Towards Modelling and Verification of Concurrent Ada Programs Using Petri Nets

A.Burns and A.J.Wellings
Real-Time Systems Research Group
Department of Computer Science
University of York, U.K.

F.Burns, A.M.Koelmans, M.Koutny,
A.Romanovsky and A.Yakovlev
Asynchronous Systems Laboratory
Department of Computing Science
University of Newcastle upon Tyne, U.K.

Abstract

Ada 95 is an expressive concurrent programming language with which it is possible to build complex multi-tasking applications. Much of the complexity of these applications stems from the interactions between the tasks. This paper argues that Petri nets offer a promising, tool-supported, technique for checking the logical correctness of the tasking algorithms. The paper illustrates the effectiveness of this approach by showing the correctness of an Ada implementation of the atomic action protocol using a variety of Petri net tools, including PED, PEP and INA for P/T nets and Design/CPN for Coloured Petri nets.

1 Introduction

As high-integrity systems become more sophisticated, the resulting complexity is easier to manage if the applications are represented as concurrent processes rather than sequential ones. Inevitably, the introduction of concurrency brings problems of process interaction and coordination. In trying to solve these problems, language and operating system researchers have introduced new high-level programming constructs. These *design abstractions* are often closely related to the specific domain being addressed. For example, in the world of software fault-tolerance, the notion of conversations [24] and atomic actions [11, 19] are introduced to facilitate the safe and reliable communication between group of processes in the presence of hardware and software failures, in addition to providing a structuring technique for such systems. Research languages such as Concurrent Pascal have been used as the basis for experimentation [18], or a set of procedural extensions or object extensions have been produced. For example, Arjuna uses the latter approach to provide a transaction-based toolkit for C++ [29]. However, it is now accepted that the procedural and object extensions are unable to cope with all the subtleties involved in synchronisation and co-operation between several communicating concurrent processes.

The main disadvantage of domain-specific abstractions is that they seldom make the transition into general-purpose programming languages or operating systems. For example, no mainstream language or operating systems supports the notion of a conversation [9]. The result is that all the hard-earned research experience is not promulgated into industrial use.

If high-level support is not going to be found in mainstream languages, the required functionality must be programmed with lower-level primitives that are available. For some years now we have been exploring the use of the Ada programming language as a vehicle for implementing reliable concurrent systems [32]. The Ada 95 programming language defines a number of expressive concurrency features [1]. Used together they represent a powerful toolkit for building higher-level protocols/design abstractions that have wide application. For example, [32] recently showed how Ada 95 can be used to implement Atomic Actions. And, as such an abstraction is not directly available in any current programming language, this represents a significant step in moving these notions into general use. An examination of this, and other applications, shows that a number of language features are used in tandem to achieve the required result. Features include:

- Tasks - basic unit of concurrency.
- Asynchronous Transfer of Control (ATC) - an asynchronous means of affecting the behaviour of other tasks.
- Protected Types - abstract data types whose operations are executed in mutual exclusion, and which supports condition synchronisation.
- Requeue - a synchronisation primitive that allows a guarded command to be prematurely terminated with the calling task placed on another guarded operation.
- Exceptions - a means of abandoning the execution of a sequential program segment.
- Controlled types - a feature that allows manipulation of object initialisation, finalisation and assignment.

The expressive power of the Ada 95 concurrency features is therefore clear. What is not as straightforward is how to be confident that the higher-level abstractions produced are indeed correct. As a number of interactions are asynchronous this presents a significant verification problem. The idea of verification using Model Checking with a finite state model (FSM) of an Ada program was first presented in [10]. This method constructed a set of FSMs of individual tasks interacting via channels, and applied analysis of the interleaving semantics of the product of FSMs using tool Uppaal (whose underlying formalism is that of CCS). In this paper, we investigate a complementary approach based on Petri nets and their power to model causality between elementary events or actions directly. This can be advantageous for asynchronous nature of interactions between tasks. Petri nets, both ordinary [26] and high level (e.g. coloured nets [17]) offer a wide range of analysis tools to model and verify the logical correctness according to two crucial kinds of properties: (i) safety - an incorrect state cannot be entered (from any legal initial state of the system); and (ii) liveness - a desirable state will be entered (from all legal initial states of the system).

Petri nets have generally been applied to the verification of Ada programs, e.g. [28, 23, 8]. This work has mostly been focused on the syntactic extraction of Petri nets from Ada code in such a way that the verification of properties, such as deadlock detection, could be done more efficiently. To alleviate state space explosion techniques like structural reduction [28] and decomposition [23] of 'Ada nets' have been proposed.

Our research is based on applying Petri nets to model concurrent Ada code, and using Petri nets tools, such as PEP and Design/CPN, to verify its correctness. However, we propose to deal with the unavoidable complexity of the resulting programs within a compositional approach employing a versatile library of design abstractions with well understood and formally verified properties (confidence in the abstraction can be significantly increased and the development activity itself supported by modelling, simulation and analysis of the dynamic behaviour of the Petri net model; the behaviour can be analysed either by exploring the set of reachable states of the net or its partial order semantics, such as the unfolding prefix). The latter can then be used to tackle the verification of complex designs. Thus, while we are ultimately interested in efficient model checking too, the main focus of this paper is on the semantic modelling of salient task interaction mechanisms from Ada 95. To the best of our knowledge, there has been no attempt of using Petri nets to analyse Ada 95 models of Atomic Actions, particularly with ATC and exceptions. However, some work on analysing Ada 95 programs (with ATC, protected objects, and requeue statement) with Petri nets has been recently reported in [15].

This paper is organised as follows. An introduction to model checking based on Petri nets is given in the next section. We use our existing study of Atomic Actions to illustrate the adopted procedure. A simple model is introduced in Section 3 and its refinement in Section 4. Conclusions are presented in Section 5.

2 An Introduction to Model Checking using Petri Nets

Model checking is a technique in which the verification of a system is carried out using a finite representation of its state space. Basic properties, such as absence of deadlock or satisfaction of a state invariant (e.g. mutual exclusion), can be verified by checking individual states. More subtle properties, such as guarantee of progress, require checking for specific cycles in a graph representing the states and possible transitions between them. Properties to be checked are typically described by formulae in a branching time or linear time temporal logic [13].

The main drawback of model checking is that it suffers from the combinatorial explosion problem. That is, even a relatively small system specification may (and often does) yield a very large state space which despite being finite requires computational power for its management beyond the effective capability of available computers. To help cope with the state explosion problem a number of techniques have been proposed which can roughly be classified as aiming at implicit compact representation of the full state space of a reactive concurrent system, or at an explicit representation of a reduced (yet sufficient for a given verification task) state space of the system. Examples of the former are algorithms based on the binary decision diagrams (BDDs) [7]. Techniques aimed at reduced representation of state spaces are typically based on the independence (commutativity) of some actions, which is a characteristic feature of reactive concurrent systems, often relying on the partial order view of concurrent computation. Briefly, in a sequential system, it is the actual order of the execution of individual actions which is usually of importance, whereas in a concurrent system the actual order in which, say, two messages were sent and then received may be irrelevant to the correctness of the whole system. Examples include partial order verification [16, 21] and stubborn set method [31]. The partial order view of concurrent computation is also the basis of the algorithms employing McMillan's unfoldings [14], where the entire state space is represented implicitly using an acyclic directed graph representing system's actions and local states.

Model checking is a technique that requires tool support. For Petri nets, there are many tools of different maturity available. These tools are categorised according to many parameters [33].

In our study, we used three relatively mature tools. One is PEP [2, 3], which uses ordinary Place/Transition nets and a number of model checking methods, such as reachability analysis and unfolding prefix. The second one is INA (Integrated Net Analyzer) [27]. The third is Design/CPN [34], which is based on the Coloured Petri nets and has extensive facilities for simulation and occurrence (reachbility) graph analysis.

2.1 The PEP Tool

The PEP tool [2, 3, 22] provides a modelling and verification environment based on Petri nets, however, its principal method of inputting large designs is to use a simple concurrent programming language. The tool compiles a program into an internal representation in terms of a 1-safe Petri net which can then be verified for correctness using a variety of techniques (also those supported by other model checking tools, such as SPIN or SMV). The relevant correctness properties, can be specified in general purpose logic notation, such as CTL* or S4. The PEP system incorporates model checkers based on unfolding and structural net theory.

The PEP tool's additional advantage is that it is based on a compositional Petri net model, both P/T-net based and high-level net based [4, 5, 6]. It therefore provides a sound ground to develop a compositional model supporting design abstractions.

2.2 The INA Tool

The INA tool is an interactive analysis tool which incorporates a large number of powerful methods for analysis of P/T nets. These methods include analysis of: (i) structural properties, such as state-machine decomposability, deadlock-trap analysis, T- and P-invariant analysis, structural boundedness ; (ii) behavioural properties, such as boundedness, safeness, liveness, deadlock-freeness, dynamic conflict-freenes; (iii) specific user-defined properties, such as those defined by predicates and CTL formulas and traces to pre-defined states. These analyses employ various techniques, such as linear-algebraic methods (for invariants), reachability and coverability graph traversals, reduced reachability graph based on stubborn sets and symmetries.

The INA tool uses a combination of interactive techniques, where the user is prompted for various specifications and queries, and file-processing techniques. The basic Petri net file format is compatible with other tools, such as PED and PEP, using Petri net graphical editors.

2.3 The Design/CPN Tool

Coloured Petri Nets (CP-nets) are an extension of the basic Petri Net model [17]. A CP-net model consists of a collection of places, transitions, and arcs between these places and transitions. The model contains *tokens* that flow around the model and are stored in the places. The essential feature of CP-nets is that they allow complex data types, i.e. objects, to be attached to the tokens. These objects contain attributes reflecting the system being modelled. The flow of the tokens is determined by so called *guards*, which are conditions, attached to the arcs of the model, that determine whether a transition is allowed to fire. These guards therefore determine the dynamic behaviour of the model; they allow sophisticated behavioural properties to be modelled. The only software tool currently capable of simulating and analysing CP-Net models and generating an executable code (in the ML programming language) is Design/CPN [34]. In the Design/CPN system, guards are specified in ML. Crucially, Design/CPN allows entry of hierarchical models, which greatly aids in the understanding of complex models.

3 Model of Simple Atomic Actions

3.1 Atomic Actions

An atomic action is a dynamic mechanism for controlling the joint execution of a group of tasks such that their combined operation appears as an *indivisible* actions [19, 25]. Essentially, an action is atomic if the tasks performing it can detect no state change except those performed by themselves, and if they do not reveal their state changes until the action is complete. Atomic actions can be extended to include forward or backward error recovery. In this paper we will focus only on forward error recovery using exception handling [11]. If an exception occurs in one of the tasks active in an atomic action then that exception is raised in *all* processes active in the action. The exception is said to be *asynchronous* as it originates from another process.

3.2 Atomic Actions in Ada

To show how atomic actions can be programmed in Ada [32], consider a simple non-nested action between, say, three tasks. The action is encapsulated in a package with three visible procedures, each of which is called by the appropriate task. It is assumed that no tasks are aborted and that there are no deserter tasks [18].

```
package simple_action is
  procedure T1(params : param); -- from Task 1
  procedure T2(params : param); -- from Task 2
  procedure T3(params : param); -- from Task 3
end simple_action;
```

The body of the package automatically provides a well-defined boundary, so all that is required is to provide the indivisibility. A protected object, *Controller*, can be used for this purpose. The package's visible procedures call the appropriate entries and procedures in the protected object.

The body of the package is given below.

```
with Ada.Exceptions; use Ada.Exceptions;
package body action is

  type Vote_T is (Commit, Aborted);
  protected controller is
    entry Wait_Abort(E: out Exception_Id);
    entry Done;
    entry Cleanup (Vote : Vote_t;
                   Result : out Vote_t);
    procedure Signal_Abort(E: Exception_Id);
  private
    entry Wait_Cleanup(Vote : Vote_t;
                       Result : out Vote_t);
    Killed : boolean := False;
    Releasing_cleanup : Boolean := False;
    Releasing_Done : Boolean := False;
    Reason : Exception_Id;
    Final_Result : Vote_t := Commit;
    informed : integer := 0;
  end controller;
```

```
-- any local protected objects for
-- communication between actions

protected body controller is
  entry Wait_Abort(E: out Exception_id)
                          when killed is
  begin
    E := Reason;
    informed := informed + 1;
    if informed = 3 then
      Killed := False;
      informed := 0;
    end if;
  end Wait_Abort;

  entry Done when Done'Count = 3 or
                     Releasing_Done is
  begin
    if Done'Count > 0 then
      Releasing_Done := True;
    else
      Releasing_Done := False;
    end if;
  end done;

  entry Cleanup (Vote: Vote_t;
    Result: out Vote_t) when True is
  begin
    if Vote = aborted then
      Final_result := aborted;
    end if;
    requeue Wait_Cleanup with abort;
  end Cleanup;

  procedure Signal_Abort(E: Exception_id) is
  begin
    killed := TRue;
    reason := E;
  end Signal_Abort;

  entry Wait_Cleanup (Vote : Vote_t;
                      Result: out Vote_t)
       when Wait_Cleanup'Count = 3 or
                        Releasing_Cleanup is
  begin
    Result := Final_Result;
    if Wait_Cleanup'Count > 0 then
      Releasing_Cleanup := True;
    else
      Releasing_Cleanup := False;
      Final_Result := Commit;
    end if;
  end Wait_Cleanup;
end controller;

procedure T1(params: param) is
  X : Exception_ID;
  Decision : Vote_t;
begin
  select
    Controller.Wait_Abort(X);
    raise_exception(X);
  then abort
    begin
      -- code to implement atomic action
      Controller.Done; --signal completion
    exception
      when E: others =>
        Controller.Signal_Abort
                  (Exception_Identity(E));
    end;
  end select;
exception
   -- if any exception is raised during
   -- the action all tasks must participate
   -- in the recovery
   when E: others =>
     -- Exception_Identity(E) has been
     -- raised in all tasks

     -- handle exception
     if handled_ok then
       Controller.Cleanup(Commit, Decision);
     else
       Controller.Cleanup(Aborted, Decision);
     end if;
     if decision = aborted then
       raise atomic_action_failure;
     end if;
end T1;

procedure T2(params : param) is ...;
procedure T3(params : param) is ...;

end action;
```

Each component of the action (*T1*, *T2*, and *T3*) has identical structure. The component executes a select statement with an abortable part. The triggering event is signalled by the *controller* protected object if any component indicates that an exception has been raised and not handled locally in one of the components. The abortable part contains the actual code of the component. If this code executes without incident, the *controller* is informed that this component is ready to commit the action.

If any exceptions are raised during the abortable part, the *controller* is informed and the identity

of the exception passed.

If the *controller* has received notification of an unhandled exception, it releases all tasks waiting on the *Wait_Abort* triggering event (any task late in arriving will receive the event immediately it tries to enter into its select statement). The tasks have their abortable parts aborted (if started), and the exception is raised in each task by the statement after the entry call to the controller. If the exception is successfully handled by the component, the task indicates that it is prepared to commit the action. If not, then it indicates that the action must be aborted. If any task indicates that the action is to be aborted, then all tasks will raise the exception *Atomic_Action_Failure*. Figure 1 shows the approach using a simply state transition diagram.

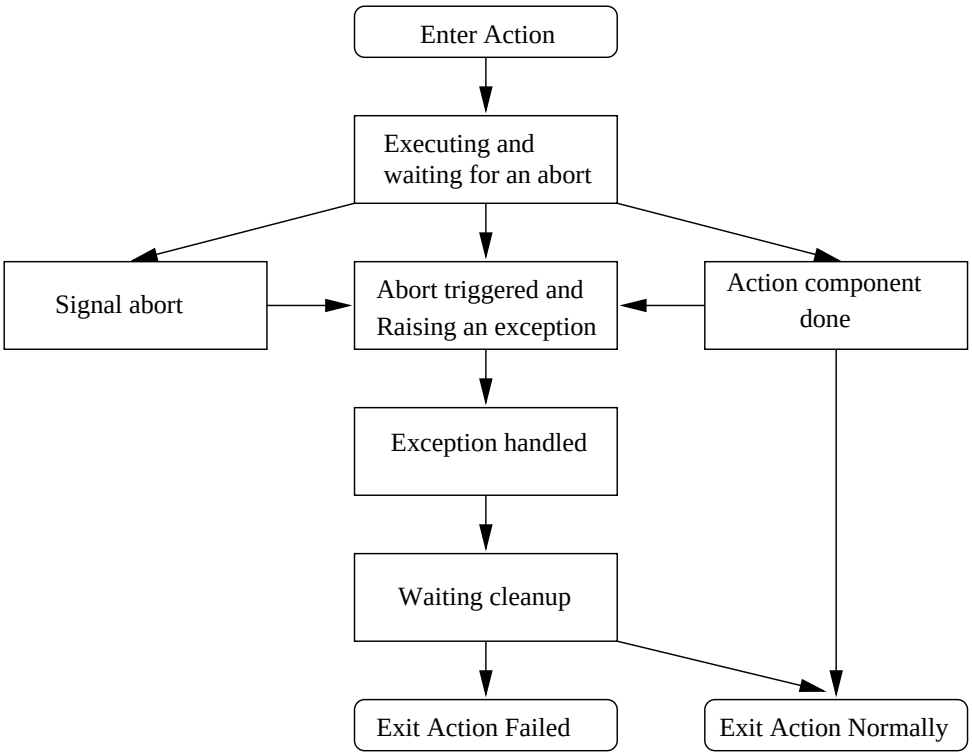

Figure 1: Simple state transition diagram illustrating Atomic Action with forward error recovery for the system with two tasks

3.3 Modelling the Ada Implementation in P/T nets

We now consider Petri nets for this Ada code. We first look at ordinary P/T nets, i.e. nets without token typing (colouring). Each of the client tasks will have an identical PN, specialised only in its labelling of transitions and places. The controller will also be modelled as a single Petri net. Our graphical support for capturing the Petri nets is a Petri net editor PED [20], which allows hierarchical and fragmented construction of P/T nets, and their export to an extensive range of formats including those accepted by analysis tools like PEP and INA. Figure 2 presents the task model (a) and the controller model (b).

Places and transitions which are not shaded, such as start1 and arr1 are individual for the task net (here we show the net for Task 1). Those places and transitions which are shaded are so called (in PED) logical places and transitions – they are used to interconnect subnets to form larger nets. In other words, by declaring places or transitions in different subnets as logical in PED, we virtually merge such places and transitions in the overall net provided that they have

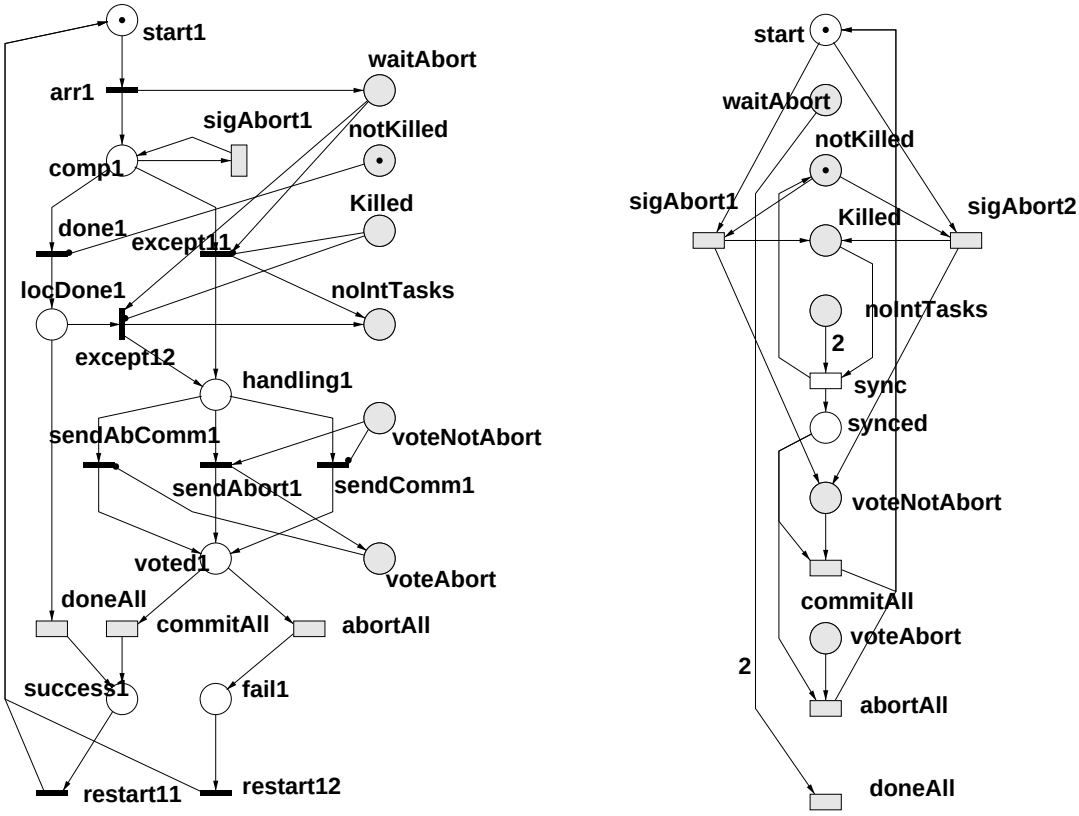

Figure 2: P/T net models: (a) Task model (b) Controller model

the same label, e.g. waitAbort and sigAbort1. Note that the net models use the so called test or read-only arcs (arcs with a black dot at the transition end), and weighted arcs. The former are used to show the fact that transitions in the task net can test the state of shared variable, such es e.g. *Killed*, which is modelled by two complementary (cannot be simultaneosuly marked with a token) places notKilled and Killed in the controller net.

Our basic idea of modelling the Ada code for the Atomic Action behaviour with P/T nets is as follows. We represent states of each task as (unshaded) places and key actions local for the task as unshaded (solid bars) transitions. Arriving in the Atomic Action by the task is represented by transition arr1. This also generates a token in the place waitAbort, which belongs to the controller and counts the number of tasks that have actually entered the Atomic Action. The place labelled com1 corresponds to the state of the task in which the task performs normal computation. From this state the task may either: (a) execute transition done1 and go to the Local Done state of normal completion of the action (place locDone1), or (b) it may raise an exception by firing transition sigAbort1 (this corresponds to executing the *Signal_Abort* procedure, which switches the state of the *Killed* flag from *false* to *true* – a token is toggled from place notKilled to Killed), or (c) it may be forced to go to the Error-Handling state (place handling1), either from the Normal Computation state or from the the Local Done state because of some task's (even itself) rasing an exception, in which case transition except12 will be fired.

Subsequent action of the task depends on whether the task ends in the Local Done or in the Error-Handling state. If the former, the task provides a condition for the controller to fire a shared transition doneAll (corresponding to the execution of the *Done* entry by all tasks). If the task is in the Error-Handling state, it handles the exception and depending on the result of the handling it votes either for Action Commit or Action Abort.

The voting mechanism used in Atomic Actions allows one task voting for Abort to force the

entire operation into Failure. In our Petri net model, this is achieved by using the following three transitions sendAbort1, sendComm1 or sendAbComm1, individual for the task. These transitions are connected to two complementary places voteNotAbort and voteAbort in the controller net. Initially, when the voting begins, a token is assumed to be placed into place voteNotAbort. While none of the tasks votes for Abort, the token remains in this place, and if the task votes for Commit (this corresponds of the *handling_ok* flag being set in the task), transition sendComm1 fires due to the reading arc from place voteNotAbort. As soon as one of the tasks votes for Abort the token is switched from it fires transition sendAbort1, which toggles the token from voteNotAbort to voteAbort in the controller. This corresponds to assigning the state of the global flag *Final_result* to *aborted* in the *Cleanup* entry. After that, in all tasks, regardless of their individual voting, transition sendAbComm1 will fire due to the reading arc from place voteAbort.

The voting is complete when the task is in the state where it is ready to check the value of the *decision* flag. This corresponds to a token in the voted1 place. At this point all tasks synchronise on firing shared transitions commitAll or abortAll, which are respectively preconditioned by the controller's places voteNotAbort and voteAbort. If the former fires it puts a token in the local success1 place, if the latter the local fail1 is marked. After that the task fires one of the two possible restart transitions which corresponds to bringing the task to the state where it is ready to execute the Atomic Action again.

Using the PED tool we constructed the model of the system from the task and controller fragments. Once the appropriate places and transitions are merged the actual behavioural interaction between task and controller is achieved through the following two main mechanisms:

- (i) synchronisation on shared transitions, which is similar to rendez-vous (blocking) synchronisation, and
- (ii) communication via shared places, which is similar to asynchronous (non-blocking) communication.

3.4 Verification of the P/T-net model

This P/T net model of the Ada code can be exported from PED to analysis tools, such as INA or PEP. We used PEP, in which we could simulate the token game and perform reachabilty analysis to verify by Model Checking the key properties of the algorithm. First, if 'Task1' is in place success1 then it must not be possible for any of the other tasks (say 2) to be in fail2. This is presented to the reachability analysis tool by the following logic statement:

```
success1,fail2
```

This test gives the <NO> result, i.e. such a marking in which these two places are marked is not reachable.

Similarly, to the test for reachability of a marking in which both tasks end in success state:

```
success1,success2
```

the tool reacts with <YES> and produces:

```
_SEQUENCE:
arr2,done2,arr1,done1,doneAll
```

which is a firing sequence leading to the global success state.

When setting the option `Calculate all paths` to true, the tool produces the following list of firing sequences:

```
_SEQUENCE:
arr2,done2,arr1,done1,doneAll
arr2,arr1,done2,done1,doneAll
arr1,arr2,done2,done1,doneAll
arr1,done1,arr2,done2,doneAll
arr2,arr1,done1,done2,doneAll
arr1,arr2,done1,done2,doneAll
```

This set, however, includes only those paths which go through the locDone states, but not those which are the result of succcesful handling and overall Commit voting. This is caused by the fact the system searches for all paths satisfying the shortest length criterion.

The effect of a coherent error handling can be tested by:

```
fail1,fail2
```

This results in:

```
_SEQUENCE:
arr1,done1,arr2,sigAbort2,except21,sendAbort2,except12,sendAbComm1,sync,abortAll
arr2,arr1,done1,sigAbort2,except21,sendAbort2,except12,sendAbComm1,sync,abortAll
...
```

all together over 600 paths. These assertions imply inconsistency is not possible.

We have also used tool INA to verify the various behavioural (safety and liveness) properties. The results of this analysis are:

```
Safety Properties:
Safe - No
Bounded - Yes
Dead State Reachable - No
Covered by Transition-Invariants - Yes

Resettable, reversable (to home state) - Yes
Dead transitions exist - No
Live - Yes
Live and Safe - No
```

The computed reachability graph has 76 states.

The INA tool allows to state properties in the form of CTL (Computational Tree Logic) [12] formulas. We can formulate properties of interest, such as whether there exists a path which leads to a state where one task ends in success while the other in fail:

```
EF((P18 &P21 )V(P19 &P20 ))
```

Here P18 (P19) stands for success1 (success2) and P21(P20) for fail2 (fail1). The result of the check is:

```
s1 sat EF((P18 &P21 )V(P19 &P20 )): FALSE
```

Another intersting property would be, whether there is a path that leads to a state in which both tasks end in success but the flag Killed (place P7 below) has been set to true:

```
s1 sat EF(P7 &(P18 &P19 )): FALSE
```

For comparison, we have tried a modified net model for a task – we omitted a read arc leading to transition done1 which tests flag notKilled. This modification may correspond to allowing the code for a task to be non-sequential – a task may signal abort and at the same time pass to Local Done (the effect of inertia or delay in reacting to the abort). Interestingly, such a modification does not lead to the violation of deadlock-freeness or the property of both tasks ending either in success or fail. But for the last property above it returns:

```
s1 sat EF(P7 &(P18 &P19 )): TRUE
```

4 CPN Modelling and Analysis

We also attempted modelling of the Atomic Actions using Coloured Petri nets (CPNs) and analysed the model using the Design/CPN tool. The three main CPNs for the model are shown in Figures 3, 4 and 5.

They capture the system hierarchically, as a composition of the controller and task nets. Due to the ability of CPNs to distinguish objects by their token colours and values, we can use the same net structure for all tasks and encode individual tasks simply by their token values. Another advantage of this type of modelling is that we can parameterise a system model with n tasks and analyse it for different number of tasks by simply setting the n parameter to an appropriate value.

The list of colour definitions (with parameter $n = |Tasks|$ set to 2) is:

```
val n=2;
color Flag = bool;
color Taskn = index task with 1..n;
color Task = record tsk : Taskn * flg : Flag;
color Comp = Task;
color Handle = Task;
color Voted = Task;
color Done = Task;
color Signal = with s;
color Wait = Signal;
color Sync = Signal;
color Vote = Signal;
color Test = Signal;
```

```
var ar, re, ts : Task;
var cp, sn, sn_ : Comp;
var ca, vt : Vote;
var va, vc : Voted;
var ab, dw, ex, sy, wt : Wait;
var ha, sa, sc : Handle;
var da, dc, dn : Done;
var aa : Sync;
var te : Test;

val init_task = 1`{tsk=task(1),flg=true}++
1`{tsk=task(2),flg=true};
```

Here are the results of the analysis using Design/CPN.

From the Design/CPN **Statistics** it can be seen that the O-graph (Occurrence Graph) for $n = |Tasks| = 2$ has 63 nodes and 114 arcs. The number of strongly connected components (Scc-graph) are less than the O-graph nodes implying an infinite occurrence sequence exists.

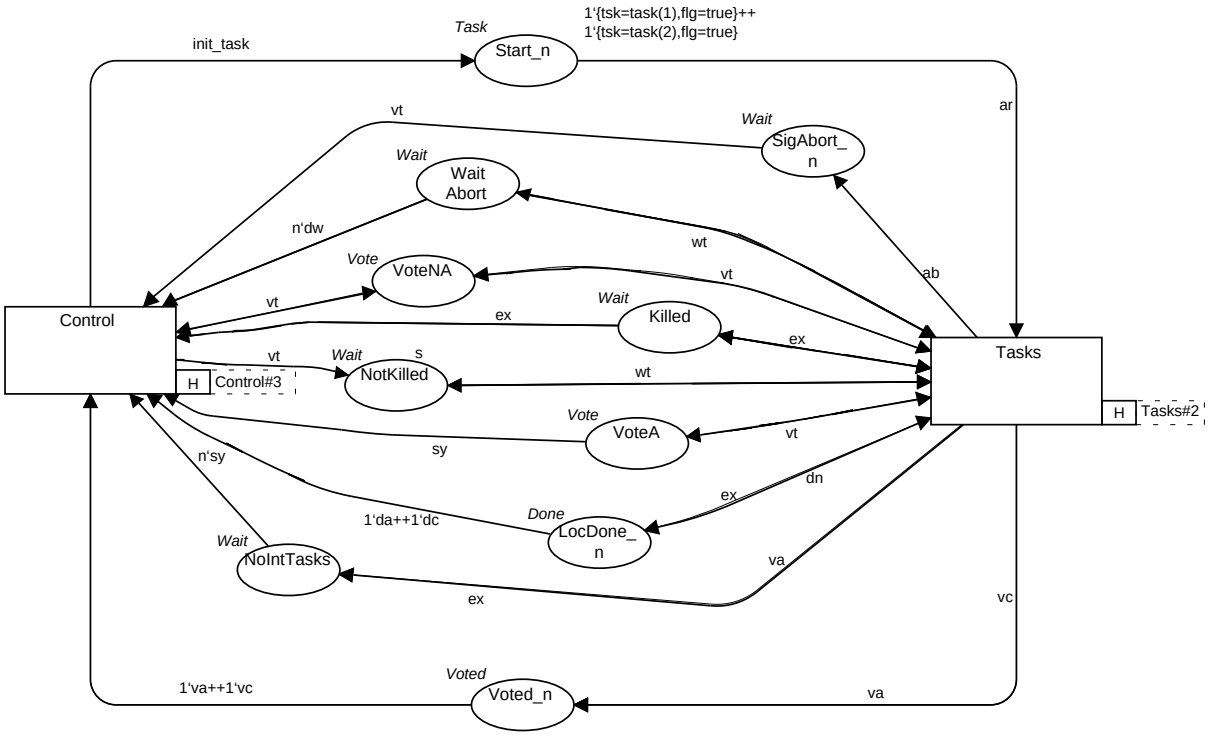

Figure 3: CPN model of the Atomic Action: Top Hierarchy level

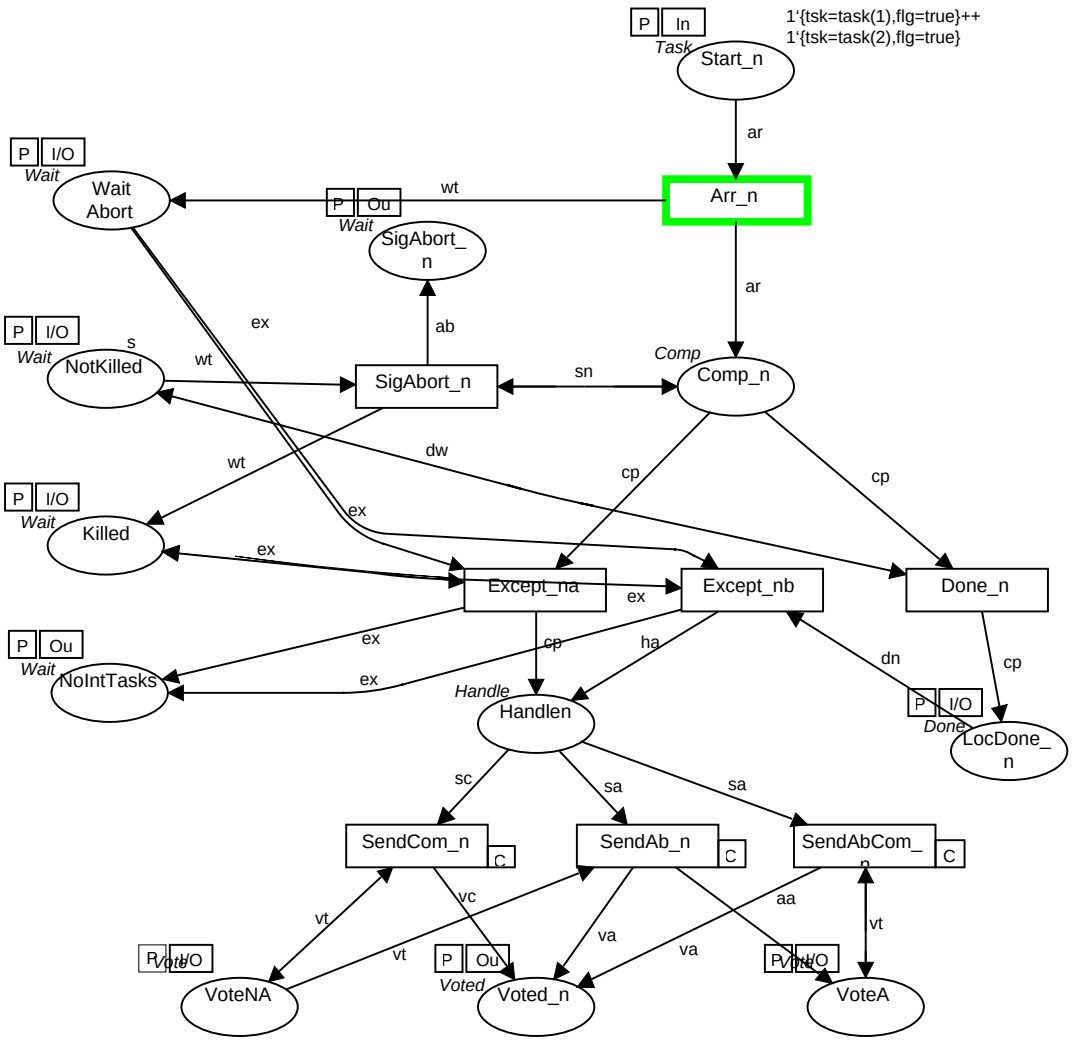

Figure 4: CPN model of Tasks

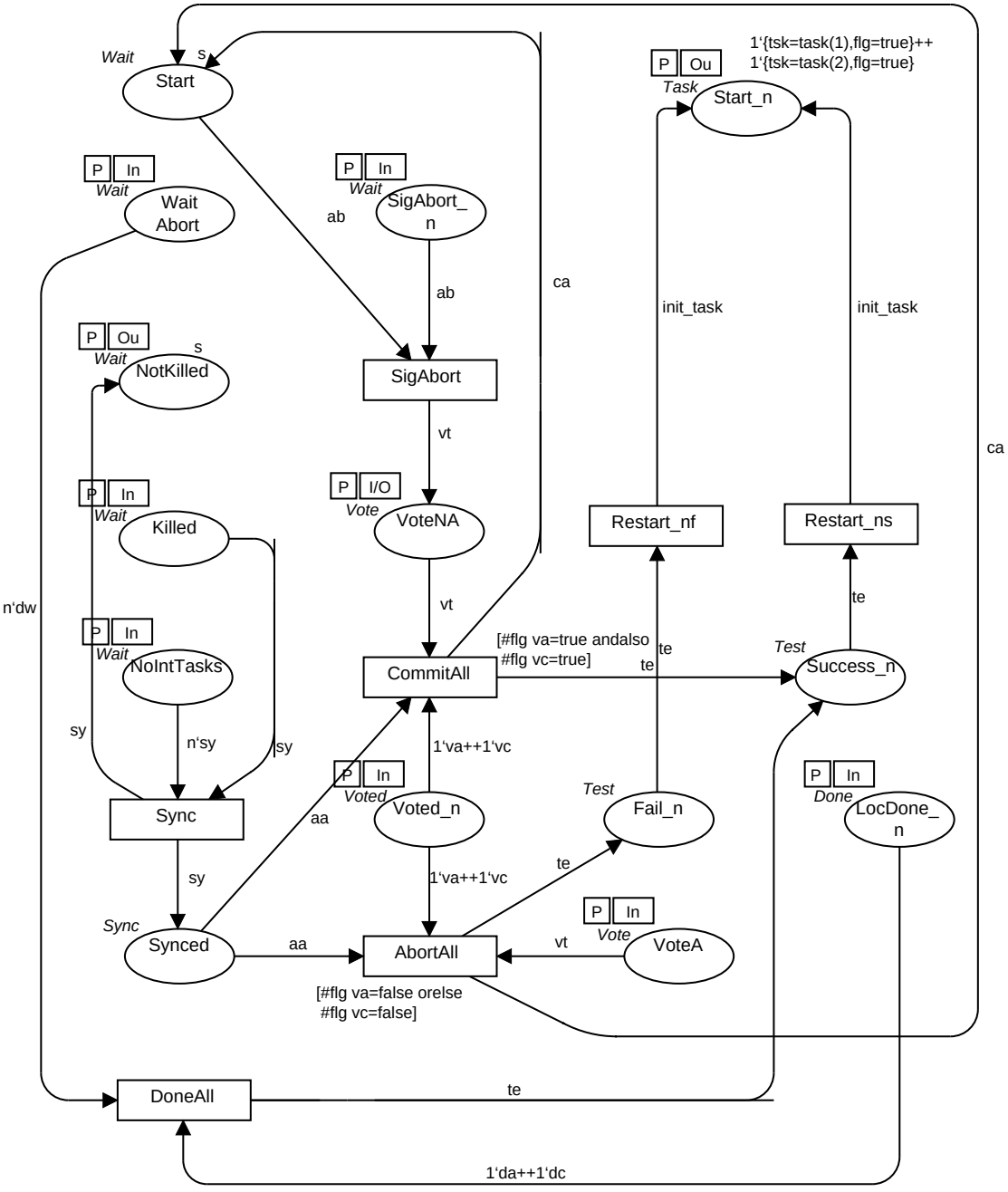

Figure 5: CPN model of Controller

```
Statistics
----------
  Occurrence Graph             Scc Graph
    Nodes:  63                     Nodes:  13
    Arcs:   114                    Arcs:   14
    Secs:   0                      Secs:   0
    Status: Full
```

For the Design/CPN **Boundedness Properties** Integer bounds are as expected. The Signal nodes can never be more than one and the Task nodes never exceed two. We show some of the Best Upper Mult-set Bounds to show the task and signal distribution. The Best Lower Multi-set Bounds are all empty.

```
Boundedness Properties
----------------------------------------
  Best Integers Bounds    Upper      Lower
  Control'Fail_n 1        1          0
  Control'Killed 1        1          0
  Control'LocDone_n 1     2          0
 ...

  Tasks'LocDone_n 1       2          0
  Tasks'NoIntTasks 1      2          0
  Tasks'NotKilled 1       1          0
  Tasks'SigAbort_n 1      1          0
  Tasks'Start_n 1         2          0


Best Upper Multi-set Bounds:
  Control'Fail_n 1     1`s
  Control'Killed 1     1`s
  Control'LocDone_n 1 1`{tsk = task(1),flg = true}++ 1`{tsk = task(2),flg = true}
  Control'NoIntTasks 1
                       2`s
...

  Tasks'Killed 1       1`s
  Tasks'LocDone_n 1    1`{tsk = task(1),flg = true}++ 1`{tsk = task(2),flg = true}
  Tasks'NoIntTasks 1   2`s
  Tasks'NotKilled 1    1`s
  Tasks'SigAbort_n 1   1`s
  Tasks'Start_n 1      1`{tsk = task(1),flg = true}++ 1`{tsk = task(2),flg = true}


Best Lower Multi-set Bounds:
  All    empty
```

The **Home Properties** show that there are no Home Markings, which implies that it is possible to reach any marking from any other marking in the O-graph. The **Liveness Properties** which show there are no Dead Markings. The **Fairness** Properties of the O-graph are shown below. Only `Arr_n` is Impartial which implies that repetition requires its occurrence.

```
Home Properties
-------------------------
  Home Markings:  None
```

```
Liveness Properties                 Fairness Properties
-------------------------------     ------------------------
  Dead Markings:  None                Control'AbortAll 1     Fair
  Live Transitions Instances:         Control'CommitAll 1    Fair
                                      Control'DoneAll 1      Fair
  Control'AbortAll 1                  Control'Restart_nf 1   Fair
  Control'CommitAll 1                 Control'Restart_ns 1   Fair
  Control'Restart_nf 1                Control'SigAbort 1     Fair
  Control'Restart_ns 1                Control'Sync 1         Fair
  Control'SigAbort 1                  Tasks'Arr_n 1          Impartial
  Control'Sync 1                      Tasks'Done_n 1         Just
  Tasks'Arr_n 1                       Tasks'Except_na 1      Fair
  Tasks'Except_na 1                   Tasks'Except_nb 1      Fair
  Tasks'SendAbCom_n 1                 Tasks'SendAbCom_n 1    Fair
  Tasks'SendAb_n 1                    Tasks'SendAb_n 1       Just
  Tasks'SendCom_n 1                   Tasks'SendCom_n 1      Just
  Tasks'SigAbort_n 1                  Tasks'SigAbort_n 1     Just
```

The following are examples of the testing of more specific properties formulated as **Queries** to O-graph and its nodes. These queries are based on functions that are defined in ML.

Function `Success_` tests all markings in which the `Success_n` node is active.

```
Function
---------------------------------------
fun Success_ (s: Test) : Node list
= PredAllNodes (fn n =>
cf(s, Mark.Control'Success_n 1 n) > 0);
---------------------------------------
```

Function `Fail_` tests all markings in which the `Fail_n` node is active.

```
Function
---------------------------------------
fun Fail_ (s: Test) : Node list
= PredAllNodes (fn n =>
cf(s, Mark.Control'Fail_n 1 n) > 0);
---------------------------------------
```

Similarly, we define functions `fun NotKilled (s: Test) : Node list` and `fun Killed (s: Test) : Node list .`

Now we apply functions `Success` and `Fail` from above and test for exclusion.

```
Test                                    Result
---------------------------------------     ------------------------------------
Success_(s);                            val it = [29] : Node list
Fail_(s);                               val it = [63] : Node list
Success_(s) <> Fail_(s);                val it = true : bool
length(Success_(s))+length(Fail_(s))=2; val it = true : bool
---------------------------------------     ------------------------------------
```

We use function `Start_n` to test for initialization.

```
Function and Test                          Result
-----------------------------------------  -----------------------------------
fun Start_n (s: Test) : Node list          val Start_n = fn : Test -> Node list
= PredAllNodes (fn n =>
cf({tsk=task(1),flg=true},
Mark.Control'Start_n 1 n)+
cf({tsk=task(2),flg=true},
Mark.Control'Start_n 1 n) > 1);
Stater_n(s);                               val it = [1] : Node list
-----------------------------------------  -----------------------------------
```

The `SearchNodes` function is used to test for specific occurrences for the `Success_n` node (node 29) to be activated. It verifies that there are only two possible occurrences that can lead to this happening, i.e. one from `Voted` causing `Commitall` (node 60) or `Doneall` (node 20).

```
Functions and Tests                        Result
-----------------------------------------  -----------------------------------
Success_(s);                               val it = [29] : Node list
InNodes(29);                               val it = [60,20] : Node list
OutNodes(60);OutNodes(20);                 val it = [29] : Node list
                                           val it = [29] : Node list

fun chk(nd,x) = nd;                        val chk = fn : 'a * 'b -> 'a

SearchNodes(EntireGraph,fn n =>            val it = 60 : Node
cf({tsk=task(1),flg=true},
Mark.Control'Voted_n 1 n)+
cf({tsk=task(2),flg=true},
Mark.Control'Voted_n 1 n)>
1,NoLimit,fn n => n,0,chk);

SearchNodes(EntireGraph,fn n =>            val it = 20 : Node
cf({tsk=task(1),flg=true},
Mark.Control'LocDone_n 1 n)+
cf({tsk=task(2),flg=true},
Mark.Control'LocDone_n 1 n)>i
1,NoLimit,fn n => n,0,chk);
-----------------------------------------  -----------------------------------
```

Finally, the following table shows how the Occurance graph increases as the number of Tasks is increased.

Size of Occurance Graph with number of Tasks					
|Tasks|	Nodes	Arcs	|Tasks|	Nodes	Arcs
2	63	114	5	7568	25883
3	298	689	6	39331	158444
4	1481	4220	7	207667	969677

5 Conclusion

We have shown that a relatively complicated Ada program using tasking can be modelled and verified using Petri nets (ordinary P/T nets and Coloured) and Model Checking. This significantly improves confidence in the correctness of higher-level abstraction such as atomic actions.

This paper is only a preliminary attempt in pursuing our chosen direction of research, in which we would like to develop a more comprehensive methodology for verifying high-integrity systems built of Atomic Actions and implemented in Ada 95.

The major new aspects of this work, which also reveal the potentially exploitable advantages of the Petri net approach over the State Machine one [10], are:

- Refinement of both states and transitions;
- Analysis of behaviour at the true concurrency and causality level;
- High-level aspects of modelling, such as parametrisation, are possible using high-level Petri nets.

For example, if refinement with threads (e.g., task spawning), recursive atomic actions, etc. were possible in the modelled systems, then Petri nets would provide a much more efficient way of modelling than state machines.

We have only shown the way of modelling interaction mechanisms at the semantical level. Part of the intended future work would be to adopt the existing or develop new methods of extracting Petri nets from the Ada 95 syntax.

Although this paper has not introduced real-time issues, the choice of tool and modelling technique implies that the approach can be extended to a timed Petri net approach.

References

[1] Ada 95: Information technology - Programming languages - Ada. Language and Standard Libraries. ISO/IEC 8652:1995(E), Intermetrics, Inc., 1995.

[2] E.Best: Partial Order Verification with PEP. Proc. of POMIV'96, Partial Order Methods in Verification. G. Holzmann, D. Peled, V. Pratt (eds), Am. Math. Soc. (1997) 305-328.

[3] E.Best and B.Grahlmann: PEP - more than a Petri Net Tool. Proc. of Tools and Algorithms for the Construction and Analysis of Systems, 2nd International Workshop, TACAS'96, Passau, March 1996, T. Margaria, B. Steffen (eds). Springer-Verlag, Lecture Notes in Computer Science 1055, Springer-Verlag (1996) 397-401.

[4] E.Best, R.Devillers, J.Hall: The Petri Box Calculus: a New Causal Algebra with Multilabel Communication. Advances in Petri Nets 1992, Lecture Notes in Computer Science 609, Springer-Verlag (1992) 21-69.

[5] E.Best, R.Devillers, and M.Koutny: Petri Nets, Process Algebras and Concurrent Programming Languages. Lectures on Petri Nets II: Applications, Advances in Petri Nets. Lecture Notes in Computer Science 1492, Springer-Verlag (1998) 1-84.

[6] E.Best, H.Fleischhack, W.Fraczak, R.P.Hopkins, H.Klaudel and E.Pelz: M-nets: An Algebra of High-level Petri Nets, with an Application to the Semantics of Concurrent Programming Languages. Acta Informatica 35 (1998) 813-857.

[7] R.E.Bryant: Symbolic Boolean Manipulation with Ordered Binary-decision Diagrams. ACM Computing Surveys 24 (1992) 293-318.

[8] D.Buchs, C.Buffard and P.Racloz: Modelling and Validation of Tasks with Algebraic Structured Nets. Proc. of Ada in Europe'95, Lecture Notes in Computer Science 1031, Springer-Verlag (1995) 284-297.

[9] A.Burns and A.J.Wellings: Real-Time Systems and Programming Languages (Second edition) Addison Wesley (1996).

[10] A.Burns and A.J.Wellings: How to Verify Concurrent Ada Programs - The Application of Model Checking. Ada Letters, Volume XIX, Number 2 (1999) 78-83.

[11] R.H.Campbell and B.Randell: Error Recovery in Asynchronous Systems. IEEE Transactions on Software Engineering SE-12 (1986) 811-826.

[12] E.M.Clarke and E.A. Emerson: Synthesis of synchronization skeletons for branching time temporal logic. In Dexter Kozen, editor, *Logic of Programs: Workshop*, LNCS, vol. 131, Springer-Verlag, 1981.

[13] E.M.Clarke and J.Wing: Formal Methods: State of the Art and Future Directions. Report, Carnegie Mellon University (June 1996).

[14] J.Esparza: Model Checking Based on Branching Processes. Science of Comp. Prog. 23, 151-195 (1994).

[15] R.K. Gedela and S.M. Shatz. Modeling of advanced tasking in Ada-95: a Petri net perspective. Proc. 2-nd Int. Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'97), Boston, MA, pp. 4-14 (May 1997).

[16] P.Godefroid and P.Wolper: A Partial Approach to Model Checking. Information and Computation, 110(2), 305-326 (1994).

[17] K.Jensen:Coloured Petri Nets. Basic Concepts. EATCS Monographs on Theoretical Computer Science (1992).

[18] K.H.Kim: Approaches to Mechanization of the Conversation Scheme Based on Monitors. IEEE Transactions on Software Engineering SE-8 (1982) 189-197.

[19] D.B.Lomet: Process Structuring, Synchronisation and Recovery using Atomic Actions. Proc. of ACM Conference Language Design for Reliable Software. SIGPLAN (1977) 128-137.

[20] *PED*. `http://www-dssz.Informatik.TU-Cottbus.DE/~wwwdssz/` – the home page of PED (a Hierachical Petri Net Editor).

[21] D. Peled: Combining Partial Order Reductions with On-the-fly Model-checking. Formal Methods in Systems design 8(1), 39-64 (1996).

[22] *PEP*. `http://www.informatik.uni-hildesheim.de/~pep/HomePage.html` – the home page of PEP (a Programming Environment Based of Petri Nets).

[23] M. Pezze, R.N. Taylor and M. Young: Graph Models for Reachability Analysis of Concurrent Programs. ACM Transactions on Software Engineering and Methodology 4/2 (April 1995) 171-213.

[24] B. Randell: System Structure for Software Fault Tolerance. IEEE Trans. on Software Engineering 1(2) 220-232 (1975).

[25] B. Randell, P. Lee and P. Treleaven: Reliability issues in computing systems design. ACM Computing Surveys 10(2): 123-165 (1978).

[26] W.Reisig: Petri Nets. An Introduction. Springer-Verlag, EATCS Monographs on Theoretical Computer Science Vol.3, (1985).

[27] S. Roch and P.H. Starke: INA: Integrated Net Analyzer, Version 2.2, Manual Humboldt-Univerität zu Berlin, Instutut für Informatik, April 1999.

[28] S.M. Shatz, S. Tu, T. Murata and S. Duri: An Application of Petri Net Reduction for Ada Tasking Deadlock Analysis. IEEE Trans. on Parallel and Distributed Systems 7 (12), 1309-1324 (December 1996).

[29] S.K.Shrivastava, G.N.Dixon and G.D.Parrington: An Overview of the Arjuna Distributed Programming System. IEEE Software 8 (1991) 66-73.

[30] S.Tu, S.M.Shatz and T.Murata: Theory and Application of Petri Net Reduction for Ada-Tasking Feadlock Analysis. TR 91-15, EECS Dept., Univ. of Illinois, Chicago (1991).

[31] A.Valmari: The State Explosion Problem. Lectures on Petri Nets II: Applications, Advances in Petri Nets. Lecture Notes in Computer Science 1492, Springer-Verlag (1998) 429-528.

[32] A.J.Wellings and A.Burns: Implementing Atomic Actions in Ada 95, IEEE Transactions on Software Engineering 23 (1996) 107-123.

[33] The Home page of Petri net Tools on the Web: `http://www.daimi.aau.dk/~petrinet/tools/`

[34] The Home page of the Design/CPN tool: `http://www.daimi.au.dk/designCPN/`