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. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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.
Static scheduling of hard real-time control software using an asynchronous data-driven execution model

Byrnes, Denise Dianne, Ph.D.
The Ohio State University, 1992
Static Scheduling of Hard Real-Time Control Software Using an Asynchronous Data-Driven Execution Model

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of the Ohio State University

by

Denise Dianne Byrnes, B.S., M.S.

* * * * *

The Ohio State University 1992

Dissertation Committee:
Dr. Bruce W. Weide
Dr. P. Sadayappan
Dr. Judith D. Gardner

Approved by
Bruce W. Weide
Advisor
Department of Computer and Information Science
To my parents, Jeanne and Robert Byrnes
ACKNOWLEDGMENTS

I wish to express my sincere thanks to my advisor, Bruce Weide. Your infinite patience is appreciated more than you will ever know. Without your guidance and intellectual clarity I would have floundered in a sea of confusion. I also wish to thank the other members of my reading committee, Judy Gardiner and P. Sadayappan. Your suggestions started this dissertation on the right track.

I am also grateful to current and past members of the Reusable Software Research Group at Ohio State. Thanks go to Lonnie Welch, Bill Ogden, Stephen Edwards, Doug Harms, Mike Stovsky and Joe Hollingsworth. Several of you offered helpful suggestions for this work, but primarily you provided support and valued friendship.

A special thanks goes to Stuart Zweben and Karsten Schwan who encouraged me to apply to graduate school at Ohio State.

Ken Supowit offered a key insight on one of the algorithms in Chapter 3 for which I am also grateful.

Finally, I wish to thank my parents who kept saying “you’ve gone this far, it would be a shame to give up now.”
VITA

April 4, 1951................................................................. Born — Youngstown, Ohio

1983................................................................. Bachelor of Science
The Ohio State University
Columbus, Ohio

1985................................................................. Master of Science
The Ohio State University
Columbus, Ohio

1984 – 1991................................................................. Teaching Assistant
The Ohio State University
Columbus, Ohio

1991 – Present................................................................. Computer Science Faculty
Math Department
The College of Wooster
Wooster, Ohio
Publications


Fields of Study

Major Field: Software Engineering
Minor Fields: Computer Graphics and Parallel Architectures
# TABLE OF CONTENTS

DEDICATION .......................................................................................................................ii  
ACKNOWLEDGMENTS .................................................................................................iii  
VITA .......................................................................................................................................iv  
LIST OF FIGURES .....................................................................................................viii  
LIST OF TABLES ........................................................................................................ix  
CHAPTER I Introduction ..............................................................................................1  
  1.1. Synchronous Clock-Driven Static Scheduling and Analysis .........................3  
    1.1.1. Synchronous Clock-Driven Execution Model ......................................6  
    1.1.2. Scheduling Constraints ...........................................................................7  
    1.1.3. Scheduling Criteria: Assessing the Merit of a Static Schedule 10  
    1.1.4. Optimizations ..........................................................................................13  
    1.1.5. Why Synchronous Clock-Driven Execution? ...............................16  
  1.2. Asynchronous Data-Driven Static Scheduling and Analysis .....................17  
    1.2.1. Asynchronous Data-Driven Execution Model .....................................18  
    1.2.2. Synchronous Clock-Driven vs. Asynchronous Data-Driven ............21  
  1.3. The Thesis ........................................................................................................21  
  1.4. Organization of the Dissertation .....................................................................23  
CHAPTER II Specification of Modules for Direct Special-Purpose  
  Algorithmic Approach ..................................................................................24  
  2.1. Simulation of Control Programs ....................................................................25  
  2.2. Specification of a Layered Software System for a Direct  
      Algorithmic Approach ...................................................................................27  
    2.2.1. Periodic_Function_Template .....................................................................28  
      2.2.1.1. Specification .............................................................................29  
      2.2.1.2. Discussion of Design Decisions .................................................39  
    2.2.2. Periodic_Boolean_Signal_Template ...................................................40  
      2.2.2.1. Specification .............................................................................40  
      2.2.2.2. Discussion of Design Decisions .................................................48  
    2.2.3. PBS_With_Delay_Template ................................................................49  
      2.2.3.1. Specification .............................................................................49  
      2.2.3.2. Discussion of Design Decisions .................................................55  
  2.3 Summary ...........................................................................................................56  
CHAPTER III Implementation of Modules for Direct Special-Purpose  
  Algorithmic Approach ..................................................................................57  
  3.1. Cost of Simulation ...........................................................................................58
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Block Diagram of a Digital Control System</td>
<td>4</td>
</tr>
<tr>
<td>2. DAG Representation of a Digital Control Program</td>
<td>5</td>
</tr>
<tr>
<td>3. Partial DAG</td>
<td>8</td>
</tr>
<tr>
<td>4. DAG Representation Without Phase-Shifts</td>
<td>12</td>
</tr>
<tr>
<td>5. DAG Representation With Phase-Shifts</td>
<td>15</td>
</tr>
<tr>
<td>6. A/D1 and A/D2 Drive the Execution of A1</td>
<td>46</td>
</tr>
<tr>
<td>7. Cyclic Graph in V Array (Example 1)</td>
<td>84</td>
</tr>
<tr>
<td>8. Cyclic Graph in V Array (Example 2)</td>
<td>85</td>
</tr>
<tr>
<td>9. DAG Representation of a Digital Control Program</td>
<td>88</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Basic System Parameters</td>
<td>7</td>
</tr>
<tr>
<td>2. Partial DAG Schedule</td>
<td>9</td>
</tr>
<tr>
<td>3. Synchronized System Parameters</td>
<td>10</td>
</tr>
<tr>
<td>4. Synchronous Schedule without Phase Shifts</td>
<td>11</td>
</tr>
<tr>
<td>5. Synchronous Schedule with Phase Shifts</td>
<td>14</td>
</tr>
<tr>
<td>6. Asynchronous Schedule with Phase Shifts</td>
<td>20</td>
</tr>
<tr>
<td>7. Make_Simple_Periodic_Function Operation</td>
<td>32</td>
</tr>
<tr>
<td>8. Shift Operation</td>
<td>33</td>
</tr>
<tr>
<td>9. Or Operation (Example 1)</td>
<td>34</td>
</tr>
<tr>
<td>10. Or Operation (Example 2)</td>
<td>36</td>
</tr>
<tr>
<td>11. Or Operation (Example 3)</td>
<td>37</td>
</tr>
<tr>
<td>12. Or Operation (Example 4)</td>
<td>39</td>
</tr>
<tr>
<td>13. An Example Periodic_Boolean_Signal</td>
<td>45</td>
</tr>
<tr>
<td>14. Example Operations in PBS</td>
<td>47</td>
</tr>
<tr>
<td>15. Effect of Delay (pbs, 2)</td>
<td>53</td>
</tr>
<tr>
<td>16. Effect of Delay (pbs, 5)</td>
<td>54</td>
</tr>
<tr>
<td>17. Effect of Delay (pbs, 3)</td>
<td>55</td>
</tr>
<tr>
<td>18. Program Representation of PBS (Example 1)</td>
<td>62</td>
</tr>
<tr>
<td>19. Program Representation of PBS (Example 2)</td>
<td>63</td>
</tr>
<tr>
<td>20. Make_Simple_Periodic_Boolean_Signal Operation</td>
<td>64</td>
</tr>
<tr>
<td>21. Shift Operation</td>
<td>66</td>
</tr>
<tr>
<td>22. Or Operation (Example 1)</td>
<td>68</td>
</tr>
<tr>
<td>23. Or Operation (Example 2)</td>
<td>71</td>
</tr>
<tr>
<td>24. Smallest Pulse Pattern</td>
<td>72</td>
</tr>
<tr>
<td>25. Sliding_Window_Approach (Example 1)</td>
<td>74</td>
</tr>
</tbody>
</table>
CHAPTER I

Introduction

It is now routinely possible to design and implement very complicated high-order
digital controllers that provide automatic control for large, complex physical systems
[Hostetter 88]. Examples of such controllers are automotive ignition control systems,
nuclear reactors, flight control systems and process control systems for chemical plants.

The software implementation of a digital controller that automates a complex
physical system must be capable of tracking all relevant state changes within the system.
This implies that the software may be required to sample numerous system inputs
repeatedly at different time intervals. Furthermore, the software may be composed of
many interdependent computational processes, each responsible for processing some
portion of the system's state information according to strict timing constraints. In other
words, a digital controller must repeatedly monitor the state of a system and process this
sensed information within predefined time bounds; failure of the system to respond in a
"timely" manner can lead to catastrophic results [Stankovic 88].

Due to the potential complexity of current digital controllers, it is crucial that their
software implementations be as efficient as possible [Warwick 86]. One factor in
ensuring this desired efficiency is the ability to schedule the digital control software
such that hardware resource requirements are minimized without jeopardizing the
performance of the automated physical system. Scheduling a large, complex software
system that must satisfy timing requirements is a very difficult optimization problem
[Wirth 77]. The problem becomes even more complex when the software system
requires execution on multiple processors [Liu 73]. Additionally, it is important that the
timing analysis of the software implementation can be performed in a computationally
tractable manner [Avrunin 90].
Much of the current research activity in real-time software systems attempts to address these issues, namely scheduling and analysis of large software systems with timing constraints on multiple processors, e.g., [Sprunt 89], [Liu 90], [Mok 87]. Unfortunately, many of these techniques involve dynamic algorithms which tend to be complex, run-time expensive and unable to guarantee that the software system meets its performance criteria [Cheng 88]. These techniques may be necessary for certain types of real-time software systems but often are not applicable to digital control software whose very correctness depends upon the guaranteed satisfaction of system timing requirements. Static scheduling techniques must be employed in order to ensure that such "hard real-time" digital control software provides adequate control of the system it automates. Although there has been some work in the past on static scheduling and analysis of software systems with multiple timing constraints, e.g., [Lee 87], surprisingly little has been published that applies to systems with properties comparable to those of a typical digital control software system.

How, then, is the efficient scheduling and analysis of digital control software currently being addressed? The responsibility for these tasks tends to fall on the shoulders of a digital control engineer who is forced to use an ad hoc "synchronous clock-driven" approach [Katz 81], [Ackerman 85], [Warwick 86]. The description of these techniques as "ad hoc" is by no means a criticism of the methods employed by digital control engineers. After all, numerous successful control programs for large systems, such as the space shuttle, have been developed and are in use today.

However, several interesting questions arise from the fact that scheduling and analysis of current digital controllers is performed via a synchronous clock-driven approach. Are there alternative techniques that would allow the efficient static construction of a feasible schedule for the software implementation of a digital controller? Would such techniques allow analysis of the system's performance criteria to be performed in a computationally tractable manner? Furthermore, could such techniques produce a schedule that is more efficient, perhaps by reducing hardware requirements or providing a faster response to system state information changes?

This dissertation explores an "asynchronous data-driven" static scheduling and analysis technique for the software implementation of a digital controller on a shared memory multiprocessor system. A formal mathematical specification of a software module (called the Periodic_Boolean_Signal_Template) that was designed and
implemented to facilitate this technique is described. This module provides an abstract
data type that models the relevant execution properties of a process autonomously
executing on a single processor. The specification abstracts away the explicit passage of
time and the operational characteristics of a process, greatly simplifying the problem of
timing analysis of a large class of digital controllers. A companion tool developed as
part of this research, the Static Analyzer/Scheduler, is also discussed. This tool
efficiently constructs a feasible static periodic schedule for the software implementation
of a digital controller. Based on the constructed software schedule, the tool analyzes
whether system timing requirements are satisfied, and performs process-to-processor
assignment for a multiprocessor implementation of the controller.

Section 1.1 describes the synchronous clock-driven method currently used by
control engineers to address the scheduling and analysis of digital control programs.
The optimization problems that arise when employing this method are discussed. An
example digital control system is used to illustrate these issues. Section 1.2 introduces
an alternative asynchronous data-driven approach to the scheduling and analysis
problem for digital control programs. This technique is contrasted with the synchronous
method in order to explore their inherent differences and similarities. Section 1.3
highlights the thesis of this work and Section 1.4 serves as an outline to the remainder
of the dissertation.

1.1. Synchronous Clock-Driven Static Scheduling and Analysis

Prior to the scheduling and analysis of a control program, a control engineer
generally must perform the following tasks.

As an end product of the design and synthesis phase, a control engineer constructs a
block diagram model of a digital controller for an automated physical system. Figure 1
depicts a block diagram model for a typical digital control system.

The controller in Figure 1 consists of two (possibly complex) computational
processes, C1 and C2, and two adders, A1 and A2. This particular control program
samples three distinct system values, two of which originate as plant output values and
one of which is available from this subsystem's external environment. The controller
contains two embedded control loops, L1 and L2.
In order to effectively analyze and schedule a digital control program, its block diagram model is translated into a directed acyclic graph (DAG). This is performed by "cutting" any closed loops embedded within the program, which can be done under most circumstances because each loop includes an external physical system (called PLANT here) that is not part of the control software. For the block diagram of Figure 1 this results in the DAG depicted in Figure 2. Vertices of the DAG correspond to computational processes in the controller and edges of the DAG correspond to control or data flow dependencies between the individual processes. In addition to the two computational processes and two adders, the DAG contains three analog/digital converters, A/D1, A/D2 and A/D3, and one digital/analog converter, D/A1. An analog/digital converter repeatedly samples an analog signal (the output of the plant or its external environment) and converts it into its digital representation for use by the digital control program. A digital/analog converter translates a digital signal (an output of the control program) into an analog signal for use by the plant.
The control engineer also must determine the following set of system parameters for the control program.

1. The maximum sampling period of each input to the system that is sampled by an A/D converter. This is determined by an analysis of bandwidth information known to the control engineer, and is based on the well-known Sampling Theorem [Shannon 49].

2. The compute delay required for execution of each computational process within the system. This is determined by executing each process on its own processor and simply timing it [Stoyenko91] [Shaw84].

After the block diagram model of a digital controller is translated to its DAG representation, and system parameters are chosen, the control engineer can apply the synchronous clock-driven approach to schedule and analyze the control software. This approach is described in the following three sections. Section 1.1.1 formally defines the process timing requirements of the synchronous clock-driven execution model. Section
1.1.2 defines the scheduling constraints for the model, and Section 1.1.3 describes measures for judging the merit of a static schedule. Section 1.1.4 presents some of the optimization problems that must be addressed when applying the synchronous clock-driven approach to the scheduling and analysis of digital controllers. Section 1.1.5 discusses a conjecture as to why an alternative to the synchronous clock-driven technique apparently has not been carefully explored in the past.

1.1.1. Synchronous Clock-Driven Execution Model

In the synchronous clock-driven execution model, process timing requirements are defined as follows:

- A process set \( P = \{p_1, p_2, \ldots, p_n\} \) represents the complete set of computational processes within a control program. This set can be partitioned into two subsets, \( SP \) and \( NSP \), for sampling processes and non-sampling processes, respectively.

- A process \( p_i \) in \( P \) is defined by a tuple \((m_i, d_i)\):
  
  (a) \( m_i \) is an integer-valued time called the \textit{maximum simple period} of \( p_i \), and \( p_i \) executes exactly once during \( m_i \).
  
  (b) \( d_i \) is an integer-valued time called the \textit{compute delay} of \( p_i \) on a single processor, and \( d_i \) is fixed and predetermined for \( p_i \). It is assumed that \( 0 < d_i \leq m_i \).

The set \( P \) corresponds to the set of vertices in the DAG representation of a control program. Each vertex in the DAG may be modeled by a process tuple. The value \( m_i \) is supplied by the control engineer for each process \( p_i \) in \( SP \). The value \( d_i \) is supplied by the control engineer for each process \( p_i \) in \( P \). An arc in the DAG defines a precedence relationship between process \( p_i \) and \( p_j \) based upon data flow constraints. An example set of basic system parameters for the DAG of Figure 2, \( m_i \)'s for the A/D converters and \( d_i \)'s for all processes, is shown in Table 1.
1.1.2. Scheduling Constraints

Scheduling constraints for the synchronous clock-driven execution model are:

1. The control program is able to track system changes at a rate fast enough to satisfy system timing requirements.

   This constraint is partially satisfied by ensuring that each process $p_i$ in SP executes at least once during each interval of length $m_i$. The other requirement is that a process in NSP executes upon initiation regardless of whether new data is available on an input arc [Mok 88], because its output also may be a function of time itself (e.g., the process may be an integrator).

2. The set $M = \{m_1, m_2, ..., m_n\}$ for all processes in SP must be "synchronized" resulting in the set $T = \{t_1, t_2, ..., t_n\}$ such that $T$ represents a set of synchronized periods with the following properties: for every $i$, $d_i \leq t_i \leq m_i$; and for every pair $(i, j)$, either $t_i$ divides $t_j$ or $t_j$ divides $t_i$. (Note that each clock-tick of the underlying real-time clock is one time unit. All process compute delays and sampling periods are integer multiples of a clock-tick and process execution is synchronized relative to a clock-tick.)
The scheduling constraint specified in (1) is obvious—the control program must sample system inputs at a rate sufficient to track any data changes—but the importance of constraint (2) is not immediately obvious and requires further discussion.

Removing a small portion of the DAG of Figure 2, the A/D1, A/D2 and A1 processes, results in the partial DAG of Figure 3.

Scheduling this small example DAG in accordance with scheduling constraint (1), and using the non-synchronized system parameters of Table 1, results in the schedule of Table 2. (Execution of a process within a schedule is depicted by its time line being in a high position, inactivity of a process is depicted by its time line being in a low position.)

The process A1 should execute each time new data is received from either A/D1 or A/D2 in order to track their changes. The execution pattern exhibited by A1, if this constraint is met, is complex periodic, that is, A1 starts execution at times 1, 11, 16, and 21 within a time period of 30. This is due to the fact that the mj's of A/D1 and A/D2 are unsynchronized. It is usually assumed to be computationally intensive to determine the complex periodic behavior of processes with unsynchronized execution. Additionally, it is assumed that implementing a schedule for processes with complex periodic behavior is difficult. These two assumptions lead to the inclusion of scheduling constraint (2) in the synchronous clock-driven execution model.
After determining the $t_i$'s for each process $p_i$ in SP it is possible to ensure that each process in NSP satisfies scheduling constraint (1). This is achieved by setting $t_i$ to be the minimum $t_j$ of all predecessor vertices $p_j$. For example, $t_{A1}$ in Figure 2 is the minimum of $t_{A/D1}$ and $t_{A/D2}$. Calculation of $t_i$'s is performed in a top-down fashion using the DAG model of the digital control software.

The $m_i$'s in Table 1 are not integer multiples of one another, so the $m_i$'s of the A/D converters must be synchronized in order to satisfy scheduling constraint (2). Table 3 depicts one possible modification to the $m_i$'s of the A/D converters and the subsequent $t_i$'s for all non-sampling processes in the control program, computed as described above. It can be seen that A/D2 is sampled twice as fast as necessary (according to the underlying real-time system behavior) in order to achieve synchronized sampling periods.
### Table 3: Synchronized System Parameters

<table>
<thead>
<tr>
<th>$p_i$</th>
<th>$t_i$</th>
<th>$d_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A/D1</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>A/D2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>A/D3</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>C1</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>C2</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>A1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>A2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>D/A1</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

1.1.3. **Scheduling Criteria: Assessing the Merit of a Static Schedule**

There are many measures that can be used to judge the merits of a particular static schedule for a control program. In this dissertation, the number of processors required for a schedule's implementation, and the end-to-end system delay incurred by the schedule, are used as criteria to assess the merit of a static schedule. It is of course, generally desirable to minimize the number of processors required for the implementation of the control software in order to minimize costs. End-to-end system delay is the time interval between the start of sampling an input to the controller and the time that sample is first reflected in an output of the controller. In certain cases, a maximum end-to-end delay interval could be a timing requirement of the physical system. This occurs whenever a process must complete its execution by a specified deadline. In this instance, end-to-end delay is no longer a measure of the merit of a schedule but becomes an important scheduling constraint.

A straightforward application of the system parameters of Table 3 results in the time line schedules depicted in Table 4. Figure 4 is a DAG representation of the system (annotated with the time line schedule of each process) used to clarify the following discussion. In Figure 4, a time line schedule for a process is depicted as a sequence of tuples, $<(a_0, a_1),..., (a_k, a_m)>$. The first member of each tuple represents the start time of
an execution initiation for a process. The second member of a tuple represents the end
time of that execution initiation, that is, the time when the outputs of a process are
available as input to its immediate successor processes. Certain end time tuple members
are shown in bold in order to more easily trace the flow of data through the system from
the time data is initially sampled by A/D1 until it becomes available as system output
from D/A1.

Table 4: Synchronous Schedule without Phase Shifts

<table>
<thead>
<tr>
<th>Pi</th>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>![Schedule Diagram]</td>
</tr>
</tbody>
</table>

*Table 4: Synchronous Schedule without Phase Shifts*
Each process starts execution at time 0 within its period and therefore must execute on its own processor. Furthermore, for this schedule the delay between the start of sampling at A/D1 and A/D2 and the time of these values affecting the output at D/A1 is very long—26 time units. This delay can be determined from Figure 4 by tracing the bold times along the path from A/D1 to D/A1. For example, the inputs to A1 are available at time 1, but their sum does not appear at the output of A1 until time 6. Furthermore, the inputs to C1 are available at time 6 but are not available at the output of C1 until time 14, and so forth. The static schedule of Table 4 is unattractive due to the large number of processors required for its implementation and its lengthy end-to-end delay.
1.1.4. Optimizations

To optimize the scheduling criteria for a static schedule, it is necessary to phase shift the start of execution of some processes within their period. Phase shifts for $P$ are chosen as $S = \{s_1, s_2, ..., s_n\}$, where $s_i$ is called the lag-time or phase for the start of execution of $p_i$ in its period of length $t_i$. It is required that $0 < s_i < t_i$.

By phase-shifting the schedules of several of the compute processes in Table 4 it is possible to optimize the scheduling criteria for this control program. A phase shift that achieves smaller end-to-end system delay and fewer processors is depicted in Table 5; see also the annotated DAG in Figure 5. Sweeping a vertical line through each process from time 0 to time 14 in the schedule demonstrates that only four processors are required for the execution of this control program schedule. The maximum time required for a sampled input to be processed by the control program is only 12 time units. This is a big improvement over the unshifted schedule, and is optimal because the total compute delay along the longest path in the DAG is 12 units.
Table 5: Synchronous Schedule with Phase Shifts

<table>
<thead>
<tr>
<th>( p_i )</th>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>A/D1</td>
<td></td>
</tr>
<tr>
<td>A/D2</td>
<td></td>
</tr>
<tr>
<td>A/D3</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>D/A1</td>
<td></td>
</tr>
</tbody>
</table>
Computing a schedule for a control program with synchronous clock driven execution therefore requires the DAG representation of the program, an \((m_i, d_i)\) tuple for each sampling process, and the \(d_i\) value for each non-sampling process. Given this information a computed schedule consists of a \((t_i, s_i, u_i)\) tuple for each \(p_i\) in \(P\) (\(u_i\) signifies that process \(p_i\) is assigned to processor \(u_i\)). Choices made for the values in sets \(T\) and \(S\) directly affect the optimization of the scheduling criteria for a static schedule.

Examination of the synchronous clock-driven execution model for a digital control program reveals that two difficult optimization problems arise. Variations of these have been shown to be NP-hard in [Garey 79]. These problems must be addressed during the analysis and scheduling of the control program if minimization of processor resources and/or minimization of end-to-end system delay is desired.
At the front end, it is necessary to adjust the synchronization of the periods of all computing processes. Even though the example digital controller in Section 1.1.1 is almost trivial, it is not directly apparent which choice of $t_i$'s leads to the best use of processor resources or to the best end-to-end system response time. For example, one might choose $t_1 = t_2 = t_3 = 10$. As the digital control system becomes more complex, synchronization of its execution becomes an overwhelming task.

At the back end, it is necessary to optimize the lag-times. These phase shifts are employed to reduce processor resource requirements but must be chosen carefully in order to minimize additional end-to-end processing delays incurred by the execution of the schedule. The choice of lag-times depends not only on the precedence relationships that exist among processes in the control program but also on the measures to be optimized.

It is interesting that several different modifications could have been applied to the $m_i$'s of the example digital controller of Section 1.1.1 in order to synchronize its execution. Each different modification choice has a direct impact upon the number of processors required for execution of the control program and also has a direct affect on the system response time. As the hardware requirements decrease, system response time generally increases, but not always. This is not a surprising effect but is one that must be taken into account when optimizing the synchronization of the execution of the control program.

1.1.5. Why Synchronous Clock-Driven Execution?

Why has the execution of digital control programs generally been viewed as strictly synchronous and clock-driven? Although this is merely a conjecture, it seems apparent that a commonly held view is that dealing with synchronous behavior is “easier” than dealing with asynchronous behavior.

The execution of a synchronous program can be controlled by a simple real-time clock. The execution times of compute processes within the program are fixed as offsets to the clock cycle. The rate of execution of a process is fixed relative to the clock cycle and can be controlled by a simple counter mechanism. All of this implies that the software execution of a synchronous program can be implemented using very simple mechanisms.
Each compute process in a synchronous program has a simple periodic execution pattern. This choice can be directly attributed to the Sampling Theorem, which states that a bandlimited continuous signal can be reconstructed by sampling its values at a set of equally spaced intervals at a rate at least twice its bandwidth.

The above discussion gives some credence to the belief that the synchronous clock driven approach leads to a straightforward implementation of a digital control program. But nothing precludes an alternative asynchronous view of the execution of a control program. Perhaps it is possible to implement an asynchronously executing controller without complex mechanisms. Perhaps it is possible to sample a continuous signal at unevenly spaced intervals and still be able to accurately track and reconstruct its value. Perhaps the optimizations required for the analysis and scheduling of an asynchronous digital control system are simpler (or at least no worse) than the optimizations required for a synchronous system. Finally, perhaps the asynchronous approach actually can result in fewer processors or better system response time.

1.2. Asynchronous Data-Driven Static Scheduling and Analysis

In order to view the execution of a digital control system as asynchronous and data-driven, the same basic process timing requirements must be satisfied. That is, each process $p_i$ in SP is defined by an $(m_i, d_i)$ tuple; and each process $p_i$ in NSP has a fixed delay $d_i$. The difference is in the definition of a feasible schedule: the execution of each process in NSP is data-driven rather than clock-driven.

The asynchronous data-driven view of a program is explored in the following two sections. The operational characteristics which define the asynchronous data-driven execution model are explained in Section 1.2.1. An example digital control system and its asynchronous schedule are used for illustration. Section 1.2.2 explores the inherent differences and similarities between synchronous clock-driven execution and asynchronous data-driven execution.
1.2.1. Asynchronous Data-Driven Execution Model

Asynchronous data-driven execution of a compute process within a digital control program was introduced in [Brown 84] and discussed further (in a different context) in [Stovsky 90]. It is also used in [Mok 87].

In the DAG representation of a control system, a process in SP has no input arcs, while all processes in NSP have at least one input arc. The execution of a process in SP is determined by \( m_i \); there is no synchronization of periods, so \( t_i = m_i \). Each process in NSP executes whenever new data is available on any input arc, unless it is already executing, in which case it is reinitiated upon completion of its current execution.

Operationally, a process \( p_i \) in NSP behaves as follows when it executes. All input data is read simultaneously, \( p_i \) computes for a fixed amount of time (its delay \( d_i \)) and then simultaneously produces outputs on all its output arcs. If new data becomes available on one or more input arcs while \( p_i \) is executing, it is reinitiated (only once) after its current execution completes. Multiple data changes along a single input arc, which occur while \( p_i \) is executing, cause only one new initiation of \( p_i \) using the most recently produced values on each input arc; there is no queuing of data values.

Processes in SP drive the initiation of all processes in NSP by periodically sampling system information with periods \( t_i = m_i \). Upon initiation, a process \( p_i \) in SP computes for \( d_i \) units of time and then initiates the execution of all processes to which it is a predecessor within the DAG. In turn, a process \( p_i \) in NSP, when initiated, computes \( d_i \) units of time and initiates all processes to which it is a predecessor. The initiation of processes filters down the DAG from top to bottom.

If the program is allowed to execute freely this way in a data-driven manner, then an initial transient period occurs where the execution pattern of processes is non-periodic. Following this transient phase, the control program settles into an equilibrium phase where all execution patterns are repetitive and periodic. The periodic behavior is generally quite complex, however, and each process in NSP may execute several times during a single period.

A static schedule for a process in SP is defined by the same parameters as in the clock-driven case. For a process \( p_i \) in NSP, a schedule consists of finding tuples \((c_i, \{s_i, j\})\) that could result from data-driven execution. Here:
(1) $c_i$ is an integer-valued time called the complex period of $p_i$. Process $p_i$ generally executes more than once within time $c_i$. For all $i$, $0 < d_i \leq c_i$.

(2) $\langle s_{i,j} \rangle = \langle s_{i,1}, s_{i,2}, ..., s_{i,m} \rangle$ is a sequence of possibly unequally-spaced integer-valued lag-times for the start of executions of $p_i$ in the interval $[0, c_i]$. Defining $s_{i,0} = 0$ and $s_{i,m} = c_i$, for all $i$ we have $0 < s_{i,k+1} - s_{i,k} \leq d_i$ for all $0 \leq k \leq m$.

Only scheduling constraint (1) of the synchronous clock-driven execution model (tracking of system changes) applies to the asynchronous data-driven execution model. There is no need to synchronize the periods of the processes. Scheduling criteria are the same for both models. In short, the basic difference between the synchronous and asynchronous execution models, with respect to static scheduling, is that all processes exhibit simple periodic behavior in the former; whereas non-sampling processes may exhibit complex periodic behavior in the latter.

Using the digital control system of Section 1.1 and the basic system parameters of Table 1, one possible schedule for the asynchronous execution of the control program in its equilibrium phase is depicted in Table 6. This schedule was produced using the techniques developed for this research and described in Chapters 2, 3 and 4.
Sweeping a vertical line through each process in this schedule, from time 0 to time 30, demonstrates that three processors are required for the implementation of this asynchronous program (vs. four for the best synchronous schedule). The end-to-end system response time of the asynchronous schedule is twelve time units, which is optimum and the same as the best synchronous schedule.

Computing a schedule for a control program with asynchronous data-driven execution requires the DAG representation of the program, an \((m_i, d_i)\) tuple for each sampling process, and the \(d_i\) value for each non-sampling process. Given this information, a computed schedule consists of a tuple \((m_i, s_i, u_i)\) for each process \(p_i\) in SP and a tuple \((c_i, \langle s_{i,k}, \langle u_{i,k} \rangle \rangle)\) for each process \(p_i\) in NSP \((u_{i,k}\) signifies that \(p_i\) is assigned to processor \(u_{i,k}\) for the execution whose phase is \(s_{i,k}\)). Choices made for the values in set \(S\) directly affect the optimization of the scheduling criteria for a static schedule in this case.
1.2.2. **Synchronous Clock-Driven vs. Asynchronous Data-Driven**

The inherent differences between these two approaches are quite simple and obvious. In the synchronous clock-driven execution model, the execution of processes must be synchronized with each other ensuring that each \( p_i \) executes once within its period. This requirement may add a degree of inflexibility when attempting to schedule the implementation of a synchronized digital control program. The optimal choice for synchronization of \( m_j \)'s is not immediately obvious even for simple control programs.

In asynchronous data-driven execution, processes in SP execute once within each period, while those in NSP are allowed to execute freely based upon the arrival of new data values and may therefore demonstrate complex periodic behavior. There is no notion of synchronized execution between processes. Because of this the optimization problem of synchronizing \( m_j \)'s does not arise.

It is important to note that phase shifting the execution of processes in SP in an asynchronously executing program has a direct impact upon the processor resources required by the program. Choosing the optimal phase shift in order to minimize resources is a complex optimization problem comparable in difficulty to the optimization of phase shifts in the synchronous clock-driven approach.

1.3. **The Thesis**

The analysis and scheduling of digital control systems currently requires that the operational characteristics of the software be defined using a synchronous clock-driven execution model. It therefore is interesting to ask whether an alternative asynchronous data-driven technique could be employed to address the problems of analysis and scheduling of these systems.

The thesis of this work has two parts and is summarized in the following claim: it is practical and potentially advantageous for static scheduling to define the operational characteristics of a digital control program using an asynchronous data-driven execution model.

1. Practical analysis of the steady state timing behavior of a digital control system, for static scheduling purposes, does not require synchronized execution of the individual processes within the system. A control
program with asynchronous data-driven execution can be analyzed in a computationally tractable manner.

To test this part of the thesis, a method for formally specifying the pertinent behavioral characteristics of a process is assessed. This method abstracts away complicating factors such as passage of time and process operational characteristics, thus providing a simple and elegant specification for the (possibly complex) periodic behavior of a process. The formal specification portion of a software module, Periodic_Boolean_Signal_Template, that embodies these abstractions, is introduced as the basis for the above claim.

In order to analyze the asynchronous behavior of a digital control system it is necessary to be able to calculate the steady state periodic execution behavior of each process in the system and to do so in a computationally tractable manner. The implementation portion of the Periodic_Boolean_Signal_Template is the basis for the claim of tractability. This implementation supports the efficient calculation of the infinite periodic execution history of a process based upon a predetermined set of system parameters.

(2) Determination of a feasible static multiprocessor schedule for an asynchronous data-driven control system can be efficiently implemented.

The Static Analyzer/Scheduler tool is introduced and described. This tool constructs a finite description of the infinite periodic execution history for each process in the control program. Analysis of this history allows the scheduler to determine if the control program is able to satisfy the timing requirements of the system it controls. If the timing requirements are verified for the control program, the Static Analyzer/Scheduler constructs a schedule for the entire control program. Based on this schedule the tool determines processor resource requirements and performs process-to-processor assignment. In some cases the resulting multiprocessor schedule uses fewer processing resources and/or has a shorter end-to-end system delay than synchronous clock-driven schedules.
1.4. **Organization of the Dissertation**

Chapter 2 describes a technique that facilitates the analysis and scheduling of an asynchronous data-driven control program. A formal specification of a software module, Periodic_Boolean_Signal_Template, designed to facilitate the analysis and scheduling process, is discussed. The decisions made during the design of the specification of this software module are explored. It is demonstrated that the Periodic_Boolean_Signal_Template provides a natural abstraction for the specification of a control program with asynchronous data-driven behavior.

Chapter 3 describes an implementation of the Periodic_Boolean_Signal_Template. The decisions made during the design of the implementation of this software module are explored. It is demonstrated that the Periodic_Boolean_Signal_Template directly supports the construction of the periodic execution history of a control program with asynchronous data-driven behavior. Furthermore, it is shown that this software module allows analysis and scheduling to be performed in a computationally tractable manner.

Chapter 4 discusses the Static Analyzer/Scheduler tool that is used to analyze and produce a multiprocessor schedule for a digital control program with asynchronous data-driven behavior. The implementation of the tool and its method for efficiently producing a feasible multiprocessor schedule are discussed. Furthermore, it is demonstrated that, in some cases, the Static Analyzer/Scheduler is capable of producing a more efficient schedule (in terms of reduced hardware requirements and/or improved end-to-end delay) than could be achieved by a synchronous clock-driven approach. A technique that can be employed to implement a static multiprocessor schedule for digital processes with complex periodic behavior is described. Special schedule implementation problems arise when the execution of a digital process is asynchronous and data-driven as opposed to synchronous and clock-driven. These problems and methods to address them are discussed.

Chapter 5 summarizes the goals and results of the research, along with possible extensions to it and open research issues yet to be resolved.
CHAPTER II

Specification of Modules for Direct Special-Purpose Algorithmic Approach

Construction of a feasible static schedule for a multiprocessor implementation of an asynchronous data-driven control program (henceforth, a "control program") requires the computation of the steady state periodic execution pattern of the control software. That is, it is necessary to determine the periodic timing behavior and interactions of individual processes in the control program if it were to execute in the "self-timed" fashion defined by the asynchronous data-driven execution model. It is desirable, of course, that the calculation of this behavior be as computationally inexpensive as possible. The computation can be carried out using one of two different techniques: by simulation of the execution of the control software, or by a direct special-purpose algorithmic approach. Both techniques were investigated in the course of this research, as reported in this chapter.

Section 2.1 discusses simulation techniques (a direct simulation of execution and a representational model of execution) to determine the steady state execution pattern of a control program. Simulation proved to be unsatisfactory for the scheduling problem posed in this dissertation so a more attractive, direct special-purpose algorithmic approach was investigated, as explained in Section 2.2. The design strategy of the direct algorithmic approach is to provide a software system that hides potentially complex algorithmic details and that supplies a simple specification of operations to construct a feasible static schedule for a control program. The system supplies operations to construct the timing behavior of a control process, to combine timing patterns, to determine whether a process is executing or initiating its predecessors at time \( i \), and to
determine the period of a process. Each of the operations is specified in a very straightforward and concise manner. Section 2.3 provides a summary of the chapter contents.

The abstract model used to capture the complex periodic timing behavior of a control process is responsible for the simplicity of the formal specifications. What information must be included in the abstract model? The model first must capture the complex periodic nature of the ON/OFF execution pattern of a control process. The Periodic_Function_Template does exactly that. It models process behavior as a complex periodic function that maps integers to booleans.

Unfortunately, the Periodic_Function_Template serves only to simplify the explanation of the other, more useful, concepts to come. The model specified in the Periodic_Function_Template is not sufficient to completely specify the timing behavior of a control process. The Periodic_Boolean_Signal_Template models this timing behavior with a pair of periodic functions that map integers to booleans. One of the functions represents the initiation pattern of the process while the other represents the execution pattern of the process.

The Periodic_Function_Template and the Periodic_Boolean_Signal_Template are general purpose modules that have a wide range of applications. The PBS_With_Delay_Template supplies an additional operation that captures the semantics of the asynchronous data-driven execution model. This module is application-specific for the special-purpose algorithmic approach discussed in Chapter 3.

The specification of each of these modules is discussed in the following subsections. It is demonstrated that the Periodic_Function_Template, the Periodic_Boolean_Signal_Template, and the PBS_With_Delay_Template modules, together, provide a natural abstract basis for determining the steady state timing behavior of a control program.

2.1. Simulation of Control Programs

Simulation of the execution behavior of a control program was initially investigated due to the relative simplicity of implementing it. A software simulation of a control program requires that the program be allowed to execute freely according to the timing requirements specified in the model described in Section 1.2.1. The simulation
progresses step-wise along an integer-valued time line beginning at time 0. At each step in the simulation a current execution trace is constructed and maintained for each computational process (as if executing on its own processor). Following each step in the simulation, the current execution trace of a process is compared to its past execution trace in order to determine whether its execution has reached a periodic steady state. Unfortunately, the number of simulation steps required before the steady state behavior of the system is recognized may be quite large, in the worst case exponential in the number of processes in the control program. For a control program with a large number of computational processes or with sampling periods and compute delays that are relatively large prime numbers, simulation is computationally expensive and therefore unattractive. Additional arguments to support this claim are found in [Avrunin 90].

In the real-time software systems community several alternative representational models have been investigated for specifying and validating the timing behavior of hard real-time system software. These models tend to be representationally complex and computationally intensive. Examples include Finite State Machines [Disarathy 85], Petri Nets [Coolahan 83], [Faustini 86] and the use of Guarded Commands in a parallel language (PARC) [Haase 81]. For the purposes of this research, each computational process in a control program was modeled as a Finite State Tranducer (FST) by adapting the techniques discussed in [Disarathy 85] and [Minsky 67]. Although it is possible to determine the periodic timing behavior of a control process using a Finite State Transducer, the model suffers from a combinatorial explosion in the number of states necessary to represent the control software, thus making this technique as unattractive as direct simulation.

Due to the computational complexity of simulation and the representational complexity of alternative models of a control program, these techniques are not discussed further in this dissertation. Instead, a direct algorithmic approach is applied to the problem of determining the steady state behavior of a control program. It is crucial that an algorithm used to solve this problem not suffer from the same complexity inherent in the techniques discussed above. Additionally, the algorithm must produce results in a form that facilitates the timing analysis and scheduling of a control program in a computationally tractable manner [XuParnas 91].
2.2. Specification of a Layered Software System for a Direct Algorithmic Approach

As previously stated, any method employed to determine the complex periodic execution pattern of an asynchronous data-driven process must be computationally tractable or there is no benefit to choosing the asynchronous data-driven execution model over the standard synchronous clock-driven one. The reason for this is that a static schedule must be determined in advance that mimics the steady state execution pattern that would arise from asynchronous execution. This complex periodic static schedule can then be used directly to determine when processes should execute during real-time control processing.

Investigation of current direct methods used to specify and validate the timing behavior of hard real-time systems reveals that these techniques tend to be very complex. It is interesting to note that they incorporate the notion of passage of time and the operational characteristics of a real-time process. Is it possible that the explicit inclusion of time and process operational characteristics are merely complicating factors in the algorithmic determination of the steady state behavior of a control program? Through a discussion, that follows, of the design and implementation of the Periodic_Boolean_Signal_Template, it is demonstrated that by abstracting away passage of time and detailed operational characteristics, the problem of determining the complex periodic timing behavior of a control program is greatly simplified.

Independently, and in parallel with our work, [Garland 90] has developed an abstract mathematical specification for an oscilloscope. An oscilloscope is modeled as a mathematical function that abstracts away the notion of time and the operational characteristics of the oscilloscope. This is particularly interesting because not only is the model greatly simplified by this abstraction, but timing analysis of the behavior of the oscilloscope becomes more tractable, too. This gives some credence to the claim that it is possible to efficiently determine the steady state behavior of a control process in a similar non-operational manner.

The RESOLVE (REUsable SOfware Language with Verifiability and Efficiency) specification language is used for the formal mathematical specification of the modules defined here. The RESOLVE programming language and system has been developed by the Reusable Software Research Group at The Ohio State University [Weide 91]. It
provides a framework for developing abstract specifications. In RESOLVE, a software component has a formal specification (in a module called a concept) written in terms of well-known mathematical theories; programming details are not included in the concept. An implementation of the component (in a module called a realization) describes the data structures and algorithms used to realize the abstraction. It is possible to layer specifications and implementations in RESOLVE, that is, to construct complex modules from more primitive ones.

Section 2.2.1 describes the mathematical specification of a basic software module, Periodic_Function_Template, that simplifies the explanation of the Periodic_Boolean_Signal_Template and the PBS_With_Delay_Template. Examples are employed to elucidate the discussion. It is shown that this specification defines a reusable and natural abstraction for a class of periodic functions. Section 2.2.2 describes the mathematical specification of another module, Periodic_Boolean_Signal_Template. The design of this module is symmetrical to the design of the Periodic_Function_Template of Section 2.2.1. It is demonstrated that the Periodic_Boolean_Signal_Template provides an abstraction for the complex periodic timing behavior of a control process. Section 2.2.3 describes a higher-level software module, PBS_With_Delay_Template, that is layered on top of the Periodic_Boolean_Signal_Template. This module is application-specific and is employed in the strategy to determine the periodic execution behavior of a control program. The module defines an abstraction for the operational characteristics of an asynchronous data-driven control process — in essence, its formal semantics.

2.2.1. Periodic_Function_Template

The next two subsections address the formal specification of the Periodic_Function_Template and the decisions made during its design. This concept is not used directly in solving the scheduling problem, but it illustrates the main ideas underlying the Periodic_Boolean_Signal_Template (presented in Section 2.2.2).
2.2.1.1. Specification

The following is a RESOLVE specification of a module, Periodic_Function_Template, that is referred to as PFT in the remainder of this discussion. The PFT module provides an abstraction for a class of periodic functions that map integers to booleans. Encapsulated in the PFT are the specifications for an abstract data type (Periodic_Function) and a set of basic operations to manipulate the data type. The conceptual context section and the interface section of the PFT module are discussed below using various examples to clarify the specification (which is complete and unambiguous but not terribly easy to understand from the formalism alone).

concept Periodic_Function_Template

conceptual context

uses

   Integer_Template

parameters

   facility Integer_Facility is Integer_Template

mathematics

   math definitions

      Is_Periodic : boolean
      parameters
         func : function from integer to boolean
         per : integer
         "per > 0 and
         for all i : integer
            (func(i) = func(i + per))"

interface

   type Periodic_Function is modeled by
      function from integer to boolean
      exemplar pf
      constraint "there exists i : integer
                  (Is_Periodic (pf, i))"
      initialization
         ensures "for all i : integer
                  (pf(i) = false)"
operation Make_Simple_Periodic_Function
parameters
  produces pf : Periodic_Function
  preserves period : Integer
requires "period > 0"
ensures "Is_Periodic (pf, period) and
  pf(0) = true and
  for all i : integer
    ((0 < i < period) implies
      pf(i) = false)"

operation Shift
parameters
  alters pf : Periodic_Function
  preserves s : Integer
ensures "for all i : integer
  (pf(i+s) = #pf(i))"

operation Or
parameters
  alters pf1 : Periodic_Function
  preserves pf2 : Periodic_Function
ensures "for all i : integer
  (pf1(i) = (#pf1(i) or pf2(i)))"

operation Smallest_Period returns period : Integer
parameters
  preserves pf : Periodic_Function
ensures "Is_Periodic (pf, period) and
  for all i : integer
    ((0 < i < period) implies
      (not Is_Periodic(pf, i)))"

operation Is_True returns control
parameters
  preserves pf : Periodic_Function
  preserves i : Integer
ensures Is_True iff "pf(i) = true"

end Periodic_Function_Template

The following discussion serves to clarify the formal specification of the Periodic_Function_Template.

Conceptual Context Section

The PFT module, through the uses section, imports the specification of an externally defined module, Integer_Template. The Integer_Template provides a program
type Integer (modeled by mathematical type integer). This template also provides the relevant operations on Integers, but these are not used in the specification (RESOLVE has no built-in types; otherwise this would not be necessary).

Parameters to the PFT module include an instance of the Integer_Template. The instantiation of the Integer_Template as a facility provides the program data type, Integer, which is the type of some of the parameters to PFT operations.

The math definitions section of the PFT module includes a boolean function, Is_Periodic. Is_Periodic has two parameters, a function that maps integers to booleans (func) and an integer (per). Is_Periodic (func, per) is true if and only if per > 0 and for every integer i, func(i) = func(i + per). In other words, Is_Periodic is true if and only if func is periodic with period per.

Interface Section

The type declaration of the PFT module specifies the abstract data type, Periodic_Function, that is exported by the module. Periodic_Function is a function that maps integers to booleans. Initially, a variable pf of this type satisfies pf(i) = false for each integer i. The invariant constraint on pf is that there exists some integer i such that Is_Periodic (pf, i) = true; i.e., pf is indeed a periodic function.

There are five basic operations on Periodic_Function variables. In the following discussion, when necessary, the result of invoking a particular operation is clarified with an example and a table (depicting the mathematical model of a periodic function, its period, and its “time line representation”). When a sequence of operations is used as an example, each operation is numbered and there is a numbered table entry for that operation.

- The operation Make_Simple_Periodic_Function (pf, period) supports the construction of a simple periodic function. That is, pf(i) = true for exactly one integer i = 0 within the interval [0, period), and pf is periodic with the given period.
As an example, if Make_Simple_Periodic_Function (pf, period) is invoked with period = 10, then pf(i*10) = true for every integer i, and pf is false elsewhere. The periodic function pf has a period of 10 and exactly one position within each period where pf = true. Thus pf(-10) = true, pf(0) = true, pf(10) = true, etc., while pf(j) = false for each integer j ≠ i*10 for some i. Table 7 illustrates the invocation of Make_Simple_Periodic_Function (pf, 10).

Table 7: Make_Simple_Periodic_Function Operation

<table>
<thead>
<tr>
<th>pf =</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>period of pf = 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The operation Shift (pf, s) performs a shift of pf by s.

As an example, let Shift (pf, shift), with shift = 2, be applied to the result produced by the above invocation of Make_Simple_Periodic_Function. Then pf(2+i*10) = true for every integer i. Thus pf(-8) = true, pf(2) = true, pf(12) = true, etc., while pf(j) = false for each integer j ≠ 2+i*10 for some i. Table 8 illustrates this operation.
Table 8: Shift Operation

<table>
<thead>
<tr>
<th>pf =</th>
</tr>
</thead>
<tbody>
<tr>
<td>F F T F F F F F F</td>
</tr>
</tbody>
</table>

| period of pf = 10 |

- The Or operation combines two periodic functions point-wise through the logical or operation.

The Or operation is a crucial operation for the abstraction of periodic functions that map integers to booleans. With just the first two operations of the PFT module, only functions with simple periodic behavior can be constructed and manipulated. The Or operation supports the construction of complex periodic functions, that is, functions that exhibit more than one position with a true value within each period. Several example applications of the operations in the PFT module illustrate this point.

Example 1

```plaintext
variable
  pf1 : Periodic_Function
  pf2 : Periodic_Function
begin
  Make_Simple_Periodic_Function(pf1, 10) (* line 1 *)
  Make_Simple_Periodic_Function(pf2, 10) (* line 2 *)
  Shift(pf1, 2) (* line 3 *)
  Shift(pf2, 4) (* line 4 *)
  Or(pf1, pf2) (* line 5 *)
end
```

Table 9 traces the actions of the operations in Example 1.
Additional examples of Or, where pf1 and pf2 have different periods, are presented after the explanation of the Smallest_Period operation.

- The operation Smallest_Period (pf) returns the smallest period of pf.
Example 2

\begin{verbatim}
variable
  pfl : Periodic_Function
  pf2 : Periodic_Function
  period : Integer
begin
  Make_Simple_Periodic_Function(pfl, 5)  (* line 1 *)
  Make_Simple_Periodic_Function(pf2, 10)  (* line 2 *)
  Shift(pfl, 2)  (* line 3 *)
  Shift(pf2, 5)  (* line 4 *)
  Or(pfl, pf2)  (* line 5 *)
  period := Smallest_Period(pfl)  (* line 6 *)
end
\end{verbatim}

Two simple periodic functions, pfl and pf2, are produced by the statements in lines 1 though 4. Application of the Or alters pfl so that it exhibits complex periodic behavior with \(pfl(2+i*5) = \text{true}\) and \(pfl(4+i*10) = \text{true}\) for every integer \(i\); pfl is false elsewhere. A period equal to 10 is returned by Smallest_Period (pfl) in line 6. It is important to note that the period of a Periodic_Function can be altered in several ways by the Or operation. Example 2 demonstrates the case where the periods of pfl and pf2 are unequal and their true values do not overlap; thus in a sense the functions are "out of phase" or "unsynchronized". When this occurs, the resulting period of pfl becomes the least common multiple of the initial periods of pfl and pf2. Table 10 traces the actions of the operations in Example 2.
Table 10: Or Operation (Example 2)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example 3

```
variable
    pf1 : Periodic_Function
    pf2 : Periodic_Function
    period : Integer
begin
    Make_Simple_Periodic_Function(pf1, 2) (* line 1 *)
    Make_Simple_Periodic_Function(pf2, 4) (* line 2 *)
    Shift(pf1, 1) (* line 3 *)
    Shift(pf2, 3) (* line 4 *)
    Or(pf1, pf2) (* line 5 *)
    period := Smallest_Period(pf1) (* line 6 *)
end
```

Two simple periodic functions are produced by the statements in lines 1 through 4. The Or operation does not change pf1 and the function still satisfies \( pf1(1+i*2) = \text{true} \) for every integer \( i \). Example 3 demonstrates the case where the periods of \( pf1 \) and \( pf2 \) are
unequal but their true values completely overlap, thus in a sense the functions are "in phase" or "synchronized". When this occurs the resulting period of pf1 becomes the minimum of the initial periods of pf1 and pf2, in this case 2. Table 11 illustrates the actions of the operations in Example 3.

Table 11: Or Operation (Example 3)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td><strong>(1)</strong></td>
<td>pf1 = TFFT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>period of pf1 = 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>(2)</strong></td>
<td>pf2 = TFFF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>period of pf2 = 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>(3)</strong></td>
<td>pf1 = FFFT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>period of pf1 = 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>(4)</strong></td>
<td>pf2 = FFFT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>period of pf2 = 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>(5)</strong></td>
<td>pf1 = FFFT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>period of pf1 = 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Example 4**

variable

\begin{align*}
\text{pf1} & : \text{Periodic\_Function} \\
\text{pf2} & : \text{Periodic\_Function} \\
\text{pf3} & : \text{Periodic\_Function} \\
\text{pf4} & : \text{Periodic\_Function} \\
\text{period} & : \text{Integer}
\end{align*}

\texttt{begin}

\begin{align*}
\text{Make\_Simple\_Periodic\_Function(}\text{pf1, 2}) & \quad (* \text{ line 1 *}) \\
\text{Make\_Simple\_Periodic\_Function(}\text{pf2, 4}) & \quad (* \text{ line 2 *}) \\
\text{Shift(}\text{pf1, 1}) & \quad (* \text{ line 3 *}) \\
\text{Shift(}\text{pf2, 3}) & \quad (* \text{ line 4 *}) \\
\text{Or(}\text{pf1, pf2}) & \quad (* \text{ line 5 *}) \\
\text{Make\_Simple\_Periodic\_Function(}\text{pf3, 5}) & \quad (* \text{ line 6 *}) \\
\text{Make\_Simple\_Periodic\_Function(}\text{pf4, 10}) & \quad (* \text{ line 7 *}) \\
\text{Shift(}\text{pf3, 3}) & \quad (* \text{ line 8 *}) \\
\text{Shift(}\text{pf4, 5}) & \quad (* \text{ line 9 *}) \\
\text{Or(}\text{pf3, pf4}) & \quad (* \text{ line 10 *}) \\
\text{Or(}\text{pf1, pf3}) & \quad (* \text{ line 11 *}) \\
\text{period} & := \text{Smallest\_Period(}\text{pf1}) \quad (* \text{ line 12 *})
\end{align*}

\texttt{end}

Example 4 is a more complex application of the operations of the PFT module. It combines examples (2) and (3) to demonstrate the Or operation applied to complex periodic functions. Or (pf1, pf3), on line 11, combines a simple periodic function, pf1, and a complex periodic function, pf3. The result of this operation produces a complex periodic function where pf1(1+i*2) = true, pf1(2+i*5) = true and pf1(4+i*10) = true for every integer i. Because the periods of pf1 and pf3 are unequal and the functions are out of phase, the period of pf1 is altered as in Example 2. Table 12 depicts only the application of line 11 since the actions of lines 1 through 10 are provided in Tables 10 and 11.
The operation Is_True (pf, i) returns a control value of true if and only if the value of pf at i is true.

This operation, like Smallest_Period, is used to observe a Periodic_Function pf after pf has been constructed by the other operations.

2.2.1.2. Discussion of Design Decisions

The Periodic_Function_Template serves to simplify the specification of a relatively complex software system designed to facilitate the scheduling of a control program. It is desirable that a specification provide a natural abstraction for the data type being exported. This implies that a specification must be clear and concise while correctly capturing the behavior to be modeled. It is also beneficial if the module can be reused in other application areas as well. The following discussion demonstrates how the design of the Periodic_Function_Template satisfies these criteria.

The abstract data type Periodic_Function that is exported by the module is modeled simply as a function that maps integers to booleans. A mapping from integers to booleans is natural for modeling the dynamic behavior of a control process that is in one of two states: idle or executing. The concept of periodicity must also be captured for sampling processes and asynchronous data-driven compute processes. The requirement that a Periodic_Function must exhibit periodic behavior is included in the design through an invariant specification using the Is_Periodic function.
The five operations that manipulate the Periodic_Function data type are primitive (each performing one simple action), leading to a simple and concise specification of behavior. They are also complete, permitting the construction and manipulation of any periodic function in an obvious way. The Periodic_Function_Template module thereby provides a very straightforward and concise specification for a module that manipulates this class of periodic functions.

Clearly there are many other alternative operations that might be designed to manipulate periodic functions in this class. The actual operations included in the design of the Periodic_Function_Template constitute one convenient basis of primitive and low-level operations that support the construction and modification of any periodic function that maps integers to booleans.

2.2.2. Periodic_Boolean_Signal_Template

As noted earlier, the Periodic_Function_Template can be used to capture the dynamics of the ON/OFF state of an individual process. But to model process interactions, a slightly more complex module is needed.

The next two subsections discuss the formal specification of the Periodic_Boolean_Signal_Template and the decisions made during its design. It is demonstrated that the Periodic_Boolean_Signal_Template provides a natural abstraction for modeling the steady state periodic timing behavior of a control process in the context of a control program with other processes.

2.2.2.1. Specification

The following is a RESOLVE specification of a module, Periodic_Boolean_Signal_Template, that is referred to as PBS in the remainder of this discussion. The PBS module provides an abstraction for the steady state periodic timing behavior of a control process. Encapsulated in the PBS are the specifications for an abstract data type, Periodic_Boolean_Signal, and a set of basic operations to manipulate the data type. The design of the PBS module very closely parallels and is symmetrical to the design of the Periodic_Function_Template (PFT) of Section 2.2.1. Only the portions of the conceptual context section and the interface section of the PBS module
that differ significantly from the PFT module are discussed below. When necessary, an example and table (formatted as in Section 2.2.1) are used for illustration.

**concept** Periodic_Boolean_Signal_Template

**conceptual context**

uses

Integer_Template

parameters

facility Integer_Facility is Integer_Template

**mathematics**

**math definitions**

Is_Periodic : boolean

parameters

func : function from integer to boolean

per : integer

"per > 0 and for all i : integer (func(i) = func(i+per))"

Is_PBS : boolean

parameters

pul : function from integer to boolean

lev : function from integer to boolean

"there exists i : integer (Is_Periodic (pul, i) and Is_Periodic (lev, i)) and for all i : integer ((pul(i) = true implies lev(i) = true) and ((lev(i) = true and lev(i+1) = false) implies (pul(i) = true))"

**interface**

type Periodic_Boolean_Signal is modeled by

**tuple**

pulse : function from integer to boolean

level : function from integer to boolean

exemplar pbs

constraint "Is_PBS (pbs.pulse, pbs.level)"

initialization

ensures "for all i : integer (pbs.pulse(i) = false and pbs.level(i) = false)"

**operation** Make_Simple_Periodic_Boolean_Signal
parameters
produces pbs : Periodic_Boolean_Signal
preserves period : Integer
preserves duration : Integer
requires "period > 0 and 1 \leq duration \leq period"
ensures "Is_Periodic (pbs.pulse, period) and
Is_Periodic (pbs.level, period) and
pbs.pulse(duration-1) = true and
for all i : integer
  (((0 \leq i < period and i \neq duration-1) implies
    pbs.pulse(i) = false) and
  ((0 \leq i < duration) implies
    pbs.level(i) = true) and
  ((duration \leq i < period) implies
    pbs.level(i) = false))"

operation Shift
parameters
alters pbs : Periodic_Boolean_Signal
preserves s : Integer
ensures "for all i : integer
  (pbs.pulse(i+s) = \#pbs.pulse(i) and
  pbs.level(i+s) = \#pbs.level(i))"

operation Or
parameters
alters pbs1 : Periodic_Boolean_Signal
preserves pbs2 : Periodic_Boolean_Signal
ensures "for all i : integer
  (pbs1.pulse(i) = (#pbs1.pulse(i) or
    pbs2.pulse(i)) and
  pbs1.level(i) = (#pbs1.level(i) or
    pbs2.level(i)))"

operation Smallest_Pulse_Period returns period : Integer
parameters
preserves pbs : Periodic_Boolean_Signal
ensures "Is_Periodic (pbs.pulse, period) and
for all i : integer
  ((0 < i < period) implies
    (not Is_Periodic (pbs.pulse, i))"

operation Smallest_Level_Period returns period : Integer
parameters
preserves pbs : Periodic_Boolean_Signal
ensures "Is_Periodic (pbs.level, period) and
for all i : integer
  ((0 < i < period) implies
    (not Is_Periodic (pbs.level, i))"

operation Is_True_Pulse returns control
parameters
preserves pbs : Periodic_Boolean_Signal
preserves i : Integer
\begin{verbatim}
ensures Is_True_Pulse iff "pbs.pulse(i) = true"

operation Is_True_Level returns control
parameters
  preserves pbs : Periodic_Boolean_Signal
  preserves i : Integer
ensures Is_True_Level iff "pbs.level(i) = true"
end Periodic_Boolean_Signal_Template
\end{verbatim}

The following discussion serves to clarify the formal specification of the Periodic_Boolean_Signal_Template. Only those sections that differ significantly from the specification of the Periodic_Function_Template are addressed.

\textit{Conceptual Context Section}

The \texttt{uses} section and the \texttt{parameters} section of the PBS module parallel the design found in the PFT module.

The \texttt{math definitions} section of the PBS module includes a boolean function, Is_Periodic, that has been discussed previously in the explanation of the Periodic_Function_Template. This section also includes a boolean function, Is_PBS. Is_PBS has two parameters, pul and lev. Is_PBS returns true if and only if pul and lev are periodic functions, if pul(i) = true then lev(i) = true for every integer i, and if lev(i) = true and lev(i+1) = true then pul(i) = true for every integer i. The significance of these conditions is explained below.

\textit{Interface Section}

The \texttt{type} declaration of the PBS module specifies the abstract data type, Periodic_Boolean_Signal that is exported by the module. Periodic_Boolean_Signal is a 2-tuple consisting of two periodic functions, pulse and level, that map integers to booleans. Initially, pbs.pulse(i) and pbs.level(i) are false for each integer i. The invariant constraint on pbs is that Is_PBS (pbs.pulse, pbs.level) = true.

The Is_PBS constraint ensures that a Periodic_Boolean_Signal correctly captures the timing behavior of a control process. A true value for pulse represents the end of a computation during an ON state. A true value in level represents an ON state.
(computing) while a false value represents an OFF state (idle). In this way, the periodic initiation pattern of a process is represented by pbs.pulse while the periodic ON/OFF execution pattern of a process is represented by pbs.level. It is clear, therefore, that pbs.pulse and pbs.level must have the same period in order to correctly mirror the periodic timing behavior of a control process. Is_PBS requires that pbs.pulse and pbs.level are periodic (with the same period). The initiation pattern and the execution pattern of a control process are related in the following manner:

- If a process initiates its successors at time i, it must also be executing at time i.
- A process must immediately initiate the execution of its successors after completing its current execution.

Table 13 depicts an example Periodic_Boolean_Signal that might be constructed via the operations supplied by the PBS module. In order to verify that Is_PBS is satisfied for this example notice that both pbs.pulse and pbs.level are periodic with a period of 8. The pbs.pulse values indicate that successor process initiations occur at times 0, 3, 4 and 7. The pbs.level values indicate that the process is executing at these times also. Additionally, this example demonstrates why two separate but dependent periodic functions are required to specify the timing behavior of a control process. According to pbs.level, the process is executing in the interval [2, 4] but without pbs.pulse it would be impossible to determine that the process completed an execution at time 3 and was immediately reinitiated at time 4. Notice that a single time line showing level values can be augmented to depict both pulse and level, because of the constraint. A down-transition of level between times i and i + 1 indicates that pulse(i) = true. Similarly, a line segment “dangling” from a high level between times i and i + 1 indicates that pulse(i) = true. Pulse is false elsewhere.
Table 13: An Example Periodic_Boolean_Signal

<table>
<thead>
<tr>
<th>pbs.pulse</th>
<th>pbs.level</th>
<th>period of pbs = 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFTTTFFT</td>
<td>TFTTTFFT</td>
<td></td>
</tr>
</tbody>
</table>

There are seven basic operations on Periodic_Boolean_Signal variables. Each operation parallels the design of its counterpart in the Periodic_Function_Template. For this reason, the details of the specification of the PBS operations are not discussed. Instead, a more application-specific discussion is provided.

The operation Make_Simple_Periodic_Boolean_Signal produces two periodic functions that are related by the Is_PBS constraint. The operation thereby supports the construction of the simple periodic timing behavior of a sampling process in a control program. The Shift operation supports modification of the start time of a sampling process within its period in an obvious way. The Or operation is necessary for the combination of the timing behavior of control processes that together drive the execution of a direct successor process. Figure 6 illustrates an example of this; two sampling processes, A/D1 and A/D2 directly drive the execution of a compute process, A1. In order to calculate the timing behavior of A1 it is necessary to combine the timing behaviors of A/D1 and A/D2 through the Or operation.
Figure 6: A/D1 and A/D2 Drive the Execution of A1

The remaining operations, Smallest_Pulse_Period, Smallest_Level_Period, Is_True_Pulse, and Is_True_Level allow simple questions to be asked about the timing behavior of a control process; what is its smallest pulse or level period, is it initiating a successor at time i, and is it executing at time i. A simple example serves to illustrate the behavior of the operations in the PBS module. Table 14 traces the execution of the operations in the example.

Example

```
variable
    pbs1 : Periodic_Function
    pbs2 : Periodic_Function
    pulse_period : Integer
    level_period : Integer
begin
    Make_Simple_Pbs(pbs1, 5, 1) (* line 1 *)
    Make_Simple_Pbs(pbs2, 3, 1) (* line 2 *)
    Shift(pbs1, 1) (* line 3 *)
    Or(pbs1, pbs2) (* line 4 *)
    pulse_period := Smallest_Pulse_Period(pbs1) (* line 5 *)
    level_period := Smallest_Level_Period(pbs1) (* line 6 *)
end
```
Table 14: Example Operations in PBS

<table>
<thead>
<tr>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{pbs1.pulse} =$</td>
<td>$\text{pbs2.pulse} =$</td>
<td>$\text{pbs1.pulse} =$</td>
<td>$\text{pbs1.pulse} =$</td>
</tr>
<tr>
<td>T F F F F T F F F F F F F</td>
<td>T F F T F F T F F T F F F F</td>
<td>F T F F F T F F F F F F F</td>
<td>T T F T F F T F F T F F F</td>
</tr>
<tr>
<td>$\text{pbs1.level} =$</td>
<td>$\text{pbs2.level} =$</td>
<td>$\text{pbs1.level} =$</td>
<td>$\text{pbs1.level} =$</td>
</tr>
<tr>
<td>T F F F F T F F F F F F F</td>
<td>T F F T F F T F F T F F F F</td>
<td>F T F F F T F F F F F F F</td>
<td>T T F T F F T F F T F F F</td>
</tr>
<tr>
<td>period of pbs1 = 5</td>
<td>period of pbs2 = 3</td>
<td>period of pbs1 = 5</td>
<td>period of pbs1 = 15</td>
</tr>
</tbody>
</table>
2.2.2.2. Discussion of Design Decisions

A Periodic_Boolean_Signal abstractly captures the relevant properties of an asynchronous data-driven control process that are necessary (for analysis and scheduling purposes) to describe the timing behavior of such a process. At this point it is important to define what timing behavior is relevant for the determination of a schedule for a control process. For such a schedule, the periodic steady state timing behavior of a control process must be calculated. This steady state behavior can be defined by the times at which a process is executing within its period and the times at which the process initiates the execution of its successors. The specification of the PBS module abstracts away the passage of time and detailed operational characteristics, leading to a clear, concise and unambiguous specification for the potentially complex timing behavior of a control process.

A Periodic_Boolean_Signal models the execution of a control process via the abstract entities, pbs.pulse and pbs.level. Pbs.pulse and pbs.level are merely functions that map integers to booleans, but each abstractly captures a relevant timing behavior of a control process. Pbs.pulse represents the times at which a process initiates the execution of all its successor processes while pbs.level represents the times at which the process is executing. The mathematical invariant, Is_PBS, ensures that a Periodic_Boolean_Signal correctly mirrors the periodic timing behavior of a control process.

The operations encapsulated in the PBS module can be used to construct the periodic steady state timing behavior of a control process as follows. The Make_Simple_Periodic_Boolean_Signal operation captures the timing behavior of a sampling process with an integer-valued compute delay. The Shift operation allows the start time, or equivalently, the initiation time of a control process to be shifted relative to the beginning of its period. The Or operation allows timing patterns to be combined. The Is_True_Level, Is_True_Pulse and Smallest_Period operations allow for the selective testing of relevant timing characteristics of a control process. These operations provide a primitive and basic set of abstract methods to construct the timing behavior of sampling processes and to manipulate the timing behavior of all control processes.
2.2.3. **PBS-With_Delay_Template**

One important timing behavior of a control program has been neglected up to this point, that is, the timing behavior of computational processes. The steady state periodic timing behavior of a computational process is determined by the initiation pattern of its predecessor processes. As stated previously, in an asynchronous data-driven control program, sampling processes drive the execution of all compute processes to which they are directly connected, and these compute processes in turn drive the execution of all processes to which they are connected. This data-driven execution is carried out in a top down fashion based on the DAG representation of a control program. For scheduling purposes it is therefore necessary to determine the timing behavior of a compute process based on the asynchronous data-driven model of execution.

The next two subsections discuss the formal specification of the PBS-With_Delay_Template and the decisions made during its design. It is demonstrated that the PBS-With_Delay_Template provides a natural abstraction for the final aspect of the semantics of the asynchronous data-driven execution model.

2.2.3.1. **Specification**

The following is a RESOLVE specification of a module, PBS-With_Delay_Template. The PBS-With_Delay_Template module enhances (adds to) the Periodic_Boolean_Signal_Template, an operation that changes a Periodic_Boolean_Signal variable to reflect the computation delay in the asynchronous data-driven execution model of a non-sampling process. As an enhancement, the PBS-With_Delay_Template inherits the entire specification of the Periodic_Boolean_Signal_Template but it is possible to implement the module so the new operation, Delay, has direct access to the representation of the Periodic_Boolean_Signal data type. The interface section of the module specification is discussed below. Various examples and tables are used for illustration.
The following discussion serves to clarify the formal specification of the PBS_With_Delay_Template.

**Interface Section**

There is one (new) operation exported by the PBS_With_Delay_Template module.

- The Delay operation has two parameters, a Periodic_Boolean_Signal (pbs) to be altered and an integer \(d > 0\).

The Delay operation is quite complex and the following discussion clarifies its specification based on the premise that the operation is an application-specific abstraction for the asynchronous data-driven model of execution. Specification statements are numbered from line 1 to line 13 to aid in this discussion.
It is helpful to view the Delay operation as a "black box" representation of a compute process. The integer parameter d represents the computational delay of the compute process. The initial pbs parameter represents the combined timing behavior pattern of the predecessors of the compute process constructed by using Or to combine the behavior of all predecessors. The final altered pbs parameter represents the calculated timing behavior of the compute process based upon its delay and the initiation pattern of its predecessors. As a reminder, pbs.pulse represents the times at which a compute process initiates its successors and pbs.level represents the times at which a compute process is executing. The notation "#" is used to differentiate between the final value of a variable (pbs) and its initial value (#pbs).

At this point several examples illustrate how the Delay operation alters a Periodic_Boolean_Signal to capture the asynchronous data-driven execution model. For each example assume that the initial pbs parameter to the Delay operation has previously been constructed via the operations of the Periodic_Boolean_Signal_Template, as described in Section 2.2.2.1. A table is used to depict the results of each example. Row 1 of the table shows the mathematical model and time line representation of an initial pbs parameter for the invocation of Delay. Row 2 of the table shows the pbs parameter after the Delay operation has executed.

Table 15 demonstrates the effect of the invocation of Delay (pbs, 2). The period of the final pbs is not altered. This example illustrates the following:
- Whenever pbs.pulse(i) = true, then pbs.level(j) = true for all j in the interval [i − d + 1, i]. Pbs.pulse is true at times 4, 6 and 9, therefore pbs.level is true in the intervals [3, 4], [5, 6] and [8, 9].

This behavior is specified in lines 2 through 5 and line 7 of the Delay operation and captures the following timing requirement: If a compute process with delay d initiates its successors at time i, then it must be executing during the time interval [i − d+ 1 , i], whose length is equal to its compute delay.
- Whenever pbs.pulse(i) = true, then pbs.pulse(j) = false for all j in the interval [i − d + 1, i − 1]. Pbs.pulse is true at times 4, 6 and 9, therefore pbs.pulse is false in the intervals [3, 3], [5, 5] and [8, 8].

Lines 2 through 6 of Delay capture the above timing requirement and state that if a compute process, with a compute delay of d, initiates its successors at time i then it could not possibly initiate its successors during the interval, [i − d + 1, i − 1] because it
is currently executing during this time period. That is, initiations may occur no less than d time units apart.

• Whenever \( \text{pbs.pulse}(i) = \text{true} \) and \( \text{pbs.pulse}(i - d) = \text{true} \), then \( \#\text{pbs.pulse}(j) = \text{true} \) for some \( j \) in the interval \([i - 2d + 1, i - d]\). Pbs.pulse is true at times 6 and 4, so \( \#\text{pbs.pulse} \) must be true at some time in the interval \([3, 4]\), and it is true at time 4.

Line 2 and lines 8 through 11 specify that if new data becomes available on one or more input arcs while a compute process is executing, it is reinitiated (only once) after its current execution completes. If \( \text{pbs.pulse}(i) = \text{true} \) then the process is executing in the interval \([i - d + 1, i]\). If \( \text{pbs.pulse}(i - d) = \text{true} \) then the process is also executing in the previous interval \([i - 2d + 1, i - d]\). Together, these imply that the process must have been initiated at some point during its previous execution, therefore \( \#\text{pbs.pulse}(j) = \text{true} \) for some integer \( j \) in the interval \([i - 2d + 1, i - d]\).

• Whenever \( \text{pbs.pulse}(i) = \text{true} \) and \( \text{pbs.pulse}(i - d) = \text{false} \), then
\[ \#\text{pbs.pulse}(i - d) = \text{true} \] Pbs.pulse is true at times 4 and 9 but false at times 2 and 7 and \( \#\text{pbs.pulse} \) is true at times 2 and 7.

Line 2 and lines 12 through 13 capture the above timing constraint: Whenever a process is initiated and it is not currently executing, the process is immediately initiated. If \( \text{pbs.pulse}(i) = \text{true} \) then the process is executing in the interval \([i - d + 1, i]\). If \( \text{pbs.pulse}(i - d) = \text{false} \) then the process cannot be executing in the interval \([i - 2d + 1, i - d]\). Together, these imply that the process must have been initiated at time \( i - d \), therefore \( \#\text{pbs.pulse}(i - d) = \text{true} \).
Table 15: Effect of Delay (pbs, 2)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pbs.pulse</strong></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td><strong>pbs.level</strong></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td><strong>period of pbs</strong></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 16 depicts an example in which the period of the altered pbs decreases due to the invocation of Delay (pbs, 5). The resulting period can be no less than the compute delay of the process (d parameter to Delay). The period of pbs becomes 5 due to the relatively large number of initiations that occur in the initial pbs parameter and due to the length of the d parameter. The compute process becomes “saturated” because it is being reinitiated at a rate at least equal to its compute delay. This example also demonstrates how the Delay operation captures the following timing requirement of the execution model. Multiple data changes along a single input arc, which occur while the process is executing, cause only one new initiation of the process (this behavior is specified in line 2 and lines 8 through 11 of the Delay operation). Notice that #pbs.pulse is true at times 4 and 7, indicating that the process is initiated twice during the execution interval [3, 7], but only one new initiation occurs at time 8.
Table 16: Effect of Delay (pbs, 5)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbs.pulse</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>pbs.level</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>period of pbs</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbs.pulse</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>pbs.level</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>period of pbs</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 17 depicts a final example in which the period of the altered pbs increases due to the invocation of Delay (pbs, 3). In the worst case, this period can be extended to a length equal to (d-1) * the period of the initial pbs parameter. This occurs when the number of initiations in the initial pbs parameter is large enough so that the execution of the compute process can not fit in the original period length but not so large as to saturate its execution. The fact that pbs.level(1) = false indicates that the calculated timing behavior of the process does not saturate its altered period length of 10.
2.2.3.2. Discussion of Design Decisions

The design of the formal specification of the PBS_With_Delay_Template is based on the asynchronous data-driven execution model. The specification is clearly specific to the problem at hand and has no general application outside of the one intended. No argument can therefore be made as to the reusability of the PBS_With_Delay_Template but it is possible to argue that the specification, although challenging, is a correct and unambiguous abstraction for the asynchronous data-driven execution model.

The specification of the PBS_With_Delay_Template captures the following timing requirements of a compute process. A process, once initiated, must execute for a length of time equal to its compute delay. A process cannot execute at a rate faster than the reciprocal of its compute delay. A process that is initiated while executing can initiate its successors immediately, only after completing its current execution. If a process is not currently executing, it is immediately initiated whenever it receives new data on a input arc. If multiple initiations occur while a process is executing it is reinitiated only once.

Table 17: Effect of Delay (pbs, 3)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pbs.pulse</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td><strong>pbs.level</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>period of pbs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>pbs.pulse</strong></td>
<td></td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td><strong>pbs.level</strong></td>
<td></td>
<td></td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>period of pbs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
after its current execution completes. Each of these timing requirements is captured in the altered values of the Periodic_Boolean_Signal (pbs) parameter of the Delay operation.

2.3 Summary

Section 2.1 discussed simulation techniques to determine the periodic steady-state timing behavior of a control program that executes in an asynchronous data-driven manner. These simulation techniques tend to be computationally expensive and representationally complex. For this reason a direct special-purpose algorithmic approach is employed to determine the steady-state timing behavior of a control program.

The algorithmic approach revolves around the design of a layered software system. The formal specification of this system was presented in Section 2.2. Section 2.2.1 described the specification of a module, Periodic_Function_Template, that is used as an explanatory tool for the other modules. Sections 2.2.2 and 2.2.3 described the specification of the software modules, Periodic_Boolean_Signal_Template and PBS_With_Delay_Template. These modules are the backbone of the algorithmic approach and together, provide a natural abstraction for the asynchronous data-driven model of execution. The data type, Periodic_Boolean_Signal, exported by the Periodic_Boolean_Signal_Template, captures the relevant timing behaviors necessary for the problem of scheduling hard real-time control software. The operations provided by the modules support construction of the periodic timing behavior of both sampling and non-sampling control processes in a straightforward manner.

Although the specifications described in this chapter are concise and unambiguous, this is only half the battle. Clearly, an efficient implementation of these modules is necessary to complete the problem of determining the complex periodic timing behavior of a control program. Chapter 3 presents an efficient implementation of the modules discussed here.
CHAPTER III

Implementation of Modules for Direct Special-Purpose Algorithmic Approach

There are two basic techniques for scheduling real-time software: dynamic scheduling and static scheduling. Dynamic scheduling algorithms attempt to compute a feasible schedule for a software implementation during run-time. These algorithms must make scheduling decisions based on their knowledge of the timing properties of processes currently in the system. Because a run-time scheduler has no information concerning those processes that have not yet arrived in the system it is impossible for the scheduler to guarantee that software timing requirements will be satisfied [Mok 84]. For this reason, dynamic algorithms are not appropriate for scheduling hard real-time control software — the focus of this dissertation. Static scheduling algorithms attempt to compute a feasible schedule for a software implementation prior to run-time. Although this requires advance knowledge of the major timing properties of processes in the system, it is the only useful method to predict the complex behavior of hard real-time software [Xu 91b].

Static scheduling algorithms are quite diverse, for example [Garey 76], [Lawler 81], [Shepard 91]. Each defines the scheduling problem using different characteristics [Xu 91b] such as process type (static, periodic, or asynchronous), release times, deadlines, number of processors, etc. Those algorithms that consider both periodic and asynchronous processes [Mok 84], [Xu 90], [Xu 91a] are of interest for the problem of determining a schedule for an asynchronous data-driven control program. Unfortunately, these algorithms convert an asynchronous data-driven process into a simple periodic process that conforms to the synchronous clock-driven execution model.
described in Chapter 1. For this reason, a new direct special-purpose algorithm is required to solve the scheduling problem for the asynchronous data-driven model of execution.

It is crucial that any direct special-purpose algorithmic approach to determine the steady-state timing behavior of a control program be as computationally tractable as possible. This is especially important if the control program contains a large number of computational processes or if system-defined parameters are subject to modification and timing behaviors must be recalculated. Furthermore, any direct approach is attractive only if its computational expense is less than some other method, such as simulation.

Section 3.1 discusses the worst-case complexity of direct simulation to determine the steady state timing behavior of a control program. This complexity analysis is used as a basis of comparison against the computational complexity of the implementation of the direct special-purpose algorithmic approach described in Section 3.2. This discussion focuses on the implementation of the Periodic_Boolean_Signal_Template and the PBS_With_Delay_Template described in Chapter 2. Section 3.3 summarizes the ideas presented in this chapter.

3.1. Cost of Simulation

Just how expensive is it to determine the complex steady state timing behavior of an asynchronous data-driven control program through simulation? As previously stated, the steady-state timing pattern of a control process is based on its compute delay, and on the periods and initiation patterns of its predecessors. In order to determine the steady-state timing pattern of a control program, via simulation, each process in the program is allowed to execute in a "self-timed" manner according to the asynchronous data-driven model of execution. When this occurs, an initial transient period develops in which the control process is not in a state of equilibrium. After the transient period, the control process settles into a periodic steady-state. Cost of simulation therefore depends on the length of the transient period, the length of the steady-state period, and the method employed to recognize that a periodic execution pattern has occurred.

A control process executing in a self-timed manner may be in one of $d + 1$ ($d$ is the process compute delay) possible states at each time $i$. Thus a control program with $N$ processes may be in one of $S = \prod (d_j + 1)$ possible states at time $i$ where $j$ runs over $1 \leq \lambda
j ≤ N. This implies that in the worst case the lengths of the transient period and the steady-state period for such a program is equal to the size of its state space. In order to determine whether a control program has settled into its equilibrium phase requires that the simulator recognize that the current state has been repeated. In other words, if the current state in the simulation matches a previous state generated by the simulation then the control program has stabilized. A direct simulation could maintain an associative table with S (as defined above) entries. A search of the table is then required for each simulation step to determine whether the current state matches a previous one. The combinatorially large worst-case complexity of direct simulation is (exponential in N) terribly costly and for this reason a more tractable special-purpose scheduling algorithm is of interest.

3.2. Implementation of a Layered Software System for a Direct Algorithmic Approach

The next five subsections describe the implementation of the software system for a special-purpose algorithm to determine the periodic timing behavior of a control program. Section 3.2.1 describes the program representation of a Periodic_Boolean_Signal, the abstract data type exported by the Periodic_Boolean_Signal_Template. The implementation of the operations of the Periodic_Boolean_Signal_Template are described in Sections 3.2.2 though 3.2.5. An analysis of the computational complexity of each operation is provided in its corresponding section. When necessary, pseudo-code is used to demonstrate the major steps performed in an operation. The Delay operation, exported in addition to the others by the PBS_With_Delay_Template, is presented in Section 3.2.6. Various examples are employed to illustrate the implementation of the Delay operation and a complexity analysis is provided.

3.2.1. Program Representation of Periodic_Boolean_Signal

Periodic_Boolean_Signal is an abstract data type exported by the module, Periodic_Boolean_Signal_Template. In order to implement a Periodic_Boolean_Signal, a program representation must be chosen with an eye toward efficient implementation of
the operations that manipulate the type. There are several intuitively plausible representations that could be chosen for the implementation of a Periodic_Boolean_Signal. Each representation produces a different run-time complexity for the operations in the encapsulating module. The goal of the representation chosen for this dissertation is to provide the most efficient method to compute the complex steady state timing behavior of a control program. No claim is made as to the relative efficiency of this representation for a general class of applications.

A good program representation of a Periodic_Boolean_Signal, referred to as PBS, consists of a record with six fields: an integer-valued pulse_period, an integer-valued level_period, an integer-valued period, an integer-valued shift, an integer-valued number_of_pulses, and a field that holds a List of integer-valued pairs (pulse_level_list). The following Modula-2 type declaration defines a PBS program data type.

```modula2
TYPE PBS = RECORD
  pulse_period : INTEGER;
  level_period : INTEGER;
  period : INTEGER;
  shift : INTEGER;
  number_of_pulses : INTEGER;
  pulse_level_list: List;
END;
```

If a PBS abstractly models the steady state timing behavior of a control process, then the pulse_period field corresponds to its smallest steady-state initiation pattern, the level_period field corresponds to its smallest steady-state execution pattern, the period field represents the smallest period for combined initiation and execution patterns, the shift field represents its phase, the number_of_pulses field represents the number of initiations in each period, and the pulse_level_list represents its periodic timing behavior. The pulse_period, level_period, period, shift, and number_of_pulses fields are maintained so that certain operations in the Periodic_Boolean_Signal_Template execute more efficiently, that is, in constant time. This is discussed further in the presentation of the implementation of the relevant operations. The List field requires further discussion at this point.

The List used in the program representation of PBS is commonly referred to as a one-way list (see Appendix A for a Modula-2 specification and implementation of a one-
way list abstract data type). Conceptually, a one-way list is a string of elements where
the front of the list is the left-most item in the string and the end of the list is the right-
most item in the string. A positional fence (represented as “I” in the following examples)
that divides the list into left and right substrings is used to control access to the items in
the list. Insertions into the list and deletions from the list are performed to the right of
the fence. The fence may be advanced one position to the right in the list, and reset to
the front or to the end of the list. At any time only the item just to the right of the fence
may be accessed or modified.

For example, a one-way list of integers might be depicted as < 1, 5, 8, 10 | 3, 4 >
where the fence divides the list into a left string consisting of 1, 5, 8, and 10 and a right
string consisting of 3 and 4.

The pulse_level_list field of a PBS is a string of integer-valued pairs (pulse, level),
which is kept ordered on the pulse element of a pair. All pulse elements in a List have
unique values. A PBS list is depicted using the following notation:
< (4,2) | (6,3) (10,1) >. In this case the fence is to the right of the item (4,2).

The correspondence between the program representation pbs_rep and
Periodic_Boolean_Signal pbs is as follows. If there is an entry (i, j) in
pbs_rep.pulse_level_list, then:
(1) pbs.pulse(i + pbs_rep.shift) = true and
(2) pbs.level(k + pbs_rep.shift) = true for all k in [i - j + 1, i].
If pbs.pulse or pbs.level is not known to be true because of rule (1) or (2), for some list
entry (i, j) satisfying 0 ≤ i < pbs_rep.period, then it is false. The values of pbs.pulse
and pbs.level outside the interval [0, pbs_rep.period) are determined by the fact that
both are periodic functions with period pbs_rep.period.

This representation is quite compact since an entry in the List is required only for
each integer within the period for which pbs.pulse is true. Several examples illustrate
the correspondence between the formal specification of a Periodic_Boolean_Signal and
its program representation. Row 1 of Table 18 shows the mathematical model of a
Periodic_Boolean_Signal variable, pbs. Row 2 of the table depicts the variable's
corresponding program representation. The fence for the List field of pbs is shown at
the front of the string, but this is not necessary.
Table 18: Program Representation of PBS (Example 1)

<table>
<thead>
<tr>
<th>(1)</th>
<th>(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbs.pulse =</td>
<td>pbs.pulse_period = 10</td>
</tr>
<tr>
<td>F F F T F F F T T F</td>
<td>pbs.level_period = 5</td>
</tr>
<tr>
<td>pbs.level =</td>
<td>pbs.period = 10</td>
</tr>
<tr>
<td>T T T T F T T T F</td>
<td>pbs.shift = 0</td>
</tr>
<tr>
<td>period of pbs = 10</td>
<td>pbs.number_of_pulses = 3</td>
</tr>
<tr>
<td></td>
<td>pbs.pulse_level_list =</td>
</tr>
<tr>
<td></td>
<td>&lt;1 (3,4) (7,3) (8,1)&gt;</td>
</tr>
</tbody>
</table>

Table 19 illustrates a final example of the correspondence between the mathematical model and the program representation. Notice that the shift field of the program representation is equal to 1. This implies that each pulse value in pbs pulse_level_list is shifted or incremented by 1. The shift field does not affect the level values in pbs.pulse_level_list, which are relative to the pulse times.
Table 19: Program Representation of PBS (Example 2)

<table>
<thead>
<tr>
<th>(1)</th>
<th>(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbs.pulse = F F F F T F T F T</td>
<td>pbs.pulse_period = 10</td>
</tr>
<tr>
<td>pbs.level = F T T T T F T F T</td>
<td>pbs.level_period = 10</td>
</tr>
<tr>
<td>period of pbs = 10</td>
<td>pbs.period = 10</td>
</tr>
<tr>
<td></td>
<td>pbs.shift = 1</td>
</tr>
<tr>
<td></td>
<td>pbs.number_of_pulses = 3</td>
</tr>
<tr>
<td></td>
<td>pbs.pulse_level_list = &lt; 1 (4,5) (6,1) (8,1) &gt;</td>
</tr>
</tbody>
</table>

3.2.2. Implementation of Make_Simple_Periodic_Boolean_Signal Operation

The Make_Simple_Periodic_Boolean_Signal operation produces a PBS variable with simple periodic behavior. Table 20 illustrates the program representation and time line of pbs for the invocation of Make_Simple_Periodic_Boolean_Signal (pbs, period, duration), with period = 10 and duration = 3. This program representation captures a control process that initiates its successors at time 2 and is executing in the interval [0, 2], and has a period of 10.
Example

BEGIN
    Make_Simple_Periodic_Boolean_Signal(pbs, 10, 3);
END;

Table 20: Make_Simple_Periodic_Boolean_Signal Operation

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbs.pulse_period</td>
<td>10</td>
</tr>
<tr>
<td>pbs.level_period</td>
<td>10</td>
</tr>
<tr>
<td>pbs.period</td>
<td>10</td>
</tr>
<tr>
<td>pbs.shift</td>
<td>0</td>
</tr>
<tr>
<td>pbs.number_of_pulses</td>
<td>1</td>
</tr>
<tr>
<td>pbs.pulse_level_list</td>
<td>&lt;1(2,3)&gt;</td>
</tr>
</tbody>
</table>

The actions performed by this operation are quite straightforward, as seen in the following pseudo-code.

PROCEDURE Make_Simple_Periodic_Boolean_Signal(VAR pbs : Periodic_Boolean_Signal; period, duration : INTEGER)
BEGIN
    pbs.pulse_period := period;
    pbs.level_period := period;
    pbs.period := period;
    pbs.shift := 0;
    pbs.number_of_pulses := 1;
    make pbs.pulse_level_list an empty list;
    insert (duration - 1, duration) in pbs.pulse_level_list;
END;
Each of the above actions is performed in constant time. Therefore the run-time complexity of the operation, Make_Simple_Periodic_Boolean_Signal, is $O(1)$.

3.2.3. Implementation of Shift Operation

The operation, Shift (pbs, shift) shifts the pulse values contained in pbs.pulse_level_list. The Shift operation does not affect the level values contained in PBS. If Shift is implemented so that it accesses each pulse element in the List then the operation has a run-time complexity of $O(\text{pbs.number_of_pulses})$. This linear run-time is unnecessary if an integer-valued shift field is maintained for a PBS variable. The implementation of the Shift operation is quite simple in this case and requires updating the shift field so that pbs.shift := pbs.shift + shift. This action can be performed in constant time, resulting in $O(1)$ run-time for the Shift operation. The shift field can then be used by operations that must access a (pulse, level) pair in a List by simply adding the field to the pulse value. Table 21 illustrates the program representation of pbs after Shift (pbs, 5) is invoked in the following example.

Example

```
BEGIN
    Make_Simple_Periodic_Boolean_Signal(pbs, 10, 3);
    Shift(pbs, 5);
END;
```
3.2.4. Implementation of Or Operation

The Or operation combines the periodic behavior of two PBS variables through the logical or operation. As previously discussed in Chapter 2, this operation can affect the period of the resulting PBS variable in several ways. The period may be unchanged or it may shortened or lengthened based on the characteristics of the combined behaviors. The smallest period of a PBS variable and its pulse and level components are explicitly maintained in its representation record. Because of this, the Or operation must ensure that these fields are updated appropriately. The Or operation performs two major functions: A merge of two Lists, pbs1.pulse_level_list and pbs2.pulse_level_list, and determination of the smallest period of the final merged list. Pseudo-code for the Or operation is as follows (note that LCM (p1, p2) stands for the least common multiple of p1 and p2).

<table>
<thead>
<tr>
<th>Table 21: Shift Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>pbs.pulse_period = 10</td>
</tr>
<tr>
<td>pbs.level_period = 10</td>
</tr>
<tr>
<td>pbs.period = 10</td>
</tr>
<tr>
<td>pbs.shift = 5</td>
</tr>
<tr>
<td>pbs.number_of_pulses = 1</td>
</tr>
<tr>
<td>pbs.pulse_level_list =</td>
</tr>
<tr>
<td>&lt; 1 (2,3) &gt;</td>
</tr>
</tbody>
</table>
PROCEDURE Or (VAR pbs1 : Periodic_Boolean_Signal;
               VAR pbs2 : Periodic_Boolean_Signal)
BEGIN
  merge lists pbs1.pulse_level_list and pbs2.pulse_level_list
  in pbs1.pulse_level_list;
  pbs1.pulse_period := smallest pulse pattern in
  pbs1.pulse_level_list;
  pbs1.level_period := smallest level pattern in
  pbs1.pulse_level_list;
  pbs1.period := LCM(pbs1.pulse_period, pbs1.level_period);
END;

The implementation of the above pseudo-code is described in the following two
subsections.

There is an additional point to discuss concerning the Or operation. According to the
specification of the operation:

\[
\text{ensures "for all } i : \text{ integer } \\
\quad \text{(pbs1.pulse}(i) = (\#\text{pbs1.pulse}(i) \text{ or } \text{pbs2.pulse}(i)) \\
\quad \text{and } \\
\quad \text{pbs1.level}(i) = (\#\text{pbs1.level}(i) \text{ or } \text{pbs2.level}(i))"}
\]

both the pulse value of pbs1 and pbs2 and the level value of pbs1 and pbs2 are logically
or'ed. The implementation of the Or operation, using the described program
representation of a PBS variable, produces a list of uniquely-valued pulse entries for
pbs1.pulse_level_list. This is not the case for the level entries in pbs1.pulse_level_list
where multiple program representations are possible for the same mathematical model.
For example, if pbs1.pulse_level_list is equal to either:
\[
< I (4,2) (5,5) > \text{ or } < I (4,3) (5,5) >
\]
then both are valid program representations for the mathematical model:

\[
\begin{align*}
\text{pbs1.pulse} &= F F F F T T \\
\text{pbs1.level} &= F T T T T T \\
\text{pbs1.period} &= 6.
\end{align*}
\]
This causes no trouble in the implementation of the operations of
Periodic_Boolean_Signal_Template.
3.2.4.1. List Merge

Table 22 illustrates the combination of two PBS variables, pbs1 and pbs2, that exhibit simple periodic behavior. Assume that each variable has been constructed via a previous application of an appropriate Make_Simple_Periodic_Boolean_Signal and Shift operation. Rows 1 and 2 depict the program representation of pbs1 and pbs2 before application of the Or operation. Row 3 depicts the value of pbs1 after Or executes. Note that pbs.pulse_period and pbs.level_period are equal to pbs.period and are therefore omitted from the program representation.

Table 22: Or Operation (Example 1)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example 1 illustrates how two PBS variables with simple periodic behavior are combined to produce a variable with complex periodic behavior. Row 3 of the table indicates that the period of pbs1 is not modified. This is because no smaller repetition
pattern can be found in the list $< 1 (3,2) (7,2) >$. Notice that the shift field of pbs1 is reset to 0. Before merging the lists, the appropriate shift value is added to each pulse element in the lists of pbs1 and pbs2 as shown below. The run-time complexity of this step is $O(pbs1\text{.number\_of\_pulses} + pbs2\text{.number\_of\_pulses})$.

PROCEDURE Reset_Shift_Field( VAR pbs1, pbs2 : Periodic_Boolean_Signal)

BEGIN
reset fence to front of pbs1.pulse_level_list;
reset fence to front of pbs2.pulse_level_list;
WHILE not at end of pbs1.pulse_level_list DO
  get (pulse1, level1) from pbs1.pulse_level_list;
  insert (pulse1+pbs1.shift, level1) in
  pbs1.pulse_level_list;
  advance fence in pbs1.pulse_level_list;
END;
WHILE not at end of pbs2.pulse_level_list DO
  get (pulse2, level2) from pbs2.pulse_level_list;
  insert (pulse2+pbs2.shift, level2) in
  pbs2.pulse_level_list;
  advance fence in pbs2.pulse_level_list;
END;
END;

The next step performed by the Or operation is a merge of pbs1.pulse_level_list and pbs2.pulse_level_list as shown in the following pseudo-code.
PROCEDURE List Merge (VAR pbs1 : Periodic_Boolean_Signal;
          VAR pbs2 : Periodic_Boolean_Signal);

BEGIN
  reset fence to front of pbs1.pulse_level_list;
  reset fence to front of pbs2.pulse_level_list;
  WHILE not at end of either list DO
    get (pulse1, level1) from pbs1.pulse_level_list;
    get (pulse2, level2) from pbs2.pulse_level_list;
    IF pulse2 < pulse1 THEN
      insert (pulse2, level2) in pbs1.pulse_level_list;
      advance fence in pbs2.pulse_level_list;
    ELSIF pulse2 > pulse1 THEN
      advance fence in pbs1.pulse_level_list;
    ELSIF level2 > level1 THEN
      update pbs1.pulse_level_list with (pulse2, level2);
      advance fence in pbs1.pulse_level_list;
      advance fence in pbs2.pulse_level_list;
    END;
  END;
  WHILE not at end of pbs2.pulse_level_list DO
    get (pulse2, level2) from pbs2.pulse_level_list;
    insert (pulse2, level2) in pbs1.pulse_level_list;
    advance fence in pbs1.pulse_level_list;
    advance fence in pbs2.pulse_level_list;
  END;
END;

Each (pulse, level) pair in the List of pbs2 is inserted (ordered on pulse) into the List of pbs1. If the two lists contain a (pulse, level) pair, where pulse values are equal but level values are unequal, then the pair with the larger level value is ends up in pbs1.pulse_level_list. The run-time complexity of the list merge is O(pbs1.number_of_pulses + pbs2.number_of_pulses).

Table 23 depicts the combination of two complex Periodic_Boolean_Signal variables, pbs1 and pbs2, with unequal periods. Assume that pbs1 and pbs2, as shown in row 1 and 2 of the table, have been previously constructed via multiple applications of the operations Make_Simple_Periodic_Boolean_Signal, Shift and Or; and that Reset_Shift_Field has been run on both.
In example 2, pbs1 and pbs2 have periods of 5 and 10 respectively. When this occurs the Or operation determines the LCM of pbs1.period and pbs2.period. If necessary, the entries in the pulse_level_list of each variable are extended to reflect the combined period. The lists of pbs1 and pbs2 are then merged.

The run-time complexity of the list extension step is based on the following. Let pbs1.period = P1 and pbs2.period = P2. Now the worst case run-time complexity of LCM (using Euclid’s algorithm) is O(log P1 + log P2)). Let LCM (P1, P2) = lcm, and let pbs1.number_of_pulses = N1 and pbs2.number_of_pulses = N2. The number of steps required to extend pbs1.pulse_level_list and pbs2.pulse_level_list is O(((lcm / P1) * N1)) + ((lcm / P2) * N2))). This is also the number of steps required to merge the extended lists. In the worst case the run-time complexity of the list extension is O(lcm).

Examples 1 and 2 demonstrate how the timing behavior of two PBS variables is combined. In the first example the period of the resulting PBS is not modified. This is
due to the fact that both variables have equal periods and their initiation patterns are out of phase. In the second example the period of the resulting PBS is extended to the least common multiple of the periods of the initial PBS variables. This occurs because the variables have unequal periods and their initiation patterns are out of phase. But what happens if the initiation patterns of two PBS variables are in phase? When this occurs, a smaller repetition pattern of (pulse, level) pairs can be found hidden in the resulting merged lists. The method to determine the smallest pulse pattern and the smallest level pattern in a pulse_level_list is described in the next subsection.

3.2.4.2. Smallest Pulse Pattern

Table 24 depicts a PBS variable that has been constructed via multiple applications of the Make_Simple_Periodic_Boolean_Signal, Shift and Or operations. Notice that the list < I (4,2) (7,2) (9,2) (14,2) (17,2) (19,2) > with period 20 contains a smaller repetition pattern of pulse elements, (4, 7, 9), with a period of 10.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first step to determine the smallest pulse pattern is a linear traversal of pbs.pulse_level_list in which an array is constructed that holds the distances between adjacent pulse values. This step is depicted in the following pseudo-code.
PROCEDURE Construct_Pulse_Distance_Array(VAR A : ARRAY OF INTEGER;
                                           pbs : Periodic_Boolean_Signal)

VAR i, current_pulse : INTEGER;
BEGIN
    reset fence to front of pbs.pulse_level_list;
    get (pulse, level) from pbs.pulse_level_list;
    advance fence in pbs.pulse_level_list;
    current_pulse := pulse;
    i := 2;
    WHILE not at end of pbs.pulse_level_list DO
        get (pulse, level) from pbs.pulse_level_list;
        A[i] := pulse - current_pulse;
        current_pulse := pulse;
        advance fence in pbs.pulse_level_list;
        i := i + 1;
    END;
    reset fence to front of pbs.pulse_level_list;
    get (pulse, level) from pbs.pulse_level_list;
END;

For the list < (4,2) (7,2) (9,2) (14,2) (17,2) (19,2) >, the array A looks like [ 5, 3, 2, 5, 3, 2 ] after the above pseudo-code step executes. The run-time complexity required for the construction of A is O(pbs.number_of_pulses).

In order to find the smallest repeated pulse pattern within A, a sliding window is used to compare array values. Table 25 illustrates the steps performed by the sliding window approach. The window is shown as a gray box positioned over array elements. Each row of the table depicts the result of one iteration of the approach.
Table 25: Sliding_Window_Approach (Example 1)

<table>
<thead>
<tr>
<th></th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>window_size = 1</td>
<td>$i = 1$</td>
<td>$i = 1$</td>
<td>$i = 3$</td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>j</td>
<td>j</td>
</tr>
<tr>
<td>j</td>
<td>k</td>
<td>k</td>
<td>k</td>
</tr>
<tr>
<td>j</td>
<td>k</td>
<td>k</td>
<td>k</td>
</tr>
<tr>
<td>k</td>
<td>k</td>
<td>k</td>
<td>k</td>
</tr>
</tbody>
</table>

*Row 1 of the table depicts the initial state of the sliding window approach for this example. The index j indicates the left-side of the window. Indices i and k indicate which array elements are currently being compared. The window_size is set to 1 and indices i, j and k are set appropriately. Array elements A[j] through A[j + window_size - 1] are compared to array elements A[i] through A[i + window_size - 1]. As long as all elements are equal, the window slides to the right by one unit (j is replaced by j + 1), k is set to j, i is incremented by 1, and the window_size is left unchanged. If some element A[k] within the window is unequal to its corresponding element in A then i is set to 1, window_size to k, and j to k + 1. In other words, the window slides to the right and the window size increases appropriately (while the new window_size is not a factor of pbs.number_of_pulses, window_size is again increased by 1 and the window slides 1 unit to the right). Row 2 and 3 of the table illustrate this occurrence. The above steps repeat until either k > pbs.number_of_pulses or window_size is > (pbs.number_of_pulses) div 2. If all window elements are equal to*
A[i] through A[i + window_size - 1], on the last iteration of the sliding window approach, then window_size indicates the smallest pulse pattern in pbs.pulse_level_list. Since this repetition pattern contains the distance between all pulse entries within their period, merely summing the entries gives the new pbs.pulse_period. For the example pulse pattern of (5, 3, 2) this gives a pulse_period of 10.

Table 26 illustrates the sliding window approach when no smaller pulse pattern exists within pbs.pulse_level_list. Assume that pbs.pulse_level_list contains the (pulse, level) entries: <(1,1) (6,1) (8,1) (13,1) (14,1) (15,1) (17,1) (19,1)>, with period = 20. The A array therefore looks like: [2, 5, 2, 5, 1, 1, 2, 2].

Table 26: Sliding_Window_Approach (Example 2)

<table>
<thead>
<tr>
<th>(1) window_size = 1</th>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 1</td>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>j = 2</td>
<td>5</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>k = 2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(2) window_size = 2</th>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 2</td>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>j = 3</td>
<td>5</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>k = 4</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(3) window_size = 2</th>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 3</td>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>j = 5</td>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>k = 5</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Row 1 of the table depicts the initial state of the sliding window approach for this example. Row 2 and 3, respectively, show the result of the second and third iteration of the approach. The algorithm terminates after iteration 3. Because the comparison
between A[k] and A[i] fails, the new window_size is set to 5 but this is greater than pbs.number_of_pulses div 2, i.e. 5 > 8 div 2. For this example, pbs.pulse_period therefore equals pbs.period.

Since one comparison is performed for each array element, in the worst case the sliding window approach to determine the smallest pulse pattern in pbs.pulse_level_list requires \(O(pbs.number\_of\_pulses)\) steps to execute.

3.2.4.3. Smallest Level Pattern

The technique used to find the smallest level pattern within pbs.pulse_level_list is similar to the approach discussed in the previous subsection. There are some minor differences due to the fact that multiple program representations are possible for the same mathematical model of pbs.level. This possibility was previously discussed in Section 3.2.4. Several examples help to clarify this point.

Table 24 is redisplayed here as Table 27 in order to supply a concrete example for the following discussion. Assume that the smallest pulse_period has been previously determined, as depicted in the table. Notice that a smaller level pattern of \(<1 (4,2) (7,2) (9,2) >\), with period = 10, exists in pbs.pulse_level_list.

<table>
<thead>
<tr>
<th>pbs.pulse_period = 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbs.level_period = ?</td>
</tr>
<tr>
<td>pbs.period = 20</td>
</tr>
<tr>
<td>pbs.shift = 0</td>
</tr>
<tr>
<td>pbs.number_of_pulses = 6</td>
</tr>
<tr>
<td>pbs.pulse_level_list =</td>
</tr>
<tr>
<td>&lt;1 (4,2) (7,2) (9,2) &gt;</td>
</tr>
</tbody>
</table>

**Table 27 : Smallest Level Pattern**
In order to determine the smallest pattern within `pbs.pulse_level_list`, level entries that are adjacent or that overlap are combined in an array of integer pairs. As an example, if `pbs.pulse_level_list` is `< (2,2) (4,3) >` then the two entries are combined as `[ (4,4) ]` or if `pbs.pulse_level_list` is `< (2,1) (3,1) (5,2) >` then the entries are combined as `[ (3,2) (5,2) ]`. The pseudo-code for this combination step is as follows.

```
PROCEDURE Combine_Level_Entries(VAR C : ARRAY OF INTEGER_PAIRS
VAR pbs : Periodic_Boolean_Signal)

VAR i, array_size, last_item : INTEGER;
BEGIN
  reset fence to front of pbs.pulse_level_list;
i := 1;
  WHILE not at end of pbs.pulse_level_list DO
    get (pulse, level) from pbs.pulse_level_list;
    C[i].pulse := pulse;
    C[i].level := level;
i := i + 1;
    advance fence in pbs.pulse_level_list;
  END;
  array_size := i - 1;
  last_item := i - 1;
  FOR i := last_item DOWNTO 2 DO
    IF C[i].pulse - C[i - 1].pulse ≤ C[i].level THEN
      C[i - 1].pulse := pulse;
      C[i - 1].level := C[i].pulse -
      (C[i - 1].pulse - C[i - 1].level + 1);
      array_size := array_size - 1;
    END;
  END;
END;
```

The while loop in the above pseudo-code merely places each `(pulse, level)` entry of `pbs.pulse_level_list` into a random access array `C`. The for loop combines adjacent or overlapping level entries in the array. The loop moves through the array from last position to first position. This is necessary so that no holes occur when combining level entries. As an example, let `C = [ (2,2) (4,3) (8,3) (9,9) ]`. If the `C` entries are combined from first to last then the resulting array is `[ (4,2) (9,9) ]`, which is incorrect. The correct combination of level values, where `C = [ (9,9) ]`, is produced by the above pseudo-code. For the example `pbs` in Table 27, `C = [ (4,2) (9,4) (14,2) (19,4) ]` after the code executes. The run-time complexity for this combination step is
After level entries are combined, the next step is to calculate the distance between adjacent pulse entries in the array C. The technique used here is identical to the one described in the previous subsection. The pulse entries in the array C are scanned and the difference between adjacent entries is placed in a distance array A along with their corresponding level entries. In this case the distance array contains pairs of integers. For the example pbs, \( A = [(5,2) (5,4) (5,2) (5,4)] \). The run-time complexity for construction of A is \( O(pbs.\text{number}\_\text{of}\_\text{pulses}) \).

Once the distance array is constructed a sliding window approach, similar to the one described in the previous subsection, can be used to find the smallest level pattern in A. The only difference between the two approaches is that in this case pairs of integers are compared. As an example, the sliding window approach determines that the pbs in Table 27 has a repetition pattern of \([(5,2) (5,4)]\). The run-time analysis for the sliding window algorithm is \( O(pbs.\text{number}\_\text{of}\_\text{pulses}) \).

Finally, the length of the smallest level period is determined based on the repetition pattern of integer pairs. The first entry of a pair gives the distance between relevant pulse entries. By merely summing their values, the level_period of the smallest repetition pattern can be determined. For the pattern \([(5,2) (5,4)]\), the level_period is therefore 10.

3.2.5. Implementation of Smallest_Pulse_Period, Smallest_Level_Period, Is_True_Pulse and Is_True_Level Operations

The operation, Smallest_Pulse_Period (pbs) returns the smallest pulse period in pbs.pulse_level_list. This information is explicitly maintained in the pulse_period field of a pbs variable and therefore the run-time of this operation is \( O(1) \).

Smallest_Level_Period (pbs) returns the level_period field of a pbs variable. This operation also has a constant run-time.

The operation, Is_True_Pulse (pbs, i), returns true if and only if there exists a pulse entry in an unshifted pbs.pulse_level_list that is equal to i modulo pbs.pulse_period (or equivalently, pbs.shift + pulse = i modulo pbs.pulse_period). This implies that in the worst case a linear search of the List must be performed, resulting in a
O(pbs1.number_of_pulses) run-time complexity for the operation. For the implementation of the special-purpose scheduling algorithm however, each (pulse, level) pair in the List of a PBS variable is accessed linearly from first to last. So if the fence controlling access to a List entry is not reset to the front of the List (after each access), Is_True_Pulse executes in O(1) time.

The operation Is_True_Level (pbs, i) returns true if and only if there exists a (pulse, level) entry in an unshifted pbs.pulse_level_list such that i modulo pbs.level_period is in the interval [pulse − level + 1, pulse] (or equivalently, [pulse + pbs.shift − level + 1, pulse + pbs.shift]). The worst case run-time of the Is_True_Level operation is O(pbs.number_of_pulses). This is due to the fact that level entries do not exhibit a unique program representation and it may be necessary to scan the entire list to determine if pbs.level (i) = true. This operation is not used in the special-purpose algorithmic approach, though, so its run-time is of no real concern.

3.2.6. Implementation of the Delay Operation

The operation Delay (pbs, d) calculates the periodic timing behavior of a non-sampling process based on the initiation pattern of its predecessors (pbs) and its compute delay (d). This operation need access only the pulse elements in pbs.pulse_level_list. Like the Or operation, Delay can affect the resulting period of pbs in several ways: pbs.period is unchanged, pbs.period = d, or in the worst case pbs.period = (d − 1) * #pbs.period. Refer to Tables 15, 16 and 17 in Chapter 2 for an example of each of these possible period modifications.

Table 17 is redisplayed here as Table 28 in order to supply a concrete example for the explanation of the implementation of Delay. The table depicts the result of invoking Delay (pbs, d) with d = 3. Row 1 shows the program representation of pbs before Delay executes and row 2 depicts the final value of pbs after execution. The table provides a time line representation of pbs in order to clarify the discussion.
Table 28: Delay Operation

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>pbs.period = 5</td>
<td></td>
</tr>
<tr>
<td>pbs.shift = 0</td>
<td></td>
</tr>
<tr>
<td>pbs.number_of_pulses = 2</td>
<td></td>
</tr>
<tr>
<td>pbs.pulse_level_list =</td>
<td>(1,1) (2,2)</td>
</tr>
</tbody>
</table>

| (2) |   |
| pbs.period = 10 |   |
| pbs.shift = 0 |   |
| pbs.number_of_pulses = 3 |   |
| pbs.pulse_level_list = | (0,3) (4,3) (7,3) |

The implementation of the Delay operation uses a sliding window approach that is similar to the one used to determine pbs.pulse_period and pbs.level_period. Each iteration of this approach is depicted in a row of Table 29 (using the example pbs of Table 28). The right side of each row represents the following: Asterisks are used to depict the times at which a pulse occurs in pbs.pulse_level_list, the time units in pbs.pulse_period are depicted as a sequence of integers 0, 1, ..., pbs.pulse_period-1, and the result of each iteration of the algorithm is shown under this sequence.

Initially the window size is set to d and placed over the first d time units in pbs.pulse_period. This window is shown as a gray box. The variable start holds pulse + 1, where pulse is the first pulse entry in pbs.pulse_level_list.
Table 29: Sliding Window Approach (Example 1)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>window_size = 3</td>
<td>i = 0</td>
<td>V[i] = (i+d) mod pbs.period</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>window_size = 3</td>
<td>i = 1</td>
<td>V[i] = (i+d) mod pbs.period</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>window_size = 3</td>
<td>i = 2</td>
<td>V[i] = (i+d) mod pbs.period</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td>window_size = 3</td>
<td>i = 3</td>
<td>V[i] = start</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>window_size = 3</td>
<td>i = 4</td>
<td>V[i] = (i+d) mod pbs.period</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The algorithm scans each time unit in pbs.period to determine where process initiations occur. The question that is asked is: If an initiation starts at the present time unit then when will the next initiation start? The start time of initiations depends on two
cases. The first case occurs when pulse values are present within the window and the second occurs when no pulse values are present in the window. Row 1 depicts the first iteration of the algorithm in which time unit 0 is processed. Notice that the window contains a pulse value at times 1 and 2. If an initiation occurs at time i and executes for d time units, then the next initiation will start at time \((i + d) \mod \text{pbs.pulse\_period}\), if any pulses exist within the window. The algorithm stores the calculated start time in an integer-valued array, V, thus \(V[0] = 3\). After each iteration of the algorithm, the window slides one time unit to the right with period wrap-around. The results shown in rows 2 and 3 are similar. Row 4 depicts the case where no pulse values exist within the window for time unit 3. When this happens the start of the next initiation depends on the closest pulse value to the right of the window area (with period wrap-around). For time unit 3 this happens to be the pulse value at time 1. Thus the next initiation begins at time start and \(V[4] = 2\). Row 5 depicts results similar to those shown in rows 1 through 3. After each time unit in \(\text{pbs.pulse\_period}\) is processed, the array V holds the start times of the initiations for the process. The run-time complexity for construction of V is linear in the \(\text{pulse\_period}\) of pbs, that is, \(O(\text{pbs.pulse\_period})\).

Table 30 illustrates an additional example of the workings of the sliding window approach for the Delay operation. Assume in this case that the value of \(d = 4\), \(\text{pbs.period} = 10\) and pulses occur at times 0, 1, 9.
Table 30: Sliding Window Approach (Example 2)

<table>
<thead>
<tr>
<th>(1) window size = 4</th>
<th>start</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 0</td>
<td></td>
</tr>
<tr>
<td>start = 1</td>
<td></td>
</tr>
<tr>
<td>V[i] = (i+d) mod pbs.period</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(2) window size = 4</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 1</td>
<td></td>
</tr>
<tr>
<td>V[i] = (i+d) mod pbs.period</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td></td>
<td>4 5</td>
</tr>
</tbody>
</table>

| (3, 4, 5, 6) window size = | *   |
| i = 2,3,4,5              |     |
| V[i] = (next_pulse+1) mod pbs.period |     |
|                     | 0 1 2 3 4 5 6 7 8 9 |
|                     | 4 5 0 0 0 0       |

| (7, 8, 9, 10) window size = | *   |
| i = 6,7,8,9               |     |
| V[i] = (i+d) mod pbs.period |     |
|                     | 0 1 2 3 4 5 6 7 8 9 |
|                     | 4 5 0 0 0 0 1 2 3   |

Row 1 and 2 of the table depict the result of the first two iterations of the sliding window approach. For both iterations a pulse exists within the window so V[i] := (i + d) mod pbs.pulse_period. Row 3 depicts iterations 3 through 6. In this case, as time units 2 through 5 are processed, a pulse is not found within the window therefore the next initiation cannot start until after the pulse at time 9. That is, V[i] = (9 + 1) mod pbs.pulse_period for all i in [2, 5]. Row 4 of the table depicts iterations 7 through 10.
As the window slides to the right there is always a pulse contained within it thus $V[i]$ is set to $(i + d) \mod \text{pbs.pulse_period}$ for all $i$ in $[6, 9]$.

The $V$ array and the start value of the sliding window approach are then used in the next step of the Delay algorithm. This step actually determines the periodic timing behavior for the process by finding the cycle that defines this behavior. The cycle can be found by traversing the $V$ entries from start to $V[\text{start}]$ to $V[V[\text{start}]]$ and so on until an entry is reached that has been visited before. For the example shown in Table 29, the $V$ array looks like $[3, 4, 0, 2, 2]$ with start $= 2$. The cyclic graph captured in the $V$ array is shown in Figure 7. The cycle begins at start $= 2$ and consists of the entries $2 \rightarrow 0 \rightarrow 3$.

![Figure 7: Cyclic Graph in V Array (Example 1)](image)

Figure 8 depicts the cyclic graph found in the $V$ array of Table 30. In this case $V = [4, 5, 0, 0, 0, 0, 1, 2, 3]$ with start $= 1$. By following the $V$ elements beginning at start $= 1$, the cycle is entered at 0 and consists of two entries $0 \rightarrow 4$. In the worst case the run time for finding a cycle in the $V$ array is $O(\text{pbs.pulse_period})$. 
Once the cycle of start times is found, each entry must be processed in the following manner. The value \( d - 1 \) is added to each cycle entry. If a cycle entry is less than its predecessor then pbs.pulse_period is added to that entry and to each of its successors. For the cycle 2 \( \rightarrow \) 0 \( \rightarrow \) 3, of Figure 7, this results in a new cycle of pulse elements 4 \( \rightarrow \) 7 \( \rightarrow \) 10. Notice that this cycle indicates that the process executes for a total of 9 time units in the intervals [2, 4], [5,7] and [8, 10]. This cycle is actually equivalent to the cycle 0 \( \rightarrow \) 4 \( \rightarrow \) 7 since the last entry in the cycle equals pbs.pulse_period (if the last cycle entry equals pbs.pulse_period then a period wrap around is performed) and results in the pulse_level_list \(<\{(0,3),(4,3),(7,3)\}>\), with a pulse_period of 10 as depicted in row 2 of Table 26. Since the cycle of initiation start times can contain at most pbs.pulse_period entries, the run-time complexity to process the cycle is \( \mathcal{O}(\text{pbs.pulse_period}) \). This is also the overall run-time of the Delay operation.

One final comment needs to be made concerning the Delay operation. Since Delay modifies the (pulse, level) pattern of pbs.pulse_level_list, the algorithm to determine pbs.level_period must be applied to the pbs variable constructed by the Delay operation. This algorithm was presented in Section 3.2.4.2. Application of this algorithm does not affect the overall run-time complexity of the Delay operation.
3.3. Summary

Section 3.1 discussed the exponential run-time complexity of determining the steady-state timing behavior of a control program via simulation. This unattractive run-time led to the design of a special-purpose algorithm to tackle this scheduling problem.

Section 3.2 described the implementation of the modules that comprise the software system used by the algorithmic approach for scheduling. The implementation of Periodic_Boolean_Signal (PBS), the data type exported by the modules Periodic_Boolean_Signal_Template and PBS_With_Delay_Template was discussed in Section 3.2.1. It provides a “good” program representation that contains pertinent information for the efficient implementation of the operations exported by these modules. In particular, the shift, pulse_period and level_period fields of a PBS are maintained so that the operations, Shift, Smallest_Pulse_Period and Smallest_Level_Period execute in constant time. The number_of_pulses field is used to speed-up the implementation of the Or, Is_True_Pulse and Delay operations. The next five subsections presented the implementation of the operations exported by the modules Periodic_Boolean_Signal_Template and PBS_With_Delay_Template. The run-time complexity of each operation was provided. Algorithms were presented for the more complex operations; Or and Delay. For general applications, the Is_True_Pulse operation has a worst-case run-time that is linear in the pulse_period of a PBS variable. Because of the use of the operation in the direct algorithmic approach for scheduling, this operation actually runs in constant time. The run-time complexity of the Is_True_Level operation is linear in the number_of_pulses contained in a PBS variable. The Or operation combines the periodic behavior of two PBS variables and its run-time depends on the lcm of the variable’s periods. Finally, the Delay operation has a run-time that is linear in the pulse_period of a PBS variable. The algorithms presented in this chapter lead to very efficient implementations of the exported operations.

The direct special-purpose algorithmic scheduling approach uses the implementation of the modules discussed in this chapter in order to efficiently determine a static schedule for an asynchronous data-driven control program. Chapter 4 describes the Static Analyzer/Scheduler tool which is the embodiment of this direct special-purpose algorithm.
CHAPTER IV

Static Analyzer/Scheduler Tool

The Static Analyzer/Scheduler tool constructs a static multiprocessor schedule for an asynchronous data-driven control program. This chapter describes the steps performed by the tool during the schedule construction phase. Section 4.1 provides a review of the information required by the tool and describes its basic operation using the example control program of Chapter 1. Section 4.2 discusses methods for choosing the optimal phase shift for sampling processes. Choice of phase shifts has a direct impact on the relative merit of a particular schedule by affecting such measures as the number of processors necessary to support schedule implementation and end-to-end delay. A method to assign control processes to processors is described in Section 4.3. Section 4.4 defines end-to-end delay and describes the method used by the Static Analyzer/Scheduler tool to determine this measure. Section 4.5 describes an efficient run-time implementation for a static multiprocessor schedule that includes processes with complex periodic behavior. Section 4.6 provides a summary of the ideas discussed in this chapter.

4.1. Constructing Operation Sequences

During the analysis phase, a control engineer translates the block diagram of a control program into its DAG representation. Each vertex in the DAG corresponds to a process in the control program and each edge in the DAG indicates a control or data-dependency between individual processes. Section 1.1 describes this translation process as depicted in Figures 1 and 2.
Each process in a control program is partitioned into two subsets: the set SP which contains sampling processes and the set NSP which contains non-sampling processes. A process $p_i$ in SP is defined by the tuple $(m_i, d_i)$ where $m_i$ is the maximum simple period of $p_i$ and $d_i$ is the compute delay of $p_i$. Each process $p_i$ in NSP has a fixed delay of $d_i$. After analysis of the control program, the control engineer supplies the tuple $(m_i, d_i)$ for each sampling process and $d_i$ for each non-sampling process. These timing requirements constitute a set of basic system parameters for a control program executing in an asynchronous data-driven manner.

The Static Analyzer/Scheduler takes as input the DAG representation of a control program and the system parameters supplied by the control engineer. As an example, Figure 9 depicts the DAG representation of the control program presented in Chapter 1. Table 31 illustrates a possible set of basic system parameters for the same control program.

![Figure 9: DAG Representation of a Digital Control Program](image)
Given the above input information, the Static Analyzer/Scheduler generates a sequence of operations to construct the steady-state timing behavior of each process in the control program. These operations, exported by the Periodic_Boolean_Signal_Template and the PBS_With_Delay_Template, are described in Chapters 2 and 3. The following sequence of operations is produced for the example input of Figure 9 and Table 31.

A Copy operation is included in the following operation sequence in order to preserve the timing behavior of each control process in the program. This operation was not described in the specification of Periodic_Boolean_Signal_Template because it can be implemented as a secondary or layered operation. The run-time complexity of this secondary Copy operation is O(pbs.period). For efficiency reasons it is preferable to implement the Copy operation as a primary operation so that it has direct access to the program representation of a Periodic_Boolean_Signal variable. In this case the Copy operation has a run-time complexity of O(pbs.number_of_pulses). The following is a formal RESOLVE specification of the Copy operation.

\begin{verbatim}
operation Copy
parameters
  preserves pbs1 : Periodic_Boolean_Signal
  produces pbs2 : Periodic_Boolean_Signal
ensures "pbs2 = pbs1"
\end{verbatim}
Sequence of Operations (Example 1)

BEGIN

Make_Simple_Periodic_Boolean_Signal(pbs1, 15, 1); (* line 1 *)
Make_Simple_Periodic_Boolean_Signal(pbs2, 10, 1); (* line 2 *)
Make_Simple_Periodic_Boolean_Signal(pbs3, 15, 1); (* line 3 *)
Copy(pbs1, pbsAD1); (* line 4 *)
Copy(pbs2, pbsAD2); (* line 5 *)
Copy(pbs3, pbsAD3); (* line 6 *)
Or(pbs1, pbs2); (* line 7 *)
Delay(pbs1, 1); (* line 8 *)
Copy(pbs1, pbsA1); (* line 9 *)
Delay(pbs1, 4); (* line 10 *)
Copy(pbs1, pbsC1); (* line 11 *)
Or(pbs1, pbs3); (* line 12 *)
Delay(pbs1, 1); (* line 13 *)
Copy(pbs1, pbsA2); (* line 14 *)
Delay(pbs1, 4); (* line 15 *)
Copy(pbs1, pbsC2); (* line 16 *)
Delay(pbs1, 1); (* line 17 *)
Copy(pbs1, pbsDA1); (* line 18 *)

END;

The Static Analyzer/Scheduler first creates a Periodic_Boolean_Signal variable for each sampling process pi in the control program using the tuple (m_i, d_i) that defines its timing criteria. This step is depicted above in lines 1 through 3 for the samplers A/D1, A/D2 and A/D3. Based on the topological order of vertices in the DAG representation of the control program, the tool then generates a sequence of Or and Delay operations. Whenever two or more vertices in the DAG are direct predecessors of a vertex then their timing behaviors must be combined via the Or operation. As an example, in Figure 9 vertex A/D1 and vertex A/D2 are predecessors of A1, so pbs1 and pbs2 are combined via the Or operation as in line 7 of the example. Their combined timing behavior, pbs1, is then fed into the Delay operation of line 8. Table 32 displays the time line schedules for the processes in the DAG of Figure 9 as produced by the Static Analyzer/Scheduler.
Table 32: Schedules Produced by Static Analyzer/Scheduler
(Example 1)

Sweeping a vertical line across the schedule in Table 32, from time 0 to time 30, indicates that four processors are necessary for the implementation of this multiprocessor schedule. The maximum end-to-end delay for this schedule is 12 time units, which is optimum.

It is often possible to decrease the required number of processors by phase-shifting the timing behavior of the sampling processes in the control program. Phase-shifting also directly impacts the end-to-end delay exhibited by the schedule. The following section describes the optimization problem of phase-shifting the timing behavior of sampling processes and the effect this has on the relative merits of a schedule.
4.2 Phase-Shifting the Timing Behavior of Sampling Processes

As stated previously, phase-shifting the timing behavior of sampling processes affects the measures used to judge the merit of a schedule, that is, end-to-end delay and number of processors required for schedule implementation. The following sequence of operations, produced by the Static Analyzer/Scheduler, illustrates this effect. The sampling processes A/D1, A/D2 and A/D3 are shifted 0, 5 and 10 time units to the right, respectively.

**Sequence of Operations (Example 2)**

```
BEGIN
  Make_Simple_Periodic_Boolean_Signal(pbs1, 15, 1); (* line 1 *)
  Make_Simple_Periodic_Boolean_Signal(pbs2, 10, 1); (* line 2 *)
  Make_Simple_Periodic_Boolean_Signal(pbs3, 15, 1); (* line 3 *)
  Shift(pbs2, 5); (* line 4 *)
  Shift(pbs3, 10); (* line 5 *)
  Copy(pbs1, pbsAD1); (* line 6 *)
  Copy(pbs2, pbsAD2); (* line 7 *)
  Copy(pbs3, pbsAD3); (* line 8 *)
  Or(pbs1, pbs2); (* line 9 *)
  Delay(pbs1, 1); (* line 10 *)
  Copy(pbs1, pbsAl); (* line 11 *)
  Delay(pbs1, 4); (* line 12 *)
  Copy(pbs1, pbsC1); (* line 13 *)
  Or(pbs1, pbs3); (* line 14 *)
  Delay(pbs1, 1); (* line 15 *)
  Copy(pbs1, pbsA2); (* line 16 *)
  Delay(pbs1, 4); (* line 17 *)
  Copy(pbs1, pbsC2); (* line 18 *)
  Delay(pbs1, 1); (* line 19 *)
  Copy(pbs1, pbsDA1); (* line 20 *)
END;
```

Table 33 depicts the process schedules produced by the Static Analyzer/Scheduler for example 2.
Sweeping a vertical line across the time-line in Table 33, from time 0 to time 30, indicates that three processors are required for this schedule implementation. The end-to-end delay is still 12.

It is important to note that another choice for phase-shifts could result in a longer end-to-end delay and a greater number of processors required to implement the schedule. Table 34 illustrates this possibility. In this case the sampling processes A/D1, A/D2 and A/D3 are phase-shifted 0, 1 and 2 time units to the right, respectively. Four processors are required to implement the schedule for this example. The maximum end-to-end delay is 16 time units. This can be found by following the path from the first initiation of A/D2 at time 1, to the resulting initiation of D/A1 at time 15. Notice that process initiations begin to stack-up for C1 and C2 in rows 5 and 7 of the table. This is a clear indication that the schedules for these processes are starting to saturate and implies that in the worst case their initiations could be delayed by as much as twice their compute delays.
The optimization of phase-shifts in order to minimize number of processors and end-to-end delay is a completely unstructured problem. It is not clear how any individual phase-shift will affect the time line schedules of the processes in a control program. Nor is it clear how the measures of end-to-end delay and number of processors interact. Typically, phase-shifts that cause a process schedule to saturate also have a deleterious effect on the measures that reflect the merit of a control program schedule, but this is not always the case.

Choosing the optimal set of phase-shifts for sampling processes can be tackled using a brute-force technique that exhaustively enumerates every possible phase-shift within the period of a process. Although this is not a terribly attractive solution in general, the problem is finite and quite tractable in the case where the periods of sampling processes are small and there are not many sampling processes.
Stochastic methods that randomly select phase-shifts and attempt to localize an optimal solution, such as Monte Carlo techniques [Mikhilov92], simulated annealing [Otten89] or genetic algorithms, might offer a viable alternative solution. This is a subject for future research.

4.3. Process-to-Processor Assignment

Once the Static Analyzer/Scheduler determines the schedule for each process in a control program, the tool then performs process-to-processor assignment. Obviously, the focus here is to ensure that there is no execution conflict for processes assigned to a particular processor and that process precedence and timing constraints are not violated. Processes are assigned to processors according to the following constraints: a process execution interval cannot be preempted but process migration is supported, that is, a process may execute on different processors for different execution intervals. The general process-to-processor assignment problem has been shown to be NP-complete [Garey 79] and much research has been dedicated to bin-packing approximation techniques for its solution; examples include [Coffman84], [Murgolo 87], [Friesen 86], etc. These approximation techniques are not directly applicable to the problem of process-to-processor assignment that must be tackled by the Static Analyzer/Scheduler. Therefore, a more straightforward greedy algorithm is employed to perform processor assignment. This algorithm is efficient and produces optimal results because process migration makes the problem easy to solve.

Given a control program with a set of N processes \( \{p_1, p_2, ..., p_N\} \) and the calculated steady-state timing behavior of each process \( p_i \), maintained in the variable \( pbs_i \), the Static Analyzer/Scheduler performs processor assignment as follows.

1. It is assumed that there is a bank of N homogeneous shared-memory processors \( \{P_1, P_2, ..., P_N\} \) available for processor assignment. (Some subset of these will be used.)
2. The period of the schedule for each processor (\( P_{\text{schedule}} \)) is set to the least common multiple of \( pbs_1.\text{period}, ..., pbs_N.\text{period} \).
3. A schedule \( S_i \) is constructed for each processor \( P_i \) required for schedule implementation. \( S_i \) consists of a sequence of triples, \( < \text{process_id, start, execution}1, ..., \text{process_id, start, execution}m > \) where \( \text{process_id} \) is an integer-valued process identifier, \( \text{start} \) is an integer-valued...
start time, and execution is an integer-valued execution length. Note that start times in $S_3$ have unique values. As an example, $S_3 = <(1, 5, 2) (2, 7, 1)>$ indicates that processes $p_1$ and $p_2$ are assigned to processor $P_3$ for the execution intervals [5,6] and [7,7], respectively.

Construction of a processor schedule is performed by scanning the pulse values contained in the pbs.pulse_level_list of each process in the control program. This requires testing the result of Is_True_Pulse (pbs, i) for each integer i in the interval [0, P_schedule - 1]. As an example, let pbs; represent the timing behavior of $p_i$ and let the compute delay of $p_i = d$. If Is_True_Pulse(pbs; j) = true and the interval $[j - d + 1, j]$ is free in the schedule of $P_1$ then $p_i$ is assigned to $P_1$ by inserting the triple $(i, j - d +1, d)$ at the appropriate position in $S_1$. If the interval $[j - d + 1, j]$ in $S_1$ is currently assigned to another process then an attempt is made to assign $p_i$ to processor $P_2$. In the case where the execution interval of a process conflicts with all currently assigned processor schedules then a new processor $P_j$ is made available from the processor bank and the execution interval is inserted in $S_j$. Table 35 depicts the processor schedules produced by the Static Analyzer/Scheduler for the control program schedules of Table 33. Three processors are employed in this process-to-processor assignment, which is optimum.

Table 35: Processor Schedules

<table>
<thead>
<tr>
<th>Processor</th>
<th>Processor Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>&lt; (A/D1,0,1) (A1,1,1) (A/D2,5,1) (A1,6,1) (A/D3,10,1) (A2,11,1) (A/D1,15,1) (A1,16,1) (C1,17,4) (A2,21,1) (A/D3,25,1) (A1,26,1) &gt;</td>
</tr>
<tr>
<td>$P_2$</td>
<td>&lt; (A2,1,1) (C1,2,4) (A2,6,1) (C1,7,4) (D/A1,11,1) (A/D2,15,1) (D/A1,16,1) (A/D2,25,1) (A2,26,1) (C1,27,4)&gt;</td>
</tr>
<tr>
<td>$P_3$</td>
<td>&lt; (D/A1,1,1) (C2,2,4) (D/A1,6,1) (C2,7,4) (C2,12,4) (C2,22,4) (D/A1,26,1) (C2,27,4)&gt;</td>
</tr>
</tbody>
</table>
The greedy algorithm discussed above has a run-time complexity of \(O(N^2 \cdot P_{\text{schedule}})\). It makes a localized decision concerning which process to assign to a particular processor. This decision is based only on whether a process execution interval is “free” within a processor’s currently constructed schedule. A localized decision of this type can guarantee that a minimal number of processors is used for schedule implementation as long as process migration is allowed.

4.4. Measuring End-To-End Delay

One of the measures for determining the relative merit of a schedule is the amount of end-to-end delay that is incurred. End-to-end delay is the amount of time between the start of sampling of an input to the control program and the time that sample is first reflected in an output of the program. The problem of measuring end-to-end delay is trivial for a control program with a single sampling process and a DAG representation that contains a single simple path between processes. The problem becomes more difficult in the general case when a control program has multiple samplers and numerous possible paths in its DAG representation. In fact, it is not clear what the term end-to-end delay should mean for this general class of control programs.

The end-to-end delay for a control program schedule is defined here as the maximum delay incurred along all paths in a DAG that begin with the start of a sampling process and terminate when the sampled input is reflected in an output of the control program. For the control program represented in Figure 9, the DAG indicates that one path exists for each of the sampling processes, A/D1, A/D2 and A/D3. Let \(path_i\) equal the number of paths for sampling process \(p_i\). Table 33 indicates that A/D1 samples data at times 0 and 15, A/D2 samples data at times 5, 15 and 25, and A/D3 samples at times 10 and 25. Let \(samples_i\) equal the number of initiations performed by a sampling process \(p_i\) within its schedule period. The number of possible paths over which end-to-end delay must be measured is the sum of \(path_i \cdot samples_i\) for each sampling process \(p_i\). Therefore for the control program schedule of Table 33 there are seven possible paths over which end-to-end delay must be measured. Each row of Table 36 illustrates one of these paths. The integer value depicted in parentheses within a path indicates the initiation time of its corresponding process. If the first initiation in a path is less than the last initiation, then the path’s end-to-end delay equals last initiation – first initiation + 1. If the first initiation is greater than the last, then the path’s end-to-end delay equals (first initiation – period of
The end-to-end delay for a control program schedule is the maximum over all path end-to-end delays. As can be seen in Table 36, the end-to-end delay for the example control program schedule is 12 time units.

Table 36: Paths Over Which End-To-End Delay is Measured

<table>
<thead>
<tr>
<th>Spi</th>
<th>Path Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A/D1</td>
<td>A/D1(1)-&gt;A1(2)-&gt;C1(6)-&gt;A2(7)-&gt;C2(11)-&gt;D/A1(12)</td>
</tr>
<tr>
<td>time = 0</td>
<td></td>
</tr>
<tr>
<td>delay = 12</td>
<td></td>
</tr>
<tr>
<td>A/D1</td>
<td>A/D1(16)-&gt;A1(17)-&gt;C1(21)-&gt;A2(22)-&gt;C2(26)-&gt;D/A1(27)</td>
</tr>
<tr>
<td>time = 15</td>
<td></td>
</tr>
<tr>
<td>delay = 12</td>
<td></td>
</tr>
<tr>
<td>time = 5</td>
<td></td>
</tr>
<tr>
<td>delay = 12</td>
<td></td>
</tr>
<tr>
<td>time = 15</td>
<td></td>
</tr>
<tr>
<td>delay = 12</td>
<td></td>
</tr>
<tr>
<td>time = 25</td>
<td></td>
</tr>
<tr>
<td>delay = 12</td>
<td></td>
</tr>
<tr>
<td>time = 10</td>
<td></td>
</tr>
<tr>
<td>delay = 7</td>
<td></td>
</tr>
<tr>
<td>A/D3</td>
<td>A/D3(26)-&gt;A2(27)-&gt;C2(1)-&gt;D/A1(2)</td>
</tr>
<tr>
<td>time = 25</td>
<td></td>
</tr>
<tr>
<td>delay = 7</td>
<td></td>
</tr>
</tbody>
</table>
The Static Analyzer/Scheduler finds end-to-end delay paths by scanning the timing behavior of appropriate processes based on the precedence relationships defined in the DAG representation of a control program. As an example, A/D1 first initiates its predecessors at time $i = 0$. The only successor of A/D1 is A1 which has a compute delay $d = 1$. The first occurrence of $\text{Is\_True\_Pulse}(\text{pbsA1}, j) = \text{true}$ where $j - d + 1 > i$ indicates the $j$th initiation of A1 that is a direct result of the initiation by A/D1 at time $i$ (or at least reflects the data of that initiation). Thus, as shown in row 1 of Table 36, A/D1 initiates A1 at time 0 and A1 in turn initiates its successors at time 1. This pulse-scanning process is continued down a sampling path until a sink process in the DAG is reached.

### 4.5 Implementation of a Processor Scheduler

The process scheduler that implements the control program schedule constructed by the Static Analyzer/Scheduler requires only a few simple mechanisms based on the following assumptions. Each processor used for schedule implementation has its own scheduler program. Each processor has access to a shared real-time clock (or processor clocks are synchronized). Processors are homogeneous and share memory. Each processor supports a simple counter mechanism such that counters may be initialized and reset.

Given a processor schedule, as constructed by the Static Analyzer/Scheduler, a corresponding resident processor schedule can be simply constructed so that the scheduler performs a constant amount of work at each clock tick. A resident schedule consists of a one way list of \((\text{process\_id}, \text{count})\) pairs. A process\_id entry represents a process identifier, while count is the distance between adjacent process start times in a processor schedule. As an example, row 1 of Table 35 is reproduced here as row 1 of Table 37. This illustrates the processor schedule of $P_1$. Row 2 of Table 37 depicts the resident schedule for processor $P_1$. A resident schedule for each processor is constructed from its corresponding processor schedule and made available to its scheduler.
Table 37: Resident Schedule Construction

<table>
<thead>
<tr>
<th>Processor</th>
<th>Processor Schedule</th>
<th>Resident Schedule</th>
</tr>
</thead>
</table>

A processor scheduler executes as follows. Initially, each scheduler sets an integer count to the start time of the first process initiation within its processor schedule. Each scheduler has a tick counter that is initialized to 0. The fence of each resident schedule list is set to the front of the list. At each clock tick, tick counter is incremented and the scheduler checks to see if tick counter = count. If so, the (process_id, count) entry to the right of the fence in the resident schedule is accessed. The process indicated by process_id is scheduled, tick counter is reset to 0, the scheduler count is reset to count, and the fence of the resident schedule is advanced one position to the right. If the fence is at the end of the list when tick counter = count then it is reset to the front of the list. In this manner the scheduler simply sequences through its resident schedule list indefinitely. With a programmable real-time clock a similar but even easier method can be used to delay scheduling exactly the proper length of time.

4.6. Summary

This chapter described the tasks performed by the Static Analyzer/Scheduler tool. A high level discussion of the implementation of the tool was provided.

Section 4.1 described how the tool constructs a schedule that reflects the steady-state timing behavior for an asynchronous data-driven control program based upon the information supplied by a control engineer. Operations from the
Periodic_Boolean_Signal_Template and the PBS_With_Delay_Template are employed in this schedule construction phase.

Section 4.2 described the optimization problem for phase-shifting the timing behavior of sampling processes. This problem is completely unstructured and it is unclear how phase shifts affect the measures of merit for a schedule. This suggests that Monte Carlo techniques are appropriate heuristics for selecting phase shifts.

The method of assigning processes to processors was discussed in Section 4.3. The Static Analyzer/Scheduler uses a greedy strategy that is efficient and that determines an optimal process-to-processor assignment.

Section 4.4 defined the term end-to-end delay for a general class of control programs in the asynchronous data-driven execution model. The method used by the Static Analyzer/Scheduler to determine end-to-end delay was described.

A simple and efficient scheduler implementation was presented in Section 4.5. This implementation requires only simple mechanisms and ensures the important property that the scheduler performs a constant amount of work at each clock tick.
CHAPTER V

Conclusion

Section 5.1 summarizes the work conducted for this research and presents conclusions drawn from it. Contributions to the field are described in Section 5.2. Future work and open questions are discussed in Section 5.3.

5.1. Summary and Conclusions

The first parts of this work were a semi-formal definition of the synchronous clock-driven execution model and the asynchronous data-driven execution model for a class of hard real-time control software. A layered software system was designed that captures the steady-state timing characteristics of an asynchronous data-driven control program and supports the efficient construction of these timing behaviors. Finally, a Static Analyzer/Scheduler tool was designed to efficiently construct a feasible static multiprocessor schedule for an asynchronous data-driven control program.

5.1.1. Definition of Synchronous and Asynchronous Execution Models

Section 1.1 provided a semi-formal definition of the synchronous clock-driven execution model for a class of hard real-time control software. As a brief restatement, the timing constraints of the synchronous execution model require that the periods of all sampling processes in a control program be synchronized, so that for any pair of processes one period is an integer multiple of the other. Synchronization of sampling periods ensures that each non-sampling process in a control program executes in a
simple periodic manner, that is, it executes exactly once within its period and with the same phase for each period.

An important aspect of the definition of the synchronous clock-driven model is that it pinpoints the optimizations that must be performed in order to determine an optimum static schedule for a control program. At the front end, process periods must be synchronized. The choice of synchronized periods directly affects the relative merit of a schedule’s implementation in terms of its processor resource requirements and its end-to-end response time. This is a very difficult optimization problem, as it is not clear which choice of synchronized periods will eventually produce an optimum schedule and there is no apparent "principle of optimality" to limit one’s choices. At the back end, process start times must be phase-shifted in order to minimize the number of processors required for schedule implementation. Unfortunately, phase-shifting the start time of a process incurs additional end-to-end delay because some processes must wait to execute even after their input data becomes available. The tradeoff between minimizing processor requirements and minimizing end-to-end response times makes the problem of optimizing phase-shifts quite difficult for any particular mix of objectives. A final optimization of process-to-processor assignment must be performed in order to minimize processor resource requirements for the schedule of a synchronous clock-driven control program. This optimization problem is directly affected by the front end and back end optimizations discussed above.

A semi-formal definition of the asynchronous data-driven execution model was presented in Section 1.2. Briefly, sampling processes in the asynchronous data-driven model execute in a simple periodic manner and drive the execution of all non-sampling processes in the control program in a top-down manner. The execution of a non-sampling process is self-timed and data-driven and therefore may be complex periodic, that is, a non-sampling process may execute at multiple unequally-spaced intervals within each period.

An important aspect of the definition of the asynchronous data-driven execution model is the removal of the front end optimization problem of synchronizing process periods that is required by the synchronous clock-driven execution model. This problem is ignored because non-sampling processes are allowed to execute freely in a complex periodic manner. Furthermore, if sampling periods must be synchronized then data may have to be sampled at a faster rate than required by the timing constraints of the control program. This does not occur in the asynchronous data-driven execution model because
each sampler executes according to its predefined minimum sampling frequency. The problems of optimizing phase-shifts and processor assignment remain for the asynchronous data-driven execution model. It is interesting to note, however, that phase-shifting start times in the asynchronous model does not automatically incur the additional end-to-end delay seen in the synchronous execution model.

Chapter 1 provided a comparison of the synchronous and asynchronous execution models. This comparison supports the first point of the thesis, namely that practical analysis of the steady-state timing behavior of a control program does not require synchronized execution of the individual processes within the system. This point is in direct conflict with current scheduling practices in the real-time systems community. Those scheduling algorithms that address the issue of asynchronous data-driven processes translate their execution so they behave in a simple periodic manner as defined by the synchronous clock-driven execution model. Point one of the thesis states that this is not strictly necessary and that the asynchronous data-driven execution model is a viable alternative model for the problem of scheduling a class of hard real-time control software. In some cases the flexibility of this methodology may produce a better schedule, with fewer processors and shorter delays, than the synchronous clock-driven execution model.

5.1.2. Specification of Software System

Chapter 2 introduced the specification of a layered software system that captures the relevant timing semantics (for analysis and scheduling purposes) of an asynchronous data-driven control program. The software modules Periodic_Boolean_Signal_Template and PBS_With_Delay_Template are the primary components of this software system. This system is the embodiment of a direct special-purpose algorithmic approach (not a simulation) designed to support the efficient analysis and scheduling of an asynchronous data-driven control program.

Section 2.2.2 described the specification of the Periodic_Boolean_Signal_Template. This module exports an abstract data type, Periodic_Boolean_Signal, that models the timing behavior of a control process via two abstract entities, pbs.pulse and pbs.level. Pbs.pulse represents the times at which a control process activates its successors while pbs.level represents the times at which a process is executing. These two entities are related through a mathematical invariant to ensure that a Periodic_Boolean_Signal
variable correctly captures the steady-state timing behavior of an asynchronous data-driven control process. The Periodic_Boolean_Signal_Template additionally exports a set of operations for construction and manipulation of the periodic steady-state timing behavior of a control process, i.e., for creating arbitrary values of type Periodic_Boolean_Signal.

The specification of the PBS_With_Delay_Template was presented in Section 2.2.3. This module exports one new operation, Delay, that allows the periodic timing behavior of a non-sampling process to be determined based on its compute delay and the initiation pattern of its predecessors.

The specification of these modules is important because, together, Periodic_Boolean_Signal_Template and PBS_With_Delay_Template provide a natural abstraction for the timing semantics of an asynchronous data-driven control program. Furthermore, the module specifications are simplified by the fact that the explicit passage of time and process operational properties are ignored. Only those timing properties that are necessary for analysis and scheduling purposes are included in the specifications.

5.1.3 Implementation of Software System

Section 3.1 demonstrated that simulation techniques for determining the timing behavior of an asynchronous data-driven control process are unattractive due to their computational complexity. For this reason an efficient special-purpose algorithmic approach for this problem is of interest. The implementation of the software modules that comprise this approach was addressed in the remaining sections of Chapter 3.

Section 3.2 described the implementation of the Periodic_Boolean_Signal_Template and the PBS_With_Delay_Template, that is, the implementation of the data type and operations exported by these modules. The program representation of the abstract data type Periodic_Boolean_Signal contains relevant information for the efficient implementation of the operations that manipulate a variable of this type. Several of the exported operations — Make_Simple_Periodic_Boolean_Signal, Shift, Smallest_Pulse_Period, Smallest_Level_Period and Is_True_Pulse — execute in constant time. Optimized algorithms were designed to ensure an efficient implementation for those operations, Or and Delay, whose run-time complexity is not constant and that are used in the direct algorithmic scheduling approach.
The implementations presented in Chapter 3 allow for the efficient construction of the steady-state timing behavior of an asynchronous data-driven control program. This supports point one of the thesis, namely, that a control program with asynchronous data-driven execution can be analyzed in a computationally tractable manner.

5.1.4 Implementation of the Static Analyzer/Scheduler Tool

The implementation of the Static Analyzer/Scheduler tool was described in Chapter 4. This tool is the embodiment of the special-purpose algorithmic approach for analysis and scheduling of an asynchronous data-driven control program.

Section 4.1 presented the method by which the Static Analyzer/Scheduler constructs the steady-state timing behavior of a control program. This method requires producing a sequence of calls to the operations of the Periodic_Boolean_Signal_Template and PBS_With_Delay_Template based upon the DAG representation of a control program and its timing parameters. This sequence of operations, when executed, constructs the steady-state timing behaviors for all processes within the control program.

The remaining sections of the chapter demonstrated how the Static Analyzer/Scheduler performs phase-shifts for sampling processes, computes end-to-end system response times, assigns processes to processors and constructs resident schedules. These activities are performed in an efficient manner supporting the second point of the thesis, namely that determination of a feasible static multiprocessor schedule for an asynchronous data-driven control system can be efficiently implemented.

5.2 Contributions

The focus of this research has been (1) to introduce an alternative model for the analysis and scheduling problem for a class of hard real-time control software, that is, the asynchronous data-driven execution model; (2) to demonstrate that the steady-state timing behavior of an asynchronous data-driven control program can be determined in a computationally tractable manner; and (3) to show that a feasible static multiprocessor schedule for such a control program can be efficiently implemented. The following are the main contributions to the field:
A semi-formal definition of the synchronous clock-driven execution model. The timing constraints of this execution model require synchronization of process sampling periods, so all processes in a control program execute in a simple periodic manner. This definition reveals the separate optimizations that must be performed in order to determine an optimum static multiprocessor schedule for a synchronous clock-driven control program.

A semi-formal definition of the asynchronous data-driven execution model. The timing constraints of this execution model require all non-sampling processes to execute in a self-timed data-driven manner, so some processes may exhibit complex periodic behavior. This definition removes the front end optimization problem of synchronizing periods and process execution as required by the synchronous clock-driven model. The asynchronous data-driven model provides an alternative tool for the problem of analyzing and scheduling control programs. In certain cases, a schedule produced using this execution model may use fewer processors and have shorter end-to-end delay than a schedule produced using the synchronous clock-driven execution model.

New abstractions for scheduling. In the course of this research three new abstractions were developed. The Periodic_Function_Template was designed as a reusable module that captures the behavior of periodic functions that map integers to booleans. It was used strictly as an explanatory tool for the remaining two modules. The Periodic_Boolean_Signal_Template was designed as a reusable module that captures the steady-state timing semantics of an asynchronous data-driven control program. The PBS_With_Delay_Template was designed as an application-specific module for the construction of the steady-state timing behavior of a non-sampling process. The specification of these modules was simplified by abstracting away the explicit passage of time and process operational characteristics.

Efficient analysis of asynchronous data-driven timing behaviors. The algorithms designed to implement the operations (exported by Periodic_Boolean_Signal_Template and PBS_With_Delay_Template) that construct the steady-state timing behavior of a control program do so in an efficient manner.
Static Analyzer/Scheduler tool. This tool efficiently constructs a feasible static multiprocessor schedule for an asynchronous data-driven control program.

5.3 Future Work

Several areas seem to offer promising avenues for future work involving extensions to the asynchronous data-driven execution model and extensions to improve the Static Analyzer/Scheduler tool. A few of the possibilities are presented below.

Incorporating communication delays for a static distributed schedule. The Static Analyzer/Scheduler constructs a feasible static schedule for a shared-memory multiprocessor system. In this type of parallel system inter-processor communication delays are typically nominal and for this reason often ignored, as is the case in this dissertation. Extending the scope of the Static Analyzer/Scheduler to construct a feasible static schedule for a distributed multiprocessor system would require modeling worst-case inter-processor communication delays. This necessitates the incorporation of communication delays in the asynchronous data-driven execution model. Preliminary work suggests that communication delays may be modeled as delay processes that postpone the execution of their successor processes but which actually don’t execute themselves. Therefore, although they would directly affect the calculation of the steady-state timing behavior of a control program they would not be candidates for scheduling. This would pose an interesting problem for the Static Analyzer/Scheduler. In order to maintain good system response times those processes that communicate frequently should be assigned to processors such that their inter-processor communication delay is minimized.

Control systems with software feedback loops. Currently, the Static Analyzer/Scheduler constructs a schedule only for those control systems with closed loops that contain an external physical system, that is, control systems that may be directly translated to a DAG representation by merely cutting embedded loops. Control systems may exhibit feedback loops that do not contain an external physical system, only compute processes. This type of feedback loop cannot be removed when translating the block diagram of a control program to its graph representation. In this case, the translation results in a directed cyclic graph representation. There has been
some preliminary work to iteratively determine the steady-state timing behavior of control programs with feedback loops. This work indicates that only minor modifications to the Static Analyzer/Scheduler are required to enhance its capability for determining the steady-state timing behavior of cyclic control programs.

Incorporating the notion of process deadlines. Much of the current scheduling research in the real-time systems community focuses on scheduling techniques that attempt to ensure that process deadlines (the latest time at which a process may start its execution after an initiation request) are not missed. Cursory investigation suggests that the notion of process execution deadline can be incorporated in the asynchronous data-driven model of execution. Furthermore, the Static Analyzer/Scheduler has access to enough information to determine whether process timing deadlines are satisfied for a particular control program schedule.

Automating the choice of phase-shifts. The Static Analyzer/Scheduler does not attempt to phase-shift the start times of sampling processes in order minimize processor resource requirements or system response time. This problem is presently left in the hands of the user of the tool who must devise some ad hoc approach, such as choosing shifts randomly until satisfactory results are obtained. It is possible that an application of probability theory to this problem may produce some interesting results. Such a technique might randomly choose a set of phase-shifts for a particular control program. It would then calculate the timing behavior for each of the random phase-shifts, and determine bounds on processor requirements and end-to-end delay for each schedule’s implementation. It would then report on the probability of finding other phase-shifts that would result in a better schedule.

Determining the relationship between synchronous and asynchronous schedules. For each synchronous schedule does an asynchronous schedule exist with equal or better measures of merit? While no counterexample to such a claim has been discovered in the course of this work, no proof has been identified, either.
APPENDIX A

One Way List Specifications

This appendix provides Modula-2 definition modules for a one way list abstract data type including ListGen, ListGenDisplay, ListFac, ListDisplay and ListFacC.

A.1 ListGen Definition Module

DEFINITION MODULE ListGen;

(*---Types-------------------------------------------------------------*)

TYPE Item = INTEGER;

(*---Operations--------------------------------------------------------*)

(*---Operations--------------------------------------------------------*)

PROCEDURE Initialize (VAR x: Item);
(*

  ensures:  x has an initial value of type Item

*)

(*---Operations--------------------------------------------------------*)

PROCEDURE Finalize (VAR x: Item);

(*---Operations--------------------------------------------------------*)

PROCEDURE Swap (VAR x1: Item; VAR x2: Item);

(*---Operations--------------------------------------------------------*)

END ListGen.

110
A.2 ListGenDisplay Definition Module

DEFINITION MODULE ListGenDisplay;

(*---Imports---------------------------------------------------------------*)

IMPORT ListGen;
(*
    Module ListGen exports type Item; and 
    Initialize, Finalize, and Swap procedures 
    for that type. 
*)

(*---Operations--------------------------------------------------------------*)

PROCEDURE Display ( 
    (* preserves *) VAR x: ListGen.Item; 
    (* preserves *) indent: INTEGER 
);
(*
    requires: indent >= 0, and output cursor is 
    at left margin 
    ensures: x is displayed indent spaces from 
    the left margin, and output cursor 
    is at left margin of next line 
*)

(*---------------------------------------------*)

END ListGenDisplay.

A.3 ListFac Definition Module

DEFINITION MODULE ListFac;

(*---Imports---------------------------------------------------------------*)

IMPORT ListGen;
(*
    Module ListGen exports type Item, and 
    Initialize, Finalize, and Swap procedures 
    for that type. 
*)

(*---Types---------------------------------------------------------------*)
TYPE List;
(*
A List variable denotes a sequence of items of type ListGen.Item, and a dividing "fence" between two of the items in the sequence, or perhaps before the first item or after the last. The fence can be moved to "start" (far left end) or "finish" (far right end) of a List's sequence, or "advanced" one position to the right. An item can be added to a List's sequence just to the right of its fence, or the item just to the right of the fence in can be removed from the sequence. It is also possible to swap (exchange) the portions (subsequences) of two Lists that lie to the right of their respective fences, and to test whether a List's fence is at the start or finish of its sequence.
*)

(*---Operations--------------------------------*)

(*------------------------------------------------*)

PROCEDURE Initialize (VAR s: List);
(*
  ensures:  s is an empty sequence
*)

(*------------------------------------------------*)

PROCEDURE Finalize (VAR s: List);

(*------------------------------------------------*)

PROCEDURE Swap (VAR s1: List; VAR s2: List);

(*------------------------------------------------*)

PROCEDURE MoveToStart (VAR s: List
  (* alters *)
);  (*
  ensures:  the sequence of s = the sequence of s, and
  the fence of s is at the far left end of the sequence of s
*)

(*------------------------------------------------*)
PROCEDURE MoveToFinish (  
  (* alters *) VAR s: List  
);  
(*  
  ensures:  the sequence of s = the sequence  
  of #s, and  
  the fence of s is at the far right  
  end of the sequence of s  
*)  

(*--------------------------------------------------------*)

PROCEDURE Advance (  
  (* alters *) VAR s: List  
);  
(*  
  requires:  the fence of s is not at the right  
  end of the sequence of s  
  ensures:  the sequence of s = the sequence  
  of #s, and  
  the fence of s is one position to  
  the right of the fence of #s  
*)  

(*--------------------------------------------------------*)

PROCEDURE AddRight (  
  (* alters *) VAR s: List;  
  (* consumes *) VAR x: ListGen.Item  
);  
(*  
  ensures:  the sequence of s = the sequence  
  of #s with #x added just to the  
  right of the fence, and  
  the fence of s has the same  
  subsequence to its left as the  
  fence of #s  
*)  

(*--------------------------------------------------------*)

PROCEDURE RemoveRight (  
  (* alters *) VAR s: List;  
  (* produces *) VAR x: ListGen.Item  
);  
(*  
  requires:  the fence of s is not at the right  
  end of the sequence of s  
  ensures:  the sequence of s = the sequence  
  of #s with the Item just to the  
  right of the fence removed, and x  
*)
is the Item removed, and the fence
of s has the same subsequence to
its left as the fence of #s

PROCEDURE SwapRights (          \parspace
    (* alters *)  VAR s1: List;
    (* alters *)  VAR s2: List
);          \parspace
    (* ensures:  the subsequences to the left of
the fences of s1 and s2 are unchanged, and the subsequences to
the right of the fences of s1 and
s2 are swapped (exchanged)

PROCEDURE AtStart (          \parspace
    (* preserves *)  VAR s: List
    ): BOOLEAN;
    (* ensures:  AtStart returns TRUE if and only
if the fence of s is at the left
end of the sequence of s

PROCEDURE AtFinish (          \parspace
    (* preserves *)  VAR s: List
    ): BOOLEAN;
    (* ensures:  AtFinish returns TRUE if and only
if the fence of s is at the right
end of the sequence of s

A.4 ListDisplay Definition Module

DEFINITION MODULE ListDisplay;
(*---Imports---------------------------------------------*)
IMPORT ListFac;
(*
   Module ListFac exports type List (of Item); Initialize, Finalize, and Swap procedures for that type; and operations MoveToStart, MoveToEnd, Advance, AddRight, RemoveRight, SwapRights, AtStart, and AtFinish.
*)

(*---Operations-----------------------------------------------*)

(*-------------------------------------------------------------*)

PROCEDURE Display (
   (* preserves *)   VAR s:
   ListFac.List;
   (* preserves *)   indent: INTEGER
);

(*
   requires: indent >= 0, and output cursor is at left margin 
ensures: s is displayed indent spaces from the left margin, and output cursor is at left margin of next line
*)

(*-------------------------------------------------------------*)

END ListDisplay.

A.5 ListFacC Definition Module

DEFINITION MODULE ListFacC;

(*---Imports-----------------------------------------------*)

IMPORT ListGen;
(*
   Module ListGen exports type Item, and Initialize, Finalize, and Swap procedures for that type.
*)

(*---Types-----------------------------------------------*)

TYPE List;
(*
A List variable denotes a sequence of items of type ListGen.Item, and an dividing "fence" between two of the items in the sequence, or perhaps before the first item or after the last. Operations can move the fence to the "start" (far left end) or "finish" (far right end) of a List's sequence, or "advance" it one position to the right. An item can be added to a List's sequence just to the right of its fence, or the item just to the right of the fence in can be removed from the sequence. It is also possible to swap (exchange) the portions (subsequences) of two Lists that lie to the right of their respective fences, and to test whether a List's fence is at the start or finish of its sequence.

*)

(*---Operations-----------------------------------------------*)

(*----------------------------------------------------------*)

PROCEDURE Initialize (VAR s: List);
(*
  ensures: s is an empty sequence
*)

(*----------------------------------------------------------*)

PROCEDURE Finalize (VAR s: List);  
(*----------------------------------------------------------*)

PROCEDURE Swap (VAR s1: List; VAR s2: List);  
(*----------------------------------------------------------*)

PROCEDURE MoveToStart (  
  (* alters *) VAR s: List 
);  
(*  
  ensures: the sequence of s = the sequence of #s, and\the fence of s is at the far left end of the sequence of s 
*)

(*----------------------------------------------------------*)
PROCEDURE MoveToFinish (
    (* alters *) VAR s: List
);

(*
    ensures: the sequence of s = the sequence of #s, and the fence of s is at the far right end of the sequence of s
*)

(*-----------------------------------------------------------------*

PROCEDURE Advance (
    (* alters *) VAR s: List
);

(*
    requires: the fence of s is not at the right end of the sequence of s
    ensures: the sequence of s = the sequence of #s, and the fence of s is one position to the right of the fence of #s
    checks: the fence of s is not at the right end of the sequence of s
*)

(*-----------------------------------------------------------------*

PROCEDURE AddRight (
    (* alters *) VAR s: List;
    (* consumes *) VAR x: ListGen.Item
);

(*
    ensures: the sequence of s = the sequence of #s with #x added just to the right of the fence, and the fence of s has the same subsequence to its left as the fence of #s
*)

(*-----------------------------------------------------------------*

PROCEDURE RemoveRight (
    (* alters *) VAR s: List;
    (* produces *) VAR x: ListGen.Item
);

(*
    requires: the fence of s is not at the right end of the sequence of s
    ensures: the sequence of s = the sequence of #s with the Item just to the
right of the fence removed, and x
is the Item removed, and the fence
of s has the same subsequence to
its left as the fence of #s
checks: the fence of s is not at the right
end of the sequence of s
*)

(*----------------------------------------*)

PROCEDURE SwapRights (
     (* alters *) VAR s1: List;
     (* alters *) VAR s2: List
)
(*
    ensures: the subsequences to the left of
    the fences of s1 and s2 are
    unchanged, and the subsequences to
    the right of the fences of s1 and
    s2 are swapped (exchanged)
*)

(*----------------------------------------*)

PROCEDURE AtStart (
     (* preserves *) VAR s: List
)
(*
    ensures: AtStart returns TRUE if and only
    if the fence of s is at the left
    end of the sequence of s
*)

(*----------------------------------------*)

PROCEDURE AtFinish (
     (* preserves *) VAR s: List
)
(*
    ensures: AtFinish returns TRUE if and only
    if the fence of s is at the right
    end of the sequence of s
*)

(*----------------------------------------*)

END ListFacC.
APPENDIX B

One Way List Implementations

This appendix provides Modula-2 implementation modules for a one way list abstract data type including ListGen, ListGenDisplay, ListFac, ListDisplay and ListFacC.

B.1 ListGen Implementation Module

IMPLEMENTATION MODULE ListGen;

(*--- Operations -----------------------------------------------*)

(*--- Operations -----------------------------------------------*)

PROCEDURE Initialize (VAR x: Item);
(*
  ensures:  x has an initial value of type Item
*)
BEGIN
  x := 0;
END Initialize;

PROCEDURE Finalize (VAR x: Item);
BEGIN
END Finalize;

PROCEDURE Swap (VAR x1: Item; VAR x2: Item);
VAR
temp: INTEGER;
BEGIN
  temp := x1;
  x1 := x2;
  x2 := temp;
END Swap;

(*-----------------------------------------------------*)

END ListGen.

B.2 ListGenDisplay Implementation Module

IMPLEMENTATION MODULE ListGenDisplay;

(*--- Imports ------------------------------------------------*)
IMPORT ListGen;
FROM Indenting IMPORT MoveOver;
FROM SInOut IMPORT WriteInt, WriteLn;

(*--- Operations ------------------------------------------*)

PROCEDURE Display (VAR x: ListGen.Item;
  (* preserves *) indent: INTEGER);
  (* preserves *) indent: INTEGER);
  (* requires: indent >= 0, and output cursor is at left margin
  ensures: x is displayed indent spaces from the left margin, and output cursor is at left margin of next line *)
BEGIN
  MoveOver (indent);
  WriteInt (x, 0);
  WriteLn;
END Display;

(*-----------------------------------------------------*)

END ListGenDisplay.
B.3 ListFac Implementation Module

IMPLEMENTATION MODULE ListFac;

(*--- Imports ------------------------------------------*)

IMPORT ListGen;

FROM Storage IMPORT ALLOCATE, DEALLOCATE;

(*--- Types ------------------------------------------*)

TYPE
  List = POINTER TO ListRep;
  NodePtr = POINTER TO Node;

  ListRep =
    RECORD
      preFront: NodePtr;
      lastLeft: NodePtr;
      lastRight: NodePtr;
    END;

  Node =
    RECORD
      data: ListGen.Item;
      next: NodePtr;
    END;

(*--- Operations --------------------------------------*)

PROCEDURE Initialize (VAR s: List);
  (*
    ensures:  s  is an empty sequence
  *)
  VAR
    dummy: NodePtr;
  BEGIN
    (*--- Allocate storage for list header and dummy node ---*)
    ALLOCATE (s, SIZE (ListRep));
    ALLOCATE (dummy, SIZE (Node));
    (*--- Set up list header so all fields reference dummy ---*)

s^preFront := dummy;
s^lastLeft := dummy;
s^lastRight := dummy;

(*--- Set up dummy node ---*)
ListGen.Initialize (dummy^data);
dummy^next := NIL;

END Initialize;

(*--------------------------------------------------------------------------------------------------------------------------------*)

PROCEDURE Finalize (VAR s: List);
VAR
  oldPreFront: NodePtr;
BEGIN

  (*--- Reclaim all nodes in s, including dummy node ---*)
  WHILE s^preFront <> NIL DO
    (*--- Remember rest of list for next iteration ---*)
    oldPreFront := s^preFront;
    s^preFront := oldPreFront^next;
    (*--- Reclaim old preFront node ---*)
    ListGen.Finalize (oldPreFront^data);
    DEALLOCATE (oldPreFront, SIZE (Node));
  END;

  (*--- Reclaim list header ---*)
  DEALLOCATE (s, SIZE (ListRep));
END Finalize;

(*--------------------------------------------------------------------------------------------------------------------------------*)

PROCEDURE Swap (VAR s1: List; VAR s2: List);
VAR
  temp: List;
BEGIN
  temp := s1;
  s1 := s2;
\[ s_2 := \text{temp}; \]
END Swap;

\[
(*---------------------------------------------------------------*)

PROCEDURE MoveToStart (  
    (* alters *)  VAR s: List  
);  
(*)  
ensures:  the sequence of \( s \) = the sequence 
of \( #s \), and the fence of \( s \) is at 
the far left end of the sequence 
of \( s \)  
*)
BEGIN  
s^lastLeft := s^preFront;  
END MoveToStart;  

\[
(*---------------------------------------------------------------*)

PROCEDURE MoveToFinish (  
    (* alters *)  VAR s: List  
);  
(*)  
ensures:  the sequence of \( s \) = the sequence 
of \( #s \), and the fence of \( s \) is at 
the far right end of the sequence 
of \( s \)  
*)
BEGIN  
s^lastLeft := s^lastRight;  
END MoveToFinish;  

\[
(*---------------------------------------------------------------*)

PROCEDURE Advance (  
    (* alters *)  VAR s: List  
);  
(*)  
requires:  the fence of \( s \) is not at the right 
end of the sequence of \( s \) 
ensures:  the sequence of \( s \) = the sequence 
of \( #s \), and the fence of \( s \) is one 
position to the right of the fence 
of \( #s \)  
*)
BEGIN  
s^lastLeft := s^lastLeft^next;  
END Advance;  

\[
(*---------------------------------------------------------------*)

PROCEDURE AddRight (
    (* alters *) VAR s: List;
    (* consumes *) VAR x: ListGen.Item
)

(*
  ensures: the sequence of s = the sequence of s with x added just to the
  right of the fence, and the fence of s has the same subsequence to its
  left as the fence of s *)
VAR
  newFirstRight: NodePtr;
BEGIN

(*--- Create new node with initialized data ---*)
ALLOCATE (newFirstRight, SIZE (Node));
ListGen.Initialize (newFirstRight^.data);

(*--- Consume x by swapping it into new node ---*)
ListGen.Swap (newFirstRight^.data, x);

(*--- Hook new node into list ---*)
newFirstRight^.next := s^.lastLeft^.next;
s^.lastLeft^.next := newFirstRight;
IF s^.lastLeft = s^.lastRight THEN

    (*--- New node is only one to right of fence; update lastRight ---*)
    s^.lastRight := newFirstRight;
END;
END AddRight;

(*--------------------------------------------------------------------------*)

PROCEDURE RemoveRight (
    (* alters *) VAR s: List;
    (* produces *) VAR x: ListGen.Item
);

(*
  requires: the fence of s is not at the right end of the sequence of s
  ensures: the sequence of s = the sequence of s with the Item just to the
  right of the fence removed, and x
is the item removed, and the fence of $s$ has the same subsequence to its left as the fence of $#s$.

*)

VAR

oldFirstRight: NodePtr;

BEGIN

(*--- Get reference to node just to right of fence ---*)

oldFirstRight := $s$.lastLeft.next;

(*--Produce x by swapping it out of old node--*)

ListGen.Swap (oldFirstRight.data, x);

(*--- Unhook old node from list ---*)

s.lastLeft.next := oldFirstRight.next;

IF s.lastRight = oldFirstRight THEN

(*--- Old node is only one to right of fence; update lastRight ---*)

s.lastRight := s.lastLeft;

END;

(*--- Reclaim old node ---*)

ListGen.Finalize (oldFirstRight.data);

DEALLOCATE (oldFirstRight, SIZE (Node));

END RemoveRight;

(*--------------------------------*)

PROCEDURE SwapRights (  
    (* alters *)  VAR s1: List;  
    (* alters *)  VAR s2: List  
    );

(*
ensures: the subsequences to the left of the fences of $s1$ and $s2$ are unchanged, and the subsequences to the right of the fences of $s1$ and $s2$ are swapped (exchanged)
*)

VAR

temp: NodePtr;

BEGIN
(*--- Swap next fields of last nodes to left of fences ---*)

\[ \text{temp} := \text{s1}^{\text{lastLeft}}.\text{next}; \]
\[ \text{s1}^{\text{lastLeft}}.\text{next} := \text{s2}^{\text{lastLeft}}.\text{next}; \]
\[ \text{s2}^{\text{lastLeft}}.\text{next} := \text{temp}; \]

(*--- Swap pointers to last nodes to right of fences ---*)

\[ \text{temp} := \text{s1}^{\text{lastRight}}; \]
\[ \text{s1}^{\text{lastRight}} := \text{s2}^{\text{lastRight}}; \]
\[ \text{s2}^{\text{lastRight}} := \text{temp}; \]

(*--- Fix up pointers to last right nodes, if necessary ---*)

IF \( \text{s1}^{\text{lastRight}} = \text{s2}^{\text{lastLeft}} \) THEN

(*--- s2 has nothing to right of fence ---*)

\[ \text{s1}^{\text{lastRight}} := \text{s1}^{\text{lastLeft}}; \]
END;
IF \( \text{s2}^{\text{lastRight}} = \text{s1}^{\text{lastLeft}} \) THEN

(*--- s1 has nothing to right of fence ---*)

\[ \text{s2}^{\text{lastRight}} := \text{s2}^{\text{lastLeft}}; \]
END;

END \text{SwapRights};

(*---------------------------------------------------------------------*)

PROCEDURE AtStart (  
( * preserves * )  VAR \text{s: List}  
): BOOLEAN;

(*  
ensures: AtStart returns TRUE if and only if the fence of \text{s} is at the left end of the sequence of \text{s} *)
BEGIN
RETURN \text{s}^{\text{lastLeft}} = \text{s}^{\text{preFront}};
END AtStart;

(*---------------------------------------------------------------------*)

PROCEDURE AtFinish (  
( * preserves * )  VAR \text{s: List}  
)
BEGIN
    RETURN s^.lastLeft = s^.lastRight;
END AtFinish;

END ListFac.

B.4 ListDisplay Implementation Module

IMPLEMENTATION MODULE ListDisplay;

(*------ Imports  * )
IMPORT ListGen;
IMPORT ListFac;
IMPORT ListGenDisplay;
FROM Indenting IMPORT MoveOver;
FROM SInOut IMPORT Write, WriteLn;

(*------ Exported operations  * )

PROCEDURE Display ( (* preserves *) VAR s: ListFac.List;
    (* preserves *)indent: INTEGER);
    (*
        requires: indent >= 0, and output cursor is at left margin
        ensures:  s is displayed indent spaces from the left margin, and output cursor
                  is at left margin of next line
    *)
    VAR
        x: ListGen.Item;
        catalyst: ListFac.List;
    BEGIN
        ListFac.Initialize (catalyst);

        (*
            ensures: AtFinish returns TRUE if and only if the fence of s is at the right end
            of the sequence of s
        *)
        RETURN s^.lastLeft = s^.lastRight;
END AtFinish;

END ListDisplay;
ListGen.Initialize (x);

(*--- Split s into left part (remains in s) and right part in catalyst) ---*)

ListFac.SwapRights (s, catalyst);

(*--- Print opening sequence delimiter ---*)

MoveOver (indent); Write ('<'); WriteLn;

(*--- Print each Item in left part ---*)

ListFac.MoveToStart (s);
WHILE NOT ListFac.AtFinish (s) DO
  ListFac.RemoveRight (s, x);
  ListGenDisplay.Display (x, indent + 4);
  IF NOT ListFac.AtFinish (s) THEN
    MoveOver (indent + 2);
    Write (',');
    WriteLn;
  END;
  ListFac.AddRight (s, x);
  ListFac.Advance (s);
END;

(*--- Print fence separator ---*)

MoveOver (indent + 2);
Write ('|');
WriteLn;

(*--- Print each Item in right part ---*)

WHILE NOT ListFac.AtFinish (catalyst) DO
  ListFac.RemoveRight (catalyst, x);
  ListGenDisplay.Display (x, indent + 4);
  IF NOT ListFac.AtFinish (catalyst) THEN
    MoveOver (indent + 2);
    Write (',');
    WriteLn;
  END;
  ListFac.AddRight (catalyst, x);
  ListFac.Advance (catalyst);
END;
ListFac.MoveToStart (catalyst);

(*--- Print closing sequence delimiter ---*)

MoveOver (indent); Write ('>'); WriteLn;
(*--- Reconstitute s from its two parts ---*)
ListFac.SwapRights (s, catalyst);
  ListGen.Finalize (x);
  ListFac.Finalize (catalyst);
END Display;

(*---------------------------------------------------*)

END ListDisplay.

B.5 ListFacC Implementation Module

IMPLEMENTATION MODULE ListFacC;

(*--- Imports  ----------------------------------------*)
IMPORT ListGen;
IMPORT ListFac;
FROM Checking IMPORT Assert;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;

(*--- Types  ----------------------------------------*)
TYPE
  List = POINTER TO ListFac.List;

(*--- Operations  -----------------------------------*)

PROCEDURE Initialize (VAR s: List);
(*
  ensures:  s is an empty sequence
*)
BEGIN
  ALLOCATE (s, SIZE (ListFac.List));
  ListFac.Initialization (s^);
END Initialize;

(*---------------------------------------------------*)

PROCEDURE Finalize (VAR s: List);
BEGIN
  ListFac.Finalize (s^);
DEALLOCATE (s, SIZE (ListFac.List));
END Finalize;

(*------------------------------------------------------------------*)

PROCEDURE Swap (VAR s1: List; VAR s2: List);
BEGIN
  ListFac.Swap (s1^, s2^);
END Swap;

(*------------------------------------------------------------------*)

PROCEDURE MoveToStart (  
  (* alters *)  VAR s: List
);
  
  (*
  ensures: the sequence of s = the sequence
  of #s, and the fence of s is at
  the far left end of the sequence
  of s
  *)
BEGIN
  ListFac.MoveToStart (s^);
END MoveToStart;

(*------------------------------------------------------------------*)

PROCEDURE MoveToFinish (  
  (* alters *)  VAR s: List
);
  
  (*
  ensures: the sequence of s = the sequence
  of #s, and the fence of s is at
  the far right end of the sequence
  of s
  *)
BEGIN
  ListFac.MoveToFinish (s^);
END MoveToFinish;

(*------------------------------------------------------------------*)

PROCEDURE Advance (  
  (* alters *)  VAR s: List
);
  
  (*
  requires: the fence of s is not at the right
  end of the sequence of s
  ensures: the sequence of s = the sequence
  of #s, and the fence of s is one
  *)
position to the right of the fence of #s
checks: the fence of s is not at the right end of the sequence of s

BEGIN
  Assert (NOT ListFac.AtFinish (s^));
  ListFac.Advance (s^);
END Advance;

(*--------------------------------------------------------------------------------*)

PROCEDURE AddRight (  
  (* alters *)  VAR s: List;
  (* consumes *)  VAR x: ListGen.Item );
  
  (*
  ensures: the sequence of s = the sequence of #s with #x added just to the right of the fence, and the fence of s has the same subsequence to its left as the fence of #s
  *)
BEGIN
  ListFac.AddRight ( s^, x);
END AddRight;

(*--------------------------------------------------------------------------------*)

PROCEDURE RemoveRight (  
  (* alters *)  VAR s: List;
  (* produces *)  VAR x: ListGen.Item );
  
  (*
  requires: the fence of s is not at the right end of the sequence of s
  ensures: the sequence of s = the sequence of #s with the Item just to the right of the fence removed, and x is the Item removed, and the fence of s has the same subsequence to its left as the fence of #s
  checks: the fence of s is not at the right end of the sequence of s
  *)
BEGIN
  Assert (NOT ListFac.AtFinish (s^));
  ListFac.RemoveRight (s^, x);
END RemoveRight;

(*--------------------------------------------------------------------------------*)
PROCEDURE SwapRights(
    (* alters *) VAR s1: List;
    (* alters *) VAR s2: List
);

(* ensures: the subsequences to the left of the fences of s1 and s2 are unchanged, and the subsequences to the right of the fences of s1 and s2 are swapped (exchanged) *)
BEGIN
    ListFac.SwapRights (s1^, s2^);
END SwapRights;

(*-----------------------------------------------------------------------------*)

PROCEDURE AtStart(
    (* preserves *) VAR s: List
    ): BOOLEAN;

(* ensures: AtStart returns TRUE if and only if the fence of s is at the left end of the sequence of s *)
BEGIN
    RETURN ListFac.AtStart (s^);
END AtStart;

(*-----------------------------------------------------------------------------*)

PROCEDURE AtFinish(
    (* preserves *) VAR s: List
    ): BOOLEAN;

(* ensures: AtFinish returns TRUE if and only if the fence of s is at the right end of the sequence of s *)
BEGIN
    RETURN ListFac.AtFinish (s^);
END AtFinish;

(*-----------------------------------------------------------------------------*)

END ListFacC.
BIBLIOGRAPHY


