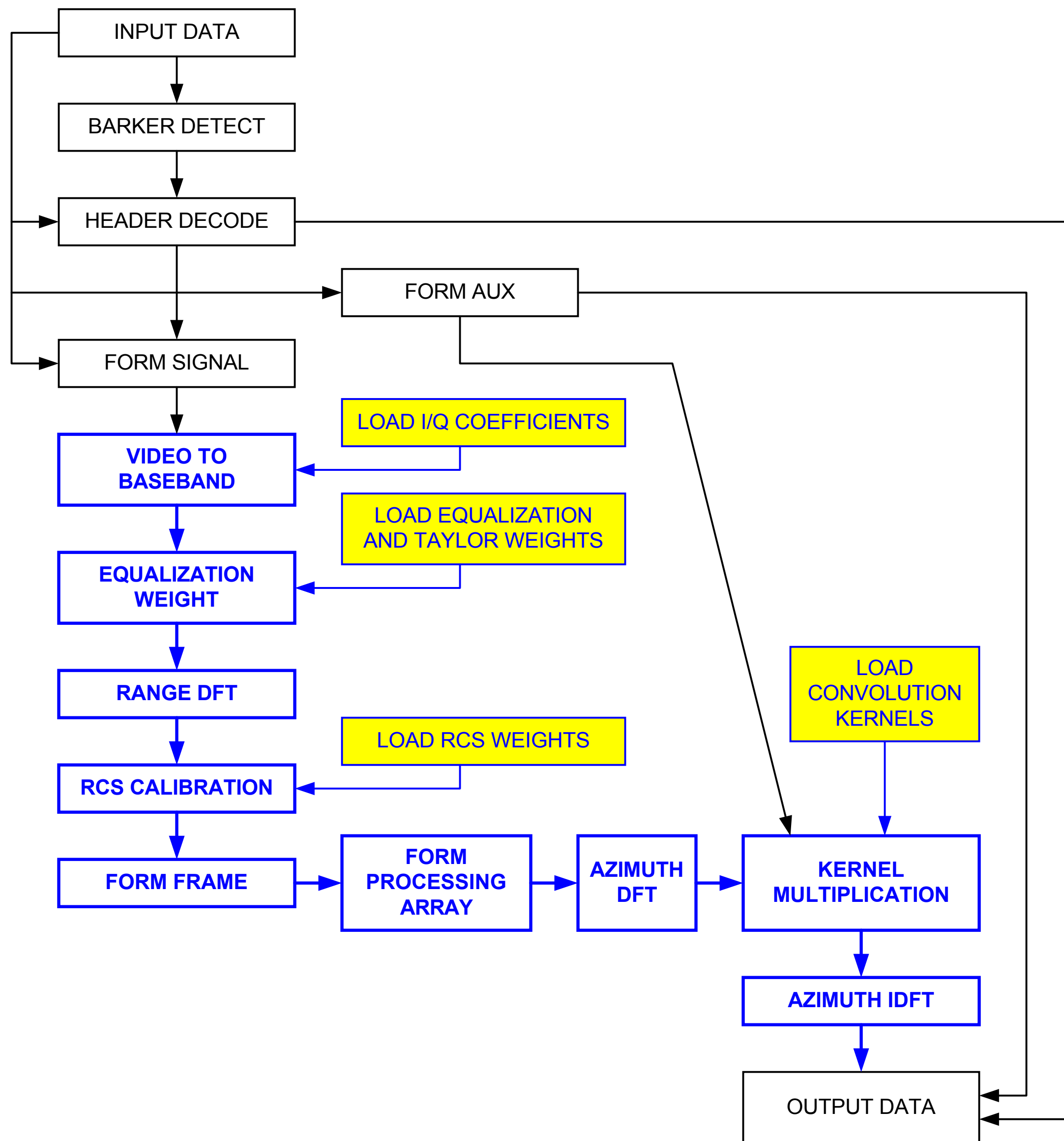

Figure 7-2. SAR image formation algorithm flow.

Assumptions we made for the design exploration were that a maximum of sixteen CEs would be available for implementation. This means the board count of a system would be six: one SBC, one custom I/O board for serial fiber channel conversion, and four multiprocessor boards. This would leave six slots available for growth and expansion (cf. Table 7-1). The limitation is also due to our chassis in the lab only having four quad-PPC boards.

Note that in the following sections (§7.2–§7.8), steps in which the MAGIC SDM establishes model continuity will be highlighted. The reader will be alerted to this by a "☞" at the beginning of the paragraph.

7.2 Tabulate Requirements

Of all the documents available that encapsulated the processor's requirements, we focused on [158, 168] because of their succinctness and [167] since it updated the processor requirements, executable specifications, and test data [159]. We also referred to [169] because of its additional detail and retrospection. We began by building an Excel workbook that captures the tabular data, culling from our requirements documents a summary of the SAR processing requirements tabulated as shown in Table 7-1.

Table 7-1. Summary of system requirements and constraints.

System Requirements and Constraints				
		Value	**Units**	**Comments**
Performance				
	Input Rate	18	MB/s	
	Output Rate	27	MB/s	
	Computational Complexity	3	GOP	3 GOP » 3*1024 MFLOPs = 3072 MFLOPs Note that baseline requirements total 1085 MOPs and enhanced requirements total 1957 MOPs. \ The 3 GOPs figure must include margin.
	Latency	3	s	
	Dynamic Range	103	dB	
	Modes	1 to 3 8 or 48	Polarizations FIR filter taps	Assume 1. Assume 8.
Non-Performance				
	Size	2.2	ft^3	10.5"H x 20.5"L x 17.5"W
	Weight	60	lbs	
	Power	500	W	
	Data Storage	80	MB	
	Interface	N/A	Bit-serial fibre	
	Frame size	2048x512	pixels	
	Scalability	2	x	Also want to allow for a 4-slot 6U VME chassis.
	Testability	N/A	N/A	"Best practice"
	Environment	N/A	N/A	UAV: non-condensing, air cooled
	Assumed Quantity	500	units	

We have a priori knowledge that a COTS MP solution is desired and so we want to establish a baseline and boundary for our options. The SWAP constraints and scalability tell us that the largest COTS 6U VME chassis that will work is a 12-slot

version [170]. We note that there is a 1.75"H difference between the requirements and a 21-slot version, which could probably be negotiated into acceptability. This could be done in the UAV or even a redesign of the 21-slot version, especially since the expected quantity of SAR processors is 500 units. However, we will constrain our consideration initially to the 12-slot chassis with dimensions of 10.5"H × 17.0"L × 16.25"W and a power supply rated at 500 Watts.

Note that since our chassis satisfies the power constraint of the processor, we do not need to consider it any further as long as our total board count remains no more than twelve. We did not appreciate this when we began, and aggressively gathered electromechanical specifications of candidate technologies into our system workbook, including the development of a spreadsheet that would compute SWAP given any combination of processor daughtercards and motherboards. While this level of computational support will probably not be needed in our particular case study, it is still useful for the specifier and designer to maintain a database for more demanding system designs.

We should also note that while the environment requirement was given as "non-condensing, air cooled," specific relative humidity requirements and minimum airflow requirements were not given. The following constraint requirements were also not given in the SAR processor documentation:

- Operating temperature range
- Storage temperature range
- Operating altitude range
- Reliability
- Maintainability

This can be explained by noting that the SAR benchmark was for evaluating competing methodologies, hence these characteristics were not specified. If the SAR processor were to go into manufacturing and deployment, these requirements would surely not be don't-cares. Regardless, we accounted for these characteristics that we collected in the SWAP worksheet of our executable workbook.

Table 7-2. Environmental non-performance characteristics of processor boards.

Boards																
Type	***Manufacturer***	***Item***														
			Size				Weight	Power	Min Airflow	$T_{operating}$		$T_{storage}$		Relative Humidity (%)		Operating Altitude
			VME	Height	Width	Thickness				(ºC)		(ºC)		non-condensing		(ft)
				(in)	(in)	(in)	(lbs)	(W)	(CFM)	Min	Max	Min	Max	Min	Max	Max
Multiprocessing Motherboards																
	Mercury Computer Systems															
		MCH6	6U	9.180	6.290	0.800	0.800	10.000	17.000	-20.000	40.000	-40.000	85.000	10.000	90.000	10000
		MCH9	9U	15.750	14.440	0.800	3.000	18.000	25.000	-20.000	40.000	-40.000	85.000	10.000	90.000	10000
Multiprocessing Daughtercards																
	Mercury Computer Systems															
		P2A16BA	1/2*(6U)	4.435	5.000		0.210	11.000		0.000	40.000	-40.000	85.000	10.000	90.000	6000
		P2A8BA	1/2*(6U)	4.435	5.000		0.210	11.000		0.000	40.000	-40.000	85.000	10.000	90.000	6000
		P2A64BD,I	1/2*(6U)	4.435	5.000		0.210	11.000		0.000	40.000	-40.000	85.000	10.000	90.000	6000
		P2A32BD,I	1/2*(6U)	4.435	5.000		0.210	11.000		0.000	40.000	-40.000	85.000	10.000	90.000	6000
		P2A16BD,I	1/2*(6U)	4.435	5.000		0.210	11.000		0.000	40.000	-40.000	85.000	10.000	90.000	6000
		P2A8BD,D	1/2*(6U)	4.435	5.000		0.210	11.000		0.000	40.000	-40.000	85.000	10.000	90.000	6000
		S2T16BD	1/2*(6U)	4.435	5.000		0.210	11.000		0.000	40.000	-40.000	85.000	10.000	90.000	6000
		S2T8BD	1/2*(6U)	4.435	5.000		0.210	11.000		0.000	40.000	-40.000	85.000	10.000	90.000	6000
		S2T32BD	1/2*(6U)	4.435	5.000		0.210	11.000		0.000	40.000	-40.000	85.000	10.000	90.000	6000
		S2T64BD	1/2*(6U)	4.435	5.000		0.210	11.000		0.000	40.000	-40.000	85.000	10.000	90.000	6000
Multiprocessing Interconnection																
	Mercury Computer Systems															
		ILK1	P2	3.740	0.760	2.100	0.125	0.500		-20.000	50.000	-40.000	85.000			
		ILK4	P2	3.740	3.120	1.900	0.313	1.750								
		ILK8	P2	3.740	6.320	1.900	0.875	10.000								
		ILK12	P2	3.740	9.520	1.900	1.250	13.000								
		ILK16	P2	3.740	3.740	1.900	1.625	13.000								
Host Single Board Computer																
	FORCE															
		8VT	6U	9.180	6.290		11.188	38.300		0.000	55.000	-40.000	85.000	5.000	95.000	3000
Bit-Serial Interface Board																
	Custom															
		Custom	6U	9.180	6.290		1.220	35.000		0.000	55.000	-40.000	85.000	5.000	95.000	

We have now tabulated our requirements with the emphasis on the non-performance constraints, since the tabular format is most suited for this type of discrete data. The communications and control requirements are tabulated. We have our computational requirements explicitly in equation form with coefficients tabulated, especially in [158] and [167]. However, since as we noted previously in §7.1.2, we have the computational requirements contained in an executable MATLAB format, we now transition into capturing the non-constraint requirements in an executable Simulink model.

☞ As we do this and subsequent MAGIC process tasks, we will be referring to and referencing data in our executable workbook, making it a key component in establishing model continuity. For now we use it as an executable depository for the non-performance constraints that bound our architectural options.

7.3 Capture Non-Constraint Requirements in an Executable Model

Initially we lay out a single-threaded version of the SAR processor in Simulink, using one block for each algorithm. Each algorithm will become a process running on one or more processors, which is a simple pipelined model. After we are sure this model is correct, we can begin parallelization. We are basically translating the MATLAB code into a Simulink model. We are translating one executable specification into another in order to have a specification model that we can translate into a system design.

7.3.1 Non-Parallel Pipelined Model

☞ Our first cut of a pipelined non-parallel Simulink SAR model is shown in Figure 7-3. Details of initializing the MATLAB workspace and data input stream are discussed in §B.1.1. This is another example of model continuity, using part of one executable specification (MATLAB) directly in another (Simulink). Maximizing use of one executable specification in another minimizes transcription error.

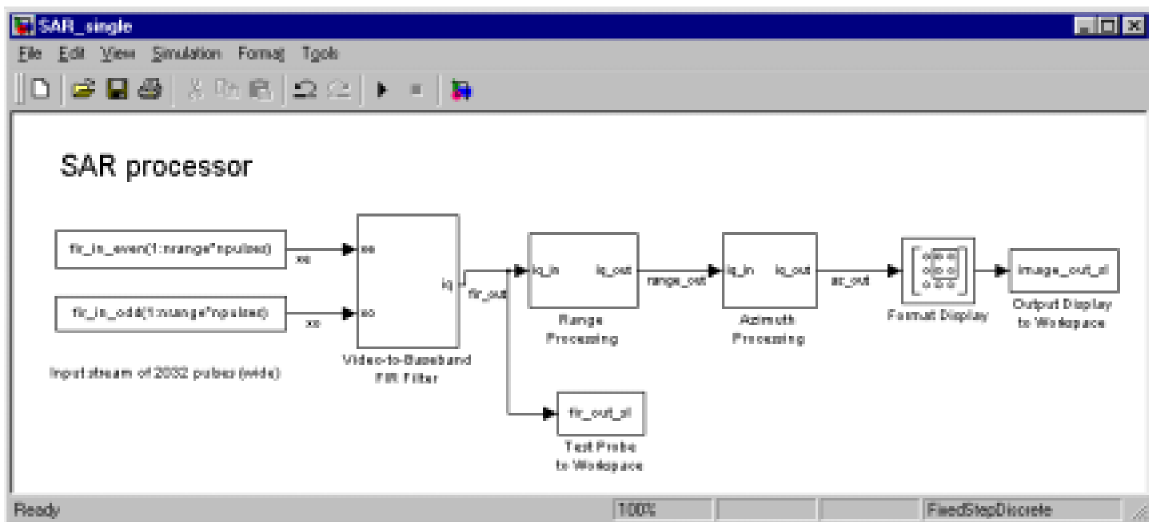

Figure 7-3. Simulink model of single threaded version of our SAR processor.

The first block(s) in the Simulink model is for data input. There is one for both the even and odd samples, which come from the data formatted for the MATLAB model as two data squares, one for the even (in-phase; "I"; real) samples and one for the odd (quadrature; "Q"; imaginary) samples. More detail on this operation is given in §B.1.2.

The next block of our model is the video-to-baseband FIR filtering, which is shown in Figure 7-4. The modmask block multiplies the even and odd samples by an alternating +1, -1, +1, … series, which modulates these two data streams. They are passed through a FIR filter and combined to form complex samples. The 8-sample transient is stripped off and then formed into a matrix for output to the next Simulink model block.

☞ Another example of model continuity is how the MATLAB executable specification is used in the Simulink model for FIR filtering. The coefficients come from a MATLAB data file and they are addressed using a MATLAB expression in the Simulink FIR filter block. This MATLAB expression and Simulink block directly translate into efficient VSIPL code as will be shown in §7.6.1 and §B.2.

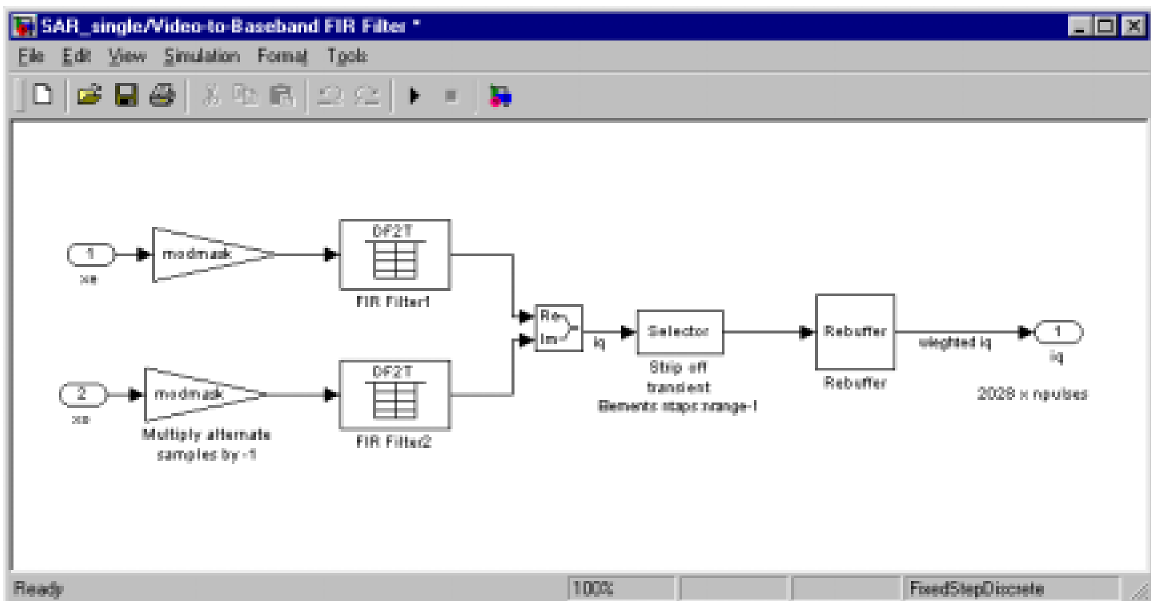

Figure 7-4. The model for video-to-baseband conversion.

The range compression processing block takes the complex data square that has the dimensions after filtering of (`nrange-ntaps`) × `npulses`. This block is shown in Figure 7-5. The Scale Rows block applies the Taylor weights to the range samples of the pulses. It does this as a diagonal matrix multiply times the data square, functionally equivalent to an element-by-element multiplication of two vectors. The output is zero padded since there are not a power-of-2 samples in the pulse columns, padding the 2024

range samples with 24 zeros per pulse, producing a 2048 × `npulses` matrix. The FFT block performs a 1-D DFT on each of the columns (pulses) of the data square, a process known as "pulse compression." The RCS weights are applied to the compressed pulses for compensation reasons also using the Scale Rows block.

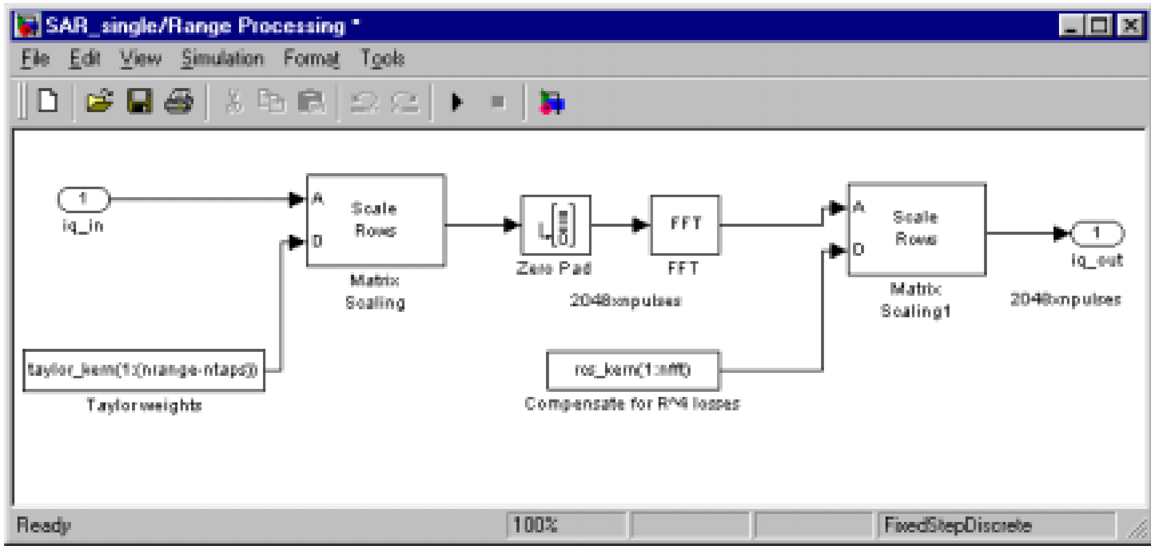

Figure 7-5. Range processing block of our Simulink SAR model.

The azimuth compression processing block as shown in Figure 7-6 is computationally more demanding than range processing. This is the stage where the data square is complemented by a matrix of the same size filled with complex zeros, forming a frame that is transposed (non-Hermetian) to perform "cross-range convolution" by computing column-wise DFTs (FFT block) across the range samples of the pulses. Azimuth kernel coefficients are then applied to the DFT outputs, followed by inverse DFTs (IFFT block).

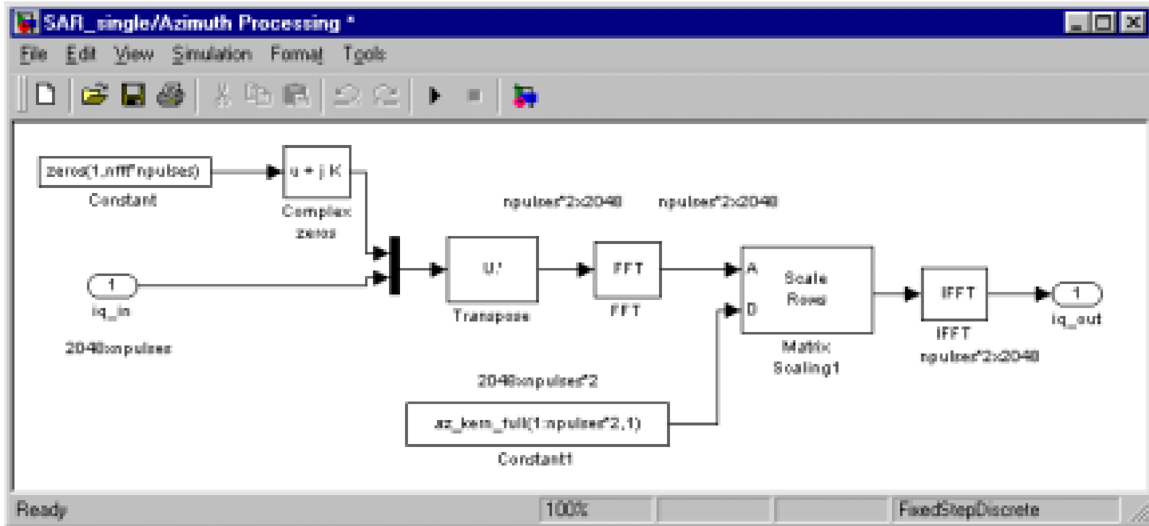

Figure 7-6. Azimuth processing block of our Simulink SAR model.

The azimuth compression processing output is formatted in the format display block (cf. Figure 7-3), which basically strips off what would be the older samples. More detail on this operation is given in §B.1.3. The SAR image processor output is stored in the MATLAB workspace where it is displayed using the following MATLAB call via the Simulink exit (`StopFcn`) command:

```
"Sim_out=Sim_SAR_display(image_out_sl,npulses,nfft,2);"
```

While displaying the image is interesting, it is more pertinent to test the output against "ground truth." This is a data set that is known to accurately represent the area being imaged by the SAR processor. In our case we used our MATLAB executable specification since it had already been validated against the RASSP SAR data set. The test code for validating our Simulink model's output against the MATLAB model's computed "ground truth" was simple:

```
function out=image_compare(image1,image2)
% Compare two 2D complex images:
%     out=image_compare(image1,image2)
%
% Inputs:  Two 2D complex images (image1,image2)
%
% Output:  Max difference of pixel magnitudes.

diff=image1-image2;
out=max(max(abs(diff)))
```

We obtained a `diff` value of zero, thus assuring us that our non-parallelized Simulink model was an accurate executable specification. More details on executing the specification are provided in §B.1.4.

7.3.2 Parallel Pipelined Model

When we had validated that our simple pipelined non-parallel model was correct, we could then begin to parse it into a parallel model to support the exploration of design alternatives while assuring we still had an accurate requirements specification model. The single-threaded Simulink model could be viewed as the specification for a pipelined architecture of four processes to be run on one processor per process as shown in blue in Figure 7-7. We choose to map the FIR processing to a single processor. This is a reasonable start since typically input data streams into the COTS MP architecture by streaming into the local memory of one of the compute elements (CEs). We also know that we can achieve some concurrency at this node controlling the streaming input by performing the video-to-baseband conversion on the data as it comes in one pulse at a time, or some other similar implementation strategy. Consequently, we defer those details to the implementation and search for parallelism in the heavy-weight processes of range and azimuth compression.

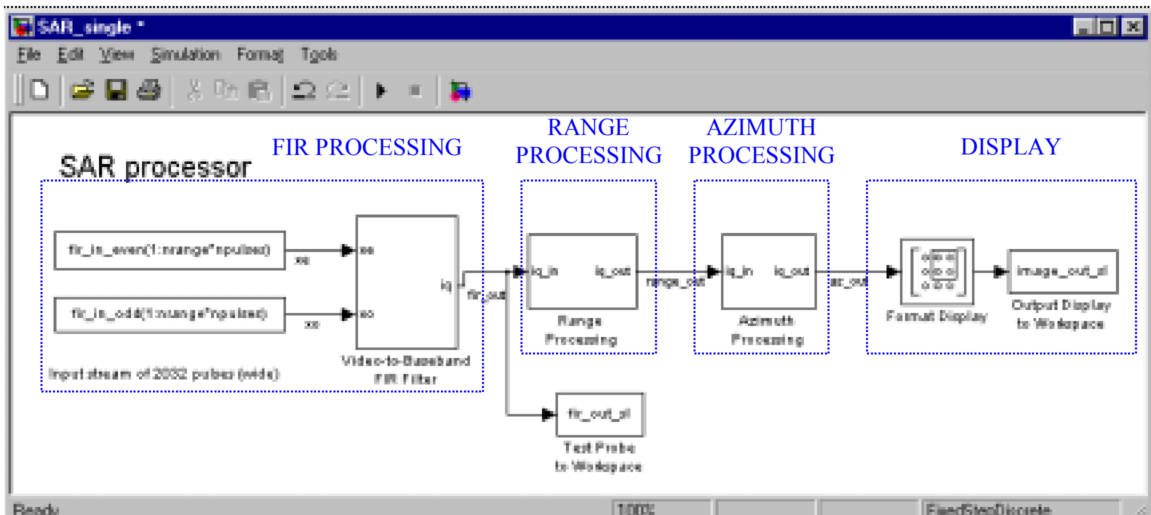

Figure 7-7. Parsing the SAR processor into separate processes.

We call range and azimuth processing heavy-weight because of their computationally intensive processing dominated by the DFT (and IDFT for azimuth processing). They are also excellent candidates for SPMD parallelization due to the coarse granularity of the data. We will be operating on columns (pulses) of the matrix (data square) and can distribute groups of columns ("tiles") of the matrix (data square) to the range and azimuth processing. The only catch here is that we need to operate on the rows (range samples) in azimuth processing. This will require the non-Hermetian matrix transpose to be a standalone process that performs the transpose of the range-processed matrix, or in radar terms, this process does a "cornerturn of the data square."

Distributing the input data that has been mixed down to baseband (output of FIR Processing) is a scatter process, sending tiles of the data square to a number of processors that will perform range compression processing on their tile in parallel. After they are done they will all send their results to the cornerturn process that collects these tiles, which is a gather function. After doing the matrix transpose of the contiguous current and last frame (zeros in our case), the cornerturn scatters tiles to the azimuth processors. The azimuth processors perform azimuth compression, then send their results to a display process that gathers these tiles and forms the final image. In parallel processing jargon this a one-to-many-to-one-to-many-to-one processing model. In Simulink where we choose four processors for range processing and eight processors for azimuth processing, the model appears as shown in Figure 7-8.

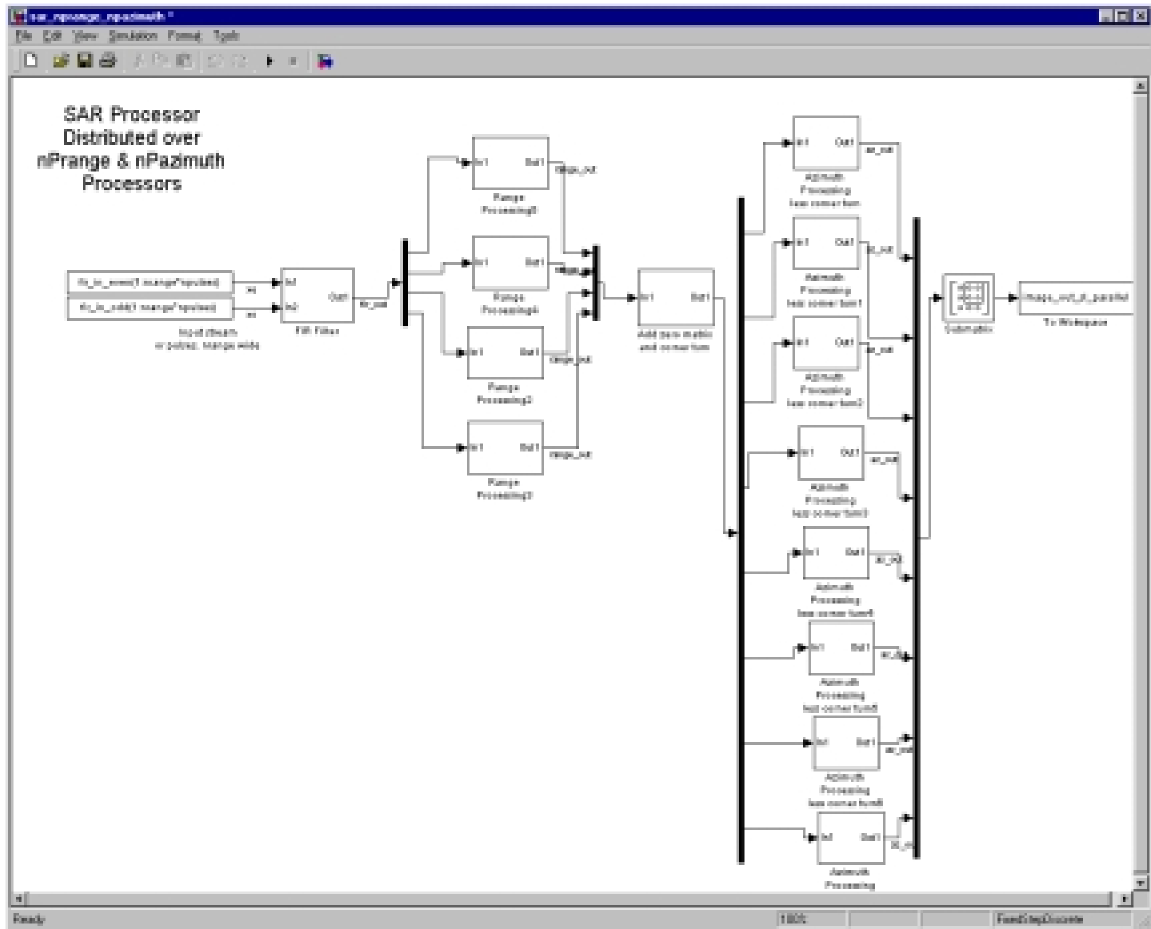

Figure 7-8. Parallelized SAR processing Simulink model
where *`nPrange`*=4 and *`nPazimuth`*=8.

It was fairly straightforward to implement the parallelized version of the SAR processor since the single-thread version was working and had system variables in the parameter blocks. It became just a matter of scaling parameters by the number of range processing processors (`nPrange`) and azimuth processing processors (`nPazimuth`). The new DSP Blockset blocks introduced in Figure 7-8 are the demux and mux blocks that perform scatter and gather, respectively. Between FIR filtering and range processing is a demux block that we use for scattering tiles to the range processors. The converse (mux/gather) is performed between the range processors and the cornerturn. More details on the use of these blocks are found in §B.1.5.

Pivotal between the range and azimuth processing is the cornerturn that gathers the range data, append the previous frame, perform a non-Hermetian transpose, then scatter the transpose to the azimuth filters. It is shown in Figure 7-9.

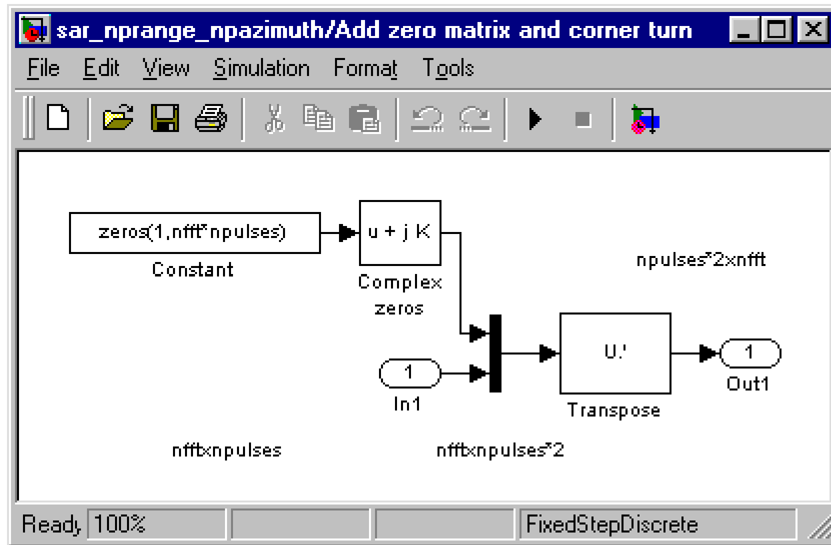

Figure 7-9. Cornerturn required when parallelizing range and azimuth processing

☞ The architecture can be changed by editing the Simulink model and the startup MATLAB script where `nPrange` and `nPazimuth` are defined. We can now use this model to do two important functions:

- Generate middleware code whose functions and parameters can be used for computing token delays in performance modeling to see if architectures are viable implementation candidates or not.
- Compute test vectors by tapping the data flow and sending it to the workspace where it can be saved and used later in implementation for verification of a node's input and/or output.

According to our methodology, we use the Simulink model to generate middleware and associated token delays for performance modeling with a given technology to find the optimal architecture. We do this by iterating through the performance modeling framework (eArchitect) and DSPW as necessary until we find the optimal architecture. When we do we will repeat this process for another technology as discussed in §7.6. Note that this is an excellent example of strong model continuity.

7.4 Build Executable Workbook with Requirements

We introduced this step in §7.2 as we began the process of culling pertinent requirements from the documentation. This becomes more important as the specifier must evaluate different technologies and quantify delay tokens in the performance modeling to be done in §7.6. Our workbook grew to 15 worksheets–and we only had access to a single vendor's VSIPL and MPI data! The worksheets contained the following data and computational worksheets:

1. Token Quantification–Worksheet computing MPI and VSIPL token delays based on benchmarks contained in other worksheets and the given architectural configuration:
    - One processor for data input and FIR processing
    - `nPrange` processors for range processing
    - One processor for the cornerturn
    - `nPazimuth` processors for azimuth processing
    - One processor for the azimuth gather and display processing
2. Token Summary–Summary of MPI and VSIPL aggregate clock cycle delays for consumption by eArchitect.
3. MPI-Pro PPC 200MHz–MPI benchmarks for point-to-point data reads/write latencies on Mercury PPC RACEway MP boards for one to 16777216 bytes; eArchitect evaluates the network latencies.
4. VSIPL PPC750 292MHz–VSIPL benchmarks for the Core-Lite profile as implemented by Mercury Computer Systems and run on their 292MHz PPC750 CE.
5. Constraints–cf. Table 7-1.
6. SWAP–cf. Table 7-2.
7. Reliability–Computational worksheet to estimate MTBF of systems implemented with a standard VME chassis, FORCE 8VT SBC, and some combination of Mercury motherboards and daughtercards.
8. MCS MTBF Computation–Of combination of Mercury motherboards and daughtercards from their respective FPMH data (following).

9. MCS Motherboard FPMH–Measured failures per million hours (FPMH) for Mercury motherboards.
10. MCS Daughtercard FPMH–Measured failures per million hours (FPMH) for Mercury daughtercards.
11. SAL PPC750 292MHz–Benchmarks for Mercury's entire Scientific Algorithm Library (SAL) run on their 292MHz PPC750 CE.
12. SAL PPC603 200MHz–Benchmarks for Mercury's entire Scientific Algorithm Library (SAL) run on their 200MHz PPC603 CE.
13. Single PPC Estimation–Using the token delays from worksheet #1 above to estimate throughput of the entire SAR processing if run on a single PPC CE.

7.5 Gather Benchmarks for Tokens

As listed above in §7.4, our executable requirements workbook contains our benchmark data. This is key to performing design analysis, because it allows us to compute accurate token delays for our eArchitect network simulations, thus obtaining realistic benchmarks of our candidate implementations *without having to prematurely commit to a given COTS MP vendor*. The eArchitect framework is a high-fidelity performance modeling framework, but as we found out for some applications, the network traffic does not have a first order effect on the throughput of a candidate architecture, making it all the more important that the benchmark data be accurate.

We were only able to gather one set of COTS MP benchmarks for VSIPL. Mercury Computer Systems was willing to release an early (pre-beta) set of VSIPL measurements, and they were only run on the PPC750 292MHz CE. Since we only had VSIPL benchmarks for Mercury's platform, we obtained only the benchmarks for MPI on Mercury hardware. We were able to get benchmarks for just the Mercury PPC from MSTI's MPI/PRO, which was for a 200MHz PPC. The clock difference was not a problem for the architectural reasons outlined in §7.9.2.

7.6 Explore Alternative Architectures and Technologies

In this step, we use "generated" middleware code and performance modeling to estimate the latency of candidate architectures by iterating with one candidate technology

to find the optimum architecture for that given technology. We repeat this with other potential technologies. The optimum architectures are then compared and the “best of the best” is chosen as the architecture and technology for implementation. In our case study we are limited to one technology, Mercury Computer Systems technology, which is more than adequate for exercising our MAGIC SDM.

7.6.1 VSIPL Code Generation

We generated middleware that the DSPW RTW should generate. This was straightforward. As we showed in §6.5.3.2, mapping the DSP Blockset to VSIPL is well-correlated with over 50% common functionality, and the commonality are the functions common to applications like those in our ADoI. For SAR processing there was no common functionality lacking; i.e., all of the Simulink 3 DSP Blockset blocks used to implement our SAR processor have VSIPL functions with which they can be implemented.

Common to all of the blocks in both the SAR processors, single-thread (cf. Figure 7-3) and parallel (Figure 7-8), are the “in” and “out” nodes that interface the blocks to the preceding and proceeding blocks (Figure 7-4–Figure 7-6). The input nodes are translated into VSIPL support functions for “binding” or “rebinding,” then “admitting” (cf. §A.1.2.2). The output nodes are translated into the VSIPL function to “release” the VSIPL block. These are the functions that bring the data out of user space and into VSIPL space (cf. §A.1.2.2). These must be called for each of the processes. We were not able to get benchmarks for these functions, but since their throughput is much less than even the lightest weight VSIPL computation, we consider their effects negligible. Also, all VSIPL functions translated are `vsip_<function>_f`, where the `_f` indicates the function is typed for single-precision floating point arguments.

The video-to-baseband block shown Figure 7-4 translates the modmask into a VSIPL `vsip_vmul_f` function that does an element-by-element (“el-wise”) multiply between two real vectors. Actually, there is a more efficient way by being a little clever with VSIPL and using the `vsip_vput_f` and `vsip_vneg_f` functions that negate every other element. Since such subtlety would probably not be written into a code generator,

we will assume we perform the more straightforward generation of `vsip_vmul_f`. We generate the `vsip_firflt_f` for the FIR filter block. We absorb the Selector block into it since the output arguments of the `vsip_firflt_f` can accomplish the same thing. the Re-Im combine block and Rebuffer block can similarly be combined into a single `vsip_vcmplx_f` VSIPL function due to flexibility of the output arguments. These latter two VSIPL code "generations" may be also be too subtle to write into a code generator. If so, a style guide would need to be written to sensitize the MAGIC SDM user to such VSIPL-friendly Simulink and DSP Blockset block combinations.

The other Simulink SAR hierarchical blocks (range, cornerturn, and azimuth) are more straightforward since they are all operating column-wise on complex matrices. In the range compression processing block (cf. Figure 7-5), the Matrix Scaling and Zero Pad blocks are translated into the `vsip_cvmmul_f` for complex el-wise vector-matrix multiplication, i.e., multiplying the baseband data columns by the Taylor weights. The zero padding can be done by judiciously sizing and initializing the matrix before the baseband data is stored in it. If this is too subtle for the code generator, a `vsipl_cmfill_f` could do the complex matrix fill with complex zeros, which is not too time demanding. The FFT block is similar to the MATLAB column-wise DFT operation, performing a DFT on each column of the matrix. VSIPL has this function, called `vsip_ccfftmop_f` where `cc` denotes complex-to-complex, `m` refers to multiple 1D, and `op` refers to out-of-place. The last Matrix Scaling also translates into a `vsip_cvmmul_f` for multiplying the compressed range cells by the RCS kernel.

The cornerturn (cf. Figure 7-9) is generated into a single `vsip_cmtrans_f`, where the appending of the previous frame does not require a run-time VSIPL computation, but a sensible set-up to allow the two frames to be contiguous. The azimuth processing (cf. Figure 7-6) uses the `vsip_ccfftmop_f` and `vsip_cvmmul_f` calls just as in range processing, but with different argument counts, which use to index the function in the table of benchmarks, which is discussed further in §7.6.3.

There are some subtleties involved in generating some VSIPL code from a Simulink DSP Blockset model description that are beyond the scope of our current discussion. We document these details in §B.2.

7.6.2 MPI Code Generation

The generation of the MPI code will require a different code generation strategy than the VSIPL code generation. The VSIPL code generation was largely a one-to-one or two-to-one mapping of Simulink and DSP Blockset blocks to VSIPL functions. While there are only two MPI calls to be generated in our SDM (`MPI_Send` and `MPI_Recv`), the RTW code generator has to do more tracing through the Simulink model data paths to generate the MPI function calls and their arguments.

Recall the parallel SAR processing architecture in Figure 7-8, the code generator must trace the data path from a source process through a mux or demux to its sink or vice-versa. The determination of the route and arguments is not trivial, but is not intractable either. All of the CASE SDMs considered in our research have the capability to generate this kind of code. So we assume that the RTW could similarly generate the correct arguments for `MPI_Send` and `MPI_Recv` as inputs to the implementation specification, and that for now, it can readily compute latencies by evaluating the block parameters in the processing blocks as well as the mux and demux blocks.

7.6.3 Latency Estimation

We now present the summary of our latency estimation. We actually have two types of latency estimation, with and without accounting for interprocessor communication (IPC) over the high-performance interconnect (e.g., RACEway, Myrinet, and SKYchannel). Even with parallelization, a multiprocessor-based architecture may still be dominated by computation and not communication. In this case, a first order estimate of the latency may be obtained without the rigor (and time and effort!) of performance modeling. However, this will only be a lower bound of a potential architecture since IPC will add to the system's latency, though it will be indicative of how the different architectures will compare. The performance model simulation will

give us a high-fidelity “second-order” estimate of latency for our system architectures being considered.

7.6.3.1 Latency Without Accounting for IPC

We began by building a worksheet in our executable workbook that linked to the benchmarks and scaled them, based on the following system values:

Table 7-3. Example of system parameters portion of Token Quantification worksheet.

		Excel Link Flag	Comments
Simulink SAR System Parameters			
		Off	⇐ MLEvalString("load system_parameters;")
		Off	⇐ MLGetMatrix("system_parameters","b5")
npulses=	512		# Azimuth Pulses = [1:512]
ntaps=	8		# FIR filter taps $\in$ {8,48}
nrange=	2032		# Range Samples = 2032
nfft=	2048		Size of FFT = $2^\lfloor \log_2(\text{nrange}) \rfloor$
nPrange=	4		# Processors for Range Processing
nPazimuth=	8		# Processors for Azimuth Processing
nrange'=nrange-ntaps=	2024		# Range Samples after FIR filtering

We used Excel Link to link the Simulink SAR model, so as we manipulated the model its architectural parameters would be reflected in the Token Quantification Worksheet. These commands are the `ML*` commands in the comments column. They were turned off at the time this snapshot was taken. The complete worksheet for this architecture appears in Figure 7-10.

Figure 7-10. Token Quantification worksheet from executable workbook (`nPrange`=4, `nPazimuth`=8).

The purpose of this figure is to show the level of detail both in the worksheet and the workbook. The data must be typed since MPI function benchmarks are a function of block size, and complex data requires twice the storage as real data. Though it is a little difficult to read in Figure 7-10, columns L through Q contain the logic of the benchmark computation. Benchmarks were used “as is” if the size of the generated middleware function was contained in the benchmark table (LUT). If not, the benchmarks were scaled or interpolated. There were a few VSIPL functions that were not in the Core-Lite profile

that Mercury benchmarked, in which case we interpolated by using SAL and VSIPL benchmarks.

Table 7-4. Example of token quantification and non-IPC latency computation (`nPrange`=4, `nPazimuth`=8).

Token Delay Computations		(steady-state)							
Stage	Type		ℝ	ℂ	#	El-type	Function	Total Time (ms)	**Clock Cycles per Call (200MHz)**
Video-to-Baseband									
	Comm	Get (2*nrange)*npulses real samples	1	0	8323072	bytes	MPI_Recv	52.038	**10407683**
	Comp	Bind&admit reals samples block (negligible)							
	Comp	2 el-wise vector multiplies per pulse for demodulation	1	0	2*512	pulses	vsip_vmul_f	119.020	23803986
	Comp	2 FIR filter calls of nrange samples each	1	0	2*512	calls	vsip_firflt_f	92.192	18438368
	Comp	Combine FIR filter outputs into complex data square	0	1	1036288	samples	vsip_vcmplx_f	81.141	16228270
	Comp	Release complex data square (negligible)							
	Comp	**VSIPL Total**							**58470624**
	Comm	Scatter nPrange nrange'×(npulses/nPrange)-sample tiles	0	1	2072576	bytes/tile	MPI_Send (*nPrange)	52.120	**2606011**
Range Processing		(per Range processor)							
	Comm	Get nPrange tile (nrange'×(npulses/nPrange) samples)	0	1	2072576	bytes/tile	MPI_Recv	13.030	**2606011**
	Comp	Bind&admit nrange'×(npulses/nPrange) samples block (negligible)							
	Comp	El-wise vector multiply for each pulse w/ Taylor weights	0	1	128	pulses	vsip_cvmul_f	28.757	5751398
	Comp	Range DFT's	0	1	128	pulses	vsip_ccfftop_f	59.558	11911642
	Comp	El-wise vector multiply for each pulse w/ RCS weights	0	1	128	pulses	vsip_cvmul_f	28.757	5751398
	Comp	Release nrange'×(npulses/nPrange) samples block (negligible)							
	Comp	**VSIPL Total**							**23414439**
	Comm	Put nPrange tile (nrange'×(npulses/nPrange) samples)	0	1	2097152	bytes/tile	MPI_Send	13.185	**2636912**
Cornerturn									
	Comm	Gather nPrange nrange'×(npulses/nPrange)-sample tiles	0	1	2097152	bytes/tile	MPI_Recv (*nPrange)	52.738	**2636912**
	Comp	Bind&admit nfft´npulses samples block (negligible)	0	1					
	Comp	Append data square with last frame. Complex non-Hermitian matrix transpose. $[2048´(2*512)]^T \Rightarrow (2*512)´2048$	0	1	2048´(2*512)	complex	vsip_cmtrans_f	170.378	34075638
	Comp	Release (2*npulses)×nfft samples block (negligible)	0	1					
	Comp	**VSIPL Total**							**34075638**
	Comm	Scatter nPazimuth (2*npulses)×(nfft/nPazimuth)-sample tiles	0	1	2097152	bytes/tile	MPI_Send (*nPazimuth)	105.476	**2636912**
Azimuth Processing		(per Azimuth processor)							
	Comm	Get (2*npulses)×(nfft/nPazimuth)-sample tile	0	1	2097152	bytes/tile	MPI_Recv	13.185	**2636912**
	Comp	Bind&admit (2*npulses)×(nfft/nPazimuth) samples block (negligible)							
	Comp	Range DFT's	0	1	256	pulses	vsip_ccfftop_f	56.184	11236864
	Comp	El-wise vector multiply for each pulse w/ azimuth convolution kernel	0	1	256	pulses	vsip_cvmul_f	29.098	5819597
	Comp	Range IDFT's	0	1	256	pulses	vsip_ccfftop_f	56.184	11236864
	Comp	Release (2*npulses)×(nfft/nPazimuth) samples block (negligible)	0	1					
	Comp	**VSIPL Total**							**28293325**
	Comm	Put (2*npulses)×(nfft/nPazimuth)-sample tile	0	1	2097152	bytes/tile	MPI_Send	13.185	**2636912**
Display									
	Comm	Gather nPazimuth (2*npulses)×(nfft/nPazimuth)-sample tiles	0	1	2097152	bytes/tile	MPI_Recv (*nPazimuth)	105.476	**2636912**
Approximate Latency (ms)								**1142**	

We repeated this iteration for `nPrange`=1,2,4,8 and `nPazimuth`=1,2,4,8, values reflecting that our data squares are characterized by power-of-2 dimensionality. The

summary of these iterations appears in Table 7-5. We can also plot these values as a surface plot, with the latency as a function of the processor counts of range and azimuth processing, which is shown in Figure 7-11. This is useful because it lets us know where to start in considering architectures. Recall we had a 3-second maximum latency requirement as well as a scalability requirement of 2, so we begin to look for architectures that deliver under a (3-second/2)=1.5-second latency. Those that do not are shaded in a revised version of Table 7-5 shown in Table 7-6.

We also note from Figure 7-11 that we start to get diminishing returns in the area of 8 CEs each for range and azimuth. This is not an option for us because we are constrained to 16-3=13 CEs for range and azimuth due to our imposed 16-CE limit and because 3 CEs are required for FIR filtering, the cornerturn, and display processing. Regardless, the 8-8 data point is illustrative. So, at this point we are restricted to some combination of range and azimuth processors whose total is less than 13 and do not fall into the shaded region of Table 7-6. We investigate these architectures in the following section, §7.6.3.2.

Table 7-5. Token Summary for performance modeling, including latency estimates without accounting for IPC.

Compute Node:	PPC603e					
Clock Frequency:	200MHz					
Clock Cycles for eArchitect Token Delays						
		proclist	nPrange or nPazimuth			
Process	Instructior	index	1	2	4	8
	MPI_Recv	n/a	10407683			
Video-to-Baseband	VSIPL	n/a	58470624			
	MPI_Send	nPrange	10424042	5212021	2606011	1303005
	MPI_Recv	nPrange	10424042	5212021	2606011	1303005
Range Processing	VSIPL	nPrange	93657756	46828878	2.3E+07	1.2E+07
	MPI_Send	nPrange	10547647	5273824	2636912	1318456
	MPI_Recv	nPrange	10547647	5273824	2636912	1318456
Cornerturn	VSIPL	n/a	34075638			
	MPI_Send	nPazimuth	21095295	10547647	5273824	2636912
	MPI_Recv	nPazimuth	21095295	10547647	5273824	2636912
Azimuth Processing	VSIPL	nPazimuth	2.26E+08	1.13E+08	5.7E+07	2.8E+07
	MPI_Send	nPazimuth	21095295	10547647	5273824	2636912
	MPI_Recv	nPazimuth	21095295	10547647	5273824	2636912
Display	VSIPL	n/a	n/a			
	MPI_Send	n/a	n/a			
Approximate Latency without Performance Modeling Simulation (ms)						
		nPrange	**1**	**2**	**4**	**8**
	nPazimuth					
		1	2746	2460	2317	2245
		2	2075	1788	1645	1574
		4	1739	1453	1310	1238
		8	1572	1285	1142	1070

Table 7-6. Ruling out architectures that do not meet scalability requirement (in black).

Approximate Latency without Performance Modeling Simulation (ms)						
		nPrange	**1**	**2**	**4**	**8**
	nPazimuth					
		1	2746	2460	2317	2245
		2	2075	1788	1645	1574
		4	1739	1453	1310	1238
		8	1572	1285	1142	1070

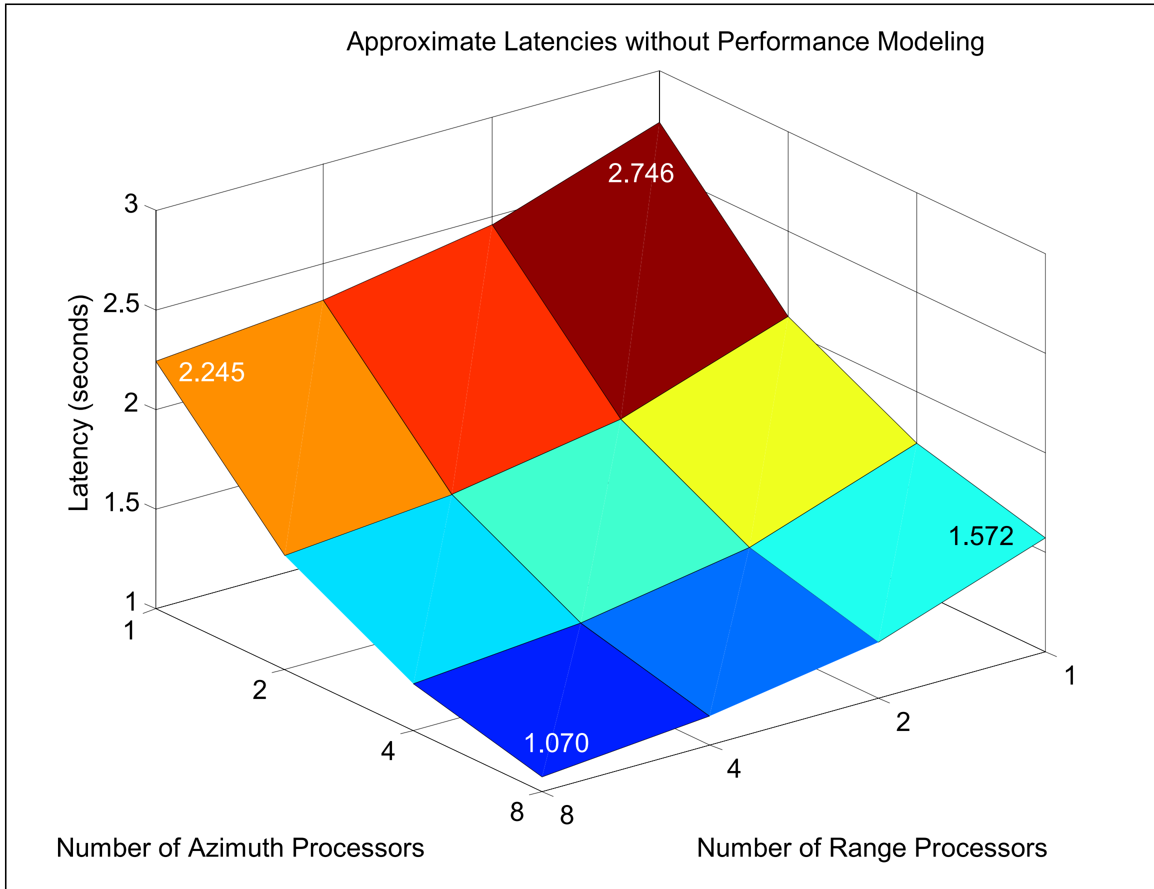

Figure 7-11. First order estimation of system latencies,
based on middleware token delays in the absence of performance modeling.

7.6.3.2 Latency When Accounting for IPC

We now turn to performance modeling to give us an accurate system simulation to see what our latencies are when we take the IPC into account. With the token delays computed in our executable workbook (cf. Table 7-5), we are able to build a model in eArchitect and simulate the behavioral performance (not the functional execution) of the candidate architectures left to consider. This required building a hardware model of a 4-board 16-PPC CE RACE COTS MP system, a software model of the architectures iterated through in Simulink, a mapping of the software to the hardware, and defining certain system values for the eArchitect framework. Details on starting up eArchitect are found in §B.3.1.

We built a hardware model of a four-board RACE system in eArchitect to model our 16 PPC CEs. This required building models of the 6U MCH motherboard, the PPC603 daughtercard (2 PPCs with 32MB/PPC), and an ILK4 backplane RACE interconnect. These designs were based on technical specifications available from Mercury. The top-level view of the hardware model created for and used by all of our performance models is shown in the Hardware Design editor window as shown in Figure 7-12. We go down into the different layers of this model in §B.3.2.

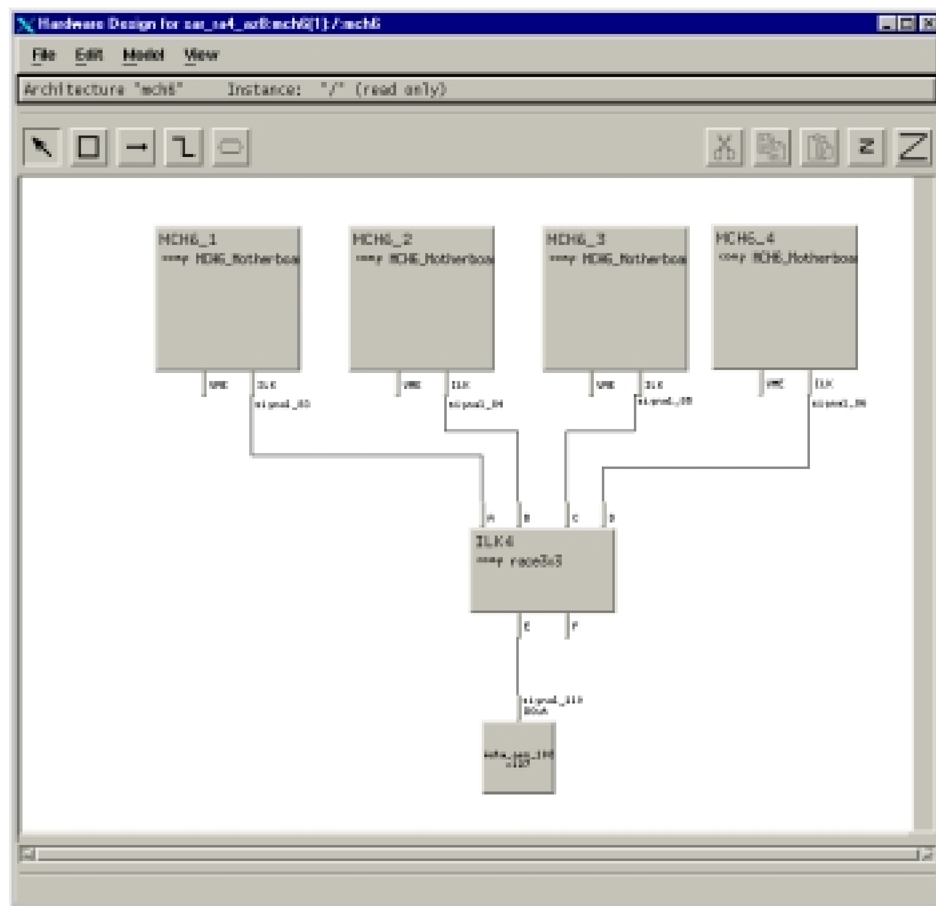

Figure 7-12. Hardware model used for all of our performance models.

Next we laid out the software model using eArchitect's Software Design editor as shown in Figure 7-13. For the sake of discussion, we use `sar_ra4_az8` (`nPrange`=4, `nPazimuth`=8), which is a model of the architecture shown in Figure 7-8. Details on the software editor's GUI are in §B.3.3.

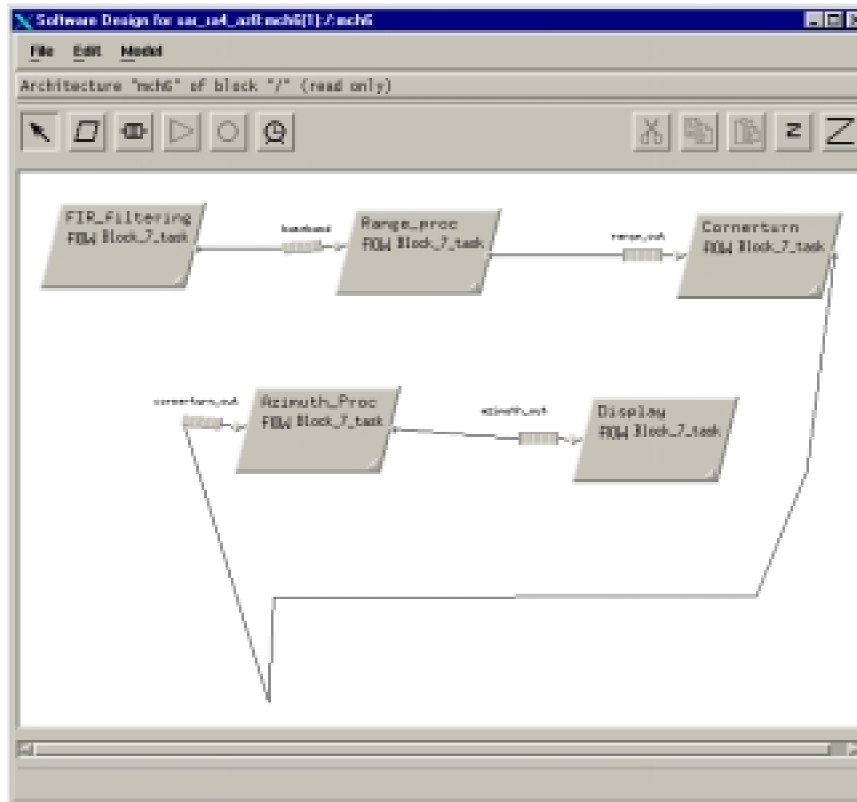

Figure 7-13. The software model of our performance model.

While eArchitect offers many ways of modeling the software, we streamlined our software modeling when we derived a single template for each of the processes. This is shown in Figure 7-14, where we show the range processor block model in particular, but its structure was duplicated for each of the blocks in the top-level Software Design editor (cf. Figure 7-13). We had some

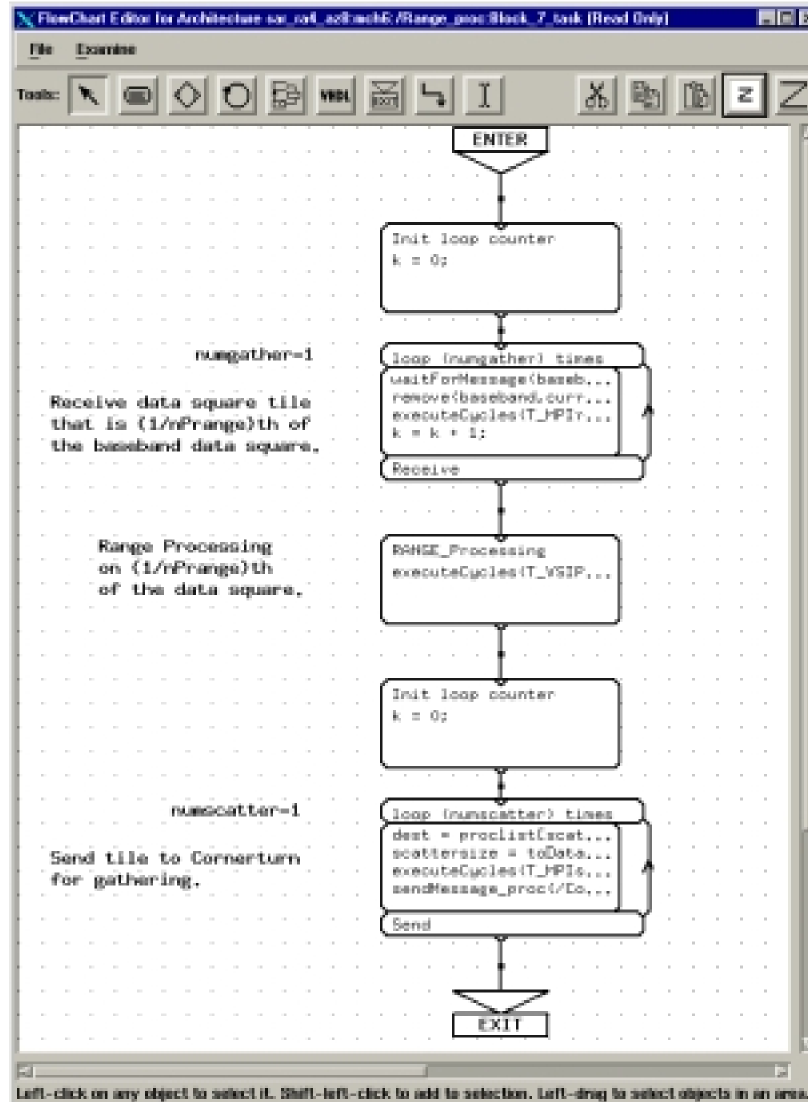

Figure 7-14. The range process with a template used in all of the blocks.

The basic flow for any process in a pipelined parallelized multiprocessor application is to receive data, process it, and then send results to the next processor or group of processors in the pipeline. In our performance model we account for the MPI overhead incurred in `MPI_Recv` and `MPI_Send`, and also for all the VSIPL functionality contained in the block, which are all computed as in §7.6.3 and summarized in Table 7-5. One difficulty regarding the display function is noted in §7.9.2.5.

While we were only interested in the non-shaded range-azimuth combinations as shown in Table 7-6, we simulated all combinations except 8 CEs each, since that

configuration is not possible given our constraints. Setting up the different software-to-hardware maps for the different nPrange-nPazimuth combinations is discussed in more detail in §7.9.2.4. We used the Simulate tool in the eArchitect framework to simulate our candidate architectures to determine the latency of processing a full 512-pulse frame of data. Further details on setting up eArchitect for simulation are in §B.3.6.

Simulations involve eArchitect generating VHDL code, compiling it, and running it through its VHDL simulator. This is all transparent to the eArchitect user, giving the user the high fidelity of VHDL modeling but without having to operate below our preferred processor level of granularity. The results of the `sim_ra4_az8_3s` simulation run are accessed through use of the Analysis Tools in the eArchitect framework. Details on its use are in §B.3.7. Repeating this process for all the other configurations (cf. §7.9.2.4) produces the results summarized in Table 7-7. Configurations that satisfied the 1.5-second latency without considering IPC but did not when considering IPC have been shaded. We also present a surface plot of these latencies in Figure 7-15.

Table 7-7. Latencies of SAR processor architectures accounting for IPC.
(Architectures that do not meet scalability requirement are shaded.)

Approximate Latency with Performance Modeling Simulation (ms)					
nPazimuth \ nPrange	1	2	4	8
1	3492	2862	2544	2388
2	2610	2054	1780	1642
4	2139	1653	1398	1268
8	1945	1451	1207	n/a

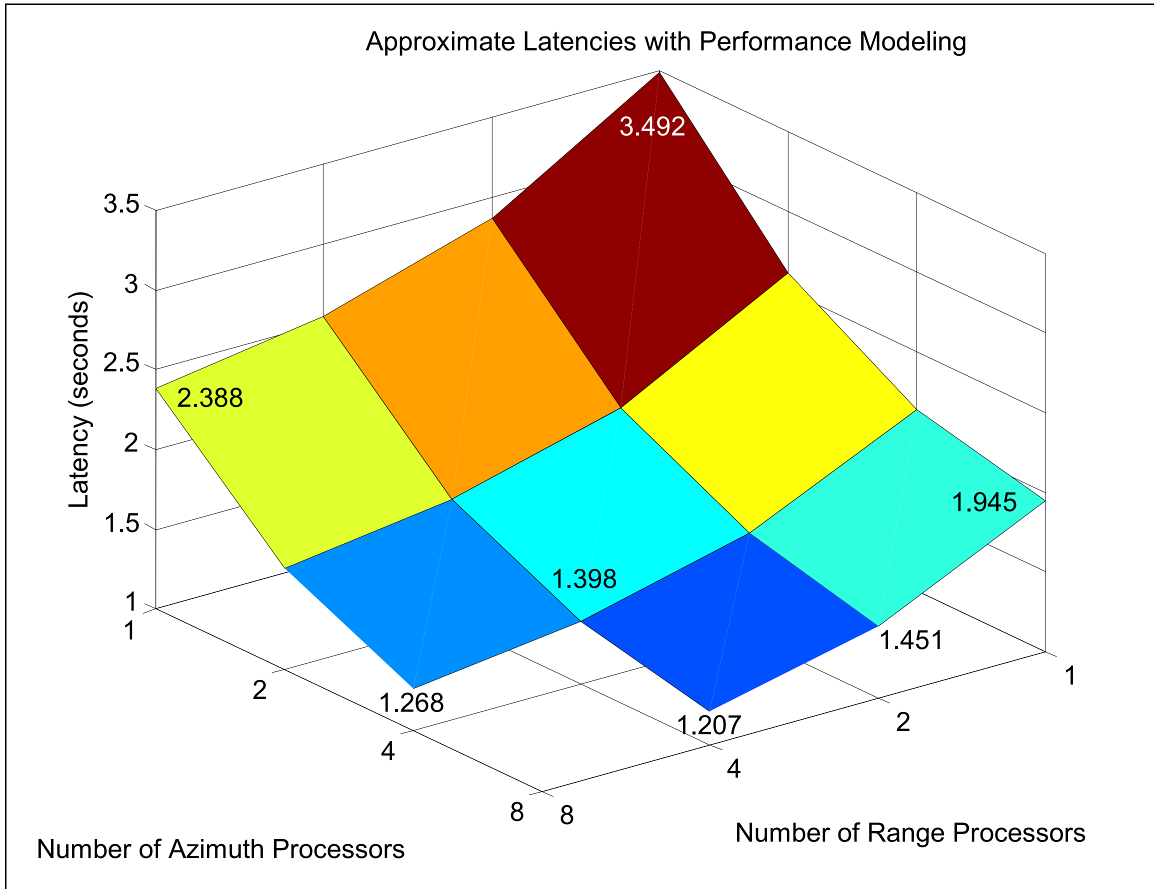

Figure 7-15. Latencies for SAR processing architectures based on performance modeling simulations.

☞ This step in the MAGIC SDM demonstrates model continuity in a powerful way. Our executable workbook contains a worksheet that computes token delays based on the number of processors used for parallelizing range and azimuth processing. These token delays are used in the eArchitect performance modeling simulations. The Simulink architecture reflecting the parallelization can be used to generate inner-loop computation and communication C code as well as test vectors that can all be used in the processor's implementation.

7.7 Make Design Decisions

With all of our performance modeling completed for our technology under consideration, we tabulate the candidate configurations in Table 7-8. We use those

configurations that satisfy the scalability requirement. We compute the board count of these potential designs using the following expression:

$$N_{boards} = \left\lceil \frac{2*(P_{range} + P_{azimuth} + 3\big|_{FIR,cornerturn,display})}{4} \right\rceil + 2\big|_{SBC,I/O}$$

The (•) expression accounts for the number of processors for range and azimuth processing, plus three for FIR processing, the cornerturn, and display processing. The coefficient of two is for the scalability requirement for a later time. We divide this processor count by four then round up to determine how many boards we need for the SAR processing. We also need two other boards, an SBC to act as the controller and a custom I/O board to bring the serial fiber raw image data in and convert it for use in our processing domain.

Table 7-8. Assessing our design options, optimizing on minimal board count, N_{boards}.

$T_{latency}$ (ms)	P_{range}	$P_{azimuth}$	P_{total}	$2*P_{total}$	N_{boards}
1398	4	4	11	22	8
1451	2	8	13	26	9
1207	4	8	15	30	10
1268	8	4	15	30	10

We define optimality here as the architecture with the minimum board count that satisfies the scalability requirement without violating any constraints. This means the following configuration of the Mercury technology components investigated is the optimum architecture:

- FORCE 8VT SBC
- Custom I/O board
- Six (6) Mercury MCH6 6U motherboards with eleven (11) PPC603 daughtercards

☞ This stage emphasizes value of model continuity by allowing us to make sure that the requirements model that we simulated in the design analysis satisfies non-performance constraints. In particular we see in Table 7-8, that all of our candidate architectures will not violate the 12-board limit imposed by the non-performance SWAP requirements.

According to our MAGIC SDM we would repeat this technology investigation with other potential technologies, using Myrinet and CSPI hardware or using SKYchannel and SKY Computers hardware. We could not get benchmarks from these vendors, so we must consider our design iterations concluded and the above configuration our design of choice. One footnote is that eArchitect has VHDL models for RACEway [171] and Myrinet in the hardware library, but not for SKYchannel, though such a model could be built.

7.8 Create Implementation Specification

The requirements specification phase of §7.3.2 and the design search phase of §7.6 has produced the following implementation specification items that can be consumed by a CASE implementation framework:

- A hardware configuration–cf. §7.7
- Generated middleware for computation and communication–cf. §7.6.1 and §7.6.2
- A software-to-hardware map–`nPrange`=4 and `nPazimuth`=4
- Test vectors for verification–cf. §7.3.2

With the creation of the implementation specification, the MAGIC SDM is complete and the detailed implementation work can commence.

7.9 Difficulties Encountered and Overcome

We document here significant difficulties encountered and how we overcame them. None of them invalidated the MAGIC SDM, but are provided as documentation of our efforts and also to provide a starting point for revising these frameworks to better support the MAGIC SDM.

7.9.1 Limitations of Simulink

While Simulink 3 and especially the DSP Blockset have evolved to support frame-based signal processor prototyping, we uncovered certain limitations in our use of them to build an executable specification. None invalidated the MAGIC SDM, but are

provided as documentation of the effort required to integrate COTS frameworks into our SDM.

7.9.1.1 Pulse Number Limitation

Unfortunately, we flushed out a limitation that exists even in the latest version of Simulink 3 (R11). It has some memory management problems that forced us to limit the number of pulses that Simulink can process. The value to which `npulses` is set depends on the workstation running Simulink. Our workstation had 384 MB and so we were limited to `npulses=16`.

While this is inconvenient and unfortunate, it does not limit our case study. This limitation affects our ability to generate test vectors; we will not be able to generate test point vectors such as `fir_out_sl` for use in verifying implementations. But since our case study concludes with an implementation specification, this limitation will not be an issue. This would also limit the code generation. Arguments of middleware function calls that include scaling an argument (e.g., dividing `npulses` by the number of processors performing range processing) would produce incorrect results since npulses is set to 16 and not 512 in our Simulink model. Since our code generation prototype is manual, this limitation will not affect code generation. For reasons of verification, we have added `npulses` as a variable in order to generate data sets in the MATLAB model for confirmation that our Simulink models are correct.

7.9.1.2 Column Major Artifices

We had to be careful in the range and azimuth processing. Simulink 3 is explicitly row major in how it views the data. This means that the columns are the samples of a pulse, but Simulink 3 treats the columns as rows since column values are contiguous in memory.

7.9.1.3 Simulink Run-Time Constraint and Artifice

If we were able to process the full frame, we would be able to display one-fourth of the entire image after completing a Simulink run. But because of the memory

management internal shortcoming of Simulink (cf. §7.9.1.1), we had to run the MATLAB model with the number of pulses (`npulses`) set to 16. This provides verification of the Simulink output to make sure the executable specification is correct.

7.9.2 Limitations of eArchitect

While created to support the design of boards that use the interconnection of COTS MP technologies, doing large multiprocessor simulations has not been how eArchitect has typically been used. Though it is designed to support just these types of simulations, we were the second users to use eArchitect to this end. Consequently, there were a few difficulties we had to overcome as we used eArchitect as a critical part of our design analysis framework. None invalidated the MAGIC SDM, but are provided as documentation of the effort required to integrate COTS frameworks into our SDM.

7.9.2.1 Reconciling Different Clock Frequencies and Cycles

At first this seemed to be a problem reconciling these two benchmarks. Our case study is meant to exercise the methodology and tool use for specification and design as well as validate its efficacy, not necessarily be a thorough product evaluation. However, we would like to be as accurate as we can for reasons outlined in §6.3.4. For now it is sufficient to note that these two seemingly disparate benchmarks can be used together by realizing that the newer PPC750 292MHz daughtercard intended for the newer MCJ motherboard (for the next generation "RACE++" backplane) can be used on the older MCH motherboard. The MPI benchmarks are reflective of the clock on the motherboard as much as on the architecture of the CE. We will therefore use the two benchmarks together with confidence.

The eArchitect tool requires clock cycles for its performance modeling, so the times were converted from time to clock cycles by multiplying the time in seconds by the 200MHz clock speed. We know that the VSIPL functions were actually run on a 292MHz CE, but this conversion is simply to provide the delays to the performance modeling framework. We modeled the hardware as 200MHz PPCs, so what is important is that the clocks accurately represent the time, so our clock counts will still be accurate.

7.9.2.2 Modeling VME Traffic

We did not model VME traffic because its presence will not affect the steady-state performance of the SAR processing. In most COTS MP systems, the VME bus is used for the downloading of the executable images to the MP CEs at setup, but usually just for power and ground at run-time.

7.9.2.3 Flowchart Reproduction Shortcomings

We reproduced the template in Figure 7-14 for each of the processing blocks, which is a *manual* task. This is something eArchitect would do well to change, to allow copy and paste in the Software Design editing window. Some processes have to perform scattering and/or gathering in addition to computation. Our template accounted for this in a semi-automated manner that greatly reduced the chance of human error; see §B.3.4 for the details.

7.9.2.4 Software-to-Hardware Mapping Shortcomings

After reproducing each of the processing blocks and editing them to represent their particular process, we turned our attention to mapping the software to the hardware. This is a key feature in any COTS MP CASE framework, and unfortunately a characteristic in which eArchitect is weak. The Mapping window for our architecture is shown in Figure 7-16 with software processes on the left and hardware processors on the right. The user has to point and click in this window and textually describe the mapping in the `dest` commands in each block's flowgraph.

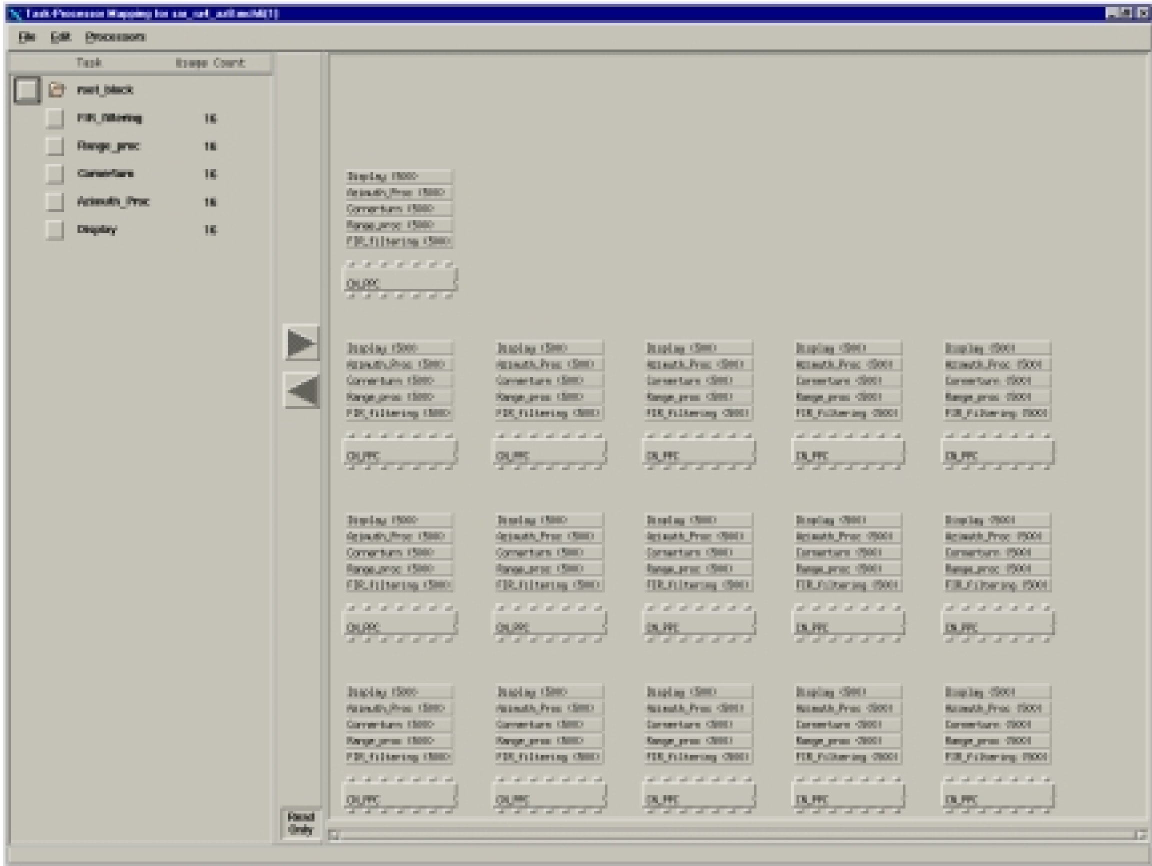

Figure 7-16. Mapping window for the SAR processing performance model.

We developed a workaround for this unwieldy and nonintuitive mapping shortcoming that allowed us to quickly reconfigure the mapping depending on the two architectural variables of `nPrange` and `nPazimuth`. We mapped all of the processes to all of the processors in the Mapping window, but set up `proclist` arrays for `dest` that are indexed by a `scatterlist` as shown in the scatter/send procedure code from range processing in Figure 7-17.

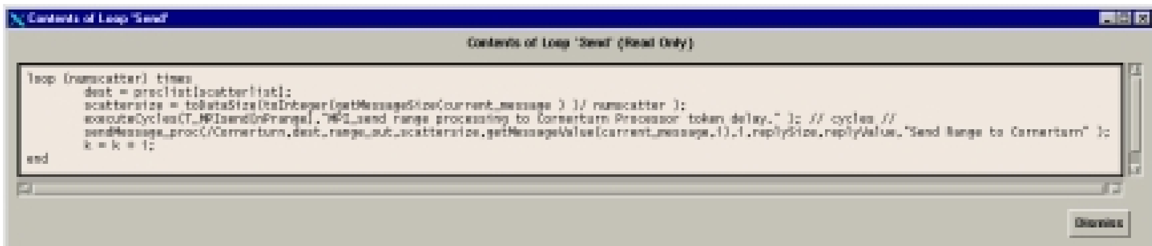

Figure 7-17. Scatter/send code that shows our flexible mapping.

To change the architecture, all we have to do is to open the local variables in our software block's flowgraph as shown in Figure 7-18 and change `nPrange` and `nPazimuth`, which control the scatter and gather loop iterations. They are also indexes into token delay arrays for the range and azimuth VSIPL tokens (`T_VSIPL`), and also for the MPI function tokens (`T_MPIrecv` and `T_MPIsend`) for all of the processes. The array `proclist` is the same in all of the software blocks. What is different is the `scatterlist` array that is used to index in it. We have assigned FIR processing to processor #0, the cornerturn to processor #1, and display to processor #2. The `scatterlist` of processors for range processing is {4, 5, …, 11}. The `scatterlist` of processors for azimuth processing is {15, 14, …, 8}. Since the range processes do not scatter or gather, the variables `numgather` and `numscatter` are both unity. Since the range outputs go to the cornerturn, the `scatterlist` is 1.

Figure 7-18. Local variables for range processing.

7.9.2.5 Token Delays for the Last `executeCycles`

The last process is the display process, which gathers the azimuth processing results. It does no processing other than iterate through multiple MPI_Recvs, since the signal processor (SAR image processor) forms the image, leaving post-processing to the data processor. Consequently, `T_VSIPL` is set to zero (cf. Figure B-15). This causes the VHDL generator and simulator to sometimes terminate immediately at that time.

Viewlogic is looking into this fault, and while it is not clean it does not seem to affect the efficacy of the simulations since that is the exact moment when the signal data is through being processed. We had some success setting `T_VSIPL` to one.

7.10 Conclusion

We have demonstrated the use of the MAGIC SDM with the use of a real-world domain-relevant benchmark, the RASSP SAR benchmark. We have also clearly shown that MAGIC accomplishes the three goals established at the beginning of this chapter, repeated here for convenience and with comment.

1) *The MAGIC SDM works as postulated, which means the rules can be followed and the tools work–especially in providing model continuity.*

We highlighted how model continuity was established in the integration of our tools that supported our rules (cf. §7.2, §7.3.1, §7.3.2, §7.6.3.2, and §7.7). Examples of model continuity included the passing of requirements model information back and forth to our design analysis performance modeling via our executable workbook, which also assured non-performance constraints were satisfied. Also, once a design was chosen, our requirements model was used to generate inner-loop computation and communication C code as well as test vectors that can all be used in the processor's implementation. The performance model provided hardware configuration and software-to-hardware mapping information to the implementation.

2) *The MAGIC SDM yields benchmarks of a full frame of data with run-times beyond the 3-second latency requirement, which is 20 times the longest VHDL simulation.*

We have run the VHDL-based simulations anywhere from 1.5 seconds to 4.0 seconds, well over the 150 ms achieved in other RASSP SAR VP-based VHDL simulations.

3) *The MAGIC SDM works in providing the framework to evaluate competitive technologies prior to implementation, which the CASE SDMs cannot do at all.*

We have demonstrated this by examining different architectures using real processor computation and communication deterministic benchmarks used to perform system performance simulations that include the nondeterministic interprocessor communication. We were able to determine which architectures would satisfy performance and non-

performance requirements, then decide on the optimum architecture for a given technology. This enabled us to specify the implementation, including its hardware configuration, software processes, software-to-hardware mapping, inner-loop implementation code, and test vectors for implementation verification.

Chapter 8

MAGIC Quantification and Conclusion

We have considered the shortcomings of specification and design methodology in our ADoI using COTS MP technologies in Chapter 1 and Chapter 2. We have considered the MoCs that should underlie a SDM in our ADoI (Chapter 3). Inspired by the SER SDM and how our design objects parallel those in the SER SDM domain (Chapter 5), we developed the rules and tools of a new specification and design methodology, the MAGIC SDM (Chapter 6). We have validated the MAGIC SDM and demonstrated its efficacy with a real-world benchmark (Chapter 7). In so doing we also demonstrated that the MAGIC SDM was clearly superior to both VHDL virtual prototyping and the deployable CASE SDMs that must commit to an implementation technology *before* performing design analysis (§7.1, §7.10).

In this concluding chapter we show how we have established model continuity in the MAGIC SDM. We also apply the quantified Sarkar unified basis to the MAGIC SDM and show that objective analytical assessment of the MAGIC SDM confirms the empirical evidence demonstrated in the SAR processing benchmark. We conclude by considering further research directions.

8.1 Model Continuity in the MAGIC SDM

We reproduce Figure 6-2 below in Figure 8-1 to show the model continuity absent in CASE SDMs. While they possess a narrow form of model continuity in that they can capture computational requirements and pass that model along to the implementation phase. However they do not allow the specifier to explore design alternatives prior to selecting a technology for implementation. CASE SDMs cannot execute a requirements specification without rapid prototyping the application software for a COTS MP

platform, which is essentially implementing the design *before* doing due diligence in regard to design analysis. So, while they offer a certain low level of model continuity, we want to emphasize how short they fall as illustrated in Figure 8-1.

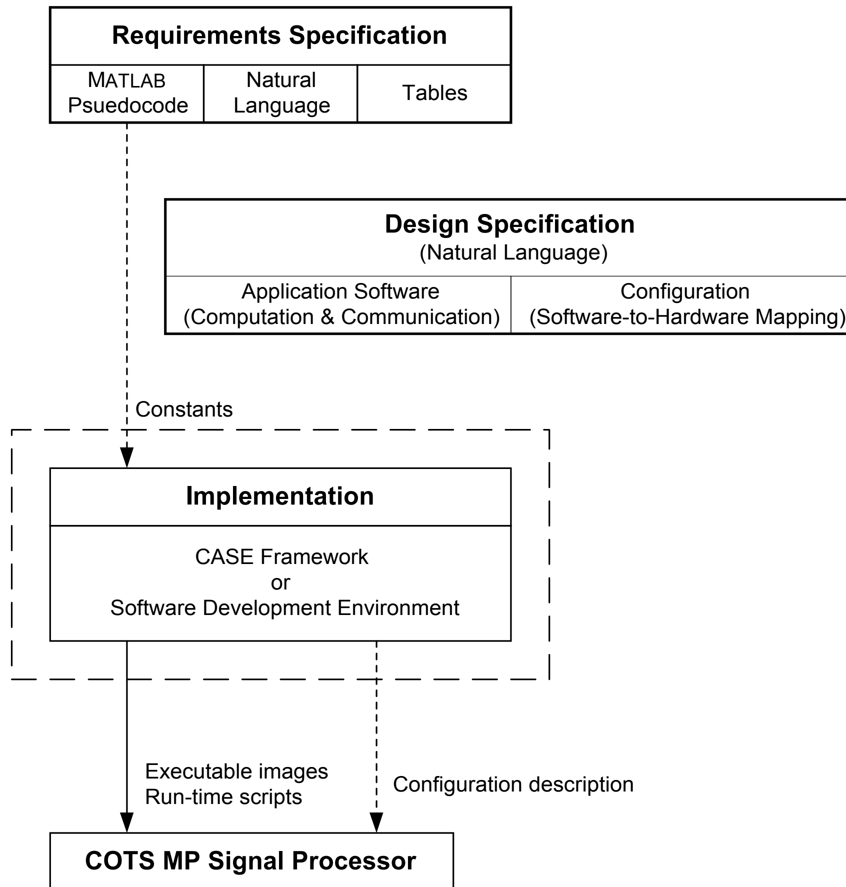

Figure 8-1. How model continuity is currently lacking in CASE SDMs (Figure 6-2).

Conversely, we want to emphasize the strong model continuity contained in the MAGIC SDM. We showed conceptually in Chapter 6 how the MAGIC SDM contained strong model continuity when we defined the MAGIC SDM. This was illustrated in Figure 6-3 and is reproduced in Figure 8-2 for ease of reference. We now show how we achieved model continuity in integrating the COTS frameworks that we chose (“tools”) to implement our MAGIC methodology (“rules”).

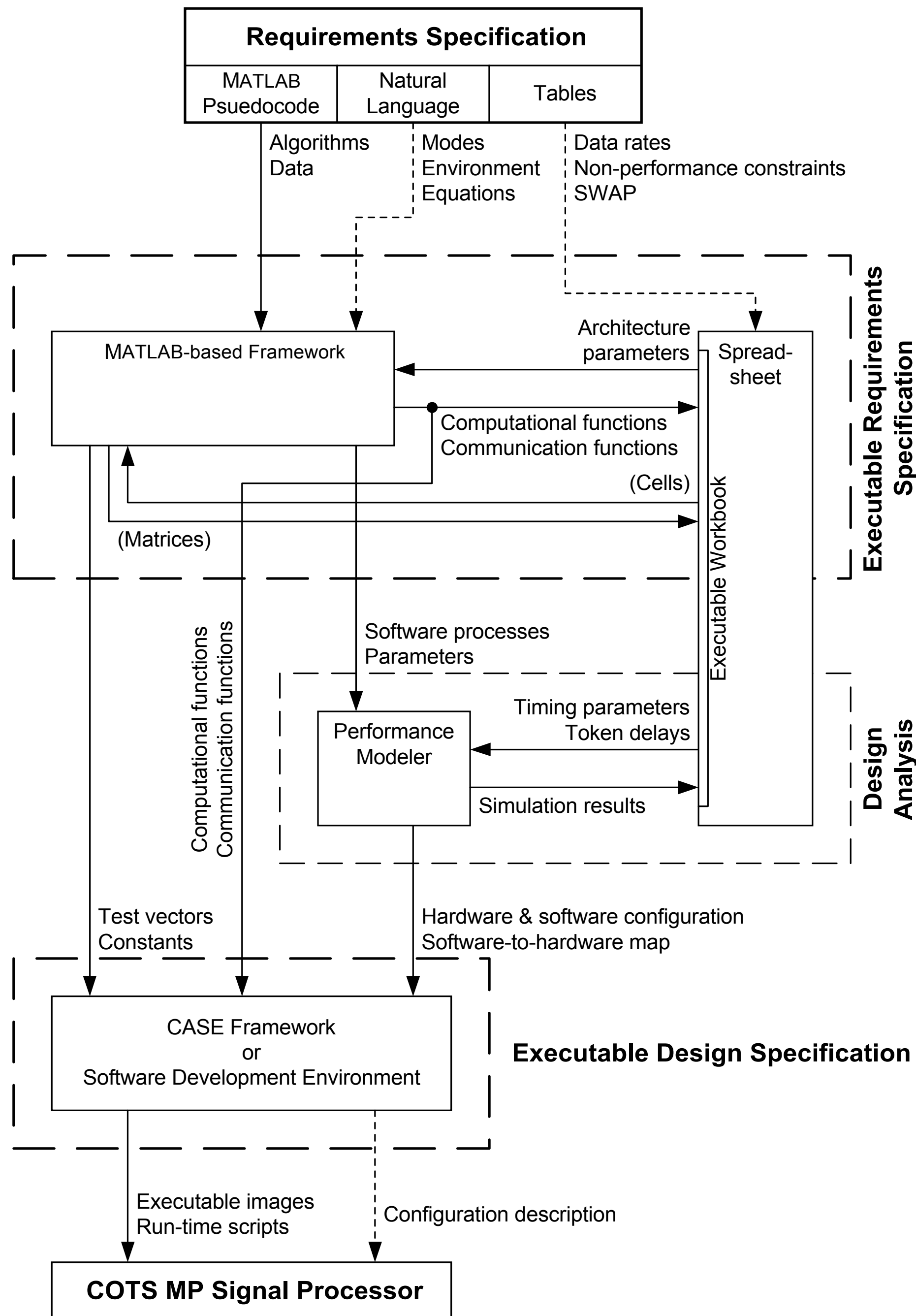

Figure 8-2. Establishing model continuity between an executable specification model and a design specification model (Figure 6-3).

The means of accomplishing model continuity using the frameworks we chose for the MAGIC SDM is illustrated in Figure 8-3. Boxes are items such as specifications, frameworks, or processes. Arrows are information, such as data, software, or

configuration information. Information flows from the natural language requirements specification into an executable requirements specification. Information iterates between the executable requirements specification and the design analysis framework. When design analysis is complete, the executable requirements specification and design analysis framework provide the inputs necessary for creating an executable design specification.

Solid boxes are documents or frameworks. Dashed boxes are aggregates of frameworks that contain executable specifications or the design analysis environment. Solid lines are automated channels, where system model information can be passed between frameworks without manual intervention. Dashed lines are semi-automated channels where some human intervention is required to move system model information between frameworks. Our contributions are highlighted in blue.

The executable workbook was fundamental in providing model continuity between specification and design. It was created using Excel with links created between worksheets that contained data (benchmarks, reliability statistics, form factor constraints, etc.) and models (benchmark conversions, process estimates, latency estimates, etc.). The data link to Simulink was manual; architectural parameters were computed in Excel and then implemented in Simulink by hand since Simulink does not support scaling for parallelization. VSIPL and MPI functions were "generated" using our code generation rules and entered into our executable workbook. Once in our workbook, we could compute token delays to be used in eArchitect for performance modeling. We would iterate this process for other candidate architectures.

We created channels of model continuity between specification and design with the implementation specification. When we decided upon an architecture, we could run Simulink and tap process outputs, dumping them into the MATLAB workspace where we could save them for testing the implementation. VSIPL and MPI code that we generated is available for use in the form of the inner-loop functions and parameter arguments. When design analysis is complete and we have made design decisions, our performance

model provides the hardware configuration, software process definition, and software-to-hardware mapping.

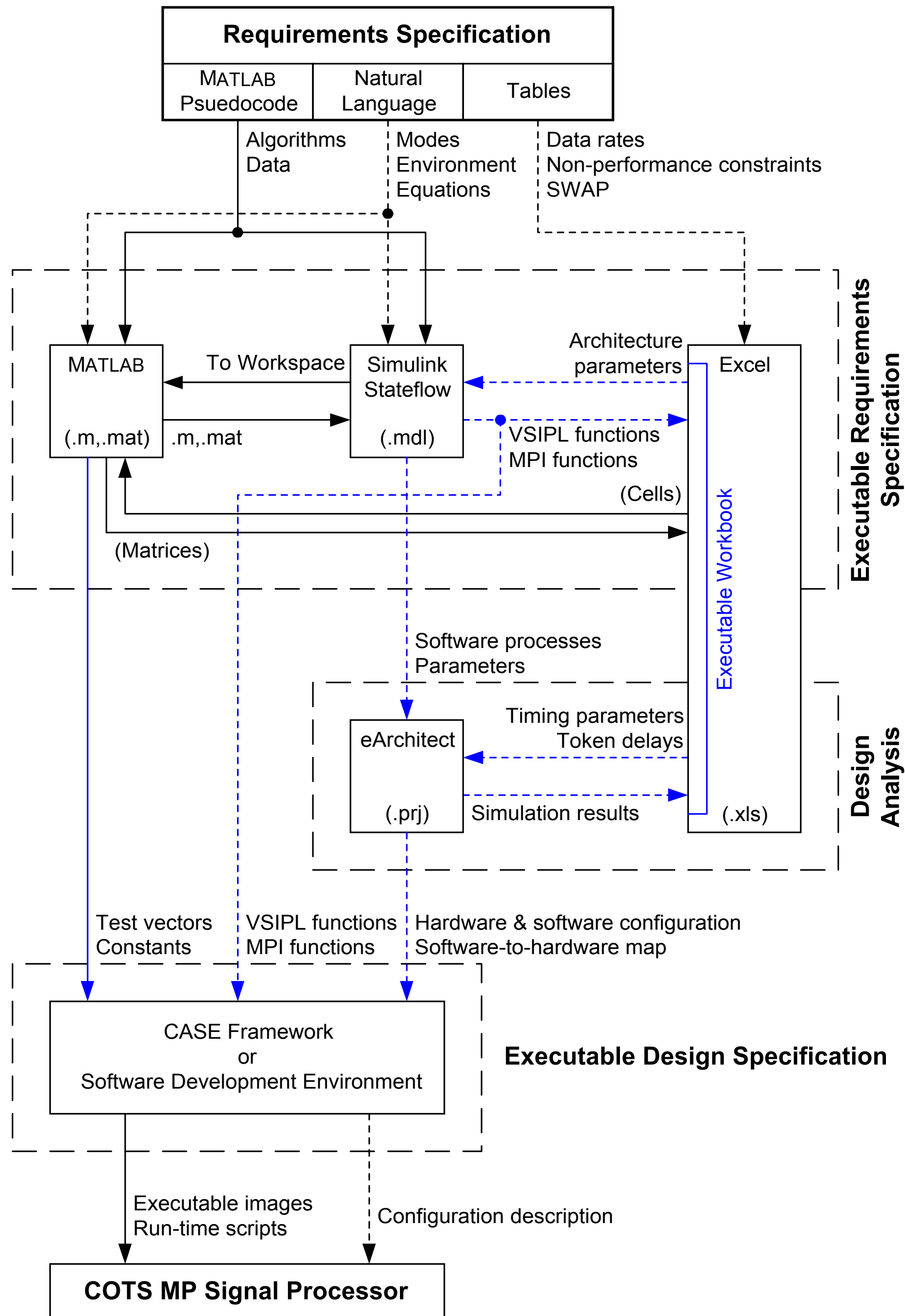

Figure 8-3. Establishing model continuity between an executable specification model and a design specification model.

8.2 *Sarkar Quantification of MAGIC SDM*

In §4.3 we quantified Sarkar's unified specification and design methodology basis and used that quantification to characterize the deployable CASE SMDs of interest as surveyed in §2.1.4.2. Having conceived, developed, and used the MAGIC SMD in a real-world class case study, it is worthwhile to apply the same quantification to our MAGIC SMD. This is reported in Table 8-2 through Table 8-6 and plotted in Figure 8-4 and Figure 8-5.

8.2.1 Language Attribute

The normalized language attribute of the MAGIC SDM did need not reach the ideal, but did provide improvement over all of the other SDMs in the range of roughly 31% to 143%. The summary is presented in Table 8-2 and the details are shown in Table 8-1. This was due especially to the MAGIC SDMs ability to support expressing the following sub-attributes:

- Modeling time–Simulink supports the explicit expression of time.
- Environmental characterization–The executable workbook supports the ability to model operational conditions, including SWAP.
- Nonfunctional characterization–The executable workbook can model the "illities" and also has a format that allows its models to be used by other frameworks, reliability frameworks in particular.

Table 8-1. Normalized language attribute values and the improvement with the MAGIC SDM.

Normalized	Language	% Improvement of MAGIC over other SDMs
Ideal	**1.00**	n/a
DSPW	0.59	30.8
RIPPEN	0.41	88.9
ACT	0.36	112.5
GEDAE	0.59	30.8
PW4R	0.32	142.9
MAGIC	**0.77**	n/a

Table 8-2. Language support sub-attributes spreadsheet including MAGIC SDM.

Sub-attribute	Sub-attribute components		Element Set	**Ideal**	DSPW	RIPPEN	ACT	GEDAE	PW4R	**MAGIC**
	Element(s)	Component								
System views	$V_i \in \{A, B, E\}$	$\lvert \mathbf{V} \rvert = A+B+E$	$\lvert \mathbf{V} \rvert \in \{0,1,2,3\}$	**3**	2	1	1	2	1	**2**
	A	Activity--data flow	{0,1}	1	1	1	1	1	1	1
	B	Behavior--control flow	{0,1}	1	1	0	0	1	0	1
	E	Entity--datatypes	{0,1}	1	0	0	0	0	0	0
Specification style	$S_i \in \{M, P\}$	$\lvert \mathbf{S} \rvert = M+P$	$\lvert \mathbf{S} \rvert \in \{0,1,2\}$	**2**	1	1	1	1	1	**1**
	M	Model--states, processes, or sets (easier to understand)	{0,1}	1	1	1	1	1	1	1
	P	Property--"black box" (less implementation-dependent)	{0,1}	1	0	0	0	0	0	0
Concurrency	$C_i \in \{C, S\}$	$\lvert \mathbf{C} \rvert = C+S$	$\lvert \mathbf{C} \rvert \in \{0,1,2\}$	**2**	2	2	1	2	2	**2**
	C	Communication (SMB's and/or MP paradigms)	{0,1}	1	1	1	0	1	1	1
	S	Synchronization (system control statements and/or comm channels)	{0,1}	1	1	1	1	1	1	1
Timing constraints	$T_i \in \{D, I\}$	$\lvert \mathbf{T} \rvert = D+I$	$\lvert \mathbf{T} \rvert \in \{0,1,2\}$	**2**	1	1	1	2	1	**2**
	D	Direct--inter-event delays, data rates, etc. (ideal--simple & flexible)	{0,2}	2	0	0	0	2	0	2
	I	Indirect--implied through lang constructs (e.g., Statecharts)	{0,1}	0	1	1	1	0	1	0
Modeling time	t	$\lvert \mathbf{t} \rvert = t$	$\lvert \mathbf{t} \rvert \in \{0,1\}$	**1**	1	0	0	1	0	**1**
		t=0. Does not support explicit expression of time.				0	0		0	
		t=1. Does support explicit expression of time.		1	1			1		1
Exeption handling	$H_i \in \{T, G\}$	$\lvert \mathbf{H} \rvert = T+G$	$\lvert \mathbf{H} \rvert \in \{0,1,2\}$	**2**	1	1	0	1	0	**1**
	T	Textual--e.g., language like Ada	{0,1}	1	0	0	0	0	0	0
	G	Graphical--e.g., visual environment like Statecharts	{0,1}	1	1	1	0	1	0	1
Environmental characterization	$E_i \in \{M, P\}$	$\lvert \mathbf{E} \rvert = M+P$	$\lvert \mathbf{E} \rvert \in \{0,1,2,3\}$	**3**	1	1	1	1	0	**2**
	M	Model--spec environment using same spec lang	{0,1}	1	1	1	1	1	0	0
	P	Property--set of hints about operational conditions (incl. SWAP)	{0,2}	2	0	0	0	0	0	2
Nonfunctional characterization	N	$\lvert \mathbf{N} \rvert = N$	$\lvert \mathbf{N} \rvert \in \{0,1,2\}$	**2**	0	0	0	0	0	**2**
		N=0=None.			0	0	0	0	0	
		N=L=1. Limited coverage (only one illity).								
		N=E=2. Extensive coverage (more than one).		2						2
Formal Analysis	A	$\lvert \mathbf{A} \rvert = A$	$\lvert \mathbf{A} \rvert \in \{0,1,2,3\}$	**3**	2	1	2	1	1	**2**
		A=0=None.								
		A=I_{lim}=1. Limited informal support--e.g., CFD's, DFD's, etc.				1		1	1	
		A=I_{sup}=2. Full informally support--i.e., lacks implementation independence, unambiguousness, & precision of process algebras.			2		2			2
		A=F=3. Formal support--e.g., process algebra like LOTOS.		3						
Model Executability	M	$\lvert \mathbf{M} \rvert = M$	$\lvert \mathbf{M} \rvert \in \{0,1,2\}$	**2**	2	1	1	2	1	**2**
		M=0=None.								
		M=L=1. Limited support--spec executable, but w/in scope of methodology.				1	1		1	
		M=S=2. Supported fully; supports direct execution of spec.		2	2			2		2
Total				**22**	13	9	8	13	7	**17**
Normalized Total				**1.00**	0.59	0.41	0.36	0.59	0.32	**0.77**

8.2.2 Complexity Control Attribute

The normalized complexity control attribute of the MAGIC SDM did need not reach the ideal, but did provide improvement over all of the other SDMs in the range of roughly 9% to 33%. The summary is presented in Table 8-3 and the details are shown in Table 8-5. This was due especially to the MAGIC SDMs ability to support expressing non-determinism, which is supported by both Simulink and eArchitect. Our rules and these two tools allow us to not necessarily specify some requirements or model them in the design phase until we absolutely need to. An example is the communication model. We have chosen to use MPI point-to-point send and receive primitive function calls to generate rather than take advantage of more sophisticated communication modes available using MPI. By using our point-to-point communication model we are able to correctly specify our requirements and design an architecture using COTS hardware known to satisfy performance and non-performance requirements *before* committing to a hardware platform. The actual communication model implemented may evolve from the one specified by our methodology in order to further minimize cost or for some other reason, but the implementation iteration can start with a specification that assures correctness.

Table 8-3. Normalized complexity control attribute values and the improvement with the MAGIC SDM.

Normalized	Complexity Control	% Improvement of MAGIC over other SDMs
Ideal	**1.00**	n/a
DSPW	0.85	9.1
RIPPEN	0.77	20.0
ACT	0.69	33.3
GEDAE	0.85	9.1
PW4R	0.69	33.3
MAGIC	**0.92**	n/a

8.2.3 Model Continuity Attribute

The model continuity attribute of the MAGIC SDM did reach the ideal, providing improvement over all of the other SDMs in the range of roughly 17% to 180%. The summary is presented in Table 8-4 and the details are shown in Table 8-5. This was due

especially to the MAGIC SDMs ability to support expressing the following sub-attributes:

- Interaction–Specification and design information can flow both ways horizontally and vertically between frameworks by means of middleware and the executable notebook.
- Implementation assistance–Performance modeling and middleware layered on optimized vendor libraries and the use of code generation that can feed directly into a CASE code generation framework provide a high degree of completeness and optimality.
- Implementation independence–The use of middleware and performance modeling allows the MAGIC SDM identify the best architecture and technology *before* committing to a vendor and platform.

Table 8-4. Normalized model continuity attribute values and the improvement with the MAGIC SDM.

Normalized	Model Continuity	% Improvement of MAGIC over other SDMs
Ideal	**1.00**	n/a
DSPW	0.71	40.0
RIPPEN	0.57	75.0
ACT	0.57	75.0
GEDAE	0.86	16.7
PW4R	0.36	180.0
MAGIC	**1.00**	n/a

Table 8-5. Complexity control sub-attributes spreadsheet including MAGIC SDM.

Sub-attribute	Sub-attribute components		Element Set	**Ideal**	DSPW	RIPPEN	ACT	GEDAE	PW4R	**MAGIC**
	Element(s)	Component								
Hierarchy	*H*	|**H**| = *H*	|**H**|∈{0,1,2}	**2**	2	2	1	2	1	**2**
		H =0=None.								
		H =L=1. Limited--cannot readily decompose spec.					1		1	
		H =S=2. Supported--supports multiple levels of spec decomposition.		2	2	2		2		2
Orthogonality	*O*	|**O**| = *O*	|**O**|∈{0,1,2}	**2**	2	2	2	2	2	**2**
		O =0=None.								
		O =L=1. Limited--cannot readily describe two behaviors independently of one another.								
		O =S=2. Supported--Can readily describe two behaviors independently of one another.		2	2	2	2	2	2	2
Representation	$R_i \in \{T, G\}$	|**R**| = *T*+*G*	|**R**|∈{0,1,2}	**2**	2	1	1	2	2	**2**
	T	Textual--Non-visual, e.g., ACL, Matlab, etc.	{0,1}	1	1	0	0	1	1	1
	G	Graphical--Visual formalism, e.g., GEDAE, ACT, RIPPEN, etc.	{0,1}	1	1	1	1	1	1	1
Non-determinism	*D*	|**D**| = *D*	|**D**|∈{0,1,2}	**2**	1	1	2	1	1	**2**
		D =0=None.								
		D =L=1. Limited--cannot incorporate non-determinism into spec in a controlled manner.			1	1		1	1	
		D =S=2. Supported--Can incorporate non-determinism into specification in a controlled manner, also allowing detection & resolution of non-determinism during specification.		**2**			2			2
Perfect-synchrony assumption	$S_i \in$ {A,S}	|**S**| =A+S	|**S**|∈{0,1,2}	**2**	1	1	1	1	0	**1**
	A	Asynchronous--Time advances implicitly as in concurrent languages such as Ada, SREM, etc.	{0,1}	1	0	0	0	0	0	0
	S	Synchronous--Time advances *iff* explicitly spec'd, as in Statecharts.	{0,1}	1	1	1	1	1	0	1
Developmental guidance	$G_i \in$ {*B*,T,M}	|**G**| = *B*+T+M	|**G**|∈{0,1,2,3}	**3**	3	3	2	3	3	**3**
	B	Bottom-up--Identify primitives, then combine upwards into subsystems which combine eventually into the system.	{0,1}	1	1	1	1	1	1	1
	T	Top-down--Decompose the spec into smaller and more-detailed components downward into components, which are then integrated upward into subsystems, etc.	{0,1}	1	1	1	1	1	1	1
	M	Middle-out--Combination of *B* and *T*, leveraging reuse.	{0,1}	1	1	1	0	1	1	1
Total				**13**	11	10	9	11	9	**12**
Normalized Total				**1.00**	0.85	0.77	0.69	0.85	0.69	**0.92**

Table 8-6. Model continuity sub-attributes spreadsheet including MAGIC SDM.

Sub-attribute	**Sub-attribute components**		**Element Set**	**Ideal**	DSPW	RIPPEN	ACT	GEDAE	PW4R	**MAGIC**
	Element(s)	Component								
Conformance	$C_i \in C_H \cup C_V$	$\lvert \mathbf{C} \rvert = \lvert \mathbf{C}_H \rvert + \lvert \mathbf{C}_V \rvert$	$\lvert \mathbf{C} \rvert \in \{0,1,2,3,4\}$	**4**	4	3	3	4	1	**4**
Horizontal	$C_H \in \{A,S\}$	$\lvert \mathbf{C}_H \rvert = A+S$	$\lvert \mathbf{C}_H \rvert \in \{0,1,2\}$	**2**	2	1	1	2	0	**2**
		H=Horizontal--Different modeling domains; e.g., between the functional-level and behavioral-level models.								
	A	A=Analysis.	{0,1}	1	1	0	0	1	0	1
	S	S=Simulation.	{0,1}	1	1	1	1	1	0	1
Vertical	$C_V \in \{A,S\}$	$\lvert \mathbf{C}_V \rvert = A+S$	$\lvert \mathbf{C}_V \rvert \in \{0,1,2\}$	**2**	2	2	2	2	1	**2**
		V=Vertical--Different levels of abstraction; e.g., between algorithmic-level and hardware-mapping-level models.								
	A	A=Analysis.	{0,1}	1	1	1	1	1	0	1
	S	S=Simulation.	{0,1}	1	1	1	1	1	1	1
Interaction	$I_i \in \{I_H, I_V\}$	$\lvert \mathbf{I} \rvert = I_H + I_V$	$\lvert \mathbf{I} \rvert \in \{0,1,2,3,4\}$	**4**	4	2	2	3	1	**4**
Horizontal	I_H	I_H=0=None.				0	0		0	
		I_H=U=1. Unidirectional--Information only flows in one direction between models.						1		
		I_H=B=2. Bidirectional--Information flows in both directions between models.		**2**	2					2
Vertical	I_V	I_V=0=None.								
		I_V=U=1. Unidirectional--Information only flows in one direction between models.							1	
		I_V=B=2. Bidirectional--Information flows in both directions between models.		**2**	2	2	2	2		2
Complexity	$P_i \in \{H,V\}$	$\lvert \mathbf{P} \rvert$ =H+V	$\lvert \mathbf{P} \rvert \in \{0,1,2\}$	**2**	2	1	1	2	1	**2**
	H	H=Horizontal--Different modeling domains; e.g., between the functional-level and behavioral-level models.	{0,1}	1	1	0	0	1	0	1
	V	V=Vertical--Different levels of abstraction; e.g., between algorithmic-level and hardware-mapping-level models.	{0,1}	1	1	1	1	1	1	1
Implementation Assistance	A	$\lvert \mathbf{A} \rvert = A$	$\lvert \mathbf{A} \rvert \in \{0,1,2\}$	**2**	0	1	1	2	1	**2**
		A=0=None.			0					
		A=L=1. Limited--Inefficient synthesis or implementation is strictly based on specification; both lead to suboptimal implementations.				1	1		1	
		A=S=2. Supported--Able to produce complete implementation with some degree of optimality.		2				2		2
Implementation Independence	N	$\lvert \mathbf{N} \rvert = N$	$\lvert \mathbf{N} \rvert \in \{0,1,2\}$	**2**	0	1	1	1	1	**2**
		N=0=None.			0					
		N=L=1. Limited--Spec has some measure of implementation bias, i.e., specs some externally unobservable properties.				1	1	1	1	
		A=S=2. Supported--Spec is w/o bias: a) Specifier can focus strictly on behavior (not implementation) of system. b) Avoids placing unnecessary restrictions on designer freedoms.		2						2
Total				**14**	10	8	8	12	5	**14**
Normalized Total				**1.00**	0.71	0.57	0.57	0.86	0.36	**1.00**

8.2.4 Attribute Aggregate Values

The total raw values of the three SDM attributes including the MAGIC SDM are tabulated below in Table 8-7. Dividing them by the ideal value for each of the attributes normalizes the attributes' total raw values. These normalized values are tabulated in Table 8-8 and plotted one at a time in the bar graph of Figure 8-4. These values are also plotted in 3-tuple space (Figure 8-5) to illustrate how they compare to one another, against the ideal SDM, and against our MAGIC SDM.

Table 8-7. Raw attributes for CASE SDMs vis à vis Ideal and MAGIC SDMs.

Raw	Attributes		
	Language	Complexity Control	Model Continuity
Ideal	**22**	**13**	**14**
DSPW	13	11	10
RIPPEN	9	10	8
ACT	8	9	8
GEDAE	13	11	12
PW4R	7	9	5
MAGIC	**17**	**12**	**14**

Table 8-8. Normalized attributes for CASE SDMs vis à vis Ideal and MAGIC SDMs.

Normalized	Attributes		
	Language	Complexity Control	Model Continuity
Ideal	**1.00**	**1.00**	**1.00**
DSPW	0.59	0.85	0.71
RIPPEN	0.41	0.77	0.57
ACT	0.36	0.69	0.57
GEDAE	0.59	0.85	0.86
PW4R	0.32	0.69	0.36
MAGIC	**0.77**	**0.92**	**1.00**

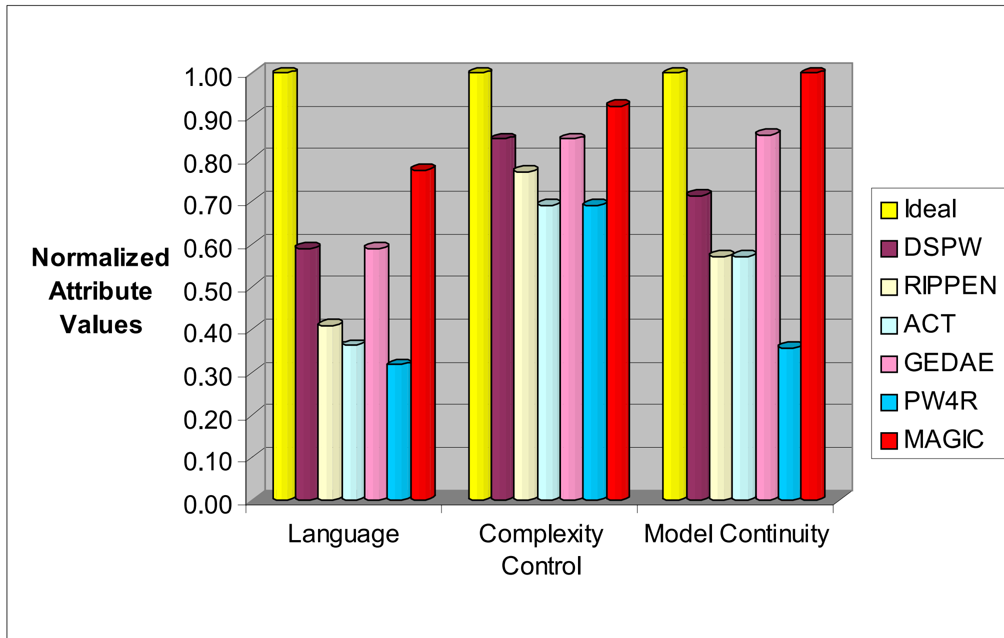

Figure 8-4. Plot of normalized attribute values for the CASE SDMs vis à vis the Ideal and MAGIC SDMs from Table 8-8.

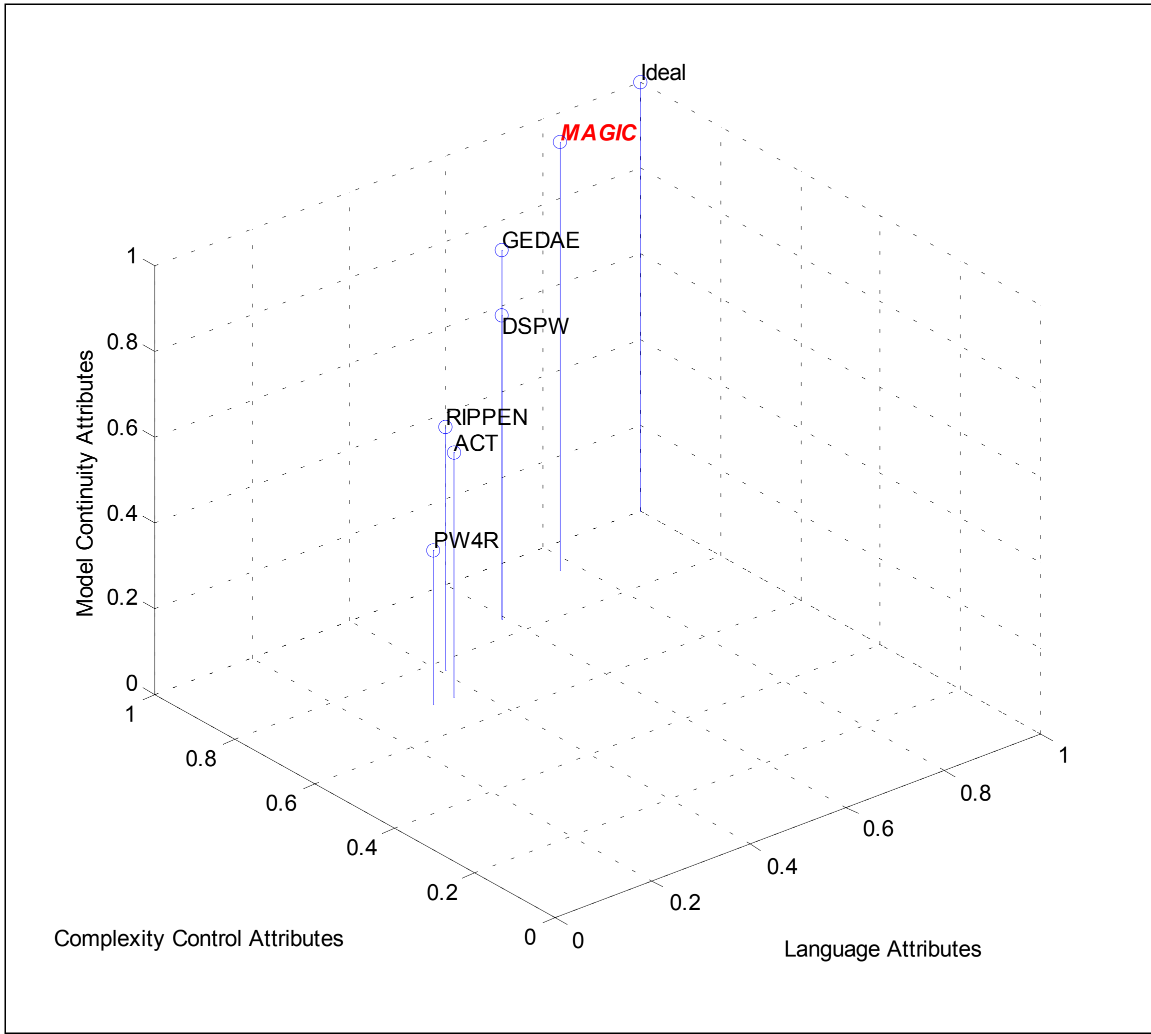

Figure 8-5. Plot of SDMs in 3-D attribute-space,
which shows MAGIC's improvement over CASE SMDs moving towards Ideal SDM.

We can get an overall comparison of our SDMs if we take the norms of the SDM 3-tuples using the following expression:

$$Norm_{3-tuple} = \frac{\sqrt{(\text{Language})^2 + (\text{Complexity Control})^2 + (\text{Model Continuity})^2}}{\sqrt{3}}$$

The results of these computations are given in Table 8-9, where we see that the MAGIC SDM has an overall improvement over the other SDMs of approximately 17% to 86%.

Table 8-9. Normalized model continuity attribute values and the improvement with the MAGIC SDM.

	Norm	% Improvement of MAGIC over other SDMs	Improvement of MAGIC over other SDMs
Ideal	1.00	n/a	n/a
DSPW	0.72	24.7	1.2 ×
RIPPEN	0.60	50.2	1.5 ×
ACT	0.56	61.6	1.6 ×
GEDAE	0.77	16.7	1.2 ×
PW4R	0.49	86.0	1.9 ×
MAGIC	0.90	0.0	1.0 ×

8.3 Summary

It has been shown that the MAGIC specification and design methodology is an improvement over existing CASE SDMs in use in our ADoI. MAGIC shows marked improvement in each SDM attribute and especially in model continuity. It does so without requiring the premature commitment to a hardware and software target. This is accomplished by using existing COTS frameworks adapted and extended to our ADoI and by prototyping the code generation of standards-based middleware for computation (VSIPL) and communication (MPI).

We have demonstrated how to use our MAGIC SDM in a case study using a SAR processor benchmark to perform the specification and design of a real-time embedded radar signal processor using COTS MP hardware and software. To quantify existing CASE SDMs and quantifiably demonstrate improvement in our MAGIC SDM, we had to take a well-respected but qualitative unified basis by Sarkar used for comparing specification and design methodologies for reactive systems and extract math models for each sub-attribute. We now have a quantified basis useful for visually comparing different SDMs and for identifying shortcomings within an attribute, allowing us to further improve SDMs.

CASE SDMs can be effective and useful frameworks for rapid system prototyping, especially in code generation and configuration model management. Unfortunately their SDM usefulness is greatly hindered because the hardware must be decided upon and acquired *before* specification and design can be done. We note that the MAGIC SDM coupled with a good CASE SDM is a powerful means of accomplishing

the specification, design, and implementation of COTS MP-based systems–with a strong thread of model continuity throughout all three phases.

8.4 Directions for Further Research

Further research is called for, both in applied terms and basic terms. Our case study has shown some fundamental work needs to be done with COTS frameworks to support our MAGIC SDM. There are also some more fundamental methodological issues that arise as a result of this work. These are covered in the following sections.

8.4.1 Applied Framework Research

Our rules are in good shape, but our tools need revision and extension to better support the MAGIC SDM. Executable requirements specification modeling and design analysis frameworks need to specifically support multiprocessing paradigms such as scaling. In practical terms, applied research includes working with The MathWorks to improve and extend DSPW:

- Improve Simulink memory management internals to support large MP-type models
- Add scalability to Simulink so parallelized processes don't have to be manually instantiated
- Translate MAGIC into RTW to generate VSIPL and MPI middleware code

Similar work needs to be done in regard to Viewlogic's eArchitect:

- Streamline mapping facility so user does not have to be clever
- Add scalability to replicate arguments and processes
- Characterize how much of eArchitect is needed for MAGIC SDM and possibly create a performance modeling tool to integrate into DSPW

8.4.2 Basic Methodological Research

More basic research issues remain as well in assessing how to characterize applications to know what level of detail is necessary for design exploration. This is important in any specification and design endeavor to perform only the level of analysis

and exploration necessary to get to the implementation phase of the product and on to market. Tools like eArchitect provide high-fidelity VHDL-based simulations, but we suspect that the level of fidelity is overkill and that better methods can be found to expedite the specification and design of COTS MP-based systems.

Another area of investigation is the characterization of a configuration-level model that could be used by all three phases of the design process, rather than our somewhat loose connectivity. This could lead to a standardization of configuration complementary to that of computation and communication.

A fundamental issue is to determine how applicable the MAGIC SDM is beyond the domain of real-time streaming data with data transformation processing applications implemented with embedded COTS multiprocessing technology. By constraining our focus to our application domain, we were therefore able to identify the frameworks and middleware that would be viable for integration into the MAGIC SDM. While we have constrained our focus to this domain, it seems promising to adapt the MAGIC SDM to other application and technology domains.

Frameworks exist for other application and technology domains, usually referred to as "EDA" (electronic design automation). Middleware is by no means restricted solely to our technology domain, and could similarly serve as the model continuity medium in other technology domains. Using the right tools at the right time should be applicable to other domains as well. We believe this would be worthwhile to pursue, especially in the system-on-a-chip domain.

Appendix A

Details of VSIPL and MPI Middleware

A.1 VSIPL Details

An API supporting portability for COTS users of real-time embedded multicomputers has been produced by a national forum of government, academia, and industry participants, known as the Vector Scalar Image Processing Library (VSIPL). VSIPL provides a type of computational middleware, which also supports interoperability with interprocessor communication (IPC) middleware such as MPI and MPI/RT. The VSIPL Forum has produced the API, a prototype reference library, and a test suite to verify API compliance. Commercial implementations are just now becoming available (Fall of 1999). Earnest consideration by various defense programs is underway and early adoption has begun.

A.1.1 VSIPL Fundamentals

VSIPL fundamentals will be introduced at a high level in this section before going into the details of the individual elements. The functionality offered by the API is discussed as well as subsets of the API known as “profiles.” How the functions operate on the “object-based” VSIPL data elements is then discussed.

A.1.1.1 Functionality

The VSIPL API standard provides hundreds of functions to the application software developer to support computation on scalars, vectors, or dense rectangular arrays. The v1.0 API specification document lays out the categories of the functionality in the following way:

- Support functions

- Object creation and interaction
    - Memory management
- Basic scalar operations
- Random number generation
- Basic vector and elementwise operations
- Signal processing
    - FFT operations
    - Correlation and convolution
    - Windowing
    - Filtering
- Linear algebra
    - Basic matrix and vector operations
    - Linear system solvers

The absence of image processing functions beyond matrix and 2D functions is acknowledged by the Forum and is being addressed in the Journal of Development (JoD). It should be noted that both Khoral Research and Colorado State[8] have done some early formative VSIPL image processing development.

A.1.1.2 Profiles

While there are hundreds of functions in the VSIPL API standard, not all functions are available in all implementations. The contents of a given implementation are defined in a *profile*. Initially the Forum has defined two profiles, "Core" and "Core Lite." The Core profile is the "80/20" subset of v1.0 API that is believed to contain the "20%" of the API that will be needed in "80%" of the applications targeted at COTS embedded processors. The Core Lite profile is the 80/20 subset of the Core profile, and is a size manageable to the participating vendors to provide initial VSIPL implementations. It is believed that the market will determine subsequent profiles.

[8] CSU's Cameron project (http://www.cs.colostate.edu/cameron/applications.html) evaluated early VSIPL image processing functionality.

A.1.1.3 Objects

The key difference between the VSIPL API standard and existing libraries is the encapsulation of memory management through an "object-based" (vis à vis object-oriented) design. In VSIPL a *block* can be thought of using the familiar model of a contiguous area of memory for data storage. A *block object* associated with the block contains the information that the VSIPL implementation needs to access the memory. A *view object* is the accessor function, or handle, VSIPL provides the user to access the block object. The view object contains information about the block object and how to view the data, including familiar parameters such as offset, stride, and length

Blocks and views are "opaque" objects, which means that they can only be created, accessed, and destroyed using VSIPL functions. The data elements associated with the block and view objects are private to hide the details of the memory management. This frees the VSIPL application software developer (the "user") from having to get buried in the details of the processor architecture and perhaps unwittingly write non-portable code. Similarly, it enables the VSIPL implementor the opportunity to differentiate his or her implementation by its performance.

Data arrays in VSIPL can then lie in one of two logical spaces, either in *user data space* or *VSIPL data space*. VSIPL functions may only operate on data in VSIPL data space, and the user's only access to that data is with VSIPL object functions. The user may access data in the user data space (such as scalars or C arrays, structures, etc.), but VSIPL may not unless that user data is brought into the VSIPL data space. Note that data can go both ways between these two data spaces, and such a move may or may not involve a data copy and the performance penalty associated with such a copy. This is discussed further in §A.1.2 and illustrated in Figure A-1.

A.1.2 VSIPL Concepts

In this section we present a discussion of VSIPL library design and how VSIPL manages memory for efficiency. We also present the application flow using VSIPL.

A.1.2.1 Library Design Principles

The VSIPL API standard supports high performance numerical computation on dense rectangular arrays. The API incorporates the following well-established characteristics of existing sound scientific and embedded algorithm libraries:

- Elements are stored in one dimensional data arrays, which appear to the application software developer as a single contiguous block of memory.
- Data arrays can be viewed as either real or complex vectors, matrices, or tensors[9].
- An offset and one or more strides are used to access subviews.
- All VSIPL operations on a data array are performed indirectly through view objects, each of which specify a particular view of a data array with a particular offset, stride(s), and length(s).
- All operators specify destinations as well as source operands. The application software developer cannot combine operators in a single statement to evaluate expressions, but must provide temporary variables for intermediate results.

For the sake of efficiency, operators are restricted to views of a data array that can be specified by an offset, stride(s), and length(s). More arbitrary views can be converted into these simple views by functions like gather, and then back again by functions like scatter. VSIPL does not currently support triangular or sparse matrices very well, though future revisions to the API may accommodate them.

To amplify §A.1.1.3, the main difference between the VSIPL API standard and existing algorithm libraries for embedded processors is the clean encapsulation of the above characteristics through an object-based design. All view attributes are encapsulated in opaque objects, which can only be created, accessed, and destroyed using VSIPL functions.

A.1.2.2 Data Space and Access: Blocks & Views

Concretely, a data array is simply an area of memory where data is stored. More abstractly in VSIPL, data arrays exist in one of two logical data spaces, either in *user data space* or in *VSIPL data space*. The application programmer may operate directly on

[9] A VSIPL “tensor” is simply a data type with a dimension greater than 2, usually 3.

data in user data space (e.g., C arrays), but not in VSIPL data space. His or her only access to data in VSIPL data space is through VSIPL functions.

A data array allocated by the application using any non-VSIPL method is considered to be in user data space and is a *user data array*. The application has a pointer to the user data array and should have implicit knowledge of its type and size, which allows access to the user data array by using pointers directly.

A data array allocated by a VSIPL function call is in VSIPL data space and is a *VSIPL data array*. The user has no correct or reliable way to use a pointer to access data in a VSIPL data array; data may only be accessed using VSIPL function calls. The way for the user to allocate data arrays in VSIPL space is to use a VSIPL memory object known as a *block*. The data array associated with a block is a contiguous series of elements of a given datatype. VSIPL has one block type for each of the VSIPL datatypes.

There are two kinds of blocks depending on the creator of the block, *user blocks* and *VSIPL blocks*. The user block is associated with a user data array and a VSIPL block is associated with a VSIPL data array. The data array that the block references is referred to as being "bound" to the block. The user must provide VSIPL with a pointer to the associated data to bind the user block. Blocks can also be created without any data and then later associated with data in user space, which is known as "binding." A block without data bound to it may not be used since there is no data on which to operate.

A block that is bound to data exists in one of two states, either *admitted* or *released*. Admitted blocks exist in the logical VSIPL data space and released blocks exist in the logical user data space. Moving blocks from one logical data space to the other logical data space is known as admission (user→VSIPL) or release (VSIPL→user).

Data in an admitted block is "owned" by VSIPL. VSIPL functions operate on this data with the presumption that only VSIPL functions will operate on the data. VSIPL blocks are always in the admitted state. User blocks are in the admitted state only if they have been explicitly admitted. If a user block is admitted the only deterministic and reliable way to operate on its data is with VSIPL functions. An attempt to directly

manipulate user data bound to an admitted block (e.g., using pointers to the allocated memory) is an error with an unpredictable outcome.

Data in a released block is available to the user, but VSIPL functions should not operate on it, since its state is outside its scope of control. User blocks are in the released state when created, and must be admitted to VSIPL before VSIPL functions can operate on the data bound to the block. A user block may be admitted to VSIPL space and released to user space as needed, depending on whom requires direct access to the data. The characteristics and interrelationship between these two blocks are illustrated in Figure A-1.

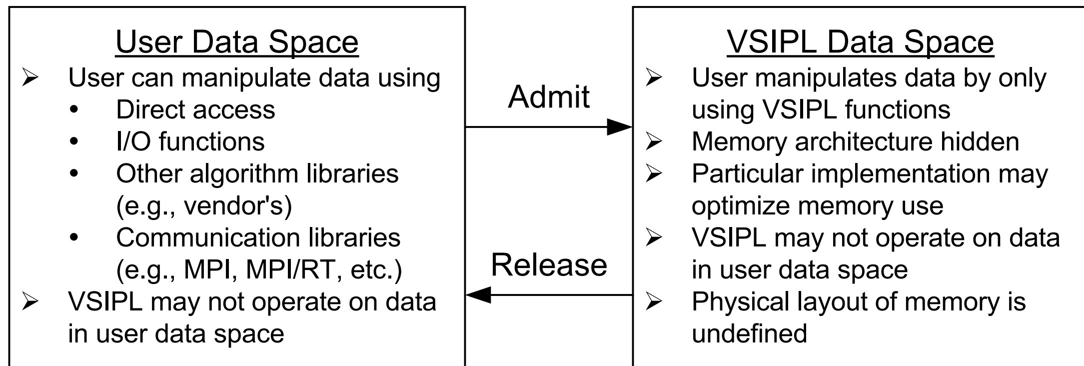

Figure A-1. Data space characteristics and interrelationship.

While blocks represent logically contiguous data areas in memory, users often require operation on non-contiguous subsets of these data areas. VSIPL provides access to the elements of the block through another VSIPL object called a *view*, regardless of contiguity. VSIPL views allow the user to specify contiguous or noncontiguous subsets of a data array, as well as specify how the data will be accessed, e.g., vector, matrix, or tensor. View parameters that need to be set include an offset from the beginning of the block, the length of the view (which is the number of elements), and a stride value specifying the number of interim block elements between view elements (as defined in the type of the block). E.g., for a block with a data array of 1024 elements, the view may have an offset of 512 (using the second half of the block), a stride of 16, and a length of 32.

It is important to note that a block may have more than one view attached to it, e.g., one matrix view may be set up to view the rows of the matrix while another view may be set up to view the columns of the matrix. Or if a vector represents multiplexed data of four channels, then four vectors of the same stride (i.e., four) and length could be created, but with four different offsets (0, 1, 2, and 3). Also, since the blocks are typed, so are the views, and they are immutable, which means that an integer view and a float view cannot both be associated with the same block. In creating multiple views, VSIPL allows them to be created from existing blocks or views, changing parameters if desired (e.g., offset, stride, and length).

A.1.2.3 Functions

VSIPL functions comprising the Core and Core Lite profiles are tabulated by the following function groups in the Appendices of [41], where the Core Lite functions have been emboldened:

I. Block & View Functions
II. Scalar Functions
III. Vector & Matrix Elementwise Functions
IV. Signal Processing Functions
V. Linear Algebra Functions

Also, the Core Lite profiles do not support all the function variations or datatypes, and there are no linear algebra functions in the Core Lite profile. See the VSIPL web site for more information on the profile specifics [172, 173].

Note that these tables are illustrative and not definitive. The naming convention details and datatypes are contained in the v1.0 API specification. A VSIPL function name is always prefixed with `vsip_` and then has leading and trailing characters before and after the function name that describe the arguments, e.g., complex and/or real, floating-point and/or integer and/or Boolean, etc.

A.1.2.4 Developing a VSIPL Application

Basic programming specifics require including the preprocessor directive that include the declarations and definitions needed for compiling a VSIPL program:

```
#include "vsip.h"
```

VSIPL uses a consistent scheme for VSIPL-defined identifiers; they all begin with "`vsip_`". The rest of the identifier includes characters indicating real and/or complex, function name, and data type and/or precision.

Canonical development of embedded signal processing applications using COTS multiprocessing hardware and software typically consists of partitioning the code into two portions. One portion is the "outer loop" where the setup and cleanup functions are executed, typically memory allocation and coefficient generation, such as FFT twiddle factors and window coefficients. The other portion is the "inner loop" where the time-critical repetitive streaming data transformation functions lie. A VSIPL application will be built similarly, with the outer loop executing heavyweight system functions that allocate memory when creating blocks and parameterized accessors called views. The block creation is substantial, while the view object handles take up very little memory, but do require system support.

The inner loop contains the computation functions, such as the scalar, elementwise, signal processing, and linear algebra functions. Assuming the application does terminate for a given mission, then the outer loop would conclude after the inner loop concludes, destroying views and blocks. This is illustrated in Figure A-2.

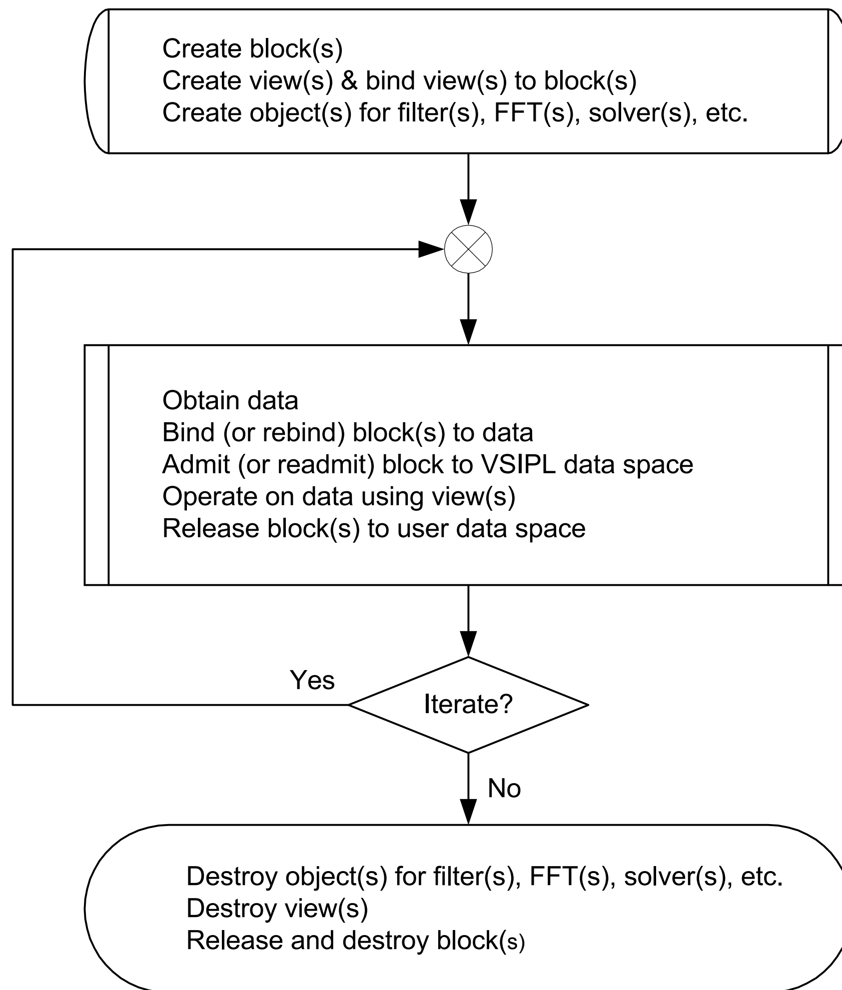

Figure A-2. VSIPL application flow.

A.2 MPI: Communications Middleware

There have been a number of approaches to accomplishing parallel processing, a topic of breadth and depth that is beyond the scope of our discussion. Suffice to say, out of the plethora of approaches (hardware and/or software) grew an approach that has gained growing support and become a standard. Rather than trying to develop a special language (such as HPF, High Performance Fortran) and concomitant compiler, a library of functions was specified to achieve parallelism by message passing, explicitly transmitting data from one process to another. Message passing is a powerful and very general method of expressing parallelism and can be used to create extremely efficient

parallel software applications. It has become a widely used method of programming for many types of parallel computers [47, 153-155].

A.2.1 Standardization and Functionality

Message passing is especially popular on scalable parallel computers (SPCs) with distributed memory, and on Networks of Workstations (NOWs). There have been many variations over the last ten to fifteen years, with each variation helping to crystallize what is core and critical to the message passing paradigm. About five years ago a consortium known as the Message-Passing Interface (MPI) Forum formed to define both the syntax and semantics of a standard core of library routines that would be useful to a wide range of users and efficiently implementable on a wide range of computers. The MPI Forum was made up of over 80 people from 40 organizations of vendors, users, and researchers. Their goals included portability but not at the expense of performance, including heterogeneous platforms, and multiple language bindings including C and Fortran [18, 156].

The MPI standard includes the following characteristics, features, and functionality:

- Point-to-point communication
- Collective operations
- Process groups
- Communication domains
- Process topologies
- Environmental management and inquiry
- Profiling interface
- Bindings for Fortran and C

The MPI standard does *not* specify:

- Explicit shared-memory operations
- Operations requiring OS support not standard during standardization
- Program construction tools

- Debugging facilities
- Explicit support for threads
- Support for task management
- I/O functions

The MPI Forum continues to meet and has established the v1.x standard. A major revision to v2.0 is under discussion. The MPI Forum maintains a web site providing up to date status and documentation: http://www.mpi-forum.org.

A.2.2 Basic Theory of Operation

For the applications in our ADoI, the parallel programming model will be single-program multiple-data (SPMD). In strict MPI terms, the executable images are identical, with the process having to identify itself and branch accordingly to operate on the data as a function of its process rank. This model as applied to our ADoI has the same computational code, but operating on different tiles of the data square. Consequently, while VSIPL is the computational middleware and MPI is the communication middleware, the application software is actually a set of MPI programs. Communication and control are accomplished by the MPI middleware, determining what processes operate on what and when. The processing itself is accomplished by VSIPL middleware.

Basic programming specifics require including the preprocessor directive that includes the declarations and definitions needed for compiling an MPI program:

```
#include "mpi.h"
```

MPI uses a consistent scheme for MPI-defined identifiers; they all begin with "`MPI_`". The remaining of most MPI constants are capital letters. The first character of the rest of an MPI identifier is capitalized with the balance being lower-case, e.g., MPI_Send. Before any other MPI function can be called, the outer-loop code must call:

```
int MPI_Init
(
     int*          argc          /* in/out */,
     char**        argv[]        /* in/out */
)
```

After inner-loop execution completes, the outer-loop cleans up by executing:

```
int MPI_Finalize(void)
```

A process finds out how many processes are in a program's execution by calling:

```
int MPI_Comm_size
(
      MPI_Comm          comm                  /* in */,
      int*        number_of_processes         /* in/out */
)
```

A *communicator* is a collection of processes that can send messages to each other. The predefined communicator consisting of all the processes running when program execution begins is `MPI_COMM_WORLD`. A process identifies its rank by calling:

```
int MPI_Comm_rank
(
      MPI_Comm          comm                  /* in */,
      int*              my_rank               /* in/out */
)
```

The two fundamental functions that accomplish the actual message passing are `MPI_Send` and `MPI_Recv`. `MPI_Send` sends a message to a designated process and `MPI_Recv` receives a message from a process. Their prototypes are:

```
int MPI_Send
(
      void*             message               /* in */,
      int               count                 /* in */,
      MPI_Datatype      datatype              /* in */,
      int               dest                  /* in */,
      int               tag                   /* in */,
      MPI_Comm          comm                  /* in */
)

int MPI_Recv
(
      void*             message               /* out */,
      int               count                 /* in */,
      MPI_Datatype      datatype              /* in */,
      int               source                /* in */,
      int               tag                   /* in */,
      MPI_Comm          comm                  /* in */,
      MPI_Status*       status                /* out */
)
```

The parameter `message` refers to the actual data being transmitted. The parameters `count` and `datatype` determine the size of the message. `MPI_Recv` doesn't need to know the exact size of the message being received, but it must have at least as much space as the size of the message

intended to be received. The `tag` and `comm` are used internally by the middleware to make sure that messages are not confused within the channels. Since `MPI_Recv` can use wildcards for `source` and `tag`, the `status` parameter returns the source and tag of the message that was actually received.

To avoid confusion in the middleware internals and to limit the information senders and receivers require, every message consists of two parts, the data being transmitted and its "envelope." The envelope contains at least the following information:

- The rank of the receiver
- The rank of the sender
- A tag
- A communicator

More information on the exact syntax and strategies can be found in [46, 47, 156].

Appendix B

Details of Case Study

B.1 Simulink Details

We provide here some of the lower level details required in building and manipulating the executable specification in the DSP Workstation.

B.1.1 Simulink Start-up and Initializing MATLAB Workspace

When this model is opened there is a MATLAB file (For_sl.m) that is executed via the `PreLoadFcn` (pre-load function) that loads MATLAB variables into MATLAB's workspace:

```
echo on;

%Global values
npulses=16;
ntaps=8;
nrange=2032;
nfft=2^nextpow2(nrange);

%Input odd and even data (Matlab matrices)
load fir_in_even
load fir_in_odd

%Truncate fir_in_* due to Simulink limitations
fir_in_even=fir_in_even(:,1:npulses);
fir_in_odd=fir_in_odd(:,1:npulses);

%FIR coefficients
load odd_kern;
load even_kern;

%Mask of 1 and -1 to multiply samples by modulation
modmask=ones(1,nrange);
modmask(2:2:end)=-1;
```

```
%Range processing coefficients
load taylor_kern
load rcs_kern

%Azimuth processing stuff
load kern_index
load az_kern_full

echo off;
```

The "Global values" would correspond to systemic parameters, perhaps declared in a `#define`. The input data of one 512-pulse frame has been split into the even and odd value matrices.

B.1.2 Addressing Matrices as Vectors

While Simulink 3 now supports matrices, it has not completely fixed the interface to allow signals to be addressed in a matrix fashion. This legacy artifact requires the user to address matrices as vectors. Double-clicking on the even input block is shown in Figure B-1.

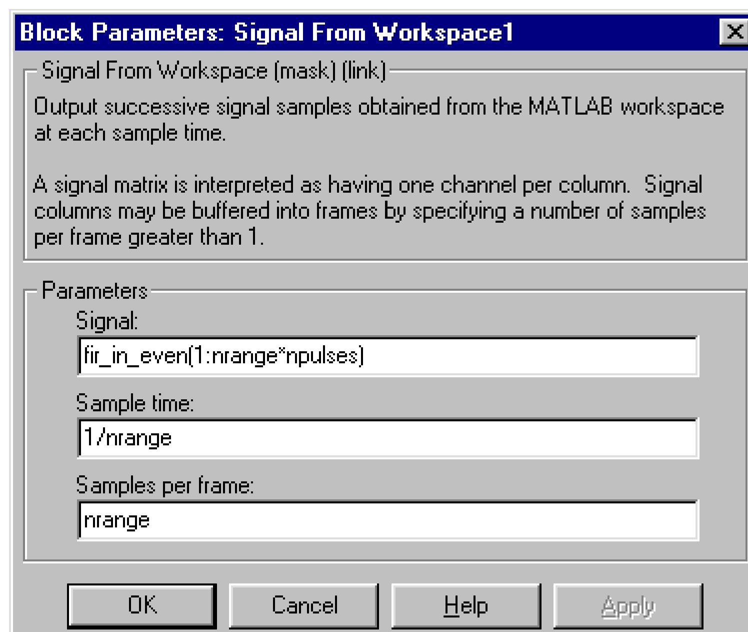

Figure B-1. Data input block.

The signal description in the parameters block implies that `fir_in_even` is a vector and not a matrix. This is another artifact of pre-Simulink 3 versions. While Simulink 3 provides frame-based processing, it cannot yet treat signals as matrices like the data square of SAR processing. The sample time and samples per frame keep our model normalized with respect to time.

B.1.3 Stripping Off Previous SAR Image Frame

Double-clicking on the format display block shows how Simulink does this using MATLAB notation in its block parameters dialog box shown in Figure B-2.

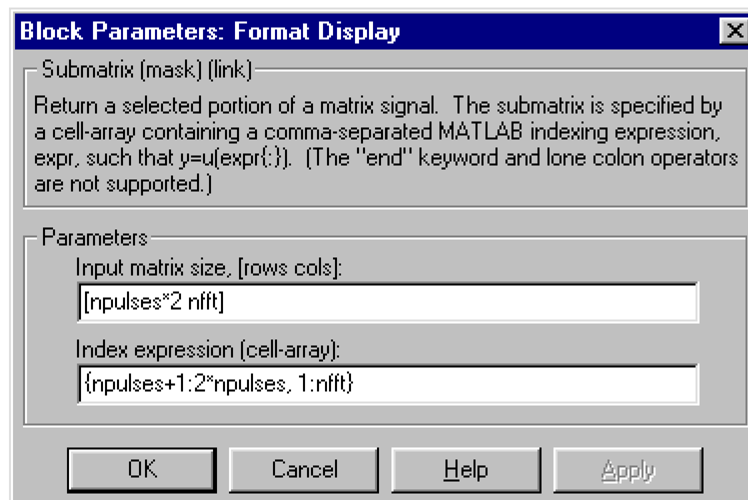

Figure B-2. Formatting the SAR image for output.

B.1.4 Executing the Specification and Flushing its Queue

To actually execute the model we go into the Simulation Parameters dialog box as shown in Figure B-3 and set the start time to 0.0 and the stop time to `npulses`. This artifice allows to make sure that the model actually executes completely and that the output gets flushed to the workspace with the image and not just zeros.

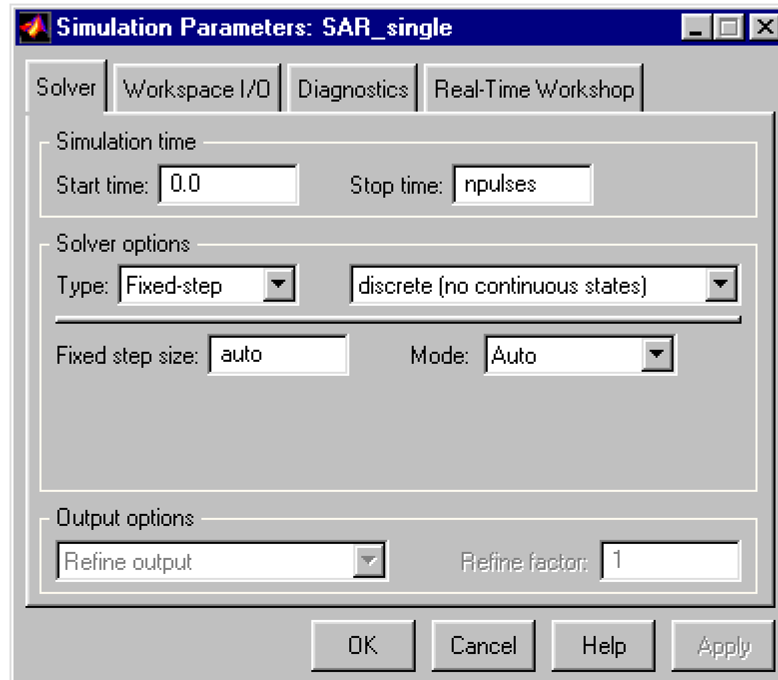

Figure B-3. Setting the model execution parameters.

B.1.5 Scatter/Gather with Demux/Mux

Double-clicking on the demux/scatter block produces the dialog box shown in Figure B-4.

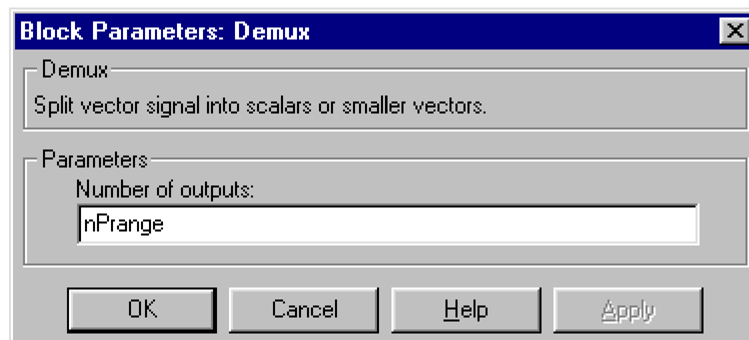

Figure B-4. Demux dialog box used for scattering the data for range processing.

Since the DSP Blockset is frame-based, demuxing distributes columns (pulses) of the matrix (data square), despite its obfuscating way of mixing memory and data models in the GUI.

The converse (mux/gather) is performed between the range processors and the cornerturn as shown in Figure B-5.

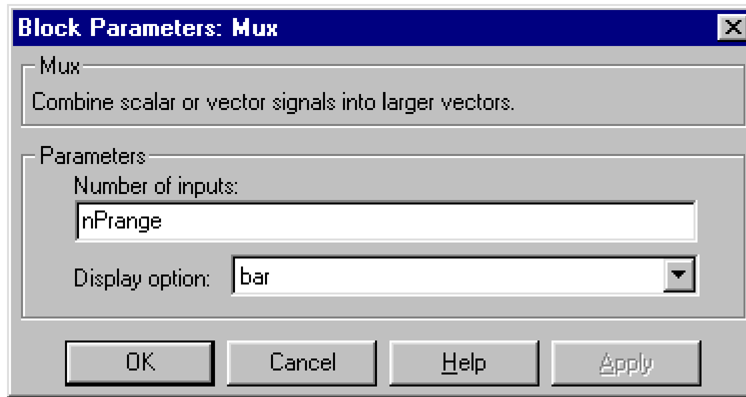

Figure B-5. Mux dialog box used for gathering range results for the cornerturn.

Demux is repeated at the output of the cornerturn when it scatters the transpose to the azimuth processors. Mux is repeated at the input of the display process when it gathers the azimuth results.

B.2 VSIPL Code Generation Subtleties

There are some subtleties in generating VSIPL code from a Simulink model that are presented here to document them, since they were out of scope in §7.6.1. We present the Simulink or MATLAB function and its VSIPL equivalent in Table B-1.

Table B-1. Subtle VSIPL code generation equivalents.

Framework	Function	VSIPL Equivalent
MATLAB	`flipud`, `fliplr`	Set offset to end and stride backwards.
Simulink DSP Blockset	Matrix constant	Use scalar-vector (`vsip_sv*`) or scalar-matrix (`vsip_sm*`) functions .
Simulink DSP Blockset	Autocorrelation	Do 1D correlation (vsip_correlate1d) using the same signal for both vector input arguments.
Simulink DSP Blockset	Difference	Use `vsip_vsub_f` using the same vector as inputs, but with the offset set to one in the minuend.
Simulink DSP Blockset	Zero pad	Create a large enough block and view and "fill" it with zero in the outer loop.
MATLAB Simulink DSP Blockset	IFFT	Use the same function as for FFT, but with direction flag set for inverse.
Simulink DSP Blockset	FIR decimation	Decimation is built into the VSIPL FIR function; a separate call is not needed.

B.3 eArchitect Details

We provide here some of the lower level details required in building the performance model and simulating our system's behavior using eArchitect.

B.3.1 Starting Up eArchitect

We begin by opening the eArchitect performance modeling framework getting the project window shown in Figure B-6. This is our access point to the different tools within the framework. At the beginning of any session we had to open a Mercury-specific library, mercury.lib as shown in Figure B-7. This library contains some Mercury RACE-specific hardware files to supplement the eArchitect framework for the performance

modeling as required by our MAGIC SDM. We found out that while eArchitect has been very useful to designers building systems targeted to run on VME or PCI and RACEway, but for single boards designed to plug into RACEway, we were the second users to try to use it to model MP architectures. Hence, we needed the mercury.lib "Band-Aids."

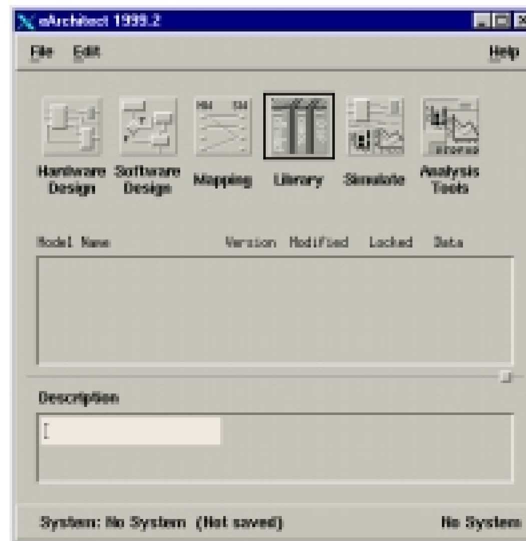

Figure B-6. Project window at the beginning of an eArchitect session.

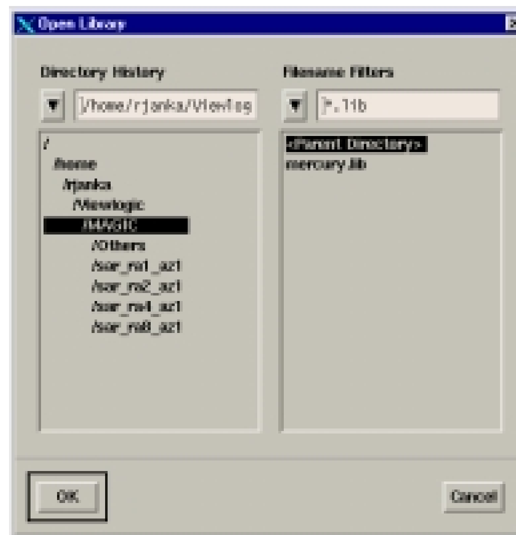

Figure B-7. Opening the supplemental Mercury library to support our MAGIC SDM.

B.3.2 Hardware Model Layers

We go down into the different layers of the chassis hardware model of Figure 7-12 in Figure B-8 to Figure B-10. The ILK4 is basically just a RACE crossbar, which is a component in the eArchitect hardware library. We created the motherboard architecture as shown in Figure B-8.

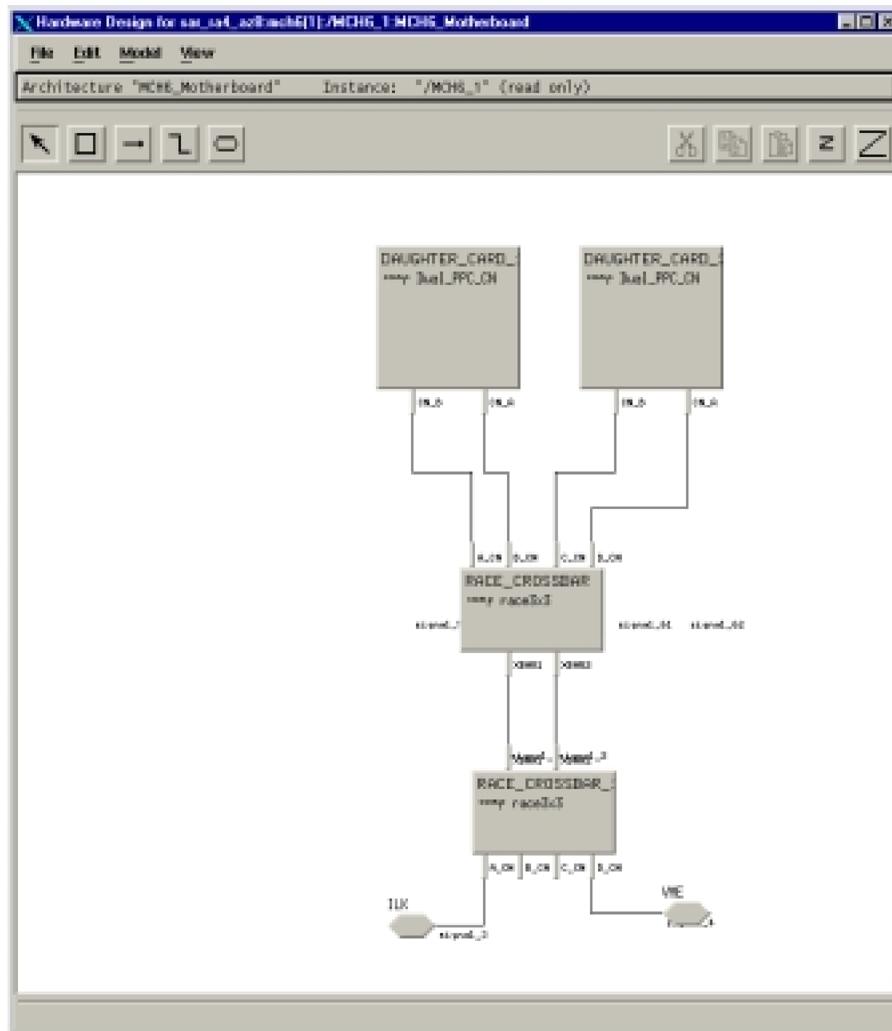

Figure B-8. Hardware model of Mercury MCH6 motherboard.

A hardware model of the daughtercard contains two PPC603 nodes and shown in Figure B-9.

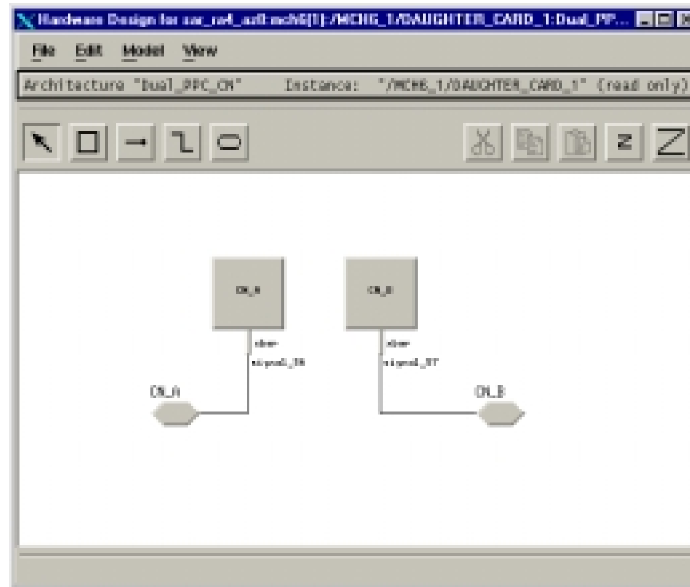

Figure B-9. Hardware model of Mercury PPC daughtercard.

The two CNs in Figure B-9 are the fundamental compute nodes (CEs) that constitute our system. One of these two CNs is shown in Figure B-10.

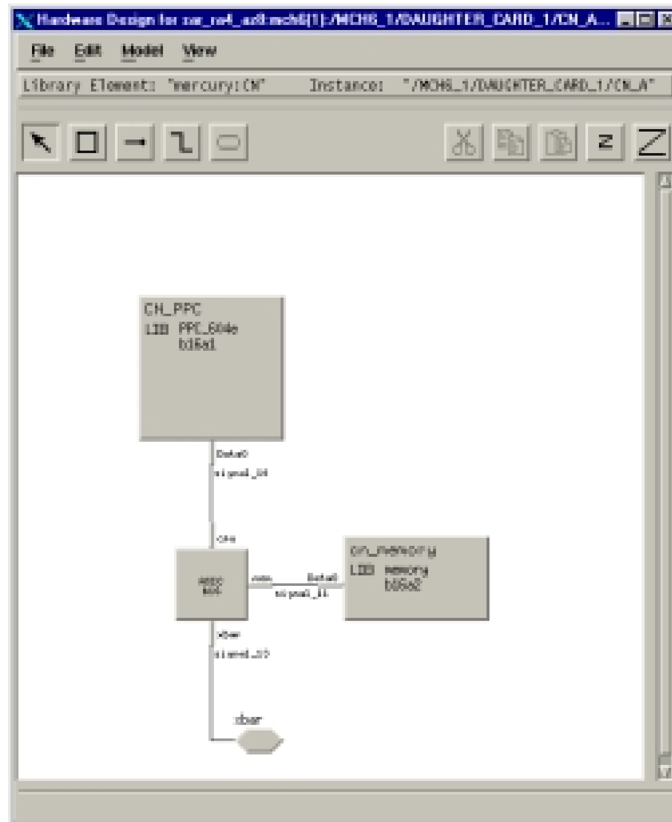

Figure B-10. One of two compute nodes (CE) on daughtercard.

B.3.3 Software Editor GUI Details

The top-level of the eArchitect software editor is shown in Figure 7-13. There are two components used to model the software, blocks and messages. There are other buttons (on the left) available for in-ports, out-ports, and timers. The buttons on the right are for cut, copy, paste, and zoom in/out.

B.3.4 Scatter/Gather Details

Some processes have to perform scattering and/or gathering in addition to computation. The template created to account for this has a loop for gathering at the beginning of the flowchart (`numgather`>1) and a loop for scattering at the end of the flowchart (`numscatter`>1).

In our application, range processing does not gather, so `numgather`=1. In other words, the range processes do not gather, they receive their data from FIR processing. Then the data is processed, and sent to the next process. There is another loop construct here to support scattering (`numscatter`>1) or single send (`numscatter`=1). There are three processing blocks where we do not process data, but account for the time delay resulting from that processing. These delays are accomplished with the three executeCycles commands as shown in Figure 7-14 and in the gather loop contents shown in Figure B-11.

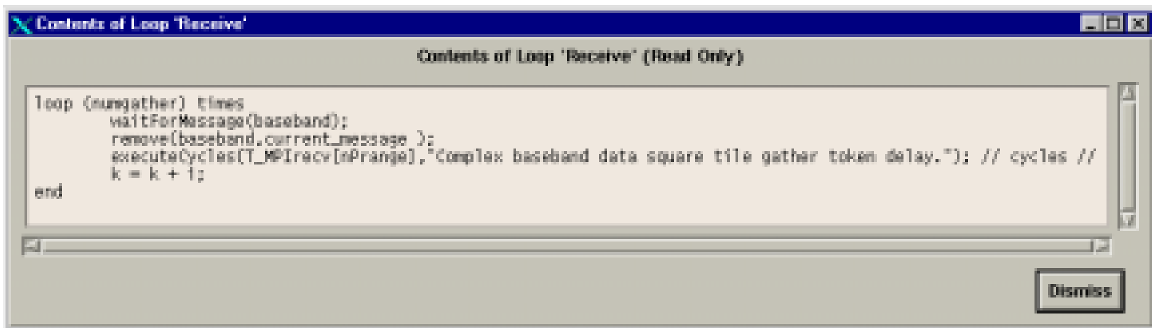

Figure B-11. Contents of a software block's process that performs a delay.

B.3.5 Coding the Processes other than for Range Processing

Having explained the basic logic of the mapping, scatter and gather internal logic, and token delays in §7.6.3.2, we present how we configured the processes other than for range processing. We show the local variables for the other software blocks in Figure B-12–Figure B-15.

Edit Local Variables/Constants for Architecture Block_7_task

Defined Local Variables/Constants

Name	Type	Size/Width	Kind	Value
current_message	Message	--	Variable	
queueStatus	Boolean	--	Variable	
replySize	DataSize	--	Variable	
replyValue	Integer	--	Variable	
nrange	Integer	--	Constant	2032
npulses	Integer	--	Constant	512
Tsample	Integer	--	Constant	1
proclist	Task/Block Name Array	16	Constant	/MCH6_1/DAUGHTER_CARD_1/CN_A/CN_PPC, /MCH6_1/
scatterlist	Int. Array	8	Constant	4, 5, 6, 7, 8, 9, 10, 11
dest	Task/Block Name	--	Variable	
scattersize	DataSize	--	Variable	0 bit_size
numgather	Integer	--	Constant	1
k	Integer	--	Variable	
nPrange	Integer	--	Constant	4
T_input	Integer	--	Constant	10407683
T_VSIPL	Integer	--	Constant	25447454
T_MPIsend	Int. Array	9	Constant	0, 10424042, 5212021, 0, 2606011, 0, 0, 0, 1303005

New | Edit | Delete | File | Dismiss

Figure B-12. Local variables for FIR processing.

Edit Local Variables/Constants for Architecture Block_7_task

Defined Local Variables/Constants

Name	Type	Size/Width	Kind	Value
current_message	Message	--	Variable	
queueStatus	Boolean	--	Variable	
replySize	DataSize	--	Variable	
replyValue	Integer	--	Variable	
nrange	Integer	--	Constant	2032
npulses	Integer	--	Constant	512
Tsample	Integer	--	Constant	1
proclist	Task/Block Name Array	16	Constant	/MCH6_1/DAUGHTER_CARD_1/CN_A/CN_PPC, /MCH6_1/DA
scatterlist	Int. Array	8	Constant	15, 14, 13, 12, 11, 10, 9, 8
dest	Task/Block Name	--	Variable	
scattersize	DataSize	--	Variable	0 bit_size
numscatter	Integer	--	Constant	4
numgather	Integer	--	Variable	4
k	Integer	--	Variable	
nPrange	Integer	--	Constant	4
nPazimuth	Integer	--	Constant	8
T_MPIrecv	Int. Array	9	Constant	0, 10547647, 5273824, 0, 2636912, 0, 0, 0, 1318456
T_VSIPL	Integer	--	Constant	340756139
T_MPIsend	Int. Array	9	Constant	0, 21095295, 10547647, 0, 5273824, 0, 0, 0, 2636912

New | Edit | Delete | File | Dismiss

Figure B-13. Local variables for the cornerturn.

Figure B-14. Local variables for azimuth processing.

Figure B-15. Local variables for display processing.

B.3.6 Setting Up eArchitect for Simulation

The data input is set up using the dialog window shown in Figure B-16, which is where we set the `data_gen` parameters in our hardware model (cf. Figure 7-12). The key parameters to set for `data_gen` are `period` (set to 3-seconds), `size` (512 pulses of 4064 real values of 4 bytes each), and `throughput` (8 MB/s), which is a misnomer, i.e., it is not our system's throughput but the simulator's throughput. It determines the granularity of the simulator's timeline display, and is best set to slightly larger than `size`/`period`.

Figure B-16. Parameters of the input data source.

B.3.7 Running Simulations in eArchitect

When we launch the individual simulations for each of the architectures, we set the run size and provide a name for the run, e.g., `sim_ra4_az8_3s` to reflect the configuration (`nPrange`=4, `nPazimuth`=8) and simulation duration (3 seconds). The dialog window for launching a simulation is seen in Figure B-17. After clicking on the start button, the VHDL code is generated and simulated, and when finally completed brings up the analysis dialog window shown in Figure B-18.

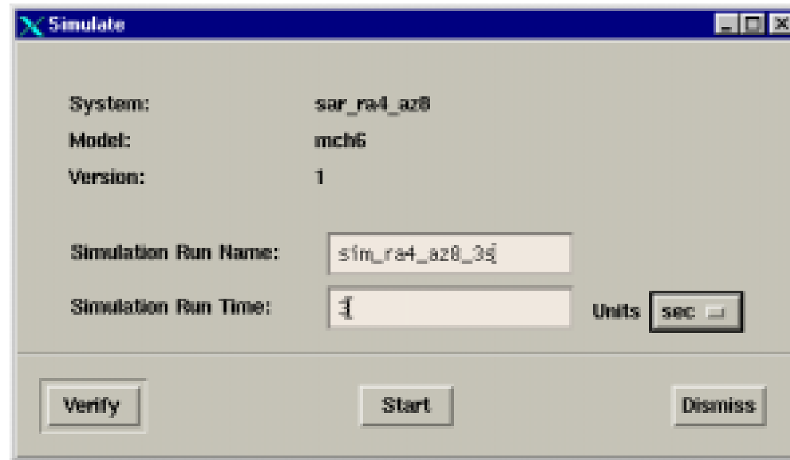

Figure B-17. Dialog window to set parameters for simulation.

Figure B-18. Tool for accessing simulation runs for analysis.

Using the File pull-down menu allows us to Create Analysis Control… and open the Analysis Control Panel shown in Figure B-19.

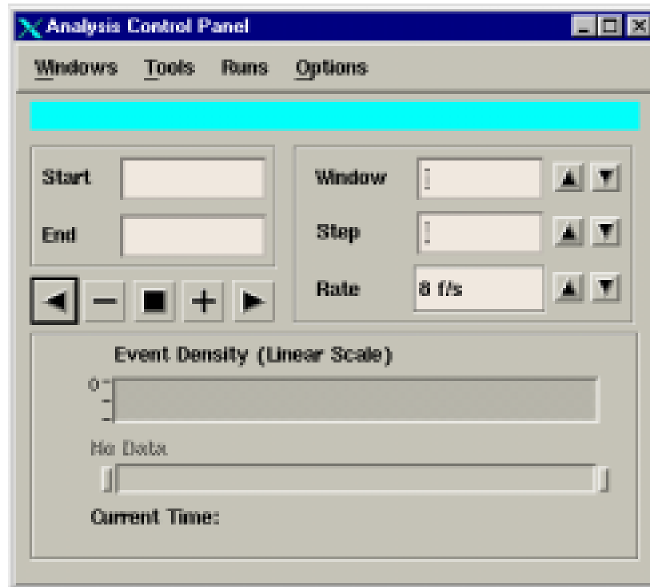

Figure B-19. Analysis tools for visualization of simulation data.

Our `sim_ra4_az8_1.6s` simulation is selected with the Runs pull-down menu in the Analysis Control Panel which brings up the bottom-line data in the Analysis Control Panel as shown in Figure B-20. Sliding the Current Time indicator to the edge of the Event Density shows that the approximate latency of this architecture is approximately 1.21 seconds.

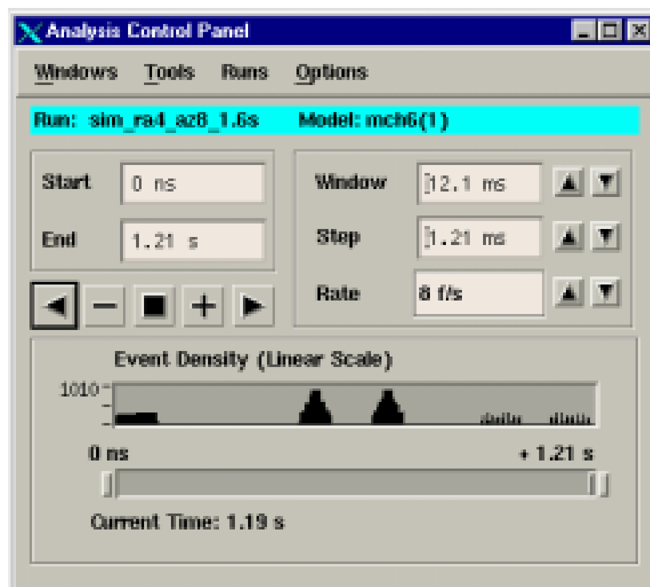

Figure B-20. After loading our simulation run we see the bottom-line latency.

Clicking on the Tools pull-down menu produces the following tools options as shown in Figure B-21.

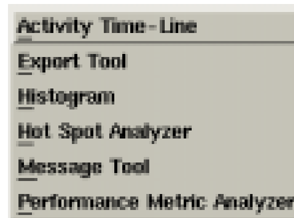

Figure B-21. Tools options in the Analysis Control Panel.

The Activity Time-Line is what we are most interested in since it will give us the exact latency as shown in Figure B-22. We use the slide bars to touch the edge of the end of the Display processing to determine the exact latency.

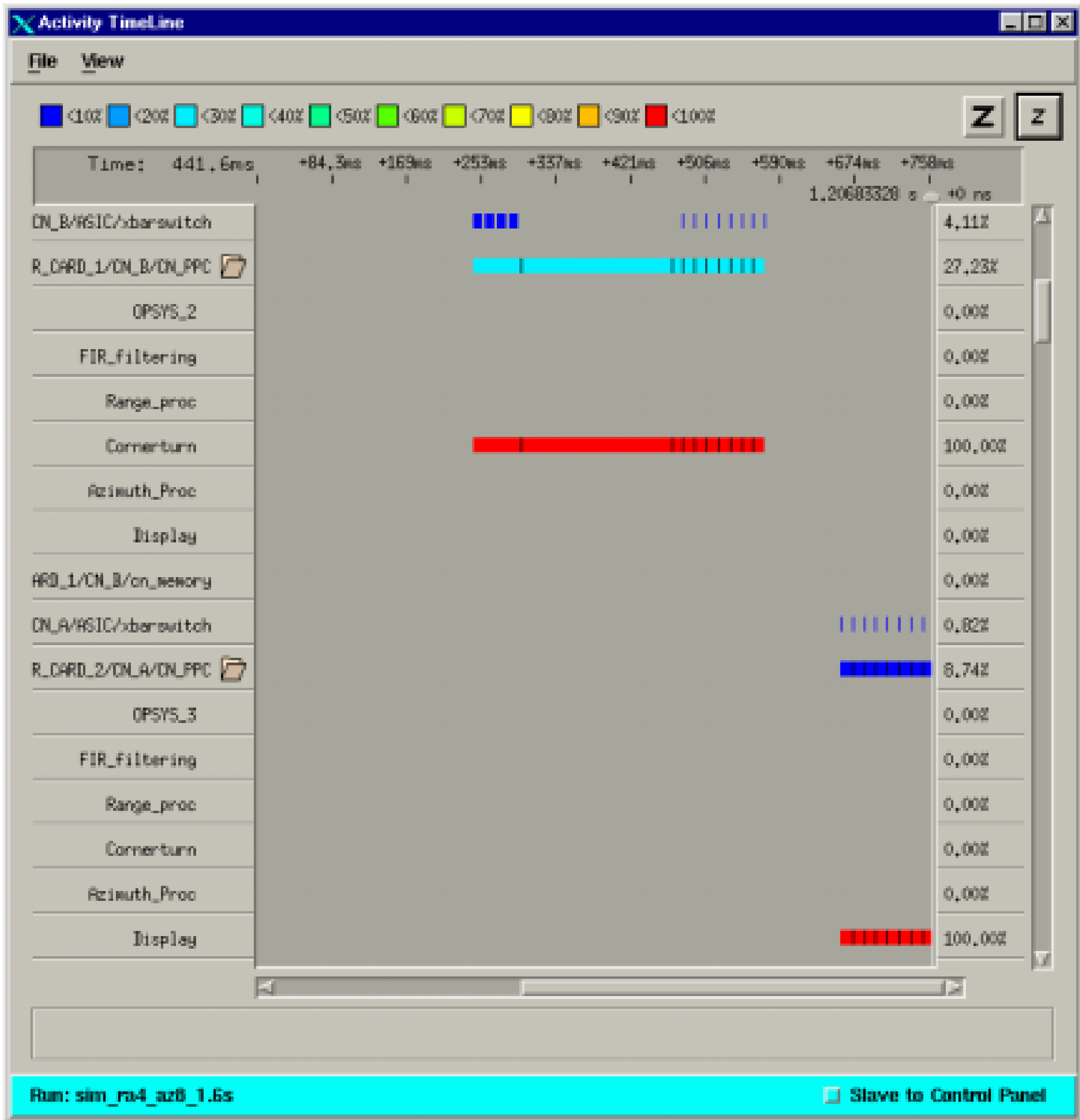

Figure B-22. Simulation time-line showing the exact latency of the simulation is 1.207 seconds.

References

[1] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 4th ed. New York, NY: McGraw-Hill, 1997.

[2] L. Scanlan, W. Lee, M. Vahey, and M. McCollough, “RASSP Methodology Evaluation and Lessons Learned Developing IRST Signal Processor,” in *Rapid Prototyping of Application Specific Signal Processors*, M. A. Richards, A. J. Gadient, and G. A. Frank, Eds. Boston, MA: Kluwer Academic Publishers, 1997, pp. 145-160.

[3] M. A. Richards, A. J. Gadient, and G. A. Frank, “Rapid Prototyping of Application Specific Signal Processors,” in *Journal of VLSI Signal Processing*, vol. 15, S. Y. Kung, Ed., 1st ed. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1997, pp. 200.

[4] A. Sarkar, R. Waxman, and J. P. Cohoon, “Specification-Modeling Methodologies for Reactive-System Design,” in *High-Level System Modeling: Specification Languages*, vol. 3, *Current Issues in Electronic Modeling*, J.-M. Bergé, O. Levia, and J. Rouillard, Eds., 1st ed. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1995, pp. 1-34.

[5] A. Sarkar, “Integrating Operational Specification and Performance Modeling for Digital-System Design,” Ph.D. thesis in *Engineering and Applied Science*. Charlottesville, VA: University of Virginia, 1995, pp. 157.

[6] V. K. Madisetti and T. W. Egolf, “Virtual Prototyping of Embedded Microcontroller-Based DSP Systems,” in *IEEE Micro*, vol. 15, 1995, pp. 9-21.

[7] V. K. Madisetti and M. A. Richards, “Advances in Rapid Prototyping of Digital Systems,” in *IEEE Design & Test of Computers*, vol. 13, 1996, pp. 9-11.

[8] V. K. Madisetti, “Rapid Digital Systems Prototyping: Current Practice, Future Challenges,” in *IEEE Design & Test of Computers*, vol. 13, 1996, pp. 12-22.

[9] J. C. Anderson, “Predicting the Future with RASSP Benchmarks,” presented at First Annual RASSP Conference, Arlington, Virginia, 1994.

[10] A. H. Anderson, G. S. Downs, and G. A. Shaw, “RASSP Benchmark-1 and -2: A Preliminary Assessment,” presented at Second Annual RASSP Conference, Arlington, Virginia, 1995.

[11] C. Hein, T. Carpenter, P. Kalutkiewicz, and V. Madisetti, "RASSP VHDL Modeling Terminology and Taxonomy - Revision 1.0," Revision 1.0, May 1996.

[12] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, 1987.

[13] E. A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, pp. 1217-1229, 1998.

[14] E. K. Pauer and J. B. Prime, "An Architectural Trade Capability Using the Ptolemy Kernel," presented at International Conference on Acoustics, Speech, and Signal Processing (ICASSP 96), Atlanta, GA, 1996.

[15] VSIPL Forum, "VSIPL v1.0 API Standard Specification," DARPA and the Navy, Draft 1999.

[16] D. A. Schwartz, "Vector, Signal, & Image Processing Standardization for Embedded Systems: VSIP 1.0 API," presented at Second Annual Workshop on High Performance Embedded Computing, Lexington, MA, 1998.

[17] MPI/RT Forum, "Document for the Real-Time Message Passing Interface (MPI/RT-1.0) Draft Standard," DARPA, Draft standard February 1, 1999.

[18] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," University of Tennessee, Knoxville, TN, Standard Version 1.1, June 1995.

[19] T. Skjellum and A. Kanevsky, "MPI/RT: Real-Time MPI Standard and Committee," http://www.mpirt.org, October 31, 1997.

[20] A. Kanevsky, A. Skjellum, and A. Rounbehler, "MPI/RT – An Emerging Standard for High-Performance Real-Time Systems," presented at 31st Hawaii International Conference on System Sciences (HICSS'98), Kohala Coast, HI, 1998.

[21] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Englewood Cliffs, NJ: P T R Prentice Hall, 1994.

[22] D. D. Gajski, S. Narayan, L. Ramachandran, F. Vahid, and P. Fung, "System Design Methodologies: Aiming at the 100 h Design Cycle," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, pp. 70-82, 1996.

[23] F. Vahid, S. Narayan, and D. D. Gajski, "SpecCharts: A VHDL Front-End for Embedded Systems," *IEEE Transactions on CAD*, vol. 14, pp. 694-706, 1996.

[24] R. Waxman, J.-M. Bergé, O. Levia, and J. Rouillard, "High-Level System Modeling: Specification and Design Methodologies," in *Current Issues in*

Electronic Modeling, vol. 4, J.-M. Bergé, O. Levia, and J. Rouillard, Eds., 1st ed. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1996, pp. 192.

[25] J. P. Calvez, *Embedded Real-Time Systems: A Specification and Design Methodology*, vol. 15, 1st ed. Chichester, West Sussex, England: John Wiley & Sons Ltd., 1993.

[26] E. A. Lee and A. Sangiovanni-Vincentelli, "Comparing Models of Computation," presented at 1996 IEEE/ACM International Conference on Computer-Aided Design, San Jose, California, 1996.

[27] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models, Validation, and Synthesis," *Proceedings of the IEEE*, vol. 85, pp. 366-390, 1997.

[28] N. Halbwachs, *Synchronous Programming of Reactive Systems*, vol. 215. Boston: Kluwer Academic Publishers, 1992.

[29] R. S. Janka, R. Judd, J. Lebak, M. A. Richards, and D. A. Schwartz, "API and Product Status of the v1.0 Vector, Signal, and Image Processing Library (VSIPL)," presented at Third Annual Workshop on High Performance Embedded Computing, Lexington, MA, 1999.

[30] R. Janka, "Multiprocessor Software Development for RACEway-based Real-Time Signal Processing Systems," presented at Real-Time Computer Show & Conference, San Jose, CA, 1997.

[31] R. Janka, "The Use of Application Software Mapping Tools for Real-Time Embedded Multiprocessor Signal Processing Systems," presented at DSP World Spring Design Conference, Washington D.C., 1997.

[32] R. Janka, "Bridging the Development Gap of Real-Time Embedded Multiprocessor Signal Processing Systems," presented at DSP World Spring Design Conference, Washington D.C., 1997.

[33] R. Janka, "Graphical Tools Enhance Productivity," in *Electronic Engineering Times*, 1997, pp. 66.

[34] R. Janka, A. Clouard, O. Debon, and J.-C. Mison, "Graphical Application Software Development for Deployable Heterogeneous Multicomputers," presented at Eighth International Conference on Signal Processing Applications and Technology, San Diego, CA, 1997.

[35] R. Janka, "Advanced Software Tools," presented at World-Wide Sales Meeting, Nashua, NH, 1997.

[36] R. Janka, “An Integrated Unified Middle-Layer Specification and Design Methodology for Large Multiprocessor DSP Systems,” Georgia Institute of Technology, Atlanta, GA, Working Paper, October 4, 1998.

[37] R. Janka, “A New Development Framework Based on Efficient Middleware for Real-Time Embedded Heterogeneous Multicomputers,” presented at 1999 IEEE Conference and Workshop on Engineering of Computer-Based Systems (ECBS '99), Nashville, Tennessee, 1999.

[38] R. S. Janka, “Specification and Design Methodology for Large Multiprocessor DSP Systems Based on Integrated versus Unified Models of Computation,” Georgia Institute of Technology, Atlanta, GA, Working Paper, April 19, 1999.

[39] R. Janka, “Models of Computation for Specification and Design Methodology Frameworks for Parallel and Distributed Real-Time Embedded Multiprocessor Signal Processing Systems,” presented at The 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), Las Vegas, NV, 1999.

[40] R. Janka and L. Wills, “Early System-Level Design Exploration of Large DSP Systems Targeted for Real-Time Embedded COTS Multiprocessors,” presented at The International Conference on Signal Processing Applications and Technology and DSP World Workshops (DSP World - ICSPAT), Orlando, FL, 1999.

[41] R. Janka, “VSIPL: Computational Middleware for Portable Real-Time Embedded Multicomputing Application Software,” presented at The International Conference on Signal Processing Applications and Technology and DSP World Workshops (DSP World - ICSPAT), Orlando, FL, 1999.

[42] R. Janka and L. Wills, “Considering Models of Computation in Developing a New Specification and Design Methodology for Large Real-Time Embedded Multiprocessor Signal Processing Systems,” presented at accepted for the IEEE International Workshop on Intelligent Signal Processing (WISP'99), Budapest, Hungary, 1999.

[43] R. S. Janka, “A Model-Continuous Specification and Design Methodology for Large Multiprocessor DSP Systems,” Georgia Institute of Technology, Atlanta, GA, Dissertation Proposal, June 25, 1999.

[44] T. W. Egolf, “Virtual Prototyping of Embedded Digital Systems: Hardware/Software Codesign, Integration, and Test,” Ph.D. thesis in *Electrical and Computer Engineering*. Atlanta, GA: Georgia Institute of Technology, 1997, pp. 204.

[45] J. Sztipanovits, G. Karsai, and T. Bapty, “Self-Adaptive Software for Signal Processing: Evolving Systems in Changing Environments without Growing Pains,” *Communications of the ACM*, vol. 41, pp. 66-73, 1998.

[46] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA: The MIT Press, 1994.

[47] P. S. Pacheco, *Parallel Programming with MPI*. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1997.

[48] MPI Software Technology Inc., “MPI/PRO,” MPI Software Technology, Inc.,, WWW http://www.mpi-softtech.com/, 1998.

[49] A. Kanevsky, A. Skjellum, and J. Watts, “Standardization of a Communication Middleware for High-Performance Real-Time Systems,” presented at IEEE Workshop on Middleware for Distributed Real-Time Systems and Services (Held in conjunction with the 18th IEEE Real-Time Systems Symposium), San Francisco, CA, 1997.

[50] Z. Cui, A. Kanevsky, J. Li, and A. Skjellum, “MPI/RT: Design and Implementation of a Real-Time Message Passing Interface,” presented at International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Las Vegas, NV, 1997.

[51] N. G. Leveson, “Software Engineering: Stretching the Limits of Complexity,” *Communications of the ACM*, vol. 40, pp. 129-131, 1997.

[52] F. P. Brooks, Jr., “No Silver Bullet,” in *Computer*, vol. 20, 1987, pp. 10-19.

[53] V. K. Madisetti and B. A. Curtis, “A Quantitative Methodology for Rapid Prototyping and High-Level Synthesis of Signal Processing Algorithms,” *IEEE Transactions on Signal Processing*, vol. 42, pp. 3188-3208, 1994.

[54] J. P. Singh, J. L. Hennessy, and A. Gupta, “Scaling Parallel Programs for Multiprocessors: Methodology and Examples,” in *Computer*, vol. 26, 1993, pp. 42-50.

[55] D. E. Thomas, J. K. Adams, and H. Schmit, “A Model and Methodology for Hardware-Software Codesign,” in *IEEE Design & Test of Computers*, vol. 10, 1993, pp. 6-15.

[56] A. Kalavade and E. A. Lee, “A Hardware-Software Codesign Methodology for DSP Applications,” in *IEEE Design & Test of Computers*, vol. 10, 1993, pp. 16-28.

[57] R. Waxman, “Spec-modeling methodologies...,” Email, January 26, 1998.

[58] K. Roy, “A D&T Roundtable: Hardware-Software Codesign,” in *IEEE Design & Test of Computers*, vol. 14, 1997, pp. 75-83.

[59] D. D. Gajski, F. Vahid, and S. Narayan, “A System-Design Methodology: Executable-Specification Refinement,” presented at European Conference on Design Automation, Paris, France, 1994.

[60] G. De Micheli, “Guest Editor's Introduction: Hardware-Software Codesign,” in *IEEE Micro*, vol. 14, 1994, pp. 8-9.

[61] W. Wolf, “Guest Editor's Introduction: Hardware-Software Codesign,” in *IEEE Design & Test of Computers*, vol. 10, 1993, pp. 5.

[62] M. Willems, V. Bürsgens, T. Grötker, and H. Meyr, “FRIDGE: An Interactive Code Generation Environment for HW/SW CoDesign,” presented at 1997 International Conference on Acoustics, Speech, and Signal Processing (ICASSP 97), Munich, Germany, 1997.

[63] C. Kuttner, “Hardware-Software Codesign Using Processor Synthesis,” in *IEEE Design & Test of Computers*, vol. 13, 1996, pp. 43-53.

[64] L. Garber and D. Sims, “In Pursuit of Hardware-Software Codesign,” in *Computer*, vol. 31, 1998, pp. 12-14.

[65] H. De Man, I. Bolsens, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest, “Co-Design of DSP Systems,” in *Hardware/Software Co-Design*, G. D. Micheli and M. Sami, Eds. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1996, pp. 75-104.

[66] B. Lin, S. Vercauteren, and H. De Man, “Embedded Architecture Co-Synthesis and System Integration,” presented at International Workshop on Hardware/Software Codesign, 1996.

[67] D. D. Gajski and F. Vahid, “Specification and Design of Embedded Hardware-Software Systems,” in *IEEE Design & Test of Computers*, vol. 12, 1995, pp. 53-67.

[68] O. Tanir, V. K. Agarwal, and P. C. P. Bhatt, “A Specification-Driven Architectural Design Environment,” in *Computer*, vol. 28, 1995, pp. 26-35.

[69] F. Balarin, M. Chiodo, D. Engels, P. Giusto, W. Gosti, H. Hsieh, A. Jurecska, M. Lajolo, L. Lavagno, C. Passerone, R. Passerone, C. Sansoè, M. Sgroi, E. Sentovich, K. Suzuki, B. Tabbara, R. v. Hanxleden, S. Y. Alberto, and Sangiovanni-Vincentelli, *Polis: A Design Environment for Control-Dominated Embedded Systems (User's Manual)*, 0.3 ed. Berkeley, CA: University of California Berkeley, 1997.

[70] F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli, and K. Suzuki, *Hardware-Software Co-Design of Embedded Systems*, vol. 404. Boston, MA: Kluwer Academic Publishers, 1997.

[71] B. Tabbara, “Design of Embedded Systems,” Email, May 14, 1998.

[72] J. C. Anderson, “Modeling RASSP Benefits,” MIT Lincoln Labs, Lexington, MA, Presentation, 1997.

[73] J. H. M. Malley, “RASSP -- Changing the Paradigm of Electronic System Design,” in *IEEE Design & Test of Computers*, vol. 13, 1996, pp. 23-31.

[74] R. M. Sedmak and J. S. Evans, “Spanning the Product Life Cycle: RASSP DFT,” in *IEEE Design & Test of Computers*, vol. 13, 1996, pp. 32-41.

[75] L.-R. Dung and V. K. Madisetti, “Conceptual Prototyping of Scalable Embedded DSP Systems,” in *IEEE Design & Test of Computers*, vol. 13, 1996, pp. 54-65.

[76] J. A. DeBardelaben, V. K. Madisetti, and A. J. Gadient, “Incorporating Cost Modeling in Embedded-System Design,” in *IEEE Design & Test of Computers*, vol. 14, 1997, pp. 24-35.

[77] S. Swamy, A. Molin, and B. Covnot, “OO-VHDL: Object-Oriented Extensions to VHDL,” in *Computer*, vol. 28, 1995, pp. 18-26.

[78] E. A. Lee, D. G. Messerschmitt, S. Bhattacharyya, and K. White, “The Almagest (v0.7),” Department of Electrical Engineering and Computer Sciences, College of Engineering, University of California at Berkeley, Berkeley, CA, 3 volumes December 1997.

[79] W.-T. Chang, S. Ha, and E. A. Lee, “Heterogeneous Simulation--Mixing Discrete-Event Models with Data Flow,” in *Rapid Prototyping of Application Specific Signal Processors*, vol. 15, *Journal of VLSI Signal Processing*, M. A. Richards, A. J. Gadient, and G. A. Frank, Eds. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1997, pp. 127-144.

[80] G. Galicia, “Description of a Code Generation Tool for the Mercury RACEway,” University of California Berkeley, Berkeley, CA, Web Site, September 30, 1996.

[81] G. Galicia, “Cosynthesis of Control and Dataflow,” University of California Berkeley, Berkeley, CA, Web Site, 1997.

[82] Mercury Computer Systems Inc., “Achieving Productivity, Performance, and Portability in Software Programming,” Mercury Computer Systems Inc., Chelmsford, MA, Multicomputer Technology Brief http://www.mc.com/Technical_bulletins/mtb7darpa/mtb7_main.html, July 10 1997.

[83] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The Statemate Approach*. Andover, MA: iLogix Inc., 1996.

[84] D. Harel and E. Gery, “Executable Object Modeling with Statecharts,” in *Computer*, vol. 30, 1997, pp. 31-42.

[85] The MathWorks Inc., “Accelerated DSP Design,” in *MATLAB News & Notes*, 1996, pp. 4-6.

[86] M. Benincasa, R. Besler, D. Brassaw, and J. Ralph L. Kohler, “Rapid Development of Real-Time Systems Using RTExpress(TM),” presented at First Merged Symposium IPPS/SPDP 1998 12th International Parallel Processing Symposium & 9th Symposium on Parallel and Distributed Processing, Orlando, FL, 1998.

[87] “One of our submarines...,” in *Computer Design*, vol. 35, 1996.

[88] ORINCON Technologies Inc., *RIPPEN Tools Reference*, Version 3.3 ed. San Diego, CA: ORINCON Technologies, 1999.

[89] ORINCON Technologies Inc., *RIPPEN User's Guide*, Version 3.3 ed. San Diego, CA: ORINCON Technologies, 1999.

[90] ORINCON Technologies Inc., *RIPPEN Tool Developer's Guide*, Version 3.3 ed. San Diego, CA: ORINCON Technologies, 1999.

[91] B. Schaming, “GEDAE: A Graphical Programming and Autocode Generation Tool for Signal Processing Applications,” Lockheed Martin Advanced Technologies Laboratory, Camden, NJ, Technical Paper, 1997.

[92] R. M. Karp and R. E. Miller, “Properties of a Model for Parallel Computations: Determinacy, Termination, Queuing,” *SIAM Journal of Applied Mathematics*, vol. 14, pp. 1390-1411, 1966.

[93] Naval Research Laboratory, *Processing Graph Method Tutorial*, 8 January 1990 ed. Arlington, VA: Department of the Navy, 1990.

[94] S. Baruah, S. Goddard, and K. Jeffay, “Feasibility Concerns in PGM Graphs With Bounded Buffers,” presented at Third IEEE International Conference on Engineering of Complex Computer Systems, Como, Italy, 1997.

[95] S. Goddard and K. Jeffay, “Distributed Real-Time Dataflow: An Execution Paradigm for Image Processing and Anti-Submarine Warfare Applications,” presented at The 17th IEEE Real-Time Systems Symposium (RTSS '96), Washington, DC, 1996.

[96] S. Goddard and K. Jeffay, “Analyzing the Real-Time Properties of a Dataflow Execution Paradigm using a Synthetic Aperture Radar Application,” University of North Carolina, Chapel Hill, NC, Technical Report TR97-007, April 1997.

[97] S. Goddard and K. Jeffay, “A Software Synthesis Method for Building Real-Time Systems from Processing Graphs,” University of North Carolina, Chapel Hill, NC, Technical Report TR98-002, January 1998.

[98] Management Communications and Control Inc., *Using the MCCI Autocoding Toolset*, Draft version 0.5 ed. Arlington, VA: Management Communications and Control, Inc., 1997.

[99] Management Communications and Control Inc., *MCCI Autocoding Toolset Tutorial*, 0.5 ed. Arlington, Virginia: Management Communications and Control, Inc., 1999.

[100] Management Communications and Control Inc., *Domain Primitive Descriptions*, 1.0a ed. Arlington, Virginia: Management Communications and Control, Inc., 1999.

[101] Mercury Computer Systems Inc., “Application Configuration Language Dictionary,” http://www.mc.com/talaris_fold/talaris/lrm/index.html, Feb. 9, 1997.

[102] M. Krueger, “A Development Tool Environment for Configuration, Build, and Launch of Complex Applications,” presented at 3rd International Workshop on Embedded HPC Systems and Applications (EHPC'98; at the First Merged Symposium IPPS/SPDP 1998), Orlando, FL, 1998.

[103] A. Clouard, “Overview of CapCASE: A Toolset Enabling Visual Automatic Source Generation for Parallel Computing Systems,” Matra Cap Systèmes, Velizy, France, White Paper v1.2-Q196-AC, Q196 1993-96.

[104] A. Clouard, A. Pool, P. Tessier, O. Debon, and J. Kulp, “CapCASE: A Graphical Development Tool Supporting Scalable, Heterogeneous Multicomputers,” presented at International Conference on Signal Processing Applications & Technology, Boston, MA, 1996.

[105] J. Salasin, “ECBS in Concept Analysis and Organizational Transformation,” presented at 1999 IEEE Conference and Workshop on Engineering of Computer-Based Systems (ECBS '99), Nashville, TN, 1999.

[106] V. Berman, “Candidate Systems Description Notations,” SLDL Committee of the EDA Industry Council [Online], 11/10/98, 1998. Available HTTP: http://www.inmet.com/SLDL/notations.html.

[107] D. Barton, “Minutes of the FDL SLDL Workshop and Meeting,” SLDL Committee of the EDA Industry Council [Online], 11/30/98, 1998. Available HTTP: http://www.inmet.com/SLDL/fdl98/fdl98.html.

[108] R. Jagannathan, C. Dodd, and I. Agi, “GLU: A High-Level System for Granular Data-Parallel Programming,” *Concurrency: Practice and Experience*, vol. 9, pp. 63-83, 1997.

[109] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*, vol. 360. Boston: Kluwer Academic Publishers, 1996.

[110] B. Tuck, “SOC design: Hardware/software codesign or a Java-based approach?,” in *Computer Design*, vol. 37, 1998, pp. 22-26.

[111] R. Goering, “EDA startup seeks to redefine design,” in *EETimes*, 1998, Reprint.

[112] C. Ussery, “System-on-a-chip era requires rethinking design approaches,” in *Computer Design*, vol. 37, 1998, pp. 94-96.

[113] P. Baraona, J. Penix, and P. Alexander, “VSPEC: A Declarative Requirements Specification Language for VHDL,” in *High-Level System Modeling: Specification Languages, Part 1*, vol. 3, *Current Issues in Electronic Modeling*, J.-M. Bergè, O. Levia, and J. Rouillard, Eds. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1995, pp. 51-75.

[114] P. Baraona and P. Alexander, “Abstract Architecture Representation Using VSPEC,” *VLSI Design*, vol. 9, pp. 181-202, 1999.

[115] P. Baraona and P. Alexander, “VSPEC: A Language for Digital System Specification,” presented at AI Models for Systems Engineering Workshop at AAAI-94 Conference, Seattle, WA, 1994.

[116] A. Finkelstein and I. Sommerville, “The Viewpoints FAQ,” *Software Engineering Journal*, vol. 11, pp. 2-4, 1996.

[117] B. Nuseibeh, J. Kramer, and A. Finkelstein, “A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification,” *IEEE Transactions on Software Engineering*, vol. 20, pp. 760-773, 1994.

[118] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, “Inconsistency Handling In Multi-Perspective Specifications,” *IEEE Transactions on Software Engineering*, vol. 20, pp. 569-578, 1994.

[119] P. Alexander, “Re: FDL SLDL Workshop comments on "super-language",” Email, 12/31/98, 1998.

[120] H. Davis, S. R. Goldschmidt, and J. Hennessy, “Tango: A Multiprocessor Simulation and Tracing System,” Stanford University, Stanford, CA, Technical Report CSL-TR-90-439, July 1990.

[121] RASSP Taxonomy Working Group (RTWG), “RASSP VHDL Modeling Terminology and Taxonomy - Revision 2.4,” Revision 2.4, February 23, 1998.

[122] W. Ecker, M. Hofmeister, and S. März-Rössel, “The Design Cube: A Model for VHDL Designflow Representation and its Application,” in *High-Level System Modeling: Specification and Design Methodologies*, vol. 4, *Current Issues in Electronic Modeling*, R. Waxman, Ed., 1 ed. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1996, pp. 83-128.

[123] C. Jones, “Software Metrics: Good, Bad, and Missing,” in *Computer*, vol. 27, 1994, pp. 98-100.

[124] T. C. Jones, “The Economics of Object-Oriented Software,” Software Productivity Research, Inc., Burlington, MA, Report, April 14, 1997.

[125] Software Technology Support Center, “Chapter 8: Measurement and Metrics,” in *Guidelines for Successful Acquisition and Management of Software-Intensive Systems*, vol. 1, Version 2.0 ed. Hill AFB, UT: Department of the Air Force, 1996, pp. 8-1 -- 8-64.

[126] C. F. Kemerer, “Reliability of Function Points Measurement: A Field Experiment,” *Communications of the ACM*, vol. 36, pp. 85-97, 1993.

[127] D. Longstreet, “How Are Function Points Useful?,” *American Programmer (now the "Cutter IT Journal")*, vol. 8, pp. 25-32, 1995.

[128] T. C. Jones, *Estimating Software Costs*. New York, NY: McGraw-Hill, 1998.

[129] M. D. Fraser, K. Kumar, and V. K. Vaishnavi, “Strategies for incorporating formal specifications in software development,” *Communications of the ACM*, vol. 37, pp. 74-86, 1994.

[130] J. P. Bowen and M. G. Hinchey, “Ten Commandments of Formal Methods,” in *Computer*, vol. 28, 1995, pp. 56-63.

[131] D. D. Gajski and R. H. Kuhn, “Guest Editors' Introduction: New VLSI Tools,” in *Computer*, vol. 16, 1983, pp. 11-14.

[132] M. C. McFarland, A. E. Parker, and R. Camposano, “The High-Level Synthesis of Digital Systems,” *Proceedings of the IEEE*, vol. 78, pp. 301-318, 1990.

[133] A. M. Davis, “A comparison of techniques for the specification of external system behavior,” *Communications of the ACM*, vol. 31, pp. 1098-1115, 1988.

[134] D. H. H. Yoon, J. Rozenblit, T. Ewing, and S. Schulz, “A Survey of System Design Methodologies,” presented at 1997 Workshop on Engineering of Computer-Based Systems (ECBS '97), Monterey, CA, 1997.

[135] S. Kumar, J. H. Aylor, B. W. Johnson, and W. A. Wulf, *The Codesign of Embedded Systems: A Unified Hardware/Software Representation*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1996.

[136] F. Rose, J. Shackleton, and C. Hein, "Performance Modeling of System Architectures," *Journal of VLSI Signal Processing*, vol. 15, pp. 97-109, 1997.

[137] T. Steeves, F. Rose, T. Carpenter, J. Shackleton, and O. v. d. Hoff, "Evaluating Distributed Multiprocessor Designs," presented at Second Annual RASSP Conference, Arlington, Virginia, 1995.

[138] M. Meyassed, R. McGraw, J. Aylor, R. Klenke, R. Williams, F. Rose, and J. Shackleton, "A Framework for the Development of Hybrid Models," presented at Second Annual RASSP Conference, Arlington, Virginia, 1995.

[139] I. The MathWorks, *Using MATLAB*, Version 5.1 ed. Natick, MA: The MathWorks, Inc., 1997.

[140] D. Hanselman and B. Littlefield, *Mastering Matlab 5: A Comprehensive Tutorial and Reference*. Upper Saddle River, NJ: Prentice Hall, 1998.

[141] I. The MathWorks, *Using SIMULINK*, Version 2 ed. Natick, MA: The MathWorks, Inc., 1997.

[142] J. B. Dabney and T. L. Harman, *Mastering Simulink 2*. Upper Saddle River, NJ: Prentice Hall, 1998.

[143] I. The MathWorks, *DSP Blockset User's Guide (version 3)*, Version 3 ed. Natick, MA: The MathWorks, Inc., 1999.

[144] I. The MathWorks, *Stateflow User's Guide*, Version 1 ed. Natick, MA: The MathWorks, Inc., 1997.

[145] I. The MathWorks, *Real-Time Workshop User's Guide*, Version 2 ed. Natick, MA: The MathWorks, Inc., 1997.

[146] Relex Software Corporation, *Relex 7 Tutorial Manual*. Greensburg, PA: Relex Software Corporation, 1999.

[147] I. Reliability Center, "What is PROACT® Software?," Reliability Center, Inc. [Online], October 20, 1999. Available HTTP: http://www.reliability.com/proact.htm.

[148] Espinoza Consulting, "The Reliability & Maintenance Analyst," Espinoza Consulting [Online], October 21, 1999. Available HTTP: http://www.mich.com/~espinoza/rma_main.htm.

[149] I. The MathWorks, *Excel Link User's Guide*, Version 1.0.8 (Release 11) ed. Natick, MA: The MathWorks, Inc., 1999.

[150] Viewlogic Systems Inc., *eArchitect User Manual*. Marlboro, MA: Viewlogic Systems Inc., 1999.

[151] Viewlogic Systems Inc., *eArchitect Reference Manual*. Marlboro, MA: Viewlogic Systems Inc., 1999.

[152] Viewlogic Systems Inc., *eArchitect Training Manual*. Marlboro, MA: Viewlogic Systems Inc., 1999.

[153] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos, “PVM and MPI: A Comparison of Features,” *Calculateurs Paralleles*, vol. 8, 1996.

[154] J. Flower and A. Kolawa, “Express is not just a Message Passing System Current and Future Directions in Express,” *Parallel Computing*, vol. 20, pp. 597-614, 1994.

[155] W. Saphir, “Tutorial: A Comparison of NX, CMMD, PVM and MPI,” presented at Supercomputing 94, Manchester, England, 1994.

[156] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference*. Cambridge, MA: The MIT Press, 1997.

[157] I. The MathWorks, *DSP Blockset User's Guide*, Version 2 ed. Natick, MA: The MathWorks, Inc., 1997.

[158] B. Zuerndorfer and G. A. Shaw, “SAR Processing for RASSP Application,” presented at First Annual RASSP Conference, Arlington, Virginia, 1994.

[159] B. W. Zuerndorfer, J. C. Anderson, R. A. Ford, A. H. Anderson, G. A. Rocco, and G. A. Shaw, “RASSP Benchmark 1 Technical Description,” MIT Lincoln Laboratory, Lexington, MA, Project Report ESC-TR-94-113, December 13, 1994.

[160] RASSP, “Proceedings of the 1st RASSP Conference,” presented at First Annual RASSP Conference, Arlington, Virginia, 1994.

[161] RASSP, “Proceedings of the 2nd RASSP Conference,” presented at Second Annual RASSP Conference, Arlington, Virginia, 1995.

[162] J. C. Anderson, “Projecting RASSP Benefits,” presented at Second Annual RASSP Conference, Arlington, Virginia, 1995.

[163] G. A. Shaw, “RASSP Benchmark Program Overview,” presented at First Annual RASSP Conference, Arlington, Virginia, 1994.

[164] G. A. Shaw, “RASSP SAR benchmark data package,” Email, January 22, 1999.

[165] A. H. Anderson, G. A. Shaw, and C. T. Sung, “VHDL Executable Requirements,” presented at 1st Annual RASSP Conference, Washington, DC, 1994.

[166] H. Zebrowitz, “Matlab model of RASSP SAR BM-2,” Email, 3/4/99, 1999.

[167] A. H. Anderson, "Scalable C and VHDL Simulators for a SAR Image Processor," MIT Lincoln Laboratory, Lexington, MA, User Guide, June 8, 1998.

[168] G. A. Shaw and A. H. Anderson, "Executable Requirements: Opportunities and Impediments," presented at 1996 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP-96), Atlanta, GA, 1996.

[169] A. H. Anderson and G. A. Shaw, "Executable Requirements and Specifications," *Journal of VLSI Signal Processing*, vol. 15, pp. 49-61, 1997.

[170] Elma Electronic, "Enclosure Products: Desktop VME On Line Product Catalog," Elma Electronic [Online], October 23, 1999. Available HTTP: http://www.elma.com/enclosure/vme/olc-desk.html.

[171] I. Mercury Computer Systems, "The RACE Multicomputer," Mercury Computer Systems, Inc.,, Chelmsford, MA, Hardware Theory of Operation: Processors, I/O Interface and the RACEway Interconnect Version 1.3, November 3, 1995.

[172] VSIPL Forum, "VSIPL Core Profile," Draft v1.0, January 6, 1999.

[173] VSIPL Forum, "VSIPL Core Lite Profile," Draft v1.0, January 6, 1999.