Model-based Development of a Course of Action Scheduling Tool

Lars M. Kristensen[1], Peter Mechlenborg[1],
Lin Zhang[2], Brice Mitchell[2], and Guy E. Gallasch[3]

[1] Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, DENMARK,
{lmkristensen,metch}@daimi.au.dk
[2] Defence Science and Technology Organisation
PO Box 1500, Edinburgh, SA 5111, AUSTRALIA
{Lin.Zhang, Brice.Mitchell}@dsto.defence.gov.au
[3] Computer Systems Engineering Centre, University of South Australia,
Mawson Lakes Campus, SA 5095, AUSTRALIA
guy.gallasch@postgrads.unisa.edu.au

Abstract. This paper shows how a formal method in the form of Coloured Petri Nets (CPNs) and the supporting CPN Tools have been used in the development of the Course of Action Scheduling Tool (COAST). The aim of COAST is to support human planners in the specification and scheduling of tasks in a Course of Action. CPNs have been used to develop a formal model of the task execution framework underlying COAST. The CPN model has been extracted in executable form from CPN Tools and embedded directly into COAST, thereby automatically bridging the gap between the formal specification and its implementation. The scheduling capabilities of COAST are based on state space exploration of the embedded CPN model. Planners interact with COAST using a domain-specific graphical user interface (GUI) that hides the embedded CPN model and analysis algorithms. This means that COAST is based on a rigorous semantical model, but the use of formal methods is transparent to the users. Trials of operational planning using COAST have been conducted within the Australian Defence Force.

Keywords: Application of Coloured Petri nets, State space analysis, Scheduling, Command and Control, Methodologies, Tools.

1 Introduction

Planning and scheduling [6] are activities performed in many domains such as building construction, natural disaster relief operations, search and rescue missions, and military operations. Planning is a major challenge due to several factors, including time pressure, ambiguity in guidance, uncertainty, and complexity of the problem. The development of computer tools that can aid planners in developing and analysing Courses of Actions such that they meet their objectives is therefore of key interest in many application domains. Recently, there has been an increased interest in the application of formal methods and associated analysis techniques in the planning and scheduling domain (see, e.g., [1, 3, 11]).

This paper presents the Course of Action Scheduling Tool (COAST) being developed for the Australian Defence Force (ADF) by the Australian Defence Science and Technology Organisation (DSTO). The aim of COAST is to support planners in *Course of Action* (COA) development and analysis which are two of the main activities in planning processes within the ADF. The framework underlying COAST has been deliberately made generic in order to make COAST applicable to a broad spectrum of domains, and so not restricting its applicability to the military planning domain. The basic entities in a COA (representing a planning problem) are *tasks* which have associated pre- and postconditions describing the *conditions* required for a task to execute and the *effect* of executing the task. The dependencies between tasks expressed via conditions capture the logical structure of a COA. The execution of a task also requires *resources*; a subset of which are released in accordance with the specifications when the task terminates. Task *synchronisations* make it possible to directly specify temporal and precedence information for tasks, e.g., a set of tasks must start simultaneously. The main analysis capability of COAST is the computation of task schedules called *lines of operation* (LOPs) which are execution sequences of tasks that lead from an initial state to a desired *goal state* which is a state in which a certain set of conditions are satisfied. COAST also supports the planners in debugging and identifying errors in COAs.

The development of COAST has been driven by the use of formal methods in the form of *Coloured Petri Nets* (CP-nets or CPNs) [7, 8] and the supporting computer tool CPN Tools [2]. CPN modelling was chosen because CP-nets support construction of compact parametriseable models, support structured data types, make it possible to model time, and allow models to be hierarchically structured into a set of modules.

The basic idea behind the development of COAST has been to use CP-nets to develop, formalise, and implement the task execution framework which forms the core of COAST. The CPN model formalises the execution of tasks according to the pre- and postconditions of tasks, imposed synchronisations, and assigned resources. The concrete tasks, conditions, synchronisations, and resources that make up the COA to be analysed are represented as *tokens* populating the CPN model. The analysis capabilities of COAST are based on *state space exploration* [14]. State space exploration relies on computing all reachable states and state changes of the system and representing these as a directed graph where nodes represent reachable states and arcs represent occurring events. Two algorithms are implemented for computing lines of operation: a two-phase algorithm consisting of a depth-first state space generation followed by a breadth-first traversal, and an algorithm that is based on the so-called sweep-line method [10]. The CPN model and the analysis algorithms that form the core of COAST have been hidden behind a domain-specific graphical user interface. This makes the use of the underlying formal method transparent to the planners who cannot be expected to be familiar with CP-nets and state space methods.

A preliminary version of the task execution framework of COAST has been informally presented in [15,16] together with the graphical user interface. Some early algorithms for computing lines of operation were presented in [9]. The contribution of this paper is to present the formal engineering aspects of COAST in the form of the underlying CPN model and the new state space exploration algorithms implemented for obtaining lines of operation. We also demonstrate how the sweep-line method [10] can be applied to the planning domain. Furthermore, we explain how COAST has been engineered via embedding of an executable CPN model. The latter demonstrates how formal methods in the form of CP-nets can be used in software development.

The rest of the paper is organised as follows. Section 2 presents the methodology used for the formal engineering of COAST. Section 3 presents selected parts of the CPN model that formalises the task execution framework. Section 4 presents the state space exploration algorithms for generating lines of operation. Finally, we sum up the conclusions in Sect. 5. The reader is assumed to be familiar with the CPN modelling language and the basic ideas behind state space methods.

2 Formal Engineering Methodology

COAST is based on a client-server architecture. The client constitutes the domain-specific graphical user interface and is used for the specification of COAs. It supports the human planners in specifying tasks, resources, conditions, and synchronisations. When the COA is to be developed and analysed this information is sent to the COAST server. The client can now invoke the analysis algorithms in the server to compute lines of operation. The server also supports the client in exploring and debugging the COA in case the analysis shows that no lines of operation exist. Communication between the client and the server is based on a remote procedure call (RPC) mechanism implemented using the Comms/CPN library [5].

This paper concentrates on the development of the COAST server which constitutes the computational back-end of COAST. The development of the server has followed a model-based engineering process [12]. Figure 1 depicts the engineering process of developing the application that constitutes the server. The first step was to develop and formalise the planning domain that provides the semantical foundation of COAST. This was done by constructing a CPN model using CPN Tools that formally captures the semantics of tasks, conditions, resources, and synchronisations. This activity involved discussions with the prospective users of COAST (i.e., the planners) to identify requirements and determine the concepts and working processes that were to be supported. The second step was then to extract the constructed CPN model from CPN Tools. This was done by saving a *simulation image* from CPN Tools. This simulation image contains the Standard ML (SML) [13] code that CPN Tools generates for simulation of the CPN model. An important property of the CPN model is that it has been parameterised with respect to the set of tasks, conditions, resources, and synchronisations. This ensures that a given COA can be analysed by setting the initial state of the CPN model accordingly, i.e., no changes to the structure of the CPN model is required to analyse a different COA. This means that the simulation image extracted from CPN Tools is able to simulate any COA, and CPN Tools is no longer needed once the simulation image has been extracted. The third step was the implementation of a suitable interface to the extracted CPN model and the implementation of the state space exploration algorithms.

The Model Interface module contains two sets of primitives:

Initialisation primitives that make it possible to set the initial state of the CPN model according to a concrete set of tasks, conditions, resources, and synchronisation that constitute the COA to be analysed.

Transition relation primitives that provide access to the transition relation determined by the CPN model. This set of primitives make it possible to obtain the set of events enabled in a given state, and the state reached when an enabled event occurs in a given state.

The transition relation primitives are used to implement the state space exploration algorithms in the Analysis module for computing lines of operation. The state space exploration algorithms will be presented in Sect. 4. The Comms/CPN module was added implementing a remote procedure call mechanism allowing the client to invoke the primitives in the analysis and the initialisation module. The resulting application constitutes the COAST server.

3 The COAST CPN Model

The conceptual framework underlying COAST is based on the notion of tasks, conditions, synchronisations, and resources representing the entities of the planning domain. The complete COAST CPN model is hierarchically structured into 24 modules. As the CPN model is too large to be presented in full in this paper, we provide an overview of the CPN model, and illustrate how the key concepts in the planning domain have been modelled and represented using CP-nets.

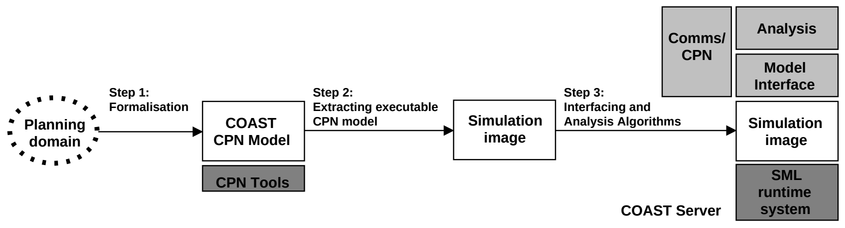

Fig. 1. Engineering process for the COAST server.

3.1 Modelling of COA Entities

A concrete COA to be analysed consists of a set of tasks (T), conditions (C), synchronisations (S), and a multi-set[1] of resources (R). These entities are represented as *tokens* in the CPN model based on the *colour set* definitions listed in Figure 2. Figure 2 lists the definitions of the colour sets that represent the key entities of a COA. Not all colour set definitions are given as the CPN model contains 53 colour sets in total.

```
colset Task = record
    name           : STRING       * duration     : Duration    *
    normalprecond  : SConditions  * vanprecond   : SConditions *
    sustainprecond : SConditions  * termprecond  : SConditions *
    instanteffect  : SConditions  * posteffect   : SConditions *
    sustaineffect  : SConditions  *
    startresources : ResourceList * resourceloss : ResourceList;

colset Resource = product INT * STRING;
colset ResourceList = list Resource;
colset ResourcexAvailability = product Resource * Availability;
colset ResourceSpecs = list ResourcexAvailability;
colset Resources = union IDLE : ResourceSpecs + LOST : ResourceSpecs;

colset STRINGxBOOL = product STRING * BOOL;
colset SCondition = STRINGxBOOL;
colset SConditions = list SCondition;
colset Condition = union STRINGxBOOL;
colset Conditions = list Condition;

colset BeginSynchronisation = list Task;
colset EndSynchronisation = list Task;
```

Fig. 2. Colour set definitions for representing COA entities.

Tasks are the executable entities in a COA and are modelled by colours (data values) of the colour set (type) `Task` which is defined as a record consisting of 11 fields. The `name` field is used to specify the name of the task and the `duration` field to specify the minimal duration of the task. The duration of a task may be extended due to synchronisations, and not all tasks are required to have a specified minimal duration since their durations may be implicitly given by synchronisations and conditions. The remaining fields can be divided into:

[1] A multi-set (bag) is required since there may be several resources of the same type.

Preconditions that specify the conditions that must be valid for starting the task. The colour set `SConditions` is used for modelling the condition attributes of tasks. A task has a set of *normal preconditions* (represented by field `normalprecond`) that specify the conditions that must be satisfied for the task to start. A subset of the normal preconditions may be further specified as *vanishing preconditions* to represent the effect that the start of the task will invalidate such preconditions. The *sustaining preconditions* (field `sustainprecond`) specify the set of conditions that must be satisfied for the entire duration of execution of the task. If a sustaining precondition becomes invalid, then it will cause the task to *abort* which may in turn cause other tasks to be *interrupted*. The *termination preconditions* (field `termprecond`) specify the conditions that must be satisfied for the task to terminate.

Effects that specify the effects of starting and executing a task. The *instant effects* (field `instanteffect`) are conditions that become immediately valid when the task starts executing. The *post effects* (field `posteffect`) are conditions that become valid at the moment the task terminates. *Sustained effects* (field `sustaineffect`) are conditions that are valid as long as the task is executing.

Resources that specify the resources required by the tasks during their execution. Resources typically represent planes, ships, and personnel required to execute a task. Resources may be lost or consumed in the course of executing a task. *Start resources* specify the resources required to start the task, and they are allocated as long as the task is executing. The *resource loss* field specifies resources that may be lost when executing the task. Each type of resources is modelled by the colour set `Resource` which is a product of an integer (`INT`) specifying the quantity and a string (`STRING`) specifying the resource name. The colour set `ResourceList` is used for specifying the resource attributes of a task.

Conditions are used to describe the explicit logical dependencies between tasks via preconditions and effects. As an example, a task `T1` may have an effect used as a precondition of a task `T2`. Hence, `T2` logically depends on `T1` in the sense that it cannot be started until `T1` has been executed.

The colour set `Conditions` is used for representing the value of the conditions in the course of executing tasks in the COA. A condition is a pair consisting of a `STRING` specifying the name of the condition and a boolean (`BOOL`) specifying the truth value. The colour set `ResourceSpecs` is used to represent the state of resources assigned to the COA. The colour set `Resources` is defined as a `union` for modelling the idle and lost resources. The assigned resources also have a specification of the *availability* of the resources (via the `Availability` colour set) specifying the time intervals at which the resource is available. Resources may be available only at certain times due to e.g., service intervals.

Synchronisations are used to directly specify precedence and temporal constraints between tasks. For example, a set of tasks must begin or end simultaneously, a specific amount of time must elapse between the start and end of certain tasks, or a task can only start after a certain point in time. Tasks that are required to begin at the same time are said to be *begin-synchronised*, and tasks required to end at the same time are said to be *end-synchronised*. End-synchronisations can cause the duration of a task to be extended. The colour sets `BeginSynchronisation` and `EndSynchronisation` represent that a set (list) of tasks are begin and end-synchronised, respectively.

3.2 Modelling Task Behaviour

Figure 3 shows the top level module of the CPN model which is composed of three main parts represented by the three substitution transitions Initialise, Execute, and Environment. The substitution transition Initialise and its submodules are used for the initialisation of the model according to the concrete set of tasks, conditions, synchronisations, and resources in the COA to be analysed. The substitution transition Execute and its submodules model the execution of tasks, i.e., start, termination, abortion, and interruption of tasks. The substitution transition Environment and its submodules model the environment in which tasks execute, and are responsible for managing the availability of resources over time and the change of conditions over time. The text in the small rectangular box attached to each substitution transition gives the name of the associated submodule.

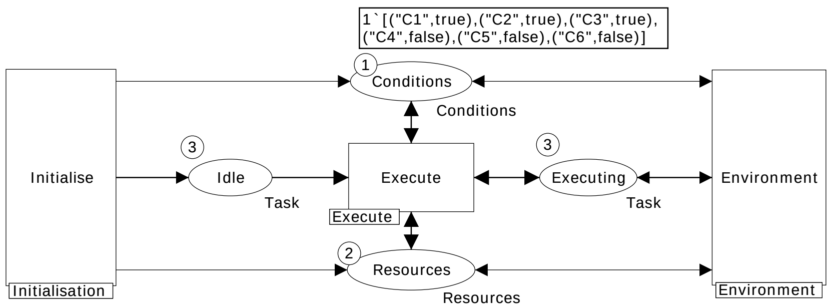

Fig. 3. Top level module of the CPN model.

There are four places in Fig. 3. The Resources place models the state of the resources, the Idle place models the tasks that are yet to be executed, the Executing place models the tasks currently being executed, and the Conditions place models the values of the conditions. The state of a CPN model is a distribution of tokens on the places of the CPN model. Figure 3 depicts a simple example state for a COA with six tasks. The number of tokens on a place is written in the small circle positioned above the place. The detailed data values of the tokens are given in the box positioned next to the circle. Place Conditions contains one token that is a list containing the conditions in the COA and their truth values. Place Resources contains two tokens. There is one token consisting of a list describing the current set of idle (available) resources, and the other token consisting of a list describing the resources that have been lost until now. Since the colours of the tokens on the places Resources, Executing and Idle are of a complex colour set, we have only shown the numbers of tokens and not the data values.

Figure 4 shows one of the submodules of the `Execute` substitution transition (see Fig. 3). This submodule represents one of the steps in starting tasks. The transition Start models the event of starting a set of begin-synchronised tasks. The two places Resources and Conditions are interface places linked to the accordingly named places of the top-level module shown in Fig. 3. The begin-synchronised tasks are represented as tokens on place Tasks. An occurrence of the transition removes a token representing the begin-synchronised tasks (bound to the variable tasks) from place Tasks, the token representing the idle resources (bound to the variable idleres) from place Resources, and the list representing the values of the conditions (bound to the variable conditions) from place Conditions. The transition adds tokens representing the set of tasks to be started to the place Starting, puts the idle resources back on place Resources, and puts a token back on place Conditions updated according to the start effects of the tasks. The updating of conditions is handled by the function `InstantEffects` on the arc expression on the arc from the Start transition to the place Conditions. All idle resources are put back on place Resources since the actual allocation of resources to the tasks are done in a subsequent step and handled by another submodule. The *guard* of the transition specified in the square brackets expresses the predicate that the transition is enabled only if the conditions specified in the preconditions of the tasks are satisfied and the resources are available. This requirement is expressed by the two predicate functions `SatPreconditions` and `ResourcesAvailable`.

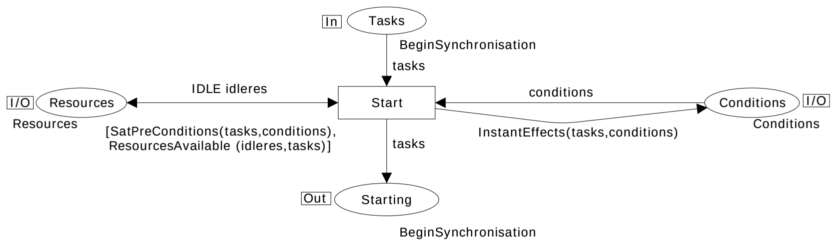

Fig. 4. Submodule for starting of tasks.

The CPN model contains a number of submodules of a similar complexity to the Start submodule shown in Fig. 4 that model the details of task execution and their effect on conditions and resources.

4 State Space Exploration

The main analysis capability of COAST is the computation of *lines of operation* (LOPs). From the previous sections it follows that a COA can be syntactically described as a tuple $COA = (T, C, R, S)$ consisting of a finite set of tasks T, a finite set of conditions C, a finite multi-set of resources R, and a finite set of synchronisations S. The semantics of task execution is defined by the CPN model discussed in the previous section. A LOP for a COA describes an execution of a subset of the tasks in the COA. A LOP of length n is a sequence of tuples (t_i, s_i, e_i, r_i) for $1 \leq i \leq n$ where $t_i \in T$ is a task, s_i is the start time of t_i, e_i is the end time of t_i and r_i is the multi-set of resources assigned to t_i. Two classes of LOPs are considered. *Complete LOPs* are LOPs leading to a *desired end-state* defined by a *goal state predicate* ϕ_{COA} on states that captures the purpose of the COA. *Incomplete* LOPs are LOPs leading to an *undesired end-state* not satisfying ϕ_{COA}. Incomplete LOPs typically arise in early phases of COA development and may be caused by insufficient resources implying that certain tasks cannot be executed, or logical errors caused, e.g., by missing tasks. COAST also supports the planning staff in identifying such errors and inconsistencies in the COA.

The computation of the set of complete and incomplete LOPs is based on state space exploration of the CPN model. Figure 5 shows the state space for an example COA with six tasks, T1, T2,…,T6. The nodes correspond to the set of reachable states and the arcs correspond to the occurrences of enabled binding elements (events). Node 1 to the upper left corresponds to the initial state and node 21 to the lower right corresponds to a desired end-state. The state space represents all the possible ways in which the tasks in the COA can be executed, given that tasks will execute as soon as they can.

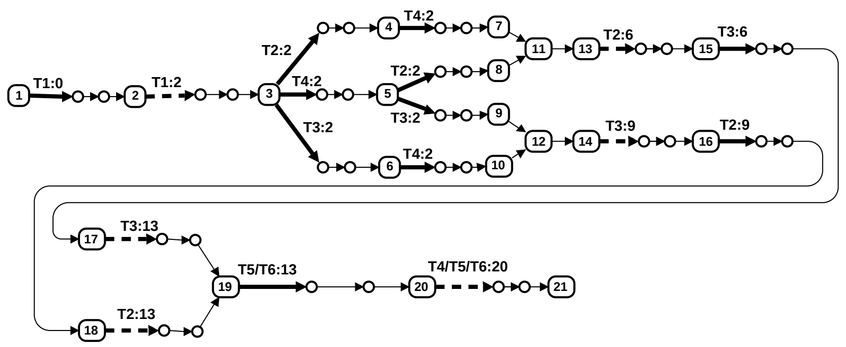

Fig. 5. Example of a state space for the CPN model.

A path in the state space corresponds to a particular execution of the CPN model, i.e., determines a LOP. It is the binding elements in the state space that represent start and termination of tasks that define the LOPs. A distinction is therefore made between visible and invisible binding elements. The *visible binding elements* represent start and termination of tasks, and allocation of resources. All other binding elements are internal events in the CPN model and are *invisible*. The thick arcs in Fig. 5 have labels of the form $Ti : t$ where i specifies the task number and t specifies the time at which the event takes place. Thick solid arcs represent start of tasks and thick dashed arcs represent termination of tasks. The thin arcs represent invisible events. As an example, task T1 starts at time 0 as specified by the label on the outgoing arc from node 1 and terminates at time 2 as specified by the label on the outgoing arc from node 2. The state space in Fig. 5 has four paths leading from the initial state to the desired end-state depending on the branch chosen at node 3 and node 5. When considering the start and termination of tasks it be seen that the four paths determine two complete LOPs L_1 and L_2:

$$L_1 = (\mathsf{T1}, 0, 2), (\mathsf{T2}, 2, 6), (\mathsf{T4}, 2, 20), (\mathsf{T3}, 6, 13), (\mathsf{T5}, 13, 20), (\mathsf{T6}, 13, 20)$$
$$L_2 = (\mathsf{T1}, 0, 2), (\mathsf{T4}, 2, 20), (\mathsf{T3}, 2, 9), (\mathsf{T2}, 9, 13), (\mathsf{T5}, 13, 20), (\mathsf{T6}, 13, 20)$$

Two algorithms to be presented in Sections 4.1 and 4.2 have been implemented in COAST for the computation of LOPs. Both algorithms are based on state space exploration and are complete in that they report all complete and incomplete LOPs. The two algorithms rely on the following theorem that can be proved from the net structure of the CPN model and by inspecting each transition observing that the occurrence of each binding element has a unique effect on the state of the CPN model. We have omitted the proof in this paper since we do not have sufficient space to present the CPN model in full.

Theorem 1. *Let $COA = (T, C, R, S)$ be a COA and let $CPN(COA)$ be the COAST CPN model initialised according to COA. Then the following holds:*

1. *The state space of $CPN(COA)$ is finite, i.e., the CPN model has a finite set of reachable states and a finite set of enabled binding elements in each state.*
2. *The state space is an acyclic directed graph.*
3. *Let σ_1 of length l_1 and σ_2 of length l_2 be two paths in the state space starting from the initial state and both leading to the state s. Then $l_1 = l_2$.*

Item 1 ensures termination of state space exploration, independently of the COA with which the CPN model is initialised. Item 2 implies that there are finitely many paths leading to a given reachable state and hence there can only be finitely many LOPs to be reported. Item 3 ensures that when visiting a state s during a breadth-first state space traversal all predecessors of s will already have been visited. This is exploited by both algorithms.

4.1 Two-Phase Algorithm

The first algorithm is a two-phase algorithm. The first phase is a depth-first construction of the full state space where complete and incomplete LOPs are reported on-the-fly as they are encountered. The second phase is a breadth-first traversal of the state space constructed in the first phase where the LOPs not found in the first phase are computed. Depth-first construction in the first phase allows LOPs to be reported as soon as they are found. The second phase is required since not all LOPs may be reported by the depth-first phase. As an example, a depth-first generation of the state space in Fig. 5 would only find one of the LOPs L_1 and L_2, since node 19 will already have been visited when it is encountered the second time (either via node 17 or 18).

The procedure DepthFirstPhase in Fig. 6 specifies the first phase of the algorithm. It uses three data structures: Nodes which stores the set of nodes (states) generated, Arcs which stores the set of explored arcs, and Stack which is used to store the set of unprocessed states and ensures depth-first generation. The procedure is invoked with the binding element corresponding to the initialisation step of the CPN model.

```
 1: procedure DepthFirstPhase $(s, a, s')$
 2: Arcs.insert $(s, a, s')$
 3: if ¬ Nodes.member($s'$) then
 4:    Nodes.insert($s'$)
 5: end if
 6: if $\phi_{COA}(s')$ then
 7:    LOP.create(Stack.prefix (),complete)
 8: else
 9:    successors = ModelInterface.getnext($s'$)
10:    if successors = $\emptyset$ then
11:       LOP.create (Stack.prefix (),incomplete)
12:    else
13:       Stack.push (successors)
14:    end if
15: end if
16: if ¬ Stack.empty() then
17:    DepthFirstPhase (Stack.pop ())
18: end if
```

Fig. 6. Depth-first Phase of LOP computation.

The procedure DepthFirstPhase first inserts the arc (s, a, s') into the set of arcs (line 2) and then checks whether s' has already been visited before (lines 3-4). If s' is a new state, then it is inserted into the set of nodes. If s' corresponds to a desired end-state then the sequence of binding elements executed to reach s' is extracted from the depth-first stack and reported as a complete LOP (lines 6-7). Otherwise the set of successors of s' in the state space is computed (line 9) using the ModelInterface module (see Sect. 2). An incomplete LOP is reported if s' does not have any successor states and is not a desired end-state (line 11). If state space exploration is to continue from s' then the set of successors is pushed onto the stack (line 13). The exploration of the state space continues in lines 16-17 as long as the stack is not empty. When the procedure terminates, Nodes contains the set of reachable states cut-off according to the goal state predicate ϕ_{COA}, and Arcs contains to the set of arcs between the reachable states.

The second phase of the LOP generation is specified in Fig. 7 and conducts a breadth-first traversal of the state space computed in the first phase.

```
 1: procedure BreadthFirstPhase ()
 2: if $\neg$ (Queue.empty ()) then
 3:    $s$ = Queue.delete ()
 4: end if
 5: lops = LOP.get ($s$)
 6: LOP.delete ($s$)
 7: successors = Arcs.out ($s$)
 8: if successors = $\emptyset$ then
 9:    if $\phi_{COA}(s)$ then
10:       LOP.create (lops,complete)
11:    else
12:       LOP.create (lops,incomplete)
13:    end if
14: else
15:    for all $(s, a, s') \in$ successors do
16:       $lops'$ = LOP.augment ($lops, a$)
17:       LOP.add ($s', lops'$)
18:       if $\neg$ Visited.member($s'$) then
19:          Visited.insert ($s'$)
20:          Queue.insert ($s'$)
21:       end if
22:    end for
23: end if
```

Fig. 7. Breadth-first Phase of LOP computation.

The procedure BreadthFirstPhase uses three data-structures: Visited which keeps track of states that have been visited, LOP which is used to associate *partial LOPs* with states, i.e., the LOPs corresponding to the possible ways in which the given state can be reached, and Queue that implements the breadth-first traversal queue. The procedure is invoked with the initial state inserted into the queue. The procedure starts by selecting a state s to be processed from the queue and obtaining the LOPs stored with the state s (lines 3-6). The partial LOPs stored with s are then deleted since all predecessors of s will have been processed according Thm. 1(3). If the state has no successors (line 8) then the associated

LOPs are reported as complete if s is a goal state; otherwise they are reported as incomplete LOPs. If the state s has successors then these successors are handled in turn (lines 15-22) by augmenting the partial LOPs from s according to the event executed to reach the successor state s'. These augmented LOPs will then be added to the LOPs associated with s' (line 17). If the successor state has not been visited before, then it is inserted into the queue and into the set of visited states (lines 18-21).

4.2 Sweep-Line Algorithm

The second algorithm implemented in COAST is based on a variant of the sweep-line method [10]. This sweep-line method reduces peak memory usage by not keeping the complete state space in memory. The basic idea of the sweep-line method is to exploit a notion of *progress* found in many systems to delete states from memory during state space construction and thereby reduce the peak memory usage. The states that are deleted are known to be unreachable from the set of unexplored states and can therefore be safely deleted without compromising the termination and complete coverage of the state space construction. The COAST CPN model exhibits progress which is reflected in the state space of the CPN model which according to Thm. 1 is acyclic and has the property that all paths to a given state have the same length. This property implies that when conducting a breadth-first construction of the state space, it is possible to delete a state s when it is removed from the breadth-first queue for processing. The reason is that when the state s is removed from the queue all the predecessors s have been processed, and hence s is no longer needed for comparison with newly generated states. The basic idea in the application of the sweep-line method in the context of COAST is therefore to construct the state space in a breadth-first order and compute the LOPs on-the-fly during the state space construction in a similar way as in the breadth-first traversal in the two-phase algorithm. At any given moment the algorithm only stores the nodes and associated partial LOPs on the frontier of the state space exploration and the peak memory usage will be reduced.

4.3 Experimental Results

Table 1 provides a set of experimental results with the algorithms presented in the previous subsections on some representative examples. All experimental results have been obtained on a Pentium III 1 GHz PC with 1Gb of main memory. The first part of the table lists the number of Nodes and Arcs in the state space. The second part gives the experimental results for the two-phase algorithm. It specifies the total CPU Time in seconds used by the algorithm and the percentage of the time spent in the depth-first phase (DFTime) and the breadth-first phase (BFTime). For the depth-first phase it also specifies the percentage of time spent on the calculation of enabled binding elements (DFEna) and inserting newly generated states and arcs into the state space (DFSto). The third part of the table specifies the results for the sweep-line algorithm by giving the total CPU

Time in seconds used for exploring the state space in percentage of the CPU time for the two-phase algorithm and the Peak number of states stored during the exploration in percentage of the total number of states. For the two-phase algorithm it can be seen that most time is spent in the depth-first phase. This is expected since this is where the actual state space construction is conducted. The time spent in the depth-first phase is divided almost evenly between the storage of nodes and arcs, and the computation of enabling. Almost no time is spent in the breadth-first phase which shows that the time used on computing the LOPs is insignificant compared to the time used to construct the state space. The results for the sweep-line algorithm show that the peak number of stored states is reduced to between 27.5 % and 70 % depending on the example. The sweep-line is faster than the two-phase algorithm which may at first seem surprising. The reason is that states are only present in the breadth-first queue, and hence the relative expensive check for whether a state has already been visited has been eliminated.

Table 1. Selected experimental results.

COA	State space		Two-Phase					Sweep-Line	
	Nodes	Arcs	Time	DFTime	DFSto	DFEna	BFTime	Peak	Time
COA_1	283	317	2.81	99.3 %	44.1 %	53.7 %	0.36 %	45.2 %	71.2 %
COA_2	1,253	1,443	16.13	99.7 %	44.7 %	53.9 %	0.25 %	27.5 %	61.9 %
COA_3	2,587	2,974	34.06	99.7 %	44.8 %	53.6 %	0.29 %	38.5 %	58.7 %
COA_4	77	85	0.36	97.2 %	38.9 %	58.3 %	2.78 %	46.8 %	83.3 %
COA_5	142	169	1.68	99.4 %	39.9 %	58.9 %	0.59 %	59.9 %	59.6 %
COA_6	8,263	10,394	101.94	99.6 %	43.2 %	54.9 %	0.35 %	70.0 %	59.8 %
COA_7	1,977	2,249	25.72	99.7 %	43.9 %	54.5 %	0.27 %	39.5 %	62.2 %

The *reachability of a goal state problem* solved by COAST server when computing the LOPs is equivalent to the *reachability of a submarking problem* which is known to be PSPACE-hard for one-safe Petri nets [4]. Since the CPN model when initialised with a COA $COA = (T, C, R, S)$ can be unfolded to a one-safe Petri net of size $\Theta(|T|+|C|+|R|+|S|)$, i.e., a Petri net which is linear in the size of the COA, and since any one-safe Petri net can be obtained by selecting the proper COA, this implies that the problem solved by COAST is PSPACE-hard.

The example COAs that COAST has been tested against consist of 15 to 30 tasks resulting in state spaces with 10,000 to 20,000 nodes and 25,000 to 35,000 arcs. Such state spaces can be generated in less than 2 minutes on a standard PC. The state spaces are relatively small because the conditions, available resources, and imposed synchronisations in practice strongly limit the possible orders in which the tasks can be executed. This observation is one of the main reasons why state space construction appears to be a feasible approach for COAST.

Both the two-phase algorithm and the sweep-line algorithm are implemented because their relative performance depends on the structure of the state space. The two-phase algorithm has the advantage that it can report LOPs early due to the depth-first construction where LOPs can be extracted from the depth-first search stack. This means that the two-phase algorithm is able to work well on very wide state spaces that may be too large to explore in full using the sweep-line algorithm. The two-phase algorithm is implemented in such a way that it is possible to terminate the LOP generation when a certain number of LOPs (specified by the user) has been found. Wide state spaces are caused by interleaving when there are very few constraints on the execution of tasks in the COA. The sweep-line algorithm, on the other hand, performs better on long and narrow state spaces, e.g., when there are many but highly constrained tasks in the COA. The two algorithms therefore complement each other.

5 Conclusions

This paper has demonstrated how formal modelling has been applied in a model-based development of the COAST server. CP-nets were applied since a COA consisting of tasks, resources, conditions, and synchronisations is naturally viewed as a concurrent system. The main benefit of using a formal modelling language for concurrent systems has, in our view, been that it allowed us to concentrate on formalisation of the task execution framework for COAST and abstract from implementation issues. Furthermore, the graphical representation of CP-nets was extremely useful for discussions in the development process. The main advantage of our approach is that the resulting formal model is then directly embedded in the final implementation. This effectively eliminates the challenging step of going from a formal specification to its implementation. Furthermore, our practical experiments on representative examples have demonstrated that state space exploration is a feasible analysis approach within the COAST domain.

Our approach requires that the modelling language is expressive enough to support a level of parameterisation that makes it possible to initialise the constructed model with problem instances. Furthermore, the computer tool supporting the formal modelling language must support the extraction of models in executable form that allows the transition relation of the model to be accessed. The methodology applied for the development of COAST is therefore also applicable to other formal modelling languages where the above requirements are satisfied, and generally applicable to cases where a formalisation of a particular domain is required as a basis for the development of a domain specific tool.

References

1. T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times - A Tool for Modelling and Implementation of Embedded Systems. In *Proc. of TACAS'02*, volume 2280 of *LNCS*, pages 460–464, 2002.
2. CPN Tools. `www.daimi.au.dk/CPNtools`.
3. S. Edelkamp. Promela Planning. In *Proc. of SPIN'03*, volume 2648 of *LNCS*, pages 197–212, 2003.

4. J. Esparza. Decidability and Complexity of Petri net Problems - An Introduction. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 374–428. Springer-Verlag, 1998.
5. G. Gallasch and L. M. Kristensen. Comms/CPN: A Communication Infrastructure for External Communication with Design/CPN. In *Proc. of the 3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 79–93. Department of Computer Science, University of Aarhus, 2001. DAIMI PB-554.
6. M. Ghallab, D.Nau, and P. Traverso. *Automated Planning: Theory and Practice.* Elsevier, 2004.
7. K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. Vol. 1-3.* Springer-Verlag, 1992-1997.
8. L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
9. L.M. Kristensen, J.B. Jørgensen, and K. Jensen. Application of Coloured Petri Nets in System Development. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 626–686. Springer-Verlag, 2004.
10. L.M. Kristensen and T. Mailund. Efficient Path Finding with the Sweep-Line Method using External Storage. In *Proc. of ICFEM'03*, volume 2885 of *LNCS*, pages 319–337, 2003.
11. J.I. Rasmussen, K.G. Larsen, and K. Subramani. Resource-Optimal Scheduling Using Priced Timed Automata. In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 220–235. Springer-Verlag, 2004.
12. B. Schätz. Model-Based Development: Combining Engineering Approaches and Formal Techniques. In *Proc. of ICFEM'04*, volume 3308 of *LNCS*, pages 1–2, 2004.
13. J.D. Ullman. *Elements of ML Programming.* Prentice-Hall, 1998.
14. A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer-Verlag, 1998.
15. L. Zhang, L.M. Kristensen, C. Janczura, G. Gallasch, and J. Billington. A Coloured Petri Net based Tool for Course of Action Development and Analysis. In *Proc. of Workshop on Formal Methods Applied to Defence Systems*, volume 12 of *CRPIT*, pages 125–134. Australian Computer Society, 2001.
16. L. Zhang, L.M. Kristensen, B. Mitchell, C. Janczura, G. Gallasch, and P. Mechlenborg. COAST – An Operational Planning Tool for Course of Action Development and Analysis. In *Proc. of 9th International Command and Control Research and Technology Symposium*, 2004.