In Proc. Encress 97. 3rd International Conference on Reliability, Quality & Safety of Software-Intensive Systems, Chapman and Hall, May 1997

Specification composition for the verification of message passing program composition

J.Y. Cotronis and Z. Tsiatsoulis
Department of Informatics, University of Athens
TYPA, Panepistimiopolis, 157 71 Athens, Greece
tel. +30 1 7291885 fax +30 1 7219561
e-mail: {cotronis, zack}@di.uoa.gr

Abstract

We present a specification composition technique which supports the message passing composition of applications by the Ensemble methodology. In Ensemble applications are built by composing reusable executable program components designed with scalable communication interfaces. We define reusable specifications of program components, using coloured Petri nets, which are then composed to obtain the specification of the application. The composition is controlled by the same script that is used to compose the application.

1 INTRODUCTION

Software composition has been suggested as a methodology for building large scale applications. Software components having an open architecture, flexible for reuse by different applications, are combined to compose applications. Software composition has three major aspects (Nierstrasz and Meijler, 1995): (i) macro expansion (ii) higher order functional composition and (iii) binding of communication channels. Significant work has been done the past few years in the area of software composition, mainly on the first two aspects and their implications in the framework of object oriented methodologies (Nierstrasz et al., 1992) and less on the third (Nierstrasz, 1995).

We have developed a message passing program implementation methodology, called Ensemble (Cotronis, 1996a; Cotronis 1996b), by which message passing applications are composed out of reusable software components by binding their communication channels. The emergence of Message Passing Environments (MPE), such as PVM (Geist et al., 1994), MPI (McBryan, 1994), Parix, provide a useful abstraction of the underlying architecture simplifying implementation. However, the software engineering step from message passing design to implementation remains a demanding task, as it involves the programming of the sequential parts, computing a result or providing a service, intermixed with the explicit programming of process management: process creation and identification, process interaction, process topologies and their mapping onto the virtual architecture. The programming imposed by process management makes programs much more difficult to develop and maintain. Important aspects of parallel programs, such as scalability and reusability are frequently neglected, as they have to be explicitly programmed. Scalability is relatively easy to program when the problem has some global regularity, usually expressed by some function; reusability of executables is more difficult as process management is usually encoded in them and processes may only operate within the context of one application.

Ensemble alleviates the development and maintance difficulties of message passing programs. Ensemble is not a new message passing environment and does not even demand any changes to MPEs; it is also independent of any MPE; it restricts, as all methodologies, the implementation space of applications to a common software architecture. An application in Ensemble is an ‘ensemble’ of a script, which specifies the application processes, their topology and mapping, and of reusable executable program components, which do not involve any process management activities, but only computations providing a result or service. The script is interpreted by programs (tools of Ensemble) which compose the application.

However, composing message passing applications from reusable components is prone to a number of errors: unspecified or incompatible binding of communication channels, wrong behaviour of the composed system, wrong result, etc. These errors may emerge during program execution in the form of undelivered messages, deadlock situations, non-terminating programs, etc. Although, the architecture of Ensemble applications supports their efficient debugging, the general problems of debugging (e.g. no guarantee of absence of bugs), as well as the problems of parallel program debugging (e.g. non-deterministic behaviour of programs, non-reproducibility of behaviour) still apply. We would like therefore to predict the behaviour of the composed applications or even formally verify that the composed programs behave according to the required specifications. The behaviour of a composed message passing application cannot, in general, be analytically determined from the known behaviour of its components. But we may compose the formal specifications of individual components to obtain a composed formal specification of the application which may then be tested and verified. In the debate about the usefulness of formal methods in software development we have followed the middle way (Jackson and Wing, 1996). In the presence of numerous formal models which all address the same problem, but very few of them are actually used (Parnas, 1996), we do not intend to present another model. We would use already developed formalisms and their associated theory and tools which are suitable for Ensemble as software engineering formal testing methods. We have used the Petri net formalism for expressing and composing specifications as it is well founded, has been widely used to specify parallel software systems and is supported by a number of tools.

In the next section we outline the Ensemble methodology and its tools. In section 3 we discuss the requirements for component specification. In section 4 we describe the general form of component specifications and present the composition. In section 5 we apply the composition to three applications. Finally, we present our conclusions and plans for future work.

2 OUTLINE OF THE ENSEMBLE METHODOLOGY AND ITS TOOLS

We outline Ensemble using as an example the Distribution of Maximum application: terminal processes which are each given an integer parameter and require the maximum of these integers; each terminal process sends its value to an associated relay process and (eventually) receives from it the required global maximum (GM). Relay processes receive values from their terminals, find their local maximum (LM), exchange LMs with the other relays, and find their maximum (GM); they finally send GM to their terminal processes. The Ensemble implementation consists of the application script and the two executable reusable components, the terminal and the relay. The application is composed by a launching program, which interprets the scripts and sets-up the application.

2.1 The Ensemble script

The script for Distribution of Maximum application (with three relays and five terminals) is shown in the first column of Figure 1. The script is structured in three main parts:

The first part, headed by PCG, specifies the Process Communication Graph (PCG) of the application independent of any MPE. PCGs are a natural structure for specifying processes and their communication dependencies and are close to program design. Nodes on a PCG denote processes and arcs the communication channels (dependencies) between them. PCGs have been used in modelling, in dynamic analysis and simulation, in mapping techniques, etc. In the PCG part we first specify the components involved (e.g. T and R), then the processes instantiated from each component (e.g. T[1],…,T[5] and R[1],…,R[3]) and finally the communication channels between the processes.

Scalability is an important aspect of parallel programs. Usually, due to programming complexity, we think of scalability as global factors in an application, e.g. sizes of dimensions of a grid topology. But there may be other local scalability factors. For example, relays 1 and 2 have two terminals and relay 3 only one; if the number of terminals increases to ten, all five of the new may be assigned to relay 3, or to two new relays, two to relay 4 and three to relay 5. We consider these possibilities as design choices which should all be supported. In general, scaling of applications requires replication of processes and their interconnections. For some process topologies, such as a tours, it is sufficient to replicate identical processes each having the same number of connections. But for other topologies, such as master/slave, each replicated process may have a distinct number of interconnections, possibly within a range. To support the global as well as local scalability of applications we specify for each process in the script, its number of ports. Process ports are identified by the name of their communication type and a unique index within the type. The terminal component, for example, has two communications types (S_{in} and S_{out}) and all terminal processes exactly one port of each type. The relay component however, has four communication types (S_{in}, S_{out}, P_{in}, P_{out}). All relay processes have two ports of type P_{in} and P_{out}, but different number of ports of S_{in} and S_{out} types. Channels are defined by one-to-one associations of process ports.

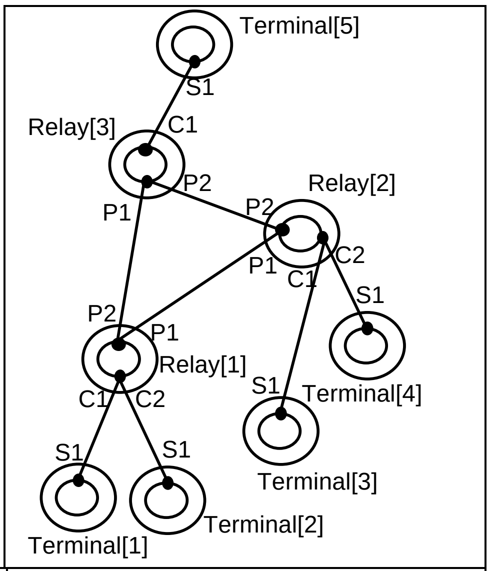
A tool program, the PCG-builder, reads the PCG part and actually generates the PCG. The PCG for our example is depicted in Figure 1, next to the PCG part of the script. For reasons of simplicity channels connecting X_{in}[i] with Y_{out}[j] ports and X_{out}[i] with Y_{in}[j] ports are depicted as one bi-directional channel connecting X[i] with Y[j].

The second script part, headed by Parallel System, specifies the annotation of nodes (processes) and arcs (channels) of the PCG with information required for the composition of the application on a specific target MPE. In the example script of Figure 1 the target system is PVM; nodes are annotated by the host name on which they will be spawned (optional in PVM); arcs are annotated by the tag number which is required to identify the abstract PVM channels between processes (default on the script annotates arcs by unique tags).

The third script part, headed by Sequential Components, specifies the further annotation of nodes with process loading information. The files of the reusable executable components are specified and for each process its command line parameters. The second and third parts are interpreted by the annotation Builder which annotates the PCG created by the PCG builder. In Figure 1, below the general PCG, the annotation of some of its nodes and channels is shown.

APPLICATION Distribution_Maximum;
PCG
Components
T port-types : S_{in}, S_{out};
R port-types : C_{in}, C_{out}, P_{in}, P_{out};
Processes
T[1], T[2], T[3], T[4], T[5] #ports=S_{out}:1, S_{in}:1;
R[1], R[2] #ports = C_{out}:2, C_{in}:2, P_{out}:2, P_{in}:2;
R[3] #ports = C_{out}:1, C_{in}:1, P_{out}:2, P_{in}:2;
Channels
T[1].S_{out}[1] -> R[1].C_{in}[1]; R[1].C_{out}[1] -> T[1].S_{in}[1];
T[2].S_{out}[1] -> R[1].C_{in}[2]; R[1].C_{out}[2] -> T[2].S_{in}[1];
T[3].S_{out}[1] -> R[2].C_{in}[1]; R[2].C_{out}[1] -> T[3].S_{in}[1];
T[4].S_{out}[1] -> R[2].C_{in}[2]; R[2].C_{out}[2] -> T[4].S_{in}[1];
T[5].S_{out}[1] -> R[3].C_{in}[1]; R[3].C_{out}[1] -> T[5].S_{in}[1];
R[1].P_{out}[1] -> R[2].P_{in}[1]; R[2].P_{out}[1] -> R[1].P_{in}[1];
R[1].P_{out}[2] -> R[3].P_{in}[1]; R[3].P_{out}[1] -> R[1].P_{in}[2];
R[2].P_{out}[2] -> R[3].P_{in}[2]; R[3].P_{out}[2] -> R[2].P_{in}[2];

P C G B u I l d e r



PARALLEL SYSTEM
Environment PVM3
PVM3_Annotation
tagID : default;
PVM3_Options
Allocation
R[1], T[1], T[2] at euridiki;
R[2], T[3], T[4] at kadmos;
R[3], T[5] at lavdakos;
SEQUENTIAL COMPONENTS
Executable files
T : path default file terminal;
R : path default file relay;
Execution Parameters
T[1]:6; T[2]:999; T[3]:7; T[4]:8; T[5]:9;

a n n o t a t i o n B ui ld er

Node 1	name	: T[1]
	allocation	: euridiki
	file	: terminal
	path	: default
	parameters	: 6
Node 6	name	: R[1]
	allocation	: euridiki
	file	: relay
	path	: default
	parameters	: (None)

Channel 1 : 1.S_{out}[1] -> 6.C_{in}[1] tagid 1
Channel 4 : 4.S_{out}[1] -> 7.C_{in}[2] tagid 4
Channel 7 : 6.C_{out}[2] -> 2.S_{in}[1] tagid 7
Channel 11: 6.P_{out}[1] -> 7.P_{in}[1] tagid 11
Channel 14: 7.P_{out}[2] -> 8.P_{in}[2] tagid 14

Figure 1 The application script and the annotated PCG.

2.2 The reusable components

They compute a result or provide a service and do not involve any process management or assume any topology in which they operate. They have open ports for communicating with any compatible processes in any application. A port is a structure, which may store communication parameters necessary for sending and receiving messages; for example in PVM these parameters are pairs of values of task identifiers, which are unique numbers identifying a process, and tag identifiers. Ports of the same type form arrays and arrays of all types form the interface of the component. All send and receive operations refer to ports, identified by a communication type and an array index. At the time of process creation the launching program provides the actual number of ports of each type, as well as, the values for the communication parameters for each port. Processes set-up their interface by calling appropriate routines. Each MPE demands its own routines for setting up the component interfaces. A common structure for components (Figure 2), has been developed which hides these differences and unifies the appearance of components of any MPE.

```
void main()                         /* terminal */
{ InterfaceType Interface[2];
  MakePorts(Interface);
  SetInterface(Interface);
  realmain(Interface);          /* main action */  }

void realmain (Interface);
{ /* terminal pseudoccode */
    send local value to relay             (to $S_{out}$ type)
    receive maximum from relay            (from $S_{in}$ type)
}
```

```
void main()                         /* relay */
{ InterfaceType  Interface[4];
  MakePorts(Interface);
  SetInterface(Interface);
  realmain(Interface); }

void realmain(Interface);
{ /* relay pseudocode  */
    receive values from terminals         (from $C_{in}$ type)
    find the local maximum (LM) of values
    send LM to all other relays           (to $P_{out}$ type)
    receive LMs from all other relays     (from $P_{in}$ type)
    find global maximimum (GM)
    send GM to terminals                  (to $C_{out}$ type)  }
```

Figure 2 The structure of Terminal and Relay program components.

For each component we declare the number of communication types that it requires, indicated by the size of the array Interface. Terminals have two and Relays four communication types. Processes first call MakePorts to set-up the appropriate number of ports in Interface, then call SetInteface to set values to ports of Interface and they call their realmain actions. The component executables are reusable in any application in the given MPE.

2.3 The Launcher program

It is the program that actual composes applications, universal for all applications in the same MPE, one Launcher program for each MPE. The Launcher visits the annotated PCG nodes and spawns processes providing to each spawned process the number of its ports of each type (to be processed by MakePorts), the port information (to be processed by SetInterface) and its command line parameters. Now the parallel program is composed and running.

We have only outlined the aspects of Ensemble methodology and its tools which are relevant in the context of this paper. A detailed description of Ensemble in PVM and Parix may be found in (Cotronis, 1996a) and (Cotronis, 1996b), respectively.

3 REQUIREMENTS FOR SPECIFICATIONS AND THEIR COMPOSITION

Our aim is to support the Ensemble methodology with formal tools for testing and verifying programs prior to their execution. To reflect the Ensemble architecture of parallel programs we need to define component specifications, process specifications (instantiations of component specifications) and their composition. Component specifications specify the behaviour of program components. They should be reusable, permitting the generation of process specifications, as required by the script. Component specifications should have scalable interfaces, specifying the valid range of values for each of their communication types, e.g. fixed (as S_{in} of terminals) or any positive integer (as C_{in} of relays) or any non-negative integer (as P_{in} of relays). They should identify their input and output ports as well as the type of data that is sent and received through them. Process specifications should be generated from component specifications as mechanically as processes are generated from program components. At the time of their generation the number of ports specified in the script should be validated and their interface should be fixed.

Specification composition involves the port interconnections integrating individual process specifications into one. During composition we have to check the compatibility of port interconnections: that each output port is connected to a single input port and vice-versa, and that the data expected on the connected ports is of the same type. In general, the compatibility of port connections also depends on being synchronous or asynchronous. We restrict our presentation to asynchronous communications. At the end of the composition we have to check for unconnected ports. Until this step all testing and validation is static.

Having composed the specifications we verify their integrated behaviour, that is to say the dynamic aspects of the composed system. Analytical tools may be employed proving general properties, such as absence of deadlock; causal graphs may be produced or simulations may be performed. We may only provide guidelines on using these tools in the context of message passing applications and Ensemble. For example, we may verify the number and ordering of the send/receive actions, which are the main reason for the ill-behaviour of message passing applications, such as deadlocks and non-termination. If more send than receive operations are performed some messages will be undelivered, if more receives are specified then there will be a deadlock. If the wrong order of send and receives is specified then the system may not deadlock, but produce a wrong result.

Having tested or verified the dynamic properties of the composed specification we may test its functionality, the correctness of the result it computes or the service it provides. We are not only concerned with send/receive operations but with internal non-communication actions.

4 COMPOSITION OF SPECIFICATIONS SUPPORTING ENSEMBLE

We have used the Petri net formalism for expressing and composing specifications. Petri-nets have a well founded theory, have been widely used to specify parallel software systems and are supported by a number of tools. Petri-net semantics have been shown to be suitable for the composition of specifications of message passing applications. In (Kindler, 1996) the composition of Petri net components is modelled with place fusion, which corresponds to asynchronous message passing. In (Best et al., 1995) large High Level nets are constructed from smaller components, by transition synchronisation, which allows composition in a manner similar to process algebras like CCS. We use Coloured Petri nets which allow the modeller to create simple and easily manageable descriptions, without losing the ability of formal analysis (Jensen, 1990).

4.1 Program component specifications: template CPN

The formalisms described in the previous section are, in a way, very close to our needs. We use Petri net components, which represent the specifications of program components of Ensemble applications, and compose them directed by Ensemble scripts. In the case where the interface of a component is fixed, as for example in the terminal component (it has one port of types S_{in} and S_{out}) its specification can be modelled directly with CPNs. The general case, however, where the interface of a component is parametric, cannot be directly modelled using CPNs, since CPNs must have a fixed structure (specific number of places and connections with transitions). Thus, we need to extend the component specifications with open scalable interfaces for them to be reusable and to generate process specifications. We define template CPNs, which resemble CPN formalism but also contain additional information to specify the interface parametrically. The template CPN is a parametric net-structure having a unique name, from which process component names may be generated by unique indexing. Similar to the Ensemble program components they also have two kinds of parameters: port interface parameters (the number of ports of each communication type) and application parameters (like the integer value of terminal components).

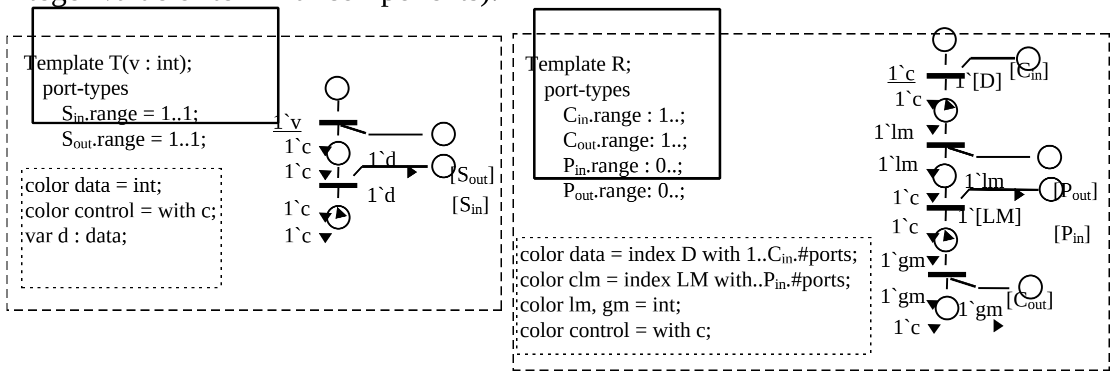


Figure 3 The template CPNs for Terminal and Relay Components.

Application parameters appear in parentheses at the heading of the template and also symbolically index the initial place of the net structure (Figure 3). The symbolic initial marking will be replaced by actual values when process specifications are generated. Following the heading of the template the valid range of ports for each communication type is specified. We have used a simple notation *from..to*, where *from* could be any non-negative number and *from<=to*. In the case where a communication type has fixed number of ports, say N, the expression becomes N..N. For example see template T in Figure 3 (S_{in} and S_{out} have a range 1..1). In the case of parametrically specified number of ports the value of *from* could be either 1 or 0. The former specifies that the component can have at least one port of this type and the latter that it may not have any ports of this type. If the value for *to* is unspecified then there is no upper bound on the number of ports. In Figure 3, R is specified to have a 1.. range for its C_{in} and C_{out} port types and a 0.. range for its P_{in} and P_{out} port types. Usually the number of ports is required as a parameter in the net. For this the P.#ports symbol is used to indicate the number of ports of type P.

Templates name their interface places according to the corresponding port types of the component, enclosed in square brackets distinguishing them from other places. The bracket notation is used for variables in the inscriptions on the arcs joining interface places, which are distinct for each port. The template structure has a local declaration node, represented as a dotted line rectangle, which contains definitions of colour sets, variables, values etc., representing data types and tokens. All generic parts of a template, are enclosed in a solid line (Figure 3).

4.2 Process specifications: composable CPN

Process specifications, which are composable CPNs are instantiated from their corresponding template CPN by providing actual values for its instance number, number of ports of each type and application parameters. All of these exist in the Ensemble script. Composable CPNs are normal coloured Petri nets uniquely named by indexing the template name with an instance number. Figure 4 illustrates the composable CPNs for components T[1] and R[3] as specified in the script in Figure 1. Appearance of the number of ports in local declarations are replaced by their actual values. The net structure is generated by replicating interface places and the input and output arcs to and from these places along with their inscriptions. Upon replication the brackets are removed and the names in the brackets are indexed by unique integers.

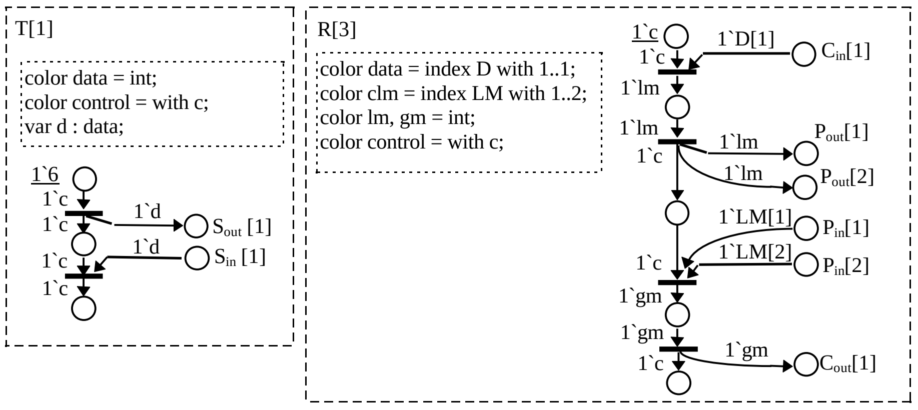


Figure 4 The composable CPNs for T[1] and R[3].

4.3 The application specification: composed CPN

The composable CPNs may now be composed in order to produce the complete application specification, which we call composed CPN. The composition of the composable CPNs is performed according to the channel section of the script. The names of the interface places are the same as the names of the corresponding ports in the script. Furthermore the name of each composable CPN is the same as the name of the corresponding process in the script. By prefixing the name of the composable CPN to the name of the interface place, a full unique name for each interface place is constructed. For each channel between two ports defined in the script the corresponding places of the composable CPNs are fused.

5 COMPOSING APPLICATION SPECIFICATIONS

In this section we present three example applications where the proposed technique is applied. All three applications use the template CPNs for terminal and relay components.

5.1 An erroneous application

In the first example we demonstrate the composition of an erroneous script; its PCG is:

Components
T port-types S_{out}, S_{in}; R port-types C_{out}, C_{in}, P_{out}, P_{in};
Processes
T[1], T[2] #ports = S_{out}:1, S_{in}:1;
R[1] #ports = C_{out}:1, C_{in}:1, P_{out}:1, P_{in}:1;
Channels
T[1].S_{out}[1] -> R[1].C_{in}[1]; R[1].C_{out}[1] -> T[1].S_{in}[1];
T[2].S_{out}[1] -> R[1].P_{in}[1]; R[1].P_{out}[1] -> T[2].S_{in}[1];

There are two terminals and one relay. The S_{out}[1] and S_{in}[1] ports of T[1] are connected with the C_{in}[1] and C_{out}[1] ports of R[1], respectively. But ports S_{out}[1] and S_{in}[1] of T[2] are connected with the $P_{in}[1]$ and $P_{out}[1]$ ports of R[1] respectively, instead of $C_{in}[2]$ and $C_{out}[2]$ ports respectively. The ports are compatible, they exchange integer values, but the behaviour of the application as specified by the script is not correct. The composition of the application is depicted in Figure 5

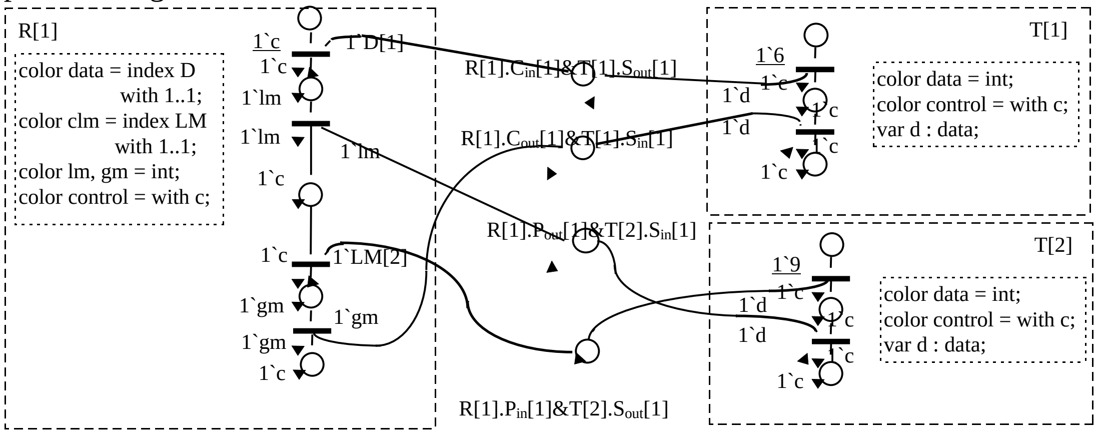


Figure 5 The composed specification of an erroneous script.

The causality graph of the composed specification will depict that the number and order of send and receive operations is not correct. R[1] performs one send to its $P_{out}[1]$ port and one receive from its $P_{in}[1]$ port, but there is no other relay in the application. Also R[1] performs a send to its $P_{out}[1]$ port before a receive from T[2]. What actually happens is that T[1] gets the global maximum, but T[2] gets the value of T[1].

5.2 Distribution of maximum application

The Ensemble script is given in Figure 1. We use a box notation for the composable CPNs, which hides their internal structure concentrating to their interface connections. Each composable CPN-box has the same name as the composable CPN and its ports are depicted by black dots in the boundaries of the box, along with their names. The net is shown in Figure 6. By using CPN tools we may verify that the behaviour of the composed specification is correct.

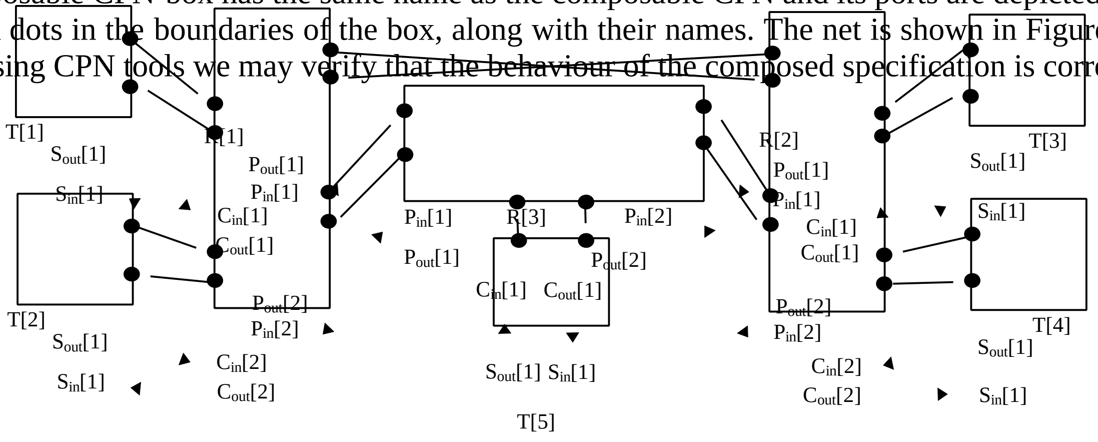


Figure 6 The composed CPN for Distribute Maximum by all-to-all topology.

5.3 Distribution of maximum by tree topology

In this variation of the Distribution of Maximum application we maintain the relationship of the five terminals to the three relay processes, but relay processes are organised in a tree, with R[3] being the root. Relays 1 and 2 have only one P_{out} and one P_{in} ports which are connected to the C_{in} and C_{out} ports, respectively, of their parent Relay 3, which has no P_{out} and P_{in} ports. The process structure is a tree of height 2: the terminal processes 1,2,3,4 are at level two; R[1], R[2], T[5] at level one; and R[3] is the root. The PCG part of the application script is:

Components
T port-types S_{out}, S_{in}; R port-types C_{out}, C_{in}, P_{out}, P_{in};
Processes
T[1], T[2], T[3], T[4], T[5] #ports = S_{out}:1, S_{in}:1;
R[1], R[2] #ports = C_{out}:1, C_{in}:1, P_{out}:1, P_{in}:1;
R[3] #ports = C_{out}:3, C_{in}:3, P_{out}:0, P_{in}:0;
Channels
R[1].C_{out}[1] -> T[1].S_{in}[1]; T[1].S_{out}[1] -> R[1].C_{in}[1];
R[1].C_{out}[2] -> T[2].S_{in}[1]; T[2].S_{out}[1] -> R[1].C_{in}[2];
R[2].C_{out}[1] -> T[3].S_{in}[1]; T[3].S_{out}[1] -> R[2].C_{in}[1];
R[2].C_{out}[2] -> T[4].S_{in}[1]; T[4].S_{out}[1] -> R[2].C_{in}[2];
R[3].C_{out}[3] -> T[5].S_{in}[1]; T[5].S_{out}[1] -> R[3].C_{in}[3];
R[3].C_{out}[1] -> R[1].P_{in}[1]; R[1].P_{out}[1] -> R[3].C_{in}[1];
R[3].C_{out}[2] -> R[2].P_{in}[1]; R[2].P_{out}[1] -> R[3].C_{in}[2];

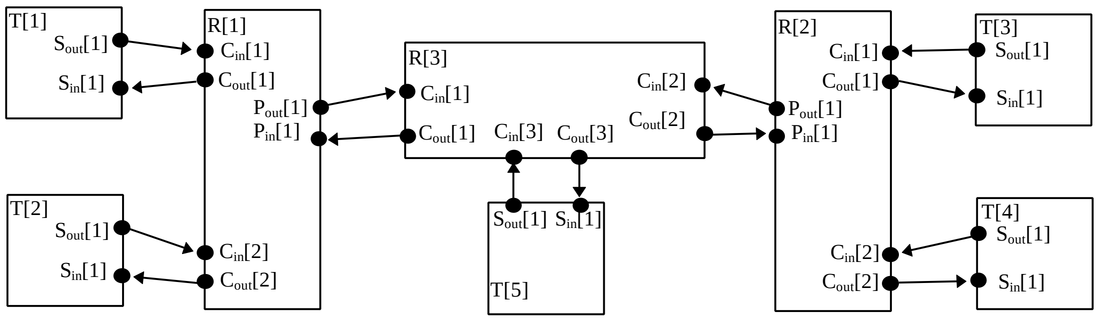


Figure 7 The composed CPN for Distribute Maximum by tree topology.

At each level, the relay processes receive the values from their clients, select the maximum and propagate it to the next level up. The root selects the maximum and sends it to its client processes, the two relays and T[5]. The relay processes below the root do the same until the maximum reaches their terminal processes. The composed specification net is shown in Figure 7. Again by using CPN tools we may verify that this solution of the Distribution of Maximum application is valid. We demonstrated that although terminal and relay template specifications were originally designed for one solution, they are reused to verify that the tree solution to the Distribution Maximum application is also correct.

6 CONCLUSIONS- FUTURE WORK

We have presented a specification composition technique which supports the message passing program composition of the Ensemble methodology. We have defined descriptions of CPNs with scalable interfaces, called template CPNs, to specify the behaviour of scalable reusable program components. From the template CPNs we generate composable CPNs, which are pure CPN descriptions. We have used the PN composition technique of (Kindler, 1996) adapted to the composition of Ensemble applications as described by the script. During composition static information as specified by the script is validated (for example, the number of communication ports within the range and the compatibility of port interconnections). The composition is directed by the script. The correspondence of program and specification composition is depicted in Figure 8. In the middle there is the application script, which is used by the application Launcher to compose applications (left hand side). The script is also used by the specification composer to compose application specifications (right hand side).

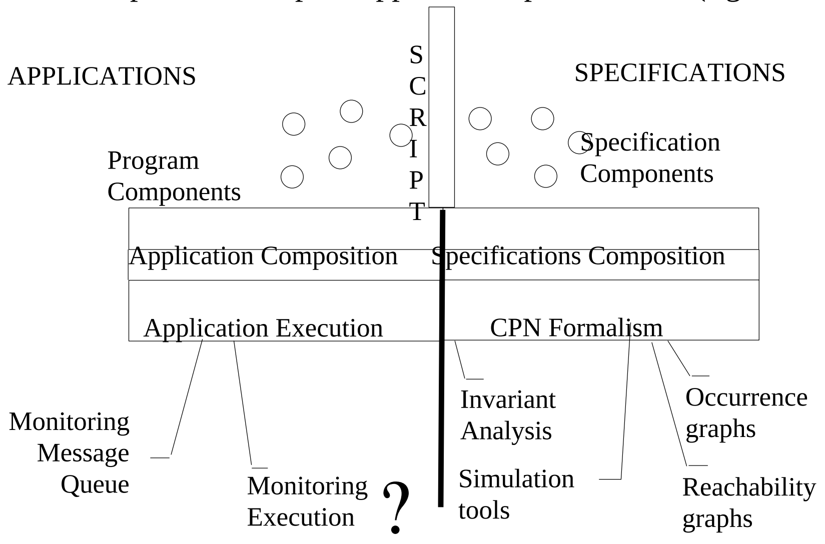


Figure 8 Ensemble methodology supported by composition of specifications.

We are currently implementing tools for designing template CPNs, generating from them composable CPNs and composing them according to Ensemble scripts. Our effort does not simply aim to support the Ensemble methodology by a formal specification tool. We envisage of using Ensemble and its associated tools as a viable means of bridging the gap between the disjoint worlds of specifications and program executions. Usually specifications are obtained before program design and program implementation. But this view is not valid in the software composition approach. Programs and their specification are composed together. Especially in Ensemble both may be independently produced from the script. In a sense the composed specifications are the semantics of composed programs under the assumption that the component specifications are correct.

To alleviate possible discrepancies between component specifications and component implementations we may use one to test the other. On the one hand, tracing information of the composed application may be passed to a simulator of the composed specifications. Thus, the behaviour of the application is not only monitored as it is running, but actually tested. The programmer is not obliged any more to inspect detailed and confusing charts, visualisation of executions, but the simulation system may check against the specifications either automatically in the background or by analysing a trace file of erroneous behaviour. The use of tracing information in conjunction with specification simulation information should always be used during individual component development. On the other hand, the specification simulator may be used as an advanced breakpoint mechanism which controls the execution of the actual program. Specifications and programs are not in disjoint worlds any more, but are inter-related. We believe that in this scheme the extra effort of designing specifications of reusable components is justified as it assures reliability and reduction of production costs of message passing applications.

7 REFERENCES

Best, E., Fleischhack, H., Fraczak, W., Hopkins, R.P., Klaudel, H. and Pelz, E. (1995) A Class of Composable High Level Nets, Application and Theory of Petri Nets ´95.

Cotronis, J.Y. (1996a) Efficient Composition and Automatic Initialization of Arbitrarily Structured PVM Programs, in *Software Engineering for Parallel and Distributed Systems* (ed. I. Jelly, I. Gorton and P. Croll), Chapman & Hall.

Cotronis, J.Y. (1996b) Efficient Program Composition on Parix by the Ensemble Methodology, Euromicro Conference 96, IEEE Computer Society Press, Prague.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V. (1994) PVM 3 User’s guide and Reference Manual, ORNL/TM-12187.

Jackson, D. and Wing, J. (1996) Lightweight Formal Methods, *IEEE Computer Magazine*, **29(4)**.

Jensen, K. (1990) Coloured Petri Nets: A High Level Language for System Design and Analysis, in *Advances in Petri nets*, (ed. G. Rozenberg), Lecture Notes in Computer Science, **483**, Springer-Verlag.

Kindler, E. (1996) A Compositional Partial Order Semantics for Petri Net Components. SFB-Bericht 342/06/96 A, Technische Universitaet Muenchen.

McBryan, O. A. (1994) An overview of Message Passing Environments, *Parallel Computing*, **20**, 417-444.

Nierstrasz, O. (1995) Regular Types for Active Objects, in *Object-Oriented Software Composition*, (eds. O. Nierstrasz and D. Tsichritzis) Prentice Hall.

Nierstrasz, O. and Meijler, T.D. (1995) Research Directions in Software Composition. *ACM Computing Surveys*, **27(2)**.

Nierstrasz, O., Gibbs, S. and Tsichritzis, D. (1992) Component-Oriented Software Development. *Communications of the ACM*, **35(9)**.

Parnas, D.L. (1996) Mathematical Methods: What We Need and Don’t Need. *IEEE Computer Magazine*, **29(4)**.

8 BIOGRAPHY

Dr. J.Y. Cotronis obtained his Ph.D. in Computer Science in 1982 from the Computing Laboratory, University of Newcastle-upon-Tyne, where he worked as a Research Associate in projects in the area of parallelism. He has been involved in a number of R&D projects in industry and academia. He is an Assistant Professor and his current research interests are on methodologies and supporting tools for composing and porting parallel applications.

Zacharias Tsiatsoulis studied Informatics at the University of Athens. He received his B.Sc. degree from the Department of Informatics in 1993. He is currently a Ph.D. student at the Department of Informatics, University of Athens. His research interests include specification, testing and verification of composed message passing applications.