6.

Examples

This chapter presents four examples. Although all examples will be shown from specification to partitioning, each of them has peculiarities which will be used to show specific details of the methodology proposed in this dissertation.

Two of them are real engineering problems: a mine pump controller and a brushless DC motor speed controller. The former has been used by several authors to illustrate specification methods [20, 21, 75, 87, 110]. In the particular case of this dissertation, it is used to show refinement phases towards producing an output suitable for the scheduling algorithm, also discussed in Chapter 3. The latter comes from an ASIC controller project [69, 70] and is used here to stress the hierarchical capabilities that were also presented in Chapter 3.

The other two are purely academic with no specific meaning in the real world and both are based on the same example by [115]. They are referred here as *York1* and *York2*, since the original work was produced at the University of York. Their complexity is used to show scheduling and partitioning issues discussed in Chapters 4 and 5.

Their partitioning reports (see an example in Appendix D) are not included in this dissertation due to the amount space it would take.

6.1 Mine Pump Controller

This example possesses many of the characteristics which typify embedded real-time systems. It consists on the design of a mine drainage system used to pump water from the sump at the shaft bottom to the surface. Once enabled by a command from the control room it works automatically based on water level sensors, and environment conditions. Detection of a high water level causes the pump to run until a low level is detected or before if requested by the operator.

Environment conditions are monitored so that a detection of methane level (CH_4) in the air beyond which it is not safe to operate the pump causes the system to shut down until safe levels are reached. The level of carbon monoxide (CO) and air flow in the mine also need to be measured in order to keep operation safe. Alarms must be raised if dangerous conditions are detected. A data log is kept during the pump operation and stores information from all sensors.

6.1.1 Initial Specification

The following devices have their reading periods defined initially:

Sensor	Period (seconds)	deadline
CH_4	5	1
CO	25	1
Water Flow	5	3
Air Flow	60	2

Table 6.1. Pump Controller's periodic tasks

The water level sensor is event triggered. Its minimum inter-arrival time is 100 seconds and the system must respond within 20 seconds. Ten seconds after turning the motor on or off, a check needs to be made on the water flow to verify if the system is working correctly. Figure 6.1 shows the initial specification net for the whole system. Table 6.2 shows the timing specifications of each task.

The transitions *Switch_OnOff* and *Read_Status* model operator requests and so are part of the external environment. All above mentioned periodic tasks are represented by periodic process nets. The *Water_Level_Sensor*, being event triggered, is represented by a sporadic task. Notice that all tasks in the system are being represented as super-transitions. This was done to stress the possibility of further refining each task specification.

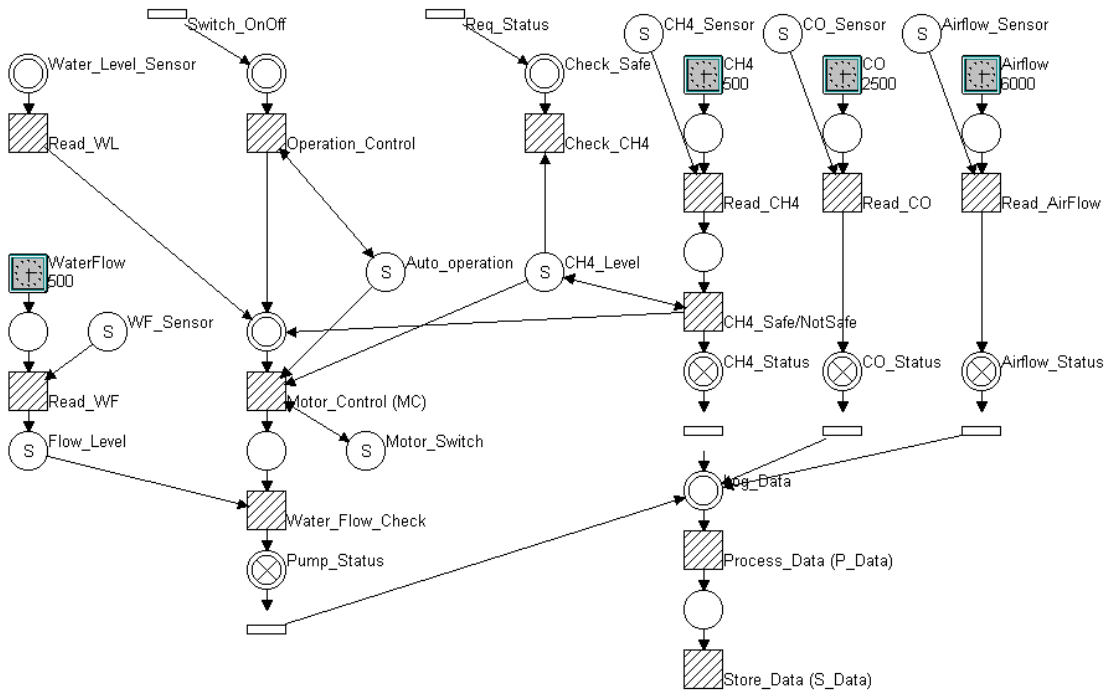


Figure 6.1. Pump Controller's initial net specification

A *Motor_Control* sporadic process net is being used to represent the control over the motor switch. This control receives data from several other process nets, some sporadic and some periodic. In this model, The *CH_4_Safe/NotSafe* task is the one responsible for verifying the CH_4 safety level. If necessary it sends a message to *Motor_Control* to turn the motor on or off.

The tasks *Process_Data* and *Store_Data* do not have pre-defined inter-arrival times or deadlines, but must be able to handle all data generated by other tasks. The *Motor_-Control* and *Water_Flow_Check* tasks have defined deadlines but no inter-arrival times. However, they must be able to handle any calls from other tasks. In particular, *Motor_-Control* must be able to turn the pump off within one second after detection of high methane level. Also the response time for handling input from the water level sensors is 20s.

Task	M.C.	WFC	Read_WL	Read_CH_4	CH_4 S/NS	Read_CO	Read_AF	Read_WF	P_Data	S_Data
Type	Spor.	Spor.	Spor.	Per.	Per.	Per.	Per.	Per.	Spor.	Spor.
r	0.0	10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
c	0.10	0.15	0.01	0.25	0.05	0.15	0.15	0.15	0.15	0.10
d	1.0	12.0	20.0	5.0	1.0	1.0	2.0	3.0	-	-
p	-	-	100.0	5.0	5.0	25.0	60.0	5.0	-	-

Table 6.2. Pump Controller's timing specification

6.1.2 Model Refinement

Due to the existence of sporadic tasks, Time Wizard cannot generate a specification file to be used by the scheduler/partitioner. Therefore, those sporadic process nets need to be modified and transformed to present a periodic behaviour.

The first change is the elimination of the process nets which answer *Switch_OnOff*, since the *On* command can be considered as a system reset and the *Off* command simply shuts down the whole system. The *Request_Status* command is also removed from the model, since it does not present a critical deadline.

The sporadic water level sensor handler (*Read_WL*) can be modelled by the tuple $(minp_s, c, d_s) = (100, 0.15, 20)$. Based on the transformations shown in Section 3.3.6.1, the following properties must be observed in order to model that task as a periodic one: $c \leq d_p \leq d_s$, and $p_p \leq d_s - d_p + 0.01$[5]. The values $p_p = 10$ and $d_p = 10$ satisfy the equations while keeping some flexibility, and so are used here. The same transformation is applicable to *Water_Flow_Check.*

Motor_Control must respond to sporadic calls from *CH_4_Safe/NotSafe* within 1s. So, the sporadic implementation of *Motor_Control* can be modelled by the tuple $(minp_s, c, d_s) = (5, 0.10, 1)$, since the periodicity of *CH_4_Safe/NotSafe* is 5s. Possible periodic implementations of *Motor_Control* and *Water_Flow_Check* are modelled as $(p_{mc}, r_{mc}, c_{mc}, d_{mc}) = (0.80, 0, 0.10, 0.21)$ and $(p_{wfc}, r_{wfc}, c_{wfc}, d_{wfc}) = (0.80, 0, 0.15, 0.20)$. Instead of using these method, *Water_Flow_Check* can be separated from *Motor_Control* and treated as a separate process net. Thus it would not have to stick to *Motor_Control* constraints. Its periodicity is still based on *CH_4_Safe/NotSafe* but its deadline is much less stringent. *Water_Flow_Check* can be modelled by the periodic tuple $(p_{wfc}, r_{wfc}, c_{wfc}, d_{wfc}) = (5, 0, 0.15, 5)$[6].

Process_Data and *Store_Data* are called whenever some event happens in the system. Since the minimum period among all read-sensor tasks is 5s (from *Read_CH_4*), this is the period that has to be imposed to those log tasks in order to avoid missing any data.

Figure 6.2 and Table 6.3 show the refined model specification.

[5] The value 0.01 is used instead of 1 since this is the smallest scheduling unit in this example.

[6] Actually, these changes are not necessary since test results have shown that the schedule would be feasible even without them. However, they are used here to illustrate techniques for handling such problems.

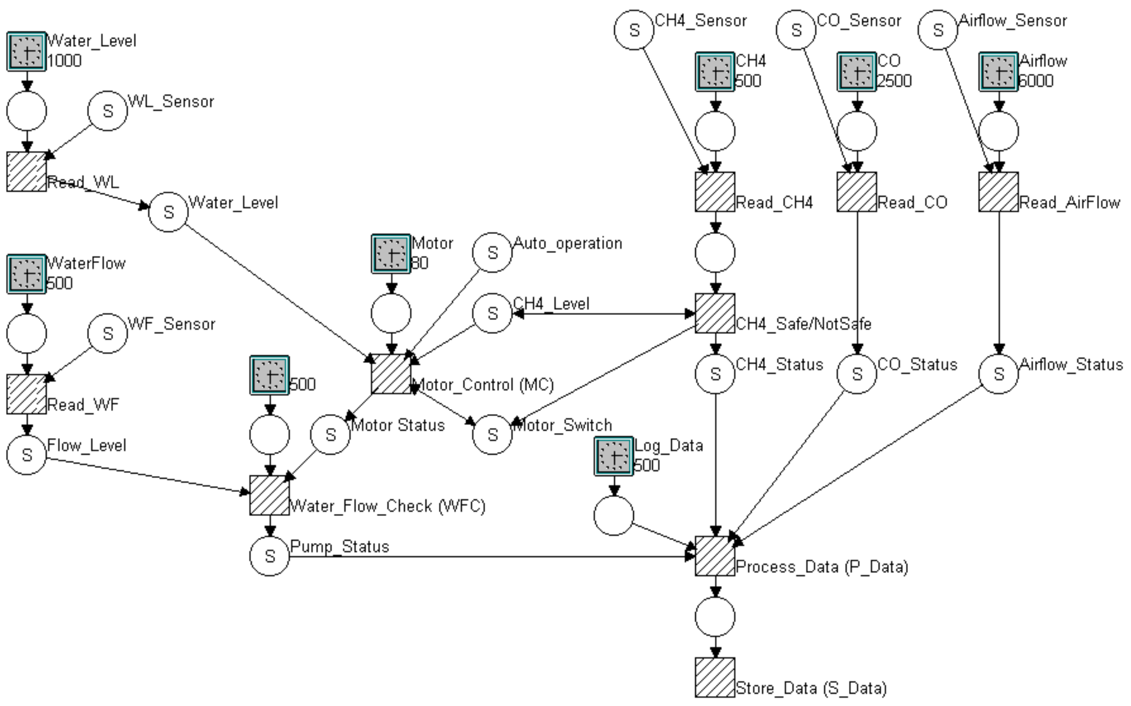


Figure 6.2. Pump Controller at the scheduling level

Task	M.C.	WFC	Read_WL	Read_CH_4	CH_4 S/NS	Read_CO	Read_AF	Read_WF	P_Data	S_Data
Type	Per.	Per.	Per.	Per.	Per.	Per.	Per.	Per.	Per.	Per.
r	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
c	0.10	0.15	0.01	0.25	0.05	0.15	0.15	0.15	0.15	0.10
d	0.20	5.0	10.0	5.0	1.0	1.0	2.0	3.0	5.0	5.0
p	0.80	5.0	10.0	5.0	5.0	25.0	60.0	5.0	5.0	5.0

Table 6.3. Pump Controller's refined timing specification

6.1.3 Results

This example is especially difficult to deal with by pre-runtime approaches, since only periodic tasks are allowed. Although a feasible system was produced, the processor utilisation is much higher than what would have been achieved if fixed-priority scheduling techniques were employed.

In particular, the periodic implementations of *Motor_Control* use the processor much more than in a fixed-priority implementation. Nevertheless, this example is valuable to illustrate how to handle such systems using transformation techniques. Also, it enhances the fact that co-design partitioning has a lot to offer to pre-runtime based design, since threads which require high responsiveness can normally be split in two parts, as shown in the example. The small but highly demanding tasks can then be analysed and, if necessary, automatically moved to hardware.

Figure 6.3 shows the first 1.36 seconds of the schedule, which are the most utilised during the scheduling period. The scheduler produced 782 segments, and solved the problem at the root node. The maximum lateness is -0.1s in a scheduling period of 300s.

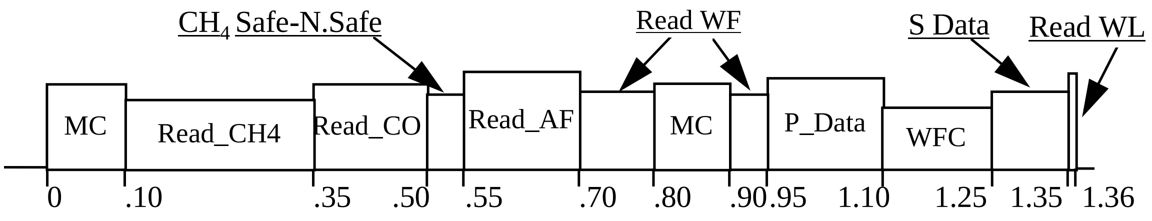


Figure 6.3. Pump Controller's partial scheduling results

6.2 Motor Controller

The example shown here is based on a real study case where ASIC technology was used to build a brushless DC motor controller [69, 70] in a highly integrated arrangement, thus requiring a small number of external components. The controller incorporates both torque and speed control loops. A host computer provides operation supervision.

The controller is intended to work with three-phase permanent-magnet motors and a power drive produces the required commutation of the motor excitation and power necessary to drive the motor (see [70] for a more detailed explanation). Speed control is produced by a proportional-integral-derivative (PID) control algorithm, which produces a current reference level for the torque control algorithm. Torque control is achieved through a proportional-integral (PI) current control, which compares the reference level supplied by the speed controller with the instantaneous current value.

6.2.1 Formulation

The speed controller produces a current reference (i_{ref}) based on the following formula:

$$i_{ref} = K_d.\frac{de_s}{dt} + K_p.e_s + K_i.\int e_s.dt$$

where K_d, K_p, and K_i are constants,

e_s = speed error.

In practice, these values are calculated as follows:

1. Measure time between interrupts (t_m) generated by the position sensors by counting clock cycles.
2. Calculate interrupt frequency $f_m = 1/t_m$.
3. Compare with desired frequency f_{ref}.
4. Calculate speed error $e_s = f_{ref} - f_m$.
5. Calculate differential error $de_s/dt = (f_{ref} - f_m) \cdot f_m$
6. Calculate integral error $\int e_s.dt = oei_s + (f_{ref} - f_m).t_m$, with oei_s = old integral error

The torque controller produces a voltage reference based on the equation below. These values are calculated in a similar way to i_{ref}.

$V_{ref} = C_p.e_c + C_i.\int e_c.dt$ where C_p and C_i are constants,

$e_c = i_m - i_{ref}$, i_m = instantaneous current value.

6.2.2 Temporal Requirements

The actual implementation of these algorithms using ASICs produced a WCET of 200μs to each speed control iteration, and 4μs to the torque control loop. It is assumed that the software implementation of such algorithms would be 10 times slower than the dedicated hardware implementation. Due to the need to maintain control loop stability, the speed control has a minimum period of 500μs and the torque control has a minimum period of 12.5μs. The table below presents a summary of the temporal specifications.

Task	Speed	Torque
μ	Sw	Sw
r	0	0
c(*sw/hw*)	2000/200	40/4
d	500	12.5
p	500	12.5

Table 6.4. Motor timing specifications

6.2.3 System Specification

Figure 6.4 shows the model produced using Time Wizard at the scheduling level. The host computer interface was not modelled since its timing constraints are considered to be soft, and so it is assumed that it can be executed as a background task.

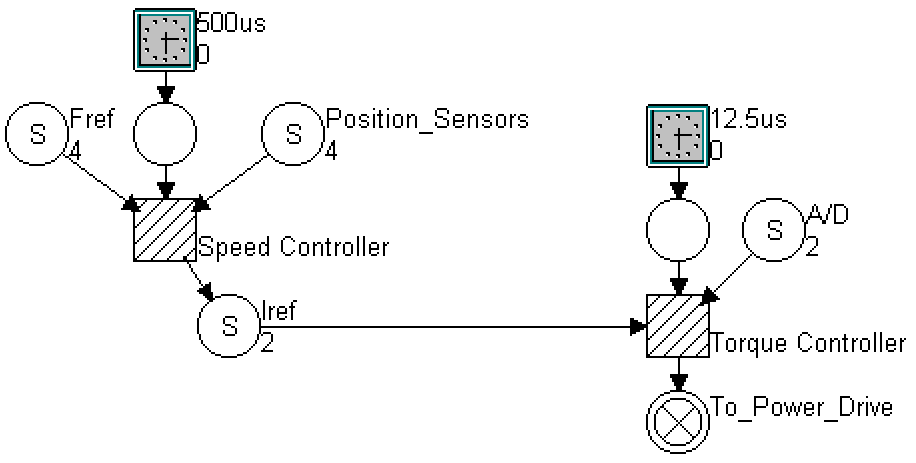


Figure 6.4. Motor Control Net

The Speed Controller and the Torque Controller are further refined in the subnets shown in Figure 6.5 and Figure 6.6. The Torque Controller has a further refinement, as shown in Figure 6.7. During system assessment, the specification can be fully or partially flattened and sent for profiling in the Cabernet environment.

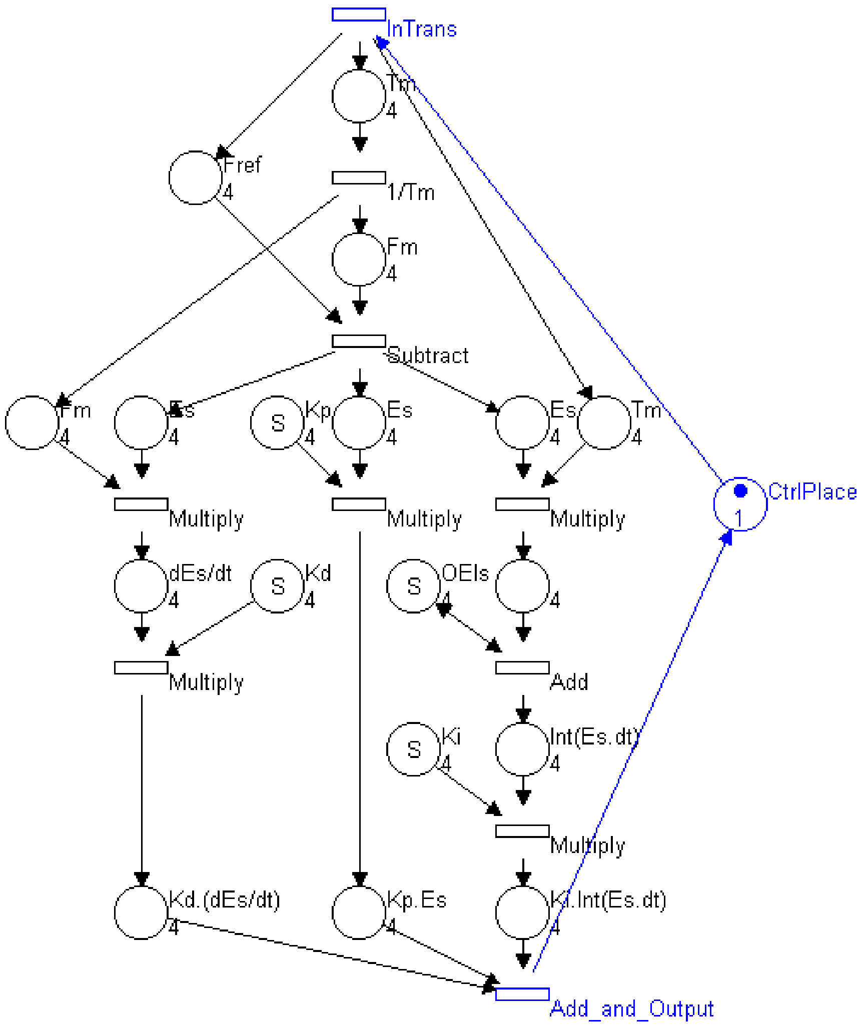
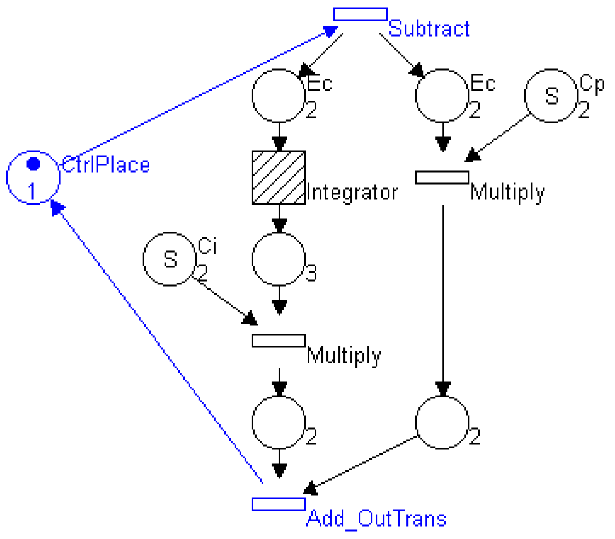


Figure 6.5. Speed Control Subnet



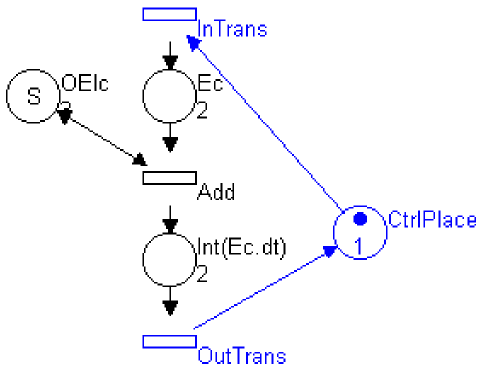


Figure 6.6. Torque Control Subnet **Figure 6.7. Integrator Subnet**

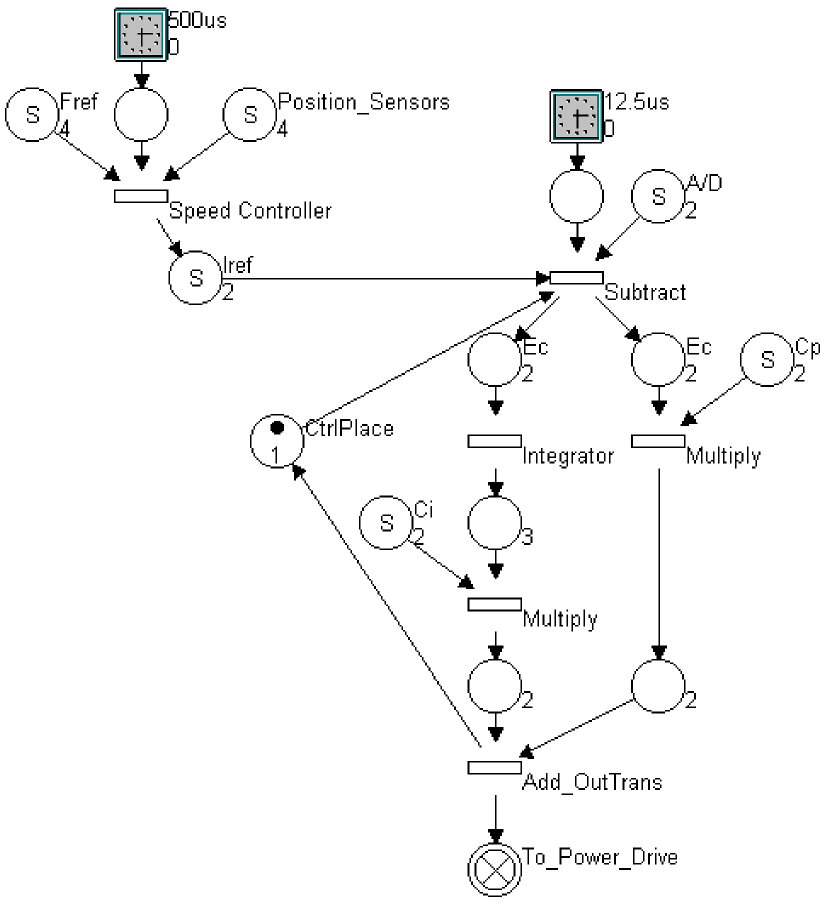


Figure 6.8. Motor flat net

Figure 6.8 shows a partially flattened version of the system where the Torque Controller subnet was replaced by its subnet specification, and the Speed Controller subnet and the integrator subnet were not flattened but only replaced by a simple transition.

During production of the output specification to the scheduler/partitioner, those sub-levels are abstracted and only the level shown in Figure 6.4 is of importance.

6.2.4 Partitioning

The partitioning based on the data shown in Table 6.4 and Figure 6.4 has successfully found a solution by moving both tasks to hardware processors. Independent of the partitioning parameter weights chosen both the Speed Control and the Torque Control are moved to hardware during pre-partitioning. This happened since $r'_i + c_i > d'_i$ for both tasks.

A second execution of the partitioner, this time using the *HwWCET* of both tasks and again allocating them to the same processor, shows that it would be possible to share the same hardware processor and still guarantee the timing constraints. In fact, even adding 50% of extra time to allow for context switch on the Torque Control algorithm, there is still 60μs of idle time in the processor, throughout the 500μs of scheduling period, which leaves 12% of spare processor utilisation. The use of a single hardware processor could be useful if reusability is possible between the operators used to implement both algorithms, which would reduce the area of ASIC produced in the design.

6.2.5 Operation Scheduling

An experiment of the use of the scheduler for operation-level scheduling was performed over the Speed Control algorithm. This sort of scheduling is useful not only for high-level synthesis of hardware but also to define precisely start and end times of tasks, e.g., for communication protocols. A slightly different Speed Control specification was produced for the operational-level scheduling, as seen in Figure 6.9.

Two possibilities are considered:

Case 1: A non-parallel implementation, with all operations sharing the same arithmetic and logic unit (*ALU1*). It is considered that each addition/subtraction takes one time unit and each multiplication/division takes two time units. Also, in this case, all operations are mutually excluded so that no pre-emption is allowed, which is normal for operation-level scheduling. This is the subject of study of several authors such as [31, 54, 77] although their methods are heuristic and as such cannot guarantee to find a feasible solution whenever one exists.

Case 2: A parallel implementation, with two ALUs, one for addition/subtraction (*ALU1*) and another for multiplication/division (*ALU2*). The execution time is considered here as before. All operations sharing a same processor mutually exclude each other. In this case the scheduler could be used as part of a high-level synthesis system where allocation and binding are performed by other tools.

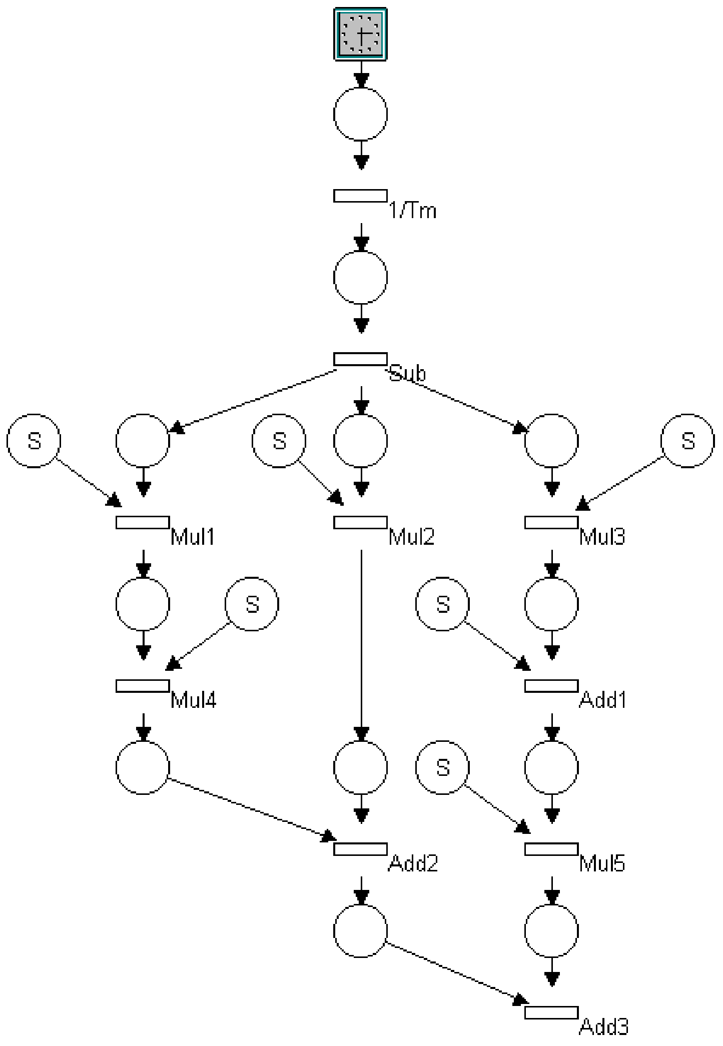


Figure 6.9. Speed Control specification for operation-level scheduling

6.2.5.1 Operation-Level Scheduling Results

Figure 6.10 and Figure 6.11 show the schedules produced. In Case 1, the minimum execution time is 16 time units, whereas in Case 2, this time goes down to 13 units.

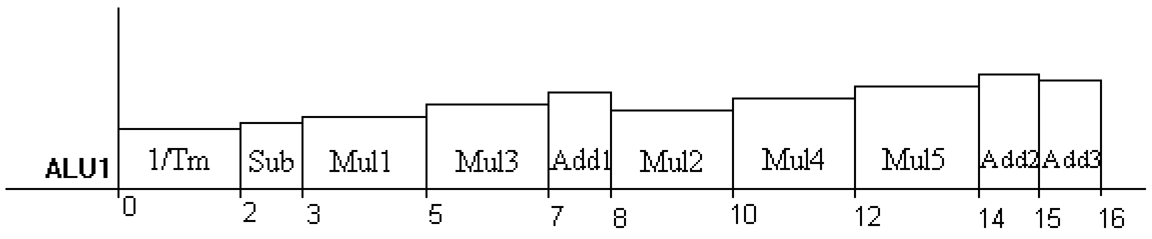


Figure 6.10. Schedule of Case 1

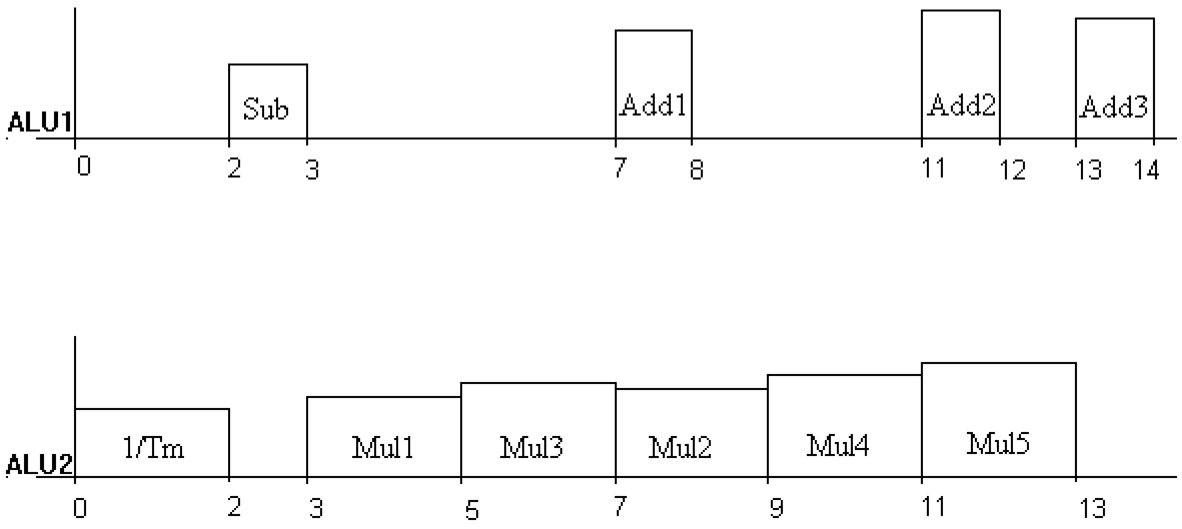


Figure 6.11. Schedule of Case 2

6.3 York1

This example is based on the work on task allocation at the University of York [115]. Its value to this dissertation is in the fact that it presents real-time constraints and it is not possible to execute in a single processor. Since its primary purpose was to show allocation problems, its original data would only exercise the system partitioning algorithm, and not the pre-partitioner. Therefore, modifications were made in other to allow for some activity during the pre-partitioning phase. Figure 6.12 shows the net specification of York1 (which is equal to York2) and Table 6.5 shows the timing specification relevant to York1 only.

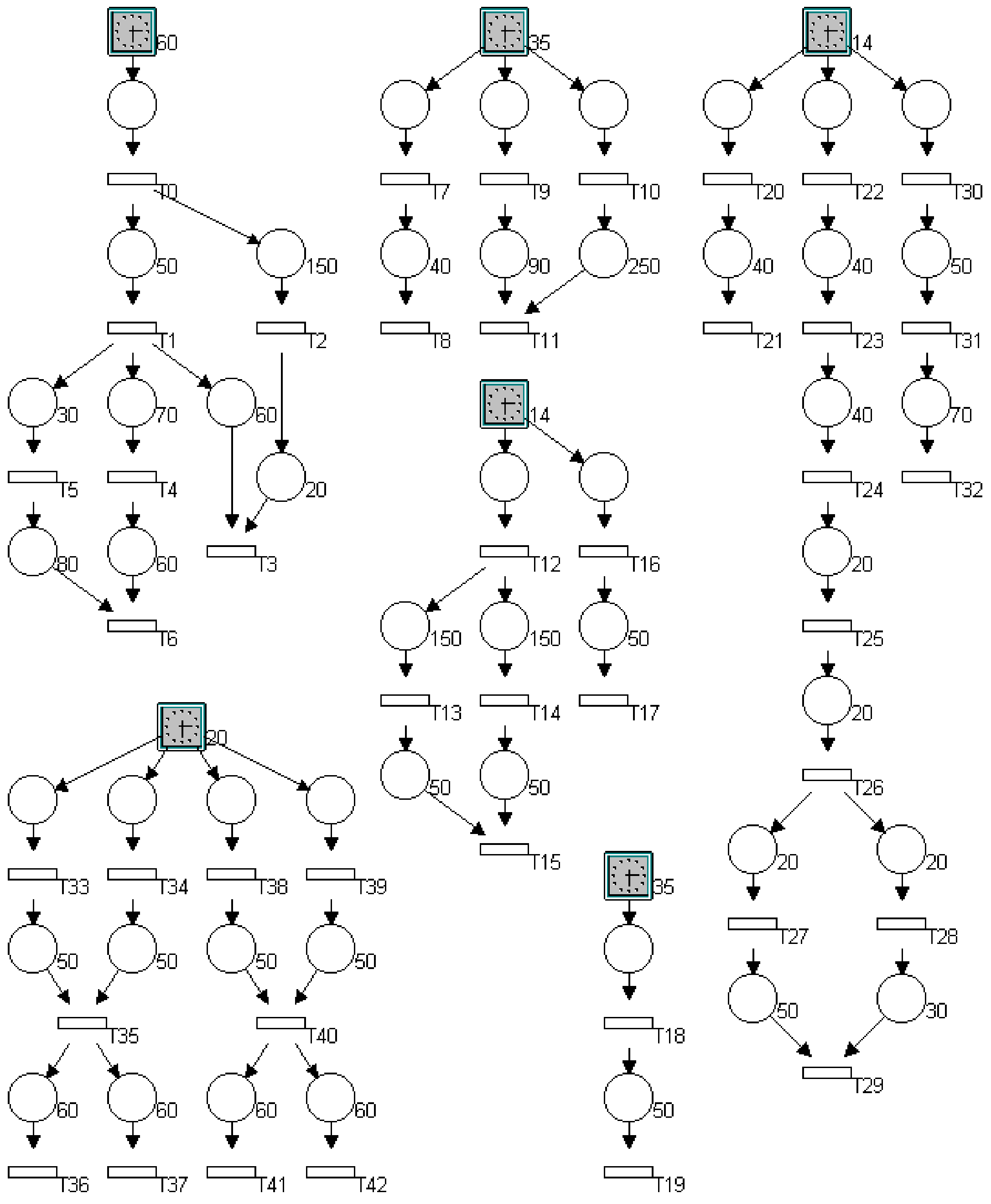


Figure 6.12. York1 and York2 Specification Net

Task	Period	SwWCET	HwWCET	μ	Messages
T0	60	4	4	sw1	50→1, 150→2
T1	60	4	4	sw1	60→3, 70→4, 30→5
T2	60	2	2	sw1	20→3
T3	60	2	2	sw1	
T4	60	2	2	sw1	60→6
T5	60	4	4	sw1	80→6
T6	60	12	6	sw1	
T7	35	2	2	sw1	40→8
T8	35	2	2	sw1	
T9	35	8	8	sw1	90→11
T10	35	28	14	sw1	250→11
T11	35	8	4	sw1	
T12	14	2	2	sw1	150→13, 150→14
T13	14	2	2	sw1	50→15
T14	14	2	2	sw1	50→15
T15	14	2	2	sw1	
T16	14	2	2	sw1	50→17
T17	14	2	2	sw1	
T18	35	1	1	sw1	50→19
T19	35	1	1	sw1	
T20	14	1	1	sw1	40→21
T21	14	2	2	sw1	
T22	14	1	1	sw1	40→23
T23	14	1	1	sw1	40→24
T24	14	1	1	sw1	20→25
T25	14	1	1	sw1	20→26
T26	14	2	2	sw1	20→27, 40→28
T27	14	1	1	sw1	50→29
T28	14	5	1	sw1	30→29
T29	14	4	1	sw1	
T30	14	1	1	sw1	50→31
T31	14	2	2	sw1	70→32
T32	14	2	2	sw1	
T33	20	3	3	sw1	50→35
T34	20	2	2	sw1	50→35
T35	20	2	2	sw1	60→36, 60→37
T36	20	2	2	sw1	
T37	20	2	2	sw1	
T38	20	3	3	sw1	50→40
T39	20	2	2	sw1	50→40
T40	20	2	2	sw1	60→41, 60→42
T41	20	2	2	sw1	
T42	20	2	2	sw1	

Table 6.5. York1 Timing and Architectural Specification

6.3.1 Partitioning Summary

The partitioning parameter Par4 provided the best solution in most of the experiments performed with the partitioner, and in this example as well. As such, the report presented here refers to all weights set to zero, but W4 which is set to one.

1. Scheduling period: 420
2. Number of temporal instances: 913
3. Twelve partitioning lists based on temporal instances of tasks {T10, T11}, and thirty based on instances of tasks {T22, T23, T24, T25, T26, T28, T29} were created during pre-partitioning.
4. Tasks T10 and T28 were chosen to move to hardware.
5. The scheduling of the root node proved to be unfeasible, presenting a lateness of 1446 time units, with a lower bound of the same value.
6. The partitioning set was created from which a minimum of 1442 time units needed to be removed.
7. The following tasks were chosen to move to hardware (in the order of removal by the automatic partitioner): T29, T9, T11, T6, T33, T38, T26, T12, T13, T14, T16, T31, T15, T17, T21, T32, T34, T39, T35, T40, T36, T37, T41, T42.
8. This move released a total of 1458 time units.
9. Another scheduling attempt based on this new configuration was successful in the root node. This solution presented a very small software host idle time of 12 units over the whole scheduling period (420 units), representing 97.1% of software host utilisation.

6.4 York2

Since all other examples in this chapter were solved in the root node, this example was created to show the branch-and-bound scheduling capabilities. The tasks are distributed over several processors and all have fixed implementation, i.e., cannot be moved to hardware (see Table 6.6). An idle time of only 10 units on the busiest processor (*swA*, 97,6% utilisation) provides a tough target to the scheduler.

Task	Period	SwWCET	μ	Task	Period	SwWCET	μ
T0	60	4	swA	T22	14	1	swN
T1	60	4	swB	T23	14	1	swN
T2	60	2	swC	T24	14	1	swN
T3	60	2	swD	T25	14	1	swN
T4	60	2	swE	T26	14	2	swN
T5	60	4	swA	T27	14	1	swN
T6	60	6	swA	T28	14	1	swK
T7	35	2	swF	T29	14	1	swN
T8	35	2	swD	T30	14	1	swM
T9	35	8	swA	T31	14	2	swL
T10	35	14	swA	T32	14	2	swM
T11	35	4	swA	T33	20	3	swO
T12	14	2	swG	T34	20	2	swI
T13	14	2	swG	T35	20	2	swL
T14	14	2	swH	T36	20	2	swM
T15	14	2	swI	T37	20	2	swP
T16	14	2	swH	T38	20	3	swP
T17	14	2	swI	T39	20	2	swD
T18	35	1	swJ	T40	20	2	swP
T19	35	1	swK	T41	20	2	swL
T20	14	1	swL	T42	20	2	swO
T21	14	2	swM				

Table 6.6. York2 Timing and Architectural Specification

6.4.1 Scheduling Results

The root node schedule presents a lateness of 2 with a lower bound of -6. A search tree containing 189 nodes is produced, which took around 20 minutes on an Intel Pentium®-based computer with 32 MB of RAM. The algorithm was not able to execute in a 16MB RAM computer, but the current implementation retains many data that are being used only for assessment of the algorithm itself, and not of the results.

6.5 Summary

This chapter presented four examples. The first was directed towards showing the specification method defined in Chapter 3. In particular, it presented some of the difficulties faced when using pre-runtime based methods in specification which comprise sporadic tasks with high responsiveness. Time Wizard was able to model the system from the initial specification through to the generation of a model suitable for scheduling/partitioning.

Hierarchy constructors were seen in action on the Motor Controller example. In particular, issues of processor reusability after partitioning and applicability of the scheduling algorithm in high-level synthesis of hardware were discussed. Techniques such as clique partitioning [84] should be able to help promote this reusability.

Examples York1 and York2 targeted mainly the partitioner and scheduler, respectively.

In all examples, Par4 produced the best results. This is explained by the fact that Par4 encompasses many of the other parameters in a single equation which attempts to use the best on each one. At the same time, the method used to chose tasks to move to hardware is based on a simple list partitioning. Although this technique is used in some co-design approaches [52], there are other methods which produce better results at the expense of processing power, such as simulated annealing [40, 42] and multi-stage clustering [8]. Nevertheless, the algorithms used to define candidates for partitioning, i.e., generation of partitioning lists and sets proved very efficient, only selecting segments which really could help improving the overall lateness when moved to hardware.

During the design the lack of a modularisation scheme was felt such as the approach shown in Section 3.3.5 for CP-nets. Also noticed was the absence of support for soft deadline tasks.