Fundamenta Informaticae 41, No 3 (February 2000)
pp. 341-366

Fundamenta Informaticae 34 (2000) 1–26

IOS Press

Testing and Debugging Message Passing Programs in Synergy with their Specifications

Z. Tsiatsoulis*
Department of Informatics
University of Athens
Athens, Greece
zack@di.uoa.gr

J.Y. Cotronis*
Department of Informatics
University of Athens
Athens, Greece
cotronis@di.uoa.gr

Abstract. We outline Ensemble, a design and implementation methodology for composing message passing (MP) applications from program components directed by scripts. We define specification components corresponding to program components and we compose them, directed by the same scripts, obtaining formal specifications of the composed applications. We use the Petri net formalism to express component and application specifications. Petri Net composition is modelled by appropriate coupling of inscriptions on interface input and output arcs to and from a unified environment place. We elaborate on testing and debugging of MP applications based on the synergy of tools for Petri-Net simulations with tools for monitoring program executions.

Keywords: Message Passing Program Composition, Specification Composition, Reuse, Petri Nets, Error Detection

*Address for correspondence: Department of Informatics, University of Athens, Athens, Greece

1. Introduction

Development of parallel applications is, to a large extent, an empirical process. The majority of research efforts in the field of parallel processing are focused on the advances in architectures and algorithms, languages, operating system interfaces or new application areas of parallel systems [18]. Software engineering aspects (e.g. processes, methods, tools) of parallel systems have been neglected. Tools and environments support later stages of development. Designing message passing (MP) applications, in particular, is a complex task and involves designing the combined behaviour of its processes, that is, their individual execution restricted by their communication and synchronisation interactions. Message Passing Environments (MPE), such as PVM [15] and MPI [27], provide a useful abstraction of underlying architectures, simplifying architecture resource management. The software-engineering step, however, from design to implementation is also a complex task, as it does not only involve programming of sequential processes, but also latent programming for management of process topologies and architecture resources.

The Ensemble methodology [8], [9], [10], [13] was introduced for alleviating implementation complexities due to process and particularly process topology management. Each MPE supports its own process model and consequently process topology management does not depend only on the design, but also on the target MPE. Implementation of the same design on distinct MPEs requires different techniques, and the porting of applications from one MPE to another requires special effort. The differences in process management impose different structures to programs, to such a degree, that programs implementing the same design (e.g. a simple ring topology), using the same sequential language on two MPEs often look very different. Differences in communication operations cause only local syntactic variations. Furthermore, MPEs favour the management of regular process topologies (e.g. pipeline, ring, grid, torus) and some specific types of process topologies, those being closer to its process management model (e.g. PVM and tree-like process topologies). Designs of irregular process topologies are much more difficult to implement. Scaling is restricted to regular topologies by globally parameterising process position functions. As there are no implementation guidelines, programmers have their personal preferences and, quite often, use features of MPEs in unusual ways. For these reasons, designs are obscured in implementations and programs are difficult to develop, debug and modify.

Ensemble provides a common software architecture for all MP applications in any MPE and does not demand any changes to MPEs. Implementations of a design on different MPEs look the same and may be ported mechanically from one MPE to another. The design is maintained in the implementation, which is an "ensemble" of an application script, specifying a process topology, and of executable components, from which application processes are spawned. A loader program, universal within an MPE, interprets the script and establishes the topology by creating processes and by setting communication channels. Instead of functions associating processes to their position in a topology, the topology (regular or irregular) is composed directly by interconnecting communication ports of the spawned processes. The loader performs all process and resource management, as specified in the script.

Although Ensemble provides a productive framework for implementing and maintaining MP

applications, it cannot guarantee absence of design or implementation errors. The problems of debugging parallel programs (e.g. non- reproducibility of behaviour) and the difficulty of detecting the cause of errors are well known. A number of monitoring and visualisation tools have been developed, which provide the developer with information on process interactions, message queues, tracing and replay mechanisms, etc. These tools, though, do not guide the developer to find errors, but only provide requested tracing information. In addition, composition is prone to new types of errors, such as use of wrong components and unspecified or incompatible binding of communication channels. It was therefore desirable to validate the correct behaviour of components as well as of the composed applications. To this end, we have proposed a specification composition technique [11], [12], which is directly associated with the MP program composition of Ensemble.

We determine formal specifications of program components and compose the formal specification of applications. The application scripts, which direct composition of applications, also direct the composition of formal specifications. We have used the Petri net (PN) formalism and in particular Coloured Petri Nets [21] for expressing and composing specifications. PNs are well founded, have been widely used to specify parallel software systems and are supported by a number of tools. The association of program and specification composition of Ensemble is a foundation for validating the implementation of the application against its formal specifications. Available PN simulators and execution monitoring tools may be integrated to work in synergy. This approach is in accordance with Agha [1] "*...the better way to think of formal methods is as techniques that help identify bugs rather than prove programs correct...*". We also strongly believe that the acceptance of formal methods will be facilitated when they are integrated with software engineering tools.

In the next two sections, we outline program and specification composition in Ensemble. In 4 we demonstrate the reuse of component specifications. In section 5, we introduce the integrated development methodology and we elaborate on the synergy between specification analysis and program executions. In 6 we discuss related work. Finally, we present our conclusions and plans for future work.

2. The Ensemble Methodology

We briefly outline the design and implementation of applications in Ensemble on an application Get Maximum. The requirement is simple: Selector processes, each getting an integer parameter, require the maximum of these integers.

We shall implement a design, called Selector-Servers-in-Ring: Selectors are connected as client processes to associated Server processes. Each Selector sends (via port Out) its integer parameter to its Server and, eventually, receives (via port In) the required maximum. Servers receive (via their Cin ports) integer values from their client Selectors and find the local maximum. Servers are connected in a ring. They find the global maximum by sending (via Pout port) their current maximum to their next neighbour in the ring, receiving (via Pin port) the maximum of

the previous server; they compare and select the maximum of these two values. Servers repeat the send-receive-select cycle M-1 times, where M is the size of the ring. Finally, Servers send (via Cout ports) the global maximum to their client Selector processes.

2.1. The Ensemble script

A topology or process communication graph (PCG) of the design for six Selectors and three Servers is depicted in figure 1, together with the Ensemble script. The script abstractly specifies an application: the program components; the processes and their interface; the communication channels; the application parameters; the use of architecture resources. The script is structured in three main parts.

The first part, headed by PCG, specifies the Process Communication Graph (PCG) of the application, independently of any MPE or underlying architecture. The PCG part has three sections. In the Components section, we specify abstractions of the components involved in the topology (e.g. Selector) by their name, their communication interface and their design parameters, explained in the sequel.

The communication interface of components supports a general scheme for scaling applications. In general, scaling of applications requires replication of processes and establishing communication channels between their ports. Ensemble components specify communication types and a valid scaling range of communication ports of each type (shown in brackets). Each process may have a different number of ports. Communication types are depicted on PCGs on the inner circle of process nodes and ports on the outer circle. We also specify parameters, which are required for the correct behaviour of the executables and are related to the designed topology and not to the application requirement. For this reason, they are called design parameters. For example, Servers must repeat the send-receive-select cycle M-1 times, where M is the size of the ring. The Processes section specifies the nodes of the PCG, naming them by uniquely indexing component names (e.g. Selector[1],..., Selector[6] and Server[1],..., Server[3]), setting their number of ports for each communication type and values for their design parameters. The Channels section specifies point-to-point communication of compatible ports. For example, Cin and Cout ports of Server are compatible with the In and Out ports of Selector.

The second part of the script, headed by APPLICATION PARAMETERS, specifies process parameters required in the application. In the example script, each Selector is given an integer parameter. Finally, the third part, headed by PARALLEL SYSTEM, specifies information required by specific MPEs and the underlying architectures.

2.2. The reusable components

The Ensemble components compute a result or provide a service and do not involve any process management or assume any topology in which they operate. Instead, they support a range of open ports for point-to-point communication with any compatible port of any process in any application and are reusable as executable library components. The ranges of their communi-

```
APPLICATION Get_Max_Selector_Servers_in_Ring;
PCG
Components
  Selector range: In, Out [1..1]
  Server range: Pin, Pout [0..1], Cin, Cout [0..]; design: M;
Processes
  Selector[1], Selector[2], Selector[3], Selector[4],
  Selector[5], Selector[6] #ports = In:1, Out:1;
  Server[1] #ports = Cout, Cin:1, Pout, Pin:1; M=3;
  Server[2] #ports = Cout, Cin:2, Pout, Pin:1; M=3;
  Server[3] #ports = Cout, Cin:3, Pout, Pin:1; M=3;
Channels
  Selector[1].Out[1] -> Server[1].Cin[1];
  Selector[2].Out[1] -> Server[2].Cin[1];
  Selector[3].Out[1] -> Server[2].Cin[2];
  Selector[4].Out[1] -> Server[3].Cin[1];
  Selector[5].Out[1] -> Server[3].Cin[2];
  Selector[6].Out[1] -> Server[3].Cin[3];
  Server[1].Cout[1] -> Selector[1].In[1];
  Server[2].Cout[1] -> Selector[2].In[1];
  Server[2].Cout[2] -> Selector[3].In[1];
  Server[3].Cout[1] -> Selector[4].In[1];
  Server[3].Cout[2] -> Selector[5].In[1];
  Server[3].Cout[3] -> Selector[5].In[1];
  Server[1].Pout[1] -> Server[2].Pin[1];
  Server[2].Pout[1] -> Server[3].Pin[1];
  Server[3].Pout[1] -> Server[1].Pin[1];
APPLICATION PARAMETERS
  Selector[1]:"6"; Selector[2]:"999"; Selector[3]:"7";
  Selector[4]:"8"; Selector[5]:"9"; Selector[6]:"5";
PARALLEL SYSTEM  PVM3
Process Allocation
  Selector[1], Server[1] at zeus;
  Selector[2], Selector[3], Server[2] at gaia;
  Selector[4], Selector[5], Selector[6], Server[3] at chaos;
Executable Components
  Selector: path default file selector.sun4;
  Server:   path default file server.sun4;
```

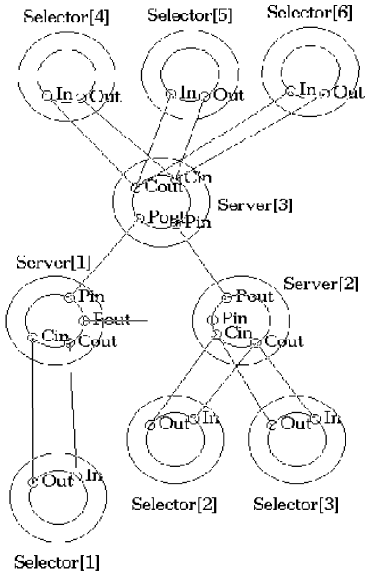

Figure 1. The application script and the PCG of Selector-Servers-in-Ring

cation types are specified in the scripts. All send and receive operations in processes refer to their interface ports, identified by a communication type and a port index within the type. The underlying architecture and tools of Ensemble handle all the details regarding the initialisation of component interface.

The structure of source code for Server and Selector components that is shown in figure 2 is typical of Ensemble reusable components. The entry point of every component is the `RealMain()` function, which accepts the same parameters, as these of function `main()` in regular C programs, as well as the Interface structure which contains all the necessary interfacing information. The programmer has the option of using abstract communication functions implemented in Ensemble's libraries or to use MPE depended functions (like `pvm_send()`, `pvm_barrier()` etc.). The component executables are reusable in any application in the given MPE. In addition, if only ab-

stract Ensemble communication functions are used, the amount of changes required to port the source of a component to a different MPE is minimal, and in most cases requires recompilation and relinking.

```
/* The common main for all reusable components */
void main(argc,argv) int argc; char **argc;
{  struct port_struct {int tid, tagid;}
   struct port_types {int PortCount; struct port_struct *port;}
   typedef struct port_types InterfaceType;
   extern int TypeCount;
   InterfaceType Interface;

   MakePorts(Interface,TypeCount);
   SetInterface(Interface,TypeCount);
   RealMain(Interface,argc,argv);
}
```

```
/* Selector Code */
void RealMain(Interface,argc,argv)
   struct port_types *Interface;
   int argc; char **argv;
{  int Typecount=2;

   GetParam(V);
   send(Out,1:V);
   receive(In,1:Max);
}
```

```
/* Server Code */
void RealMain(Interface,argc,argv)
   struct port_types *Interface;
   int argc; char **argv;
{  int Typecount=4;

   GetParam(M);
   LMax=0;
   for (j=1;j<=Interface[Cin].PortCount;j++)
   {  receive(Cin,j:V); if (V>LMax) Lmax=V;  }
   Gmax=Lmax;
   for (i=1;i<=M-1;i++)
   {  send(Pout,1:GMax); receive(Pin,1:V);
      if (V>GMax) Gmax=V;  }
   for (j=1;j<=Interface[Cin].PortCount;j++;)
      Send(Cout,j:Gmax);
}
```

Figure 2 The common skeleton code and the actual source code for the Selector and Server components

Typically the programmer develops the application script. A skeleton code for each component is generated, depicted in the top row of figure 2. The programmer adds the actual code implementing application computations (depicted in the second row of figure 2) and builds the executables. Ensemble tools handle all the details regarding library linking, makefile generation, appropriate header files inclusion etc.

2.3. Composition of applications

An Ensemble tool called the Loader interprets the script, spawns processes from component executables and establishes their interconnection scheme. There is one universal Loader program for all applications in each MPE. In this section we outlined the basic aspects of Ensemble methodology and its tools, which are relevant in the context of this paper. Ensemble tools for PVM [8], [10], Parix [9] and MPI [13] have been developed. Additional information on Ensemble and its tools may be found on `http://www.di.uoa.gr/ẽnsemble`.

3. Specifications and their Composition

To reflect the Ensemble architecture of parallel applications we have defined specification components, process specifications (instantiations of specification components) and their composition, corresponding to reusable program components, processes and the composed application, respectively.

Specification components are themselves reusable, permitting the generation of process specifications, as required by scripts. Specification components have scalable interfaces, specifying the valid range of ports for each of their communication types, i.e. always fixed (as In of Selectors), or any positive integer (as Cin of servers), or any non-negative integer (as Pin of Servers). They identify their input and output ports, the type of data that is send and received through them as well as any design or application parameters.

Process specifications are generated from specification components as mechanically as processes are generated from program components. At the time of their generation, the actual number of ports of each type in their interface is validated and the values for all parameters are provided.

Specification composition is based on modelling point-to-point communication by binding interface ports. During composition, compatibility of ports is validated, e.g. binding output to input ports passing messages of the same type.

We have used the Coloured Petri Net (CPN) formalism for expressing and composing specifications. Various classes of high-level PNs (HLPNs) are well founded, have been widely used to specify parallel software systems and are supported by a number of tools for validation, simulation, analysis and verification. Furthermore, HLPNs allow designers to create simple and easily manageable descriptions, without losing the ability of formal analysis [21], they have been extended with hierarchy constructs which resemble the notion of components in a composed system and are supported by a number of tools e.g. design/CPN [22], PEP [4], [19], LOOPN [24], SYROCO [32] and others.The definition of CPNs according to [21] is the following:

Definition 3.1. A **Coloured Petri Net** is a tuple CPN = (Σ,P,T,A,C,G,E,I) where:

(a) Σ is a finite set of non-empty types, called **colour sets**
(b) P is a finite set of **places**
(c) T is a finite set of **transitions** with $P \cap T = \emptyset$
(d) A is s finite set of **arcs** such that $A \subseteq P \times T \cup T \times P$ and $A \cap (P \cup T) = \emptyset$
(e) C is a **colour** function, $C : P \rightarrow \Sigma$
where $C(p) = C^*$ for $p \in P$ and $C^* \in \Sigma$
(f) G is a **guard function**, $G : T \rightarrow expr$
where $\forall\, t \in T : [Type(G(t)) = bool \wedge Type(Var(G(t))) \subseteq \Sigma]$
(g) E is an **arc expression** function, $E : P \times T \cup T \times P \rightarrow expr$
where $E(x_1,x_2) = \emptyset$ if $(x_1,x_2) \notin A$
and $\forall\, a \in A : [Type(E(a)) = C(p(a))_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma]$
where p(a) is the place of arc a

(h) I is an **initialisation** function, $I : P \rightarrow expr$
where I(p) is a closed expression (i.e. it contains no free variables)
and $\forall\ p \in P : [Type(I(p)) = C(p)_{MS}]$.

In the above definition, the MS subscript denotes that the expression must evaluate to multisets over the type of the associated place.

In the following three paragraphs we elaborate on composition of Petri Nets. In paragraph 3.1 we present the techniques we have used for modelling the ranges of ports of the components. In paragraph 3.2 we formally define specification components and in paragraph 3.3 we present the algorithm for specification composition by Ensemble scripts.

3.1. Place Fusion and Place Unification

The derivation of specification components is done manually, i.e. the specification components are designed in a way that represents the behaviour of the associated program component. Heiner has studied in [20] the association of the metanotions of a "reduced grammar for code statements" to PN constructs. We will use these associations to derive our specification components. As an example, figure 3 depicts the CPN Server component.

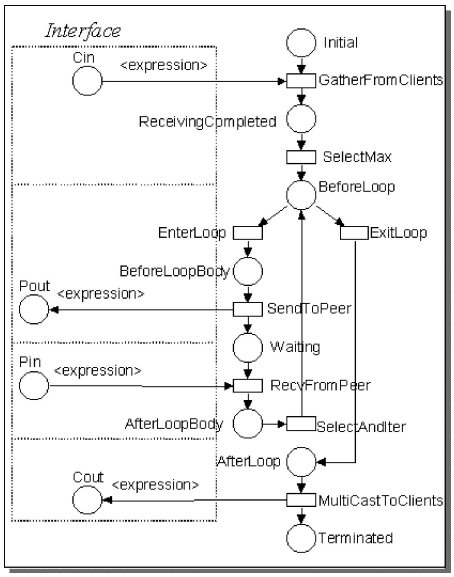

Figure 3. The CPN for the Server component

The doted rectangle surrounds the interface of the component. The remaining elements of the net are the static net structure, which corresponds to the internal actions of the component. In CPNs, communication operations are modelled by transitions connected to interface places, which model interface ports. Collective communication operations (e.g. gather and multicast) are modelled by a single transition for conciseness. For example in figure 3, transitions Gather-

fromClients and MultiCastToClients, model receive and, respectively, send operations from and to all client processes.

The interface places and the arcs connecting collective communication transitions denote a communication type, having a range of ports. In generating process specifications from specification components, the actual number of ports is specified. In our earlier work [11], [12], we modelled the ranges of ports by replication of interface places and their connecting arc. Composition was based on place fusion. This approach is depicted on the left- hand side of figure 4 (in which the static net structure is even more simplified) and models point-to-point channels explicitly. This approach leads to an explosion of interface places.

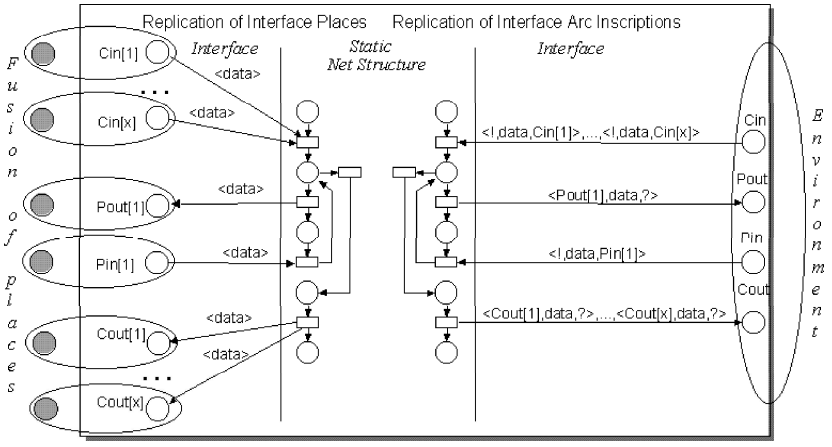

Figure 4. The two approaches to modelling the interface

In this paper, we model ranges of ports within a communication type by maintaining a single interface place and replicating inscriptions on the connecting arc, as depicted on the right hand side of figure 4. Ports within a type are identified by the tokens in the inscriptions of the arcs connecting interface places. The tokens have the structure of <SendPort,data,ReceivePort>. The data field represents messages. To realise open interfaces of components, SendPort in the tokens of the input arc inscriptions is left unspecified, denoted by **!**, as for example in the input arc of Cin in figure 4. Similarly, ReceivePort in the tokens of the output arc inscriptions is also left unspecified, denoted by **?**, as for example in the output arc of Cout in figure 4. Composition is based on unification of all interface places into a single environment place, maintaining arcs and their inscriptions. Point-to-point channels are modelled implicitly by coupling tokens of replicated inscriptions on the connecting arcs. By coupling tokens T1 and T2 of input and output arcs respectively, we mean the substitution of **!** in T1 by the SendPort of T2 and the substitution of **?** in T2 by the ReceivePort of T1. The common names of SendPort and ReceivePort in T1 and T2 uniquely determine their point-to-point channel.

The use of a single interface place for the composition of PetriNets was proposed in [6]. The unified environment place resembles modelling the tuple space of Linda [17]. However, the tuple space in Linda does not define channels, whereas in the aforementioned approach point

to point channels are implicitly defined. If we omit the SendPort and ReceivePort fields of the tokens, this approach will be very close to Linda's paradigm.

3.2. Specification Components

A component specification can be modelled by standard CPNs when its interface is fixed, as for example in the Selector component (it has one port of types In and Out), but parametric interfaces cannot be directly modelled using CPNs. We extend CPNs by template CPNs, which contain additional information for open scalable interfaces. Template CPNs are very close to the notion of pages in [21], [22] as they are also "*flat*" non- hierarchical structures, although Template CPNs are parametric. A template CPN has a unique name, from which process specifications, called composable CPNs, can be instantiated, as a page having several page instances. Instantiation of composable CPNs from templates involves structural modification of the net, whilst page instances are exact copies of the original page. The formal definition of the template CPN is the following:

Definition 3.2. A **Template CPN** is a tuple TPN = (NAME,SNS,IS) where:

(a) NAME is the name of the template.
(b) SNS (**Static Net Structure**) is a coloured Petri Net (Σ,P,T,A,C,G,E,I)
(c) IS is an **Interface Specification** (IP,IA,IE)
  (i) IP is a set of **interface places** such that
    IP $\cap$ (P $\cup$ T $\cup$ A) = $\emptyset$, P,T and A $\in$ SNS
    The elements of IP have the same names as the *communication types*
    of the component, as used in the Components section of the script.
    Furthermore, the elements of IP are of two distinct sorts {input,output}:
    $\forall$ p $\in$ IP, *if* $^\bullet$p = $\emptyset$ *then* sort(p) = input *else if* p$^\bullet$ = $\emptyset$ *then* sort(p) = output.
  (ii) IA is a set of **interface arcs** such that
    IA $\subseteq$ (IP $\times$ T) $\cup$ (T $\times$ IP)
  (iii) IE is an **interface arc expression** function,
    IE : IA $\rightarrow$ interf-expr, such that $\forall$ a $\in$ IA,
      if sort(p(a)) = output *then* IE(a) = <p(a),expr,?>
      else if sort(p(a)) = input *then* IE(a) = <!,expr,p(a)>
    and [Type(IE(a)) = C(p(a))$_{MS}$ $\wedge$ Type(Var(IE(a))) $\subseteq$ $\Sigma\wedge$
    Var(IE(a)) $\in$ Var(SNS) $\cup$ {portindex}]
      where Type(portindex) = integer and p(a) is the place of arc a.

In the above definition, we use an integer variable named portindex, which is used to index port names in tokens of inscriptions in order to indicate scaling of interfaces. The MS subscript, again, denotes that the expression must evaluate to multisets over the type of the associated place.

We have defined a syntactic form for expressing template CPNs, which is suitable for mechanical composition. We use a variation of EBNF notation to present this syntactic form of the template CPNs:

Template *TemplateName* (**#ports** {*ctypename:int,*}+;[**design** {*dpname:colour,*}+;]
[**application** {*apname:colour,*}+;]);
Declarations
val {*Sname:=actual-value;*}+
type {*colour=datatype;*}+
var {{*variable,*}+:*colour;*}+
Interface
{**Ctype** *ctypename;*
range: *from..to;*
arc: {{**output** *TransitionName* **inscription** <*port,expression,*?>;}|
{**input** *TransitionName* **inscription** <!,*expression,port*>;}}
End Ctype }+
Net Structure
Places {*PlaceName(colour),*}+;
Transitions {*TransitionName([guard,][action]),*}+;
Arcs
{*PlaceName*>*TransitionName* **inscription** *expression;*}+
{*TransitionName*>*PlaceName* **inscription** *expression;*}+
Marking {*PlaceName(Sname);*}+
End Template

In figure 5 we give the description of template Selector of figure 1 together with its graphical equivalent. In the sequel, we will use the graphical representation in figures for the sake of conciseness. The heading of the template CPN is an abstract description of the script components. The name is given after the keyword **Template**. Similar to the Ensemble program components, template CPNs have also three kinds of parameters: port interface parameters (the number of ports of each communication type), design parameters (related to the topology) and application parameters. These parameters appear in parentheses at the heading of the template, immediately after its name. The keyword **#ports** is followed by a formal parameter list of the number of interface ports for each communication type. Keywords **design** and **application** precede the design and application parameters respectively. Design and application parameters also symbolically index the initial place of the net structure. The symbolic initial markings and the #ports parameters will be replaced by actual values when process specifications are generated. The port interface parameters need to be validated for being within accepted range.

Following the heading of the template there is a **Declarations** part, where all constants (**val**), data types (**type**) and variables (**var**) of the net are specified. The data types are the basic data types i.e. integer, real, character and boolean, as well as composite data types, i.e. arrays, sets and structures.

The next part is the **Interface** part, where we specify communication types. For every communication type we define a **Ctype** construct. The first element in the construct is the

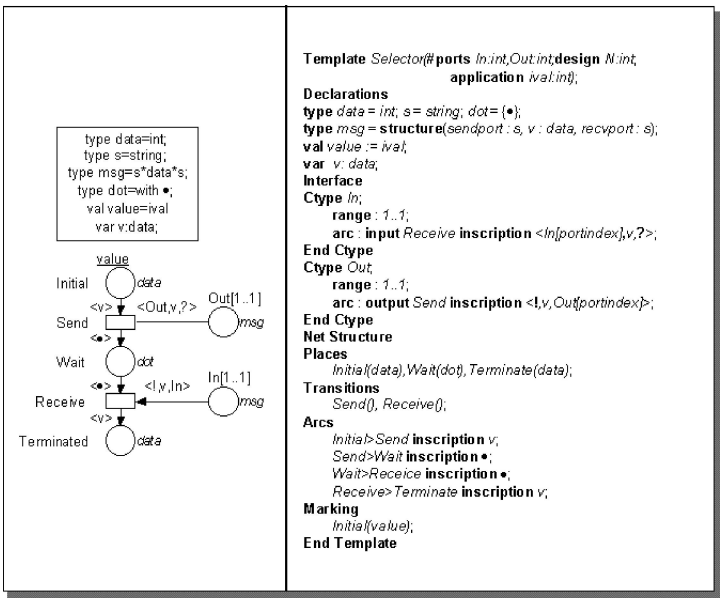

Figure 5. The template CPN of selector component in textual and graphical form

range declaration. It specifies the valid range of ports for this communication type. We have used a simple notation *from..to*, where *from* could be any non-negative number and *fromleqto*. When a communication type has fixed number of ports, say N, the expression becomes *N..N*. As an example, refer to the range declaration part of Selector in figure 1, where In and Out are defined to have a range 1..1. When the value of *from* is zero, the component may have no ports of this type. If the value for *to* is unspecified then there is no upper bound on the number of ports. In figure 1, Server is defined to have a 0.. range for its Cin and Cout port types and a 0..1 range for its Pin and Pout port types. Next, an **arc** declaration is included, which defines the arc that connects the interface place of Ctype with the net structure together with its inscription. There are two alternatives for the arc declaration, depending on being an input or output interface place. Keywords **input** and **output** denote the direction of the arc and the transition of the static net structure to which it is connected. The **inscription** part declares the tokens that are exchanged, as described in section 3.1. The variables in the *expression* must be declared in the **Declarations** part of the template. As we mentioned before, we assume a predefined local variable named **portindex** of type **int**, which indexes port names in tokens of inscriptions to indicate scaling of interfaces. Variable portindex may also be used to index array variables.

The **Net Structure** part defines static, non-interface elements. In the **Places** section, the places of the net are defined along with their associated colour. In the **Transitions** section, we declare the transitions and their guards. The *guard* is a condition that must hold for the transition to fire. The **Arcs** section declares the directed arcs that connect places to transitions and transitions to places. The annotations of these arcs are also defined in the **inscription**

declaration. The final part of the template is the **Marking** part, where the initial marking of the net is specified. The marking of a place is a multiset of tokens of the place colour.

3.3. Composition of Specifications

We now present the composition of specifications, directed by Ensemble scripts. The algorithm for the composition has four steps.
Algorithm

1. Retrieve the Template CPNs of the components in the script.
2. Create the Composable CPNs:

   (a) for each process in the script, make a copy of the corresponding template and name it by indexing NAME of the TPN with the instance number of the process,

   (b) for each TPN instance, NAME_i, check the validity of port interface parameters. If the interface is valid create the process specifications. For each *communication type* q, create the **expression replication list** ERL(a,n), where n is the actual number of interface ports and a ∈ IA of NAME_i with p(a)=q.
   The expression replication list is:

   $\text{ERL(a,n)} = \text{IE(a)}_1,\text{IE(a)}_2,...,\text{IE(a)}_n$ if n> 0
   $\text{ERL(a,n)} = \varepsilon$ if n= 0,

   where IE(a)_i = <q[i], expr_i,?> or <!, expr_i, q[i]> such that any occurrence of portindex in the interface expression is substituted by i.

   (c) Read values for design and application parameters from the script.

3. Create the composed CPN i.e.
   Merge the Composable CPNs according to the *Channels* section of the script, by **coupling** the elements of the expression replication list associated with channels i.e. <q[i],expr_i,?> or <!,expr_j,q[j]> and **unifying** all interface places of all Composable CPNs into a single *environment place.*

   (a) Coupling is performed by substituting ! and ? in expressions with a place that constitutes a channel along with the specified port q[i] or q[j].

   (b) Unify all interface places into a single environment place, by substituting the place of all interface arcs with the environment place:

   $\forall\ \text{NAME}_i, \forall\ \text{a} \in \text{IA}, \text{p(a)} = \textit{environment}$

   (c) $\text{C}(\textit{environment}) = \cup\ \text{C}(\text{p}_i)$, $\text{p}_i \in \text{IP}$, $(\forall\ \text{NAME}_i)$.

4. Validate composition, i.e.
   Check if there exists an interface arc expression that contains ? or ! instead of actual port names, that is check if all ports specified in the script are actually connected.

The composed CPN, corresponding to the script of figure 1 is illustrated in figure 6. Only Server[1] and Selector[1] are depicted analytically as composable CPNs. For brevity all other components are depicted in a *"box"* representation, where only the interface arcs and their inscriptions are visible.

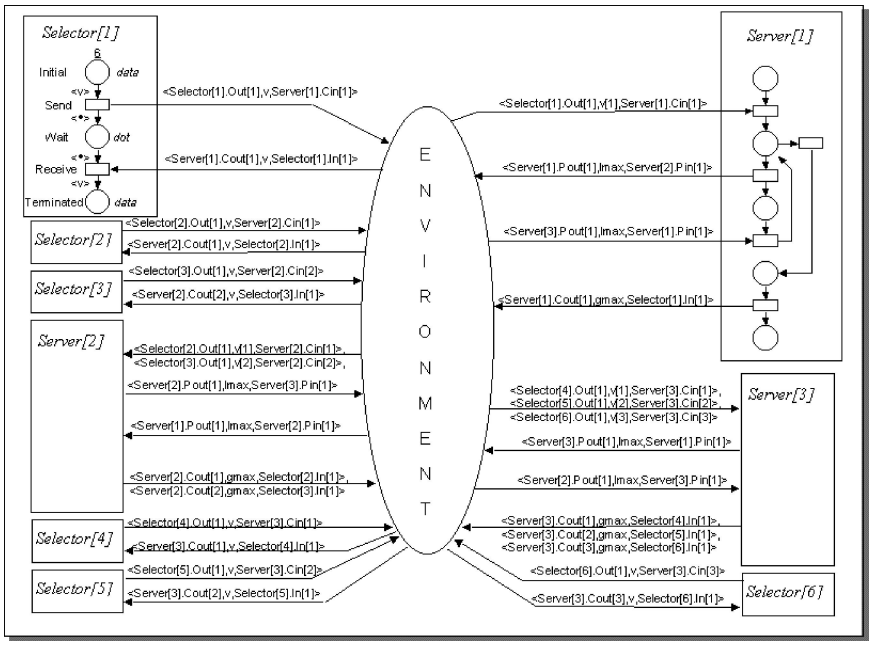

Figure 6. The composed CPN of Get Maximum Selector Servers in ring

4. Reuse of Specification Components

In this section, we demonstrate the reusability of specification components. We design a variation of Get Maximum, which reuses the template CPNs for Selector and Server. As in the Selector-Servers-in-Ring we use three Servers and six Selectors. In this variation, Server processes are organised in a tree, with Server[3] being the root, which has no Pout and Pin ports. Server[1] and Server[2] have one Pout and one Pin ports which are connected to the Cin and Cout ports, respectively, of their parent Server[3]. Server[3] has also Selector[6] connected as a client process. Server[1] has Selector[1] and Selector[2] as its client processes and Server[2] has Selector[3], Selector[4] and Selector[5] as its client processes. The process structure is a tree of height 2: the Selector processes 1,2,3,4,5 are at level one; Server[1], Server[2] and Selector[6] are at level two; and Server[3] is the root. The application script and the PCG of the application are depicted in figure 7.

Server[1] and Server[2] receive values from their clients, find their local maximum and send it to their Pout port, which is connected to a Cin port of Server[3]. Server[3] finds the global maximum, and as it is not connected in a ring, it directly sends the global maximum to its

```
APPLICATION Get_Max_Selectors_and_Servers_in_Tree
PCG
Components
  Selector range: In, Out [1..1], design: N;
  Server range: Pin, Pout [0..1], Cin, Cout [0..], design: M;
Processes
  Selector[1], Selector[2], Selector[3], Selector[4],
  Selector[5], Selector[6] #ports = In, Out:1;
  Server[1] #ports = Cout, Cin:2, Pout, Pin:1, M=2;
  Server[2] #ports = Cout, Cin:3, Pout, Pin:1, M=2;
  Server[3] #ports = Cout, Cin:3, Pout, Pin:0, M=1;
Channels
  Selector[1].Out[1] -> Server[1].Cin[1];
  Selector[2].Out[1] -> Server[1].Cin[2];
  Selector[3].Out[1] -> Server[2].Cin[1];
  Selector[4].Out[1] -> Server[2].Cin[2];
  Selector[5].Out[1] -> Server[2].Cin[3];
  Selector[6].Out[1] -> Server[3].Cin[1];
  Server[1].Cout[1] -> Selector[1].In[1];
  Server[1].Cout[2] -> Selector[2].In[1];
  Server[2].Cout[1] -> Selector[3].In[1];
  Server[2].Cout[3] -> Selector[5].In[1];
  Server[3].Cout[1] -> Selector[6].In[1];
  Server[3].Cout[2] -> Server[1].Pin[1];
  Server[3].Cout[3] -> Server[2].Pin[1]
  Server[1].Pout[1] -> Server[3].Cin[2];
  Server[2].Pout[1] -> Server[3].Cin[3];
APPLICATION PARAMETERS
  Selector[1]:"6"; Selector[2]:"999"; Selector[3]:"7";
  Selector[4]:"8"; Selector[5]:"9"; Selector[6]:"5";
```

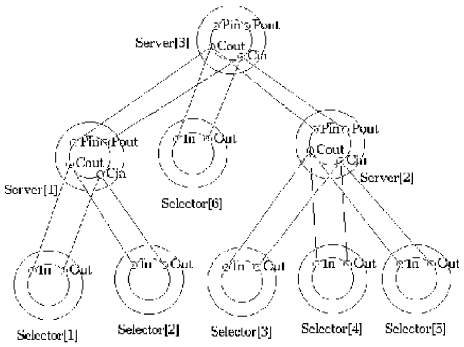

Figure 7. The application script and PCG for Get Maximum Selectors and Servers in tree

clients. Selector[6] gets the global maximum, as well as Server[1] and Server[2]. Server[1] and Server[2] "select" the global maximum and send it to their client processes.

The composed CPN obtained from the script of application get maximum selectors and servers in tree is illustrated in figure 8. Components are depicted in the "box" representation.

As can be seen in figures 6,8 the structure of the composed CPNs of the two variations of Get Maximum are almost identical; they differ in their interface arc inscriptions.

We demonstrated that although Server and Selector template specifications were originally designed for the needs to Selector-Servers-in-Ring, they are reused in a different design of Get Maximum application.

5. Implementation and Testing Methodology

Ensemble supports the design and implementation phases of the software development life cycle. Gorton and Jelly present in [18] a number of challenges that must be addressed by a distributed systems designer. The architecture of Ensemble deals with a number of these challenges: scalability requirements, inter-component communications, design validation, choosing synchronisation and MP mechanisms, portability constraints. In this section, we propose the in-

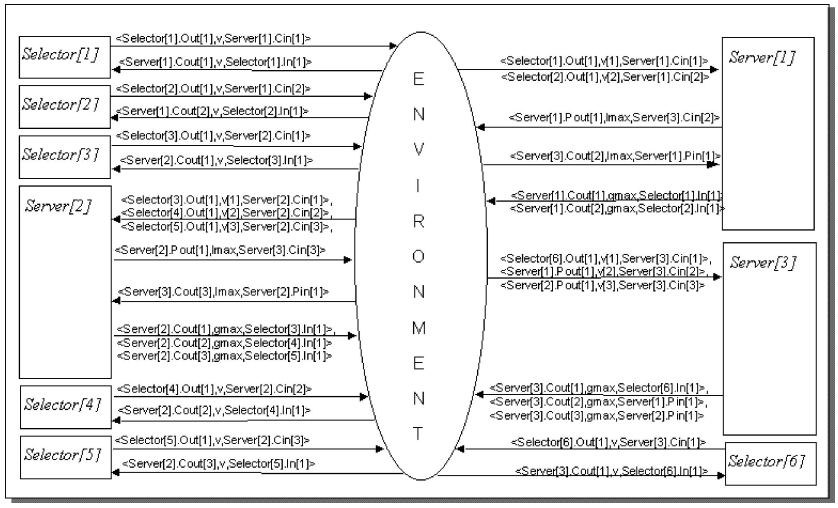

Figure 8. The composed CPN of application get maximum selectors and servers in tree

tegration of formal methods and execution analysis which will cover challenges that apply to the testing and debugging phase of software development life-cycle. An overview of the framework of the proposed methodology is depicted in figure 9.

The row headed by specification represents the formal design phase of Ensemble, as described in section 3. The components cell contains the reusable specification components that participate in the application. The script drives the composition performed by the specification compositor. The Composed specifications may then be validated (analysed and/or simulated) by PN tools.

The row headed by program represents the implementation phase of Ensemble. In fact, the implementation of an Ensemble application requires the implementation (or reuse) of individual components, and of the PARALLEL SYSTEM part of the script, which contains information about the execution environment. The PCG and APPLICATION PARAMETERS parts of script are the same as in the composition of the specifications. This can be seen in the script column, where the rectangle representing the script is common except from the part distinguished by a dotted line, which represents the extra information for the execution environment, described in section 2.1. The application is actually composed by the Loader (section 2.3). The vertical dimension refers to the software-engineering step from design to implementation. In the components column, the grey ellipse depicts the testing and debugging of individual components and in synergy of tools column testing and debugging of composed applications. The testing and debugging involves the use of formal analysis and visualisation or monitoring tools in synergy under the general framework of the methodology, and will be elaborated in the subsequent sections.

In Ensemble, errors may fall in three categories: (i) *design related errors*, which can be detected by specification analysis, (ii) *implementation related errors*, which can be detected by testing individual program components and (iii) *faults related to the execution environment*,

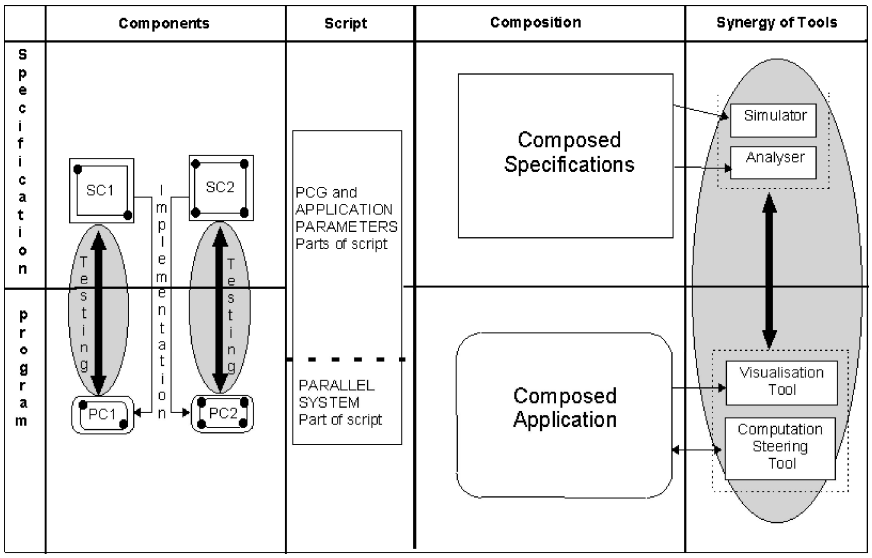

Figure 9. Overview of the development methodology

which can be detected during execution of composed applications.

5.1. Implementation, Testing and Debugging of Program Components

Implementation of program components is guided by the associated specification components. The specification components may reflect various levels of abstraction. Their interface part must be represented in detail, even at the highest abstraction level. If we are interested only in the behaviour of the application that is related to its parallel nature, there is no need to use a detailed representation for the internal actions. The highest abstraction level that can be used, is the explicit modelling of the communication operations. All other operations may be represented *"abstractly"*, since their participation in the behaviour of the application is restricted in the internal behaviour of the components. Heiner in [20] associates all usual program constructs, such as for loops, if-then-else statements etc. to specific PN constructs. If required, we can model program constructs in detail by using these associations.

5.1.1. Implementation of Program Components

The implementation of Program Components is based on the skeleton described in section 2.2 and involves providing their interface and their sequential code (RealMain actions). The interface consists of the communication types and their range, as specified in the Interface part of the specification components. Implementation of RealMain actions is guided by the corresponding specification component's internal structure.

5.1.2. Testing Individual Components

For a message passing component to be tested it must have some interconnections with other components. We provide a *testing environment*, which will simulate the actual environment by providing interconnections for the component. The testing environment consists of *"stub"* environment processes that involve message passing activities of components. Environment processes may be either input or output, respectively connected to input and output ports of the component.

The testing environment itself is based on Ensemble and consists of *"stub" environment components*, from which environment processes are generated, and of *test scripts* specifying the interconnection of the program component under test to environment processes. An environment component defines a simple port interface compatible with the ports of the component to which it will be connected. An input process gets input values interactively and sends them to the process under test. Similarly, an output process receives values from the process under test and displays them. Environment components are simple to implement and in certain cases are produced automatically. Environment components may be reused. The test scripts should specify typical values in the range of the ports of the program component, in order to test its behaviour in different positions of a topology (e.g. for a grid topology there are 9 different positions for a component). The loader will compose the testing application and the results of the execution are validated.

The testing environment along with the program component is in fact a downsized parallel program. The program components may be *"instrumented"* and monitoring, visualisation, and computation steering tools, e.g. [16], [26] are used to analyse parallel aspects of program behaviour. Special breakpoints can be inserted before and after each communication operation when producing the executable code of the program component. A computation steering tool may use these breakpoints to steer computations into states of particular interest.

We introduce a more advanced testing, based on the synergistic error detection of specification and program components, which is depicted in figure 10 together with the environment components and the test script. Also depicted are analysis and simulation tools of specifications, as well as monitoring and debugging tools of programs. Specification and execution tools cooperate. On the one hand, tracing information of the composed application drives the simulation of the specification component. The simulator detects invalid events and gives the earliest possible warning. On the other hand, the specification simulator is used as an advanced computation steering tool to direct the execution of the program. The specification simulator may be used to derive valid (i.e. reachable) states or other properties of the system, driving the computation steering tool to reach corresponding program states. Any errors should be detected immediately, since the monitoring tool will report the failure of the execution to reach that state.

For the synergistic testing of specifications and program components the corresponding *specification testing environment* must be provided, consisting of *specification environment components*. The test scripts are the same. The specification environment components consist of an initial state (place), a single send or receive transition that will consume or produce tokens, a

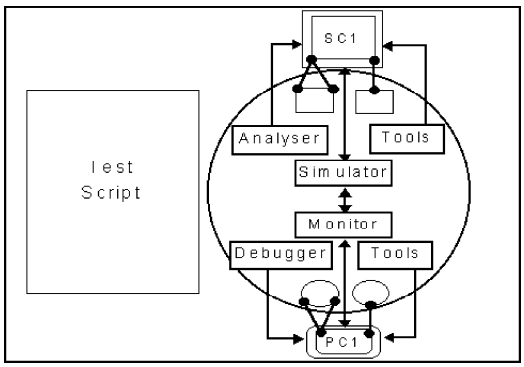

Figure 10. Synergistic testing of program and specification components

final state and an interface place. The type of the environment component, i.e. input or output, the colour of the tokens and the number of ports corresponding to the interface place must be specified.

Eisenstadt in [14] presents the results of a study on the reasons of the difficulty to trap errors, as well as the techniques used to locate the errors. The main reasons errors are difficult to locate fall in four categories. (i) *Cause/effect chasm* (ii) *Tools inapplicable or hampered* (iii) *WYSIPIG (what you see is probably illusory governor)* (iv) *Faulty assumption/model or misdirected blame.* Eisenstadt reports also four major error detection techniques. (i) *Gather data* (ii) *Inspeculation (inspection- hand simulation-speculation)* (iii) *Expert recognised Clichés* (iv) *Controlled experiments.*

The proposed testing and debugging of program components conforms and improves the techniques to locate errors reported by Eisenstadt. The simple testing of a component by using environment components corresponds to controlled experiments and inspeculation. Testing the instrumented component by monitoring, computation steering and visualisation tools correspond to data gathering. Finally, the synergistic testing of specification and program components corresponds to expert recognised clichés, but analysis of the specifications and programs in synergy results into objective conclusions about the errors, instead of subjective conclusions made by a person.

The difficulties to trap errors are also alleviated. The cause/effect chasm is reduced since the analysis of the specification components will detect the error when it occurs, even if its effect will appear much later in the execution. Thus, the notion of immediacy in debugging as presented by Ungar et al. in [34] is satisfied. The specification analysis and the monitoring tools do not interfere in the execution and do not alter program states. Furthermore, the WYSIPIG and faulty assumption cases are covered from the specifications as they are modelling the actual behaviour of the system.

5.2. Testing Composed Ensemble Applications

The composed application is tested to detect errors related to execution environment that could not be detected in the previous stages of testing. Errors related to design and implementation of program components that have not been detected, because previous tests failed to generate their enabling conditions, may also be detected. Monitoring and visualisation tools are used, which provide the developer with information of process interactions, message queues, tracing and replay mechanisms, etc. These tools on their own do not guide the developer to find errors, but merely provide requested tracing information. The direct association of program and specification composition of Ensemble is a foundation for validating the implementation of the application against its formal specifications.

Available simulation tools and program execution monitoring tools may work in synergy, using the same co-operation principles we used in testing individual components. Tracing information of the composed application may be passed to drive the specification simulator of the composed specifications. The simulator detects invalid events and gives the earliest possible warning. Thus, the behaviour of the application is not only monitored, but also actually validated as it is running. The developer is not obliged to inspect detailed views of visualisation of executions, since the simulator validates the execution against the specifications. The validation may be performed in the background, as the application is running, or by analysing a trace file suspected of erroneous behaviour.

The specification simulator may be also used as an advanced computation steering tool to direct the execution of the program. Specification simulation may steer the program directly, analysing programs by a "bisimulation" principle. The breakpoints are already set in the program components. In addition, specification properties (e.g. reachable states) may be used to validate associated program properties.

Figure 11, depicts the proposed testing strategy of composed applications. If during execution an error occurs, we first check if the error can be detected in the specifications. In this case, we modify the initial design. If the error cannot be detected in the specifications, we first check if the error can be detected in component implementations. In this case the components are modified accordingly. If it cannot be detected, then we attribute it to execution environment factors. In order to correct an execution environment error, we may have to modify the specifications or the program component implementation, or the PARALLEL SYSTEM part of the script.

Let us exemplify the three cases of environment errors. Consider asynchronously sending a long message over a buffer smaller than the message. In some MPI implementations the system automatically switches to synchronous mode, in order to send the complete long message in smaller packets. This may result into a deadlock, that could not be detected in any of the previous stages. In this stage, the specification analysis will show that the design is valid, the implementation monitoring will depict the specific problem, and the user may immediately detect that the error is due to this automatic conversion of the communication mode. Thus, the programmer should modify the design of the application, by using synchronous communication.

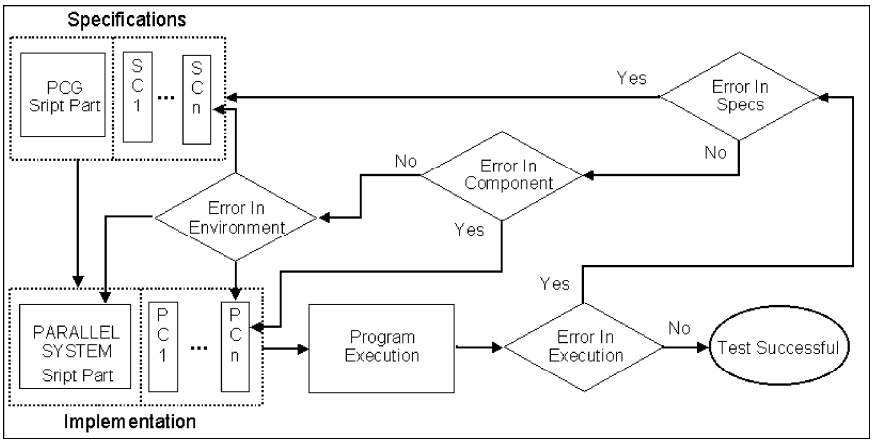

Figure 11. Testing Strategy under Ensemble

An error that requires modification of the program component is an overflow of some data type, e.g. an assumed 64-bit integer implementation instead of an actual 32-bit one. Correction involves the variable definition to be modified from integer to long. Finally, an example of an error that must be corrected in the PARALLEL SYSTEM part of he script, is the case where a host specified is inoperable. The designer must modify the script, in order to use some other host.

6. Relative Work

A number of formalisms have been proposed for the composition of PN. In our approach of particular interest is modelling of communication since we compose PN components through binding of communication channels. Most of the proposed formalisms model asynchronous communication via fusion of places, and hence model the explicit point to point communication scheme.

In [23], a partial order semantics for Petri net components has been proposed and components and composition of systems are formally defined. A Petri net component is a Petri net equipped with distinguished interface, input and output places. A component communicates with its environment through the interface places. The fusion of components at input and output places corresponds to asynchronous Message Passing. The basic difference from our approach is that the communication interface in [23] is static, in contrast to the open scalable interface of our approach.

In [31] Sibertin-Blanc proposes communicative and cooperative nets. Components, defined as a variation of PNs, have an interaction layer in order to communicate and cooperate. Some places of the communicative net are declared as accept-place where any net can put tokens. Transitions may have an action: a function call, creation of a new object or sending a token to an accept-place of another object. Cooperative nets are a variation of Communicative nets.

The major difference is that the interaction mechanism is changed from the message passing paradigm to the client/server paradigm. Thus, accept-places are replaced by a couple of places for each service, an accept-place for receiving the parameters of requests and a return-place for the produced results; services are requested by request-transitions, which act as send-transitions; retrieve transitions are needed, whose occurrences take a token from a return-place. Hence, the interactions are always asynchronous. Furthermore, in [30] in order to overcome the problem that tokens of different data types may occur in the accept places, Sibertin-Blanc uses the definition of the specialisation of a data type: the type t of the tokens in the accept place specialises the type t' of the accept place, that is Dom(t) $\subset$ Dom(t'). In a sense, this is similar to the unification of places we have defined in section 3.

Although our template CPNs resemble objects, we did not use object oriented notations [6], [25], [32]. Template CPNs are "flat" like the pages in HCPNs [21], [22] and since our composed CPN is build automatically, we do not need any further abstraction or other organisational structures. Furthermore, the analysis tools support flat representations of elementary PNs or restricted versions of CPNs.

The algebra of M-nets was introduced in [2] as an abstract and flexible metalanguage for the definition of compositional semantics of concurrent programming languages. M-nets have been applied [3] to the B(PN)^2 programming language [5]. B(PN)^2 is a language for the specification of concurrent algorithms, parallel or distributed systems, which incorporates within a simple syntax many of the constructs used in concurrent programming languages.

The most distinguishing feature of M-nets is given by the rich set of composition operators they provide. These allow the compositional construction of complex nets from simple ones, thereby satisfying various algebraic properties. M-nets are a mixture of coloured net features and low level labelled net ones. The main difference between M-nets and coloured nets [21] is that M-nets carry additional information in their place and transition inscriptions to support composition operations. Annotation of places (set of allowed tokens), arcs (multiset of structured annotations) and transitions (occurrence conditions) support the unfolding of an M-net into an elementary net. Communication capabilities are denoted by additional labelling of transitions (communication interface), whilst additional labelling of places denotes their interface capabilities (status). Furthermore a composition technique for M-nets via a single interface place, called refinement, has been proposed in [6].

Based on these semantics a programming environment for B(PN)^2 programs including verification of program properties by model checking has been developed within the PEP project [4], [19]. PEP is a Programming Environment based on Petri Nets, which supports different types of objects in order to model parallel systems, such as low level nets (Petri Boxes), M-nets, Parallel Finite Automata (PFA), Petri Box Calculus terms, and B(PN)^2 programs. Furthermore, PEP allows the user to define a set of temporal logic formulas, in order to check a custom designed system property. The PEP system consists of a number of editors for the different object types, a number of compilers between the different object types (e.g. B(PN)^2 $\Rightarrow$ M-net, M-net $\Rightarrow$ Petri Box etc.), simulators for B(PN)^2 programs, M-nets, Petri Boxes etc., a verification

component with some standard algorithms (i.e. checking the free choice/T-system properties, liveness, deadlock freeness, reachability and reversibility) and some model checking algorithms that can determine whether a PN satisfies a custom property given in terms of a temporal logic formula and, finally, a reference component which controls the interplay between the different object types (e.g. transforms program formulas into net formulas or triggers program simulation by net simulation).

PEP provides a powerful environment for designing and verifying parallel algorithms and systems. Our approach is more Software Engineering oriented. Our primary goal is not the formal verification of algorithms, but error detection (i.e. testing and debugging supported by formal methods) of actual message passing programs. Furthermore, our methodology applies to real programs running under several popular MPEs (e.g. PVM, MPI, Parix [29]) on actual parallel machines. The compositional approach can be considered as a structured way to derive and validate complete parallel program, by testing and composing the semantics of sequential components. Thus we derive the semantics only of sequential components, which have open and scalable interfaces. In [33] we have presented a mechanical way to derive these specification components from the corresponding program components, when using a specific tool (i.e. GRADE). Our aim is to correlate the execution of programs with simulation/analysis of specifications.

In the context of reusable software components, Corba, JavaBeans [28] etc. have been developed specifically for distributed environments. These models are based on object oriented features, their interface is fixed and communication is achieved through Remote Method Invocation. The reusable components of Ensemble are designed for the Message Passing model of parallel programming, their interface is adapting to the specifications of the applications, and their interface ports send and receive plain data over channels in order to communicate. The relevance of Corba etc. in the context of our methodology will be investigated in the future.

7. Conclusions - Future Work

The integration of formal methods with software engineering methods improves testing and debugging of message passing applications. The proposed methodology takes advantage of the direct association of Ensemble specifications and programs. They are both composed from reusable components and their composition is directed by the PCG part of the script, which specifies the application topology.

The compositional approach facilitates the construction of application specifications, as we need only to construct specifications of sequential components. Implementation is similarly simplified since it requires only the implementation of the sequential actions and the interface of the components, together with a part of the script, that provides information about the execution environment.

We have presented a composition technique based on pair-wise “coupling” of the interface arc inscriptions and the “unification” of all interface places into a single common environment place. Thus, channels are implicitly determined by arc inscriptions. In our previous work composition

was based on replication of interface places and channels were explicitly determined by the pairwise fusion of input and output places. The two approaches have advantages and disadvantages. In the place fusion approach, it is relatively easy to incorporate ordering of messages over the same channel: either require interface places of capacity 1 or model fused places by a FIFO queue. In the place unification approach ordering of messages is more difficult since it requires the environment place to be modelled by multiple queues, one for each implicitly determined channel. The place fusion approach leads to an explosion of replicated places, whereas the place unification approach only replicates arc inscriptions to and from the environment place. In fact, the number of tokens is the same in the two approaches, but the place unification approach requires more complex tokens that, nevertheless, have a common structure.

The proposed synergistic testing and debugging methodology is applied to individual program components as well as to composed applications. The synergy of formal and program execution tools detects errors in the design, in the implementation of program components and in the execution environment. This synergy also provides objective conclusions about the cause of errors and reveals any discrepancies between design and implementation. The extra effort of designing specifications for message passing components is justified as it assures reliability and reduction of production costs of message passing applications.

At the present we have completed the specification composer, which composes template CPNs (in the textual form presented in paragraph 3.2) driven by the same script that drives the composition of the actual parallel program. Our next task is to investigate the integration of existing tools (simulators, visualisation and computation steering tools) with our methodology. The features of the PEP tool are very close to our needs. We consider the possibility to derive our specification components and their associated sequential program components automatically, by using the B(PN)^2 specification language to model the specifications. PEP can automatically produce the associated M-net as well as some form of C-Code that implements the algorithm. The effort to extend or modify the relevant PEP modules in order to derive the Ensemble components is justified from the fact that in this case, we could use the already implemented tools and the interplay mechanism of PEP to handle the interaction between program executions and specification simulations.

We intend to further extend our methodology for performance evaluation of the Message Passing applications. Our long-term aim is to create an integrated software engineering support tool (that uses advanced monitoring/debugging or computation steering systems driven by a specification simulator/analyser etc.) for the development of reliable message passing applications.

References

[1] Agha, G. A. "The Emerging Tapestry of Software Engineering", *IEEE Concurrency*, **5**(3), 1997, 2–4.

[2] Best, E., Fleischhack, H., Fraczak, W., Hopkins, R. P., Klaudel, H. and Pelz, E. "A Class of Composable High Level Petri Nets", *Proc. of Application and Theory of Petri Nets'95*, LNCS 935, Springer, Torino, 1995, 103–118.

[3] Best, E., Fleischhack, H., Fraczak, W., Hopkins, R. P., Klaudel, H. and Pelz, E. "An M-net Semantics of B(PN)^2 ", *Proc. of Structures in Concurrency Theory*, Workshops in Computing, 85–100.

[4] Best, E. and Grahlmann, B. "PEP-More than a Petri Net Tool", *Proc. of TACAS'96*, LNCS 1055, Springer, 1996.

[5] Best, E. and Hopkins, R.P. "B(PN)^2-A Basic Petri Net Programming Notation", *Proc. of PARLE'93*, LNCS 694, Springer, Munich, 1993, 379–390.

[6] Best, E. and Thielke, T. "Refinement of Coloured Petri Nets", *Proc. of 11th International Symposium on Fundamentals of Computation Theory*, LNCS 1279, Springer, Krakow, 1997, 105–116.

[7] Biberstein, O. and Buchs, D. "Structured Algebraic Nets with Object-Orientation", *Proc. of Workshop on Object-Oriented Programming and Models of Concurrency'95*, Torino, 131–145.

[8] Cotronis, J.Y. "Efficient Composition and Automatic Initialisation of Arbitrarily Structured PVM Programs", *Proc. of 1st IFIP International Workshop on Parallel and Distributed Software Engineering*, Chapman & Hall, Berlin, 1996, 74–85.

[9] Cotronis, J.Y. "Efficient Program Composition on Parix by the Ensemble Methodology", *Proc. of Euromicro Conference'96*, IEEE Computer Society Press, Prague, 1996.

[10] Cotronis, J.Y. "Message Passing Program Development by Ensemble", *Proc. of PVM/MPI'97*, LNCS 1332, Springer, Cracow, 1997, 242–249.

[11] Cotronis, J.Y. and Tsiatsoulis, Z. "Composition of Specifications of Message Passing Applications Composed by the Ensemble Methodology", *Proc. of 6th Hellenic Conference on Informatics*, Ekdoseis Neon Technologion, volume I, Athens, 1997, 299–312.

[12] Cotronis, J.Y. and Tsiatsoulis, Z. "Specification Composition for the Verification of Message Passing Program Composition", *Proc. of 3rd IFIP International Conference on Reliability, Quality and Safety of Software Intensive Systems*, Chapman & Hall, Athens, 1997, 95–106.

[13] Cotronis, J.Y. "Developing Message Passing Applications on MPICH under Ensemble", *Proc. of PVM/MPI'98*, 1998.

[14] Eisenstadt, M. "My Hairiest Bug War Stories", *Communications of the ACM*, **40**(4), 1997, 30–37.

[15] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V. "PVM 3 User's guide and Reference Manual", *ORNL/TM- 12187*, 1994.

[16] Geist, J.A., Kohl, J.A. and Papadopoulos, P.M. "CUMULVS: Providing Fault-Tolerance, Visualisation and Steering of Parallel Applications", 1996.

[17] Gelernter, D. and Carriero, N. (1992) "Coordination Languages and their Significance", *Communications of the ACM*, **35**(2),1992, 97–107.

[18] Gorton, I. and Jelly, I.E. (1997) "Software Engineering for Parallel and Distributed Systems: Challenges and Opportunities", *IEEE Concurrency*, **5**(3), 1997, 12–15.

[19] Grahlmann, B. "The PEP Tool", *Proc. of Application and Theory of petri Nets'97*, LNCS 1248, Springer, Toulouse, 1997.

[20] Heiner, M. "Petri Net Based Software Validation", *International Computer Science Institute ICSI TR-92-022*, Berkeley, California, 1992.

[21] Jensen, K. "Coloured Petri Nets: A High Level Language for System Design and Analysis", *Advances in Petri nets 1990*, LNCS 483, Springer, 342–416.

[22] Jensen, K. "Design/CPN Reference Manual", *Aarhus University*, 1999.

[23] Kindler, E. "A Compositional Partial Order Semantics for Petri Net Components", *Proc. of Application and Theory of petri Nets'97*, LNCS 1248, Springer, Toulouse, 1997.

[24] Lakos, C.A. "LOOPN++ User Manual", *University of Tasmania, Department of Computer Science*, 1997.

[25] Lakos, C.A. "The Consistent Use of Names and Polymorphism in the Definition of Object Petri Nets", *Proc. of Application and Theory of Petri Nets'96*, LNCS 1091, Springer, Osaka, 1996, 380–399.

[26] Maillet, E. "TAPE/PVM: An Efficient Performance Monitor for PVM applications - User Guide", *LMC-IMAG*, 1995.

[27] Message Passing Interface Forum "MPI: A Message Passing Interface Standard."

[28] Orfali, R. and Harkey, D.: *Client / Server Programming with JAVA and CORBA, 2nd Edition*, New York, John Wiley And Sons Inc., 1998, ISBN: 047124578X.

[29] Parsytec Computer Gmbh, "Parix v. 1.9 Manual", 1993.

[30] Sibertin-Blanc, C. "A Client-Server Protocol for the Composition of Petri Nets", *Proc. of Application and Theory of Petri Nets'93*, LNCS 691, Springer, Chicago, 1993, 377–396.

[31] Sibertin-Blanc, C. "Cooperative Nets", *Proc. of Application and Theory of Petri Nets'94*, LNCS 815, Springer, Zaragoza, 1994, 471–490.

[32] Sibertin-Blanc, C., Hameurlain, N. and Touzeau, P. "SYROCO: A C++ Implementation of Cooperative Objects", *Proc. of Workshop on Object- Oriented Programming and Models of Concurrency*, Torino, 1995.

[33] Tsiatsoulis, Z, Dozsa, G, Cotronis, J.Y, Kacsuk, P, "Associating Composition of Petri Net Specifications with Application Designs in Grade", *Proc. of PDP'99*, 1999.

[34] Ungar, D., Lieberman, H. and Fry, C. "Debugging and the Experience of Immediacy", *Communications of the ACM*, **40**(4), 1997, 38–43.