INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600
MODEL-BASED CODESIGN FOR
REAL-TIME EMBEDDED SYSTEMS

by

Stephan Schuiz

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA
2001
As members of the Final Examination Committee, we certify that we have read the dissertation prepared by Stephan Schulz entitled Model-Based Codesign for Real-Time Embedded Systems and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Jerzy W. Rozenblit  
Bernard P. Zeigler  
Francois E. Cellier  
A. Larry Wright

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College. I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director  Jerzy W. Rozenblit
STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: 

[Signature]
ACKNOWLEDGMENTS

First and foremost I want to thank my girl-friend Karolina for the support she has given to me over the entire course of my doctoral studies. My gratitude also goes out to my family, especially my parents Brigitte and Eckehard, for their continued encouragement over the years.

I would like to thank my mentor Professor Jerzy W. Rozenblit for introducing me to codesign back in 1995, his encouragement to pursue of this degree, and providing me with a large amount of freedom in the location of my studies. I appreciate the time and efforts of my committee members Professor Bernard P. Zeigler, Professor François E. Cellier, Professor A. Larry Wright, and Professor Richard D. Schlichting.

Financial support for numerous years of study and research has been provided by Infineon Technologies (formerly Siemens AG), München, Germany. Here, I am most indebted to Dr. Michael Mrva and Dr. Klaus J. Buchenrieder for their vision and belief in the importance of this project. In addition, I want to thank the other members of the former high-level design research group Alexander Sedlmeier, Dr. Rainer Kress, and Andreas Pyttel for providing me with excellent design experience, feedback, and a great working environment during many summers and especially in my last year of research.

This work could not have been completed without the support and input from the members of the codesign group in our Model-based Design Laboratory Steve J. Cunning, Tony C. Ewing, John T. Olson, Sanjaya K. Wijeratne, Pallav Gupta, and Pat Sandfort.

I sincerely appreciate the efforts of Joe Hanson at Altera Corporation, Richard Kilgore at ThreadTec, and Heinz Fischer at TQ Components for supporting me with state-of-the-art development environments, simulation software, and prototyping boards at no cost. Without their support this research would have never gained the momentum and viability that it has now.

Finally there is a number of people and places that have inspired and contributed to the research and writing of this dissertation in one way or another. I attached a compilation of names which at least deserve an honorable mention in Appendix C. I thank everyone for their help, contributions, and friendship.
TABLE OF CONTENTS

TABLE OF FIGURES .................................................................................................................. 15

LIST OF TABLES ...................................................................................................................... 19

ABSTRACT .................................................................................................................................. 20

1. INTRODUCTION .................................................................................................................. 22

1.1. MOTIVATION .................................................................................................................... 23

1.2. MODELING OF COMPUTING SYSTEMS ........................................................................... 24

1.2.1. A Brief History of Modeling ...................................................................................... 25

1.2.2. Levels of Abstraction in the Specification of Computing Systems ......................... 26

1.2.3. The System Level ........................................................................................................ 28

1.2.4. Modeling Specifications ............................................................................................. 29

1.2.5. Modeling in Computing Systems Design ................................................................... 29

1.3. HARDWARE/SOFTWARE CODESIGN ........................................................................... 30

1.4. EMBEDDED SYSTEMS .................................................................................................... 32

1.4.1. Embedded Systems Hardware .................................................................................. 32

1.4.2. Embedded Systems Software ................................................................................... 33

1.4.3. Future Outlook in Embedded Systems Applications .................................................. 34

1.5. CONTRIBUTIONS ............................................................................................................. 35

1.6. DISSERTATION OVERVIEW ......................................................................................... 37

2. RELATED RESEARCH ........................................................................................................ 39
TABLE OF CONTENTS - Continued


2.2. Work in Hardware/Software Codesign ........................................ 41

   2.2.1. Early Approaches to Codesign ........................................... 42

   2.2.2. Shortcomings of Implementation Level Codesign ....................... 43

   2.2.3. System Level Codesign ..................................................... 45

2.3. Codesign Environments ........................................................... 47

   2.3.1. Polis ................................................................. 48

   2.3.2. COSYMA ............................................................. 49

   2.3.3. Virtual Codesign Composer ............................................. 49

   2.3.4. DEVS/DOC ........................................................... 51

   2.3.5. SONORA ........................................................... 52

2.4. Summary .............................................................................. 54

3. Model-Based Codesign ............................................................... 56

   3.1. Conventional Design of Embedded Systems .................................. 56

   3.2. System Modeling for the Design of Computing Systems ................. 57

   3.3. The Model-Based Codesign Methodology ..................................... 58

   3.4. Steps in the Methodology ..................................................... 60

      3.4.1. System Specification .................................................... 60

      3.4.2. Modeling ............................................................... 61

      3.4.3. Design Model Refinement ............................................. 62

      3.4.4. Design Model Analysis and Evaluation ............................... 63
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.5. Design Model Transformation</td>
<td>64</td>
</tr>
<tr>
<td>4. MODEL-BASED EMBEDDED SYSTEMS DESIGN</td>
<td>65</td>
</tr>
<tr>
<td>4.1. MODEL-BASED DESIGN AND CODESIGN</td>
<td>65</td>
</tr>
<tr>
<td>4.2. ESSENTIAL INGREDIENTS FOR A POWERFUL DESIGN SPECIFICATION</td>
<td>66</td>
</tr>
<tr>
<td>4.2.1. Concurrency</td>
<td>66</td>
</tr>
<tr>
<td>4.2.2. Hierarchy</td>
<td>67</td>
</tr>
<tr>
<td>4.2.3. Determinism</td>
<td>68</td>
</tr>
<tr>
<td>4.2.4. Modularity</td>
<td>68</td>
</tr>
<tr>
<td>4.3. REAL-TIME EMBEDDED SYSTEMS</td>
<td>69</td>
</tr>
<tr>
<td>4.4. THE MODEL-BASED DESIGN PROCESS</td>
<td>69</td>
</tr>
<tr>
<td>4.4.1. The Construction of Design Models</td>
<td>70</td>
</tr>
<tr>
<td>4.4.2. Validation of Design Models</td>
<td>71</td>
</tr>
<tr>
<td>4.4.3. Design Model Development</td>
<td>72</td>
</tr>
<tr>
<td>4.5. MODELING OF EMBEDDED Computing SYSTEMS BEHAVIOR</td>
<td>72</td>
</tr>
<tr>
<td>4.5.1. The Behavior of Embedded Computing Systems</td>
<td>73</td>
</tr>
<tr>
<td>4.5.2. The Basic Model Component Specification</td>
<td>74</td>
</tr>
<tr>
<td>4.5.3. Specification of Design Models for Complex Computations</td>
<td>76</td>
</tr>
<tr>
<td>4.6. DERIVATION OF REQUIRED EXECUTION TIMES FOR COMPUTATIONS</td>
<td>78</td>
</tr>
<tr>
<td>4.6.1. Unknown Required Execution Times</td>
<td>79</td>
</tr>
<tr>
<td>4.6.2. A Measure for Computational Complexity</td>
<td>80</td>
</tr>
<tr>
<td>4.6.3. The Computation of Model State Execution Times</td>
<td>82</td>
</tr>
</tbody>
</table>
4.7. MODELING COMPONENT COMMUNICATION AT THE SYSTEM LEVEL .................. 83

4.7.1. The Communication Model Component Specification .......................... 84

4.8. AN EXAMPLE FOR A SYSTEM MODELING AND SIMULATION ENVIRONMENT FOR EMBEDDED SYSTEMS DESIGN .......................................................... 86

4.8.1. DEVS Implementations ........................................................................ 87

4.8.2. Modeling of the Structural Aspect ..................................................... 88

4.8.3. Modeling of the Behavioral Aspect ................................................... 89

4.8.4. The Discrete Event Communication Model ........................................ 91

4.8.5. The Passage of Time .......................................................................... 92

5. DESIGN MODEL REFINEMENT .................................................................. 93

5.1. DESIGN MODEL REFINEMENT IN MODEL-BASED CODESIGN ............... 93

5.2. FORMS OF DESIGN MODEL REFINEMENT ......................................... 94

5.2.1. Communication Refinement ................................................................. 95

5.2.2. Structural Refinement ........................................................................ 95

5.2.3. Functional Refinement ........................................................................ 97

5.2.4. Timing Refinement ............................................................................. 99

5.3. THEORETICAL ASPECTS OF DESIGN MODEL REFINEMENT .................. 100

5.3.1. Internal Data Dependencies and the Internal Event ................................ 101

5.3.2. The Basic Model Component Specification Revisited .......................... 102

5.3.3. Correct Design Model Refinement ..................................................... 105

5.3.4. Types of Resulting Model Component Specifications .......................... 106
TABLE OF CONTENTS - Continued

5.3.4.1. Type I Scenarios ................................................................. 107
5.3.4.2. Type II Scenarios ................................................................. 111
5.3.4.3. Type III Scenarios ................................................................. 114
5.3.5. Adjustment of Communication Model Components .................. 115
5.3.6. Execution Time Violations in Modified Original Model Specifications .... 116
5.3.7. Correct Refinement of More Complex Behaviors ....................... 118

6. DESIGN MODEL ANALYSIS ............................................................... 120

6.1. DESIGN MODEL EVALUATION ......................................................... 120
6.1.1. Simulation versus Formal Verification ....................................... 121
6.1.2. Validation in Model-Based Codesign ....................................... 122

6.2. DESIGN MODEL ANALYSIS METHODS ............................................. 123
6.2.1. Functional Analysis ............................................................... 123
6.2.2. Performance Analysis ............................................................. 124

6.3. ANALYSIS METHODS FOR DESIGN MODEL REFINEMENT ................ 125
6.3.1. Static Model Component Data Analysis ..................................... 126
6.3.2. Static Model Component Behavioral Analysis .............................. 127
6.3.3. Model Component Communication Analysis .................................. 129
6.3.4. Composed Design Model Communication Analysis ....................... 130
6.3.5. Model Component Cycle Time Analysis ..................................... 132

6.4. DESIGN MODEL PERFORMANCE ANALYSIS METHODS .................... 134
6.4.1. Model Component Utilization Analysis ...................................... 135
TABLE OF CONTENTS - Continued

6.4.2. Composed Model Utilization Analysis .................................................. 136

6.5. Automation of the Design Model Refinement and Analysis ....................... 137

7. Model Compilation ....................................................................................... 139

7.1. The Generation of Processing Platforms for Design Specifications .... 139

7.1.1. Code Generators for Design Model Specifications ......................... 140

7.1.2. The Model Continuity Problem ............................................................ 140

7.1.3. What Constitutes a Good Target Architecture? .............................. 141

7.2. Elements of the Processing Platform ....................................................... 143

7.2.1. Processing Elements ................................................................. 143

7.2.2. Virtual Processors ...................................................................... 144

7.2.3. Hardware Elements ...................................................................... 145

7.2.4. Communication Elements ............................................................ 146

7.3. The Processing Platforms and Modular Modeling Specifications .... 148

7.3.1. The Processing Platform Specified by the Design Model Specification .......................... 148

7.3.2. The Ideal Processing Platform ......................................................... 149

7.3.3. Back in the Real World ................................................................. 150

7.3.4. A Second Level of Structural Hierarchy ........................................ 151

7.4. Model Compilation as a Search Problem ............................................ 152

7.4.1. The Reduction of the Search Space ................................................. 153

7.4.2. The Model Compilation Algorithm .................................................. 153

7.5. Model Compilation Analysis ............................................................... 156
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.5.1</td>
<td>Model Component Specification Assessment</td>
<td>157</td>
</tr>
<tr>
<td>7.5.2</td>
<td>Estimation of Effective Model Component Cycle Times</td>
<td>158</td>
</tr>
<tr>
<td>7.5.3</td>
<td>Elimination of Invalid Dedicated Processor Mappings</td>
<td>159</td>
</tr>
<tr>
<td>7.6</td>
<td>Design Model Adaptation</td>
<td>159</td>
</tr>
<tr>
<td>7.7</td>
<td>Scheduling Analysis</td>
<td>160</td>
</tr>
<tr>
<td>7.7.1</td>
<td>Benefits of RTOS-based Software Implementations</td>
<td>160</td>
</tr>
<tr>
<td>7.7.2</td>
<td>Shared Processor Implementations and Concurrent Design Specifications</td>
<td>161</td>
</tr>
<tr>
<td>7.7.3</td>
<td>Scheduling Analysis for Multi-Component Mappings</td>
<td>162</td>
</tr>
<tr>
<td>7.7.4</td>
<td>Resolving Scheduling Conflicts for Shared Processor Implementations</td>
<td>163</td>
</tr>
<tr>
<td>7.7.5</td>
<td>An Algorithm For Rescheduling Conflicting Components</td>
<td>166</td>
</tr>
<tr>
<td>7.7.6</td>
<td>Scheduling Analysis with Processing States Sequences</td>
<td>168</td>
</tr>
<tr>
<td>7.8</td>
<td>Processor Communication Analysis</td>
<td>168</td>
</tr>
<tr>
<td>7.8.1</td>
<td>Bus-based Processor Communication</td>
<td>169</td>
</tr>
<tr>
<td>7.8.2</td>
<td>Direct Processor Communication</td>
<td>170</td>
</tr>
<tr>
<td>7.8.3</td>
<td>Hardware Interfaces</td>
<td>172</td>
</tr>
<tr>
<td>8</td>
<td>Implementation From The Behavioral Model Aspect</td>
<td>174</td>
</tr>
<tr>
<td>8.1</td>
<td>The Transformation Of the Behavioral Aspect</td>
<td>174</td>
</tr>
<tr>
<td>8.1.1</td>
<td>The Execution Of the Behavior in Model Components</td>
<td>174</td>
</tr>
<tr>
<td>8.1.2</td>
<td>The Transformation Of Model Component Specifications</td>
<td>175</td>
</tr>
<tr>
<td>8.1.3</td>
<td>A Formal Discussion Of Model Component Transformation</td>
<td>176</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS - Continued

8.1.4. Mapping of DEVS Atomic Models to Timed CSP Processes .................. 177
8.1.5. Example Mapping of a DEVS Atomic Model ..................................... 180

8.2. Implementation of Model Behavior in Software ..................................... 181

8.2.1. Software Implementations .......................................................... 181
8.2.2. Construction of RTOS processes .................................................. 182

8.3. Implementation of Model Behavior in Hardware ................................. 183

8.3.1. Guidelines for possible Hardware Implementations .......................... 185

8.4. Implementation of Communication Interfaces ..................................... 186

8.5. Verification at the next Level of Abstraction ..................................... 187

9. Testing in Model-Based CoDesign ...................................................... 189

9.1. Design and Test ................................................................................. 189

9.1.2. Model Testing in System Modeling .............................................. 191

9.2. Multi-Level Testing ........................................................................... 191

9.2.1. Design Model Testing ................................................................. 193
9.2.2. Integrated System Testing ............................................................ 194

9.3. A Real-Time System Testing Environment ......................................... 195

9.4. The Implementation of the System Testing Environment ..................... 196

9.4.1. STE Test Processes ........................................................................ 198
9.4.2. Conversion of Test Modules into Test Processes .............................. 199
9.4.3. Communication Protocol .............................................................. 200
TABLE OF CONTENTS - Continued

9.5. AN INTEGRATED SYSTEM TESTING EXAMPLE .............................................. 200

9.5.1. AICC Test Modules .................................................................................... 201

9.5.2. The Conversion of the AICC Test Modules into Test processes ............... 203

9.5.3. System Testing Results for Different Hardware/Software Implementations 204

9.5.4. Analysis of Test Data over Time ................................................................. 205

10. THE APPLICATION OF MODEL-BASED CODESIGN CONCEPTS ............ 208

10.1. DESIGN MODEL CONSTRUCTION AND REFINEMENT FROM A SYSTEM

   Specification ..................................................................................................... 208

10.1.1. The Alarm Clock System Specification ..................................................... 209

10.1.2. Abstract Design Model Construction for the Alarm Clock ................. 210

10.1.3. Correct Refinement of the Alarm Clock Design Model ....................... 211

10.1.4. The Shared Resource Problem ................................................................ 215

10.1.5. Implementation and Assessment of Design Model Specifications ........ 216

10.2. THE FINAL DESIGN MODEL SPECIFICATION ......................................... 217

10.2.1. The Ideal Processing Platform ................................................................. 219

10.2.2. Model Compilation Analysis and Design Model Adaptation ............... 220

10.3. MODEL COMPIILATION OF THE ALARM CLOCK ................................... 221

10.4. IMPLEMENTATION ...................................................................................... 223

11. CONCLUSIONS .............................................................................................. 225

11.1. SUMMARY OF RESULTS ............................................................................. 225
# TABLE OF CONTENTS - Continued


11.3. Design and Test ...................................................................... 229

11.4. Future Work ........................................................................ 229

APPENDIX A: PDEVS IMPLEMENTATION OF THE ALARM CLOCK MODEL . 232

APPENDIX B: PDEVS IMPLEMENTATION OF GENERIC COMMUNICATION COMPONENT .................................................................................................................. 245

APPENDIX C: EXTRA CREDITS ......................................................................................... 249

REFERENCES .................................................................................................................. 251
# TABLE OF FIGURES

- **Figure 1. Levels of Abstraction** ................................................................. 26
- **Figure 2. Abstract Sonora Design Flow** .................................................... 53
- **Figure 3. Conventional Approach to Embedded Systems Design** ............... 57
- **Figure 4. Model-Based CoDesign** ............................................................. 59
- **Figure 5. Basic Model Component Specification** ......................................... 74
- **Figure 6. Compact and More Explicit Representation of the Waiting State** .... 76
- **Figure 7. Single Model Component Specification of Complex Behavior** ........ 77
- **Figure 8. Coupled Model Specification of Complex Behavior** ..................... 78
- **Figure 9. Simplified Behavior of a Communication Model Component** ....... 84
- **Figure 10. Two Representations of a Coupled Design Model Specification** ...... 88
- **Figure 11. Revised Diagram of a Basic Model Component Specification** ....... 103
- **Figure 12. Original Behavior and Result from Behavioral Refinement in Type I Scenario** ................................................................. 108
- **Figure 13. Results from Structural Refinement Based on Data Dependency in Type I Scenario** ................................................................. 109
- **Figure 14. General and Special Case of Type II Scenario Results after a Behavioral Refinement** ................................................................. 112
- **Figure 15. Results from Structural Refinement Based on Data Dependencies in Type II Scenario** ................................................................. 113
- **Figure 16. General and Special Case of Type III Scenario Results after a Behavioral Refinement** ................................................................. 114
TABLE OF FIGURES - Continued

Figure 17. Results from Structural Refinement based on Data Dependencies in Type III Scenario ................................................................. 115

Figure 18. Static versus Dynamic Model Information ........................................ 120

Figure 19. Algorithm for Computation of Required Data Transferal ............ 128

Figure 20. Example Model State Execution Trace of a Design Model Component ........................................................................................................ 131

Figure 21. Example Execution Trace for a Model Specification with a Processing State Sequence ................................................................. 133

Figure 22. Pseudo Code for a Possible Search Algorithm Implementation ...... 154

Figure 23. Model State Execution Trace of Schedulable Model Specifications ........................................................................................................... 162

Figure 24. Execution Trace with a Scheduling Conflict during Time Interval (T_{P_{B_{i}}}, T_{W_{C_{x}}}-1) ................................................................................................................ 163

Figure 25. Example Simulation Trace after Rescheduling ............................ 165

Figure 26. Example Rescheduling Algorithm for Components with Scheduling Conflicts ........................................................................................................ 167

Figure 27. Example Communication Window (T_{WAT_{i+1}}, T_{P_{B_{i+1}}}) in Simulation Trace ........................................................................................................ 170

Figure 28. Generic Execution Trace for Model Components with a Processor and Hardware Mapping ................................................................. 172

Figure 29. Formal Mapping of DEVS to Timed CSP ........................................ 178
TABLE OF FIGURES - Continued

FIGURE 30. DEVS AND TIMED CSP SPECIFICATIONS FOR EXAMPLE APPLICATION "DEVICE"
...................................................................................................................................................... 180

FIGURE 31. SOFTWARE IMPLEMENTATION OF EXAMPLE APPLICATION "DEVICE" .......... 183

FIGURE 32. VHDL SPECIFICATION OF APPLICATION EXAMPLE "DEVICE" ................. 186

FIGURE 33. MULTI-LEVEL CODESIGN .............................................................................. 192

FIGURE 34. IMPLEMENTATION OF THE STE .................................................................. 197

FIGURE 35. AICC TEST MODULES ............................................................................... 202

FIGURE 36. EXAMPLE MAPPING OF DESIGN AND TEST MODEL COMPONENTS TO AN
IMPLEMENTATION ................................................................................................................. 203

FIGURE 37. MICROCONTROLLER UTILIZATION OVER TIME .................................... 206

FIGURE 38. ABSTRACT INITIAL DESIGN MODEL AND TYPE I INPUT ARRIVAL SCENARIO 210

FIGURE 39. DESIGN MODEL SPECIFICATION AND TRACE AFTER FIRST REFINEMENT STEP
.......................................................................................................................................................... 212

FIGURE 40. DESIGN MODEL SPECIFICATION AFTER SECOND REFINEMENT STEP ...... 213

FIGURE 41. MODEL EXECUTION TRACE FOR WORST CASE SCENARIOS AFTER SECOND
REFINEMENT STEP ....................................................................................................................... 214

FIGURE 42. DESIGN MODEL SPECIFICATION AFTER THIRD REFINEMENT STEP ........ 214

FIGURE 43. ABSTRACT MODEL COMPONENT SPECIFICATION OF LED MANAGER ....... 216

FIGURE 44. DUAL PROCESSING PLATFORM FOR ALARM CLOCK DESIGN ............... 219

FIGURE 45. ALARM CLOCK EXECUTION TRACE WITH SCHEDULING CONFLICT IN (TPKT,

TWMT) ........................................................................................................................................ 221
TABLE OF FIGURES - Continued

FIGURE 46. IMPLEMENTATION OF FINAL DESIGN MODEL SPECIFICATION OF $M_{AC}$............ 224
LIST OF TABLES

Table 1. Sample Data Table for Processing States of a Model Component .... 127
Table 2. Example Characterizations of Some Processing Elements .............. 157
Table 3. All Software Implementation Comparison ..................................... 204
Table 4. Hardware/Software versus All Software Implementation Using the
68HC11 ........................................................................................................ 205
Table 5. Data Table for M_{KT} .................................................................... 218
Table 6. Adjusted Model Component Execution Times .............................. 220
ABSTRACT

This dissertation presents a model-based codesign framework for real-time embedded systems applications. The presented research provides a theoretical modeling foundation for the construction of design models and their implementation. Whereas most current codesign approaches leverage from a complete specification of an application design at the implementation level, a completely modular, implementation independent system level specification is pursued here. Benefits of the approach presented include: a stepwise refinement of abstract design models for complex applications, a larger design space for possible application implementations, a late design partitioning into hardware and software components, the representation of concurrency inherent to the application, as well as an implementations thereof on a parallel processing platform.

A formal abstraction for a general, system level specification of real-time embedded systems is derived which may be used with a variety of executable discrete event modeling specifications. Furthermore, the construction of abstract design models from textual system specifications is discussed based on this abstraction as well as their correct refinement. A set of analysis methods which evaluate system simulation results is introduced to validate and improve abstracted design model performance during each refinement step.

A direct transition from the design model to an efficient implementation is addressed though a model compilation algorithm which validates alternative processing platforms for a detailed design model specification against real-time constraints. In addition, a formal mapping of system model component specifications to implementation
specifications is given. Separately, this *model continuity* problem is also addressed through a definition of a multi-level approach to the testing of integrated application implementation prototypes. The presented model-based codesign concepts are illustrated with an embedded systems application example.
1. INTRODUCTION

In the past century mankind has witnessed an impressive evolution in the miniaturization and available processing power of computing systems. About fifty years ago computers filled entire rooms and information was fed into them using punched cards. Today computers are so small that they can be in essence worn, and information can be communicated without a physical connection to the device. These devices are able to store a lot more information and process it at a much faster speeds than their predecessors. Along with the technological advances the computer has in essence been redefined from an advanced scientific calculator to a device which can be used to access, administrate, process, and evaluate enormous amounts of information in split-seconds.

One result of these advances have been embedded computing devices. Embedded devices are tightly integrated computing systems which are embedded into either a physical or computing environment. These devices interact with the environment which they are embedded in by gathering information from a set of sensors, process this information, and in most cases influence the environment via actuators based on the results of the computation. Contrary to other computing systems embedded systems devices are generally designed to operate only on very basic human user interaction. The operator is mainly regarded as a source of input data. Example applications include cellular phones which access the Internet, electronic notebooks which schedule meetings, or engine control units in a car which optimize ignition or monitor emissions. In some cases the life of human beings may depend on the correct performance of such embedded computing devices, e.g., in safety restraint systems or aircraft flight control systems.
Every year new applications challenge the processing capabilities of embedded computing systems implementations. According to recent surveys approximately ninety percent of all processors are part of an embedded system [1]. At the same time the available time-to-market window, i.e., the time to turn an innovative idea into successful product, is decreasing steadily. Other commonly encountered constraints in the design of embedded systems include the use of commercially available hardware components or reuse of previous developed software components, real-time performance requirements, and the integration of components from other engineering domains in its implementation.

1.1. Motivation

In the beginning of the last decade hardware/software codesign emerged as a research incentive to address a shortcoming of conventional design approaches for embedded systems, which is to develop hardware and software of an application components in isolation. Most of the current research on codesign [2,3,4,5,6,7,8,9] focuses on integration of components from the different domains in the specification of design implementations. Given further advances in implementation technologies, increases in system complexity, and the demand for high performance embedded systems applications, the need for implementation independent, system level design approaches becomes apparent. In [10,11,12], we have proposed such a system level codesign approach called model-based codesign which uses executable modeling specifications to develop design models.

Executable system level modeling specifications have established themselves over the last years as a tool for the conceptualization and analysis of system specifications for
embedded systems applications [7,13]. Although some system modeling tools offer the
generation of software implementations, application models are rarely used in the actual
implementation of embedded systems. One of the reasons for this reluctant acceptance
are the processing requirements for the generated code, which often exceeds resources of
embedded systems processing platforms. We believe that only a structured approach to a
construction and transformation of design models into prototype implementations can
result in an efficient design implementation.

Although basic concepts and a design flow for model-based codesign have already
been previously published in the context of a model-based codesign methodology
[10,12]. Here, the implementation of design models has so far been largely based on a
combination of system modeling and design experience. In this dissertation we provide a
theoretical foundation for this methodology by the means of a formal abstraction from
which allows the derivation of multi-processor implementations for design models.

1.2. Modeling of Computing Systems

System modeling has been studied extensively to describe, observe, visualize, model,
replicate, predict, or communicate ideas about physical systems, i.e., things we find in
nature. This field of research originally evolved from systems theory which studies the
general description or representation of systems in the form of mathematical models.
System modeling investigates techniques to abstract or model the behavior of physical
systems and to execute resulting system model specifications on computers properly over
time.
1.2.1. A Brief History of Modeling

Since the beginning of recorded history modeling has been used by mankind for the
design and visualization of complex problems. Sketches or miniature scale replicas of
immense buildings can be considered as first architectural models. Maps have
accompanied us for ages as a geographic model of nature. Early models focused mainly
on the physical structure of systems since space can be captured fairly easily in the form
of drawings.

System behavior or the dynamics of physical systems has been much harder to
describe since it is fundamentally based on time. The behavioral aspect of such systems
has been traditionally captured mathematically by a set of differential and difference
equations [15]. The assessment of system behavior, i.e., the evaluation of the
mathematical system model, becomes a tedious task if done by hand since the number of
equations to be evaluated increases quickly with the complexity of the system model.
This limited the behavioral modeling in the early days to systems which exposed only
simple behavior.

With the advent of the computer it became feasible to compute solutions to larger
equation systems and assess the behavior of much more complex systems. Initially
though, computer-based modeling remained limited to research institutions which could
afford to buy fast and expensive computers. Computer-based model execution naturally
depends on the execution speed of the computer which calculates behavior for the system
model. Therefore, a mainstream use of computer-based model execution became only
feasible or realistic in the last decade when even standard personal computers reached the
capability to execute large, complex system models. Today computer modeling is used in many fields of engineering with applications ranging from crash test simulations for vehicles to predictions of the stock market, from our every day weather forecast to the design of computer-based systems.

### 1.2.2. Levels of Abstraction in the Specification of Computing Systems

Models present an abstraction of a real-world system. Abstraction can be used to simplify the structure or behavior of a physical system, e.g., the operation of a computing system. In systems theory, six different levels of abstraction have been identified for the specification of computing systems [7,14]. They illustrate the gradual abstraction of a computing system behavior and can be considered the foundation for our model-based codesign approach for embedded systems.

![Levels of Abstraction](image)

**Figure 1. Levels of Abstraction**

As shown in Figure 1, the first level of abstraction is the electrical or physical level. At this level of abstraction we describe behavior of circuits or computation by the changes in current or voltage in electrical components. The computer can here be merely
considered as a special case of an electrical circuit [15]. It is perceived as a collection of semiconductor devices. Simulation time equals here physical time.

The next level of abstraction describes system behavior at the switch or transistor level. Here we assume that the computer is composed of transistors, i.e., we restrict the behavior of circuits to digital operation. Here, the behavior of an electrical circuit is no longer considered to occur due to changes in current or voltages in electrical elements but has been restricted to transistors switching either "off" or "on", or "0" and "1".

At the logic gate level, the behavior of digital circuits is further abstracted by introducing a set of basic elements which perform Boolean operations, i.e., AND, OR, and NOT gates. Here, simulation time is no longer based on physical time (which may change due to technology advances) but an elementary step or gate delay, i.e., discrete time. The lowest increment of time is defined by the time required for a transistor to switch from an "on" to an "off" state, or vice versa.

The following abstraction level is commonly referred to as the register transfer level (RTL). Functional blocks, and libraries of such are composed of logic gates in different configurations. Registers are introduced as a first concept of memory. From these elements, storage devices or processor architectures can be composed.

The algorithmic level represents an abstraction of computing behavior for software programs. Here, the concept of time becomes a logical one where the smallest time step represents a processor cycle. Behavioral specification at this level of abstraction can express concurrency and causality within a computation. Here, the basic elements for
describing system behavior become a set of instructions for a processing element which are executed in a sequential manner to specify the computation or algorithm.

1.2.3. The System Level

The system level can be considered an abstraction of either the algorithmic or RTL abstraction level. Here, a computing system is perceived as a collection of interacting components which communicate instantaneously.

The most fundamental difference of a device specification at the system level is a change in the concept of time: The interpretation time changes from a logical to a symbolic definition, i.e., simulation time does not necessarily correspond to physical time. This allows for the specification and execution of concurrent system model behavior. Here, discrete event modeling specifications are commonly used to encode digital computing system behavior. System behavior is here described using modeling representations which extend Finite State Machines [16,17,18,19].

This departure from modeling systems based on some elementary behavior, i.e., a transistor or an underlying processor architecture, allows also to broaden the scope to any physical system. As a matter of fact system level modeling specifications can be used to specify and study the behavior of systems which are not part of the computer engineering domain, e.g., the population on planet earth, a transistor, a waterfall, a vending machine, or the engine of a car. A computing system can here be merely considered as one of many physical systems which can be modeled.
1.2.4. Modeling Specifications

In modeling, we specify a system using a variety of modeling representations which may be textual or graphical nature. Graphical representations are here the more desired form of specification since they are easiest to comprehend for humans and - if understood properly - can be used convey complex information better a textual description. This follows the saying "a picture is worth a thousand words". Unfortunately this advantage is also accompanied by the danger of misunderstandings [20]. An unnecessarily complex, incomplete, or ambiguous definition of symbols in a graphical representation can easily lead to misinterpretations of a system specification.

Traditionally modeling representations have been used in computing systems design for a structural or static description of design implementations which attempt to address only part of a system design. A far greater potential resides in the integration of the behavioral aspect, i.e., system dynamics, with the structural aspect in an executable model representation which is more commonly referred to as a modeling specification. A model specification relates structural and behavioral aspects of a system specification with a mathematical formalism. Modeling specifications allow the computer-based execution of system models, i.e., their simulation, and allow the observation of system behavior over time.

1.2.5. Modeling in Computing Systems Design

As we have pointed out in the previous sections, modeling concepts and tools have already established themselves fairly well in computing systems design. In the realization of hardware components computer-aided design tools allow a graphical specification of
the physical implementation structure as well as assessment of its compliance to set of specified design rules. Hardware description languages (such as VHDL [21]) are frequently employed today to virtually prototype or validate the correct execution of hardware components at the RTL level.

In the software domain, modeling is commonly used in object oriented design approaches to describe the implementation of large software programs [22,23,24]. The concept of an object may actually be considered as an abstraction of an algorithmic level system specification. Some object-oriented modeling representations have also been used in system level specifications of software systems.

Many modeling methods for computing systems tend to be fairly domain specific, i.e., they favor either a hardware or software implementation of a specified system behavior. Critical issues like implementations of system design models in a heterogeneous processing environment or the compliance with performance constraints have received little attention at the system level.

1.3. Hardware/Software Codesign

Embedded systems development is commonly based on separate, hardware and software design tracks for complex applications. Such an approach leads to an early partitioning of the design into hardware and software components from the device systems specification and a disjoint development thereof. In such a design approach design components are only integrated in the implementation phase of the application. At this point in the application development the location of errors which occur during the interaction of system components is generally a tedious task and may have a profound impact on the
final release of the product. Failures to fulfill specified performance constraints can require a complete redesign of the application.

Starting in the early 90's a group of researchers started a research incentive called hardware/software codesign which set out to address this problem [2,8,25,26,27]. The driving force behind codesign is its potential for reducing the time-to-market, minimizing size, and maximizing the processing power of systems. Today hardware/software codesign is a well recognized field of research. Codesign is different from the conventional approach in that it continuously relates hardware and software development cycles each other.

During the development of computing systems applications decisions made during hardware design may significantly affect software design activities and vice versa. In codesign, the design problem is treated as a whole. The "co" in codesign here primarily refers to "together". However, it also expresses a design flow that is properly coordinated; a conjoint effort among designers from different areas. The effort is inherently concurrent in that the majority of all design steps are carried out in parallel by a team that guides, coaches, and supervises the design process. Although only few codesign environments utilize parallelism, many experts in the field are confident that exploiting concurrency among design threads and tools will result in significant gains in the development process [28,52].

A number of new computing system specification and simulation environments, soft-prototyping techniques, formal verification methods, high-level synthesis tools, and computer-aided design frameworks have been developed to aid in the design of
heterogeneous systems to this day [2,7,25,29,30,31]. Nevertheless it is our belief that only revised set of design methods can enforce a structured integration of complex, mixed hardware and software system. Numerous authors point to the deficiencies of the traditional codesign frameworks [10,32]. They strongly advocate a process that fosters the integration in one common design perspective. Thus, a unified representation is needed for modeling a system independently of its implementation in hardware or software.

### 1.4. Embedded Systems

Increases in memory densities as well as advances in the manufacturing of embeddable processor cores and sensing technologies have considerably increased the potential for more "intelligent" (i.e., computationally intensive) embedded systems applications. Frequently encountered requirements for embedded systems products include restricted budgets, constraints on the time-to-market, physical limitations, real-time performance constraints, an application implementation based on COTS (components-off-the-shelf [82]), and the integration of various hardware and software components.

#### 1.4.1. Embedded Systems Hardware

Most notably, improvements in transistor packaging and the connection of configurable control logic blocks (CLBs) of Field Programmable Gate Arrays (FPGA) have drastically increased the potential of this particular technology and moved hardware out of its classically rigid implementation domain [33]. Instead of fairly basic combinatorial circuits, full-fledged custom processors, which may even outperform high-end
conventional processors [34], can be specified and simulated using graphical design entry or hardware description languages.

Processors employed in processing platforms of the embedded systems applications are quite diverse and can cover the entire spectrum of processing power. Processor data path widths can vary from 4 bits to 32 bits. Their processing speeds may range today from 2 MHz to 500 MHz. Processors may be sophisticated microprocessors with integrated memory management and floating point units, or mere microcontrollers consisting only of the basic processor core and a basic set of peripherals. Processor instruction sets or architectures frequently vary between manufacturers.

1.4.2. Embedded Systems Software

Software tends to be custom tailored in most applications which makes an efficient but yet general approach to embedded systems design at the implementation level almost impossible. Many attempts to standardize software development have failed in the embedded systems community [35]. Historically, software development of high performance applications has been biased low level assembly programming which provides the most efficient interface to hardware components. Due to significant performance improvements in processor technology, the specification of embedded systems software is however in the midst of a shift towards higher level languages. Severe memory constraints of embedded systems can probably considered the main reason for the limited success of high level programming languages so far.

Another important issue in embedded systems development are real-time processing constraints. Many embedded systems have only a relatively small time window to gather
and evaluate data before the next data package arrives. Here, the inability to represent such constraint in conventional programming languages has been addressed by the use of real-time operating systems. These operating systems can be used to enforce deterministic program execution and allow the specification of processing deadlines. The large number of available RTOS available in the market today reflects again the previously noted spectrum of different processing platforms but also the need for optimizations specific to particular applications.

From the above discussion it is fairly obvious to conclude that software development is of major concern to the embedded systems community. Already today most of the cost and time in the design and development of embedded systems is invested in software development.

1.4.3. Future Outlook in Embedded Systems Applications

What does the future hold for embedded systems application? In the terms of implementation requirements prospects are that further advances in storage, sensing, and processing devices will unlock even greater computational power and result in bigger more complex software implementations [36].

In addition, mobility is expected to become a key issue in the future of embedded systems. The vision is here that applications may migrate between different devices and to roam around computing networks. This requires a change in thinking about rigid implementations of current embedded systems. Flexibility is required from both sides, the processing platform as well as the software elements, of embedded systems applications.
Currently two different approaches have been proposed to assist and enable the
design of these kind of embedded systems. The first approach focus an improvements of
design specifications at the implementation level in order to achieve improvements in
processing performance which will be discussed in the following chapter.

The second approach is to raise the abstraction level in the design specification and to
leave implementation details behind for a moment. Here, a new generation of high level
languages is under development which attempts to integrate system level concepts into
popular programming languages [37].

This dissertation offers an alternative system level design approach which is based on
the development of design model specifications.

1.5. Contributions

Previous publications [10,12,29,38,39,40] have introduced basic concepts as well as an
outline of a development environment for a model-based codesign methodology for
embedded systems design. This approach leverages off the hierarchical and modular
construction of design models based on system modeling concepts. We introduce here a
formal abstraction of embedded systems computing which allows the construction and
refinement of design models for real-time embedded systems applications. By encoding
the models in an executable discrete event modeling specification we can assess the
design model at many levels of abstraction.

An important contribution in many aspects of this work is the introduction of a model
compilation algorithm which validates alternative processor platform configurations for
an efficient design model implementation against specified real-time constraints.
Candidate processing platforms can consist here of possibly multiple, shared general purpose processors or custom hardware components. For the implementation of design model behavior on a selected platform we show a formal mapping from a discrete-event model specification to an implementation level design specification.

The benefits of our approach are several fold:

1. A system level design model specification allows the specification and refinement of concurrent and distributed embedded systems behavior.

2. Implementation independent design descriptions can be specified and validated at the system level using simulation at any level of abstraction. This also fosters component reuse regardless of their technology and allows a late partitioning of the application into design into hardware and software components.

3. The capability to specify abstract design models and a stepwise refinement process simplify the development of complex embedded systems applications.

4. We are able to trade-off design alternatives early on and select a design solution that meets the application system specification and requirements best. Therefore, application development does not require the implementation of - possibly multiple - physical design prototypes.

5. Abstract application prototypes in the form of a design model can be used to identify inconsistencies in the system specification or to communicate design ideas from engineers to customers who may not have the technical background necessary to understand the details of a final design implementation.

6. Design components can be specified at multiple levels of abstraction or detail.
7. The foundation on formal system modeling concepts allows also observation of the design model interaction with systems of other application domains by simply introducing these as separate system models. In addition, the design model can be observed in conditions which may be hard or even impossible to create at the implementation level.

8. The model continuity problem is addressed by providing a structured approach for the transformation of the application system model specification into a mixed hardware/software implementation.

1.6. Dissertation Overview

We begin with a survey of related research in the following chapter which mainly focuses on the engineering of computer-based systems and hardware/software codesign. Chapter 3 provides an overview of model-based codesign methodology and outlines its design phases.

The first part of this dissertation, chapters 4 though 6, introduces the reader to the implementation independent specification and refinement of design models for real-time embedded systems applications. Chapter 4 presents the theoretical foundation for the modular construction of such system level design models using discrete event modeling specifications. Chapter 5 discusses practical as well as theoretical aspects of a design model refinement. Chapter 6 continues with a presentation of analysis techniques for the evaluation of simulation results, i.e., the execution of design models over time.

The second part of the dissertation discusses a structured approach for the transformation of detailed application design model specifications into implementation
level design specifications. Chapter 7 presents model compilation which derives valid processing platforms for a given design model specification based on application implementation constraints. Chapter 8 shows the implementation of design model behavior in the form of hardware and software components for the previously selected processing platform.

Chapter 9 then departs from the design context and illustrates how our model-based design approach can be used for the testing of the integrated application implementations or system prototypes. Chapter 10 illustrates the application of the presented model-based codesign concepts in the context of a simple embedded systems application. The last chapter finally summarizes the achievements of this dissertation and offers some directions for future research.

Since the research presented in this dissertation is heavily based on system modeling concepts, a solid understanding of modeling and simulation theory is required to follow the theoretical sections in the following chapters. Good references for more information on systems modeling are [15,19,38,41].
2. RELATED RESEARCH

In this section we want to present a brief summary of some research which has influenced the development of our model-based codesign methodology. We believe our approach is unique in the sense that the derivation of real-time embedded computing systems implementations has never been presented from systems modeling perspective.

Nevertheless, we want to provide here some background on the fields of research which can be considered as its foundation: We introduce first some relevant research on systems modeling as well as engineering of computer-based systems, and survey then more extensively other proposed codesign approaches. Finally we survey some of the currently available codesign environments.

2.1. Work in System Modeling for Computing Systems Design

Many of the theoretical concepts in this dissertation evolved from research with the Discrete Event System Specification formalism DEVS which was developed by B.P. Zeigler and his research group. DEVS concepts were first published in [18] and have since then been and worked on constantly [19]. DEVS is a system theoretic approach to the modeling of discrete-event systems. The mathematical foundation of DEVS focuses on the change of state variables and the generation of time segments where time intervals between event occurrences may vary. This formalism has been applied to model a wide spectrum of physical systems in various application domains including manufacturing, geography, military, and process control. Probably their contributions on real-time DEVS, a DEVS-based design methodology, and distributed codesign [31] can be considered most relevant to our work. Real-time DEVS constitutes an extension of the
original formalism for real-time simulation. A DEVS-based design methodology has presented in [19]. It focuses though mainly on the formulation of a DEVS definition language which does not address hardware design aspects nor issues related the generation of an implementation.

A big impact on the modeling computer-based applications had the publication of the StateCharts formalism by David Harel [13,16]. StateCharts were designed to provide an intuitive and compact specification of reactive systems behavior using a visual representation. Reactive systems include most control oriented, high performance computing systems. The initial publication of the formalism inspired not only its revision [42] and successful commercial modeling tools, but also lead to a number of StateCharts variants [43]. The Statecharts formalism was later on extended by Harel and others with Activity Charts [13,42] and Module Charts for the system modeling tool STATEMATE [42]. STATEMATE allows for code generation mainly for software prototyping purposes. Publications on this topic though have so far not revealed any formal concepts or details about it. In addition, no support for the generation of mixed hardware/software implementations is offered at this point.

Work on the integration of various systems modeling paradigms is presented by Sztipanovits et al in [44]. Their research is fundamentally based on multi-graph architecture in which concepts of various domain specific modeling paradigms can be specified and represented. It also provides a framework for the conjoint execution of systems which have components specified in different modeling paradigms. The implementation of this architecture is based on a set of model interpreters, which each are
associated with a particular modeling paradigm, a multi-graph kernel, which handles model interpreter communication, an object oriented database, and a user interface. Their tool implementation for heterogeneous system modeling provides some insight into automatic construction of model interpreters as well as model code execution. Their environment is mainly targeted towards the modeling, integration, and simulation of system models from various engineering disciplines as opposed to embedded systems.

2.2. Work in Hardware/Software Codesign

Even though codesign is a fairly young field of research it has already become a buzz word. The term codesign can actually cover a variety of aspects in the design cycle which go beyond the literal design of computing systems. Within codesign we can identify a number of issues, e.g., co-specification, which addresses languages or specification for a unified specification of hardware and software components, hardware/software partitioning of complete design descriptions, co-synthesis, which is concerned with a conjoint synthesis of hardware and software components from a complete design specification, co-simulation, which encompasses the conjoint simulation of low level software descriptions and RTL level hardware descriptions, or recently even distributed codesign [3,45], which examines hardware configurations in the design of large distributed computing networks.

In early approaches [8,46,27], codesign has been defined as the conjoint development of hardware and software components of computing systems at the implementation level. Hardware here is considered to offer a medium for high performance computation with fast execution times, high implementation cost, which has its limitations in size and little
flexibility once it is deployed, i.e., in the form of a Application Specific Integrated Circuits (ASIC). Software implementations are considered to exhibit greater flexibility where size and cost is not being of relevance, but having the drawback of a considerably slower execution time as compared to hardware implementations. Codesign pursues mixed implementations instead of pure software solutions to improve application performance in critical operation phases.

In the following section, we will provide a brief overview and summarize some of the research in hardware/software codesign which the author believes to representative of this field. Gupta and Micheli offer a good summary of many proposed codesign approaches in [3]. An extensive set of abstracts and references of research in codesign have been collected by Buchenrieder in [47].

2.2.1. Early Approaches to Codesign

Good representatives of the early work in codesign are the approaches by Gupta and De Micheli [8], and Ernst et al [27]. The first approach is based on an initial hardware specification in HardwareC. Parts of the design specification are then moved into software as long as specified performance constraints are met. The second approach, which is often referred to as the COSYMA approach, accepts a pure software specification in the form of communicating processes. Performance bottlenecks in the design specification are first identified. Critical parts of the design are then assigned to execute on a custom coprocessor. The latter approach will be discussed in more detail in Section 2.3.2.
Eles et al. describe in [4] an integration of a suit existing design tools in a VHDL based specification of embedded computing systems. Their approach is fundamentally based on extended timed Petri net notation as a unified design representation which is coupled with an automated, transformation-based system synthesis. The authors point out some shortcomings of VHDL as a system level specification language and advocate some changes for a better representation of software communication. By applying iteratively a set of design transformations this unified representation is then refined to meet specified design constraints. Innovative is also their research on a cost function for a hardware/software partitioning in [48].

Another approach which is supports the prototyping of heterogeneous multi-processor computing systems using existing tools and languages is presented in [49] by Färber et al. Their research focuses on concepts for hardware/software communication and interface synthesis for abstract computing system specifications. They address the implementation of queue-based communication for codesign approaches using a task classification model [50] for a system specification which supports hard real-time system analysis. The communication synthesis for a mixed hardware/software implementation is presented in the context of a SDL system specification language.

2.2.2. Shortcomings of Implementation Level Codesign

Our previous definition of "hardware and software" leaves a lot to be desired. For example, we notice communication overhead between the various components is not considered in the previous definition at all. Staunstrup et al. also criticize in [51] this simplified definition and point out other fundamental problems with this approach in the
context of a simple priority queue design, e.g., radically different design styles of pure hardware or software implementations. Axelsson dampens enthusiasm about possible performance improvements using implementation level codesign by referring to the fact that an acceleration of a design implementation executing on a single processor is limited and bound to the famous Amdahl's law [52,53].

Next to mere processing power codesign also has to face challenges of a different kind in the years to come. Computing applications, especially in the domain of embedded systems, may today also list distributed, concurrent, extremely flexible, self-contained, mobile, or low power processing as requirements. Next generation applications should be able to run anywhere, anytime, and on any platform. Of course they still need offer performance comparable to previous implementations. The only way of handling these problems is to raise the level of abstraction from the implementation level, i.e., the RTL and algorithmic abstraction level, to the system level.

Early on, design at the system level was primarily justified to manage the design of large complex computing applications. This argument probably originated from the domain of object oriented programming where system level design concepts are applied to address this problem. System level design though also follows the current trend in the evolution of computing systems design practice: Gajski states in [7] that design methodologies have here shifted from a design-and-simulate period from the 1960s to 80s, to a describe-and-synthesize period in the 1980s to 2000, to a specify-explore-and-refine period in the near future.
This vision is supported by improvements in hardware and software implementation technologies in the last decade. Rapid advances have been observed, e.g., in flexible hardware components: most notably we notice a steady increase in the performance of reconfigurable Field Programmable Gate Arrays (FPGA) which slowly start to replace Application Specific Integrated Processor (ASIP) solutions [54]. Similarly, there is a strong trend in the next generation processors away from rigid general purpose processors to configurable ones [55,56]. On the software side programming languages are moving from high level languages like C to object oriented and even platform independent languages like Java. Finally the use of intellectual property modules (IP) [57], i.e., the packaging of hardware or software components into small, independent, self contained blocks instead of entire boards or programs, is rapidly gaining importance in computing system design in general.

2.2.3. System Level Codesign

Buchenrieder and Rozenblit were among the first to advocate system level codesign. In a compilation of codesign approaches published in 1994 [10] they were among the first to outline the idea of a system level codesign approach called model-based codesign. At the time the transformation of application models into implementations was not discussed. This dissertation presents the current state of research on this design approach.

Early publications of Buchenrieder and his research group at Siemens worked intensively on efficient code and hardware generation for systems specified in the StateCharts formalism. Their work includes the development of a flexible FPGA-based simulation target architecture [58] and an intermediate model format based on Extended
Programmable Random Access Machines [46]. Later work presented in [6] proposes codesign based on a system specification in Java. A corresponding method for a cosynthesis from such a specification is introduced. Here, they discuss an optional translation of multi-threaded Java code at the method level into VHDL. Similar as in [27] this approach to codesign starts out with a pure software implementation which is then accelerated with custom hardware components in critical sections after a profiling step.

A first discussion of a more general transition from a system level specification to an implementation level is provided in [25,59]. Calvez proposes here a refinement of executable system level design specifications in the context of the MCSE methodology using a four step partitioning process: functional, geographical, resource, and hardware/software partitioning. Publications outlines these phases as well as their goals and requirements but remains ambiguous on how exactly the transitions are facilitated or justified. The approach also focuses mainly on the refinement of a hardware specification and hardly addresses the software implementation. Nevertheless the importance for system level design for an optimal design implementation is emphasized.

Jerraya et al. were among the first to point out the importance of communication aspects in the synthesis of system level specifications. In [30], the group discusses the integration of communication synthesis in their codesign methodology COSMOS. Modeling of communication channels starts after converting standard system specifications into their intermediate system specification early on in the design cycle. After a manual partitioning onto a multi chip architecture system components are synthesized from a library of available hardware and software implementations as well as
various communication schemes. The product of this result is a mixed implementation specified in VHDL and C or possibly even assembly code.

Kumar presented a first detailed approach to system level codesign called the unified codesign. In his dissertation [32], which later on was published as book in 1996, he describes a design approach starting from a functional specification for the application under design. He introduced a concept of virtual machines to show similarities in hardware and software execution. Although no approach to a partitioning is provided a late system integration is recommended. Kumar used the design environment ADEPT to illustrate his design approach with some abstract examples.

Axelsson was among the first to discuss the shortcomings of implementation level codesign approaches and to advocate a concentration of codesign research efforts on a system level specification. Coming from a software background he proposes an approach for the design of heterogeneous real-time systems in [52] where he uses concepts of virtual prototyping and transformations to explore possible design implementation alternatives. He introduces a new notation enabling him to perform advanced task schedulability analysis for multi-processor systems. In addition, he describes an approach to software synthesis from a system specification using this notation in detail. It assists a partitioning on generated target multi-processor architectures based on various search algorithms.

2.3. Codesign Environments

A number of codesign environments have been implemented by various parties in academia and industry. Since a closed environment is the best way for introducing a
novice to codesign concepts and to evaluate these for his or her own purpose we decided to present some of the available and more well-known tools which evolved from the previously surveyed research. It also helps to put theoretical concepts into more of a practical application context. This list is by no means complete and is intended to provide the reader also with a more detailed insight into the actual design process associated with some of the codesign methodologies.

2.3.1. Polis

The codesign framework POLIS [2] is a joint project of the University of California at Berkeley and Cadence Design Systems. Its goal is to enhance the design of control-dominated embedded systems applications. There, the system specification provided in a combination of C and Esterel which are translated into a Codesign Finite State Machines (CFSM) representation (also sometimes in literature referred to as Communicating Finite State Machines) to be formally verified. A functional validation can be obtained by generating an appropriate description in a simulation framework where modules of different underlying computational models can be connected together graphically.

After a manual partitioning by the designer followed by automated hardware and software synthesis, the generated implementation is evaluated using a co-simulation engine. One benefit of this development environment is the ability to synthesize the formal CSFM representation automatically into hardware/software components. The partitioned components can be co-simulated at a cycle accurate level to guide the designer to the next partitioning iteration. In POLIS the hardware/software partitioning step is left to be a designer decision.
2.3.2. COSYMA

COSYMA, a design environment developed at the Technical University of Braunschweig [27], focuses on the generation of a dual processor platform for the exploration of the codesign implementations. Targeted are applications that require intensive data processing. Computing systems are first specified using an extended version of the programming language C called C'. If a run-time analysis fails to satisfy the specified timing constraints, processes are then moved to an automatically generated application specific co-processor in an iterative manner. Automatic design support guides the designer in scheduling the software tasks to satisfy hard and soft real-time processing constraints. Hardware/software partitioning is supported by simulated annealing - a stochastic optimization algorithm. Recent approaches in COSYMA try to incorporate reactive systems into their suite of applications and also address systems composed of multiple heterogeneous processors.

2.3.3. Virtual Codesign Composer

In 1998, Cadence, Philips, and Infineon Technologies AG (then Siemens AG) started the COSY project. The goal of the COSY project was to develop and deploy a design flow and methodology with the necessary supporting tools and models for system design based on re-usable hardware and software IP modules. COSY and Cadence's Felix initiative lead to the commercially available Cierto Virtual Codesign Composer (VCC) environment from Cadence [60,61]. The theoretical concepts incorporated in this tool are similar to the previously presented POLIS environment [2]. VCC enables designers to make trade-off early at the system-level by evaluating and comparing the impact of
architectural choices. This is achieved by high-level performance estimation and modeling techniques. Basically the tool splits into four tasks: functional modeling, architectural modeling, mapping, and links to implementation.

During the functional modeling phase the specification of the system takes place. At this stage, the system designer is not interested in detailed interfaces. Different modules exchange data using abstract data types. An algorithmic IP library aids the designer reusing behavioral models. The VCC discrete-event simulation engine allows the execution of the functional application specification. The modeling of the application target architecture is based on a library of abstract hardware and processor IP modules, as well as communication busses.

The mapping phase merges the functional application and the abstract target architecture description. It basically adds timing information to the functional model depending on whether a functional model component is mapped to hardware or software. Performance of software components is here estimated automatically by abstract processor models. Bus load, scheduling policies, or partitioning strategies can thus be checked on a high-level. Execution estimates for functional components which are mapped to hardware components have to be supplied by the designer. Notice that also the partitioning of the functional model is done manually. After interface refinement of the mapped model, the mixed hardware/software design is exported to the implementation level which is referred to as the Implementation Analysis. This step mainly performs as a consistency check of hardware and software descriptions, and generates software tasks for functional components which have assigned to processors in the mapping phase.
Hardware descriptions are assumed to be available as IP modules. Communication patterns between hardware and software can be chosen to support the implementation step.

2.3.4. DEVS/DOC

Recent work at the University of Arizona published in [31,62] presents a system modeling framework for distributed codesign called DEVS-DOC. Contrary to all previous approaches, which address the development of high performance embedded systems, this formal approach is targeted towards the codesign of complex distributed computing systems. Similar as in the VCC approach the design is separated into a model of the actual application itself and an architectural model of the computing network. This allows the designer to perform trade-off analysis of different hardware configurations for large software applications.

Fundamental to the framework is the set theoretic formalism of Distributed Object Computing (DOC) [63]. It models systems in two distinct layers: a software layer (labeled distributed coorporative objects), and a hardware layer (labeled a loosely coupled network). In addition, an object system mapping defines the allocation of software objects to hardware objects. An execution environment has been provided which extends the discrete event specification DEVS with components that encode these abstraction concepts.

The modeling and design of computing systems applications is achieved in four phases: First the distributed hardware architecture is defined in the form of processing nodes, gates, and communication links. Similarly software objects are specified in the
form of system models to encode the desired application behavior. Then a mapping of the software object onto processing nodes is defined. Finally an experiment is specified to stimulate the application during simulation and to collect pertinent data for a behavioral analysis. In a computer-based execution of DEVS-DOC models the designer can then monitor a variety of design metrics to make design decisions on the selection and partitioning of software and hardware components. These metrics include number of active software objects, computational workload, execution times for computational domains, memory utilization of processors, data traffic for specific communication links, as well as any other metrics of interest to the designer. Since this framework has been primarily targeted towards the analysis of different distributed design implementations it does not yet offer a link to automatic software synthesis.

2.3.5. SONORA

Another research effort at the University of Arizona is the development of an integrated CAD environment for the design of heterogeneous embedded systems called SONORA [11,29]. This environment is intended to realize the conceptual tenets of model-based codesign methodology presented in [12]. Figure 2 depicts the proposed design flow.

The Functional and Behavioral Requirements Specification and Modeling block in the diagram represents the initial phase of a complete design cycle and embodies requirements solicitation and documentation which is followed by development of an executable model specification. The Behavioral Simulation and Model Refinement Loop is used to iteratively refine the design model until it is functionally correct. Structural Requirements Specification and Modeling relates physical design constraints to a
proposed physical architecture. In the Performance Simulation and Model Refinement Loop, the model is enhanced with performance estimates for computation and communication based upon the proposed target architecture. Here, innovative research has been conducted on using a Bayesian Belief Network (BBN) based approach [64] to evaluate performance results during the simulation cycle which guides the designer during the hardware/software partitioning process. Synthesis and Implementation involves extracting design information from the models in order to produce a physical prototype. Experimental Frame Development and Testing involves the creation of a set of test cases based upon the system requirements that are used to assess the current design at all stages of the design process [40,65,66,67].
2.4. Summary

As pointed out by various authors [12,32,52] many approaches in the field of codesign have been traditionally biased and centered around solutions originating from the hardware domain. This can be explained by examining the background of many researchers which tends to be in high-level hardware synthesis. This may explain the initial surge of codesign approaches based on detailed, implementation level design specifications. It is only recently that researchers coming from the software domain have positively influenced this field and helped raising the level of abstraction in codesign. In the light of the abundant criticism on implementation level codesign presented in this chapter the author wants to point out that this form of codesign still remains of interest to embedded systems design, e.g., in the optimization of application implementations.

It is our belief that given the changes in hardware technology and requirements in the next generation of embedded systems applications a system level approach to codesign is mandatory to derive flexible, well integrated, efficient design implementations. Our model-based design approach from a system modeling perspective allows a completely implementation independent design specification and evaluation of virtual application prototypes.

An often overlooked aspect of embedded computing systems design, the incorporation of its environment, is another good reason to choose model-based design approach. In systems modeling, we have already a large body of established research in modeling a variety of physical systems which be used directly in the application development. Embedded computing systems are usually highly interactive and strongly
dependent on their environment which is likely to be a physical system of a different engineering discipline.
3. MODEL-BASED CODESIGN

After a brief review of the commonly encountered design flow for embedded systems applications we review the tenets of a model-based codesign methodology. This methodology follows embedded systems application development from its system specification to the final implementation. This chapter serves in essence as a road map for the following chapters which then present the individual design phases of this methodology in more detail.

3.1. Conventional Design of Embedded Systems

Conventional heterogeneous systems design is based on multiple, subsequent hardware and software development steps in which designers refine design specifications and construct a system prototype [12]. Based on experiments and system profiling at the implementation level, functionality is moved from software to hardware implementations and vice versa. Ad hoc approaches result an immediate partitioning into hardware components or a processing platform, and software components. Hardware and software development threads are pursued in isolation from each other. This conventional approach to design is depicted in Figure 3.
Figure 3. Conventional Approach to Embedded Systems Design

Many authors point to the deficiencies of the conventional codesign frameworks [2,8,9,10,12,52]. They strongly advocate a process that fosters the integration of the hardware and software perspectives and postpones a hardware/software partitioning until design specifications have been fully refined and verified. Thus, a unified representation is needed for modeling a system independently of its implementation in hardware or software components.

3.2. System Modeling for the Design of Computing Systems

System modeling and simulation techniques have been applied extensively to study already existing physical [19] or to assess a system specification of computing systems [38,39]. Nothing prevents us though to use these modeling techniques also in the actual
design of such systems. As matter of fact, such a design approach contributes numerous benefits. One of the most important benefits of a system level design approach is a completely implementation independent application specification. Such a design also specification fosters an early system integration as well as a late design partitioning into hardware and software components which is not addressed in conventional embedded systems design approaches.

In addition, many system specifications of complex embedded systems mostly describe the behavior and interface of the application in its entirety and provide too little detail for a direct complete design implementation. In our model-based design approach, multiple levels of abstraction can be used to specify and evaluate such applications without the knowledge of such details.

3.3. The Model-Based Codesign Methodology

Our position is that the use of formal specification techniques is only of limited effectiveness without a systematic modeling methodology guiding their use throughout the codesign process. It is such a methodology that has been the core of our research effort. We have developed a model-based design framework in which abstract models of heterogeneous systems can be developed, and validated through simulation, and mapped onto their physical realization [10,12]. Here, the modeling of embedded systems applications is based on the construction and refinement of modular design models using a discrete event modeling specification. The flow of design activities in this framework is shown in Figure 4.
The depicted design process is based on the following activities: given a set of design specifications, requirements and constraints, a model of the system under design is developed. To build a simulatable design model, we first specify its structural aspect in the form of model components from which a computing system will be built, their relationships, and attributes. Such models are given a behavioral specification so that they can be simulated.

Simulation is then used to verify the functionality of the proposed solution, i.e., the execution of the model's dynamics to ensure that the behaviors are consistent with those
perceived for the system being designed, and secondly as a way of assessing how well stated performance requirements are met by the proposed design solution. Embedded systems can be subject to challenging performance constraints. By simulating our system design models over time we can acquire performance measurements and validate their compliance with specified real-time constraints. An assessment of simulation data leads to further model refinement and design modifications.

An efficient design implementation, which consists of mixed integrated hardware and software components, is generated from the design model specification after it has reached a sufficient detail for a design model transformation. Based on the information encoded in this design model specification we determine a proper partitioning into hardware and software components, and specify required interfaces between components as well as with the environment.

3.4. Steps in the Methodology

As depicted in Figure 4 model-based codesign is based on a multitude of design phases. In this section we summarize these design phases to provide the reader with a better picture of the overall design process.

3.4.1. System Specification

In our first step, we create as an initial high level model of our embedded systems application from its system specification. Here, we capture system requirements provided in an abstract design model specification. In order to verify a design solution, a
systematic way of converting the requirements into a formal representation is needed [66, 68, 69].

The application system specification should define the interface between the system and its environment and the system's functionality. Structural requirements for the application can be omitted in the construction of our initial design model. These and other requirements, e.g., power, size or restrictions in the processing platform, are introduced later either in refinement of the design model specification or its transformation into an implementation.

It is often difficult to obtain or agree on one concise system specification from a customer early in the design process. We address this problem with an implementation independent system modeling approach which allows the assessment of an application even at a high level of abstraction.

3.4.2. Modeling

A model is a set of instructions for generating behavioral data. Valid model-generated data should be equivalent to real-system data (actually a subset of it since we cannot possibly build models accounting for all input-output behavior of the real system). We construct models that correspond to a set of questions, objectives, and purposes for which a design modeling effort is undertaken. Models are constructed at the level of abstraction that corresponds to those questions.

Using adequate model specifications facilitates transitions across different levels of abstraction. In systems modeling theory, the behavior desired from an embedded systems component is described formally by transition functions which describe changes in the
values of system variables. The behavior of the entire embedded system is further described by interactions between model components. Discrete event specifications based on mathematical formalisms (e.g., DEVS[19], StateCharts[42], CSP[70,71], Petri nets[4,7], etc.) can be used to specify such a design model specification. Model components can be interpreted as design "blueprints" because they encode behaviors. No decisions regarding the realization of the application have to be made at this stage of the design process.

3.4.3. Design Model Refinement

The most common approach in model-based design is a top-down refinement of the design model specification. Application development starts with an abstract specification where and details of the application operation may be put aside. In the refinement process, more details are added to design model in a stepwise fashion. After each design iteration, the design can be assessed in its entirety and then returned to the design team for further development. The level of abstraction does not have to be uniform across the entire design specification. Model components can be specified at different levels of resolution. Some components in a design model may even be exempted from refinement at a certain stage because they have been already completely designed previously.

In a structural model refinement, we decompose the application design into smaller components in order to reduce the overall complexity. The derivation of a modular model components is here of primary importance. It allows reuse of already designed components in an application model as well as easier composition of design alternatives. It also provides an excellent foundation for the derivation of concurrent or distributed
application implementations. In addition, modular design model specifications support the testing and development of design model components in isolation. A hierarchical representation allows us to further condense the information of these modules.

3.4.4. Design Model Analysis and Evaluation

Simulation, i.e., the execution of models over time, can be used to test a design model specification for a multiplicity of objectives. An evaluation of simulation results can be used to establish, e.g., the performance limits and bottlenecks of the design model. Objectives can be converted into design model experiments which reflect operating scenarios in its final deployment, i.e., circumstances under which a model (or the real system) is to be observed [72]. In systems modeling theory, this concept has been already addressed in the context of experimental frames [73].

In design of embedded systems, an experimental frame is a representation of the environment which the application interacts with. Experimental frames consist of model components which are specified separately from the design model. Three types of model components have been identified to be elemental to an experimental frame: a generator, which produces data to be processed by the model under test, a transducer, which is used to analyze its response, and an acceptor, which controls the experiment.

The simulation of a design model with an experimental frame can be used to verify that a set of system requirements are met. Different experimental frames can be created to encode or test for different design objectives, i.e., observe these models under different experimental conditions, or to assess alternative design model specifications.
Experimental frames may also be reused to test the application design at the implementation level.

3.4.5. Design Model Transformation

After the design model specification of the application has been refined to an appropriate resolution, we generate a feasible processing platform in a phase we call model compilation. This target architecture is then used for the implementation of design model components and addresses a well-known shortcoming of current system-level design approaches which is known as the model continuity problem [32,40].

Target architectures can be composed of various architectural elements which include any type of conventional processors, microcontrollers, custom hardware components, software processes, as well as dedicated or shared communication links. A processing platform, i.e., a composition of these architectural elements, can be directly derived from the structure of a modular design model specification. Alternatively, a design model specification can be adapted to a given processing platform. A model compilation attempts to associate design model components with architectural elements.

The implementation of the design model follows after a successful model compilation. Each model component is here either converted into a generic hardware or software implementation. Therefore, the composed design model is converted into a mixed implementation consisting of both, hardware and software components. The universal applicability of this final transition to the implementation level can be shown with a formal mapping from a system modeling specification to an implementation level system specification.
4. MODEL-BASED EMBEDDED SYSTEMS DESIGN

In this chapter we introduce briefly model-based design at the system level, present a new abstraction of embedded systems computation for the construction of system level design model, and finally conclude with the presentation of a candidate modeling environment.

Since our design process is iterative this chapter should be read in conjunction with the following chapters on Design Model Refinement and Design Model Analysis. There we will present in more detail refinement and analysis techniques which facilitate the eventual transformation of system level design model specifications into implementation level design specifications.

4.1. Model-Based Design and Codesign

Model-based design attempts to solve the hardware/software codesign problem from a true system level perspective. As we discussed previously, many codesign approaches attempt to optimize a given implementation level design specification by moving parts of the design between custom hardware or software implementations. We discuss here a design step which is to be taken beforehand: The derivation of such a design specification from a given system specification which is most likely to be given in a textual and informal manner.

In our opinion it is the early phase in computing system design where the designer has the most leverage on an efficient final application implementation. The forced bias towards any specific implementations in a design specification significantly restricts the design space for a given application. Design at any lower level than the system level also burdens the designer with the details of optimizations and system integration issues
which introduce unnecessary complexity in the initial design phase and divert from the original intend of the design: the desired behavior of the application itself.

4.2. Essential Ingredients for a Powerful Design Specification

It is our goal to use modeling specifications to eventually arrive at an efficient complete design specification for embedded computing systems applications. Therefore, it is worthwhile to examine desirable properties for modeling specifications which are facilitate the derivation of efficient but yet flexible design specifications. We have identified four such properties which we believe to be essential: concurrency, hierarchy, determinism, and modularity. In the following discussion we justify these choices. Notice that it is most desirable that the modeling specification used for model-based codesign not only allows the representation of these properties but also enforces them.

4.2.1. Concurrency

Parallel processing architectures ultimately lead to the best performance results for a given computing application. Benefits in execution performance are strongly dependent on the level of concurrency exhibited by the particular application which is to be designed. Another classical drawback of parallel processing architectures is that their design complexity rises dramatically as the application of parallel execution concepts drops [74], i.e., its granularity. This growing complexity is caused by the inherent need for synchronization in such architectures which is necessary to guarantee its correct execution. Arguably best results can be achieved for a given application behavior by extracting as much coarse grain parallelism from a behavioral specification as possible.
The extraction of concurrency from a given behavioral description can be best achieved at the system level. By definition, system level modeling specifications allows the expression of concurrent computation in the behavioral description of a composed design model. At the system level we can easily experiment with concurrent definitions behavior for a given abstract application system specification, or identify such behavior from application model specifications. In addition, the system level allows us to focus directly on the application behavior in isolation from its final implementation.

4.2.2. Hierarchy

Hierarchical representations allow us to condense design information and therefore reduce design complexity. This concept has already proven to be extremely useful at the implementation level: In the hardware domain different levels of hierarchy help to simplify design descriptions, e.g., gates, circuits, chip, board, etc. In the software domain, this concept is used extensively in object oriented programming.

A hierarchical representation of the design models simplifies the specification of composed application models by describing member model components as components which may in turn contain other components, etc. At each level of the hierarchy some more information about the internal application structure and behavior is added to the design model. By constructing design models in a hierarchical fashion we can selectively hide details of design specification and may observe an application design or its components at various levels of resolution.
4.2.3. Determinism

Deterministic execution of behavior specified in a design model specification leads in general to more reliable application implementations. Here, every transition of the system behavior has to be specified in a unique manner and requires an explicit specification of the application design behavior at every point in time. Most modeling specifications used today allow only the deterministic specification of model component behavior. An implementation of deterministic design specification guarantees predictable execution of our final application implementation which is mandatory for real-time or safety critical embedded systems applications.

4.2.4. Modularity

A completely modular design model specification requires the a rigorous specification of the external interface of a composed system model and its components. Modularity is achieved by avoiding any direct sharing of data or information between any two system components, i.e., disallowing the use of global data structures. Instead any shared data items need to be explicitly communicated between components. The concept of a modular specification is well known in software domain. Increasingly it is also becoming more important in hardware domain as the hardware market demands the introduction of IP modules.

In model-based codesign, the use of modular design model specification simplifies the mapping to efficient design implementations significantly. It allows us, in theory, to allocate any design model component to any hardware or software realization since modular model components are by definition completely self-contained. Modular model
specifications have also been shown to be extremely useful in system model development and the execution of discrete event modeling specifications [41].

4.3. Real-Time Embedded Systems

Embedded system applications which operate on some specified timing constraints are commonly referred to as real-time computing systems. The importance of meeting their computing deadlines creates two classes of real-time systems: hard and soft real-time systems. Soft real-time systems are computing systems which may also accept a late servicing or dropping of requests in respect to specified timing constraints. Hard real-time systems on the other hand are considered to behave incorrectly if even a single timing constraint is violated [52]. For these type of real-time embedded systems application such a violation may lead to disastrous outcomes, e.g., which are part of safety restraint systems for any kind of vehicle.

The critical importance of processing deadlines and their observance in the final application design requires explicit specification of timing in a design model specification. Formal executable modeling specifications support the representation of such timing constraints already at the system level, and allow either the validation of such constraints using simulation or a formal verification.

4.4. The Model-Based Design Process

We can distinguish between two phases in a model-based system development: the initial construction of a design model, and its development. Here, the construction of design models is concerned with derivation of a design model specification form a general
application system specification. Our starting point for the construction of a design model is likely to be a textual, informal description of the application as it is perceived by our customer. These descriptions may vary in detail and accuracy in the early design stages, and even contradict itself in very large system specifications [69]. The absolute minimum is some idea of the external interface of our application, i.e., the interaction of the application with its environment.

4.4.1. The Construction of Design Models

As discussed in the previous chapters we build our design model specification in a hierarchical, modular fashion, adhering to the well accepted principles of decomposing the design model space into a structural and behavioral aspect [12,75,76]. The design model structure is commonly specified in some form of a block diagram. Initial design model specifications tend to contain initially only little information about the internal structure of the application. This fact should be considered a benefit rather than a drawback since it increases the number of possible design model compositions or configurations which can be derived in the refinement of this initial model specification.

Design model behavior is extracted from descriptions of the application operation as well as performance requirements stated in its system specification. Application behavior is encoded in discrete event model specifications based on some variant of a FSM representation. As we will see in the remainder of this chapter we can specify the behavior of embedded systems applications by identifying computational segments and associating model states with them. These model states represent the execution or processing of such a computational segment. Model state transitions specify an execution
order for multiple segments in a model component. Performance requirements can be represented in the behavioral aspect of the design model specification using timed model state transitions.

4.4.2. Validation of Design Models

Our initial system model specification represents a first executable design specification of our application. It can be simulated or executed over time to validate a correct functioning of our abstracted application design, and to detect and remove inconsistencies in its system specification. For a proper assessment of our design we also need to specify model components which encode relevant behavior of environment that our application is embedded in. We refer to these models as experimental frames [12,38,66].

The construction and development of experiment frames is similar to the one of design models but instead of modeling a computing system we may need to model a physical system that the application interfaces with. In the construction of experimental frames, it is in our interest to keep its models as abstract as possible. The number of the model components within the experimental frame itself may still vary from the bare essentials, i.e., model components which generate design model inputs and analyze its response, to full fledged system models of the environment itself [77]. The verification of system response times for embedded systems application can for example be encoded in such components. In experimental frame development, we have to ensure that interface and data types used in communication with the design model match in resolution at all times in order to guarantee a valid experiment.
4.4.3. Design Model Development

We recommend to start the model-based design process with a fairly abstract initial design model behavioral specification. The behavioral aspect of this model specification is specified based on event communication instead of detailed protocols and with information flows at a low level of resolution, i.e., computations which use abstract data types. The structural model aspect should be kept fairly simple, i.e., it should consist of as few model components as possible. Design model components may of course also be specified at lower levels of resolution if such model specifications should already be at hand. A high degree of abstraction or a low resolution of the design model specification is desirable since it will ultimately provide us with the greatest flexibility in the following design model refinement.

Model development is then mainly concerned with a correct refinement of this initial design model specification. Here, application requirements are introduced in a stepwise refinement process to increase the design model fidelity. After each step the refined design model is analyzed with respect to various objectives, e.g., validity of the design model, correctness of its refinement, or suggestions for subsequent refinement steps. During model development the resolution of the design model specification is increased until it allows the generation of a complete design specification.

4.5. Modeling of Embedded Computing Systems Behavior

We introduce now an abstraction of an embedded systems behavior which we use for our construction of design models. We present a more formal discussion of the behavioral aspect for embedded systems design models and their model structure. In addition, we
present the modeling of communication within composed design model specifications. For the moment we disregard here any particular embedded application context or any kind of system requirements which may be specified, and focus on merely on a general specification of a embedded computing systems design models.

4.5.1. The Behavior of Embedded Computing Systems

Computation, which is performed by any kind of digital computing system, can be fundamentally decomposed into two phases: a computation on some specific data, and prior to that waiting for these data to arrive. Similarly, we can view a more complex computation as a collection of either sequentially or concurrently executing computational segments (or a mixture of both) [70,71], and their waiting for input data. Based on this model of computing systems behavior, a decomposition of any computation eventually results in such basic computational segments which wait for input data, perform some computation on it, and then generate outputs for other segments.

Since we intend to implement our embedded systems application in the form of a digital computing device we may also refer to its behavior as a computation. In its system specification we can identify computational segments fairly easily: An indicator for the beginning of a computational segment is the requirement of data from its environment or other computing system components, i.e., other computational segments. Similarly, the end of a computational segment is marked by the generation of data outputs for other computational segments. Other such indicators include repetitive processes in the application description.
4.5.2. The Basic Model Component Specification

In a modeling specification, we can describe such a computational segment in a model component specification as illustrated in Figure 5. This model component specification can be considered the most abstract, or basic representation of any behavior exposed by an embedded computing system. Here, the model states $S = \{ W, P \}$ represent "waiting" and "processing" phases of computation. Here, $P$ is associated with the actual computational segment or some abstract specification of it. The initial model component state of this model specification is $W$. The maximum time spent in $W$ can be infinite (the data never arrives), and is in model state $P$ equivalent to the execution time for the computation once the required data has arrived.

The set of external input events $X_w$ represents external data sent from other model components which is required by the computation encoded in this basic model specification. These input data are then processed within some required execution time which is specified in the form of application performance constraints. The data, which is modified during the computation and required by other computations, is generated in a set of external output events $Y_p$. These output data only become valid after the
completion of the computation. Internal model state transitions lead from W to P, and from P back to the same waiting state.

The behavior, which is defined by this basic model component, can be described from a systems modeling perspective as follows: A basic model component stays in the waiting state W until the required external input data for the computational segment, which is associated with P, have arrived. At this point an internal event is generated which changes the current model component state to the processing state P where the specified computational segment is executed. After some specified execution time the computation completes and an internal event as well as external output events are generated instantaneously. The model component then transitions back to the same waiting state.

Notice that the waiting model state in the basic model component specification really represents two model states, i.e., a waiting state W* and collecting state C* as shown in Figure 6. We merge these model states into a single waiting model state W to simplify the illustration of the behavior which describes more complex computations. A special case of this behavior is specified if the computational segment associated with P does not require any external data. In this particular case, the model state W can be eliminated from the basic model component specification since it reduces to an superfluous transitional model state.
Again we want to emphasize that the computational segment associated with P may be described abstractly since the application system specification may not provide an accurate enough description of such. We may just simplify the description of a computation as basic input/output relationships: for a given external event A we produce an event B, yet the details of how we compute B from A have not been completely specified. In the design model development then, we refine abstract computations to more detailed computations. Notice also that this does not restrict us to a software implementation for the computation. We are merely interested in a more accurate information flow for the corresponding computation.

4.5.3. Specification of Design Models for Complex Computations

In the model specification of more complex computations, i.e., computations which consist of multiple computational segments, we may pursue one of two approaches: either extend out single model component behavioral specification, or specify our computation as a group of basic model component specifications, i.e., a coupled design model specification. Figure 7 and Figure 8 show simple examples of these approaches. The appropriate selection between these two choices is best determined during design model refinement. In practice, we use often a blend of both forms of model specification when constructing design models for more complex system behavior.
In the first approach shown in we have to associate two unique model states, i.e., in our example P1 and P2, for each identified computational segment in a single model component specification. The component states in such a model specification can be generally specified as \( S = \{W_1, P_1, W_2, P_2, W_3, P_3, \ldots, W_n, P_n\} \). Although we show a transition from P1 to W2 we may then transition theoretically to the waiting state of any other computational segment after the completion of a processing state. In addition, output ports are created for each processing state which produces external outputs, i.e., for each set \( Y_{P_i} \subseteq Y \) where \( Y = \{Y_{P_1} \cup Y_{P_2} \cup Y_{P_3} \cup \ldots \cup Y_{P_n}\} \) and \( Y_{P_i} \neq \{\emptyset\} \).

![Figure 7. Single Model Component Specification of Complex Behavior](image)

Notice that in such model specifications each set of external output events \( Y_{P_i} \) (\( Y_{P_1} \) or \( Y_{P_2} \) in Figure 7) only reflects a subset of the outputs produced by a computational segment for computational segments which are part of other model component specifications. Modifications which affect computational segments in the same model component specification (e.g., changes by P1 which affect P2 and vice versa) are excluded from \( Y_{P_i} \). In the component model specification these modifications are internally propagated as internal events.

We may also specify sequences of processing states in our model component specification. These cases arise if a computational segment does not require any external data from other components but only component internal data. In such cases the waiting
state can be eliminated since it is purely transitional (e.g., if $X_{W2} = \{\emptyset\}$ in Figure 7 $W2$ may be eliminated and the model specification then directly transitions from $P1$ to $P2$).

In the second approach illustrated in Figure 8, we have to explicitly communicate data dependencies between computational segments by sending external events between different model component via specified output to input port couplings, i.e., along model component communication channels. Here, each set of external output events $Y_{P1}$ (e.g., $Y_{P1}$ and $Y_{P2}$ in Figure 8) reflects again data required by computational segments which are assigned to other model component specifications.

![Figure 8. Coupled Model Specification of Complex Behavior](image)

4.6. Derivation of Required Execution Times for Computations

As we emphasized in the previous sections, timing constraints are of primary importance in real-time computing systems design. We can validate these constraints by executing our design model specification over time and to monitor the events involved in timing constraints. In our abstract modeling of computation we have been assuming so far that the execution time of each computational segment is known for the design model.

In an informal system specification, these execution times are usually only available as required execution times for the application to be designed in the form of performance requirements. But even if performance requirements include specific timing constraints
they are mostly specified as system response times, i.e., for the entire composed design model behavior, instead of specific computational segments.

Nevertheless, execution times need to be specified for each and every computational segment in order to reason about design model performance and to synchronize concurrent computations properly. In addition, modeling specifications, which are based on explicit timing, require the specification of execution times (or internal transition times) for each system model state.

4.6.1. Unknown Required Execution Times

For unknown execution times for computational segments can use simulation results to distribute global performance constraints onto the computational segments of each model component specification. We approach this problem from a modeling perspective where required transition times are due to our definition of design model components equivalent to respective processing model state execution times. Given a model specification for some complex computation with unknown execution times and one timing constraint, we can first run a controlled simulation where we specify all processing state execution times in our design model to be some common value, e.g., one. The purpose of this value is merely to establish an ordering in the processing of computational segments.

We then execute this first design model specification and apply stimuli specified for a particular timing constraint. In the case of multiple timing requirements we may need to evaluate multiple simulation runs. If there are no timing constraints specified we have to provide an educated guess for a reasonable timing constraint [66]. By monitoring the design model component processing states during a simulation we can derive a design
model state execution trace for each simulation run. This trace allows us to identify which computational segments are executed and how often. Notice that at this point we do not make any assumption on how or where any of the design components will be eventually implemented.

Based on this information and some measure for the computational complexity of each computational segment we could then theoretically compute execution times for each of the corresponding model states which are part of these execution traces. In the case of multiple timing constraints we would then choose the lowest computed execution time for each respective computational segment derived from the different simulation runs.

4.6.2. A Measure for Computational Complexity

What measure do we use to estimate the computational complexity of computational segments best? As we pointed out to earlier, simulators can be interpreted in modular discrete event model specifications as processors which execute the behavior (or model instructions) for a specific model component [41]. Here, a model component specification can be simply interpreted as an application executing on an abstract processor. Notice that in a composed design model we have actually a simulator attached to every model component. Therefore, we are dealing here with a problem which is already well known in computer engineering: How can we compute processor performance in an unbiased manner and also consistent across multiple processors?

Patterson and Hennessy evaluate in [78] numerous measures for an unbiased performance comparison of different real-world processors which execute the same
application program. They suggest the following equation (1) for an unbiased computation of application execution time:

\[ \text{Execution Time} = \text{Instruction Count} \times \text{CPI} \times \text{Processor Cycle Time} \]  

(1)

We can adapt this formula as shown in equation (2) to arrive at a good estimate for model state execution times: Here we use the model instruction count as a measure for the computational complexity of our application, assume a CPI (cycles per instruction) count of one, and introduce the concept of a model component cycle time. The model component cycle time will be considered a model component property: once it is properly established from the evaluation of the respective model component execution trace the execution times of computational segments, which are part of this model component specification, will be computed from that model component cycle time.

\[ \text{Execution Time} = \text{Estimated Model Instruction Count} \times \text{Model Component Cycle Time} \]  

(2)

Notice that an estimated model instruction count may differ from the actual model instruction count. The use of the latter measure may not necessarily represent a measure for the computational complexity of the application. Remember that computational segments can be specified abstractly. The actual model instruction count only represents a measure for the computational complexity of the current model specification, i.e., the computed model state execution times would in this case reflect the time needed to execute the computational segment at its current level of resolution.

Therefore, we deem an estimated model instruction count to be the best alternative to obtain a resolution independent measure for the computational complexity of computational segments. What exactly corresponds to a model instruction is as in [78]
left to the designer. What matters is that its definition is consistently applied throughout the design model specification. We suggest an instruction to be a single operation on a data item, e.g., assignment, addition, multiplication, etc. Model instruction counts for abstract computational segments, special functions (library functions such as sin(x)), or abstract data type manipulations have to be initially estimated. Notice that in the following sections and chapters we will assume that any instruction count specified for model component specifications refer to model instruction counts.

4.6.3. The Computation of Model State Execution Times

The model component cycle time is calculated by inspecting the processing model state execution trace of the initial design model as described in Section 4.6.1, and relating it properly to the estimated instruction counts of computational segments associated with the respective processing model states. The lowest required cycle time for a computational segment in a particular model component specification will be selected as its model component cycle time, and used in the calculation of the other execution times. Notice that this approach to execution time estimation also allows the derivation of execution times for model states which are not part of a model state execution trace.

Again we want to emphasize that this approach is intended to only provide us with an initial estimate for unspecified execution times for model development. Based on these estimates we may assess our design model if it complies with global timing constraints specified in the system specification. The computation of execution times is, especially for initial design models, often based on instruction count estimates which may not necessarily correspond to the instruction counts in our final design model specification.
As the design model is refined and gains in fidelity, instruction count estimates are refined and result in higher quality of required execution time estimates. Ultimately the highest quality required execution times are achieved in the final stages of model refinement when abstract computational specifications are eliminated from the design model components, i.e., design model components reach the highest resolution level: here the actual model instruction count can be used as the instruction count.

4.7. Modeling Component Communication at the System Level

As we have discussed earlier the communication of data and its proper synchronization is of major importance in the design of computing systems which exhibit concurrency. At the system level communication along couplings is by definition instantaneous. In order to reflect the required synchronization in component communication we use generic communication components which are placed in between the source and destination model component specified by such a coupling. They model in essence a communication channel. As we will see in the next chapter the introduction of communication components is in fact necessary to facilitate a correct timing refinement of design model.

The purpose of communication components is twofold. Primarily, they allow us to monitor various parameters of a communication channel such as the size and required time to transmit data properly between model component specifications. Secondly, we use communication components to synchronize communication. The need for a buffering of communicated data arises since model components may react to stimuli of the environment which arrive at different data rates. Inputs generated by human interaction
4.7.1. The Communication Model Component Specification

A simplified communication model component specification is illustrated in Figure 9. The model structure of a communication component is based on a data input port which is connected to the source model component, and a data output and notification input port connected to the destination component. Notice that the event for notification input is assumed to be generated by the destination component specified in the coupling. This requires a slight modification in the behavior of destination model component behavior where such an event has to be generated with the internal transitions form processing model states.

![Figure 9. Simplified Behavior of a Communication Model Component](image_url)

The set of model states for a communication component includes the initial waiting state \( W_i \), buffered waiting state \( W_b \), unbuffered waiting state \( W_u \), and a delay processing state \( P_d \). The execution time for \( P_d \) is computed from a delay parameter \( t_{delay} \) which is unique to every communication model component specification. This parameter denotes
a lower bound on the communication time required to transfer data (depicted as the set of events $X_S$ in Figure 9) from the source to the destination component. At the time of instantiation the delay parameter is zero. It is frequently modified when execution time for the processing model state of the source model component, which generates the corresponding external events, is updated during design model refinements.

The behavior exhibited by a communication component can be described as follows. Initially the component waits in $W_i$ either for a set of external events $X_S$ from the source model component, or a notification event $x$ from the destination component when it is ready to receive data. Should $X_S$ arrive first the component transitions to the buffered waiting state $W_b$ where it awaits the notification event of the receiving component. Once the destination component sends the notification signal the communication component transitions to the delay processing state $P_d$. The processing state then delays the propagation of the data to be transferred based on the elapsed time waiting in the previous state $W_b$ for the arrival of the notification event $e_t(W_b, x)$ and the value of the delay parameter $t_{delay}$. Therefore, the time spend in $P_d$ $t_a(P_d) = t_{delay} - e_t(W_b, x)$ if $e_t(W_b, x) < t_{delay}$ or $t_a(P_d) = 0$ otherwise. The data $X_S$ is then generated and send to the destination components in the form of external output events, after which the component transitions back to $W_i$.

If the destination component should have sent already its notification signal prior to the arrival of $X_S$ the component transitions from the initial state $W_i$ to the unbuffered waiting state $W_u$. Once the data arrives from the source component it transitions again to $P_d$. Similarly, if the inputs events from the source model component and the notification
event arrive at the same time we transition directly to Pd. In both of the latter cases \( t(Pd) = t_{\text{delay}} \). Notice that communication components may also be used to buffer or model communication delays in the generation of the external inputs from experimental frame which is attached to the design model specification.

A more advanced version of a generic communication model component specification which also differentiates between “ready” and “busy” notification signals is attached in Appendix B.

4.8. An Example for a System Modeling and Simulation Environment for Embedded Systems Design

A number of discrete-event modeling specifications have been implemented in a suite of modeling and simulation tools. Major differences can be noted between tools developed in industry and academia. Many commercially available tools today try to top each other by the animation of model execution. This is of course an important aspect of a modeling tool if it is used to portray design ideas to higher management. Our position though is that it is the consistent model execution across tool releases which should deserve greater attention in model-based systems design.

We briefly introduce here an implementation of the modeling specification parallel DEVS called pDEVS [80] which was used for creating application design models and obtaining simulation results referenced in this dissertation. Notice that our model-based design approach, which will be discussed in the following chapters, could be similarly adapted to any other system level modeling specification, e.g., in a StateCharts-based approach.
4.8.1. DEVS Implementations

A variety of implementations of the DEVS formalism have been developed at the University of Arizona using the object oriented design approaches [41,79]. Object oriented software implementations allows a fairly simple component driven design of simulation engine and external event propagation for coupled model specifications. In these implementations the modeler builds user models by inheriting desired structure and behavior from the appropriate set of model base classes from which all models are derived. Model development is in some cases supported by graphic design entry but has usually a textual programming feel to it.

The first Java based implementation of the DEVS specification called DEVS-Java was developed by Zeigler at al [62,79]. DEVS-Java follows the design of previous implementations in Scheme and C++, and extended its scope by including new theoretical concepts of this formalism. For this dissertation, a new DEVS simulation engine was implemented by the author called pDEVS [80]. A reimplementation was necessary to simplify the use of DEVS for model-based design. New features of the software include hooks into the simulation engine for system model analysis and constructs for the mapping of DEVS models to mixed hardware/software implementations.

To simplify its implementation pDEVS was implemented on top of the modeling tool SILK [81]. SILK is an efficiently implemented, commercially available Java simulation engine which was originally developed process-oriented simulation. All of the DEVS simulation results presented in this thesis were obtained using the pDEVS simulation
The interested simulationist is welcome to download this simulation engine or more detailed documentation about the most recent version of pDEVS from our laboratory web site at http://www.ece.arizona.edu/~edl.

4.8.2. Modeling of the Structural Aspect

In pDEVS, design model components can be specified as one of two types: atomic models and coupled models. Atomic model components are considered the smallest entities within a system or application which is to be modeled. Its component structure is specified by its interface, i.e., a set of input and output ports. Atomic models are self-contained components which control and react to their environment by communicating via ports. Information can only be retrieved from completely modular model component specifications by sending messages to its port interface.

Coupled models describe a network of multiple atomic, coupled model components, or a blend of both, which are contained and interacting within a design model. The component structure of a coupled model is specified by its interface (a set of input and output ports), member model components (atomic and/or coupled models), and their
connections, more commonly referred to as couplings, with each other. Coupled model components at the lowest level of the design model hierarchy can only contain atomic models.

Coupled model components can be used to describe composed models in a hierarchical manner. Note that we distinguish between a coupled model which refers to a coupled model component, and a composed model which refers to the entire model consisting of multiple atomic and coupled models. The master coupled model is the coupled model specification situated at the top of the composed model hierarchy which is commonly referred to as the root node. Figure 10 shows two possible representations of a composed model where the “Design Model” is the master coupled model.

In pDEVS, the user has to specify the interface of atomic and coupled models in the respective model constructors. These constructors establish the external interface, i.e., model input and output ports, and the set of model states in the case of atomic model components. Instantiations of the two component types and their coupling are specified in coupled model constructors. An exception is the instantiation of the master model or the root node in the design model hierarchy which has specified separately for the simulation engine. Note that the master model may be an atomic model in the case of a single component simulation but more frequently a coupled model for composed model simulation.

4.8.3. Modeling of the Behavioral Aspect

In DEVS, the behavioral aspect of the design model is encoded in atomic model specifications. The specification of behavior for atomic model components can be split
into the identification of model states and state transition functions. Due to enforced modularity the behavior of each coupled model is implicitly defined by the behavioral specifications of its member model components, e.g., the coupled model state is the union of the model states of all its member components at any point in time.

As noted previously the notion of a state is important in any model specification and has to be handled with care. DEVS derives its notion of a state from its definition in general systems theory. Here, any physical system can be mathematically characterized by a set of system variables which change values as a function of time: input, state, internal, and output variables [18]. The system state at any specific point in time is defined by the values the system variables at that time. Therefore, a change in state always implies the passage of time. A DEVS atomic model state corresponds to such a system state. System or model behavior is defined by a set of states and state transitions in between these states.

Most DEVS implementations require the user to specify the behavior of atomic models in a model constructor, various transition functions, and an output function. In the constructor, all of the model states as well as an initial model state are specified, and component data structures are initialized. Behavior which is dependent on the arrival of events on specific input ports and their resulting model state changes are specified in the external transition function. Similarly behavior, which are not dependent on external input arrivals, or sequences of such are specified with their resulting model state changes in the internal transition function. The confluent transition function determines the order of model execution in the event that internal state changes should collide with an external
ones. Finally, outputs generated by computational segments are specified in the output function based on the processing model state they are associated with.

4.8.4. The Discrete Event Communication Model

Discrete event modeling specifications generally endorse an asynchronous communication model. Model components always execute if they receive message and a corresponding model state transition is defined, and continue execution after they produce outputs. If so desired, synchronous communication can be achieved based on asynchronous communication. Communicated events are propagated instantaneously.

Point-to-point communication is specified in coupled model specifications using model couplings which connect either member model output ports to member model input ports, coupled model input ports to member model input ports, or coupled model output ports to member model output ports. Therefore, all ports are uni-directional. Bidirectional ports can be modeled using port pairs, input and output port. The content of the information flow is not restricted in any way and includes events, basic, and complex data types.

In the specific context of embedded systems design, communication between computational segment of the same component can be implemented by using model variables or parameter passing. Outbound communication to other model components is specified in the output function. Outputs can only be generated during a model state change. Inbound communication from other model components is processed in the external transition function. In this function the user has access to the elapsed time in the current model state.
4.8.5. The Passage of Time

Arguably an important feature of DEVS is the ability to express timing explicitly in specifications of the behavioral aspect. Timing constraints are specified using timeouts, which in the DEVS terminology are referred to as the "time advances". The time advance function associates a timeout for each model state which may be infinite. In DEVS implementations, time advances are specified at the end of each external or internal state transitions in conjunction with the next model state.
5. DESIGN MODEL REFINEMENT

The objective of design model development is to generate a detailed, valid design specification from abstract application models. For that purpose, we have been advocating an iterative model refinement and evaluation process which results in a gradual increase in the resolution of the design model specification.

In this chapter we present design model refinement first from a practical and then from a more theoretical perspective. The practical perspective discusses the refinement of embedded systems design models mainly by the introduction of various system requirements while the following theoretical discussion focuses on the correct derivation of refined design model specifications.

5.1. Design Model Refinement in Model-Based Codesign

The iterative refinement of the design model specification is the most time intensive part in model-based codesign. Our general, formal discussion of design model refinement shows though that we may automate many aspects of the refinement process and significantly reduce the time spent in this part of the application design.

During a design model refinement, we gradually increase fidelity of the structural and behavioral aspects of the design model specification based on the introduction of application requirements as well as information obtained in an extensive design model analysis which will be discussed in the following chapter.

The refinement of the structural aspect encompasses the decomposition of already specified design model components or the addition of new components to the design model. It is either achieved by the introduction of structural requirements, e.g., some
specific parts of the behavioral aspects have to execute on the same processor, or a refinement of the behavioral aspect.

The refinement of the behavioral aspect is achieved by either introducing additional behavior, modifying already specified behavior, or increasing the resolution of model component behavior. The introduction or modification of behavior is either caused by the introduction of behavioral requirements, i.e., functional requirements with timing constraints, or the discovery of incorrectly specified behavior in a design model analysis. An increase in model resolution of some model component behavior can also be driven by a failure to conform to specified performance constraints. A more likely cause is the revision of an estimated instruction count for a given computational segment, or the decomposition of a given computational segment into two segments as a result of a design model performance analysis.

The end result of design model refinement is a design model prototype which incorporates all requirements stated in its system specification of our application and where each component is specified at the appropriate level of resolution. Here, the behavioral aspect of each model component specification should have all abstractly specified computational segments replaced by detailed descriptions of information flows.

5.2. Forms of Design Model Refinement

From a practical perspective we can distinguish between four forms of refinement in design model development: communication, structural, functional, and timing refinement. Here, communication and structural refinement modify the structural aspect of a design model specification while functional and timing refinement modify the behavioral aspect.
5.2.1. Communication Refinement

This most important type of refinement in model-based design refers to the refinement of model component interfaces. The level of detail in the modeling of design component communication is one measure of design model fidelity: Ultimately it is the external interface of our design model which needs to operate at an appropriate level of detail in order to allow the integration of our application into its real-world environment.

During design model development, event-based communication and abstract data types are refined to a message-based communication only composed of basic data types, i.e., bits/enumerated types (signals), or a composition of such in the form of characters or integers. Since the model component structure is defined by its external interface, a communication results in the change of the structural aspect of model component specifications. It also requires in an adaptation of component internal data structures, and component behavior since an update of the component interface is likely to require a modification in the processing of inputs and the generation of outputs.

A recommended approach is to initiate communication refinement by modifying the external interface of the entire design model first. These changes can then be cascaded through all of its components. This approach warrants a most efficient pruning of the application design space since it stresses the importance of the environment on the application design.

5.2.2. Structural Refinement

This type of refinement includes all other modifications of the structural aspect in the design model specification. It encompasses the decomposition of a model component in
multiple smaller model components. Such decompositions can be either identified directly from structural application requirements or indirectly from an analysis of the design model performance (which we will discuss in the next chapter). The worthiness of a structural refinement is strongly dependent on the data dependencies in the computational segments which are to be separated. This matter is discussed extensively in theoretical assessment of model refinement.

Structural refinement based on structural system requirements, e.g., the use of specific IP modules, available library elements, or hardware COTS (Components Off The Shelf [82]), is fairly trivial. Here, the designer isolates the computational segments or behavior from the corresponding components of the design model specification, and introduces a coupled model component specification for the known component - a second level in our hierarchy of the design model specification. The isolation of behavior is achieved by identifying this behavior within a component model specification and performing a correct structural refinement.

Notice that this new model component specification can be established as a grouping of multiple model components, i.e., a coupled model specification, which is composed of design model component specifications and communication components, i.e., level 1 model component specifications (see Figure 10). It may though be advisable to combine the isolated model behaviors in a single model component specification using the same model refinement techniques to reduce model execution time. Such a model component specification can then be considered "locked" since its behavior is completely specified from a component specification, e.g., a custom device specification.
Locked model components are excluded from the refinement process. Although a complete detailed behavioral specification of such components may be available, their behavioral specification should remain as abstract as possible. This will result in the most efficient model execution of the entire design model. We may still need to refine its communication interface (and as a result of it, its behavior to a minor extend) as our design model specification increases in fidelity. Locked model components simplify the refinement process but also limit the available design space for a given application since they restrict the possible exploitation of concurrency inherent in its behavior.

Application behavior which requires complex computations is also likely to result in a structural refinement of a design model specification. Here, advanced mathematical functions or algorithms which require, for example, the use floating point numbers require special attention. They are likely to be candidates for later realizations on a custom processor or other kinds of implementation optimizations since simple processor cores support only basic integer-based computation.

Global functional requirements can decompose model components located near the external interface of the design model. For example, one application requirement may state our application interacts with other computing systems via a specific bus. This implies decomposition of a system design model into a bus interface component and a component which handles the remaining computation.

5.2.3. Functional Refinement

A functional refinement of model components is triggered by the introduction of behavioral (or functional) application requirements, the modification of existing
computational segments, or by the necessity to adapt a model component specification to a communication refinement. Similarly, a functional refinement may trigger communication refinement. Notice that we have used the term "behavior" as a conjunction of its functional and dynamic aspect. Therefore, we refer to this kind of refinement as functional refinement. Nevertheless any functional refinement has to be followed by a timing refinement in order to constitute correct model refinement.

A functional refinement of abstractly specified behavior may be appropriate after the introduction of a behavioral requirement or new computational segments. Here, either an abstract model state in a model component specification is split or extended it with a new model state. The resulting behavior may be specified at any level of resolution or abstraction as long as it uses or extends the existing external interface of the model component specification properly.

Functional refinement also encompasses an increase in model resolution, i.e., the transformation of abstract descriptions of computational segments into more detailed descriptions. Commonly, a communication refinement of a model component or changes in its data structures or the model component instruction count of a computational segment lead to an increase in model resolution. The more detailed we specify the computational segment of a model state, the better we are able to estimates to its instruction count.

The functional decomposition of already existing detailed computational segments is another kind of functional refinement. Here, the computational segment and its associated model state is decomposed into multiple segments and associated model states based on
the information flow and internal data dependencies within the original computational segment. Trivial candidates for such decompositions in detailed computational segments are procedure calls, conditional branching, or looping constructs such as "if", "switch", "while", and "for" statements. Kumar provides in [32] a more formal discussion about decomposing functions in smaller sequences of functions.

5.2.4. Timing Refinement

The purpose of a timing refinement is to preserve the external interface of a model component specifications in a design model refinement. The main objective is to increase the accuracy of required execution times of computational segments in model component specifications. This is in essence achieved by improving or revising instruction count estimates for computational segments associated with model component states. Ultimately, the best estimates are achieved once the model structure, i.e., the model component interface or its data structures, is based on basic data types (see Section 5.2.1). In such model specifications, computational segments operate at the highest achievable resolution at the system level. In this case, all instruction count estimates can be replaced by the actual model instruction counts.

The refinement of timing aspects is required after any functional or structural refinement. Functional refinement combined with timing refinement is equivalent to behavioral refinement. Timing refinement encompasses the revision of the required execution time for computational segments based on their instruction count and the model component cycle time.
A timing refinement following a functional refinement of an abstract computational segment requires a revision of the execution time of its processing state if its estimated instruction count changes. If the revised execution time is lower than its original the delay time of the associated communication component has to be adjusted if the segment produces external outputs. If the revised execution time exceeds the original execution time the respective model component cycle time has to be decreased which in turn affects again the delay parameters of associated communication components.

A revision of execution times in a design model specification is almost always necessary when additional behavior, i.e., computational segments, is introduced to one of its components. Here, timing refinement proceeds for any type of resulting component in the same way as discussed for a timing refinement of an abstract computational segment.

5.3. Theoretical Aspects of Design Model Refinement

Any design model refinement is only of value if it is performed correctly and preserves the behavior of the previous model specification. Therefore, a proper refinement requires also a careful inspection of internal data dependencies within the original model component specification and its external interface to other model components. This is best addressed in a theoretical discussion of design model refinement.

Helpful in a discussion of correct refinement is to extend the concept of an internal event in order to describe internal data dependencies between computational segment in the context a basic model component specification (which was introduced in the previous chapter). We will therefore first introduce such an extension and then continue with a presentation possible types of model components which may result from a correct
refinement of a basic model specification. Afterwards we extend the results to more complex model specifications.

5.3.1. Internal Data Dependencies and the Internal Event

Internal data dependencies are important to consider in a structural refinement of model component specifications. Data dependencies between computational segments are caused with a model specification by sharing common variables or data structures.

In systems modeling, a system or component model state is captured by the values of a set of system state variables. In our design model specification, this model state is defined by the values of the global data structures or variables in a model component specification prior to any model state transition. We will refer to these component data structures and variables from now on as the set V, where any subset of V is some specific part of these data structures. Each model state \( s \in S \) in a model component specification can be defined as a unique assignment of values to the set V.

In systems modeling, we use model states and two kinds of model state transitions, external and internal, to specify the behavioral aspect of the application to be designed. External transitions are triggered if some specific inputs arrive from other model components. So far an internal transition between two model states has been explained as representation for a change in the values of system state variables in a model component which become valid after the generation of the "internal event".

In our design context, where processing model states are representatives for computational segments, internal events represent some access and possibly modifications to model component data structures. We may describe an internal event
more specifically as a collection of internal inputs $I_s$ and outputs $O_s$ (for some model state $s \in S$) which are "propagated" during internal model state transitions. Notice that this does not affect the concept of external inputs and outputs which are propagated as events between model component interfaces or absorbed in external model state transitions.

The concept of internal inputs and outputs allows a discussion of data dependencies within a model component specification: From a modeling perspective, each internal input and output represents an access to some specific state variable $v \in V$ in a model state. Similarly, multiple internal inputs and outputs represent the access to a set of system variables $V^* \subseteq V$. From a design perspective, internal inputs and outputs can be thought of as data or a specific part of the component data structure which is shared between computational segments of a model component specification.

Internal data dependencies can be classified by their temporal relationships as direct, near, or distant. Here, a direct dependency refers to a data dependency of a particular computational segment (or model state) to the computational segment (or model state) executed immediately before or after it. Similarly near and distant try to relate the access or modifications to some specific part of the model component data structures with respect to the current model state.

**5.3.2. The Basic Model Component Specification Revisited**

A revised basic model component specification, which has been modified by showing internal inputs and outputs, is illustrated in Figure 11. Notice that for the moment we ignore the communication components at its ports and we show a previous and next
model state. This revised model component specification allows a more detailed description of its behavior from a systems modeling perspective.

![Figure 11. Revised Diagram of a Basic Model Component Specification](image)

By definition, the waiting model state $W$ is either the initial model state or entered via some internal transition, i.e., from another previous processing model state $P$. This previous computational segment may have modified the values of component model state variables which are required by the following model states, i.e., they may have changed model component data structures. The purpose of $W$ is to collect external inputs required by the computational segment associated with processing model state $P$, and to convert them into internal outputs. We label this set of external inputs $X_w \subseteq X$, where $X$ is the set of all accepted inputs by the model component. Entering of $W$ also implies access to the corresponding part of the component data structure which will store the values of arriving input events. This is represented by the set of required internal inputs $I_w$, where $I_w \subseteq V$. This subset of the component data structure enters $W$ with their current values which will be overwritten by the values specified by $X_w$.

Theoretically, $W$ has an infinite execution time and an instantaneous internal transition to $P$ once all required input data for $P$ has arrived. The time spent in $W$ is likely to be a positive rational number but may be infinite. Since multiple external inputs may
arrive in an arbitrary order at different points in time we do not want to immediately transition to P at the arrival of just any external input. Instead we reenter W anytime a subset for $X_W$ arrives, turn these external inputs into corresponding internal data structure modifications $O_w$, and keep track of them in an arrival set $X_A$. When the $X_A$ matches the required set of external inputs $X_w$, i.e., $X_A = X_w$, an internal transition triggers the generation of internal outputs $O_w$, and the update of the same part of the data structure previously accessed by $I_w$ - but with different values, so that $O_w = I_w$ and new $V = V' = (V - I_w) \cup O_w$.

P can by definition only be reached via an internal state transition from W. The execution time of P is computed as previously discussed from instruction count of the its associated computational segment and the cycle time of the model component specification. Notice that the set of received inputs by P only consist of internal inputs. This set $I_p$ consists of some internal outputs of W and possibly other internal data, i.e., $I_p \subseteq V'$ and $O_w \subseteq I_p$. During its internal transition to a next waiting model state $nW$, P can generate a set of internal outputs $O_p$ which may be smaller than the set of internal inputs (the computational segment may have only read but not overwritten component data), so that afterwards the new $V = V'' = (V' - (I_p \cap O_p)) \cup O_p$ since $O_p \subseteq I_p$. At the same time a set of external outputs $Y_p$ can also be instantaneously generated from P. Modifications of the component data structure may enter the next basic model behavior again as internal inputs, i.e., $I_{nW} \subseteq V''$.
5.3.3. Correct Design Model Refinement

Both, structural and behavioral refinement, require a proper adaptation of the behavioral aspect in the generated model component specifications in order to constitute a correct refinement. A correct refinement is achieved when the external interface as well as former internal data dependencies of the original model component specification are preserved. In our discussion of stepwise refinement we consider first the behavioral and structural refinement of the elementary case, i.e., a basic model specification. Afterwards we extend results to the refinement of more complex model behavior.

A refinement of a basic model specification leads either to the decomposition of the original processing state $P$ into two processing states $P_1$ and $P_2$ (e.g., in the increase of model resolution) or the addition of a new processing state where $P_1 = P$ and $P_2 = \text{new } P$ (e.g., when extending behavior). Waiting states for each processing state are generated in the course of this decomposition. Each set of external inputs can be specified as $X_{w1} \subseteq X_W$, $X_{w2} \subseteq X_W$ and each set of external outputs as $Y_{p1} \subseteq Y_P$, $Y_{p2} \subseteq Y_P$, or $X_{w1} = X_W$, $X_{w2} = X_{\text{new } w}$, $Y_{p1} = Y_P$, and $Y_{p2} = Y_{\text{new } P}$, respectively. The selection of which of these two processes corresponds to $P_1$ and $P_2$ only matters in the case of a data dependency where we assume that the computational segment associated with $P_1$ alters data required by the segment of $P_2$. Notice that causality prohibits that $P_2$ may be at the same time also dependent on $P_1$.

In a behavioral refinement, we consider the case where the model states are kept in the same model component specification. Here, a correct refinement merely constitutes the preservation of external behavior which was specified for the original model
specification. In addition, we need to observe the single state condition in the resulting revised model specification which states that any model component can by definition only be in a single model state at a time. Data dependencies between the computational segments associated with the processing states do not directly affect the external interface since they are resolved internally in the model component.

In a structural refinement, processing states are assigned to different model component specifications. Here, the single state condition does not apply. Nevertheless, temporal constraints on the external inputs of the original model specification have to be respected and former internal data dependencies have to be addressed explicitly in the form of external event communication between the resulting model components.

5.3.4. Types of Resulting Model Component Specifications

From a theoretical perspective, a design model refinement can result in a number of correct refined model specifications. These resulting model components can be classified based on temporal relationships of external events arrivals in waiting model states, and data dependencies between the two processing states. Based on the temporal relationships in the original model specification we can distinguish between three different scenarios prior to a refinement:

a) Type I: Both external events arrive at the same point in time

b) Type II: The last event of the set of external inputs for W1 arrives prior to the one for W2

c) Type III: The last event of the set of external inputs for W1 arrives after to the one for W2
Since in both kinds of refinements the behavioral aspect is significantly affected by these temporal relationships we present the derivation of correct resulting model component specifications from the perspective of performing a behavioral refinement at first, and then continue with a discussion of a structural refinement. Since behavioral requirements usually outweigh structural ones in an application system specification of embedded systems applications we include also the feasibility a structural refinement step to follow a behavioral refinement for each scenario.

Any refinement of a model specification is accompanied by a revision of computational segments and instruction counts. Based on the required modifications required execution times for P1 and P2 have to be computed. In addition, the external outputs originally produced by P are now generated by either P1 or P2. These changes need to be also addressed in the resulting model component specifications.

Finally, we introduce the following notation to simplify the our discussion to some degree: "ta(s)" will refer to the execution time for the computational segment associate with the processing model state s, "ts" to the point in time that the model state s is entered, and "et(s, x)" to the total elapsed time in a model state s for the last external input x of the set X_s.

5.3.4.1. Type I Scenarios

A Type I scenario is depicted in Figure 12. Here, the temporal relationship between the external inputs which are to be received in the waiting state W1 and W2 of the refined model specification can been observed in the original model specification as et(W, x1) = et(W, x2), i.e., both of these external events arrive at the same point in time t_p.
In a pure behavioral refinement, we have to keep the same waiting state \( W \) which receives both sets of external input events instead of specifying two separate waiting \( W_1 \) and \( W_2 \) states to observe the single state condition as shown in the second part of Figure 12. The ordering of the processing states is either based on the data dependency between their computational segments or can be arbitrary if there are none (i.e., \( O_{P_1} \cap I_{P_2} = \{\emptyset\} \)). In the latter case an ordering may be established based on the generation of the former external outputs \( Y_P \). The generation external outputs at an earlier point in time allows to establish lower boundaries on delays of its associated communication.

A correct behavioral refinement is achieved when the execution times of the processing \( P_1 \) and \( P_2 \) are at most equal the original execution time of \( P \), i.e., \( \tau_a(P_1) + \tau_a(P_2) \leq \tau_a(P) \). If the computed execution time of both processing states exceed the original, i.e., \( \tau_a(P_1) + \tau_a(P_2) > \tau_a(P) \) we have to adjust the cycle time in the model component specification so that \( \tau_a(P_1) + \tau_a(P_2) = \tau_a(P) \). This has a considerable impact on more complex model specifications and its discussion is deferred for the moment to Section 5.3.6.
If the processing states either have an execution time less than the original model specification or produce external outputs part of the original set (i.e., $ta(P_1) + ta(P_2) < ta(P)$; $Y_{P_1} \subseteq Y_P$ or $Y_{P_2} \subseteq Y_P$ given that $Y_P \neq \{\emptyset\}$) then we modify the delay parameter of the communication component associated with these external output events. Proper adjustments of communication components will be discussed in more detail in Section 5.3.5. Otherwise we do not need to perform any modifications, e.g., $ta(P_1) + ta(P_2) < ta(P)$ and $Y_P = \{\emptyset\}$.

![Figure 13](image.png)

**Figure 13. Results from Structural Refinement based on data dependency in Type I Scenario**

In a structural refinement a new basic model component specification $B$ is created with an identical model cycle time as in the original model component specification. Waiting model states are specified for each component, i.e., $W_1$ and $W_2$. The required adaptation of each component behavior as depicted in Figure 13. Here, the original model component $A$ exhibits the behavior specified by $W_1$ and $P_1$, and component $B$ the behavior specified by $W_2$ and $P_2$. Notice that two cases arise from a possible data dependency of $P_2$ on $P_1$. The first part of Figure 13 illustrates the case that there is a data
dependency, i.e., $O_{P1} \cap I_{P2} \neq \{\emptyset\}$, while the second case, i.e., $O_{P1} \cap I_{P2} = \{\emptyset\}$, is illustrated in the second part of the figure.

A structural decomposition is obviously not advisable from a theoretical perspective in the first case since the behavior does not exhibit any potential for concurrent execution. In the second case, such a decision is advisable if there is no or a low degree of data dependency of $P2$ on data structures which are modified by previous processing states in the original model component $A$, i.e., the set of internal inputs $I_{P2^*} = O_{P2} \cap I_{P2}$ should be empty or small. $I_{P2^*}$ should be a small set since its values have to be communicated properly between the model components $A$ and $B$.

In the latter case the execution times for each processing state should be at most equal to the original processing time, i.e., $ta(P1) \leq ta(P)$ and $ta(P2) \leq ta(P)$. The same techniques are applied as in a correct behavioral refinement in Type I scenarios to adjust revised execution times for each component to fit that constraint. For example, if $ta(P1) < ta(P)$ we have to adjust its communication component if $Y_{P1} \subseteq Y_P$. Notice that if the execution time of the isolated behavior does not equal the original execution time, i.e., $ta(P2) \neq ta(P)$ when $O_{P1} \cap I_{P2} = \{\emptyset\}$, we can adjust the cycle time of component $B$ directly to $MCT_B = \frac{ta(P)}{IC_{r2}}$ since component $B$ is by definition a basic model specification.

If there are any data dependencies between $A$ and $B$, i.e., $I_{P2^*} \neq \{\emptyset\}$, we have to modify both model component specifications to reflect these dependencies properly. In component $A$, $P1$ has to generate the required subset of internal data for component $B$ in its set of external outputs $Y_{P1}$, so that the new $Y_{P1} = Y_{P1}' = (\text{values for } I_{P2^*}) \cup Y_{P1}$. The
waiting state W2 in component B needs to be modified in the same manner to also receive these former internal inputs as external inputs (i.e., I'w2 = Iw2 ∪ Ip2* and X'w2 = Xw2 ∪ (values for Ip2*)), and convert them back into internal outputs (O'w2 = I'w2). If component B changes data structures pertinent to component A, which is required by any of the following model states, i.e., Op2 ⊆ Inp, then these values have to be also communicated back properly, i.e., Y'p2 = (values for (Op2 ∩ Inp)) ∪ Yp2. Finally, the next waiting state nW has to be also modified accordingly in the model specification of A, i.e., new Xnw = X'nw = Xnw ∪ (values for (Op2 ∩ Inp)) and O'nw = I'nw. The sets Iw1, Ip1, Ip2, Op1, and Op2 do not have to be modified.

Notice that any change in the set of external inputs and outputs may also require modifications in the model structure of each participating model component specification, i.e., the creation of corresponding input ports, output ports as well as communication components, corresponding component connections, and data structures for Iw1 in component B.

5.3.4.2. Type II Scenarios

The results of a behavioral refinement (of the original behavior shown previously in Figure 12) in a Type II scenario is illustrated in Figure 14. Here, the temporal relationship between the external inputs, which are to be received in the waiting state W1 and W2 of the refined model specification, are observed in the original model specification as \( t(W, x1) < t(W, x2) \), i.e., in the original model specification the last external input for W2 arrives after the one for W1 at time \( t_p \) (see Figure 12). Generally we can perform the
same adjustments as during a behavioral refinement in a Type I scenario which is depicted as the general case in Figure 14.

![Diagram](attachment:image.png)

**Figure 14. General and Special Case of Type II Scenario Results after a Behavioral Refinement**

A different adjustment is possible (but not required) if the first input event of the set $X_{W_2}$ arrives after the last external event of the set $X_{W_1}$ and there is no data dependency between $P_1$ on $P_2$, i.e., $O_{P_1} \cap I_{P_2} \neq \emptyset$. This special case is shown in the second part of Figure 14 is though subject to an additional constraint based on $x_{2f}$ which represents the first event of set $X_{W_2}$. Here, the revised execution times of the processing $P_1$ has to be within the arrival of the $x_1$ and $x_{2f}$, i.e., $ta(P_1) \leq et(W, x_2) - et(W, x_{2f})$ and $ta(P_2) \leq ta(P)$.

If any of recomputed execution times should exceed these values, i.e., $ta(P_1) + ta(P_2) > ta(P)$ or $ta(P_1) > et(W, x_2) - et(W, x_{2f})$ and $ta(P_2) > ta(P)$, we have to adjust the cycle time in the model component accordingly (see Section 5.3.6). If the revised execution times should be less than these values or produce external outputs part of the original set (i.e., $ta(P_1) < et(W, x_2) - et(W, x_{2f})$ or $ta(P_2) < ta(P)$; $Y_{P_1} \subseteq Y_p$ or $Y_{P_2} \subseteq Y_p$ given that $Y_p \neq \{\emptyset\}$) requires again a modification of communication components (see Section...
5.3.5). Otherwise no further modifications are necessary. In general, the previous described approach for a pure behavioral refinement is the better choice. Model specifications which exhibit the behavior of the special case are better resolved by performing a structural refinement.

Figure 15. Results from Structural Refinement based on data dependencies in Type II Scenario

In a structural refinement, a new basic model component specification B is created similarly as in a Type I scenario with an identical cycle time as the original model component. The results of such a refinement are depicted in Figure 15. In the first case (i.e., \( O_{P1} \cap I_{P2} \neq \{\emptyset\} \)) a correct structural refinement is achieved when revised execution time for processing state A is within the arrival times of \( x_1 \) and \( x_2 \) while the execution time for \( P_2 \) should be at most equal to the original processing time, i.e., \( t_{a(P1)} \leq e_t(W, x_2) - e_t(W, x_1) \) and \( t_{a(P2)} \leq t_{a(P)} \). Again the same techniques are applied to adjust either the cycle time or respective communication components in the event that revised execution times exceed this constraint. Note that in the first case the isolation of behavior A instead of B may be more attractive since the cycle time adjusted most easily for basic
model component specification and the condition that ta(P2) > ta(P) is not likely to arise in most refinements.

In the second case (i.e., O_{P1} \cap I_{P2} = \emptyset) a correct structural refinement is achieved when the execution time for processing state of A is within the arrival times of x1 and x2 and the original execution time while the execution time for P2 should be at most equal to the original processing time, i.e., ta(P1) ≤ et(W, x2) - et(W, x1) + ta(P) and ta(P2) ≤ ta(P).

From a theoretical perspective, this second case is obviously the most attractive for a structural refinement since there is concurrency inherent to this computation. The communication of required data structures between components A and B is achieved in the same manner as described in the Type I scenario.

![Figure 16. General and Special Case of Type III Scenario Results after a Behavioral Refinement](image)

5.3.4.3. Type III Scenarios

The results of a behavioral refinement (of the original behavior shown previously in Figure 12) in a Type III scenario is illustrated in Figure 16. Here, the temporal relationship between the external inputs, which are to be received in the waiting state W1 and W2 of the refined model specification, are observed in the original model.
specification as $\text{et}(W, x_1) > \text{et}(W, x_2)$, i.e., in the original model specification the last external input for $W_1$ arrives after the one for $W_2$ at time $t_p$. A pure behavioral refinement in a Type III scenario follows basically the one of a Type II scenario. The general case follows again a Type I scenario refinement and for the special case indexes can be interchanged. Adjustments are done in the same manner.

In the same manner results of a structural refinement shown in Figure 17 follows the cases of Type I and Type II scenarios, respectively. Obviously a structural refinement is again most advisable if there is no data dependency between $P_1$ and $P_2$ (i.e., $O_{p_1} \cap I_{p_2} = \{\emptyset\}$).

![Figure 17. Results from Structural Refinement based on data dependencies in Type III Scenario](image)

5.3.5. Adjustment of Communication Model Components

In order to preserve the external output interface an adjustment of communication components is required in the course of a correct refinement. The need for such an intervention arises when $P_1$ or $P_2$ produce after a revision parts of the original external
output set $Y_p$ at a time prior to $t_p + \tau_a(P)$, i.e., if $Y_{p1} \subseteq Y_p$ and $t_{p1} + \tau_a(P1) < t_p + \tau_a(P)$ or $Y_{p1} \subseteq Y_p$ and $t_{p2} + \tau_a(P2) < t_p + \tau_a(P)$ given that $Y_p \neq \{\emptyset\}$.

In the original model component specification $Y_p$ has been send via a communication component with a delay time $\tau_a(P_d)$. After a refinement this component has either become associated with $Y_{p1}$ if $Y_{p2} \not\subseteq Y_p$, $Y_{p2}$ if $Y_{p1} \not\subseteq Y_p$, or split into two communication components, i.e., $\tau_a(P1d) = \tau_a(Pd)$ or $\tau_a(P2d) = \tau_a(Pd)$, or $\tau_a(P1d) = \tau_a(P2d) = \tau_a(Pd)$ respectively. In order to reflect decreases in execution time we need to increase the delay times of the communication components, which produce any part of original set of external outputs, so that either the new $\tau'_a(Pd1) = \tau_a(Pd) + ((t_{p1} + \tau_a(P1)) - (t_p + \tau_a(P)))$ or $\tau'_a(Pd2) = \tau_a(Pd) + ((t_{p2} + \tau_a(P2)) - (t_p + \tau_a(P)))$. Notice that decrease in the model component cycle time does not offer an alternative for the modification of individual execution times since it would also affect all other processing state execution times of that model specification.

5.3.6. Execution Time Violations in Modified Original Model Specifications

We have not yet discussed a proper adaptation in any refined model specification for the case where the revised execution time of the processing states $P1$ or $P2$ exceed the value specified for $P$ in the original model component specification, e.g., $\tau_a(P1) + \tau_a(P2) > \tau_a(P)$ after a behavioral refinement in a Type I scenario. This situation may also arise after the modification of the instruction count for an already existing computational segment. The latter scenario can be interpreted as a behavioral refinement of the original model
specification where \( P2 \) has an estimated instruction count of zero, receives no inputs, and generates no outputs, i.e., \( I_{P2} = \{ \emptyset \} \) and \( ta(P2) = 0 \).

The action to be taken in such a case is to revise the model cycle time of the model specification properly so that the violated condition hold, e.g., \( ta(P1) + ta(P2) = ta(P) \). As a matter of fact this approach has been suggested previously for violations in newly created components \( B \) a structural refinement: We compute the cycle time of the model component based on the respective instruction counts so that it our violated condition holds, e.g., \( MCT_i = \frac{ta(P)}{ICr_i + ICr_j} \).

An adjustment of the cycle time of model component \( A \) though requires - in contrast to the case of a basic model specification of component \( B \) - additional changes once the behavioral complexity of the original model component has a larger number of component model states, i.e., \( |S| > 2 \). When we change the model cycle time not the execution time for, e.g., model state \( P1 \) and \( P2 \), is affected but also execution times of all other processing model states part of this model specification, e.g., \( P3, P4, \ldots, Pn \). Without proper compensation the external interface of this model component specification is thrown off at other places if merely one of these other processing model state produces a single external output.

The only way to guarantee a correct refinement in this case is to adjust the other communication components which are part of this model component specification. The required change in each delay time parameter \( ta(Pid) \) is calculated from the difference of the execution time for the processing state \( Pi \) in the design model component specification, which produces the respective external outputs, with the original model
cycle time \( t_a(P_i) \), and the revised one with the new model cycle time \( t_a'(P_i) \), so that the new \( t_a(P_i) = t_a'(P_i) = (t_a(P_i) - t_a'(P_i)) + t_a(P_i) \). Notice that this difference is always positive since a modification in the model cycle time can only result in a smaller revised execution time. Notice that the external input interface is not directly influenced by modification of the model component cycle time. The only change is that the model specification may spend more time in waiting model states to receive external inputs.

5.3.7. Correct Refinement of More Complex Behaviors

In the course of design model refinement the behavioral aspect for each model component specification obviously grows in complexity. With each refinement step more model states and model state transitions are introduced. But as we also saw in the previous chapter complex behaviors are merely compositions of basic model component behaviors due to our construction of the design models.

Therefore, a refinement of complex behavior can be simply performed on one processing state at a time. In a behavioral refinement, we select a basic behavior from a complex model specification, refine it, and place the refined behavior back to its original location. In a structural refinement, we also proceed in the same manner as for a basic model component specification. This is of course only possible if we follow changes specified for a correct refinement where the external interface as well as internal data dependencies are respected in each refinement step.

In the structural refinement of more complex model component specifications we may improve our communication of data dependencies, i.e., when \( I_{P_2}^* \neq \varnothing \). Internal data has so far been communicated in the form of external outputs from the previous
processing state pP. In some cases the last processing state which manipulates this required part of the data structures in component A is not the immediately previous state but an "earlier" state. In more complex model specifications, we can move the generation of the respective outputs back to the state pP* which is last to modify an element of the set Ip2*. Notice though that such a modification also requires a different adaptation of the delay parameter in the associated communication component, i.e., ta(pP*d) = ta(pPd) + Σ tp where Σ tp represents the sum of all processing state execution times specified in between the processing model state pP* and the original pP.

Similarly, the subset of internal outputs Op2 of the model state P2 may also not be required by the immediately next processing state nP in the original model component specification. In that case we can modify the delay time of the communication component which was created for the communication if relevant internal outputs O*P2 of P2, i.e., O*P2 = Op2 ∩ Iw, so that the new ta(Pd2) = ta'(P2d) = ta(P2d) + Σ tp0 where Σ tp0 represents the sum of all processing state execution times in model component specification A between its next model state nW and nW*.

At this point it may seem that during model refinement delay times of communication model component specifications can only increase which would ultimately result in design model specification with large communication delays and small model component cycle times. But as we shall see in the next chapter we can apply communication analysis methods to also adjust delay times in the reverse direction.
6. DESIGN MODEL ANALYSIS

This chapter introduces analysis of design models which is performed after each refinement step. Here, we evaluate static model information, which is encoded in the design model specification, as well as dynamic information, which is generated during the execution of the design model in a simulation run. We discuss analysis methods which can be used to assess a refinement, the validity of the design model, its performance, or to suggest future refinement steps.

![Figure 18. Static versus Dynamic Model Information](image)

6.1. Design Model Evaluation

In relevant design model data we can distinguish between data encoded in the structural and behavioral aspect of the design model specification called static model information, and data obtained from the execution of the model specification called dynamic model information. Dynamic model information includes, in addition to the data generated by
design model specification, information generated by the underlying modeling specification itself (e.g., the scheduling of model components). Figure 18 shows some examples for static and dynamic model information.

We can retrieve static information by parsing the entire design model specification. Pertinent dynamic model information is collected from the simulation of the design model. This information can be retrieved by placing hooks into the simulation engine and connecting experimental frames to the design model which stimulate and monitor its external behavior over time. Alternatively we can apply formal verification techniques to retrieve the same dynamic model information directly from the design model specification.

6.1.1. Simulation versus Formal Verification

Both approaches for dynamic model information retrieval have benefits and drawbacks. Since simulation is in practice nothing more but a program execution it allows a straightforward application for system designers who have some background in software development. In a simulation-based testing (or debugging) of the design model, we can encode multiple input stimuli for a design model (or even provide them interactively [77]) and special model components which analyze its response already during model execution. This forces the designer to actively participate in the construction of these test cases. This is desirable in the development of new embedded systems applications where there are no other real-world systems but only an informal system specification against which we can compare our design. The drawback of a simulation-based design assessment is that the required model execution time for a simulation grows rapidly with
design complexity. In addition, a validation of a design model is only as good as the test scenarios applied in the simulation. The latter issue requires a structured approach for the development of such test scenarios which has been addressed in the context of our model based codesign methodology in [66].

Formal verification methods can be used to guarantee properties about a design and offer a great potential for the automation of design model assessment [83]. Automation naturally allows for a faster design validation. But even the execution time for verification software increases with design complexity but usually at a much slower rate than for simulation. Drawbacks here are that it is often only possible to verify a design specification for a single input at a time. Testing is fixed in that it does not allow interactive inputs. In our design context, a formal verification approach also gains complexity since the retrieval of dynamic model information requires the observation of both, the design model specification as well as the underlying the modeling specification itself, in a design evaluation.

6.1.2. Validation in Model-Based Codesign

We advocate simulation for design model validation in model-based codesign since we are mainly interested in monitoring and collecting a wide spectrum of model information from our design model. Formal verification techniques require complex tool implementations and an intimate understanding of formal verification techniques by the designer. In addition, too much automation is not really desirable in application development [52]. It tends to separate the designer from the design model analysis which is essential in the refinement and modification of the design model. Formal verification
techniques are in our opinion better suited for verification at implementation level. Here, we can use formal verification techniques to verify our design implementation against the formal, detailed design model specification.

6.2. Design Model Analysis Methods

The analysis of design models serves a twofold purpose: Functional analysis methods are used to validate a design model in respect to its functional requirements. Performance analysis methods examine then how our application design model specification responds to various environmental stimuli. Functional analysis is fairly trivial in its application and commonly practiced in system modeling today.

Performance analysis at the system level has only attracted little attention to date. In our model-based codesign methodology these methods are used to derive a processing platform for an application behavior and to determine structural refinements which improve design model performance. Different performance analysis techniques are applied in during design model refinement and model compilation in our design methodology. We focus in this chapter on methods which can be applied during design model refinement.

6.2.1. Functional Analysis

Functional analysis is probably the most common form of analysis in current system level design approaches. It analyzes the generation and propagation external events in the design model or its member model components. Such observations mainly focus on a correct sequence and values in the application response where time is mainly establishes
an ordering of external events. The collected information is evaluated to determine if it meets stated system requirements for a particular application. We ask the following types of questions about our design: Does the design model produce a specific set outputs for a given set of inputs in a correct sequence? Does the design model exhibit the behavior of the application at the required level of resolution? Which model component fails to service requests? Are there any unhandled external input events in any model component?

6.2.2. Performance Analysis

Performance analysis is only indirectly concerned about the behavior of the design model and requires a more intimate knowledge of the underlying modeling specification. The results of a performance analysis attempt to detect the performance limits or possible improvements for our application design from execution profiles of the model specification. It enables the designer to assess information that is usually not available to a designer at the system level, e.g., communication cost, data dependencies, behavioral bottlenecks, predicted response times, etc. Performance analysis is applied most effectively once the design model has achieved a fairly low resolution.

The basis for this type of analysis is the interpretation of our modular design model specification as an application running on an ideal, flexible architecture consisting of custom processors (which are represented by simulator components associated with each model component specification [41]). In our initial design model we are likely to have no knowledge at all about this virtual processing platform. As we refine each model component though we gain more insight to the underlying architecture. Performance
analysis methods provide in essence a means for the designer to indicate and suggest proper refinements to improve for example the utilization of this virtual processing platform.

For single design model component specifications we ask the following types of questions: Which computational segments are the processing bottlenecks in a particular model component? How can these bottlenecks be most appropriately compensated for? Is any further structural refinement advisable for a model component? Should we increase model component resolution? Is there any concurrency to be extracted from the model specification? What is the best model component cycle time for each component?

An analysis of the composed design model specification can also be used to assess previous or to suggest further refinements. Here, we can add the following types of questions: Which components are the bottlenecks in the design model? How can these bottlenecks be most appropriately compensated for? What are the required data rates for each communication channels? Which model components actually execute concurrently in different operating conditions?

6.3. Analysis Methods for Design Model Refinement

It is difficult to select one single classification scheme for all analysis techniques which are applicable for design model refinement. Some analysis methods always wind up qualifying for more than one category. We present in this section some rather non-traditional functional analysis techniques which we developed specifically for model-based codesign. They are intended to guide the refinement of behavioral and structural aspects of the design model. The following section will then introduce analysis methods
which assess the design model performance directly. In both sections the design model specification is assumed to be constructed based on the model of computation which was introduced in the previous chapter.

6.3.1. Static Model Component Data Analysis

This first analysis method provides the basis for establishing data dependencies within model component specifications. Here, we gather information about the data received in each waiting state, and establish also data tables for processing model states. The results of this data analysis can be used, e.g., to assess model component resolution or level of detail of a computation segment specification.

For each waiting state we create a list of the global data items it updates. These data items can be easily identified since they have to be declared as global model component variables in an implementation of a model component specification.

For processing states we tabulate each required data item as shown in Table 1. Local data items are shown with a model state name prefix. Here each data item is listed with its size in bits, a classification of read or write access (where a write access is to be selected in the case of the occurrence of both), as well as an approximate location of its first access (FA) and last access (LA) in the computational segment. The later measure is based on the current estimate of its instruction count (where for any data item FA(data item) ≤ LA(data item)). A first access to a particular data item at model instruction 10 of 100 would result in FA(data item) = 10/100 = 0.1. In abstractly or partially abstract segments, it can be assumed FA(data item) = 0 and LA(data item) = 1 (or estimated by the designer). For data items accessed within looping or conditional constructs we use the
starting location of the loop body for its first access reference and the end for it as its last access reference.

<table>
<thead>
<tr>
<th>Model State Name</th>
<th>Data Item</th>
<th>Size</th>
<th>R/W</th>
<th>FA</th>
<th>LA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComputeTime</td>
<td>hour</td>
<td>8 (char)</td>
<td>W</td>
<td>.7</td>
<td>.8</td>
</tr>
<tr>
<td>ComputeTime</td>
<td>min</td>
<td>8 (char)</td>
<td>W</td>
<td>.5</td>
<td>.6</td>
</tr>
<tr>
<td>SetTime</td>
<td>setTime.digit</td>
<td>2 (boolean)</td>
<td>W</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>SetTime</td>
<td>setTime.blink</td>
<td>16 (int)</td>
<td>R</td>
<td>0.4</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1. Sample Data Table for Processing States of a Model Component

6.3.2. Static Model Component Behavioral Analysis

This type of analysis can be used to identify model states within model component specifications which qualify for a functional refinement in the context of an increase in model resolution or a structural decomposition. This form of analysis requires the availability of data tables for each model state (see Section 6.3.1) and model state transitions of the respective model component specification. It evaluates the impact of some selected model states on the rest of the model component specification and suggests possibilities for further refinement. The model states which are to be evaluated are assumed to be selected either by the designer or a result of another analysis method.

For a possible increase in model resolution we evaluate data dependencies within a model state (or computational segment) to check. This is achieved by parsing the data tables generated for the corresponding processing state, i.e., sets of data items A and B for which LA(any data item of set A) > FA(any data item of set B), and identifying disjoint computational segments within a single existing computational segment.
Computational segment with disjoint local data sets may then be split at appropriate locations, i.e., between after the last access to a member of the set A and the first to a member of B. If an increase in resolution should be required for some specific segment a location with the least amount of access overlap can be suggested. Notice that data tables change after such a refinement since some local data items will become global, i.e., data items which need to be propagated as internal outputs and inputs between the derived segments. A correct behavioral refinement is achieved based on the input arrival scenarios described in the previous chapter.

\[
\text{data transferal cost} = 0; \\
\text{for each global data item of model state} \{ \\
\quad \text{multiplier} = 1; \\
\quad \text{if ( write access is true ) multiplier} = 2; \\
\quad \text{data transferal cost} = \text{data transferal cost} + (\text{multiplier} \cdot \text{size of data item}) \\
\} // \text{end for}
\]

Figure 19. Algorithm for Computation of Required Data Transferal

The analysis of model component behavior is also of interest for a structural decomposition. An analysis of the waiting state tables can be used to determine waiting states which accept the same set of external inputs. A further examination of the processing states may reveal matches in the following processing state behavior which can then be isolated in a separate model component. As discussed previously a structural decomposition is most effective if there is only little shared data items between such processing states. The isolation of the model states can based on the estimated data transferal cost (a simple example algorithm is shown in Figure 19). In practice, the
designer can limit the data transferal cost to some maximum amount for which a structural decomposition is still acceptable.

6.3.3. Model Component Communication Analysis

This form of analysis assesses the communication complexity as well as fidelity of a model component specification. Here, we examine the static and dynamic model information encoded in each communication component model specifications.

For these model component specifications we can determine the amount of data transfer from and to a specific model component using the same static model component analysis techniques as used for regular design model components (see Section 6.3.1). From this information and delay time parameters we can deduce required size and amount of transfer data over time for each communication channel.

Large amounts of data transferals in communication channels indicate performance bottlenecks in the design model which may have been caused by an inefficient structural refinement. Here, receiving and sending design model components should be reinspected using a Static Behavioral Analysis to evaluate if the required communication could be reduced by transferring parts of one model component behavior to the other component.

In addition, we can monitor the model state of each communication component over a simulation run for occurrences of the buffered waiting state Wb. If the time spent in Wb exceeds at any point the specified delay parameter an improper behavioral refinement has been performed in the receiving model component. Here, a proper adjustment of the execution time for the processing state, which the component is in prior to receiving these inputs, is required. The cycle time of the receiving model component specification has to
be decreased so that the revised execution time reduces this difference to zero. Notice that a revision of the model component cycle time also requires an adjustment of the delay parameters in associated communication component which has been discussed in detail in the previous chapter.

6.3.4. Composed Design Model Communication Analysis

In the course of timing refinements we may decrease model component cycle times and increase delay parameters of communication components frequently to preserve the external interface of each model component. In a composed design model, such increases in delay parameters can be forwarded to their destination components. As we shall see in the next section this eventually enables us to compute increases in model component cycle times.

Based on the arrival of external inputs in the waiting states of a given design model component we adjust delay parameters of communication components which sent these events. While we adjust delay parameters we take the opportunity to synchronize the arrival of multiple events in each waiting state to a single point in time. This adjustment simplifies the following model compilation phase significantly.

In a composed design model communication analysis we focus first on the evaluation of waiting model states in the member components of the design model. Here, we record for each a waiting state $W_i$ the elapsed time $et(W_i, x_k)$ for the arrival of each external input $x_k \in X_{W_i}$ (or set of input events $X_k \subseteq X_{W_i}$) as well as the delay time of the communication component which produces this event $t_{delay}(x_k)$. A composed model analysis is only feasible if the time spent in $W_i$ is greater than zero, i.e., $\exists x_j \in X_{W_i}$ for
which $\text{et}(W_i, x_j) > 0$. The waiting state $W_1$ in the model component behavior shown Figure 20 will be used as an example in the following discussion with $\text{et}(W_1, x_1) = 2$, $t_{\text{delay}}(x_1) = 4$, $\text{et}(W_1, x_2) = 4$, and $t_{\text{delay}}(x_2) = 1$.

As we mentioned earlier an appropriate time $t_{\text{sync}}$ for the synchronization of external event arrival is of first goal in an adjustment of delay time parameters. This point in time is determined from the largest valid adjustment for any of the external input arrival, i.e.,

$$t_{\text{sync}} = \text{et}(W_i, x_j) - t_{\text{delay}}(x_j)$$

where $x_j \in X_{Wi}$ and $\forall x_k \in X_{Wi} \text{ et}(W_i, x_j) - t_{\text{delay}}(x_j) \geq \text{et}(W_i, x_k) - t_{\text{delay}}(x_k)$. For each external event $x_k \in X_{Wi}$ we then revise the delay parameter $t_{\text{delay}}(x_k)$ of the communication component, which produces the corresponding event so that each $x_k$ arrives at time $t_{\text{sync}}$. If the computed $t_{\text{sync}}$ is less than zero we adjust the each delay time parameter by respective the value of its elapsed time $\text{et}(W_i, x_k)$. Formally we can describe these adjustments as follows: if $t_{\text{sync}} > 0$ then $\forall x_r \in X_{Wi}$ where $\text{et}(W_i, x_r) \geq t_{\text{sync}}$ new

$$t'_{\text{delay}}(x_r) = t_{\text{delay}}(x_r) = t_{\text{delay}}(x_r) - (\text{et}(W_i, x_r) - t_{\text{sync}})$$

and $\forall x_s \in X_{Wi}$ where $\text{et}(W_i, x_s) < t_{\text{sync}}$

$$t'_{\text{delay}}(x_s) = t_{\text{delay}}(x_s) + (t_{\text{sync}} - \text{et}(W_i, x_s))$$

otherwise if $t_{\text{sync}} \leq 0$ then $\forall x_k \in X_{Wi}$ $t'_{\text{delay}}(x_k) = t_{\text{delay}}(x_k) - \text{et}(W_i, x_k)$. In our previous example we compute $t_{\text{sync}} = \text{et}(W_2, x_2) - t_{\text{delay}}(x_2) = 3$ for $W_1$ since $\text{et}(W_1, x_2) - t_{\text{delay}}(x_2) > \text{et}(W_1, x_1) - t_{\text{delay}}(x_1)$, and revise the delay.
parameters accordingly to $t'_{\text{delay}}(x_1) = t_{\text{delay}}(x_1) + (t_{\text{sync}} - \text{et}(W_1, x_1)) = 4 + (3-2) = 5$ and $t'_{\text{delay}}(x_2) = t_{\text{delay}}(x_2) - (\text{et}(W_1, x_2) - t_{\text{sync}}) = 1 - (4-3) = 0$.

If any of the immediately following processing states (in our example P1) produces external output events the delay time parameter of its associated communication component(s) has to be also adjusted properly, i.e., increased by the maximum value of $t_{\text{sync}} - \text{et}(W_i, x_r)$. In our example this adjustment would be $t_{\text{sync}} - \text{et}(W_1, x_1) = 1$. We repeat the same analysis for all waiting states of the composed design model specification.

### 6.3.5. Model Component Cycle Time Analysis

So far we have only discussed decreases in the model component cycle time as a result of some form of a model component refinement, i.e., behavioral refinement, structural refinement, or a revision of estimated instruction counts. Ultimately though it is the highest possible model cycle time (or lowest processor speed) that is of interest for the generation of an efficient design implementation from a model specification. A model cycle time analysis provides us with a means to increase the value of cycle time for design model component specifications.

In a cycle time analysis we evaluate design model components individually. Increases in model component cycle times are possible if when a common minimum delay time for the communication components which associated with a particular model component can be detected. Since a correct timing refinement always results in at least one zero delay parameter in an associated communication component, a composed design model
communication analysis (see previous section) may be required prior to model component cycle time analysis.

We determine first the minimum delay time $t_{\text{min}}$ of all communication components which are associated with the same design model component. If $t_{\text{min}} > 0$ we determine the processing model state or a proper sequence of such in that model component specification which has the largest total instruction count $IC_{\text{max}}$. Here, a proper sequence constitutes a set of processing states $SEQ$ where only the last state in the sequence generates external outputs, i.e., for any $P_i \in SEQ$ only $Y_{\text{last } P_i} \neq \emptyset$. An example model state simulation trace of a model component containing a sequence of such processing states is shown in Figure 21 where $SEQ = \{P_1, P_2\}$ and $IC_{\text{max}} = IC_{P_1} + IC_{P_2}$. Notice again that we assume here that $SEQ$ has the largest instruction count in this model component specification (e.g., $IC_{P_3} < IC_{P_1} + IC_{P_2}$).

The best cycle time ($MCT_{\text{best}}$) is then computed from the current cycle time ($MCT$) as shown in equation (3). An increase in model component time (when $MCT_{\text{best}} > MCT$) requires the similar modifications as decreases in such. First of all every processing state

![Figure 21. Example Execution Trace for a Model Specification with a Processing State Sequence](image-url)
execution time needs to be updated based on this revised model component cycle time. In addition, the delay time parameters of associated communication components have to be adjusted accordingly.

\[
MCT_{\text{ben}} = \left(1 + \frac{t_{\text{run}}}{IC_{\text{run}} \cdot MCT}\right) \cdot MCT
\]  

(3)

6.4. Design Model Performance Analysis Methods

Contrary to previously discussed analysis methods, performance analysis methods examine the design or application model in different operating conditions. A performance analysis focuses on a more detailed study of design model behavior which based on the values and order of arrival in different operating scenarios. A model state execution trace over the simulation run serves again as a basis for a performance analysis.

Next to an evaluation of the design model over time, a performance analysis can suggest further refinements for its member components. In general, we can distinguish in these methods between traditional systems modeling evaluation methods which assess the system model response over time (e.g., throughput, turn around time, response time), and methods specific to model-based codesign which assess of the utilization of design model components. We will focus in this section on latter type of performance analysis methods.

A performance analysis obviously provides best results after all abstract computational segments have been replaced by detailed descriptions. As we will discuss in the following chapter, the results of a performance analysis are especially of interest when the required processing power for our design model specification exceeds the
available processing power of a given target architecture. Here, we may improve the performance of our design model specification by recommending a proper structural refinement.

6.4.1. Model Component Utilization Analysis

A model component utilization analysis focuses on individual model component specifications which are part of our design model. Previously, the model component cycle time has been introduced based on the concept that each model component specification is understood as an application executing on a dedicated virtual processor. Model component utilization can therefore be assessed by measuring the time spent in waiting as well as processing model states, i.e., executing its computational segments. Secondly, we introduce the concept of computational intensity of the model states which measures the number of executions of computational segments in simulation run.

We measure the idle time of a virtual processor by accumulating the total elapsed time in each component waiting state during a simulation run. Based on the recorded values we can reason about the degree of utilization in a model component specification, e.g., components with idle times close to zero exhibit high utilization. A high degree of variance in the recorded elapsed times indicate that the model component cycle time may be decreased by a proper structural decomposition. Here, computational segments with the smallest idle times are of interest for an isolation in a separate model component specification. Similar as in Section 6.3.2, structural decompositions are only advisable if the required data transfer is manageable.
An analysis of processing model states in the composed model execution trace provides more information about the computational intensity of computational segments in a design model component. The computational intensity is computed from the number of occurrences of a particular processing state in a simulation run and its specified execution time.

### 6.4.2. Composed Model Utilization Analysis

A composed model utilization analysis focuses on the performance of model components within a composed design model specification over a set of performance critical simulation runs. Indicators for a possible performance improvement of such a model specification can be determined from model component cycle times and a comparison of overall computational intensities of particular model component.

Performance bottlenecks are located within model components which have either a small model component cycle time or a high total computational intensity which we will simply define as the sum of computational intensities of all its processing model states. After the identification of performance critical design model components we can apply other model component analysis methods, e.g., a model component utilization analysis, to determine an appropriate structural decomposition for this component.

Alternatively, design model performance can be improved by applying the two well-known processor design techniques of component duplication or pipelining to candidate model components at the system level. The appropriate choice in the latter approach depends in general on the required transferal of external data to or from the candidate model component which can be determined using previously introduced analysis.
techniques. The required amount of data transferal can be computed using the algorithm illustrated previously in Figure 19.

If the data transfer is fairly small then a duplication of this model component is most appropriate. In theory, a duplication results in revised cycle times with twice the value of the original cycle time in the two resulting model component specifications. In practice however, this solution may require the creation of an additional model component which administers the dispatching and receiving of these external data and introduces some additional processing overhead.

If a large amount of data is received by the original component we may consider a structural decomposition or pipelining of the original model component behavior. Such a proper decomposition has been described in the previous chapter for different types of event arrival scenarios. Here, the adjustments of the model cycle time for each model component generally increase its value but are dependent on the largest computational segment part of the respective component specification.

6.5. Automation of the Design Model Refinement and Analysis

Since the design model refinement and analysis phase is clearly the most labor intensive part in model-based codesign, the automation of this design phase is an important issue. Since refinement of the behavioral and the structural aspect has been completely formally specified we can easily automate the actual refinement steps: Here, we write a model component parser program (which establishes for example model state data tables), or a program which realizes a proper timing refinement during the behavioral as well as structural refinement for the different types of resulting model components. Such an
automation would actually be desirable since it would help to guarantee a correct model component refinement.

We can also place appropriate hooks into the simulator, create a set of standard dynamic model information collector components or transducers (which are fine-tuned to some extend for specific applications and placed in the experimental frame), and write analysis programs which would identify and possibly execute refinement steps based on the previously introduced analysis methods.

Unfortunately, automation is hard to achieve for some forms of refinement: The refinement of the external model component interface, and as a result the increase in detail of component data structure or abstractly specified computational segments can at this point only envisioned to be done by a designer.
7. MODEL COMPILATION

A design model transformation phase represents the final phase in our model-based codesign methodology. Here, we attempt to derive a complete implementation level design description from a completely refined, system level design model specification. In this chapter we introduce the generation of efficient multi-processor target architecture for real-time embedded systems applications which we call model compilation. The next step in this transformation phase, which is concerned with the implementation of model component behavior, is discussed in the following chapter.

The derivation of an implementation architecture for a system level design model specification is shown from the perspective of a direct mapping as well as matching of the design model based on a set of given architectural elements or even an available complete target architecture. For the latter approach a model compilation algorithm is presented which selects the most efficient composition of a processing platform in regards to the utilization of its processors by introducing physical implementation constraints.

7.1. The Generation of Processing Platforms for Design Specifications

In current system level design approaches, we notice an inability to generate efficient design specifications for system level design models since commonly neither application performance nor implementation constraints are properly included in the generation of application implementations. In addition, we notice that most codesign approaches lack the consideration of parallel processing target architectures for an implementation of design specifications.
7.1.1. Code Generators for Design Model Specifications

Some system level modeling tools [42] offer already today optional code generation for
the system level design specification. These code generators are often only intended to
speed up model execution by disabling the most time consuming part of the simulation
engine - the animation of model diagrams. Software implementations are here directly on
the implementation of the modeling specification supported by the tool and not feasible
for an actual deployment. They are in general limited to single processor target
architecture implementations and require large operating systems to coordinate the
execution of the generated application software implementation. Processing requirements
for these target architectures tend to exceed resources for many smaller embedded
systems even by several orders of magnitude: The memory requirements for the required
operating system alone exceed the available memory budget for some embedded systems
applications. The inefficiency of the generated code demands high-end processors to
execute real-time applications within an acceptable time frame. Other modeling
environments [23] offer an automatic generation of code stubs, i.e., the structure of the
implementation, but leave the implementation of model behavior to the user.

7.1.2. The Model Continuity Problem

As it has been noted in [32], system modeling tools today still do not offer an acceptable
transition to an actual design implementation. This problem is referred to as the model
continuity problem in literature. In embedded systems development, this problem leads to
use of system level modeling tools for application implementation analysis instead of
design implementation generation. Simulation results are here transferred to a set of
hardware and software engineers as a refined system specification as opposed to a detailed design specification. In academic research, the model continuity problem has only received little attention.

In model-based co-design, we have already addressed the model continuity problem with our systematic and formally defined approach to the construction and refinement of system level design model specifications. Model compilation and implementation merely presents the second design phase which is the transformation of a design model specification to a mixed hardware/software application implementation.

7.1.3. What Constitutes a Good Target Architecture?

A number of performance metrics have been developed to assess the quality of embedded system design in the field of hardware/software codesign [4,7,9,48,52]. At the first glance it may seem that the fastest target architecture which offers the fastest response is the most desirable. Obviously, the answer here is that a complete custom hardware implementation ultimately results such response times. Such implementations are too expensive for many embedded systems budgets. Nevertheless, application response times are important for real-time embedded system applications. Rather than adjusting the response time during a model compilation we have encoded such constraints in our refined design model specification.

We feel that the quality of processing platform is best assessed in general by the utilization of its processing elements as shown in equation (4). Here, the highest utilization of all processing elements is most desirable. In a multi-processor configuration
the platform utilization can be computed as an average from the utilization of each processing element.

\[
\text{processing element utilization} = \frac{\text{computing time}}{\text{idle time} + \text{computing time}} \cdot 100\% \quad (4)
\]

Utilization is the most appropriate measure for the generation of efficient implementations for a number of reasons. It can be easily monitored and computed from our design model specification from the model component cycle times and model execution profiles. Design model specifications can be quickly adapted to processing platforms which will yield higher or lower amounts of utilization. Notice that the model component cycle time is the only design model parameter which is directly affected by changes in implementation technologies.

Starting with an architecture which consists of fully utilized processing elements we can easily compute the feasibility of an adaptation (or implementation) of any design model specification to any specified processing platform. If a mapping can be established we can not only guarantee that some timing constraint on any particular response time is met but we have even the possibility even further decrease it - if so desired.

A commonly encountered physical requirement for many embedded systems applications is to implement them with the least expensive (which in practice translates to generic, slowest, and the smallest number of) hardware components to keep reoccurring production cost at a minimum. Given a composed design model specification with the highest possible utilization we already have in essence such a processing platform already
at hand, i.e., our design model structure, since utilization is inversely proportional to the speed (and roughly the implementation technology).

7.2. Elements of the Processing Platform

Prior to a discussion of an appropriate processing platforms or target architectures for a given design model we introduce what we consider to be its architectural elements. As we have pointed out in previous chapters, we encounter the somewhat blurry statement in codesign literature that a target architecture consists of "hardware and software components". For our model-based codesign approach we suggest a different classification of such architectural elements into processing, hardware, and communication elements.

Software elements are also indirectly considered to be an architectural element, i.e., in the form of an operating system process. Therefore, one could argue that some behavioral part of the design model specification is in the traditional sense "mapped to a software implementation" when it is assigned to a processing element. Nevertheless, we advocate a departure from the previous conventional view and present these architectural elements as elementary building blocks in the eventual composition of different processing platform configurations.

7.2.1. Processing Elements

Our definition of a processing element is actually a much broader than a conventional micro-processor. Processing elements execute software programs or perform a set of instructions on data stored in memory. They can be characterized by a clock rate (the
inverse of the cycle time), a set of instructions, its data path width, and the amount of available external memory. We consider processor cache to be part of our definition of external memory but exclude processor registers.

Within the class of processing elements we can further distinguish between virtual processors (operating system processes), general purpose micro-processors, micro-controllers, Application Specific Integrated Processors (ASIPs), but also FPGAs and ASICs (with a considerable amount of external memory). Another possible classification separates custom processors (ASIP, FPGA, ASIC) from the class of conventional processors which contain the remaining processing elements. Custom processors such as Application Specific Integrated Processors are assumed to serve as dedicated processing elements for a single task. Such a processor may be any commercially available chip other than a conventional processor. An ASIP which has to be custom designed for an application is considered an ASIC.

7.2.2. Virtual Processors

An operating system introduces another layer of abstraction to a conventional processor. Although operating systems are actually implemented in software they can also be regarded as transforming a single conventional processor into multiple virtual processors [32,84] from a user perspective. From the processor perspective, a processing element which hosts multiple virtual processors can be considered a shared processing element. In the software domain a "virtual processor" is better known as a "process".

From the user perspective, an operating system can provide the illusion of concurrent task execution. This concept is implemented by only executing processes whose data is
immediately available. Operating systems allow to increase the utilization of conventional processing elements substantially in embedded systems software implementations. The stimuli for an application may here arrive at different input rates where even the exact time of arrival of such external data may only be known as a time window. If we adjust the entire computation to the fastest input rate some computational segments we may require faster processors or end up with computational segments which wait most of their time for data and lock up an entire processing resource without performing any true computation.

Operating systems can come in various flavors. At a minimum they consist of a scheduler, which coordinates the execution of processes, and some message passing primitives which are used for inter-process communication. This kind of operating system introduces the least processing overhead. There is virtually no limit in the additions one can make to operating systems, e.g., we can incorporate various other aspects of process management such as a user shell, memory management, entire file systems, etc. If the process management also considers timing constraints we have a Real-Time Operating Systems (RTOS) at hand. In this dissertation we will assume that the term "operating system" refers to a minimum RTOS.

7.2.3. Hardware Elements

We consider hardware elements to be synchronous combinatory circuits, i.e. hardware functions, which have no (or very little) external memory. Function or instructions which are implemented in the form of a hardware element are not associated with a fixed cycle time and can be considered to execute within a single clock cycle of a conventional
processor. Examples of hardware elements are standard FPGAs or ASICs. The choice in an implementation technology for a hardware element depends again on the complexity of the assigned behavior and its performance requirements, or more importantly the expected production count of the application to be implemented.

We use hardware elements to assist conventional processors in their sequential execution of application behavior or to accelerate their communication interface with other processing elements or the environment. In the first case, hardware functions can be considered as custom instructions or instruction set extensions of a conventional processor [85,86]. In the second case, hardware elements can be used to increase the utilization of the processor as well as the communication links by relieving the processor of the burden to handle communication, e.g., in the form of a custom communication chip. Notice that we can group multiple hardware elements onto a single FPGA or ASIC chip since multiple activities can execute by definition concurrently in pure hardware implementations.

7.2.4. Communication Elements

Communication elements are used to establish connections between all of the previously introduced architectural elements. Similarly as for processing elements we distinguish between dedicated (or custom) and shared communication elements. Dedicated communication elements are used in element-to-element data transfers. Such communication channels may be implemented as direct data transfers, e.g., signals from one hardware element to another, a message passed between processes which run on the same processor, direct I/O communication of a processing element with a hardware
element, and vice versa. If the size of communicated data items exceeds the physical size of the communication channel communication protocols may be used to transfer the data in a stepwise fashion. Direct communication channels provide the highest attainable bandwidth. For protocol-based communication the bandwidth is obviously dependent on the processing overhead introduced by the protocol. In the latter case, we may overcome performance problems by introducing custom communication chips which encode and decode the data passed between elements. But as with dedicated processing elements we face the problem of possible under-utilization of dedicated communication channels.

Shared communication channels or buses are also commonly encountered in of embedded systems applications. This type of communication uses of a set of custom communication chips which administer and the passing of information across the shared communication line, e.g., a system bus. Similarly, we can achieve a high utilization of the common communication channel but the maximum attainable bandwidth is limited. The processing elements which are connected to a shared communication link are for the most part relieved of the details of the data transmission which simplifies its implementation compared to an equivalent dedicated communication channel solution.

Physical implementations of communication channels may range from a connection on a multi-chip module or system-on-a-chip, wires between I/O ports of chips on a printed circuit board (PCB), copper cables, to optical fiber links. PCB style channels are most common place today for the physical implementation in applications since they still offer the lowest cost in a final product today. The communication delay introduced the data transfers along the presented physical communication links increases in the
presented order with the notable exception of optical connections. Communication channels are implemented most efficiently in SOC type implementations but have the fundamental drawback that they require a custom chip design and development.

7.3. The Processing Platforms and Modular Modeling Specifications

The first step in the transformation of a refined design model specification is the configuration of a processing platform that we map our model to. We may approach this problem from two perspectives: We may be interested in the best architecture (in regard to the utilization of its elements as defined in Section 7.1.3) for our application design, or the use of an already existing processing platform.

In this section, we present a detailed discussion on an appropriate processing platform for our design model specification. As we have been developing and refining our design model specification at the system level we have in essence already implicitly derived a multi-processor architecture for our application.

7.3.1. The Processing Platform Specified by the Design Model Specification

The processing platform encoded in our design model specification is mainly defined by its structural aspect. Each basic model component behavioral specification can be understood as a program which encodes some part of the desired application behavior based on some computational segments. During a model execution this program is in essence executed on a dedicated custom processing or hardware element, which is represented by the simulator associated with the respective model component [41]. In
order to arrive at the most efficient implementation we should therefore select processing elements which execute the model behavior just as the respective simulators do.

Simulators represent quite unique processing elements: They have an instruction set which matches exactly the set of model instructions used in the description of the behavioral aspect. Their clock rate is specified by the model component cycle time, i.e., the inverse of it. Their data path has in essence the size of the biggest operand used in the specified model instructions.

The coupling or connections between components of our modular composed design model specification specify dedicated communication channels between elements. The communication of data between component is considered to occur instantaneously between internal computational segments internally within a model component and externally between model components at a rate determined from the size of the data structure and the delay time of corresponding communication components.

7.3.2. The Ideal Processing Platform

We conclude that the ideal processing platform for a given application needs to be an exact match of the architecture described above. Basically it consists on a network of custom processors, i.e., ASICs with external memory. The implementation of each custom processor is here be guided by a data flow analysis of each model component specification, which determines the appropriate data path width for each processor, and an analysis of model instructions to compose the processor instruction set. The required clock rate is specified by each model component cycle time.
One minor modification which would have to be made is to round the cycle time, i.e., downwards for each model component so that it would specify a realistic clock rate value for the respective custom processor, and to adjust the its model behavior accordingly (using a design model communication analysis). An implementation of this ideal processing platform would then be possible if the required clock rate values for each model component are within the boundaries of a physically achievable maximum value.

Notice that we can compensate physically unachievable communication rates by increasing the clock rates of custom processors slightly which participate in the communication (which is equivalent to decreasing the cycle time for the respective model component) in order to increase this value. Internal communication can be assumed to be negligible by implementing data transfers between computational segments using direct memory references.

7.3.3. Back in the Real World ...

What makes such an ideal processing platform mostly unrealistic is the budget necessary to design and develop possibly a large number of such custom processors and their software development environments from scratch. More realistic are system specifications for embedded system applications which require the use of a limited number of commercially available processing elements which have with a fixed processor speed, instruction set, and data path width.

For many embedded systems applications we may be even restricted to a single processor design. In addition, it is likely that we are restricted in our mapping of the design model specification to an already existing processing platform configuration
instead being able to derive a custom processing platform. At best we may extend or improve such a platform by adding some hardware elements to it.

7.3.4. A Second Level of Structural Hierarchy

From a systems modeling perspective we can represent any processing platform which differs from this ideal platform as a second level in the structural hierarchy of our design model specification. Here, level 1 model component specifications capture in our structural hierarchy either application behavior in design model components based on single or multiple computational segments, or communication cost between design model components using generic communication components. Communication components are necessary to specify application behavior in more than one a single model component specification.

A second level in our structural hierarchy introduces its physical implementation. Level 2 model components can be used to group level 1 model components, which are assigned to the same physical processing element or hardware element, in a single coupled model specification. If only a single design model component is part of a level 2 component its behavior can be implemented in a hardware element or on a dedicated custom processing element. If multiple design model components are part of a level 2 model component each design model component behavior are realized as a process on a regular processor or as multiple hardware elements part of a single physical hardware element.

The concept of a second level of hierarchy helps us to understand the concept of a model compilation from a systems modeling perspective which is to determine an
appropriate design model specification at level 2. A similar idea called module charts has been introduced in [13] to show an assignment of StateCharts design models to an implementation at the system level. In essence, module charts are a graphical representation for level 2 model components.

7.4. Model Compilation as a Search Problem

As shown in the previous section we have been assuming an underlying parallel processing platform during our design model construction and refinement. As a matter of fact any performance assessment of our design model specification in the previous design model analysis has been based on this assumption. In practice, processing platforms for the implementation of embedded systems applications tend to be fairly well constrained. Model compilation takes such constraints into account and assesses the validity of a mapping a design model specification onto a given processing platform.

In the more general case (where we assume to have a set of architectural elements but no fixed configuration of them) we can define the mapping of a design model specification as an artificial intelligence search problem. Here, the first search space are all possible combinations of available architectural elements where each element can be uniquely described using some specific parameters. The design model specification provides us with a set of components which we attempt to map to different configurations of architectural elements, i.e., a set of processing platforms. A valid mapping is achieved if a design model implementation is able to produce the behavior encoded within our design model specification within the encoded timing constraints.
7.4.1. The Reduction of the Search Space

The search space can be quite large if we take all possible architectural elements into account which exist in the marketplace today. Assuming that we have a set $N$ of possible architectural elements and a set $C$ of design model components the number of possible mappings is on the order of $\Theta(|N|^C)$ since multiple model components may also be mapped to the same processing element. Notice that each element of $N$ represents a specific physical architectural element.

The search space can be reduced by taking some obvious facts into account: First, we can exclude communication elements from $N$ since such an element can not exhibit the behavior of a design model component. Secondly, the maximum number and kind of processing elements is usually a small number, i.e., there is only little choice in the selection of processors or microcontrollers for a given application. In addition, custom processors only contribute three elements to $N$ according to our previous definition. This implies that in practice the size of $N$ is fairly small.

The size of the set of model components $C$ is dependent on the concurrency inherent to the application which tends to be a reasonably small number in practice. Locked model components (see Chapter 5) reduce the number of possible combinations further since they are used in a model compilation as fixed assignments.

7.4.2. The Model Compilation Algorithm

A variety of search algorithms can be found in research literature [27, 52, 87] which propose different approaches to solve this search problem in reasonable time. Figure 22
shows a simple branch and bound algorithm which explores different candidate processing platforms for different component mappings. Each mapping is here evaluated if it fulfills performance constraints which are encoded in our composed design model specification. Valid mappings of the design model specification are stored in the set "List". In the following paragraphs we will first introduce this algorithm in a fairly general manner and then continue with in-depth discussions in the following sections:

```plaintext
while ( iteration < |N|^C ) { 
  Processor_Configuration = Select_Processor_Configuration( Processing Elements, Hardware Elements ); 
  for ( each unique Mapping = Select_Component_Mapping( Design Model, Processor_Configuration ) ) { 
    if ( Status = Possible_Dedicated_Processor_Solution( selected Mapping ) is Success ) { 
      for ( each Multi_Component_Assignment_in_Mapping; if Status = Success ) { 
        Status = Scheduling_Analysis( selected Multi_Component_Assignment_in_Mapping ); 
      } end for 
      if ( (Communication_Assessment( Mapping, Locked Components Assignments, Communication Constraints ) and Status is Success ) { 
        List = Save_Candidate( List, Locked Components, Processor_Configuration); 
        [exit;] 
      } end if 
    } end if 
    Increment( iteration ); 
  } end for 
} end while 
```

Figure 22. Pseudo Code for a Possible Search Algorithm Implementation

- In the illustrated algorithm, we first retrieve all of the elements from N and select a unique mapping of design model components which are part of the set C. Here, the selection of a unique mapping may eliminate redundant alternations in the order of assignments for the same multiple model component to the same shared processing element. Although it is not explicitly shown in this figure, the iteration count needs to
be incremented for each eliminated mapping in order to terminate the presented algorithm properly.

- In its next step each model component assignments implied by the selected mapping is assessed individually, i.e., as a dedicated processor implementation. Notice that any failed dedicated processor implementation already rules out a multi-component mapping which includes this model component.

- If the mapping to a dedicated processing element (or a hardware element) succeeds we proceed with a scheduling analysis for each multi-component mapping. It determines if the assigned components can be scheduled sequentially on the designated shared processing element. Such an assessment requires an adaptation of each model component behavior, which is part of the multi component mapping, to execution characteristics of the targeted processing element. If the scheduling analysis fails for any multi-component assignment we select a new component mapping. In this selection, we can eliminate variations of the multi-component mapping which failed due to scheduling problems.

- If a scheduling is possible for all assignments we verify that the given processing platform is able to fulfill communication constraints encoded in the adapted design model specification which also includes locked model components. If these communication requirements are beyond the capabilities of a physical realization the component mapping is considered to be invalid and the next mapping is selected.
• In the case of a successful communication analysis we accept the processing platform for a valid design implementation and save it on a list of candidate processing platforms. We proceed then with the next iteration.

From this search process possibly multiple solutions or configurations can be derived which differ in properties, e.g., number of processing elements, production cost (i.e., implementation cost of its components), utilization of processing elements, etc. The selection of the appropriate implementation solution then depends on the importance of these properties in the application implementation.

Such a constraints can also be encoded in the model compilation algorithm by introducing appropriate search heuristics to the selection of the next processor configuration to consider, e.g., a desired implementation should consist of less than 3 processors. By selecting processor configurations according to this search heuristic prior to each iteration we can further reduce the execution time of the algorithm implementation. Similarly the algorithm can be terminated once the first processing platform which fulfills the desired constraints is found.

7.5. Model Compilation Analysis

We can limit our possible processing platform configuration space by identifying irrelevant "dedicated" processor mappings either during or prior to a model compilation by performing a proper analysis of design model components. A similar approach can be taken to assess the implementation of a given model component in form of a custom hardware element.
A model compilation analysis serves as a preparation for the actual model compilation and may already be performed during the development of our design model specification in conjunction with the other design model analysis.

7.5.1. Model Component Specification Assessment

We can estimate required processor characteristics for each model component specification by collecting processor relevant information, e.g., the composition of the data structure or the complexity of its model instructions. In essence, we assess here if available processing elements are able to efficiently reproduce to encoded design model behavior. Some examples of processing element characteristics are shown in Table 2.

For each model component mapping Cycles-Per-Instruction (CPI) counts need to be computed for a specific processor implementation and may be estimated for custom processors where $\text{CPI}_{\text{FPGA}} \leq 1.0$. An alternative for model components, which are mapped to general purpose micro-processors or micro-controllers, is to derive this measure from cycle accurate simulation results of its computational segments [61].

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Data Path Width</th>
<th>Instruction Set</th>
<th>Clock Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infineon C1610</td>
<td>Microcontroller</td>
<td>16-bit</td>
<td>Simple, pipelined</td>
<td>16 MHz</td>
</tr>
<tr>
<td>Intel i386</td>
<td>Microprocessor</td>
<td>32-bit</td>
<td>Simple + FPU</td>
<td>33 MHz</td>
</tr>
<tr>
<td>Altera MAX9320</td>
<td>FPGA</td>
<td>Flexible</td>
<td>Custom</td>
<td>25 MHz</td>
</tr>
</tbody>
</table>

Table 2. Example Characterization Table of Processing Elements

In addition, components should be identified which require the use of special instructions or floating point numbers. Such instructions may already limit the implementation choice to a specific processing element, e.g., a custom processor or
microprocessor (with an integrated floating point unit (FPU)). From model component
data tables we may select an appropriate data path with for an appropriate the processing
element.

A required memory budget for a specific component-to-element mapping can be also
computed as shown in equation (5). Here, an estimate for the expected code size can be
obtained all instruction counts (IC) of all computational segments (CS) in the model
component specification and an average instruction size (AIS) for the particular element.
The data size is computed from global (i.e. shared) as well as local data items (DI) used
in these computational segments.

\[ \text{Required Memory} = AIS_{\text{element}} \cdot \sum_{\text{all CS}} IC_i + \sum_{\text{all DI}} \text{sizeof}(DI_i) \]  \hspace{1cm} (5)

7.5.2. Estimation of Effective Model Component Cycle Times

From this collected data we can also compute an effective model cycle time (MCT_{eff}) for
processing elements in a specific component-element mapping which can then be used to
assess the feasibility of the mapping. The MCT_{eff} for a particular processing element is
derived using a formula similar to the one used for the estimation of unbiased application
execution time which was presented in Chapter 4. Equation (6) enables us to perform a
rough estimation where the "element" label denotes a processing element property and
the "component" labels a model component property. Notice that this formula constitutes
a rather cautious estimation since it even single occurrence of a complex data item in
some computational segment may result already in a substantially smaller cycle time.
7.5.3. Elimination of Invalid Dedicated Processor Mappings

In a dedicated processor mapping, we compare then the information generated during a model compilation analysis with the characteristics of available processing (or hardware) elements. If the computed MCT\textsubscript{er} for a given processing element has value lower than the actual model cycle time (MCT) of the model component we consider the mapping to be valid, i.e., the processing element will be able to reproduce the behavior specified by the model component faithfully.

If the MCT\textsubscript{er} should evaluate to slightly less than the MCT of a given component we should consider a further refinement of this model component. Here, we may attempt to increase the MCT by performing a structural refinement on the component on the basis of a model component performance analysis (see Chapter 6).

7.6. Design Model Adaptation

Continuing in the model compilation algorithm we adapt each model component behavior to match the processor characteristics of the assigned target element for the following scheduling and communication analysis. This adaptation is straightforward and combines the concepts introduced in the previous section with correct model refinement techniques presented in Chapter 5.

Here, the MCT for each model component specification is replaced by the MCT\textsubscript{er} of the target element specified in the mapping. A result of this revision of the cycle time
requires again an adjustment of execution times for all computational segments in the respective design model and communication component specifications.

7.7. Scheduling Analysis

In our next step of the model compilation algorithm we assess the mapping of multiple model components to virtual processors, i.e., a shared conventional processing element. Mapped model components execute on this processor in the form of RTOS processes. Given the importance of real-time constraints and the ability to execute components concurrently with other components at the system level this mapping requires a more detailed assessment of the design model specification. Here, we need to investigate a correct sequential execution of all model component specifications which are assigned to the same processor.

7.7.1. Benefits of RTOS-based Software Implementations

An operating system provides us with the possibility to implement a sequential implementation of concurrent behavior. It also allows to decouple the invocation of processes from the arrival of external input data. This is achieved by blocking, temporarily removing, or swapping out processes which wait for the arrival input data. Notice that these processes still execute on a single processor, i.e., only one process can be active at any point in time.

Process activation is managed by a scheduler which can also be used to resolve invocation conflicts between concurrent processes. When desired input data arrives for a process it is invoked by the RTOS scheduler to process its data. Process invocations may
be delayed by holding back the required input data. The generation of outputs can also be
delayed internally in the RTOS and hidden from the processes themselves as long as an
appropriate interface is provided for each process to specify such a desired delay.

7.7.2. Shared Processor Implementations and Concurrent Design Specifications

In the previous chapters we have been strongly encouraging and supporting the
development of a highly concurrent design model specification. As a matter of fact we
have even optimized our design model specification during design model refinement to
spent as little time as possible in waiting states. Therefore, operating scenarios where
multiple design components require an execution at the same time seem to a big obstacle
for a shared processor implementation.

Although our system level design specifications provide the capability to express
concurrency in the specification of design model behavior they may not necessarily do so
during a model execution. The arrival of some application stimuli may be a periodic.
Similarly, the acceptance of some input events by the application may be dependent on
its mode of operation. Given the potential for concurrent execution in an application
design model this argument is still not very convincing.

Secondly, our detailed design model specification was derived in the attempt to
derive the largest lowest MCT attainable for each model component specification in order
to maximize the utilization of processing elements. During a scheduling analysis we are
dealing with a design model specifications which has been adapted to a processing
element where in general $MCT_{eff} < MCT$ and often $MCT_{eff} << MCT$. The results of a
smaller model component cycle time are smaller processing state execution time and
more time spent in receiving waiting states. But even if model components still execute concurrently in their processing states model execution it may still be possible to schedule processes sequentially in a software implementation.

7.7.3. Scheduling Analysis for Multi-Component Mappings

A scheduling analysis assesses the feasibility of such a multi model component assignment. This is again done by observing adapted design model behavior in simulation experiments. Of interest are especially the time spent by each respective design model component in its processing states. A scheduling analysis is performed for each shared processing element in isolation and can not only verify a possible implementation but also devise an appropriate scheduling order for these components.

A group of \( n \) model components is schedulable on a single processing element if the simulation traces indicate that only a single model component is active at the same time, i.e., the coupled model state of a group of \( n \) model components contains only of a single model component processing state and \( n-1 \) waiting states at any point in time.

![Model State Execution Trace of Schedulable Model Specifications](image)

Figure 23. Model State Execution Trace of Schedulable Model Specifications

Figure 23 shows an example of such a coupled model simulation trace which starts at simulation time \( t^* \) and focuses on the behavior of model components A and B. We
assume that component A and B have been assigned to the same shared processing element. Their interaction along communication channel z can be converted into a delayed invocation of process PBj. If PBj should generate any external outputs we may adjust the delay parameter of its associated communication component. It is trivial to extract the scheduling of processes from a model state execution trace. In our example, the scheduler would impose the following execution order: \( \ldots \Rightarrow PA_i \Rightarrow PB_j \Rightarrow \ldots \)

7.7.4. Resolving Scheduling Conflicts for Shared Processor Implementations

Scheduling conflicts occur if the model state execution trace of model components, which are mapped to a shared processing element, contains overlapping processing states. An example simulation trace of such a scenario is shown in Figure 24. A general assessment of this case requires a more formal discussion based in the derivation of scheduling windows. For this purpose we will refer to the model component specifications assigned to this processing element as the set of model components MCPE.

![Execution Trace with a Scheduling Conflict during Time Interval (t_{PBj}, t_{WCk+1})](image)
For the components part of MCPE we first focus on the set components MCo which have overlapping processing states where MCo ⊆ MCPE (in our example MCPE = {A, B, C, D} and MCo = {A, B, C}). For each component c ∈ MCo the size of its scheduling window \( t_{sch\_w}(c) \) is computed by summing up the values of the following three parameters: its execution offset \( o_c \) measured from a reference point, the execution time of its processing state \( t_{ac}(Pi) \), and flexibility window \( t_{nex}(c) \). The reference point for the execution offset is the point in time at which the first component of MCo enters its processing state. The size of the flexibility window \( t_{nex}(c) \) is dependent on the external outputs generated by Pi. If Pi produces external outputs for model components, which not part of MCPE, the size of \( t_{nex}(c) \) is the smaller value of the delay time parameter of its associated communication component \( t_{delay}(Y_{pi}) \) and the time spent in the next waiting state, i.e., \( t_{nex}(c) = \min(t_{delay}(Y_{pi}), t_{ac}(W_{i+1})) \). In all other cases, the size of this window is equal to the latter value, i.e., \( t_{nex}(c) = t_{ac}(W_{i+1}) \). A rescheduling is possible for these components under the condition that the scheduling window for any MCO component c exceeds the total sequential execution time of these overlapping processing states:

\[
\forall c \in MCo \ t_{sch\_w}(c) \geq \sum_{MCo} t_{ac}(Pi)
\]

If MCo is only a subset MCPE we need to verify that there is a similar scheduling window available in the waiting states of the remaining components MC (i.e., MC ⊂ MCPE, MC ∩ MCo = {Ø} and d ∈ MC). In our example MC = {D}. The window size for each component \( t_{sch\_w}(d) \) is here computed by from the difference in total execution time before and after a rescheduling of MCo components, i.e., \( t_{sch\_w} = t_{pseq} - t_{pcon} \) where
$t_{pseq} = \sum_{MC_{o}} t_a(P_i)$ and $t_{pcon} = t_{lastWo} - t_{firstPo}$. In the latter formula the subscripts of $t_{lastWo}$ and $t_{firstPo}$ refer to the time that the waiting state following the last processing state of the sequence of overlapping processing states is entered, and the time the first processing state in this sequence is entered. This results in the following second condition for MC components based on the time $t_{pi}$ when component $d$ enters its next processing state $P_i$:

$$\forall d \in MC \ (t_{pi} - t_{firstPo}) \geq (t_{pcon} + t_{Schw(d)})$$

Notice that a rescheduling can only achieved if both of these conditions are fulfilled.

Figure 25. Example Simulation Trace after Rescheduling

We use the previous example from Figure 24 to illustrate the scheduling analysis. Here, component A produces external outputs which are destined for B. Components C and D are waiting for external inputs from components which are not part of the set MCPE. Our reference point is $t^*$. Lets assume that the delay time for $ta(P_d) = 7$, the offset for component A $o_A = 2$, the required execution time of $PA_i$ is $ta(PA_i) = 3$, and the time spent in $WA_i+1$ is $ta(WA_i+1) = 10$. Similarly, $o_B = 0$, $ta(PB_j) = 3$, $ta(WB_j+1) = 9$, $ta(PB_j+1) = 3$ for B, and $o_C = 3.5$, $ta(PC_k) = 3$, $ta(WC_k+1) = 12$ for C, and $ta(WD_l) = 17$.
for D. From our formulas it follows that $t_{schwo}(A) = 15$, $t_{schwo}(B) = 12$, $t_{schwo}(C) = 18.5$, $t_{seq} = 9$, and $t_{sch}(D) = 9 - 6.5 = 2.5$. In this scenario a rescheduling is possible since the first condition ($15 > 9$, $12 > 9$, and $18.5 > 9$) as well as the second condition (since $t_{pd1} - t_{pbi} > 15 \Rightarrow 15 > 9$) are fulfilled - in the latter case by an examination of the execution trace. As depicted in Figure 25, an appropriate execution order which would resolve the scheduling conflict could constitute: $\ldots \Rightarrow PB_j \Rightarrow PA_i \Rightarrow PC_k \Rightarrow PB_{j+1} \Rightarrow \ldots$

7.7.5. An Algorithm For Rescheduling Conflicting Components

The rescheduling of components or processes can then be achieved in a variety of ways. A result of any modification of the original scheduling is that communication components associated with the MCo components may have to be modified. We show here an algorithm as shown in Figure 25 which maximizes the time available for communication between processing elements, i.e., processor-external communication in the final design implementation.

From the set of overlapping model components MCo we first select components with the lowest offset value. If multiple components qualify then we select components which produce external outputs first. After that we prioritize components with a destination component which is not part of the set MCPE. Components which are coupled to another MCPE component are then ordered so that they communication is preserved properly, i.e., for all their corresponding communication components $t_{delay}(Y_{pi}) \geq 0$. For conflicts within any of these selections we order the reduced set according to the size of their adjusted flexibility window $t'_{flex}(c)$. The adjustment is computed for each component on
the candidate list from its relative offset relative to the start of the conflict t* and required execution time of its processing state.

```
Unscheduled Set = MC; Scheduled Set = {}; Schedule Time = 0;

While ( Unscheduled Set is not empty ) |
    pick first components based on their offset value of their scheduling window
    Candidate Set = Components with Offset less or equal than Schedule Time in Unscheduled Set;
    if ( size of Candidate Set > 1 ) then |
        // pick components based if they produce external inputs
        Output Component Set = Components which produce a set of External Outputs in Candidate Set;
        if ( Output Component Set is not empty ) then |
            pick components that produce external outputs for non MCPE components first
            Non-MCPE Set = Components with non MCPE Destinations in Output Component Set;
            if ( MCPE Set is empty ) |
                then these components have MCPE destinations; restrict further to resolve MCPE scheduling
                Candidate Set = Components with lowest Offset in Output Set;
            else |
                then these components have non MCPE destinations; restrict further based on communication components
                Candidate Set = Components with lowest output delay in Non-MCPE Set;
        end if |
    end if |
    for ( all the components in Candidate Set > 1 ) |
        Adjusted Flex Time = Component Flex Time;
        if ( Adjustment = Schedule Time - (Component Offset - Component Execution Time) ) > 0 ) |
            Adjusted Flex Time = Component Flex Time - Adjustment;
            Candidate Set = Components with lowest Adjusted Flex Time so far;
        end if |
    end for |
    if ( size of Candidate Set > 1 ) then Candidate Set = First Entry in Candidate Set;
    if ( Component in Candidate Set produces External Outputs ) then |
        Decrease Delay Time of Communication Component of Candidate Component by corresponding Adjustment Value;
        if ( negative Delay Time ) then delay = 0; // some cases involving external output components not part of MCPE
    end if |
end if |
Schedule Time = Schedule Time + Execution Time of Component in Candidate Set
Scheduled Set = Scheduled Set + Candidate Set; Unscheduled Set = Unscheduled Set - Candidate Set;
end while
```

Figure 26. Example Rescheduling Algorithm for Components with Scheduling Conflicts
7.7.6. Scheduling Analysis with Processing States Sequences

Some model specifications which are part of MCo may execute a sequence of processing states in a simulation trace, i.e., processing states which are not separated by waiting states (see Figure 21). In a scheduling analysis we can treat such sequences as a single processing state where the execution time $t_{\text{a}(P_i)}$ for the model component is the total of all the respective execution times.

The computation of $t_{\text{exec}(c)}$ also changes if one of the processing states in the sequences generates external outputs for model components assigned to other processing or hardware elements. In this case, the delay time $t_{\text{delay}(Y_{P_i})}$ for external outputs is here the minimum value of the adjusted delay parameters $t^*_{\text{delay}(Y_s)}$ where $s$ refers to a processing state in this sequence SEQ which generates such external outputs. Here, each adjusted delay parameter is computed by adding the execution times of stated preceding $s$ in SEQ to its original delay time $t_{\text{delay}(Y_s)}$.

7.8. Processor Communication Analysis

The final step in our model compilation algorithm to examine the proposed processing platform from a communication perspective. In this assessment, we focus on communication components between model components which are assigned to different architectural elements. We evaluate here model execution trace of the composed design model specification where we isolate communication components in a similar manner as we isolated design model components in the previous scheduling analysis.

Based on the behavior of such communication components over the same set of simulation runs we suggest or assess different implementations using available
communication elements. For most embedded systems applications the implementation of communication between components in the target architecture is commonly well constrained. In practice, application system specifications require a PCB type processing platform where we can expect dedicated communication channels between architectural elements for single processor and shared channels, i.e., communication buses for multi-processor configurations. Hardware elements are usually connected with a dedicated channel, i.e., such links are most efficiently implemented in processing platforms by mapping the hardware component to into the external processor memory space.

7.8.1. Bus-based Processor Communication

A bus-based communication scheme is actually the closest fit with our model of communication at the system level and therefore is fairly easy to validate. Here, the source element sends the data to a communication chip, e.g., a bus controller, which sends then the data to the communication chip of destination element. When multiple elements are connected to this bus conflict may arise in the distribution of data, e.g., if element A may want to send data to B at the same time that C sends data to D over the same communication bus. Here, a bus arbiter resolves the order of data transmission. This is just another setting for the scheduling conflict problem which we encountered previously in the assignment of design model components to a single processing element.

In our design model specification, communication components have abstracted the behavior of these communication chips. An assessment of bus-based communication in our processing we can therefore perform a similar scheduling analysis on the execution trace of all the communication components which are associated with couplings of
components which are assigned to architectural elements connected to the same bus. Results of such an analysis can be used to identify bus collisions, i.e., bus scheduling conflict. We can also derive other important information which can be used in the selection or validation of a proper implementation candidates, e.g., the minimum required bus bandwidth, expected bus traffic, utilization, etc. Execution performance estimates can be derived from the model state execution trace by using the delay time parameters of communication components in the same manner we used execution time of processing states in our previous analysis. In addition, the results of a static communication component analysis (which was presented in the previous chapter) provides us with estimates for the required amount of data transfers which can be expected from each design model component.

Figure 27. Example Communication Window \((t_{W,A_i+1}, t_{P,B_j+1})\) in Simulation Trace

7.8.2. Direct Processor Communication

In a direct communication scheme, i.e., dedicated communication channels between any two processing elements, require again a more detailed analysis of the behavior in source and destination model component specifications. Here, a common communication window has to available in the behavior between two interacting processing elements in
order to warrant a correct implementation. We use again simulation traces to assess the feasibility of such means of communication. Figure 27 shows an excerpt of generic simulation trace where model components A and B communicate along a channel represented by communication component z. The illustrated case assumes that A and B reside on different processing elements.

For each interaction between model components A and B we can compute communication windows between the component pair, i.e., $t_{\text{comwin}} = \min(t_{PBj+i}, t_{PAi+i}) - \max(t_{WAi-1}, t_{WBi+1})$. The smallest time window of all interactions specifies the final implementation constraint for a dedicated communication channel. An additional constraint is that $t_{\text{comwin}}$ has to be large enough to allow the slower of the two processing elements a timely completion of the processing associated with data transfer. A method for such an assessment will be discussed in the following section.

Notice that in such an assessment $t_{\text{comwin}}$ is not affected by the delay parameter $t_{\text{delay}}(Y_i)$ of the communication component which models the data transmission since the time spend in WAi+1 may be smaller than $t_{\text{delay}}(Y_i)$. Nevertheless we may again use a static communication component analysis to determine the size of expected data transfers and compute the required bandwidth using $t_{\text{comwin}}$.

If one of required bandwidths of the dedicated communication links exceeds the capabilities of a given processing platform or available communication elements the entire design model mapping is considered invalid. Notice that in the case we may go back and refine our design model specification to enlarge communication windows which
were violating these conditions if the model compilation failed merely due problems in the communication analysis.

7.8.3. **Hardware Interfaces**

In our definition of architectural elements, we have delegated hardware elements to act as performance boosters for conventional processors which are connected via dedicated communication links, i.e., they accelerate parts of a computational segments or speed up their external communication interface. We can therefore expect only a small amount of required data transferal between hardware and processing elements. Nevertheless, the feasibility of introducing such an accelerator component is strongly dependent on the time spent in the actual data transfer since hardware implementation the execution times of these two architectural elements vary greatly.

![Generic Execution Trace For Model Components with a Processor and Hardware Mapping](image)

**Figure 28.** Generic Execution Trace For Model Components with a Processor and Hardware Mapping

In processor communication analysis, we can assess this feasibility in a fairly straightforward manner by examining again their model state execution trace as well as
performing a static communication component analysis of the respective model components. A representative model state execution trace for such a scenario is shown in Figure 28. Here, component A is assigned to a processing element and B to a hardware element. Communication component y is associated with the coupling of A to B, and z with B to A.

For the processing platform we can assume that any communication can be entirely contributed to the processor based on the reasonable assumption that the hardware element executes its behavior in less that a single processor cycle. The transfer time for the implementation data transfer for can therefore be determined from the processor cycle time and its data path width using equation (7). Here, the computation of data transfer time is based on the size of the data items (DI) sent in the delay state s of the communication component. This equation also includes a fudge factor \( t_{\text{protocol}} \) for communication protocol overhead.

\[
\text{transfer}(s) = \left( \sum_{\text{all } DI \in s} \frac{\text{sizeof}(DI)}{\text{DPW}_{\text{processor}}} \right) \cdot \frac{1}{\text{clock rate}_{\text{processor}}} + t_{\text{protocol}} \tag{7}
\]

In a model compilation, we accept then such a mapping design model specification if selected or any available communication elements for an implementation are able to perform all required data transferals as specified in the design model, so that \( ta(WAi+1) \geq t_{\text{transfer}}(PdAB) + ta(PBj) + t_{\text{transfer}}(PdBA) \), or simply \( t_{\text{transfer}}(PdAB) \leq ta(PdAB) \) and \( t_{\text{transfer}}(PdBA) \leq ta(PdBA) \).
8. IMPLEMENTATION FROM THE BEHAVIORAL MODEL ASPECT

At this point in the design cycle the structural aspect of our design model specification has been captured in form of a processing platform for our application. This platform has been specified as collection of architectural elements such as specific processing elements (e.g., general purpose processors), communication elements (e.g., a bus connecting such processors), hardware elements (e.g., FPGA), and a process scheduling order for the operating systems running on each shared processor.

In this chapter we finish the transformation of our design model into an implementation level design description with the implementation of model component behavior. This last step turns our system level design model to a complete implementation level design specification.

8.1. The Transformation of the Behavioral Aspect

A closer inspection of our detailed design model specification reveals that we already have developed algorithmic level descriptions of computational segments which are left to be implemented. What remains however is their integration into the discrete event modeling specification. In order to convert a model component behavior into an implementation we therefore need to decouple each design model component specification from its underlying simulator component.

8.1.1. The Execution of the Behavior in Model Components

In our construction of model component specifications application behavior has been mainly encoded by the form of computational segments which are associated with
processing states. Waiting states were introduced to synchronize interactions of computational segments assigned to different design model components at the system level. What remains a fundamental difference between our refined design model and an implementation level design specification is that the stimulus required for the execution of some behavior and its generated outputs is handled indirectly by the simulator attached to each model component specification instead of the model component itself.

This approach has proven to be very useful for the execution of abstract computational segments. Here, we were able incorporate computational segments even as basic input/output relations, and observe their effect on application behavior. Prior to a model compilation however the composed design model specification has been refined to the lowest level of resolution where all computational segments are completely specified. At such a specification level our simulator harness turns from an essential execution infrastructure into pure processing overhead.

8.1.2. The Transformation of Model Component Specifications

In order to generate simulator independent implementations from the behavior encoded in design model component specifications we continue to apply the same concepts as during model compilation. Basically we consider model component states to represent the internal state of the hardware or processing element state in prior to the execution of its associated computational segment. Each processing model state maps to an "executing" the state of the architectural element that it is mapped to. Waiting states are regarded as a idle processing state which can be realized in form of a blocking read primitive in a
software implementation or as a waiting for a buffer to be updated in a hardware implementation.

Similarly as in the model specification, the proper distribution external inputs and outputs on shared processing elements is implemented for each computational segments separately from the computation itself, i.e., in the form of some receive and send primitives. These primitives handle the connection to other processing element, the supervision of the data transmission, and possibly even the transmission protocol itself, e.g., for dedicated communication channels. For custom processing elements such communication primitives need to be included in the processor design.

Contrary to the model specification though we move the invocation of these primitives to the beginning and the end of respective computational segments so that each segment initiates the retrieval and posting to other processes itself. This modification is possible due to our restriction of constructing system models based on waiting and processing states only. Model state transitions are used to establish an order in the execution of multiple computational segments on a processing element.

8.1.3. A Formal Discussion of Model Component Transformation

Formal representations offer the benefit show a general mapping between formalisms, i.e., a general procedure to derive an equivalent specification from the first specification. In this section we use a formal mapping to show the derivation of implementation specification from a modular system modeling specification. In our case we choose parallel DEVS [19,88] as a formal modeling specification, which allows the specification of modular design models, and timed CSP as a formalism [71] which can be used to
model the real-time computing system implementations, i.e., the specification of multiple sequential executing applications which execute on a network of independent processors.

The timed CSP notation is of interest as a target representation for a number reasons. Not only provides it an excellent basis for a formal discussion of a mapping, but it can also be used to further analyze a design implementation for liveliness and deadlock conditions [71]. Secondly, publications [70] indicate a fairly simple direct transition to a software implementation from a timed CSP specification. In addition, extensions of CSP have been already been successfully used as a basis for implementation level codesign approaches [27].

8.1.4. Mapping of DEVS Atomic Models to Timed CSP Processes

Our mapping between the two formalisms illustrate formally the implementation of a generic model component specification. Figure 29 shows how we can express a complete of a DEVS model component specification in the timed CSP\(^1\) notation. Here, we map DEVS input ports directly to a CSP channel and the external set of DEVS input events to a set of CSP events. The same mapping is applied to output ports and external output events. Coupling information of the output ports is used to complete the definition of each CSP communication channel. The set of DEVS model states is directly mapped to a set of recursive CSP processes. For each CSP processes, which implements a DEVS processing states, we include the associated computational segment using event transitions (here represented by the compute function). As in DEVS, CSP event transitions

\(^1\) In the remaining discussion any reference to CSP will refer to the timed CSP notation.
transitions are also considered to execute instantaneously. Therefore, we are required to introduce infinite time delay, i.e., the STOP process, for each computation. The DEVS transition functions and output function are encoded in a separate set of mutually recursive CSP processes. The SKIP process used in the CSP output processes generates a termination event instantaneously.

- For each DEVS external input event $x_i$ in $X_{DEVS}$ there is a CSP event $f_i$ in $X_{CSP}$:
  $$x_i \in X_{DEVS} \Rightarrow f_i \in X_{CSP}$$

- For each DEVS external output event $y_i$ in $Y_{DEVS}$ there is a CSP event $g_i$ in $Y_{CSP}$:
  $$y_i \in Y_{DEVS} \Rightarrow g_i \in Y_{CSP}$$

- For each DEVS model state $s_i$ we define a CSP process $P(s_i)$:
  $$s_i \in S_{DEVS} \Rightarrow P(s_i) \text{ where } P(s_i) = \text{compute}(s_i) \rightarrow \text{STOP}$$

- For each set of DEVS external transitions from $s_i$ we define a CSP process $\text{EXT}(s_i)$:
  $$\delta_{ext}(s_i, e_h, x_k) = s_j \Rightarrow \text{EXT}(s_i) = (x_k @ e_h ? X_{CSP} \Rightarrow P(s_j) \mid \ldots )$$
  or:
  $$\{ \delta_{ext}(s_1, e_1, x_1) = s_3; \delta_{ext}(s_1, e_2, x_2) = s_4; \text{etc.} \} \Rightarrow \text{EXT}(s_i) = (x_1 @ e_1? \rightarrow P(s_1) \mid x_2 @ e_2? \rightarrow P(s_4) \mid \text{etc.} )$$

- For each internal transition from $s_i$ we define a CSP process $\text{INT}(s_i)$:
  $$\delta_{int}(s_i) = s_j \Rightarrow \text{INT}(s_i) = P(s_j)$$

- For the output function for $s_i$ we define a CSP process $\text{OUT}(s_i)$ where $Y_{h_i} \subseteq Y_{DEVS}$ and $G_{h_i} \subseteq Y_{CSP}$:
  $$\lambda(s_i) = Y_{h_i} \Rightarrow \text{OUT}(s_i) = G_{h_i} : Y_{CSP} \rightarrow \text{SKIP}$$

- The mapping results in the following recursive process description for each derived CSP process $P(s_i)$:
  $$P(s_i) = ((\text{compute}(s_i) \rightarrow \text{STOP}) \Delta \text{EXT}(s_i)) \uparrow^{(n)} (\text{OUT}(s_i); \text{INT}(s_i))$$

Figure 29. Formal Mapping of DEVS to timed CSP

The CSP process equivalent of the external transition function $\text{EXT}$ is based on choices between specific mapped CSP event arrivals with timed prefixes. As depicted in the final recursive process description the $\text{EXT}$ process is entered if an interrupt occurs in the respective process state description, i.e., external inputs arrive. The transformation of
the internal transition function and the output function into CSP process descriptions INT
and OUT is straightforward. These processes are executed when the time spent in the
CSP processes, which are derived from a former DEVS model states, exceed the value
specified by the former DEVS time advance function.

A formal timed CSP process description of this final recursive process description is
also shown in Figure 29. Here, the interrupt and the timed external choice operators are
derived as discussed from the semantics of DEVS external and internal transition
functions in order to produce an identical execution sequence of equivalent CSP
processes. The order of the output and internal transition function is resolved. Notice that
we have not considered the confluent transition function. Timed CSP leave the resolution
of external and internal event collisions to the implementation by the designer.

The presented mapping actually does not require any further modifications for
parallel DEVS coupled model specifications. The underlying assumptions for a CSP-
based [71] multi-component system specification match our restrictions we made for the
specification for our discrete event design models: an instantaneous propagation of
events, newtonian and real time (which matches our definition of simulation time),
maximal progress and parallelism (which matches our definition of modularity).

Since our abstraction of embedded systems behavior is a special case of system
model behavior this general mapping implies that we can map any of our design model
specifications to an implementation specification.
8.1.5. Example Mapping of a DEVS Atomic Model

A simple example for a conversion of a DEVS atomic model specification into timed CSP notation is shown in Figure 30. The model specifies a device with the following system specification: The application first resets itself in one time unit. The device then transitions to a waiting state. If a "ring" signal is detected in this waiting state it will perform some computation in a "busy" state. It stays in this state for 10 time units after which it will produce a "done" signal or until it receives a "drop" signal. In both cases the device again resets itself.

\[
\begin{align*}
\text{DEVS model component device} \\
X &= \{\text{ring, drop}\}; Y = \{\text{done}\} \\
S &= \{\text{wait, busy, reset}\}; s_0 = \text{reset}; \\
\delta_{in}(\text{reset}) &= \text{wait}; t_a(\text{reset}) = 1; \\
\delta_{in}(\text{wait}) &= \text{wait}; t_a(\text{wait}) = x; \\
\delta_{in}(\text{busy}) &= \text{reset}; t_a(\text{busy}) = 10; \\
\delta_{out}(\text{wait, e, ring}) &= \text{busy}; \\
\delta_{out}(\text{busy, e, drop}) &= \text{reset}; \\
\lambda(\text{busy}) &= \{\text{done}\}; \\
(\text{all other } \delta_{out}(s_i, e, x_k) &= s_i; \lambda(s_i) = \{\emptyset\})
\end{align*}
\]

\[
\begin{align*}
\text{// timed CSP process device} \\
P(\text{device}) &= P(\text{reset}) \\
P(\text{reset}) &= ((\text{compute}(\text{reset}) \rightarrow \text{STOP}) \Delta P(\text{wait})) \\
P(\text{wait}) &= (\text{STOP} \Delta (\text{ring} \rightarrow P(\text{busy}))) \\
P(\text{busy}) &= ((\text{compute}(\text{busy}) \rightarrow \text{STOP}) \Delta (\text{drop} \rightarrow P(\text{reset}))^{10}) \\
&\quad (\text{done} \rightarrow P(\text{reset}))
\end{align*}
\]

Figure 30. DEVS and timed CSP Specifications for Example Application "device"

The timed CSP process description presented in Figure 30 can be condensed in a single process description \( \text{DEVICE} = P(\text{device}) \) by resolving the mutually recursive process descriptions as follows:

\[
\text{DEVICE} = (\text{comp}(\text{reset}) \rightarrow \text{STOP}) \Delta (\text{STOP} \Delta (\text{ring} \rightarrow ((\text{comp}(\text{busy}) \rightarrow \text{STOP}) \Delta (\text{drop} \rightarrow \text{DEVICE})^{10} (\text{done} \rightarrow \text{DEVICE})))
\]
8.2. Implementation of Model Behavior in Software

A mapping to software is required if a design model component has been assigned to a shared processing element. This mapping has been based on the assumption that a small real-time operating system administers the execution of multiple processes. For the implementation of the specified model component behavior in the form of a software process we assume the availability blocking read and non-blocking send primitives. These primitives reflect our definition of basic behavior exhibited by computing systems, i.e., waiting indefinitely for external input arrivals in waiting states and immediately generating outputs in a processing model state.

8.2.1. Software Implementations

Software is encodes model component behavior in the form of programs for a processing element. Software implementations can be achieved using a variety of programming languages. These could be classified as follows:

a) processor configuration or micro code
b) assembler
c) domain specific programming languages (e.g., HandleC, SystemC, etc.)
d) procedural (so-called) high level programming languages (e.g., Fortran, C, Visual Basic, etc.)
e) object oriented languages and functional languages (e.g., C++, Java, Lisp, etc.)

The attainable execution performance of software implementations drops in the order of the presented software classification. Class 2 implementations are often used for the implementation of small computational segments with high performance constraints, and class 3 or 4 in other cases. Tools for class 5 software implementations are at this point in
time not mature enough to generate machine code efficient enough for small, hard real-time embedded systems applications. The quality of the generated code from the higher level programming languages is also dependent on factors like the compiler optimizations, the coding style of the programmer, the efficiency of available library components, etc.

A choice for the software implementation of a particular model component from this list is strongly dependent on the processing element and the execution performance required from the behavior to be implemented. In general, it is difficult to suggest a general choice in implementation languages. A choice is really only available for conventional processors. In practice, a selection is also constrained by the already available tool set.

8.2.2. Construction of RTOS processes

The conversion into a software implementation is achieved by transforming each design model component specification into a RTOS process. For each model component computational segments descriptions are isolated and implemented in a selected programming language. Each computational segment is bounded by a read primitive and send primitive. The external inputs or output events specified for the former waiting state and output function are encoded as messages references in these primitives. Model state transitions determine the order in which the implemented segments have to be placed and executed in each process realization.

Figure 31 shows a software implementation of the previous DEVICE design example in some generic high-level language. Notice that the signal "drop" which was received as
the external input in the processing model state "busy" is here handled as an processor
interrupt routine. The assertion of this interrupt causes the RTOS to kill and then invoke
again the DEVICE process.

```c
init_process_Device( ) {
    DeviceMB = RTOS_Create_MailBox();
} // end init_process_Device

interrupt drop {
    RTOS_Reboot( process_Device );
} // end interrupt drop

process_Device( ) {
    int ringValue, doneValue = 0;
    while (true) {
        RTOS_Delay(1);  // reset state processing
        RTOS_Read( DeviceMB, ring, &ringValue);
        RTOS_Delay(10);  // busy state processing
        RTOS_Send( someOutMB, done, doneValue );
    } // end while
} // end Device
```

Figure 31. Software Implementation of Example Application "device"

8.3. Implementation of Model Behavior in Hardware

Custom processors are one form of a hardware implementation for model component
behavior. Here, we need to create a custom instruction set for the processor control unit
[78] which encodes model instructions used in the description of the behavioral aspect.
The implementation of custom hardware elements for a processing platform is much
simpler. We create synchronous combinatorial circuits for either model instructions or algorithms [89].

A circuit specification can be either achieved by using graphical design entry, i.e., a layout of the circuitry based on available logic components or library cells, or a textual specification based on a hardware description language (HDL). We suggest the use of the latter method since it allows at least a partial automation of the transformation process.

At first glance many hardware description languages look like standard programming languages. There are some fundamental differences in the specification of a hardware design, e.g., that the designer has to base the application design on a set of predefined hardware elements and basic operations instead of instructions. Also only a subset of a hardware description language is usually synthesizable. Many programming constructs have been introduced to these languages to facilitate the efficient construction of design test benches for simulation purposes. Nevertheless they are not implementable. The usable subset of a HDL is usually limited to the standard simple arithmetic operations and basic data types in a behavioral description.

The automatic generation of RTL level design specifications from a system level behavioral descriptions, which is commonly referred to as behavioral synthesis, has been a topic of intense research for the entire last decade. Many attempts to find an efficient but yet general approach for such a transformation have failed [52]. We only intend to provide some guidelines for the implementation of model behavior in hardware but do not claim it to be most efficient. Similar as in high level software programming
languages, the efficiency of design implementations from HDLs is also strongly dependent on the expertise of the designer in hardware design.

8.3.1. Guidelines for possible Hardware Implementations

A transformation of model component behavior into a hardware implementation can be achieved using a state machine based hardware description which uses sequential process descriptions. A state machine based hardware implementation for our previous example is shown in Figure 32. Here, we proceed in a similar manner as discussed in Section 8.2.2, e.g., we convert external inputs into variables which are placed in the sensitivity list of a VHDL process. For a hardware implementation we also need to create a component which keeps track of time and can be referenced for delays in the generation of outputs, i.e., a timer module.

Another approach for the transformation of more complex specifications is to derive control flow and data flow diagrams from a model component specification. Control flow diagrams can be constructed from the specified model component state transitions while a static model component data analysis can be used to extract the data flow within a model component. This approach basically advocates well-known design techniques for a custom processor [78].

The transformation of model components also is more intricate when required communication with other architectural elements exceeds the level of mere signals. A solution to this problem is offered in [4] by Eles et al. who present an extension to the VHDL which introduces send and receive primitives.
-- VHDL Specification of "device" (drop and ring declared as "in", and done as "out" signals as part of previous entity declaration)
ARCHITECTURE behavioral_description OF device IS
    TYPE COMPONENT_STATES IS (reset, busy, waiting); -- declare hardware element states
    SIGNAL state : COMPONENT_STATES;
    VARIABLE delay : INTEGER RANGE 0 TO 10;
BEGIN
    PROCESS (elk, drop, ring) -- clk signal is assumed to arrive every time unit
        BEGIN
            IF drop = '1' THEN -- specify handling of the "reset" signal / interrupt
                state <= reset; delay := 0;
            ENDIF;
            IF ring = '1' THEN -- specify transition from waiting states
                state <= busy WHEN state = waiting;
            ENDIF;
            IF (clk'EVENT AND elk = '1') THEN -- specify processing states and corresponding transitions
                delay := delay - 1; -- based on common clock for delay measure
            CASE Slate IS
                WHEN reset =>
                    IF ( delay > 1 )
                        state <= waiting; delay := 0;
                    ENDIF
                WHEN busy =>
                    IF ( delay > 10 )
                        done <= '1';
                        state <= reset; delay := 0;
                    ENDIF
            END CASE;
            END IF;
    END PROCESS;
END behavioral_description;

Figure 32. VHDL Specification of application example "device"

8.4. Implementation of Communication Interfaces

Another result from the model compilation phase are communication constraints for the processing and hardware elements. In the implementation phase of the design, model physical communication channels and protocols need to be selected and configured to comply with these derived constraints. Here, the evaluation of collected data on
processing-processing element or processing-hardware element communication, e.g., available communication windows or bus collisions, essentially restricts the choice in the selection of available communication components or bus arbitration patterns. In addition, communication components needs to be adapted to the particular processing platform, e.g., an processing element addressing scheme has to be devised. Software-software communication, i.e., the interface between virtual processors, needs no further attention since efficient communication primitives have already been implemented in the RTOS.

8.5. Verification at the next Level of Abstraction

The transformation of model component behavior should be followed by yet another validation phase at the implementation level. The computed execution times for computational segments by definition present merely a deadline which - if met - guarantees that the desired behavior will be exhibited. The underlying assumption here though is that all architectural element parameters accurately reflect their performance. An assessment of these parameters at the implementation level can verify the correctness of the final application implementation.

Design model component implementations can be first tested in isolation and then within the integrated, physical application prototype. The correct behavior of single component implementation can be achieved, e.g., using instruction set simulators for the selected target processor for software implementations [61]. Similarly, hardware component specifications can be verified at the RTL level using the simulation tools included in their development environments [21]. The following chapter will illustrate a
testing approach for mixed, integrated application prototypes which is based on multi-level testing.
9. TESTING IN MODEL-BASED CODESIGN

In this dissertation we have presented application development mainly from the design perspective. Although testing received some attention at model level in the form of experimental frames we want to present here the bigger picture which discusses testing in the context of our model-based codesign methodology.

9.1. Design And Test

As pointed out in [35,66] structured approaches to the testing of embedded systems design is far away from being practiced in industry. Although a lot of research has addressed testing in the context of domain specific methods, the testing of embedded application designs composed of heterogeneous components is still to a large extent performed ad hoc. We refer to this kind of testing as integrated system testing. The term "system" should here be understood as an application implementation which can consist of multiple, heterogeneous components.


Testing methods and objectives differ in the hardware and software domain. Embedded software development uses specialized compilers and development software that offer means for debugging. Future on-chip debug support (OCDS) promises to improve software performance and estimation and analysis [90]. The testing of hardware components concerns itself mainly with the verification of functionality and self tests of chips [91]. Self tests mainly ensure proper operation after the implementation of these circuit models.
The testing of embedded systems implementations is usually performed separate hardware and software development groups, and possibly even integrated by a third group. We argue that this conventional approach to testing is incomplete and focuses merely on two critical aspects of implementation testing separately. It fails though to address the embedded systems composed of multiple heterogeneous component, e.g., mixed hardware/software codesign implementations, in its entirety.

Failures discovered in the testing of the final integrated system prototype are in general a time for tough decisions. For applications with high performance requirements and safety constraints, such an event is of major concern since we have to guarantee a well tested and debugged final product. In theory, unsatisfactory performance of the system under test though should lead to a redesign of the application. In practice though, every additional design iteration increases application development cost and more importantly causes a delay in the time-to-market for the product, which may endanger the success of the application. For these reasons we advocate the use of design methodologies for such applications which emphasize early design assessment and enable integrated system testing.

To date, integrated system testing has only received attention in the research community as "co-verification" [45] which simulates entire mixed system prototypes on powerful computer systems. The basic idea here is to link already existing hardware development tools with a software development environment. This can be achieved by interleaving implementation of the simulation engine, which is part of the hardware development environment, with debugger software, which is part of the programming
environment. With this information co-simulation engine can coordinate the proper conjoint execution of these development environments.

9.1.2. Model Testing in System Modeling

A modular, system level application specification offers an ideal means for the integrated system testing of an application design. Our model-based codesign methodology enables such specification. As we have introduced in the previous chapters, model testing can be fairly easily integrated by encoding testing scenarios in special testing model components or modules in experimental frames, and assessing the design model by connecting it to these frames. This general approach to model testing can be applied at various levels of design model resolution.

Notice again though that we face a similar problem of model continuity when we transition from the model level to the implementation level. This transition though holds the key to some important benefit. Given such a transition we can reproduce model level test scenarios at the implementation level. It also allows a reuse of the entire test base from the model level and an automatic generation of implementation level tests. Most importantly though the resulting test scenarios achieve consistency in our testing of the application design.

9.2. Multi-Level Testing

While working on a design methodology for mixed embedded systems, we noted a lack of research in the early verification of system prototypes and started working on such an integrated system testing approach based on system modeling concepts. Following our
development of embedded systems applications in model-based codesign we devise a multi-level testing approach which addresses testing from the conceptual phase of the application development to its final design implementation as shown in Figure 33.

![Figure 33. Multi-Level Codesign](image)

We emphasize an early separation of test and design at the model level. Designers use simulation to explore the feasibility of virtual prototypes at different levels of resolution using testing modules which are based on the theoretical concept of experimental frames. At the model level the design is implementation independent. Here, testing modules verify design models against application system specifications, and monitor their validity as well as component integrity.
The refinement of these testing modules follows the refinement of the design model. Eventually we convert test modules into test processes in a test module transformation. At the implementation level integrated system testing is then finally realized in a real-time testing environment. It enables us to verify specified performance constraints on reconfigurable physical application design prototypes.

A conjoint, multi-level design and test approach offers a gradual transition for embedded systems development from the application model to its implementation. It allows the specification and application of test scenarios during all the stages in the application development where changes can easily be propagated to lower as well as higher levels of abstraction. Contrary to conventional approaches, the proposed multi-level testing offers the following advantages:

a) consistent generation of scenarios throughout all levels of testing
b) an early assessment of alternative design implementations
c) component as well as integrated system testing
d) real-time system performance measurements which are necessary to evaluate different, mixed hardware/software application implementations

9.2.1. Design Model Testing

Although a number of system modeling tools exist only little published research directly addresses model testing at the system level [18, 73]. In our model based codesign methodology we address design model testing with a concept which was previously introduced as experimental frames. In this chapter, we refer to model components which are part of such frames as testing modules. They model the environment that our
application is embedded in. Test modules encode test scenarios, which can be fixed or interactive[77], from specified design requirements. For the construction of testing modules a set of fixed test scenarios for the system model is created from the system requirements specification [66,68]. Test modules can be classified as generating, monitoring, or analyzing model components which are created separately from the design model. These components basically represents different parts of the environment which stimulate or monitor behavioral requirements of the design model in its deployment.

During application development these modules and their interfaces follow the incremental refinement of the design model. The simulation of the test modules together with the system model represents an experiment where the application interacts with its environment. In essence, we create a test bench at the model level which is used to validate various aspects of either the entire application or components of it. A modular specification of test modules allows their reuse in different applications or in design alternatives of the same application. As discussed in the previous chapters testing modules are our vehicle for a design model analysis which stimulates the application design and collects (and sometimes already processes) model information which includes the application response.

9.2.2. Integrated System Testing

In product testing, the application is then tested in the realistic operating conditions, i.e., physical time as opposed to simulation time. We assume about our application prototype that software components have been fully debugged and that target architecture has already been tested for integrity. The former design model components have been
mapped to this architecture: board design and communication architecture have been realized based on the results of a successful model compilation, hardware elements have been manufactured or configured, software processes have been generated, and the scheduler of the operating system has been configured properly. Prior to the integrated system testing each individual element should been already tested using conventional testing approaches.

We follow the model compilation of our design model specification into a heterogeneous application implementation with a similar transition for testing modules [92]. Here testing modules are converted to a set of test processes for a real-time system testing platform. At the implementation level we can then observe the interaction of the various components of the application prototype realistically in conjunction. Test processes subject the embedded application system prototype to the previous testing scenarios in real-time.

During the test runs, the system testing environment records and analyzes the response and other execution parameters of the prototype, e.g., utilization of communication links, processing element idle times. Operating in real time we are able to assess true system performance results from the implementation. When the application prototype complies with the specified application performance requirements the design can be then handed over to production and testing scenarios to product testing.

9.3. A Real-Time System Testing Environment

A real-time system testing environment in essence realizes our transition from simulation time to real time and inserts another level of testing between the model and product
testing. Here, we can leverage an already accumulated repository of test scenarios in the form of test modules and reuse them in an integrated system test of the application prototype.

The modeling level has allowed us an early assessment of design performance requirements using a set of fairly good execution estimates. These may not suffice to accurately verify application implementations which have high utilization of processing elements. Such implementation are generated when the processor cycle time in a dedicated processor implementation is fairly close to the original cycle time value of a model component or if the idle time of shared processors is low.

To obtain true performance measures we apply test scenarios generated by the test modules to a physical realization of the system model. Such a system prototype is derived using model compilation and implementation techniques. Such a prototype may be implemented on a reconfigurable processing platform which consists of standard processing elements, and reconfigurable hardware components. A small operating system which coordinates the communication of the software components and creates an efficient interface with the testing environment.

9.4. The Implementation of the System Testing Environment

We have developed a real-time system testing environment (STE) which was implemented for a standard personal computer. The software was designed to minimize processing overhead in the generation of stimuli for the application under test and allows an accurate evaluation of its response.

As shown in Figure 34, the STE software is composed of:
• test processes
• a kernel based on the minimal real-time operating system μC/OS [93]
• a process management layer which handles scheduling, inter-process communication, test analysis, real-time compliance of experiments, etc.
• a high bandwidth communication protocol
• general user interface for test data analysis

Figure 34. Implementation of the STE

The STE has been designed to operate based on a set of test processes which execute on top of a real-time operating system platform. The architecture of this testing environment was intentionally kept fairly general to provide a general testing tool for a variety of embedded system applications. Test processes can be created as generator, monitor, or performance processes.
9.4.1. STE Test Processes

Generator processes are created for each test module which generates outputs. During the experiment, these processes reproduce the input portion of test scenarios in real-time from test scripts which match the specified behavior of corresponding test modules of previous simulation runs. Scripts consist of time stamped messages which are sent to the system prototype via the STE communication channel.

The monitor process is derived from the transducer test modules. During testing this process verifies the correctness of the content in the system prototype response against the results of previous collected data or specified constraints on its values.

The performance process verifies the timely behavior of the system behavior form its response and gathers also information about the execution behavior of system prototype if such an option is supported by the operating system on the application prototype. Duplications of such a process may here also instantiated directly on the system prototype to gather system components performance data. Raw performance measures could be recorded during the execution of the application and then reported during idle times of the system prototype. All of these processes have the ability to interrupt the experiment in case of significant deviations from specified behavior or invalid system responses.

During the experiment the central process manager coordinates the STE process communication within the testing environment and with the system prototype, and records incoming and outgoing data. The monitor process tracks the system response
while the performance process collects run-time information of specified components on the physical prototype.

9.4.2. Conversion of Test Modules into Test Processes

The conversion of test modules into test processes can be achieved using one of two possible approaches. The simplest approach is to record each event at the output and input ports of the experimental frame along with a time stamp in a simulation output file during model level testing. Then equivalent test conditions can be recreated with a set of generic real-time processes which generate corresponding messages based on the output events in the simulation output file. Similarly monitor processes can be generated for the recorded input events. This method allows us to repeat test scenarios from model testing in an identical.

We may also use these test processes to specify any other desired testing scenario at the implementation level. The benefit of such an approach is that it is very efficient in the production and reception of stimuli which can include large amounts of data transferal between the environment and the application. For smaller amounts of data we can also allow real-time interaction of the user with the application can also be tested using this approach by reading keyboard inputs instead of script files.

In some cases we may want to only provide some inputs by the user interactively while the other - possibly large - portion of the testing stimulus should be generated by another physical component. As discussed in previous chapters and in [77] we can create such settings at the model level. Here, the second approach is to treat the test modules in the same manner as a design model itself. In order to compile model components or
testing modules of the experimental frame we have to restructure each behavioral model component specification in such a manner that it follows our model of computation, i.e., as a collection of waiting states and processing states (see Chapter 4). In all experimental frame components we will set the model component cycle time to the processor cycle time of the processor which runs our testing environment. Similar as for the design specification these testing modules will then have to be compiled to an implementation.

9.4.3. Communication Protocol

An efficient communication channel to the system prototype was realized by the implementation of an efficient protocol which operates via from the PC parallel port, called EPIC [94]. The protocol is tightly integrated with the STE inter-process communication as well as the operating system running on the target application. The compact design of the STE allows for the execution of system tests with a control resolution of microseconds on a regular PC. To minimize run-time overhead the STE user interface is not active while experiments are in progress. The user interface can be used to manage the test environment and directly access generated test data.

9.5. An Integrated System Testing Example

In a case study, an autonomous, intelligent cruise controller (AICC) [95,96] was designed using our model-based codesign methodology. The AICC system is an extension of the regular automotive cruise control which does not only keeps a fixed speed, but also adapts to the speed of a lead vehicle. Our example focuses on the design of the control unit of this device, which interacts with sensors and actuators in the vehicle. Behavioral
requirements for the control unit include keeping the vehicle speed within a narrow margin of error and having a minimum response time when interacting with other components of the AICC system.

Multi-level testing was applied in the design process of this unit. A model of the environment was created to represent the vehicle under different driving conditions, and to provide a user interface to the driver. Testing scenarios for the STE were directly recorded from the interface of the test modules. Converted into test processes they were used to evaluate the performance of three selected, mixed hardware/software implementations of the control unit.

9.5.1. AICC Test Modules

The test modules for the AICC system are designed to create the environment that the unit is embedded in. One approach is to decompose the computing environment into five model components as shown in Figure 35: a sensor, vehicle, bus controller, monitor, and graphical user interface (GUI) component. The sensor component provides the model with information about the lead vehicle speed and distance. The vehicle component models the behavior of the car engine. It derives the current speed from throttle and brake positions computed by the AICC control unit. The monitor component checks the validity of the AICC response, e.g. that the speed reading is within the specified margin. The GUI and bus controller components are interfaces to the driver and the car that convert information into appropriate internal data formats for other model components.
We notice in this example that vehicle component is more than a mere generator module modeling the environment. It uses the response of the AICC control unit as feedback in its calculation of the next speed value. At the model level, the vehicle component can assume the properties of any car engine, such as an acceleration or deceleration pattern. The car test module can be enhanced by introducing additional model components describing, for example, incline, curvature of the road, or weather conditions.

Evaluation of simulation results of our first design attempt [77] indicated that our AICC design was unstable: Instead of a smooth acceleration to the desired coast speed the control unit oscillated between acceleration and deceleration when the vehicle got close to the desired speed. While tracking the system state to locate the design flaw would have taken considerably more effort in the implementation phase it was in this case fairly easy to accomplish in a high level simulation of the system model.
9.5.2. The Conversion of the AICC Test Modules into Test processes

For most of the test modules, the conversion the mapping to test processes is trivial since they constitute already generating test modules at the model level. In the case of the vehicle component we map the module into both generator and monitor processes. The stimulus produced by this component was converted into a test script, while its recorded response was encoded into the monitor process. Real-time processing requirements for the AICC are encoded into the performance process.

The AICC design model specification was then mapped to a selected target architecture which consists of a micorcontroller and a FPGA. The conversion of testing modules to test processes and design components into a mixed hardware/software implementation is depicted in Figure 36. Here, the design model is shown to be implemented as a set of software processes where a custom hardware function is
accelerating the execution of state manager process. In addition, the operating system management and communication routines are also shown as processes.

9.5.3. **System Testing Results for Different Hardware/Software Implementations**

The previously introduced STE was used to assess three different design alternatives for the AICC control unit: an all software solution employing a Motorola 68HC11 microcontroller, a mixed Altera Field Programmable Gate Array (FPGA) and 68HC11 solution, and an all software solution based on a Infineon C161O microcontroller. The following tables show the obtained results in two selected test scenarios: A simple test scenario A with 11 input events, and a more computationally intense scenario B involving 108 test messages. The observed data was averaged over consecutive test runs for each test scenario.

<table>
<thead>
<tr>
<th>Test Scenario</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcontroller</td>
<td>68HC11</td>
<td>C161O</td>
</tr>
<tr>
<td>Avg. Response Time in μs</td>
<td>28</td>
<td>3</td>
</tr>
<tr>
<td>Total Idle Time versus Total Test Time in s</td>
<td>2.665</td>
<td>2.791</td>
</tr>
<tr>
<td>% Utilization of microcontroller</td>
<td>5.62</td>
<td>0.49</td>
</tr>
</tbody>
</table>

Table 3. All Software Implementation Comparison

Table 3 shows the performance of the two processing environments for both test scenarios. "Average response time" refers to the average time the AICC control unit takes to return of throttle and brake positions. Initially this response time was constrained to be less than 100 ms.
The results show that the Infineon microcontroller implementation dominates both performance tests in all categories. The computational advantage is especially noticeable in the average response time which is 10 times faster for the C1610. This is not surprising since the C1610 is a 16-bit pipelined microcontroller running at twice the speed of its 8-bit counterpart. Our data suggest that the Infineon microcontroller is powerful enough to handle many more computational tasks which would help reducing the number of processors in a vehicle.

9.5.4. Analysis of Test Data over Time

The test data can also be used to show performance differences over time, e.g., in the processor load of the alternative designs. In Figure 37, the microcontroller utilization in 100 ms time intervals for all configurations is plotted against the test time for test scenario B.
In test scenario B the AICC control unit is activated at 2.7 seconds of test time. The car reaches its coast speed 1.8 seconds later, resulting in a decrease in computation. The control unit is disabled at 6.1 seconds of test time. In its active state the control unit frequently performs throttle and brake position computations, leading to an increase in microcontroller utilization. In its disabled state it only performs basic data management tasks.

Again, the C1610 outperforms any 68HC11 configuration since utilization stays consistently below 4.2% during the entire test scenario. The 68HC11 data series for the software-only implementation indicates a performance increase after activating the control unit which reaches a maximum of 56% processor utilization. We note that the hardware/software implementation improves the performance of the application during
the active period of the AICC, where the 68 HC11 controller reaches merely a maximum utilization of 45%. Although this is a noticeable increase in performance, our results show that even a mixed hardware/software configuration is not able to compete with the more powerful C161O.
10. THE APPLICATION OF MODEL-BASED CODESIGN CONCEPTS

Finally we present an example of the application of the model based codesign concepts which were discussed in the previous chapters. Up to this point we have been concentrating our efforts on presenting a general approach to the model-based design of embedded real-time systems which could be used with a variety of system level modeling specifications [7,13,19,23,71]. In this chapter, we apply the previously presented concepts in the design model construction, refinement, analysis, compilation, and implementation of a simple application based on pDEVS (see Section 4.8).

Over years of research on model-based codesign a number of more complex embedded systems applications have been implemented using our methodology which include a heating system [88], room temperature controller [64,66], autonomous intelligent cruise controller [38,39], embedded Java virtual machine [61], and more [66]. Next to the presented alarm clock example also a more involved elevator controller application has been studied in the course of research on this dissertation. Nevertheless, we feel that a simple, well-know application is better suited for an illustration of the entire proposed model based codesign process.

10.1. Design Model Construction and Refinement from a System Specification

We present first the design model construction for our selected application from a textual system specification. After a brief informal specification of the device we continue with the illustration of a correct stepwise refinement across various levels of abstraction by introducing behavioral and structural requirements.
10.1.1. The Alarm Clock System Specification

We assume the following system specification for the alarm clock: The time is to be kept in 24h style during all operating modes based on a processor or PCB clock. There are four modes of operation which encompass the display of the current time, the setting of the current and alarm time, and the triggering of the alarm. The user has a time, alarm, hrs, min, and alarm-off buttons to control the alarm clock. For the implementation of this device a standard LED display and generic microcontroller are available.

In its initial operating mode, the device should display the current time. In the display mode a digital digit display is to be updated within the second that the current time (hours or minutes) changes. If the time button is asserted the alarm clock enters the set-time mode. Here, the current time can be modified by asserting the min and hrs buttons. A second assertion of the time button returns the alarm clock the display mode. The same behavior is exhibited when the alarm button is asserted with the difference that the alarm time can be displayed and modified. A second assertion of the alarm button enables the alarm and returns the device to the display mode. When the alarm time is reached the device enters asserts an alarm which has to be disabled by the user using the alarm-off button. Any other input scenarios may be ignored. The current time should be kept during any mode of operation.

This informal description of the alarm clock serves as our system specification which basically consists of functional and implementation requirements. We assume that alarm clock display and response in set-alarm and set-time operation modes should occur within
0.01 seconds - "instantaneous" - to the user. Further we assume the arrival of clock pulses every minute from a timer peripheral device, e.g., on the microcontroller chip.

### 10.1.2. Abstract Design Model Construction for the Alarm Clock

In the following discussion model state diagrams illustrate the described refinement of the design model. Arrows indicate transitions between model states where double-headed arrows are used to simplify the illustration of state transitions which return to their original state.

![Figure 38. Abstract Initial Design Model and Type I Input Arrival Scenario](image)

We start the development with a single abstract design model component $M_{AC}$ as shown in the first part of Figure 38. This abstract model specification waits first for either the arrival of a button from the user or a clock pulse from the timer in $W_{AC}$. After the arrival of each clock tick $M_{AC}$ transitions to the processing state $P_{AC}$ and then generates either some external outputs for the LED display. An initial processing state which initializes model component data structures has been omitted from this illustration.

Since the button input may arrive aperiodically, i.e., at any point in time, we derive a the worst case testing scenario, i.e., a stress test, which is depicted as a model execution
trace in the second part of Figure 38. Such a scenario is the conjoint arrival of a button and tick event, i.e., a Type I input arrival scenario. Notice that the external inputs from the environment to $M_{AC}$, i.e., the user and the timer, are buffered with communication components to allow a sequential processing in such a scenario for $P_{AC}$. We select the required execution time to be the stated response time, i.e., $ta(P_{AC}) = 0.01$ seconds. Based on an average estimated instruction count (IC) for $P_{AC}$ of $IC_{AC} = 10$ we compute a first model component cycle time (MCT) of $MCT_{AC} = 0.001$.

10.1.3. Correct Refinement of the Alarm Clock Design Model

The first refinement of the initial design model results in the single model specification shown in the first part of Figure 39. Here, we have first introduced the behavior requirement which concerns the operation modes of the alarm clock. Here, $P_{SA}$ resembles the setting of the alarm time, $P_{ST}$ that of the current time, and $P_{KT}$ keeps track of time from each arrival of clock pulses. Here, we can specify the behavior of $P_{KT}$ in more detail by specifying in its computational segment the manipulation of time based on the external tick input. Lets assume revision of its instruction count to $IC_{KT} = 9$. The instruction counts for $P_{ST}$ and $P_{SA}$ can be left as $IC_{ST} = IC_{SA} = 10$ since they remain abstractly specified. Since there is a data dependency between $P_{KT}$ and $P_{ST}$ (they both modify or access information about the current time) we revise $MCT_{AC} = 0.01/(IC_{KT}+IC_{ST}) = 1/1900$. 
Figure 39. Design Model Specification and Trace after First Refinement Step

The second part of Figure 39 shows the model execution trace of the refined model specification using the previous Type I test scenario. Notice that the button event has been refined to specific buttons in a communication refinement. Our worst case scenario can now be more accurately described as the tick event arriving just before a time or alarm event. The system response in the reverse case (a button prior to a pulse) remains well within the specified bounds, e.g., \( \tau_a(P_{SA}) + \tau_a(P_{KT}) = 0.01 < 1 \).

In our second refinement step, we perform a structural refinement triggered by introducing the requirement that time is to be kept in parallel at any point of the alarm clock operation. The resulting composed design model specification is shown in part of in Figure 40. In the structural refinement, we isolate state \( P_{KT} \) in a new design model component specification \( M_{KT} \).

Now each modification of the current time now needs to be communicated explicitly in \( P_{KT} \) and \( P_{ST} \) to the other model component specification. Since the time event is an may arrive at any point in time we can introduce in this case some extra model states to address this data dependency explicitly The state \( P_{SC}, P_{UC} \) handle send or update the current time in \( M_{TK} \), whereas \( P_{GCT}, W_{CT} \) request and wait for this information in \( M_{AC} \).
Notice that state $P_{ST}$ triggers in $M_{AC}$ the update of the current time in $M_{TK}$. We simplified our illustration of this refinement by omitting $P_{SA}$ from the diagram.

Notice that an alternative structural refinement following the procedures described in Section 5.3.4.1 is also possible. But since the button input is of aperiodic nature the communication overhead between $M_{AC}$ and $M_{TK}$ considerably reduced by introducing these extra model states.

Since $P_{GCT}$ and $P_{ST}$ are directly involved in our stress test we decrease both cycle times to $MCT_{AC} = MCT_{KT} = 1/2300$ (assuming that $IC_{GCT} = IC_{SCT} = 2$). The introduction of these additional processing states changes our worst case scenario to $t_{\text{time}} = t_{\text{tick}} + ta(P_{GCT})$ as depicted in Figure 41. An analysis of the refined design model specification shows that due to the required synchronization of $M_{AC}$ and $M_{KT}$ our design model results actually in a slight performance loss performs as compared to our single component design.
In our third refinement step (the results of which are shown in Figure 42), we decide to further refine the behavior abstracted in $P_{ST}$ by introducing the functional requirements for the modification of time. After initializing the LED display in the remaining behavior of $P_{ST}$ which we will label $P_{ST}$, we introduce the waiting state $W_{MT}$ to wait for the next
button, and processing states $P_{\text{MIN}}$, $P_{\text{HRS}}$, and $P_{S}$ to modify minutes or hours, and save the updated current time. The update of the current time in $M_{\text{KT}}$ is implemented in the processing state $P_{\text{UCT}}$.

From our specification of detailed computational segments for these states we derive instruction counts of $IC_{\text{ST}} = 6$, $IC_{\text{MIN}} = 3$, $IC_{\text{HRS}} = 3$, $IC_{S} = 4$, and $IC_{\text{UCT}} = 3$. Notice that all processing states of $M_{\text{AC}}$ (if we ignore $P_{\text{SA}}$ for a moment) have now smaller execution times than required by their respective behavioral constraints, i.e., $ta(P_{\text{ST}}) < ta(P_{\text{ST}})$, $ta(P_{\text{MIN}}) < 0.1$, $ta(P_{\text{HRS}}) < 0.1$, $ta(P_{S}) < 0.1$. We can therefore increase $MCT_{\text{AC}}$ using a model component cycle time analysis to $MCT'_{\text{AC}} = (1 + \frac{t_{\text{min}}}{IC_{\text{MAX}}\cdot MCT_{\text{AC}}})\cdot MCT_{\text{AC}} = \frac{1}{1380}$ where $t_{\text{min}} = ta(P_{\text{ST}}) - ta(P_{\text{ST}}) = 4/2300$ and $IC_{\text{MAX}} = IC_{\text{ST}} = 6$. The model execution trace of the worst case testing scenario remains the same for the simultaneous arrival of time and tick as previously shown in Figure 41. In all other cases a parallel execution of $M_{\text{AC}}$ and $M_{\text{TK}}$ occurs.

The refinement of $P_{\text{SA}}$ follows in the same procedure. Notice that in our previous structural refinement similar to $P_{\text{ST}}$ any change in the alarm time or status will have to be explicitly communicated from $P_{\text{SA}}$ to $M_{\text{KT}}$.

10.1.4. The Shared Resource Problem

Although we have correctly refined our design model in the previous section, a resource sharing problem arises in our design model after the structural refinement into $M_{\text{AC}}$ and $M_{\text{KT}}$. Here, $M_{\text{KT}}$ happily continues to send its updated current time to the LED display during the composed design model execution while the user is trying to set this time.
Where as in a single component design model this problem can be solved fairly easily by introducing global variables, we have two alternatives to resolve the situation in a modular composed design model. The first approach is to introduce an output flag to \( M_{KT} \) which is cleared in the \( P_{SCT} \) state. Similarly, it is set after the arrival of the new valid time in \( P_{UCT} \) of \( M_{TK} \). This request can then be used in \( M_{TK} \) to set a flag which indicates if external outputs should be generated for the display. Alternatively, we can introduce a third model component which acts as a multiplexer for the information generated by the respective output ports of these two components which is shown in Figure 43.

![Figure 43. Abstract Model Component Specification of LED Manager](image)

10.1.5. Implementation and Assessment of Design Model Specifications

We may our design model specification using one of many discrete event modeling specifications. As pointed out before the parallel DEVS formalism provides the capability to specify design models as well as their experimental frames.

In our case, the alarm clock example was implemented and tested using pDEVS - an Java-based implementation of the parallel DEVS formalism - to implement models and obtain the results used in the following sections. The code for its design model components and some simulation results are listed in Appendix A.
Here, waiting states are specified in the external transition and processing states in the internal transition function. Notice though that the specification of the behavior in waiting states has been split into a waiting state and a collecting (or trigger) state with zero execution time (see Chapter 4). The generation of outputs for each processing state is defined in the output function. In addition, a generic initial model state "processInit" is specified in each model component specification to initialize model component data structures properly. It transitions to the initial waiting state specified in the previous diagrams. Finally, the correct implementation of interactions with communication components requires also the introduction of some additional code.

10.2. The Final Design Model Specification

After the complete refinement of our alarm clock design model specification we arrive at composed design model which consists of the two components $M_{AC}$ and $M_{KT}$ with a total of 6 processing and a single waiting state, and 7 processing and 3 waiting states. Instruction counts for computational segments vary between 2 (for $P_{GCT}$, $P_{GAT}$ in $M_{AC}$, and $P_{DA}$ in $M_{KT}$) and 13 instructions (for $P_{KT}$ in $M_{KT}$).

The data table, which was obtained from a static model component analysis of $M_{KT}$, shows in Table 5 that all computational segments in the design specification have been refined to the lowest attainable level, i.e., they operate and communicate at the level of basic data types. From the evaluation of this table, only $P_{KT}$ and $P_{ST}$ qualify for further increase in design model resolution. Nevertheless, the benefits from such a refinement are questionable in this case.
<table>
<thead>
<tr>
<th>Model State Name</th>
<th>IC</th>
<th>Data Item</th>
<th>Size</th>
<th>R/W</th>
<th>FA</th>
<th>LA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Init</td>
<td>6</td>
<td>Timemin</td>
<td>4 (byte)</td>
<td>W</td>
<td>0</td>
<td>.16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Timehour</td>
<td>4 (byte)</td>
<td>W</td>
<td>.17</td>
<td>.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AlarmEnabled</td>
<td>2 (boolean)</td>
<td>W</td>
<td>.34</td>
<td>.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Alarmmin</td>
<td>4 (byte)</td>
<td>W</td>
<td>.51</td>
<td>.66</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Alarmhour</td>
<td>4 (byte)</td>
<td>W</td>
<td>.67</td>
<td>.84</td>
</tr>
<tr>
<td></td>
<td></td>
<td>send2display</td>
<td>2 (boolean)</td>
<td>W</td>
<td>.85</td>
<td>1</td>
</tr>
<tr>
<td>Keep Time</td>
<td>13</td>
<td>Timemin</td>
<td>4 (byte)</td>
<td>W</td>
<td>0</td>
<td>.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Timehour</td>
<td>4 (byte)</td>
<td>W</td>
<td>.08</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AlarmEnabled</td>
<td>2 (boolean)</td>
<td>R</td>
<td>.54</td>
<td>.62</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Alarmmin</td>
<td>4 (byte)</td>
<td>R</td>
<td>.63</td>
<td>.69</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Alarmhour</td>
<td>4 (byte)</td>
<td>R</td>
<td>.70</td>
<td>.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>send2display</td>
<td>2 (boolean)</td>
<td>R</td>
<td>.85</td>
<td>.91</td>
</tr>
<tr>
<td>Update Current Time</td>
<td>3</td>
<td>Timemin</td>
<td>4 (byte)</td>
<td>W</td>
<td>0</td>
<td>.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Timehour</td>
<td>4 (byte)</td>
<td>W</td>
<td>.34</td>
<td>.66</td>
</tr>
<tr>
<td></td>
<td></td>
<td>send2display</td>
<td>2 (boolean)</td>
<td>W</td>
<td>.67</td>
<td>1</td>
</tr>
<tr>
<td>Update Alarm Time</td>
<td>4</td>
<td>Alarmmin</td>
<td>4 (byte)</td>
<td>W</td>
<td>0</td>
<td>.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Alarmhour</td>
<td>4 (byte)</td>
<td>W</td>
<td>.26</td>
<td>.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>send2display</td>
<td>2 (boolean)</td>
<td>W</td>
<td>.51</td>
<td>.75</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AlarmEnabled</td>
<td>2 (boolean)</td>
<td>W</td>
<td>.76</td>
<td>1</td>
</tr>
<tr>
<td>Send Time</td>
<td>5</td>
<td>SendTime.timeType</td>
<td>2 (boolean)</td>
<td>W</td>
<td>0</td>
<td>.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>send2display</td>
<td>2 (boolean)</td>
<td>W</td>
<td>.81</td>
<td>1</td>
</tr>
<tr>
<td>Disable Alarm</td>
<td>2</td>
<td>AlarmEnabled</td>
<td>2 (boolean)</td>
<td>W</td>
<td>0</td>
<td>.5</td>
</tr>
</tbody>
</table>

Table 5. Data Table for $M_{KT}$

The final design model specification in pDEVS is provided in Appendix A of this dissertation. The next step is to analyze this system level design model specification from an implementation perspective in a model compilation.
10.2.1. The Ideal Processing Platform

The ideal processing platform for our alarm clock application is defined by the structure of our composed design model. It is depicted in Figure 44. Here, the required processor cycle can be determined directly from the MCTs of the respective model component which have been computed as $MCT_{KT} \approx 0.332$ ms $MCT_{AC} \approx 0.574$ ms for the final design model description, i.e., clock rates of $Clock_{KT} \approx 3.009$ kHz and $Clock_{AC} \approx 1.743$ kHz. In this design model though all communication components have been derived with zero delay parameters.

![Figure 44. Dual Processing Platform for Alarm Clock Design](Image)

From the information about the size of the data transfers of the respective communication components (which can be obtained from a communication component analysis) and corresponding transfer capabilities of given communication elements we could compute the required increase in the processor clock rates to make such a processing platform realistic. But as we have pointed out before we consider a complete custom processor based implementation as impractical and focus therefore on the derivation of an implementation using our processing elements, i.e., a single general purpose microcontroller.
10.2.2. Model Compilation Analysis and Design Model Adaptation

The derivation of an effective model component cycle time for each model component in our first step towards a model compilation. Previously the system specification restricted our processing platform to an Infineon Technologies C1610 microcontroller (a specification of processor characteristics is given in Section 7.5.1).

<table>
<thead>
<tr>
<th>Model Component</th>
<th>Processing State Name</th>
<th>Instruction Count</th>
<th>Estimated Execution Time</th>
<th>Adjusted Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Initialization</td>
<td>6</td>
<td>1.99488 ms</td>
<td>0.03750 ms</td>
</tr>
<tr>
<td>$M_{KT}$</td>
<td>Keep Time</td>
<td>13</td>
<td>4.32224 ms</td>
<td>0.08125 ms</td>
</tr>
<tr>
<td></td>
<td>Update Current Time</td>
<td>3</td>
<td>0.99744 ms</td>
<td>0.01875 ms</td>
</tr>
<tr>
<td></td>
<td>Update Alarm Time</td>
<td>4</td>
<td>1.32992 ms</td>
<td>0.02500 ms</td>
</tr>
<tr>
<td></td>
<td>Send Time</td>
<td>5</td>
<td>1.66240 ms</td>
<td>0.03125 ms</td>
</tr>
<tr>
<td></td>
<td>Disable Alarm</td>
<td>2</td>
<td>0.66496 ms</td>
<td>0.01250 ms</td>
</tr>
<tr>
<td></td>
<td>Initialization</td>
<td>3</td>
<td>1.72086 ms</td>
<td>0.01875 ms</td>
</tr>
<tr>
<td>$M_{AC}$</td>
<td>Get Current Time</td>
<td>2</td>
<td>1.14724 ms</td>
<td>0.01250 ms</td>
</tr>
<tr>
<td></td>
<td>Get Alarm Time</td>
<td>2</td>
<td>1.14724 ms</td>
<td>0.01250 ms</td>
</tr>
<tr>
<td></td>
<td>Set Time</td>
<td>5</td>
<td>2.86810 ms</td>
<td>0.01875 ms</td>
</tr>
<tr>
<td></td>
<td>Set Min</td>
<td>7</td>
<td>4.01534 ms</td>
<td>0.04375 ms</td>
</tr>
<tr>
<td></td>
<td>Set Hours</td>
<td>7</td>
<td>4.01534 ms</td>
<td>0.04375 ms</td>
</tr>
<tr>
<td></td>
<td>Save Time</td>
<td>7</td>
<td>4.01534 ms</td>
<td>0.04375 ms</td>
</tr>
<tr>
<td>$M_{AC2KTR}$</td>
<td>Delay Request</td>
<td>-</td>
<td>0.0 ms</td>
<td>1.13474 ms</td>
</tr>
<tr>
<td>$M_{TK2AC}$</td>
<td>Delay Time</td>
<td>-</td>
<td>0.0 ms</td>
<td>4.24099 ms</td>
</tr>
<tr>
<td>$M_{AC2KTU}$</td>
<td>Delay Update</td>
<td>-</td>
<td>0.0 ms</td>
<td>3.97159 ms</td>
</tr>
</tbody>
</table>

Table 6. Adjusted Model Component Execution Times

A static model component data analysis reveals that no data item in either component of the design specification exceeds the size of a character. Therefore, the ratio of the maximum data item size and the processor data path width is set to one. Estimating a $CPI_{C1610}$ of 100 cycles per model instruction a bit pessimistic (since we intend to cross-
compile our computational segments from a high level language) we compute an effective cycle time for both components on this processor, i.e., $MCT_{eff} = 6.25 \mu s$. Alternatively, we can derive a more accurate CPI from a cycle accurate simulation in our a C161O development environment.

Based on $MCT_{eff}$ we adjust the execution time of all computational segments in both model components as well as delay time parameters of their associated communication components. Table 6 lists the respective instruction counts in the final design specification as well as their adjusted values using their respective processing states.

10.3. Model Compilation of the Alarm Clock

Finally we are ready to proceed to a model compilation of the alarm clock design model specification. Here, the execution of the model compilation algorithm will in this case require a single iteration since we have just a single processing element available. The only mapping possible is to assign both design model components are mapped to the selected processor - an all software solution.

Figure 45. Alarm Clock Execution Trace with Scheduling Conflict in $(t_{PKT}, t_{WMT})$
In the first assessment of the design model, a dedicated processor solution obviously succeeds based on the results which were obtained in Section 10.2.2, i.e., $M_{CT}^{eff} \ll M_{CT}^{AC}$ and $M_{CT}^{eff} \ll M_{CT}^{KT}$. In the following step we assess the multi-component assignments in this mapping for which our example qualifies which in our example is our mapping of $M_{KT}$ and $M_{AC}$ to C161 microcontroller.

Next we apply our previous test scenarios. Figure 45 shows the model execution trace of one such simulation run where $tick$ arrives shortly after a time event, i.e., $t_{\text{tick}} = t_{\text{time}} + \tau_{\text{a}}(\text{PGCT}) + \tau_{\text{a}}(\text{PSC})$. Here, $M_{AC}$ and $M_{KT}$ execute concurrently and cause a scheduling conflict. Based on the data presented in Table 6 the computation of the scheduling window for $M_{TK}$ is straightforward since $P_{KT}$ does not generate any external outputs for $M_{AC}$, i.e., $t_{\text{SchW}}(M_{KT}) = 0 + \tau_{\text{a}}(P_{KT}) + \eta (W_{KT}, tick) \approx 60$ seconds where $t^* = t_{\text{tick}}$. Nevertheless, the computation of the flexibility window poses a $M_{AC}$ problem since button events are aperiodic. Here, we have to introduce a reasonable lower bound on its arrival which we assume to be 0.01 seconds after to the last assertion of a button. Using this assumption we can compute $\eta (W_{MT}, \text{button}) = 0.01 - (\tau_{\text{a}}(\text{PGCT}) + \tau_{\text{a}}(\text{PSC}) + \tau_{\text{a}}(\text{PS})) \approx 0.1$ and $t_{\text{SchW}}(M_{AC}) = 0 + \tau_{\text{a}}(P_{ST}) + \eta (W_{MT}, \text{button}) \approx 0.01$. Clearly we can perform a rescheduling of components since our first condition for a rescheduling holds (see Chapter 7) since the sequential reordering is possible. Notice that the second condition does not need to apply since $MC = \{\emptyset\}$.

Other critical test scenarios which cause a concurrent execution of design model components are then applied and verified in the same manner. These scenarios are $t_{\text{tick}} = t_{\text{alarm}}$ in the normal and set-alarm operation modes, $t_{\text{tick}} = t_{\text{hrs}}$, $t_{\text{tick}} = t_{\text{min}}$, and during the
initialization of the device. Notice that any assessment concerning the arrivals of $t_{\text{tick}}$ with $t_{\text{alarm}}$ are not required since the execution times in $M_{\text{KT}}$ and $M_{\text{AC}}$ are the same as in the $t_{\text{tick}}$ with $t_{\text{time}}$ scenarios, e.g., for $t_{\text{tick}} = t_{\text{alarm}}$.

In this example we determine this mapping to be valid under the condition of a proper rescheduling. For this purpose we may use the previously presented rescheduling algorithm. Relevant cases, i.e., $t_{\text{tick}} = t_{\text{button}}$, are then encoded in our RTOS handling of external inputs. Notice that then a simple scheduling of RTOS processes based on the arrival of inputs will automatically resolve any scheduling conflicts in the desired order.

A candidate operating system for such an implementation is SMOS which was presented in [38].

The last phase of the algorithm, i.e., the processor communication analysis is superfluous in this particular case since we are mapping our design model specification to a single processor. In addition, communication delays between processes $M_{\text{KT}}$ and $M_{\text{AC}}$ (from the mapping of communication component) may be eliminated because of this reason.

10.4. Implementation

The final implementation of design model components is straight forward. We extract the computational segments of each model component specification and organize their execution in the order described by the internal model transitions. In addition, we place receive primitives, which implement the waiting and retrieval of required inputs, in the beginning of each segment. Send primitives are placed at the end of segments which produce outputs for the LED, alarm, or the other design model component
implementation. We show in Figure 46 an example of such a software implementation of
alarm clock component $M_{AC}$.

```plaintext
Process $M_{AC}$() {
  Byte Digit1 = 0, Digit0 = 0, Mode = WAITING, button, value;
  while (true) {
    receive( &button, &value );
    while ( button != TIME & button != ALARM ) {
      switch( button ) {
        case TIME: send( ProcessMcT, GETCURRENTTIME, 0 );
                    Mode = SETTIME; break;
        case ALARM: send( ProcessMcT, GETALARMTIME, 0 );
                    Mode = SETALARM; break;
        case default: receive( &button, &value);
      } //end switch
    } //end while
    receive( &button, &Digit1 );
    receive( &button, &Digit0 ); // button can be ignored here!
    send( LED, DIGIT1, Digit1 );
    send( LED, DIGIT0, Digit0 );
    receive( &button, &value );
    while ( ! (button == TIME && Mode == SETTIME) ||
            (button == ALARM && Mode == SETALARM) ) {
      switch( button ) {
        case MIN: Digit0++;
                  if ( Digit0 > 59 ) Digit0 = 0;
                  send( LED, DIGIT0, Digit0 ); break;
        case HRS: Digit1++;
                  if ( Digit1 > 23 ) Digit1 = 0;
                  send( LED, DIGIT1, Digit1 );
      } //end switch
      receive( &button, &value );
    } //end while
    if ( mode == SETTIME ) {
      send( ProcessMcT, TIMEHRS, Digit1 );
      send( ProcessMcT, TIMEMIN, Digit0 );
    } else {
      send( ProcessMcT, ALARMHRS, Digit1 );
      send( ProcessMcT, ALARMMIN, Digit0 );
    } //end if else
    send( LED, DIGIT1, Digit1 );
    send( LED, DIGIT0, Digit0 );
  } // end while
} //end process
```

Figure 46. Implementation of Final Design Model Specification of $M_{AC}$
11. CONCLUSIONS

In this chapter we emphasize the accomplishments of this dissertation and suggest some directions for future research on the presented material. We first summarize the major contributions of the work presented in the preceding chapters, the benefits of embedded systems design from a systems modeling perspective, and discuss briefly another important aspect of our model-based codesign methodology - the derivation of proper testing scenarios for design models.

11.1. Summary of Results

We have presented the theoretical foundations for a general, system level model-based design approach for development and implementation of real-time embedded system applications. We outlined a model-based codesign methodology which leverages heavily off systems modeling concepts to derive a concurrent, implementation independent design models using modular discrete event modeling specifications.

One contribution of this dissertation has been a structured approach to the formal construction and refinement of discrete event design models for real-time embedded systems applications from an informal system specification. We introduced the development of such applications by constructing a system model specification using a new model of abstraction for the computation performed by embedded systems. We introduced the concept of a model component cycle time as means of specifying execution times for computations.

In addition, a generic communication model component specification was introduced to model the interaction of application components. These components enforce a proper
synchronization of computation for any proposed application design at the system level and allow an early assessment and evaluation of communication within a composed design model specification.

A major benefit of our approach to embedded system design is the ability to easily integrate also models physical systems in the testing of application design model, i.e., the environment that the application is to be embedded in, due to our foundation in systems modeling. Here, we assess the design model of the application using experimental frames which encode operate different experimental conditions or operating scenarios. It was also shown that even aperiodic inputs from the environment, such as human user interaction, can be specified in such experimental frames and included in the assessment of the application design.

We defined in detail an iterative refinement process for such design model specifications from a practical as well as a theoretical perspective. Refinement was discussed in the context of introducing structural and behavioral requirements stated in the application system specification as well as the increase in resolution of model component specifications. A number of design model analysis methods were presented to guide this refinement process, to validate its correctness, and to assess the performance of the design model specification at any level of abstraction during design model development.

Our proposed design approach has explicitly addressed the model continuity problem, which is the inability to transform system level model specifications into efficient design implementations. We showed the derivation and validation of appropriate parallel
processing platforms for any design model specification specified using our proposed approach. The architectural elements of such a platform were defined as well as an ideal processing platform which maximizes the utilization of its processing elements.

A model compilation algorithm was presented which derives such a processing platform for a set of specific application implementation constraints. Different phases of the algorithm attempt to assign model components either to hardware, dedicated processing or shared processing elements (i.e., processors which have an minimum real-time operating system installed). For the latter assignment a proper scheduling order of processes can be derived which then allows a real-time execution of software implementations for multiple design components on a single processor.

We discussed also the implementation of design model component behavior for the previously derived processing platform in the form of hardware and software components. This allows the creation of a first implementation prototype for the given application. A formal mapping of design model component specifications to an implementation level design specification was shown based on DEVS and timed CSP. We outlined a transformation into hardware and software component based on this mapping.

11.2. Systems Modeling for the Codesign of Embedded Systems

This dissertation has attempted to expose formal systems modeling concepts and its benefits to a greater audience. Our most important contribution is here our specific but yet application independent approach to the construction of system model specifications for embedded systems applications. So far systems modeling research has been
advocating a far more general construction of model specifications. We feel that a model-based approach to codesign is not only suited but also beneficial in the context of embedded computing systems design.

In this dissertation, a model-based design approach has shown to contribute numerous benefits. Most important to mention here is the ability to specify the desired behavior of embedded systems applications at various levels of abstraction, assess an design at any point during application development, as well as the derivation of an implementation independent design specification. A number of additional benefits arise from the theoretical foundation of our work in systems modeling, i.e., the ability to include models of the environment based on a well established body of research on physical system specifications [19], verify or ensure a correct refinement of an application design in a structured manner, and most importantly the universal applicability of this approach to any given embedded computing systems which can be specified as a discrete event model.

We want to emphasize also the feasibility of using a model-based design approach in practice. In an implementation of this design approach, a user or application developer can be effectively hidden from the underlying modeling specification or its theoretical foundations. The construction of design models only requires a fairly basic knowledge of modeling concepts. Many of the analysis and refinement methods introduced in this dissertation can also be automated.
11.3. Design and Test

This dissertation has only focused on the design aspects in the development of embedded system applications. Applications have been designed in a stepwise refinement process where its design specification is assessed validated after each refinement using simulation. A functional validation or the verification of specified performance constraints has been achieved in this setting using a special model specification - experimental frames. This system level approach was in a later chapter further extended to multi-level testing where test scenarios were derived for a real-time system testing environment which then assessed a generated mixed hardware/software application prototype at the implementation level.

Obviously, such an assessment at either level also requires a structured approach to derive testing scenarios which determines worst case or relevant testing scenarios for the application design. Cunning has provided in [66] the theoretical foundation for this testing aspect in the context of our model-based codesign methodology. Here, the automatic generation of testing scenarios is presented for event-oriented real-time embedded systems. It is shown how informal application requirements stated in a system specification can be converted efficiently into minimal sequences of event stimuli, and how the response of a design model specification can be verified against these requirements.

11.4. Future Work

Certainly more detailed work is required on the conversion of model component behavior into hardware component descriptions. Our description only provides a general outline
for such a conversion and omitted details such as a precise definition of an instruction set or a RTL level control unit design for custom processors. However, it has been pointed out that a general approach for an efficient behavioral hardware synthesis remains to be an open research topic today.

The model specification of communication components could be further extended to include or gather more information in the transmission of data, e.g., some means to allow the generation of an appropriate protocol. At this point of our research we merely generate lower communication bounds for a selected implementation during a model compilation. Care has to be taken to preserve implementation independence in such step. Similarly, appropriate communication model analysis methods could be developed to analyze the behavior of these components more thoroughly in the model state execution trace.

Larger case studies are ultimately required to assess the practicality of this extending approach. At least the author is confident that this design approach could be also applied to a much wider range of applications, i.e., heterogeneous computing systems. Here, it would be of interest to evaluate the increase in the size of the design model specification it is refined to a required level of resolution. Note that in this case we would have to provide a more general definition of what exactly constitutes an appropriate final resolution. For larger applications we also have to further refine our definition of architectural to include, e.g., different level of processor cache or virtual memory.

An emerging trend in embedded computing systems is the design for adaptable application implementations. From a conceptual standpoint model-based design offers a
great potential for the design of such systems since it allows an implementation independent design specification. Here, it would be of interest to introduce more flexibility to the concept of a model component cycle time in order to simulate changes in the implementation of system components on the fly. Additional design model analysis methods need to be developed to determine appropriate implementation or location changes of design components during the operation of the application. Similarly, the delay time parameter in communication components could be made more flexible to reflect a change of the relative location of a design component in respect to others.

Finally this work may be useful in system modeling to derive a new approach to the real-time simulation of discrete event model specifications. Here, further research could address a structured approach for the conversion of any system model specification to a model specification based our abstraction of computation, i.e., waiting and processing states. This revised system model specification would then be treated and even further refined as a design model specification of an embedded systems application. A model compilation could then convert or assess this design model specification for the implementation for a specified processing platform.
APPENDIX A: PDEVS IMPLEMENTATION OF THE ALARM CLOCK MODEL
import com.threadtec.silk.util.*; // import link list routines

// pDEVS Mode Control Model realizes M_A
public class ModeControl extends Atomic {
    protected final double ClockRate = 16E6; // Infineon C1610 Clock Speed
    protected final double MCT = 157.0/273700.0; //1/ClockRate;
    protected Message m;
    protected String operationMode;
    protected byte Digit1, Digit0;

    public void initAtomic(){
        disableStatistics();
        disableVerbose();
        addInport("buttons");
        addInport("clockin");
        addOutport("clockout");
        addOutport("LEDdisplay");

        // communication component notification ports
        addOutport("buttonack");
        addOutport("clockack");

        // waiting states
        addState("wait4button");
        addState("wait4time");
        addState("wait4set");

        // invalid button states
        addState("invalidButton");
        addState("invalidSet");
        addState("invalidTime");

        // transitional waiting states and processing states
        addState("triggerGetCurrentTime");
        addState("processGetCurrentTime");
        addState("triggerGetAlarmTime");
        addState("processGetAlarmTime");
        addState("triggerSetTime");
        addState("processSetTime");
        addState("triggerSaveTime");
        addState("processSaveTime");
        addState("triggerSetMin");
        addState("processSetMin");
        addState("triggerSetHr");
        addState("processSetHr");

        Digit1 = 0; Digit0 = 0; // initialize LED digit values
        operationMode = new String("normal");
initialState("processinit", 3 * MCT);
} // end initAtomic

public void extTransFunction() {

    String Button, Identifier;

    if ( currentState( "wait4button" ) ) {
        while( MsgsAt("buttons") ) {
            m = getMsg("buttons");
            Button = m.getStringValue();
            if ( Button.compareTo("time") == 0 ) {
                hold_in("triggerGetCurrentTime", 0.0);
            } else if ( Button.compareTo("alarm") == 0 ) {
                hold_in("triggerGetAlarmTime", 0.0);
            } // end if else
        } // end while
    } else if ( currentState( "wait4time" ) ) {
        while( MsgsAt("clockin") ) {
            m = getMsg("clockin");
            Identifier = m.getIdentifier();
            if ( Identifier.compareTo("hours") == 0 ) {
                Digit1 = m.getByteValue();
            } else if ( Identifier.compareTo("min") == 0 ) {
                Digit0 = m.getByteValue();
            } // end else
        } // end while
        hold_in("triggerSetTime", 0.0);
    } else if ( currentState( "wait4set" ) ) {
        while( MsgsAt("buttons") ) {
            m = getMsg("buttons");
            Button = m.getStringValue();
            if ( Button.compareTo(operationMode) == 0 ) {
                hold_in("triggerSaveTime", 0.0);
            } else if ( Button.compareTo("min") == 0 ) {
                hold_in("triggerSetMin", 0.0);
            } else if ( Button.compareTo("hr") == 0 ) {
                hold_in("triggerSetHr", 0.0);
            } // end else if Button
        } // end while
    } // end else if
} // end extTransFunction

public void outFunction() {

    if ( currentState( "processSetTime" ) ||
        currentState( "processSetMin" ) ||
        currentState( "processSaveTime" ) ||
        currentState( "processSetHr" ) ) {
        if ( operationMode.compareTo("alarm") == 0 ||
            currentState( "processSaveTime" ) )
            addMsg( "LEDdisplay", new Message("Mode", (byte)0) ); // normal
        else
            addMsg( "LEDdisplay", new Message("Mode", (byte)1) ); // blink
addMsg( "LEDdisplay", new Message( "Digit1Value", Digit1 ) );
addMsg( "LEDdisplay", new Message( "Digit0Value", Digit0 ) );
// and notification signal
addMsg( "buttonack", new Message( "ready" ) );
}
else if ( currentState( "processGetCurrentTime" ) ) {
addMsg( "clockout", new Message( "get", "time" ) );
avMsg( "clockack", new Message( "ready" ) );
}
else if ( currentState( "processGetAlarmTime" ) ) {
addMsg( "clockout", new Message( "get", "alarm" ) );
addMsg( "clockack", new Message( "ready" ) );
}
// send out to clock module if leaving the set time operating mode
if ( currentState( "processSaveTime" ) ) {
if ( operationMode.compareTo("time") == 0 ) {
addMsg( "clockout", new Message( "timehours", Digit1 ) );
addMsg( "clockout", new Message( "timemin", Digit0 ) );
} else {
addMsg( "clockout", new Message( "alarmhours", Digit1 ) );
addMsg( "clockout", new Message( "alarmin", Digit0 ) );
}
// more notification signals
if ( currentState( "invalidButton" ) ||
currentState( "invalidSet" ) ||
currentState( "processinit" ) ) {
addMsg( "buttonack", new Message( "ready" ) );
}
// end if
}
// end outFunction

public void intTransFunction() {

// first check transitions to waiting states
if ( currentState("processinit") ||
currentState("processKeepTime") ||
currentState("processSaveTime") ||
currentState("invalidButton") ) {
hold_in("wait4button", INFINITY);
}
else if ( currentState("processSetTime") ||
currentState("processSetMin") ||
currentState("processSetHr") ||
currentState("invalidSet") ) {
hold_in("wait4set", INFINITY);
}
else if ( currentState("processGetCurrentTime") ||
currentState("processGetAlarmTime") ) {
hold_in("wait4time", INFINITY);
}
// end else if

// then transitions to processing states
if ( currentState("triggerGetCurrentTime") ) {
// change operating mode and get time from clock module
operationMode = "time"; // = set time
// instruction count 1 + 1 output
hold_in("processGetCurrentTime", 2 * MCT );

} else if ( currentState("triggerGetAlarmTime") ) {
// change operating mode and get time from clock module
operationMode = "alarm"; // = set alarm
// instruction count 1 + 1 output
hold_in("processGetAlarmTime", 2 * MCT );

} else if ( currentState("triggerSetTime") ) {
// instruction count 0 + + 2 inputs + 3 Outputs
hold_in("processSetTime", 5 * MCT );

} else if ( currentState("triggerSetMin") ) {
Digit0++;
if ( Digit0 > 59 ) Digit0 = 0;
// instruction count 3 + 1 output decision + 3 outputs
hold_in("processSetMin", 7 * MCT );

} else if ( currentState("triggerSetHr") ) {
Digit1++;
if ( Digit1 > 23 ) Digit1 = 0;
// instruction count 3 + 1 output decision + 3 outputs
hold_in("processSetHr", 7 * MCT );

} else if ( currentState("triggerSaveTime") ) {
// change operating mode and send new time to clock module
operationMode = "idle"; // = none
// instruction count 1 + 1 output decision + 5 outputs
hold_in("processSaveTime", 7 * MCT );

} // end else if

} // end intTransFunction

} // end ModeControl class
pDEVS Model For the Time Keeping Component (M_{KT})

```java
import com.threadtec.silk.ucil; // import link list routines

// pDEVS Clock Model realizes M_{KT}
public class Clock extends Atomic {

protected final double ClockRate = 16E6; // Infineon C1610 Clock Speed
protected final double MCT = 13.0/39100.0; // 1/ClockRate;

protected Message m;
protected byte timehours, timemin;
protected byte alarmhours, alarmin;
protected String timeType;
protected boolean Send2Display, SendAlarm, AlarmEnabled;

public void initAtomic() {
    disableStatistics();
    disableVerbose();

    addInport("tick");
    addInport("modein");
    addOutport("modeout");
    addOutport("LEDdisplay");
    addOutport("alarm");

    // communication component notification ports
    addOutport("modeack");
    addOutport("tickack");

    // waiting states
    addState("wait4any");

    // transitional waiting states and processing states
    addState("triggerKeepTime");
    addState("processKeepTime");

    addState("triggerUpdateCurrentTime");
    addState("processUpdateCurrentTime");

    addState("triggerUpdateAlarmTime");
    addState("processUpdateAlarmTime");

    addState("triggerDisableAlarm");
    addState("processDisableAlarm");

    addState("triggerSendTime");
    addState("processSendTime");

    timehours = 23; timemin = 57;
    alarmin = 0; alarmin = 0;
    Send2Display = true; // after reset send to display
    AlarmEnabled = false;
    SendAlarm = false;  // not true instruction .. only required in DEVS
    initialState("processinit", 6 * MCT);
}
```

} // end initAtomic
public void extTransFunction() {

    String Identifier;

    if (currentState( "wait4any" ) ) {
        while( MsgsAt("tick")) {
            m = getMsg("tick");
            hold_in("triggerKeepTime", 0.0 );
        } // end if

        while( MsgsAt("modein")) {
            m = getMsg("modein");
            Identifier = m.getIdentifier();
            if ( Identifier.compareTo("get") == 0 ) {
                timeType = m.getStringValue();
                hold_in("triggerSendTime", 0.0 );
            } else if ( Identifier.compareTo("timehours") == 0 ) {
                timehours = m.getByteValue();
            } else if ( Identifier.compareTo("alarminhours") == 0 ) {
                alarmhours = m.getByteValue();
            } else if ( Identifier.compareTo("timemin") == 0 ) {
                timemin = m.getByteValue();
            } else if ( Identifier.compareTo("alarmmin") == 0 ) {
                alarmmin = m.getByteValue();
            } // end else if
            if ( Identifier.startsWith("time") ) {
                hold_in("triggerUpdateCurrentTime", 0.0);
            } else if ( Identifier.startsWith("alarm") ) {
                hold_in("triggerUpdateAlarmTime", 0.0);
            } // end if
        } // end while
    } // end if

} // end extTransFunction

public void outFunction() {

    if (currentState( "processKeepTime" ) && Send2Display ) {
        addMsg("LEDdisplay", new Message( "Mode", 1 ) );
        addMsg("LEDdisplay", new Message( "Digit1Value", timehours ) );
        addMsg("LEDdisplay", new Message( "Digit0Value", timemin ) );
    } else if (currentState( "processSendTime" ) ) {
        if ( timeType.compareTo("time") == 0 ) {
            addMsg("modeout", new Message( "hours", timehours ) );
            addMsg("modeout", new Message( "min", timemin ) );
        } else {
            addMsg("modeout", new Message( "hours", alarmhours ) );
            addMsg("modeout", new Message( "min", alarmmin ) );
        } // end else
    } // end else if

    if (currentState( "processKeepTime" ) && SendAlarm ) {
        SendAlarm = false;
        addMsg( "alarm", new Message( "alarm", "on" ) );
    }
}
// end if

// communication component notification signals
if ( (getModelStateName()).startsWith("process") ) {
    addMsg("modeack", new Message("ready");
    addMsg("tickack", new Message("ready");
} else if (currentState("triggerSendTime") ||
    currentState("triggerUpdateCurrentTime") ||
    currentState("triggerUpdateAlarmTime") ){
    addMsg("tickack", new Message("busy");
} else if (currentState("triggerKeepTime") ) {
    addMsg("modeack", new Message("busy");
}

// end else if

} // end outFunction

public void intTransFunction( ) {

    // first check transitions to waiting states
    if ( (getModelStateName()).startsWith("process") ) {
        hold_in("wait4any", INFINITY);
    } // end else if

    // then transitions to processing states
    if ( currentState("triggerKeepTime") ) {
        timemin--;
        if ( timemin > 59 ) {
            timehours++;
            timemin = 0;
        } // end if
        if ( timehours > 23 ) {
            timehours = 0;
        } // end if
        if ( AlarmEnabled ) {
            if ( timehours == alarmhours & timemin == alarmmin ) {
                SendAlarm = true;
            } // end if
        } // end if

        // instruction count 9 + 1 output decision + 4 possible outputs
        hold_in("processKeepTime", 13 • MCT );
    } else if ( currentState("triggerSendTime") ) {
        // returns time to mode control and produces outputs either time or alarm
        Send2Display = false;
        // instruction count 1 + 1 input + 1 output decision + 2 outputs
        hold_in("processSendTime", 5 • MCT );
    } else if ( currentState("triggerUpdateCurrentTime") ) {
        // changes display mode back and updates the current time
        Send2Display = true;
        // instruction count 1 + 2 received inputs and
        hold_in("processUpdateCurrentTime", 3 • MCT );
    } else if ( currentState("triggerUpdateAlarmTime") ) {
        // changes display mode back and updates the current time

Send2Display = true;
AlarmEnabled = true;
// instruction count 2 + 2 received inputs and
hold_in("processUpdateAlarmTime", 4 * MCT );

} // end else if

} // end intTransFunction

} // end Clock class
pDEVS Coupled Model “Alarm Clock Design Model”

// the composed design model for the Alarm Clock
public class AlarmClockDesign extends Coupled {

    public void initCoupled() {

        // model components have to be first created in Simulation.java !!
        enableExecutionTrace();

        addImport("tick");
        addImport("buttons");
        addOutport("tickAckOut");
        addOutport("buttonAckOut");
        addOutport("LED Display");
        addOutport("alarm");

        // Atomic Models
        addAtomic("ClockModule");
        addAtomic("ModeModule");

        // Communication Components
        addAtomic("Clock2Mode");
        addAtomic("Mode2Clock");

        // component couplings
        // - M<sub>x</sub> module
        addCoupling("this", "tick", "ClockModule", "tick");
        addCoupling("ClockModule", "tickack", "this", "tickAckOut");
        addCoupling("ClockModule", "LEDdisplay", "this", "LED Display");
        addCoupling("ClockModule", "alarm", "this", "alarm");

        // - communication component couplings
        addCoupling("Mode2Clock", "out", "ClockModule", "modein");
        addCoupling("ClockModule", "modeack", "Mode2Clock", "ack");
        addCoupling("ClockModule", "modeout", "Clock2Mode", "in");

        // - M<sub>c</sub> module
        addCoupling("this", "buttons", "ModeModule", "buttons");
        addCoupling("ModeModule", "buttonack", "this", "buttonAckOut");
        addCoupling("ModeModule", "LEDdisplay", "this", "LED Display");

        // - communication component couplings
        addCoupling("Clock2Mode", "out", "ModeModule", "clockin");
        addCoupling("ModeModule", "clockack", "Clock2Mode", "ack");
        addCoupling("ModeModule", "clockout", "Mode2Clock", "in");

    } // end initCoupled

} // end AlarmClockDesign class
pDEVS Master Coupled Model “Alarm Clock”

// the alarm clock master coupled model
public class MyAlarmClock extends Coupled {

    public void initCoupled() {

        // model components have to be first created in Simulation.java !!
        addInport("tick");
        addInport("buttons");
        addOutport("out");

        // Coupled Models
        addCoupled("EF");
        addCoupled("Design");

        // couplings
        addCoupling("this", "tick", "EF", "tick");
        addCoupling("this", "buttons", "EF", "buttons");
        addCoupling("EF", "tickOut", "Design", "tick");
        addCoupling("Design", "tickAckOut", "EF", "tickAck");
        addCoupling("EF", "buttonOut", "Design", "buttons");
        addCoupling("Design", "buttonAckOut", "EF", "buttonAck");
        addCoupling("Design", "LED Display", "EF", "in");
        addCoupling("Design", "alarm", "EF", "in");

    } // end initCoupled

} // end MyAlarmClock class
pDEVS Coupled Model Trace Output recorded for composed “Design” model (Test Scenario shown in Figure 41):

Coupled Model State Execution Trace for the Design coupled model
Simulation Time and coupled model state trace (consisting of ClockModule, ModeModule models)

0.0  processinit  processinit
0.001720862257946657  processinit  wait4button
0.0019948849104859338  wait4any  wait4button
1.0  triggerKeepTime  wait4button
1.0  processKeepTime  wait4button
1.0043222506393863  wait4any  wait4button
1.1  wait4any  triggerGetAlarmTime
1.1  wait4any  processGetAlarmTime
1.1011472415052979  wait4any  wait4time
1.1011472415052979  triggerSendTime  wait4time
1.1011472415052979  processSendTime  wait4time
1.1028096455973695  wait4any  wait4time
1.1028096455973695  triggerSetTime  wait4any
1.1028096455973695  processSetTime  wait4any
1.105677749360614  wait4any  wait4set
1.3  wait4any  triggerSetMin
1.3  wait4any  processSetMin
1.3040153452685423  wait4any  wait4set
1.8  wait4any  triggerSaveTime
1.8  wait4any  processSaveTime
1.8040153452685423  wait4any  wait4button
1.8040153452685423  triggerUpdateAlarmTime  wait4button
1.8040153452685423  processUpdateAlarmTime  wait4button
1.8053452685421996  wait4any  wait4button
2.0  triggerKeepTime  wait4button
2.0  processKeepTime  wait4button
2.0043222506393863  wait4any  wait4button
3.0  triggerKeepTime  wait4button
3.0  processKeepTime  wait4button
3.0043222506393863  wait4any  wait4button
Output of “Collector” model in experimental frame in that simulation run:

Collector Output from simulation

At simulation time 1.0:
  Clock tick received!

At simulation time 1.0043222506393863:
  Display Mode is NORMAL   Current Time: 23:58

At simulation time 1.1:
  alarm button was pressed!

At simulation time 1.105677749360614:
  Display Mode is NORMAL   Current Time: 0:0

At simulation time 1.3:
  min button was pressed!

At simulation time 1.3040153452685423:
  Display Mode is NORMAL   Current Time: 0:1

At simulation time 1.8:
  alarm button was pressed!

At simulation time 1.8040153452685423:
  Display Mode is NORMAL   Current Time: 0:1

At simulation time 2.0:
  Clock tick received!

At simulation time 2.0043222506393863:
  Display Mode is NORMAL   Current Time: 23:59

At simulation time 3.0:
  Clock tick received!

At simulation time 3.0043222506393863:
  Display Mode is NORMAL   Current Time: 0:0
APPENDIX B: PDEVS IMPLEMENTATION OF GENERIC COMMUNICATION COMPONENT
import java.util.*; // import dynamic arrays

// the MBC communication component
public class Communication extends Atomic {

  protected Vector EventList = new Vector();
  protected Message m;
  protected double DelayTime; // delay time parameter
  protected double AdjustedDelayTime; // ... by already elapsed waiting time
  protected double WaitInBuffered;

  public void initAtomic() {
    disableStatistics();
    disableVerbose();

    addInport("in"); // data input port
    addInport("ack"); // notification input port
    addOutport("out"); // data output port

    // waiting states
    initialState("wait");
    addState("waitBuffered");
    addState("waitUnbuffered");

    // processing states
    addState("processDelay");

    DelayTime = 0.0; // initialize always to 0.0
    // a modification of the Delay Time possible using
    // setModelParameters(double) method after its instantiation in a Coupled
    // Model Specification using addAtomic(String)
    WaitInBuffered = 0.0;
  } // end initAtomic

  public void extTransFunction() {

    String NotificationSignal = new String();
    if (MsgsAt("in") ) {
      if (EventList.isEmpty() ) {
        while (MsgsAt("in") ) {
          m = getMsg("in");
          // insert to list
          EventList.addElement(m);
        } // end while
      } else {
        // otherwise give a warning and ignore incoming messages
        while (MsgsAt("in") ) m = getMsg("in");
      } // end if else
    } // in the case that if data arrives without any notification signal
    if (MsgsAt("ack") ) {
      if (currentState("wait") || currentState("waitBuffered") ) {
        if (currentState("waitBuffered") ) {
          WaitInBuffered += getElapsedTime();
        } else if (currentState("processDelay") ) {
          AdjustedDelayTime -= getElapsedTime();
        } // end if
      } // end if
    } // end if
  } // end extTransFunction
} // end class Communication
hold_in( "processDelay", AdjustedDelayTime );
} else { // if Unbuffered waiting State
    AdjustedDelayTime = DelayTime;
    hold_in( "processDelay", AdjustedDelayTime );
} // end else if

} else { // if ALSO notification signal arrives at the same time
    // get message from port
    m = getMsg("ack");
    NotificationSignal = m.getStringValue();
    // first case: notification signal is a kill signal
    if ( NotificationSignal.compareTo("busy") == 0 ) {
        // then keep current state and update waiting times
        if ( currentState( "waitUnbuffered" ) ) {
            hold_in("wait", INFINITY);
        } else if ( currentState( "processDelay" ) ) {
            // count previous processing time as Buffered Time
            WaitInBuffered = DelayTime - AdjustedDelayTime + getElapsedTime();
            hold_in("waitBuffered", INFINITY);
        } else {
            WaitInBuffered += getElapsedTime();
        } // end if else
    } else ( // in second case assume "ready" notification signal
        if ( currentState( "waitUnbuffered" ) ) { // then change to process delay state
            AdjustedDelayTime = DelayTime;
            hold_in( "processDelay", AdjustedDelayTime );
        } else if ( currentState( "waitBuffered" ) ) {
            WaitInBuffered += getElapsedTime();
            if ( WaitInBuffered > DelayTime ) {
                hold_in( "processDelay", 0.0 );
            } else {
                AdjustedDelayTime = DelayTime - WaitInBuffered;
                hold_in( "processDelay", AdjustedDelayTime );
            } // end if else
        } // end if else state
    } // end if else specific notification signal
} // end if else also signal at "ack" port
} // end if messages at "in" port

// if component is only receiving a notification signal while ( MsgsAt("ack") ) {
    // remove message from port
    m = getMsg("ack");
    NotificationSignal = m.getStringValue();
    // again as in the previous scenario
    // first case: notification signal is a kill signal
    if ( NotificationSignal.compareTo("busy") == 0 ) {
        // then keep current state and update waiting times
        if ( currentState( "waitUnbuffered" ) ) {
            hold_in("wait", INFINITY);
        } else if ( currentState( "processDelay" ) ) {
            // count previous processing time as Buffered Time
            WaitInBuffered = DelayTime - AdjustedDelayTime + getElapsedTime();
            hold_in("waitBuffered", INFINITY);
        } else {
            WaitInBuffered += getElapsedTime();
        } // end if else
public void outFunction( ) {
    int i, end = EventList.size();

    if ( currentState( "processDelay" ) ) {
        for ( i = 0; i < end; i++ )
            addMsg( "out", (Message)(EventList.elementAt( i )) );
    } // end if
    EventList.removeAllElements();
} // end outFunction

public void intTransFunction( ) {
    // only internal transition is from processDelay state
    hold_in("wait", INFINITY);  
} // end intTransFunction

public void setModelParameters( double Value ) {
    if ( Value < 0.0 ) {
        System.out.println("ERROR in " + getModelName() + " communication model component!!");
    } else {
        DelayTime = Value;
    } // end if else
} // end setModelParameters

} // end Communication class
APPENDIX C: EXTRA CREDITS
Numerous additional people, places, and events inspired the research done for this dissertation, or helped me through the tough times in the pursuit of this degree. Since there is unfortunately a single page limit to the Acknowledgement Section I want to take the opportunity to thank everybody I had to leave out previously in this compilation:

People

Jonah Lavi
Hessam Sarjoughian
Harold Lawson
Tami Whelan
Hugo Simpson
Krzysztof Jagiello
Bruce Shriver
Dan Nicolaescu
David Harel
J. Sean Keane
Ismael Diaz Mozas
Andre Windisch
Jan Reisch
Thomas Schneider
Anke Schulz
Traintafillus Karayiannis
Caren Schulz
Robert Völker
Annedörthe Anker
Robin Boch
Karl Anker
Klaus-Dieter Schubert
Joachim Buth
Joachim Buth
Jakob Axelsson
Petru Elecs
Anette Muth
Edward Lee
Petru Eles
Harry Hsieh
Tucson, AZ, USA
Rolf Ernst
New York, NY, USA
Georg Färber
München, Germany
Franz Fischer
Finse, Norway
Annette Muth
Helsinki, Finland

Conferences

- International Workshop on Hardware/Software Codesign 1998, Seattle, WA, USA
- Engineering of Computer-Based Systems Conference 1998, Jerusalem, Israel
- Winter Simulation Conference 1999, Phoenix, AZ, USA
- Embedded Systems Conference 2000, Nürnberg, Germany
- Engineering of Computer-Based Systems Conference 2000, Edinburgh, Scotland
REFERENCES


