Modelling and validation of tasks with algebraic structured nets

D. Buchs C. Buffard P. Racloz
e-mail {buchs, buffard, racloz}@di.epfl.ch

Swiss Federal Institute of Technology in Lausanne
Software Engineering Laboratory
EPFL-DI-LGL
CH-1015 Lausanne
Switzerland

Abstract. Ada offers several mechanisms for expressing concurrency, like tasks and protected types. The use of concurrency can introduce significant problems which are inherent in the program's interactions or can require some particular properties. Amongst these problems or necessary properties, we can mention deadlocks, fairness and particular temporal characteristics. The modelling of concurrency behavior with tools can help prevent these problems. For this purpose, we show in this paper that programs written in Ada can be modelled using the formalism CO-OPN, based on Petri nets and algebraic specifications, which offers the possibility to select the level of abstraction of the modelling. These modelling can be used to detect the program anomalies.

Keywords: Ada, protected type, task, abstract data type, concurrency, CO-OPN, Petri net.

1 Introduction

Concurrent programs execute faster the same work than sequential programs on the same resource. Nevertheless, their complexity imply that they must be carefully designed using well defined software engineering principles such as modularity, composition techniques and behaviourally predictable properties. We mean by predictable behavior the fact that it is possible to decide some interesting properties of the considered software system and this is only possible if a formal semantics is given to the analysed program. We will see latter that our approach can be used to prove properties such as deadlock, fairness or temporal properties by modelling Ada programs using a well defined formal model called CO-OPN (Concurrent Objected Oriented Petri Nets).

Ada tasking and rendez-vous based communication provide a mechanism to implement both concurrent and distributed software. The interaction between tasks give rise to issues such as avoiding deadlocks and guaranteeing fairness of tasks scheduling. Moreover, the design of concurrent programs is much more difficult and mastering the interactions between tasks can be quite complex.

The problems inherent in concurrency are not simple to detect, thus we are interested in the modelling of Ada programs, in particular the task parts, in order to perform simulation and analysis as a mean of detecting problems. The goal of our project is to develop a concurrent object model for Ada, based on Petri nets and abstract data types, within the CO-OPN framework. Our idea is that the main aspects dealing with concurrency must be modelled through the use of Petri nets while depending on the focus of attention, particular aspects of data structures and procedures can either be modelled using Petri nets or algebraic abstract data types.

Other authors treat the transition from Ada code to an analysis model such as [DNC] and [MSS 89]. Our approach introduces the possibility to model both the communication and the data structures that are not present in the other studies.

The goal of the software engineering is the production of quality software that is delivered on time and within budget and which satisfies all its requirements [SE 90]. An important aspect of software engineering is that software production should be helped by tools. The formalism CO-OPN is supported by a computer tool including a graphical editor, a compiler, a simulator and a temporal verifier. We plan to implement in the near future a tool supporting the user guided translation of Ada programs into CO-OPN specifications. Another domain of application is the Software re-engineering which improves the understanding of the software, prepares or improves the software itself, usually for increased maintainability, reusabilty or evoluability [Arnold 93]. The software re-engineering, sometimes called in the literature software-renewal or software-renovation, is also a term applied to software development where an existing system is undergoing some degree of redevelopment.

This paper is structured as follows. Section 2 describes the CO-OPN formalism and Section 3 gives a brief overview of tasking in Ada (95). We examine the transition from Ada code to the CO-OPN model in Section 4. A short description of the different forms of analysis that may be performed on the model is presented in Section 5. Finally, concluding remarks are given in Section 6.

2 CO-OPN Model

2.1 The CO-OPN Approach

CO-OPN (Concurrent Oriented Object Petri Net) is a specification language used for the specification and the design of large concurrent systems. The last version called CO-OPN/2 extends the CO-OPN language by means of some object-oriented features such as class, inheritance and sub-typing. The object paradigm is introduced into CO-OPN in order to solve the problem of large specification management. *Encapsulation* and *abstraction* are key notions included in the model. Each class or template can be used to instantiate the objects which compose the modelled system. A CO-OPN object can be viewed as a set of *encapsulated* state variables or attributes which can be used for implementing accessible its methods. Moreover, autonomous object reaction is implemented by means of internal transitions. According to the common intuition, each object has its own behavior and evolves in concurrence with other objects. As the purpose of CO-OPN is to capture the abstract concurrent behavior of each modelling entity, the concurrency granularity is not found in the objects but rather in the methods. Intra object concurrency can be obtained by expressing the causality relation between the behavior of the different methods. The relations between objects, such as object synchronization, must also be captured using method synchronization schemes allowing for the building of hierarchies of *abstraction*.

CO-OPN uses Petri nets, which are an elegant model of true concurrency (true in opposition to interleaving semantics). Unfortunately, Petri nets have very weak modelling capabilities in term of the functional aspects and structuring capabilities. CO-OPN adds to Petri nets a synchronization scheme for each method of the objects. This approach generalizes other approaches such as, the transition fusion [DHP 91] and the hierarchical nets [Kie 89].

Real systems must include other features such as data structures with the associated operations acting upon these data. This is the reason why data structures are incorporated into Petri nets. This class is formed of Petri net specifically called high-level nets i.e. Petri net with tokens modelling by data structures instead of anonymous entities. We need an abstract formalism to describe structured tokens. In our case we chose algebraic abstract data types which combine abstraction through functions with a simple system of properties notation derived from the equality relation and the first order logic.

2.2 Petri Nets

The first component on which CO-OPN is based are the Petri nets [Reisig 85]. These nets are used to model and analyse discrete systems in which notions such as interacting components, concurrency and synchronisation play a key role. A Petri net modelling embeds both the structural aspect of the system and its evolution. Among the pleasant features embodied in Petri nets we can mention:

- the distribution of state and the change-of-state in the net,
- the common synchronization schemes such as sequentiality, non-determinism, concurrency and conflict naturally handled,
- the existence of graphical representation.

The states of a system are represented through *places*, these one can contain a varying number of *tokens*. A given distribution of tokens in the places represents a state. The events of the system are represented by transitions and the occurrence of an event corresponds to the firing of a transition which depends on the existence of a minimal number of tokens in some places of the net.

When a transition is fired, a new state is reached. This state is given by a new distribution of tokens. Informally, the tokens can be considered as resources needed to the occurrence of events. The firing of a transition consumes some resources and generates new ones. The semantics of Petri nets can be expressed by a transition system.

2.3 Abstract Data Types

The second component on which CO-OPN is based, are the abstract data types (ADT). An ADT is a representation of a data structure coupled with a set of operations which manipulate this data structure (data type) independently of any particular implementation (abstract).

The algebraic approach [EM 85] to specification considers a system in terms of abstract data types represented, in a mathematical discourse, by many-sorted algebras in combination with axioms or equations. An algebra is simply a domain provided with a set of operations. An algebraic specification $Spec =< S, F, X, Ax >$ consists of a set S of sort names, a set F of operation names including their profile, a set X of variables and a set Ax of positive conditional axioms (Horn clauses). With an algebraic approach the terms sort and type are distinguished. A type represents a domain and a set of operations whereas a sort is simply the name of a type. The axioms embody the operations properties and they are described as conditional equalities, as follows: $Condition \Rightarrow term = term'$ where $Condition$ is the conjunction of equalities $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n$.

3 Ada Tasks

Ada 95 offers several mechanisms for expressing the concurrency: tasks, which are active entities, and protected types, which can be considered as passive tasks. Tasks express concurrency and their execution can take place on one or more processors depending on the run-time implementation. Tasks allow simultaneous execution and the order of execution is undefined except when two tasks are participating in a rendez-vous, which is the only form of direct communication between tasks. A rendez-vous is asymmetric: the caller must know the called task while the called task is unaware of the identity of the caller.

Calling an entry of a task is very similar to call a procedure with the exception that the called task can decide when and how to accept the call. If the called task is not ready to accept the entry then the execution of the calling task is suspended until the called task is ready to accept the entry call. Once the called task accepts the entry the rendez-vous begins. The duration of the rendez-vous is controlled by the called task, and is equal to the duration of the accept statement.

The general mechanisms studies in the concurrency for the cooperation between tasks are either based on message passing where a task communicate with another by sending a message through the tasking kernel,or based on the explicit use of synchronisations between the involved parties.

Ada provides a concurrency model based on the asynchronous execution of each task as if each task had its own processor. In contrast to the asynchronous model of execution, a communication between two tasks takes place through a rendez-vous, which is synchronous.

4 Construction of CO-OPN Model

We can represent Ada code in the CO-OPN model with varying levels of abstraction depending upon the point of view adopted. For example we can stay very close to the code or take a much more abstract view. A program is a sequence of statements and only the necessary statements with respect to synchronisation will be considered in the model. The principal objective is the detection of errors such as deadlocks. The deadlocks do not depend only on the synchronisation skeleton of the communicating entities, but also on the data structures exchanged. Our approach is to model intertask communication while conserving the structures of the exchanged data. Data are modelled with the necessary abstract level depending on the analysis point of view. The novelty of our approach, as compared to others in this domain, is the modelling of both the communication and the data structures involved. The correspondences between Ada data structures and the CO-OPN components are shown in figure 1. For intertask communication we have retained two categories of streams. The first stream which can alter the control flow, is defined by the Ada reserved words **if**, **loop**, **requeue** and **select**, while the second constitute the rendez-vous defined by the Ada reserved word **accept** and the *entry calls*.

Ada	CO-OPN	
	Petri net	Algebraic Spec
Abstract Data Type inclusion of types		•
procedures	•	
tasks protected types	•	
functions(pure)		•
Exceptions	?	?

Fig. 1. Correspondence between Ada data structures and CO-OPN components

As we said above the main goal is to anticipate errors due to concurrency in programs. We choose to retain only the functions and procedures that are directly linked with statements expressing concurrency in Ada. We have two ways for modelling a function or a procedure, calling a task or accepting an entry call: either by a Petri net or with a transition defined in a object specification.

In this way, we keep only the necessary representation of concurrence and data. Tasks are modelled with CO-OPN objects, and the task's functions are modelled in the same way as other functions. Protected types are modelled like tasks.

The **requeue** statement can be use to complete an **accept** statement or entry body, while redirecting the corresponding entry call to a new entry queue. The requeue statement is modelled in CO-OPN a Fifo ADT (cf. figure 5). Methods must be added on the interface in order to be complete. These methods can be called by clients, they are not implemented yet. The Fifo ADT is also used to model the waiting line of entries and tasks priorities and provides functions for manipulating this waiting line.

Exceptions are not modelled in the present work, we plan to study this complicated problem in the near future.

```
Adt Standard;
Signature
  Sorts integer, boolean;
  Sorts character, float;
  Generators
  0 : → integer;
  succ : integer → integer;
  Operations
  _+_ : integer integer → integer;
  _=_ : integer integer → boolean;
  ⋮
Body
  Axioms
  0+n = n;
  (succ n)+m = succ(n+m);
  ⋮
  0=0 = true;
  0=(succ n) = false;
  (succ n)=0 = false;
  (succ n)=(succ m) = n=m;
  ⋮
  Where
  n, m : integer;
End Standard;
```

```
Adt Stack_Of_Item
Signature
  Use Standard;
  Use Item_Spec;
  Sorts Stack;
  Generators
  Assign : Stack, Stack → Stack;
  Pop : Stack → Stack;
  Push : Stack, Item → Stack;
  Operations
  TopValue : Stack → Item;
Body
  Axioms
  ⋮
End Stack_Of_Item;
```

Fig. 2. Ada Standard package and an Ada Package specification

4.1 Data Type Model

A CO-OPN data specification is built using an ADT. An ADT include a *Signature* and a *body*. All the components accessible from the outside are given in module's signature. The signature of a formal specification is composed of the *sort* and *function*. The signature contains informations similar to those found in an Ada specification, where a type is called a *sort* and a subprogram an *operation*. The *axioms* and the *variables* do not appear in Ada specifications because Ada specifications do not carry information regarding the behavior of the component. On the contrary the CO-OPN body part includes all the local aspect such as the properties of operations or the operational description of the object.

We can find, as in Ada, two different types of specification, a generic and non-generic ones. Contrary to non-generic specifications, generic ones takes a formal parameter which specify the properties of the acceptable parameters. figure 2 illustrates a part of the package Ada Standard, some operations and axioms are missing.

Figure 2 gives an overview of a package specification, the Ada package is presented in figure 3. It is also possible to model a procedure which return more than one parameter, by mean of a Cartesian product.

```
 type Stack_Type  is limited private;

Empty_Structure_Error:  exception;

-- CONSTRUCTORS:

 procedure Assign (Destination: in out Stack_Type;
                   Source: in Stack_Type );
-- OVERVIEW:
-- Begins by a call to DESTROY(DESTINATION) and
-- then copies SOURCE into
-- DESTINATION. Note the "in out" mode of the formal
-- parameter DESTINATION.

 procedure Pop (Stack:  in out Stack_Type);
-- ERROR:
--  If the STACK is empty, then EMPTY_STRUCTURE_ERROR is raised.

 procedure Push (Stack:  in out Stack_Type;  Item:  in Item_Type);

-- QUERIES:

 function Top_Value (Stack:  in Stack_Type)  return Item_Type;
```

Fig. 3. An Ada Package

4.2 Model of the Constructions

In this section we give an example of the way an Ada rendez-vous and protected type is handled in our framework. As we mentioned in Section 4, the package module can be included in either of the parts of the CO-OPN model (Petri nets and Abstract Data Types). When a package contains a type description, it is owned by the ADT part of the model, otherwise when a package contains a procedure, a function or a task body related to synchronisation aspects, it is represented with Petri nets. This correspondence is depicted in figure 1 while the dependences of the models are suggested in figure 4.

Overview of an Ada rendez-vous: an example We take the Ada program of figure 5 to illustrate our approach.

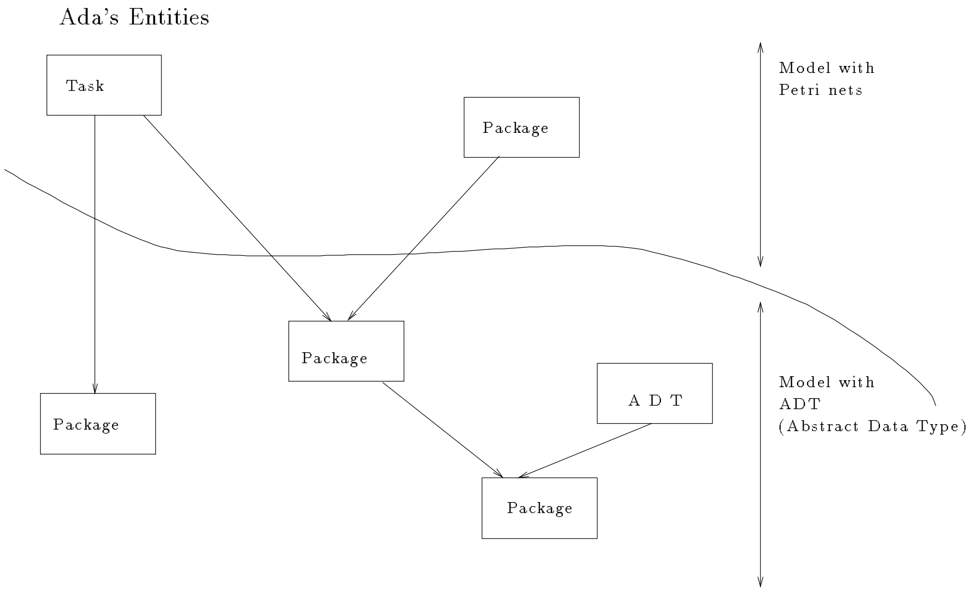

Fig. 4. Petri nets and ADT model of Ada's module

As we have seen, a calling task is suspended until the other task is ready to accept the call. We model this example with two tasks: the client and the server. A rendez-vous can be modelled as a temporary association between client and server task with a CO-OPN object. This temporary association can be in either of the following two states:

- the client task is suspended and the tasks are not yet synchronised.
- the client and server tasks are synchronised and the rendez-vous is taking place.

A rendez-vous binds two tasks during the time that they need to exchange information. We call this temporary association an *interface* in order to facilitate the translation from Ada code to the CO-OPN model.

We also assume a hierarchical (cf. figure 7) order in the representation of the model. Hierarchical means that the direction of the arrows are oriented from the interface to tasks. CO-OPN synchronisation arrows do not represent the data flow but the dependencies.

When the client task is calling there are two possibilities(cf. figure 7).

- the server task is busy. In this case, the data are stored in place *p1* and the client is waiting for the server.
- the server task is free. The internal transition *t1* is fired and the data are sent to the server task via *p2*.

```
task body Task1 is
begin
   while Cond1 loop
         Task2.Entry1;
   end loop;
end Task1;

task Task2 is;
   entry Entry1;
end Task2;

task body Task2 is
begin
   accept Entry1;
end Task2;
```

```
Adt Generic Fifo(Item);
Signature
  Use Item, bool;
  Sorts fifo;
  Generators
  empty : → fifo;
  insert _ _ : item fifo → fifo;
  Operations
  first _ : fifo → item;
  extract _ : fifo → fifo;
Body
  Axioms
  extract(empty) = empty;
  extract(insert(item),f)) = f;
  first(insert(item,f)) = item;
  Where
  item : Item;
  f : Fifo;
End Fifo;
```

Fig. 5. Code example and The Fifo ADT

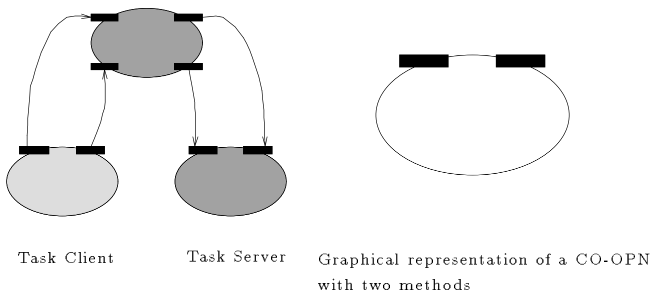

Fig. 6. CO-OPN model of Ada's module

The data structure that the place can contain are:

- *p1*: contains an instantiation of the generic Fifo waiting line for the structure that must be sent to the server task,
- *p2*: black token, (as in usual Petri nets, i.e. without any structure), and an internal data structure,
- *p3*: black token,
- *p4*: contains an instantiation of the generic Fifo waiting line for the structure that must be sent to the server task.

Places *p2* and *p3* keep the mutual exclusion of two calling task for the same server. The place *p1* and the place *p4* contain a stack of the user's data. The place *p1* contains a stack data structure which allows to store data sent by the client. The place *p4* contain a stack data structure which allows to store data sent by the entry that must be sent back to the client. The transition *m* model the task body.

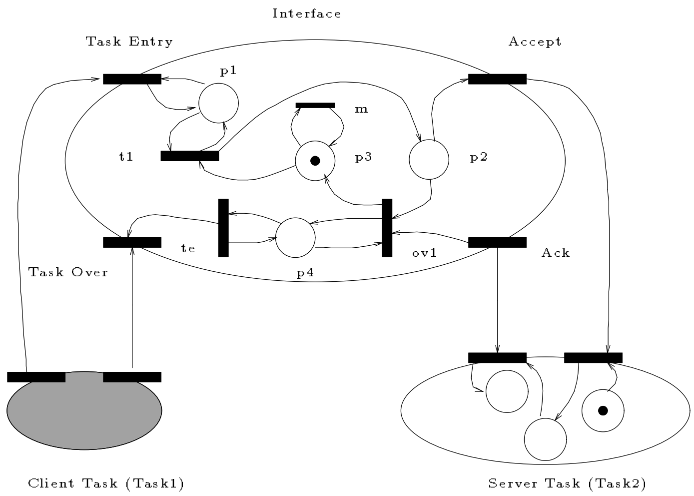

Fig. 7. Representation of the temporary association

Multiple Clients, Single Server: In figure 8 we show an example of the possibility for multiple clients to communicate with a server from the code given in figure 9. The data structure exchanged between a client and a server contains communication data (the data processed by the entry) and model data (model data contain information about client and functions used to extract, eg. its ID, useful data for the client).

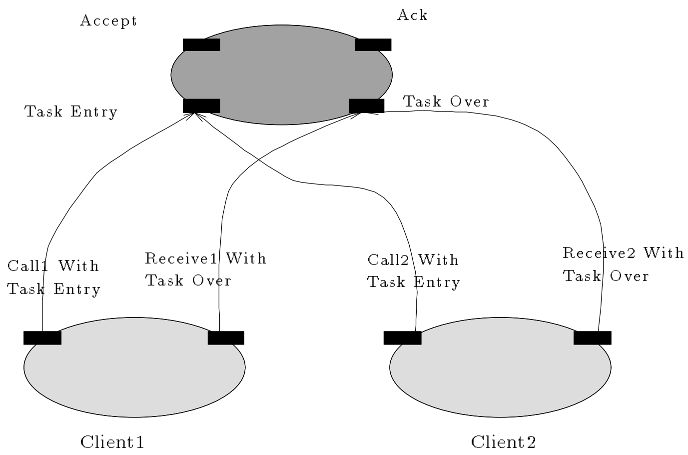

Fig. 8. Multiple Clients and one Server

```
with Text_IO;  use Text_IO;
procedure Main  is

task Task1;

task Task2  is

task Task3  is
  entry Entry1;
end Task3;

task body Task1  is
begin
if Cond   then
   ⋮
   Task3.Entry1;
end if;
⋮
end Task1;
```

```
task body Task2  is
begin
if cond   then
   ⋮
   Task3.Entry1;
end if;
⋮
end Task2;

task body Task3  is
begin
loop
   accept Entry1  do
         ⋮
   end Entry1;
end loop;
end Task3;

begin
null;
end Main;
```

Fig. 9. Multiple Clients, Single Server Ada program

Multiple Servers, Single Client: The code given figure in 11, gives the model of figure 10 and the code given figure in 13, gives the model figure 12. These examples show the possibility for a single client to send request to multiple servers or multiple entries. The server selects requests that are addressed to it by reading a server ID contained in the structure exchanged between servers and clients. The structure exchanged between tasks is hidden from the user.

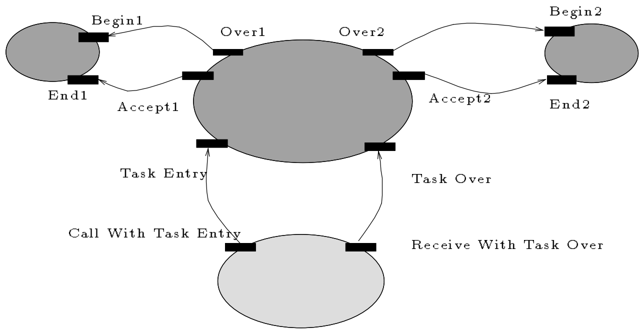

Fig. 10. Multiple entries and one client

```
with Text_IO;  use Text_IO;
procedure Main  is

task Task1;

task Task2  is
   entry Entry1;
   entry Entry2;
end Task2;

task body Task1  is
begin
if cond   then
  :
  Task2.Entry1;
else
  :
  Task2.Entry2;
end if;
end Task1;
```

```
task body Task2  is
begin
select
   accept Entry1  do
      :
   end Entry1;
or
   accept Entry2  do
      :
   end Entry2;
end select;
end Task2;

begin
null;
end Main;
```

Fig. 11. Multiple entries and one client Ada program

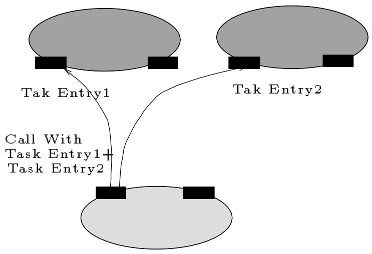

Fig. 12. Multiple servers and one client

Protected Types: Ada language specifies protected types, they formalize and expand on the use of passive tasks of Ada 83. Protected types give the possibility of defining a passive object, such as a monitor. A protected type consists of functions, procedures and entries. Which are mutually exclusive. It can also be defined entries that are different from the functions or procedures that define a barrier condition. We model this protected type in CO-OPN with a waiting line, which is defined using the same abstract data type Fifo ADT. Protected Types are modelled like tasks, except that Protected Types are active only when a rendez-vous is called.

5 Model Analysis

The analysis of the system can be roughly devised into two cases: the verification of whether the system is bounded or not (in terms of the amount of data needed

```
task Task1;

task Task2 is
   entry Entry1;
end Task2;

task Task3 is
   entry Entry1;
end Task3;

task body Task1 is
begin
if cond then
  :
  Task2.Entry1;
else
  :
  Task3.Entry1;
end if;
end Task1;
```

```
task body Task2 is
begin
accept Entry1 do
  :
end Entry1;
end Task2;

task body Task3 is
begin
accept Entry1 do
  :
end Entry1;
end Task3;
```

Fig. 13. Multiple servers and one client Ada program

for its evolution) and whether parts of the system or the entire system has the possibility to evolve. The checking of these properties, well developed for the usual Petri nets is difficult in our framework because of the use of the structured tokens and the modularity. Currently, the means we investigate cover the different techniques. In some cases, from the underlying structure of the Petri nets (projection of the data to unstructured tokens) some properties can be deduced. The simulation [BFR 93] is very useful to give a first impression about the intended behaviour of the system. Lastly, we study the use of the temporal logic as a tool to the specification and verification of the system in terms of the occurrence of its events over the course of the time [RB 93] and [BFR 93].

6 Conclusion

We have presented an overview of modelling Ada code by mean of the CO-OPN formalism. The advantage of this model is that it allows us both to model the data structures and to preserve the oriented object paradigm across CO-OPN. Much work remains in order to model all of Ada features particularly the exceptions and requeue statements that are still under study. The way Ada 95 components are represented, either by Petri nets or algebraic specifications, as proposed in figure 1 is still subject to be modified. Up to now, this correspondence is intuitive in that dynamic items are handled by Petri nets and data structures by ADT, but this correspondence can be re-evaluated by future examples. Our efforts bear also on the extensions of the verification techniques to the entire CO-OPN framework.

References

[Arnold 93] R. S. Arnold. *Software reengineering*, IEEE Computer society, Inc., 1993.

[BFR 93] D. Buchs, J. Flumet, P. Racloz. *SANDS: Structured Algebraic Net Development* System 14th Int. Conf. on Application and Theory of Petri Nets, Tool presentation abstract (Ugo Buy ed.) page. 25-29, Chicago, USA, 1993.

[BG 91] D. Buchs, N. Guelfi. *CO-OPN: a Concurrent Object Oriented Petri Net approach* 12th Int. Conf. on Application and Theory of Petri Nets, page 432-454, Aahrus,1991.

[DHP 91] C. Dimitrovici, U. Hummert, and L. Petrucci. *Semantics, composition, and properties of algebraic high-level nets.*In Advances in Petri nets, volume 524 of LNCS, page 93-117. Springer Verlag, 1991.

[DNC] M. B. Dwyer, K. A. Nies et L. A. Clarke. *A compact Petri net representation for Ada tasking programs* University of Massachusetts.

[EM 85] H. Ehrig and B. Mahr. *Fundamentals of algebraic Specification 1: Equations and Initial Semantics*, volume 6 EATC Monographs. Springer-Verlag, 1985.

[Kie 89] A. Kiehn. *Petri net systems and their closure properties.* In Grzegorz Rozenberg, editor, Advances in Petri nets, volume 424 of LNCS, pages 306-328, Berlin, 1989.

[MSS 89] T. Murata, B. Shenker et M. Shatz. *Detection of Ada static deadlocks using Petri net invariants*,IEEE Transaction on software Engineering volume 15 number 3, 1989.

[RB 93] P. Racloz, D. Buchs. *Symbolic Proof of CTL formulae over Petri Nets* 8th Int. Symp. on Computer and Information Sciences pp. 189-196, Istanbul, 1993.

[Reisig 85] W. Reisig. *Petri nets: an introduction.* EATCS Monograph on Theoretical computer Science, Vol 4, Springer Verlag 1985.

[SE 90] S. R. Schach *Software engineering*, Irwin, Aksen associates, 1990.

This article was processed using the LaTeX macro package with LLNCS style