Towards Formal Specification and Analysis of Avionics Mission Systems *

Laure Petrucci†

LSV, CNRS UMR 8643
ENS de Cachan
61, avenue du Président Wilson
94235 CACHAN Cedex
France
Email: `petrucci@lsv.ens-cachan.fr`

Lars Michael Kristensen
Jonathan Billington

Computer Systems Engineering Centre
School of Electrical and Information Engineering
University of South Australia
Mawson Lakes Campus, SA 5095
Australia
Email: `Lars.Kristensen@unisa.edu.au`
`Jonathan.Billington@unisa.edu.au`

Zahid H. Qureshi

Aeronautical and Maritime Research Laboratory
Defence Science and Technology Organisation
Edinburgh, SA 5111
Australia
Email: `zahid.qureshi@dsto.defence.gov.au`

Abstract

The avionics mission system of an aircraft is a complex real-time system consisting of a mission control computer, sensors, displays, controls, and connecting data buses. The complexity and real-time requirements of avionics mission systems represent major challenges to the Australian Defence Force when upgrading and maintaining aircraft. This has motivated research into the application of formal techniques to investigate and predict the effects of upgrades to aircraft. This paper is based on a joint research project between the University of South Australia and the Australian Defence Science and Technology Organisation into the modelling and analysis of avionics mission systems. Coloured Petri Nets have been applied for modelling a representative generic avionics mission system, and state spaces have been used for analysis. The initial state space analysis has focused on the scheduling of tasks on the mission control computer and on input/output processing.

Keywords: Avionics mission systems, validation and verification, real-time systems, Coloured Petri Nets.

1 Introduction

Avionics system upgrades to aircraft platforms are labour intensive and require significant capital expenditure by the Australian Defence Force (ADF). The main reason for this is that changes to system components require substantial redesign and testing, and the difficulty of making predictable incremental changes to the avionics architecture because of the complex interaction between the components. The avionics mission system (Spitzer 1992) is a critical avionics system for military aircraft. It is a complex real-time system (Liu 2000), responsible for the tactical control of devices such as radars, sensors, and weapons of the aircraft during a mission. The tasks performed by the avionics mission system include data collection from sensors, fusion of collected data, display of information to pilots, and controlling devices in response to inputs from the pilots.

The generic architecture of an avionics mission system (Locke, Lucas & Goodenough 1990), depicted in figure 1, consists of a mission control computer, a serial data bus, and external devices such as navigation systems, radars and sensors. The control of devices, displays, and pilot controls, is handled by a collection of tasks executing on the mission control computer. A major concern when upgrading and maintaining avionics mission systems is the scheduling of tasks on the mission control computer, and the data transfer across the data busses connecting the mission control computer and the external devices. To ensure that the system is working properly, the scheduling must guarantee that the execution of tasks satisfies specific hard deadlines.

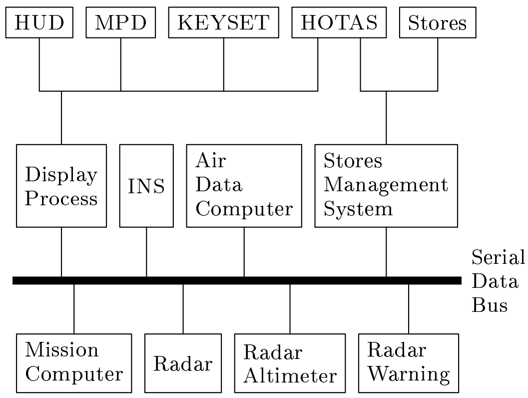

Figure 1: AMS architecture.

Scheduling of tasks in real-time systems has traditionally been conducted using purely algorithmic approaches (Liu 2000). Recently, there has been an increased interest in applying timed automata (Alur & Dill 1994) and model checking techniques to scheduling problems (Sun 2001, Amnell, Fersman, Mokrushin, Pettersson & Yi 2002, Behrmann, Fehnker, Hune, Larsen, Pettersson & Romijn 2001, Fehnker 1999). The basic idea is to turn the scheduling problem into a reachability problem that can be solved by analysis tools using a state space search. The advantages of formal modelling and state space methods in this setting is that the same model of the system can be used to analyse scheduling as well as other properties. Hence, it represents an integrated approach to analysis of the system. Moreover, the simplifying assumptions required by the algorithmic approaches are not needed when applying state space methods. Petri Nets have previously been used for modelling and analysis of real-time systems, in particular for ADA programs (see e.g. (Burns, Wellings, Burns, Koelmans, Koutny, Romanovsky & Yakovlev 2000, Shatz, Tu, Murata & Duri 1996)) and train systems (Berthelot & Petrucci 2001). Earlier work has however mainly focused on deadlock de-

*This work has been sponsored by the Australian Defence Science and Technology Organisation under contract no. 687237.

†The work of Laure Petrucci was done while visiting the University of South Australia.

tection and violations of safety properties. This paper presents our application of Timed Coloured Petri Nets (CP-nets or CPNs) (Jensen 1997, Kristensen, Christensen & Jensen 1998) and the supporting Design/CPN tool (*Design/CPN Online* 1996) for the scheduling and input/output processing analysis in avionics mission systems. This work is based on the initial modelling presented in (Qureshi, Billington & Kristensen 2001).

The paper is organised as follows. Section 2 briefly introduces the basic concepts of Coloured Petri Nets. Section 3 presents the CPN model of a generic avionics mission system, while section 4 presents our analysis approach and results. Finally, in section 5 we provide conclusions and discuss future work.

2 Coloured Petri Nets

This section briefly explains the basic ideas of CP-nets including the time concept and the hierarchy concept. A detailed treatment of CP-nets, including their formal semantics can be found in (Jensen 1997). A complete informal introduction to CP-nets can be found in (Kristensen et al. 1998). The informal introduction to CP-nets given in this section is based on (Kristensen et al. 1998), and can be skipped by readers already familiar with CP-nets.

2.1 Basic Concepts

CP-nets are a graphically oriented modelling language for concurrent and distributed systems. CP-nets are based on Petri nets (Desel & Reisig 1998, Murata 1989) and the functional programming language Standard ML (SML) (Ullman 1998). Petri nets provide the primitives for modelling concurrency and synchronisation, whereas SML provides the primitives for modelling data manipulation in systems and for creating compact and parameterisable CPN models. Construction and analysis of CP-nets are supported by the Design/CPN tool (*Design/CPN Online* 1996) and the emerging CPN Tools (*CPN Tools* 2001).

A CPN model of a system describes the states in which the system may be and the transitions between these states. The states of a system are modelled by means of *places* (which are drawn as ellipses or circles). A state of a CPN model is called a *marking*, and consists of a distribution of *tokens* on the places of the CPN model. Tokens in a CPN model are data values (they are coloured). This is in contrast to ordinary Petri nets, where tokens are black dots. Each place has an associated *colour set* (type) determining the kind of tokens that can reside on the place. By convention, the type of a place is written to the lower left or right of the place. The colour set of a place is an SML type, and may be complex, e.g. a record, union, or list.

The marking of a place is, in general, a *multi-set* of tokens. A multi-set is similar to a set, except that there may be several appearances of the same element. This means that a place may have several tokens carrying the same value. A CPN model has a distinguished marking – the *initial marking*, which is used to describe the initial state of the system. The initial marking of a place is, by convention, written on the upper left or right of the place.

The actions of a CPN model are represented by means of *transitions* (which are drawn as rectangles). Transitions and places are connected by *arcs*. The *input places* of a transition are the places connected to the transition by incoming arcs, while the *output places* are those connected by outgoing arcs. The execution of a CPN model consists of *occurrences* of *enabled* transitions. An occurrence of a transition removes tokens from the input places, and adds tokens to the output places, thereby changing the marking (state) of the CPN model. The exact number of tokens added and removed by the occurrence of a transition, and their values are determined by the *arc expressions*. Arc expressions are expressions written in SML and positioned next to the arcs.

Data values need to be assigned to the (free) *variables* occurring in the arc expressions on the surrounding arcs of the transition. Otherwise, it is impossible to *evaluate* the arcs expressions. Variables can be assigned data values belonging to the type of the variable. This means that transitions may occur in different modes depending on the assignment of data values to the variables of the transition. An assignment of data values to the variables of a transition is called a *binding*. In addition to the arc expressions, it is possible to attach a boolean expression (with variables) to each transition. The boolean expression is called a *guard*. It specifies that only bindings for which the boolean expression evaluates to true are accepted.

When a transition *occurs*, the multi-set of tokens removed from an input place is determined by evaluating the arc expression on the corresponding input arc. Similarly, the multi-set of tokens added to an output place is determined by evaluating the arc expression on the corresponding output arc.

2.2 Time Concept

The time concept of CP-nets is discrete and based on the introduction of a *global clock*. The clock values represent *model time*, and they may either be integers or reals. In addition to its value, a token may include a time value, called a *time stamp*. Intuitively, the time stamp describes the earliest model time at which the token can be used, i.e. removed by the occurrence of an enabled transition.

In a timed CP-net a binding element is said to be *colour enabled* when it satisfies the enabling rule for untimed CP-nets (i.e. when the required tokens are present at the input places and the guard of the transition – if present – evaluates to true). A binding element is a pair consisting of a transition and an assignment of data values to the variables of the transition. However, to be enabled in a timed CPN model, the binding element must also be *ready*. This means that the time stamps of the tokens to be removed must be less than or equal to the current model time.

The marking of places with a *timed colour set (type)* is a *timed multi-set* of token values. A timed multi-set is similar to a multi-set, except that each member of the multi-set carries a time stamp. To model that an activity/operation takes r time units, we can let the corresponding transition create time stamps for its output tokens that are r time units larger than the clock value at which the transition occurs. This implies that the tokens produced are unavailable for r time units.

The execution of a timed CP-net is time driven, and it works in a way similar to that of event queues found in many languages for discrete event simulation. The system remains at a given model time as long as there are colour enabled binding elements that are ready for execution. When no more binding elements can be executed, at the current model time, the system advances the clock to the next model time at which binding elements can be executed. Each marking exists in a closed interval of model time (which may be a point, i.e. a single moment).

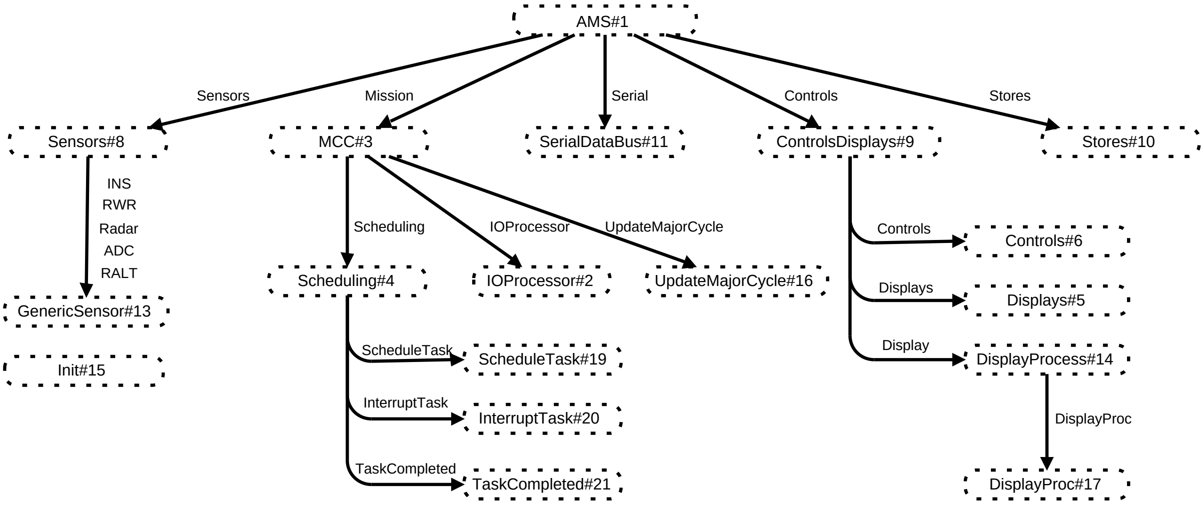

Figure 2: The hierarchy page.

2.3 Hierarchy Concept

The basic idea underlying the hierarchy concept of CP-nets is to allow the modeller to construct a large model by using a number of small CPN models. These smaller CPN models are called *pages*. In a hierarchical CP-net, it is possible to relate a *substitution transition* (and its surrounding places) to a separate CP-net called a *subpage*. A subpage provides a more precise and detailed description of the activity represented by the substitution transition. Each subpage has a number of *port places* (tagged with a boxed P) and they constitute the interface through which the subpage communicates with its surroundings. The relationship between a substitution transition and its subpage is specified by describing how the port places of the subpage are related to the *socket places* of the substitution transition. This is achieved by providing a *port assignment*. When a port place is assigned to a socket place, the two places become identical. The port place and the socket place are just two different representations of a single conceptual place. More specifically, this means that the port and the socket places always have identical markings. When a socket place receives a token from the surroundings of the substitution transition that token also becomes available at the assigned port place of the subpage, and hence the token can be used by the transitions on the subpage. Analogously, the subpage may produce tokens on a port place. Such tokens are also available at the corresponding socket place and hence they can be used by the surroundings of the substitution transition.

It should be noted that substitution transitions never become enabled and never occur. Substitution transitions work as a macro mechanism. They allow subpages to be conceptually inserted at the position of the substitution transitions – without doing an explicit insertion in the model.

3 CPN AMS Model

In this section, we give the necessary background on avionics mission systems, and show how a generic avionics mission system (Locke et al. 1990) has been modelled by means of a Timed CP-net.

3.1 Overview

The *hierarchy page* of the CPN model is depicted in Figure 2. This page shows the overall organisation of the modules constituting the CPN model. Each module of the CPN model is represented by a node on the hierarchy page. An arrow from a node to another node indicates that the latter is a subpage (submodule) of the former. The mechanism linking two pages will be illustrated with the scheduling part of the `Mission Control Computer` in Section 3.2. The full CPN model of the generic avionics mission system is rather complex, and we can only present selected parts of the CPN model in this paper.

The core of an `Avionics Mission System` (AMS) is the `Mission Control Computer`, which is responsible for executing a set of tasks. The `Mission Control Computer` communicates with the `Sensors`, `Controls`, `Displays` and `Stores` of the aircraft via a `Serial Data Bus`. All of these parts of the AMS are represented in figure 2. A task can make input/output requests via an `IOProcessor` which transmits the requests to the `Serial Data Bus`. The `Serial Data Bus` then passes the request to the appropriate external device (e.g. the radar). It waits for the information to be transferred (e.g. radar result) and passes it back to the `IOprocessor`. When all the input/output requests for a task have been served, the `Mission Control Computer` can resume the execution of the task.

3.2 The Mission Control Computer

The mission control computer (MCC) is the core of an avionics mission system. Several tasks on the MCC have to be performed periodically to e.g. update displays with vital information for the pilot and operators. These tasks are scheduled and executed by the MCC according to two sorts of time slots: the *major cycle* and the *minor cycle*. The minor cycle is a subdivision of a major cycle. One can also distinguish two types of tasks that are scheduled differently: background tasks and rategroup tasks. The execution of a background task and a rategroup task within a major cycle is illustrated in Figure 3.

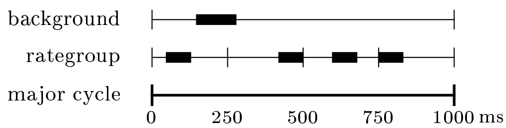

Figure 3: Execution of background and rategroup tasks.

Background tasks have to be executed once within

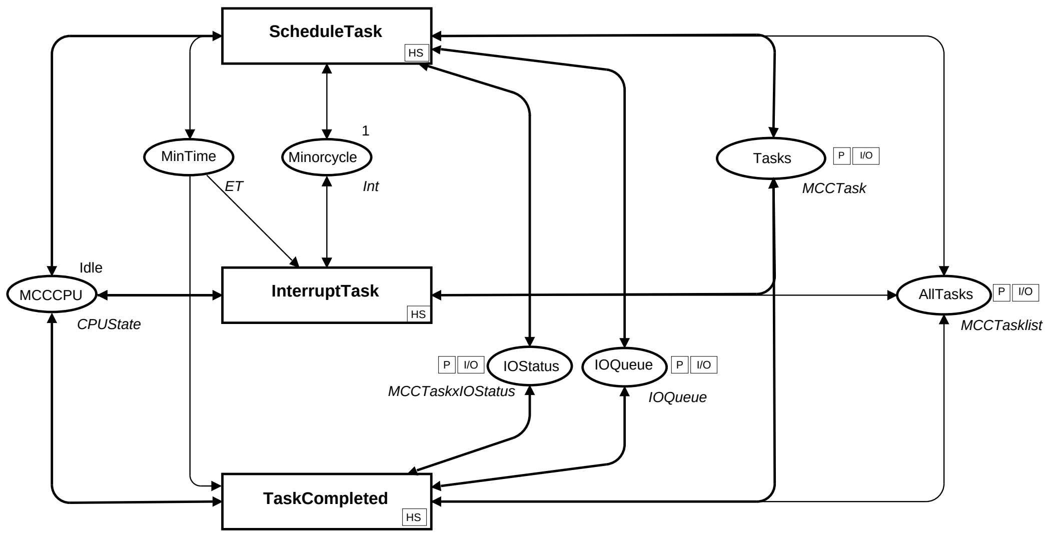

Figure 4: The `Scheduling` page.

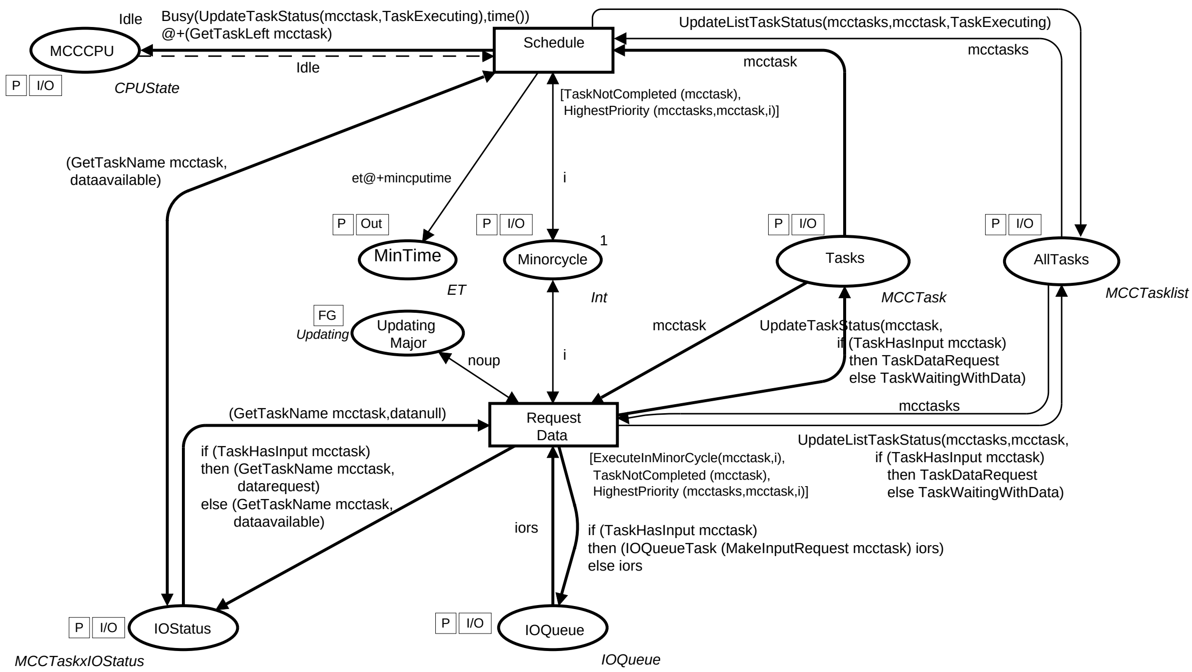

Figure 5: The `ScheduleTask` page.

each major cycle and are required to complete before the beginning of the next major cycle.

Rategroup tasks occur periodically. They occur within a time-slot, the value of which is task-dependent. The time slot for each rategroup task is chosen so that the major cycle is a multiple of each rategroup time slot. The task must execute once within each time slot. Thus, each rategroup task must execute several times within a single major cycle. It must be completed before the time when it is to be executed again. As a rategroup time slot is a subdivision of the major cycle, the execution of a rategroup task cannot overlap two major cycles. The rategroup task in Figure 3 has on average a period of 250 milliseconds while a major cycle lasts 1000 milliseconds.

When it is time for tasks to start their execution, they are scheduled by the MCC. Since we do not have access to the detailed implementation of tasks, we assume that tasks request their input (e.g. from sensors and radars) at the beginning of their execution, and send their output (e.g. to displays) at the end of their execution. The input requests that have to be performed before execution can start are passed by the Input/Output (I/O) processor to the serial data bus. When all the information required has been received via the I/O processor, the task can be executed by the MCC. When its execution is complete, output can be performed. Then the task must wait until it has to be rescheduled.

3.3 Task Scheduling and Execution

Page `Scheduling`, presented in Figure 4, models the general scheduling mechanism on the MCC. It consists of seven places and three *substitution transitions*. Each substitution transition has an associated subpage (generally having the same name) which describes the behaviour of the activity represented by the substitution transition. Place `MCCCPU` is used to model the state of the `MCC`'s processor. It is either `Idle` or `Busy` executing a task. Place `Minorcycle` contains the number of the current minor cycle. `MinTime` represents the minimum amount of time a task should spend on the CPU before it can be preempted by a task with a higher priority. Place `Tasks` contains all the possible tasks and `AllTasks` a list of all possible tasks. These two places allow us to deal with complex functions such as priorities in a simple way. Finally, place `IOstatus` models the status of tasks w.r.t. input/output and `IOQueue` is used for modelling the input/output request queue. Page `InterruptTask`, which is the subpage associated with substitution transition `InterruptTask`, models how a task executing on the CPU can be interrupted by a task with a higher priority. Page `TaskCompleted` describes the completion of a task. We will not go into further detail with these two pages in this paper.

The `ScheduleTask` page, shown in Figure 5, models the part of the scheduling mechanism responsible for starting tasks and executing them on the CPU. It consists of two main operations modelled by the following transitions.

`RequestData` allows a task with the highest priority which has to start at the current time to request its data. The priority criteria can be of various types, e.g. it may depend on the type of task (background or rategroup), the current time and the number of tasks already queued. Note that `HighestPriority` (used in the guard) is a SML function operating on the list of all tasks. Hence, changing the priority criteria is done by changing only this function. This is extremely useful when testing different scheduling strategies.

`Schedule` selects the task with the highest priority which already has its data, and puts it on the CPU for execution.

When a task is put on the CPU for execution, the transition `Schedule` occurs. The token corresponding to the task is removed from place `Tasks` and added to place `MCCCPU`. Tasks are modelled as a token of a complex structured colour set (type) named `MCCTask`. The attributes of a task include information about its type (periodic or background), the size of the task in terms of the time its takes to execute the task, input/output requests made by the task when it executes, and how much time is left for the task to complete. When the `Schedule` transition occurs, the token put on place `MCCCPU` is given a timestamp which is the model time at which `Schedule` occurs plus the remaining execution time of the task. This is done by using the @+ operator in the expression of the arc from transition `Schedule` to place `MCCCPU`. Thus, unless it is preempted by a higher-priority task, the task will run to completion. The attributes of the task are also updated to record when it was put on the CPU. This information is required later if the task becomes preempted – in which case the remaining execution time of the task needs to be updated according to the difference between the time it was put on the CPU and the time it was preempted.

3.4 Serial Data Bus

The `Serial Data Bus` page shown in Figure 6 comprises three places and two transitions. There are two places describing the state of the `Serial Data Bus`, `Idle` and `Busy`. It is initially `Idle`. The third place, `SDB`, represents the information transiting on the `Serial Data Bus`.

When a request arrives on the `SDB` place from the I/O processor, it starts transmitting the request to the appropriate device (transition `Start Transmit` occurs). When the request has been transmitted, the `Serial Data Bus` becomes `Idle` again and signals that the request has been completed so that the I/O processor handles the next request (transition `Transmit Complete`). Places `Idle` and `Busy` ensure that a transfer on the data bus can start only if no

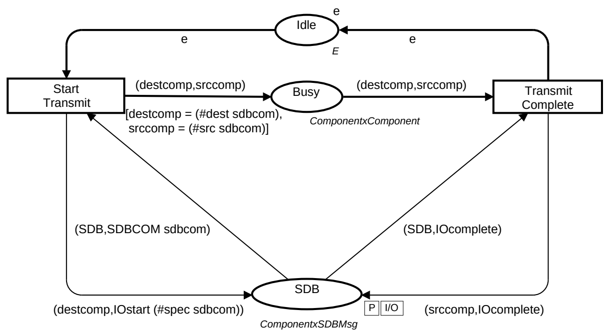

Figure 6: The `Serial Data Bus` page.

other transfer is already being processed.

3.5 External Devices

The information necessary for tasks is gathered by external devices such as radars. The execution of tasks causes data to be transferred between sensors, controls, displays, and stores. The interaction between the MCC and the external devices consists of data transfers. We model the avionics mission system at a level of abstraction where there is no need to distinguish the different kinds of external devices. Hence we create the `Generic Sensor` page shown in Figure 7.

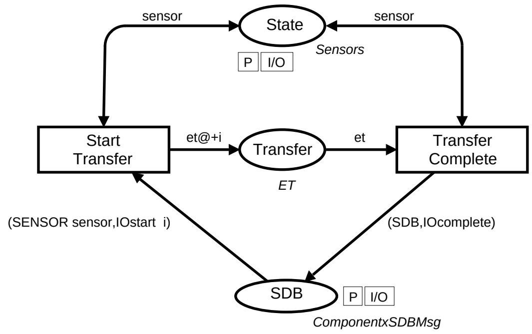

Figure 7: The `GenericSensor` page.

When a request arrives on the `Serial Data Bus`, the relevant `Sensor` is activated and the data transfer starts (Transition `Start Transfer` occurs). Each sensor takes a certain amount of time to operate, and thus must remain in `Transfer` all that time. This is achieved by the time stamp given to the token created in place `Transfer` when transition `Start Transfer` occurs. The token in place `Transfer` cannot be consumed before this amount of time has elapsed. When the sensor has terminated, i.e. transmitted all its information, it signals the `Serial Data Bus` that the request has been completed.

4 Analysis

Our primary goal is to find whether the set of tasks can be scheduled on the mission control computer so that all tasks are completed in due time. As the execution of tasks does not overlap major cycles, all major cycles have similar behaviour. We therefore consider only a single major cycle. To investigate this property, we generate the state space of the CP-net using Design/CPN. Our analysis attempts to discover a schedule of tasks such that all tasks meet their deadlines, i.e. background tasks are fully executed before the next major cycle starts, and rategroup tasks are fully executed before the next time they are to start execution. Such a schedule can then be repeated in the next major cycles. For the analysis

```
Rategroup {name = TargetTracking,      /* name of task */
           rate = 40,                  /* rate at which task occurs */
           next = 40,                  /* next execution time */
           status = TaskIdle,          /* task status */
           size = 4,                   /* length of execution on CPU */
           left = 4}                   /* execution time left */

Background {name = ChangeDisplayMode, /* name of task */
            exeinmajor = false,       /* true if the task has been executed
                                         during the current major cycle */
            status = TaskIdle,        /* task status */
            priority = 8,             /* priority among background tasks */
            size = 1,                 /* length of execution on CPU */
            left = 1}                 /* execution time left */
```

Figure 8: Specification of tasks.

```
fun TaskInputs TargetTracking = [(DISPLAYPROC (CONTROL HOTAS),1),
                                 (SENSOR RADAR,1)]
|   TaskInputs ChangeDisplayMode = [(DISPLAYPROC (DISPLAY MPD),1),
                                    (SENSOR RADAR,2),
                                    (SENSOR RWR,1)];

fun TaskOutputs TargetTracking = [(DISPLAYPROC (DISPLAY HUD),1)]
|   TaskOutputs ChangeDisplayMode = [(DISPLAYPROC (DISPLAY MPD),1)];
```

Figure 9: Input/output of tasks.

presented in this paper, we considered a major cycle with a length of 1000 milliseconds, while a minor cycle lasts 1 millisecond.

We use DESIGN/CPN facilities in order to generate the state space for only one major cycle. DESIGN/CPN allows the user to provide *branching* and *stop* criteria to control state space generation. The stop criteria can be used to specify that the generation should stop when a given state predicate is satisfied. Similarly, a branching option can specify a predicate that a node should satisfy to have its successor states computed. The use of branching and stop options can be combined. In particular, we used the following options:

- A branching option such that only the nodes within the first major cycle are further developed. This effectively allows us to search the state space covering one major cycle.
- We consider a task to be *late* if it is a background task and it is not completed when a new major cycle starts, or it is a rategroup task which should already have been restarted. This notion is extended to *late nodes* which contain at least one late task. Late nodes violate our scheduling goal. We thus only construct successors of non-late nodes.
- A stop option used to halt the state space generation as soon as a schedule is found. In the state space, a *schedule* is a *path* leading from the initial state to a state where the major cycle is completed and no task is late.

To inspect the state space, we designed several *query functions*:

- `OGMaxTime` computes the maximum creation time of a state in the state space. This function helps us to decide whether it is worth continuing the construction or not. For example, after a lengthy state space generation, if the time is close to a full major cycle, we would continue the construction, suspecting that a result can be obtained. If, on the contrary, the time is still small, we can guess that we will run out of memory. Then we would not pursue the construction.
- `AllExecutedTasks` and `ExecutedTasks` allow us to find all the tasks that can be put on the Mission Control Computer. It is used to detect tasks that can never be executed.
- `OutputLateNodes` records the late nodes with their creation time and the late task. This shows which tasks may cause problems and when.

4.1 Parameters

To make analysis feasible, we start by selecting a small set of tasks and gradually introduce additional tasks. The tasks we consider are described in (Locke et al. 1990). Those we report on in this paper are summarised in Table 1. These sets were chosen for the consistency of the tasks they contain. The `RG` column gives the number of rategroup tasks in the task set, and `BG` indicates the number of background tasks in the set.

The sets of tasks are loaded from files during an initialisation phase which constructs the relevant initial state of the generic avionics mission system.

Figure 8 shows the specification of both rategroup and background tasks for the two tasks `TargetTracking` and `ChangeDisplayMode`. The execution times specified in the tasks' descriptions are given as a number of minor cycles. The priority of background tasks is a number allowing us to (partially) order them.

The tasks' I/O is specified by listing, for each task, the devices to/from which data is being transferred. An example is shown in Figure 9. Function `TaskInputs` gives the input for each task. Similarly, function `TaskOutputs` gives the output for each task. For each device the size of the data transfer measured in milliseconds is specified.

As we will see when discussing the analysis results, we consider several policies for I/O queueing as well as for priorities among rategroup and background tasks. Each of these is parameterised by setting a variable in the input files, which can take different values:

rategroupqueue is either `Fifo` or `Overtake` where: `Fifo` means that all tasks, i.e. both rategroup

Task set	Tasks	RG	BG
S1	**Displays and Controls** HUDDisplay, MPDHUDDisplay, MPDTacticalDisplay, MPDButtonResponse, ChangeDisplayMode, MPDStoresDisplay, MPDStatusDisplay, KeysetResponse, HOTASDesignation, HOTASBombButton	6	4
S2	**S1 + built-in test** PeriodicBIT, BITFailureWarning, InitiatedBIT	7	6
S3	**S2 + radar control** RadarSearch, RadarTracking, RadarInitiateTracking	9	7
S4	**S3 + targeting** DesignateTarget, ConfirmDesignation, TargetTracking, TargetSweetening	10	10
S5	**S2 + threat response** PollRWR, ThreatResponseDisplay	8	7
S6	**S5 + RWR control** RWRProgramInput, RWRProgramming	9	8
S7	**S6 + weapon control (except WeaponRelease)** InputWeaponSelection, WeaponSelectionProc, AutoCCIPtoggle, WeaponTrajectory, ReinitiateTrajectory	10	12
S8	**S7 + targeting** DesignateTarget, ConfirmDesignation, TargetTracking, TargetSweetening	11	15

Table 1: The different sets of tasks used for testing.

and background tasks queue in a first in, first out manner. `Overtake` means that I/O requests from rategroup tasks have higher priority than I/O requests from background tasks.

backgroundpolicy is either `Nopriority` or `Withpriority`. `Nopriority` means that all background tasks have the same priority to enter the I/O queue. `Withpriority` means that the priority field of the background task is taken into account. Moreover, only one background task is allowed to queue for I/O at a time.

rategrouppolicy is either `Nopriority` or `Withpriority`. `Nopriority` indicates that all rategroup tasks have the same priority to enter the I/O queue. `Withpriority` means that a priority among rategroup tasks is defined. The rategroup task with the lowest rate has the highest priority as it must occur more often (recall that the rate specifies the number of minor cycles during which an execution must take place, and the periodicity, as shown on figure 3). If two tasks have the same rate, the one with less input will have the higher priority so that the execution can start on the CPU as early as possible.

In the following, the parameters used are written in the form: `(rategroupqueue, rategrouppolicy, backgroundpolicy)`. When *scheduling a task on the CPU*, the rategroup tasks will have the highest priority. Among rategroup tasks, those with the smallest rate have the highest priority. As background tasks all have to terminate before the end of the major cycle, they have the same priority to be put on the CPU. The *preemption policy* is such that a rategroup task which has its data available can interrupt a background task on the CPU. No other interruption is permitted.

4.2 Initial Results

In this section, we give the size of the state space for different cases, which motivate the introduction of our policies. Results were first obtained with few tasks: the rategroup task `WeaponRelease` cannot execute in time. It fails, i.e. becomes late during its first execution. This was discovered using the inspection function `OutputLateNodes`. Thus, this task has to be discarded and is therefore not included in Table 1. When a task set does not appear in a table, it means

Set	# Nodes	# Arcs	I/O SS	I/O SP
S1	394,690	671,414	14	10
S2	439,642	757,030	25	19

Table 2: State spaces (Fifo, Nopriority, Nopriority).

Set	# Nodes	# Arcs	I/O SS	I/O SP
S1	388,686	661,265	12	6
S2	390,687	664,776	18	17

Table 3: State spaces (Fifo, Nopriority, Withpriority).

that the state space could not be generated in a reasonable amount of time given the available computing resources.

When there is no priority policy for background tasks, they all compete for queueing at the beginning of a major cycle. Thus there is a huge amount of interleaving at the very start. Moreover, the rategroup tasks, that do not start in the first minor cycle, are queued after all background tasks, and are thus delayed. This delay means that most rategroup tasks become late, and no schedule exists. To avoid this, we introduced priorities among the background tasks, so that they queue one at a time. Comparing Tables 2 and 3 allows us to see the influence of the priority policy for background tasks. In the particular case of our sets of tasks, the difference is rather small. This is due to the low amount of background tasks involved. When there are more background tasks the difference between results with and without this policy increases.

The tables list the number of nodes and arcs in the state space, the maximum length of the I/O request queue (`I/O SS`) observed in the state space, and the length of the I/O request queue (`I/O SP`) along a path in the state space corresponding to a legal schedule for the tasks. In some tables, the number of arcs and the length of a schedule are missing. In those cases, the state space was so large that, in order to evaluate its size in terms of nodes, we had to discard the storage of arcs.

The rategroup tasks must execute in a given amount of time, which is smaller than for background tasks. Therefore, they are in some sense more urgent and should be served first. Comparing Tables 3 and 4, we can see the influence of this queueing policy for rategroup tasks on the size of the state space.

Set	# Nodes	# Arcs	I/O SS	I/O SP
S1	388,726	661,305	12	6
S2	390,489	664,452	18	17
S5	1,193,645		18	
S6	1,292,248		18	

Table 4: State spaces (Overtake, Nopriority, Withpriority).

Set	# Nodes	# Arcs	I/O SS	I/O SP
S1	77,982	127,316	12	6
S2	78,734	128,715	18	17
S3	485,054	811,734	18	17
S5	144,780	235,769	18	17
S6	142,022	234,257	18	17
S7	409,888	702,831	18	17

Table 5: State spaces (Overtake, Withpriority, Withpriority).

We still face state space explosion due to the interleaving between rategroup tasks. Therefore, we have added priorities among rategroup tasks to reduce the size of the state space while keeping the execution which has highest chance to succeed. Comparing Tables 4 and 5, we can see the influence of this priority policy for rategroup tasks on the size of the state space.

4.3 Depth-First Search

Although the use of priorities reduces the size of the state space, thus enhancing our analysis possibilities, we cannot handle yet as many tasks as are needed. As we are interested in the existence of a correct schedule, finding one satisfactory path is sufficient (as long as we are not interested in finding the "best" schedule). Therefore, we construct the state space using a depth-first generation. Depth-first search is an option normally not considered for timed systems as the state space of a timed CPN model is usually infinite due to increasing global time. However, in our case, we are only interested in a single major cycle. Thus the branching criterion will avoid the generation of an infinite path. As depth-first search is not available for timed systems within DESIGN/CPN, we had to implement it.

Set	# Nodes	# Arcs	I/O SS	I/O SP
S1	4,370	4,369	7	6
S2	4,514	4,513	17	14
S3	5,694	5,693	17	14
S4	1,299,299		17	
S5	4,798	4,797	17	14
S6	5,044	5,043	17	15
S7	5,661	5,660	17	15
S8	6,845	6,844	17	15

Table 6: Depth-First search (Overtake, Nopriority, Withpriority).

As we can see from the results in Tables 6 and 7, depth-first generation proved very useful. A schedule is found very quickly in all cases but one: a few thousand nodes are generated instead of hundreds of thousands or even millions. The advantage of using depth-first exploration is that we try to go as far as possible within a path. If the path is unsatisfactory, we backtrack. The disadvantage is that if an inappropriate choice is made at a very early stage, all of its successors are generated.

Set	# Nodes	# Arcs	I/O SS	I/O SP
S1	4,120	4,119	7	6
S2	4,263	4,262	17	14
S3	5,394	5,393	17	14
S4	6,483	6,482	17	15
S5	4,497	4,496	17	14
S6	4,643	4,642	17	15
S7	5,109	5,108	17	15
S8	6,204	6,203	17	15

Table 7: Depth-First search (Overtake, Withpriority, Withpriority).

4.4 Strategies for Further Analysis

We have presented our initial experiments with conducting state space analysis of the CPN AMS model. To alleviate the state explosion problem there are still several options available. It can be by modifying the model:

- use priorities among rategroup tasks to access the CPU. For the moment, rategroup tasks have priority over background tasks to access the CPU. But they have the same priority when accessing the CPU. A first approach would be to give priorities to rategroup tasks based on the rate, then consider the execution time one the CPU, and the output length.
- enhance the preemption policy. This is similar to the CPU access. The preemption takes place when a task is executing on the CPU and another one, with a higher priority, has its data available. The task which is executing is then removed from the CPU and put back in the set of tasks competing for access to the CPU. A task with the highest priority is then put on the CPU. For the moment, we only considered preemption of rategroup tasks over background tasks.
- study the allocation of the background tasks' priorities. The background tasks have in their description a priority that allows them to be (partially) ordered. Strategies in the allocation of these priorities would certainly enhance the results.
- design new priority strategies. As all background tasks must be completed at the same time (i.e. the end of a major cycle), it can be argued that they should have the same priority. However, if the task which uses the CPU less starts first, the chance of success increases. Other similar strategies could be designed.

We can also try to directly alleviate the state space explosion, using dedicated state space reduction techniques:

- abandon hopeless cases earlier. Currently, we stop exploring a branch in the state space when we find that the node contains a late task. However, this task might be found in advance. If a task has more time left to execute (and output data) than is left before it is due, there is absolutely no chance of obtaining a schedule. Hence, the exploration can already be cut-off at that node.
- in order to investigate several possible schedules, we can use the I/O queue length obtained with the depth-first search to restrict the state space generation, thus avoiding the computation when the queue length is large. Such a schedule would typically not be desirable.

- use equivalences between tasks. Two tasks would be equivalent if they have the same nature and the same rate in case of rategroup tasks, and also the same input, execution and output durations. A few tasks in our full set are equivalent to another one. Modelling all tasks of an equivalence class by several occurrences of a single task would reduce the state space.
- push the limits of computation using the sweep-line technique (Christensen, Kristensen & Mailund 2001*b*). It allows to compute a huge state space by forgetting the states that have already been explored and will never be visited again. As we have a model which progresses in time, we could easily apply this method. However, currently, the width of the state space is too large to gain from this.
- apply the state space caching method (Godefroid 1996) to search for a schedule. This method is based on depth-first generation of the state space, and may delete states which are not currently on the depth-first search stack. This may significantly reduce memory requirements. The open question is however whether the time penalty caused by multiple exploration of states will become problematic.

5 Conclusion and Future Work

We have presented our CPN modelling approach to avionics mission systems. The CPN model incorporates the key components of an avionics mission system, i.e. mission control computer, the data bus, sensors, and displays. We have demonstrated how analysis of scheduling and input/output queue of an avionics mission system can be done using state spaces. A virtue of our modelling approach is that the CPN model is highly parametric. As illustrated in this paper, this makes it easy to analyse different sets of tasks and in this way investigate the impact of adding tasks to the mission control computer. Another advantage of our modelling approach is that specific scheduling mechanisms can easily be changed, to investigate their impact. Our CPN model currently does not incorporate any notion of environment capturing external events (such as pilot behaviour) which also affects the execution of tasks. This will require further development of the CPN model.

Our analysis focuses on task scheduling of the mission control computer and only considers a single major cycle. This is sufficient because all tasks are required to have been executed at the end of a major cycle. Once we incorporate the modelling of an external environment into the CPN model, considering one major cycle may no longer be sufficient, since the execution of tasks triggered by external events may now cross over major cycle boundaries. The main problem to overcome here will be the increasing global clock of timed CP-nets. This increasing global clock implies that the state space of the CPN AMS model is infinite unless the number of major cycles is bounded. In (Christensen, Kristensen & Mailund 2001*a*) it was shown how to factor out the global notion of time in the state space of timed CP-nets. Hence, (Christensen et al. 2001*a*) may provide a way of lifting the restriction of considering a bounded number of major cycles.

The input/output processing has also been considered by looking at the size of the I/O queue. This could be achieved by an inspection of the state space on the one hand, and considering the particular case of a feasible schedule on the other hand.

Timed CP-nets use discrete time. This is in contrast to continuous time used in timed automata (Alur & Dill 1994). Concurrently with the modelling and analysis reported in this paper, we are using the UPPAAL tool (Larsen, Pettersson & Yi 1997) for modelling and analysing the same generic avionics mission system. This will allow us to make a direct comparison between timed automata and timed CP-nets in terms of ease of modelling, complexity, and feasibility of analysis.

References

Alur, R. & Dill, D. (1994), 'A theory of timed automata', *Theoretical Computer Science* **126**(2), 183–235.

Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P. & Yi, W. (2002), Times - A Tool for Modelling and Implementation of Embedded Systems, *in* 'Proc. 8th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2002), Grenoble, France, Apr. 2002', Vol. 2280 of *Lecture Notes in Computer Science*, Springer-Verlag.

Behrmann, G., Fehnker, A., Hune, T., Larsen, K., Pettersson, P. & Romijn, J. (2001), Efficient Guiding Towards Cost-Optimality in UPPAAL, *in* 'Proc. 7th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2001), Genova, Italy, Apr. 2001', Vol. 2031 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 174–188.

Berthelot, G. & Petrucci, L. (2001), 'Specification and Validation of a Concurrent System: An Educational Project', *International Journal on Software Tools for Technology Transfer* **3**(4), 372–381.

Burns, A., Wellings, A. J., Burns, F., Koelmans, A., Koutny, M., Romanovsky, A. & Yakovlev, A. (2000), Towards Modelling and Verification of Concurrent ADA Programs Using Petri Nets, *in* 'Proceedings of Workshop on Software Engineering and Petri Nets', Department of Computer Science, University of Aarhus, pp. 115–134. DAIMI-PB 548. ISSN 0105-8517.

Christensen, S., Kristensen, L. M. & Mailund, T. (2001*a*), Condensed State Spaces for Timed Petri Nets, *in* 'Proceedings of ICATPN'01', Vol. 2075 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 101–120.

Christensen, S., Kristensen, L. & Mailund, T. (2001*b*), A Sweep-Line Method for State Space Exploration, *in* 'Proceedings of TACAS'01', Vol. 2031 of *LNCS*, Springer-Verlag, pp. 450–464.

CPN Tools (2001), `http://wiki.daimi.au.dk:8000/cpntools/cpntools.wiki`.

Desel, J. & Reisig, W. (1998), Place/Transition Petri Nets, *in* 'Lectures on Petri Nets I: Basic Models', Vol. 1491 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 122–173.

Design/CPN Online (1996), `http://www.daimi.au.dk/designCPN/`.

Fehnker, A. (1999), Scheduling a Steel Plant with Timed Automata, *in* 'Proceedings of 6th International Conference on Real-Time Computing Systems and Applications (RTCSA)', IEEE Computer Society, pp. 280–286.

Godefroid, P. (1996), *Partial-Order Methods for the Verification of Concurrent Systems, An Approach to the State-Explosion Problem*, Vol. 1032 of *LNCS*, Springer-Verlag.

Jensen, K. (1997), *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volumes 1, 2 and 3*, Monographs in Theoretical Computer Science, Springer-Verlag. (2nd edition).

Kristensen, L. M., Christensen, S. & Jensen, K. (1998), 'The Practitioner's Guide to Coloured Petri Nets', *International Journal on Software Tools for Technology Transfer* **2**(2), 98–132.

Larsen, K., Pettersson, P. & Yi, W. (1997), 'UPPAAL in a Nutshell', *International Journal on Software Tools for Technology Transfer* **1**(1+2), 134–152.

Liu, J. (2000), *Real-Time Systems*, Prentice-Hall.

Locke, C. D., Lucas, L. & Goodenough, J. B. (1990), Generic Avionics Software Specification, Technical Report CMU/SEI-90-TR-8, Software Engineering Institute, Carnegie Mellon University.

Murata, T. (1989), Petri Nets: Properties, Analysis and Application, *in* 'Proceedings of the IEEE, Vol. 77, No. 4', IEEE Computer Society.

Qureshi, Z., Billington, J. & Kristensen, L. (2001), Modelling Military Airborne Mission Systems for Functional Analysis, *in* 'Proceedings of 20th Digital Avionics Systems Conference'. CD-ROM.

Shatz, S., Tu, S., Murata, T. & Duri, S. (1996), 'An Application of Petri Net Reduction to ADA Tasking Deadlock Analysis', *IEEE Transactions on Parallel and Distributed Systems* **7**(12), 1309–1324.

Spitzer, C. R. (1992), *Digital Avionics Systems – Principles and Practices*, McGraw-Hill. (2nd edition).

Sun, H. (2001), Modelling and Schedulability of Real-Time Tasks with Timed Automata, *in* 'Proceedings of International Conference on Parallel and Distributed Processing Techniques and Application', Vol. 4, CSREA Press, pp. 2146–2151.

Ullman, J. (1998), *Elements of ML Programming*, Prentice-Hall.