On Developing and Verifying Design Abstractions for Reliable Concurrent Programming in Ada

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

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, which allows building large multi-tasking applications. Much of the complexity of these applications stems from the interactions between the tasks. Design abstractions (such as atomic actions, conversations etc.) have been proposed to deal with such complexity. This paper argues that Petri nets offer a promising, tool-supported, technique for checking the logical correctness of abstractions. 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.

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, software fault-tolerance has adopted the notion of conversations [18] and atomic actions [7, 13] 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 [12], 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++ [24]. 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 [4]. 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 [26]. 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. E.g., [26] showed how Ada 95 can be used to implement Atomic Actions. 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, namely: tasks, Asynchronous Transfer of Control (ATC), protected types, requeue, exceptions and controlled types.

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 [5]. 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 [19] and high level (e.g. coloured nets [11]) 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. [23, 17]. 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 [23] and decomposition [17] of 'Ada nets' have been proposed.

Our research aims at developing a set of design abstractions for reliable concurrent programming in Ada, and at applying Petri nets to model and verify them, using available tools, such as PEP and Design/CPN [6]. Furthermore, we distinguish between these abstractions and application code which uses them. 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 [10].

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 Petri net model is introduced and verified in Section 3. Section 4 discusses design abstractions to be developed, together with their properties. 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 [9].

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. 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.

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 [27]. In our study, we used three relatively mature tools. One is PEP [2], 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) [20]. The third is Design/CPN [28], which is based on the Coloured Petri nets and has extensive facilities for simulation and occurrence (reachbility) graph analysis.

3 Model of Simple Atomic Actions

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 [13]. 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 section we will focus only on forward error recovery using exception handling [7]. 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.

Atomic Actions in Ada. To show how atomic actions can be programmed in Ada [26], 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 [12].

```
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;
```

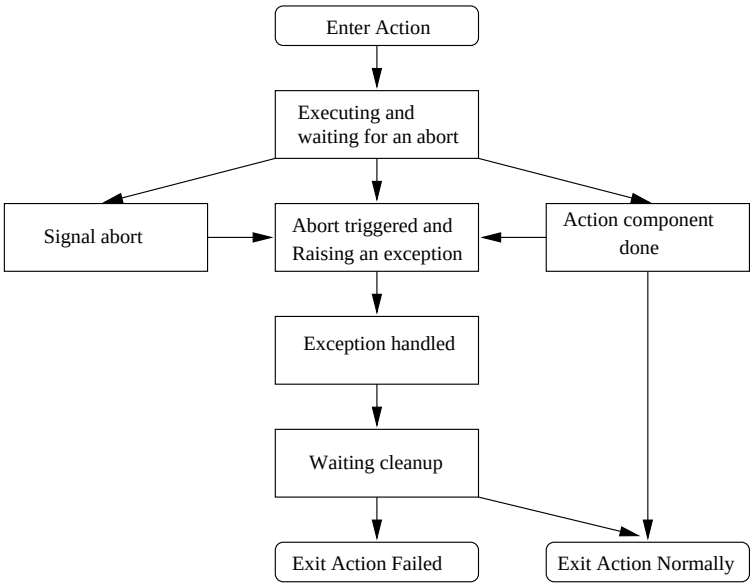

Figure 1: Simple state transition diagram illustrating Atomic Action with forward error recovery for the system with two tasks

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.

3.1 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 [15], 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 the same label, e.g.

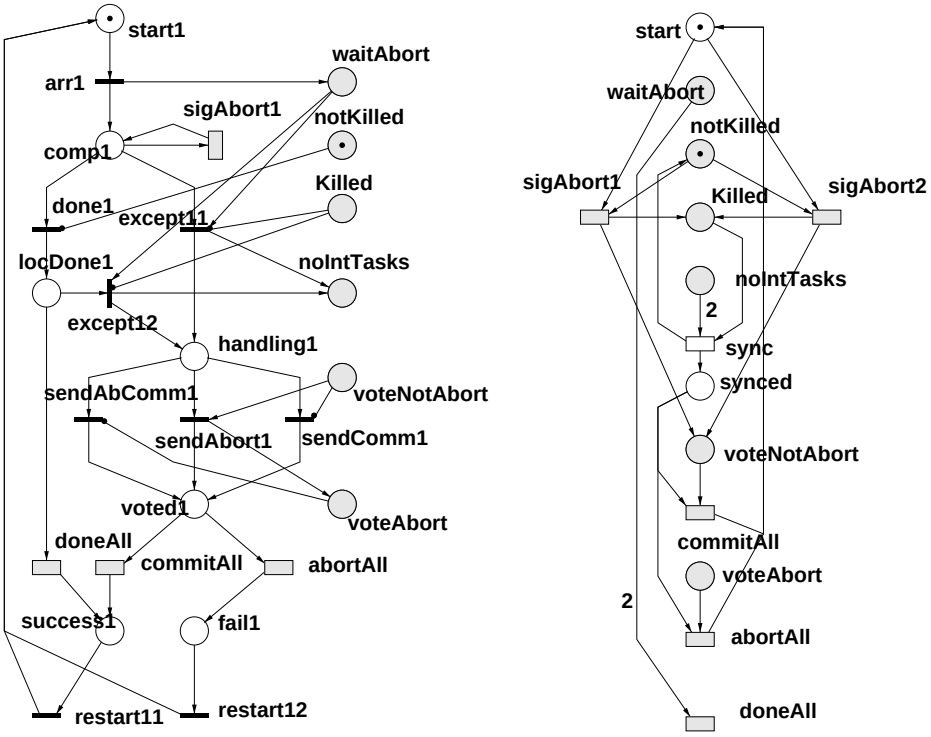

Figure 2: P/T net models: (a) Task model (b) Controller model

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.

3.2 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) [8] 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
```

Our preliminary results on using Colured Petri nets for modelling and analysing the Atomic Action scheme were recently reported in [6].

4 Developing Design Abstractions

In this section we outline abstractions which are important for reliable concurrent programming in Ada and our reasons for choosing them. Our understanding has been built on analysing the existing schemes supporting some abstractions. We believe that it is vital to develop a systematic approach for choosing and developing such abstractions. In particular, this choice has to be driven by our ability to check the abstractions (this might depend on the Ada features used to develop them). Other important issues to address are expressing general abstraction properties in the way in which they can be formally checked and developing guides for applying the abstractions correctly. Our first experiments with modelling and checking the atomic action abstraction have contributed to better understanding of these topics [6]. This section discusses some preliminary results of the on-going research.

Abstraction Overview There has been a considerable body of research on developing reliable abstractions in Ada (see, for example, [26, 21, 14]. The first stage of our research builds on analyses of several existing schemes as possible candidates for formalisation and model checking. These include conversation schemes of different types (e.g. concurrent recovery blocks), different atomic action schemes (with backward or forward error recovery, action nesting, detecting the entry and the exit deserters, with or without entry synchronisation), N-version programming with concurrent version execution [22], replicated systems [25], etc.

The following step is to come up with a systematic approach for developing such abstractions. The intention here is to propose a range of schemes which work under different fault assumptions (including software design faults, environmental faults, transient errors, exceptions raised by the underlying support, hardware faults) and which are suitable for designing concurrent systems and applications of different types.

Our further intention is to propose a set of basic abstractions which are useful for developing both reliable decentralised applications and decentralised controls for the above-mentioned reliable abstractions such as atomic actions and replicated systems [25]. These can include some of the following mechanisms: message ordering, broadcast protocols, agreement protocols, group membership support.

Another important avenue to explore here is the development of new abstractions suitable for designing real-time Ada systems (e.g. periodic and sporadic tasks, scheduling) and extending the abstractions discussed above for dealing with time concerns (e.g. by including time constraints).

Principles. In this research we are following several general principles:

- making the schemes implementing the design abstractions as reusable and as general as possible;
- relying on basic building blocks which can be used for designing several abstractions (for example, features detecting deserter processes are common for conversations, atomic actions and N-version programming);
- separating the re-usable code from the application code as much as possible.

Special measures have to be taken to make modelling and checking simpler. This can be done by applying an evolutionary development with developing the right abstractions in the right order. For example, paper [14] presents a basic distributed atomic action scheme and its several extensions, which, we believe, are much easier to model and to check when the basic scheme is checked. Model checking can be facilitated by applying several architectural solutions:

- composability: developing several simple basic abstractions and demonstrating how more complex ones can be composed;
- separating concerns: developing abstractions which are concerned with different orthogonal properties related to reliability (and dependability in general). This can make the construction of more complex abstractions and their verification simpler;
- layering: building new abstractions on the top of existing and verified ones which are designed as the underlying service layers.

Another important issue to be taken into account is developing a better understanding of how different Ada constructs can complicate the checking and cause state explosion. This will allow us to program abstractions which are easier to check. One of the solutions could be to define an Ada subset (although it is clear for us that the Ravenscar subset [3] is too simple for our needs). Furthermore, composability might make it difficult to subset. For example, nested ATC blocks add complexity but it is unlikely that we would want to disallow them.

Some of these principles as well as the intentions outlined in the previous subsection are contradictory and finding a right balance is a very important issue in this research.

Abstraction Properties. To ensure the correctness of an abstraction we should be able to formulate and check a complete set of its properties (generally speaking, it might be reasonable to develop a necessary and sufficient set of properties). To avoid any ad hoc approaches we should rely on rigorous definitions of such abstractions. Their properties are usually defined in terms of system design and use to help programmers rather than formally. For example, hardware fault tolerance schemes often rely on the fact that data saved in a stable storage can be recovered after a crash. To verify the ability of the system to be recovered we should be able to model and to check that such schemes always save data sufficient for the restart and for guaranteeing the continuous service. Other examples of such properties are: atomicity and isolation of atomic actions (absence of

information exchange with the outside world), absence of the deserter processes in conversations, all-or-nothing effect of actions, mutual exclusion of the access to shared, correct action nesting. These properties should be modelled in a formal way suitable for checking the correctness of the abstractions. Developing systematic approaches for describing and formalising a complete set of properties for each abstraction is the only general way for ensuring their correctness.

Guides. After the correctness of the design abstractions has been demonstrated, the programmers can apply them for system design. Unfortunately this can be an error prone process because it is not supported by the compiler or run-time checks and because the correctness of the design abstractions does not mean that they are always applied correctly. Although some schemes may perform several run-time checks, it is usually not practical to develop and to use schemes which are able to detect and tolerate all possible types of misuse. This is why all schemes supporting design abstractions of interest assume that there are some rules and restrictions on using them and that the programmers follow them. This shows the importance of developing guides explaining how to apply design abstractions implemented as the concrete Ada schemes. These guides will include templates and a set of conventions for programmers. They are to be prepared by system or fault tolerance programmers. For example, it is nearly impossible to prove or to guarantee in the run time that a set of Ada tasks inside an atomic action do not exchange information with the outside world. The only practical solution is to describe how this can be achieved in a programmers' guide.

In addition it might be possible to develop some supplementary tools which work with the designed Ada code (maybe with some annotations included as comments) to check that such rules have been followed while applying the design abstractions. In this case the guides can be applied together with such tools.

5 Conclusion

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 [5], are: (i) refinement of both states and transitions; (ii) analysis of behaviour at the true concurrency and causality level; (iii) 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.

We have outlined the important aspect of the development of new design abstractions, which must be more amenable to model checking in general and use of Petri nets in particular. We have also identified a way for formulating properties of these abstractions to be checked, and proposed an approach to develop practical guidelines for applying the abstractions in real designs.

References

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

[2] 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 Int. Workshop, TACAS'96, Passau, March 1996, T. Margaria, B. Steffen (eds), LNCS 1055, Springer-Verlag (1996) 397-401.

[3] A. Burns. The Ravenscar Profile. Ada Letters, v. XIX, N 4, 1999, pp.49-52.

[4] A.Burns and A.J.Wellings: Real-Time Systems and Programming Languages (2nd ed.) Addison Wesley (1996).

[5] 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.

[6] A. Burns, A.J. Wellings, F. Burns, A.M. Koelmans, M. Koutny, A. Romanovsky, A. Yakovlev. Towards Modelling and Verification of Concurrent Ada Programs Using Petri Nets. Accepted for Int. Work. on Soft. Eng. and Petri nets (SEPN'2000), to be held within the 22nd Int. Conf. on Appl. and Theory of Petri Nets (PN'2000), June 2000, Aarhus, Denmark.

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

[8] 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.

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

[10] 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).

[11] K.Jensen:Coloured Petri Nets. Basic Concepts. EATCS Monographs on Theor. Comp. Sci. (1992).

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

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

[14] S.E. Mitchell, A.J. Wellings, A. Romanovsky. Distributed Atomic Actions in Ada 95. Computer J., v. 41, N 7, 1998, pp.486-502.

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

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

[17] 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.

[18] B. Randell: System Structure for Software Fault Tolerance. IEEE Trans. Soft. Eng. 1(2) 220-232 (1975).

[19] W.Reisig: Petri Nets. An Introduction. EATCS Monogr. on Theor. Comp. Sci., Springer-Verlag (1985).

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

[21] A. Romanovsky. A Study of Atomic Action Schemes Intended for Standard Ada. J. of Systems and Software, v. 43, 1998, pp.29-44.

[22] A. Romanovsky. Class Diversity Support in Object-Oriented Languages. J. of Systems and Software. v. 48, 1999, pp.43-57.

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

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

[25] A.J. Wellings, A. Burns. Programming Replicated Systems in Ada 95, Computer J. v. 39, N 5, 1996, pp.361-373

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

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

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