Microprocessing and Microprocessors, 21 (1998), 595-603, 1998

Specification composition for the verification of message passing program composition

J.Y. Cotronis and Z. Tsiatsoulis

Department of Informatics, University of Athens, 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, for improving the reliability of message passing applications composed by the Ensemble methodology. In Ensemble, applications are built by composing reusable executable program components designed with scalable communication interfaces. The composition is controlled by scripts. We define reusable specification components associated to program components, as well as their composition directed by the same Ensemble scripts, thus obtaining specifications of applications. We propose an extension of coloured Petri nets, which is used to define specification components. Composed specifications and applications may be validated or verified by available tools.

Keywords: *Message Passing Composition, Specification Composition, Reusable Program Component, Reusable Specification Component, Software Reliability, Coloured Petri Net.*

1 INTRODUCTION

Software composition has been suggested as a methodology for building large-scale applications. Reusable software components having an open architecture are combined to compose applications. Software composition has three major aspects [15]: (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 [16] and less on the third [14].

We have developed a message passing (MP) program implementation methodology, called Ensemble [4, 5], by which MP applications are composed out of reusable software components by binding point-to-point communication channels. Ensemble provides a common software architecture for MP applications, on any Message Passing Environment (MPE). The emergence of MPEs, such as PVM [6], MPI [12] and Parix [18], provide a useful abstraction of the underlying architecture thus simplifying implementation. However, the software engineering step from 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, only when the problem has some global regularity. Reusability of executables is limited as process management is usually encoded in them and consequently, processes may only operate within the context of a single application.

Ensemble alleviates development and maintenance difficulties of MP programs. 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. The script is interpreted by programs (tools of Ensemble) which compose the application.

However, composing MP applications from reusable components is prone to a number of errors: wrong components, unspecified or incompatible binding of communication channels, etc. These errors may emerge during program execution in the form of undelivered messages, deadlock situations, non-terminating programs, etc. Furthermore, general problems of debugging (e.g. no guarantee of absence of bugs), as well as problems of parallel program debugging (e.g. non-deterministic behaviour of programs, non-reproducibility of behaviour) still apply. To improve the reliability of applications, we would like 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 MP application cannot be, in general, analytically determined from the known behaviour of its components. Nevertheless, 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 on the usefulness of formal methods in software development, we have followed the middle way [8]. In the presence of numerous formal models which all address the same problem, but very few of them are actually used

[17], we do not intend to present another model. We would use already developed formalisms and their associated theory and tools that are suitable for Ensemble as software engineering testing methods. We have used an extension of 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 the specification of components and their composition. We also examine relative work on composition of Petri nets. In section 4, we describe the general form of component specifications and define their composition directed by scripts. In section 5, we apply the composition on example 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 a vehicle the Distribution of Maximum application: There are terminal processes, each of which is given an integer parameter and requires 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 global maximum (GM); they finally send GM to their terminal processes. The Ensemble implementation consists of the application script and the two executable reusable components, terminal and 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:

Figure 1

The first part, headed by PCG, specifies the Process Communication Graph (PCG) of the application, independently of any MPE. PCGs are a natural structure for specifying processes and their communication dependencies and are close to program design [1]. Nodes on a PCG denote processes and arcs denote point-to-point 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, which due to programming complexity, is usually considered in terms of global parameters in an application, e.g. sizes of dimensions of a grid topology. Nevertheless, there may be other local scalability parameters. 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 an application requires replication of processes and their interconnections. For some process topologies, such as a torus, 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 global as well as local scalability of applications, we specify for each process in the Ensemble 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 (C_{in}, C_{out}, P_{in}, P_{out}). All relay processes have two ports of type P_{in} and P_{out}, but different number of ports of C_{in} and C_{out} types. Point-to-point 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.

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 [4]. 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 specifies the annotation of arcs by unique tags).

The third script part, headed by Sequential Components, specifies the further annotation of nodes with process loading information. The executables corresponding to the reusable components are specified and, for each process, the 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.

2.2 The reusable components

The reusable components 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.

Figure 2

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 SetInterface 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

The Launcher is the program that actually composes applications, universal for all applications in the same MPE. There is one Launcher program for each MPE. The Launcher visits the annotated PCG nodes and spawns processes. To each spawned process the launcher provides 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. When the launcher terminates, the complete program is composed and running.

We have only outlined the aspects of Ensemble methodology and its tools that are relevant in the context of this paper. A detailed description of Ensemble in PVM and Parix may be found in [4] and [5], 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 point-to-point 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 interconnections also depends on being synchronous or asynchronous. Although Ensemble supports synchronous communications, we restrict our presentation to asynchronous communications, for reasons of conciseness.

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; occurrence graphs may be produced or simulations may be performed.

We use the Petri net formalism for expressing and composing specifications, as they have a well-founded theory, they have been widely used to specify parallel software systems and are supported by a number of tools. More specifically, we use Coloured Petri nets (CPNs), which allow the modeller to create simple and easily manageable descriptions, without losing the ability of formal analysis [6]. Although, nets of higher level of abstraction (e.g. objects, hierarchies) provide more design possibilities, they are not necessary in our methodology and furthermore, they would restrict the choice of available analysis tools. In any case, analysis can be performed only on completely refined (flat) Petri nets. Petri net semantics have been shown suitable for the composition of specifications of MP applications.

In [11] 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 MP, in contrast to composition based on fusion of transitions, which corresponds to synchronous communication.

In M-nets [2], large High Level nets are constructed from smaller components, by transition synchronisation, which allows composition in a manner similar to process algebras like CCS [13]. Communication is modelled with specific channel constructs [3]. These channels act as a FIFO buffer of size b. To enable concurrent send and receive actions, the contents of a channel are modelled by two sequences, held on two different places. There is one transition for the send action, which appends the communicated value to the channel. There is one transition for the receive action, removing the communicated value from the channel. When b=1, the channel may either be empty, or contain a singleton sequence of values. When, b=0, we have the case of synchronous communication, and the send and receive transitions are merged into a single transition.

A summary on how communication can be modelled with elementary Petri-nets can be found in [7]. There are two variations for synchronous communication, called synchronous and rendezvous respectively. For asynchronous communication there are also two variations, called asynchronous and semi-asynchronous. Also in [7], Heiner studies the association of a “reduced grammar for code statements” to PN constructs, providing PN building blocks for constructs usually met in programs (e.g. for-while loops, if-then-else statements etc).

4 COMPOSITION OF SPECIFICATIONS SUPPORTING ENSEMBLE

In this section, we present the specifications of program components, according to the requirements of the previous section. We introduce the component specifications, named template CPNs, which are CPN components extended with open scalable interfaces. Template CPNs are very close to the notion of pages in [9] and in the design/CPN tool [10]. They are also “flat” structures (pages are non-hierarchical CPNs), and from them process specifications may be instantiated (as a page may have several page instances). The difference lies in the fact that from template CPN to process specifications a structural modification of the net occurs, where the page instances are exact copies of the original page.

4.1 Program component specifications: template CPN

Coloured Petri nets are, in a way, very close to our needs. 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 (fixed number of places, transitions and arcs). 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 template CPNs 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

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*. When a communication type has a fixed number of ports, say N, the expression becomes N..N. For example in template T in Figure 3, S_{in} and S_{out} have a range 1..1. In the case where, there is no upper bound on the number of ports, the notation becomes *from..*, i.e. the value for *to* is unspecified. In Figure 3, R is specified to have an 1.. range for its C_{in} and C_{out} port types and an 0.. range for its P_{in} and P_{out} port types. The special case, where the value of *from* is 0, specifies that the component may not have any ports of this type. Usually the number of ports is required as a parameter in the net. For this reason 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, e.g. [Sout] in template R. The bracket notation may also be used for array variables in the inscriptions on the arcs joining interface places to indicate distinct array elements associated with each port, e.g. [LM] on arc inscription of [Pin] of template R. 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, named composable CPNs, are instantiated from their corresponding template CPN, by providing actual values for instance numbers, number of ports of each type and application parameters, as specified in Ensemble scripts. 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. Occurrences of *PortName.#ports* strings in the 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. The brackets of template interface place names, e.g. [Pout], are removed and unique place names are obtained by uniquely indexing the port name, e.g. Pout[1], Pout[2]. Array names in brackets, e.g. [LM], are also replaced by specific array elements, e.g. LM[1] and LM[2], associated with each port.

Figure 4

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 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, provided they are compatible as explained in section 3.

5 COMPOSING APPLICATION SPECIFICATIONS

In this section, we present three example applications, which are design variations of Distribution of Maximum application, 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 reachability graph of the composed specification will show that there is an invalid marking, where the two interface ports of T[2] are both marked. This indicates that its input value (the global maximum) is already determined before its output value has been consumed and processed by the relay. 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, similar to CPN pages, 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.

Figure 6

We may verify that the behaviour of the composed specification is correct.

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 and the PCG is depicted in Figure 7:

Figure 7

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 8.

Figure 8

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 of Maximum application is also correct.

6 CONCLUSIONS - FUTURE WORK

We presented a specification composition, which improves the reliability of MP applications, composed by the Ensemble methodology. We 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 [11] adapted to the composition of Ensemble applications as described by the script. During composition, static information 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.

Although the composition is mechanical, it cannot be implemented on CPN tools using a graphical representation of nets, as we don't know the internal representation of nets. We are currently developing equivalent grammatical representations of template CPNs and tools for generating composable CPNs and performing their composition by string manipulation. For our future work, we intend to extend template CPNs for synchronous point-to-point communications and to define non-deterministic communication channels. By using timed or stochastic PNs for component specifications, other aspects of application behaviour could be represented and analysed (e.g. real-time latency and throughput).

The correspondence of program and specification composition is depicted in Figure 9. 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 9

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 worlds of specifications and programs. Usually specifications are obtained before program design and program implementation. However, this view is not valid in the software composition approach. Programs and their specification may be composed independently of each other. In Ensemble, both may be composed from scripts. 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 that 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 MP applications.

7 REFERENCES

[1] Andrews, G R, Paradigms for Process Interaction in Distributed Programs, *ACM Computing Surveys* 23(1) (1991) 49-90.

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

[3] Best, E, Fleischhack, H, Fraczak, W, Hopkins, R P, Klaudel, H and Pelz, E, An M-net Semantics of $B(PN)^2$, in *Proc. of Structures in Concurrency Theory*, Workshops in Computing, Springer, 1995, 85-100.

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

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

[6] 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.

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

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

[9] Jensen, K, Coloured Petri Nets: A High Level Language for System Design and Analysis, in: Rozenberg G, ed., *Advances in Petri nets*, LNCS 483, Springer-Verlag, 1990, 342-416.

[10] Jensen, K, *Design/CPN Reference Manual*, Aarhus University-Metasoft, 1996.

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

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

[13] Milner, R, *Communication and Concurrency*, Prentice Hall, 1989.

[14] Nierstrasz, O, Regular Types for Active Objects, in: Nierstrasz O and Tsichritzis D, eds., *Object-Oriented Software Composition*, Prentice Hall, 1995.

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

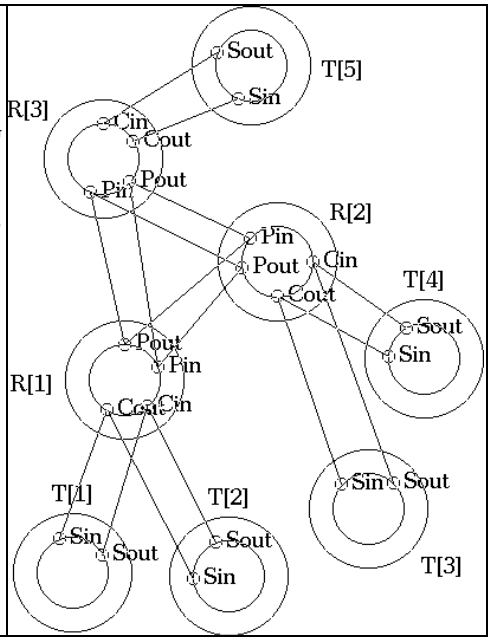
[16] Nierstrasz, O, Gibbs, S and Tsichritzis, D, Component-Oriented Software Development, *Communications of the ACM* 35(9) (1992).

[17] Parnas, D L, Mathematical Methods: What We Need and Don't Need, *IEEE Computer* 29(4) (1996).

[18] Parsytec Computer Gmbh, *Parix 1.9 Manual*, 1993.

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];

PCG Builder



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;	**annotation**	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

1

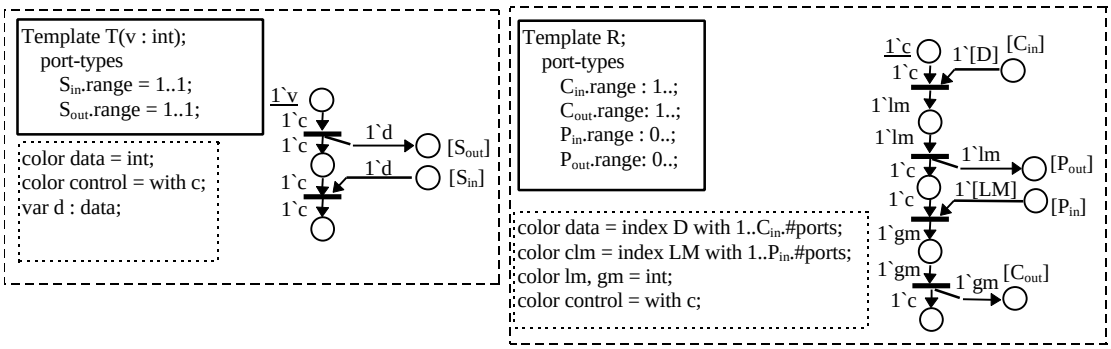
```
void main()                          /* terminal */
{ InterfaceType Interface[2];
  MakePorts(Interface);
  SetInterface(Interface);
  realmain(Interface);               /* main action */  }

void realmain (Interface);
{ /* terminal pseudocode */
    send local value to relay              (to Sout type)
    receive maximum from relay             (from Sin type)
}
```

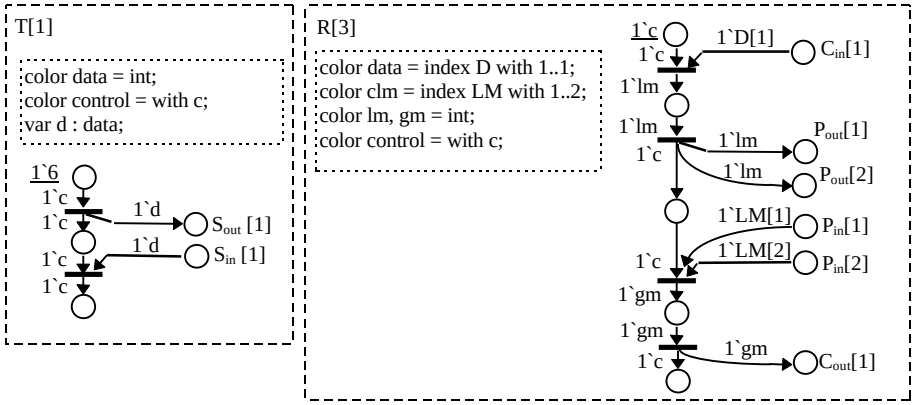
```
void main()                          /* relay */
{ InterfaceType  Interface[4];
  MakePorts(Interface);
  SetInterface(Interface);
  realmain(Interface); }

void realmain(Interface);
{ /* relay pseudocode  */
    receive values from terminals          (from Cin type)
    find the local maximum (LM) of values
    send LM to all other relays            (to Pout type)
    receive LMs from all other relays      (from Pin type)
    find global maximimum (GM)
    send GM to terminals                   (to Cout type)  }
```

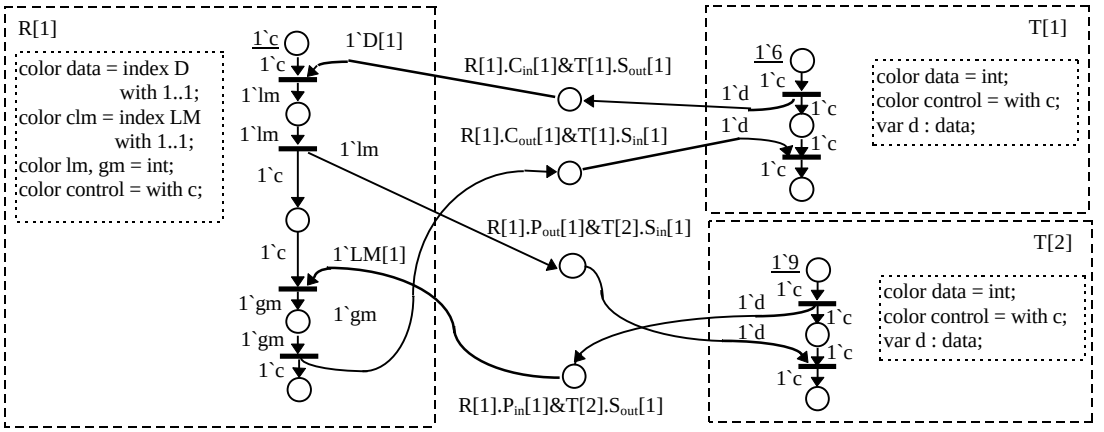
2

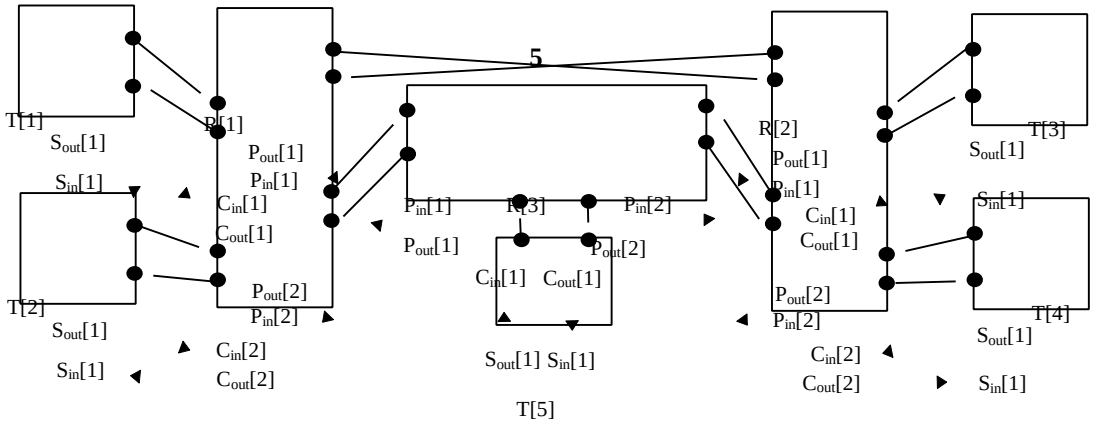


3


T[1]
color data = int;
color control = with c;
var d : data;
1`6
1`c
1`c
1`d
1`d
1`c
1`c
S_{out} [1]
S_{in} [1]
R[3]
color data = index D with 1..1;
color clm = index LM with 1..2;
color lm, gm = int;
color control = with c;
1`c
1`c
1`D[1]
C_{in}[1]
1`lm
1`lm
1`c
1`lm
1`lm
P_{out}[1]
P_{out}[2]
1`LM[1]
1`LM[2]
P_{in}[1]
P_{in}[2]
1`c
1`gm
1`gm
1`c
1`gm
C_{out}[1]

4

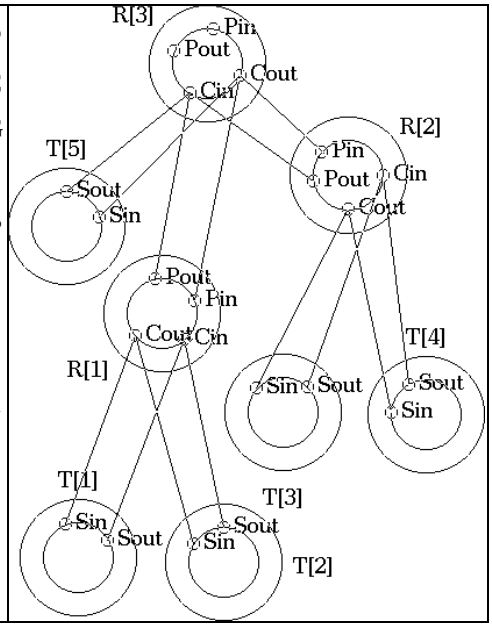

R[1]
color data = index D with 1..1;
color clm = index LM with 1..1;
color lm, gm = int;
color control = with c;
1`c
1`c
1`D[1]
1`lm
1`lm
1`lm
1`c
1`LM[1]
1`gm
1`gm
1`gm
1`c
R[1].C_{in}[1]&T[1].S_{out}[1]
R[1].C_{out}[1]&T[1].S_{in}[1]
R[1].P_{out}[1]&T[2].S_{in}[1]
R[1].P_{in}[1]&T[2].S_{out}[1]
T[1]
color data = int;
color control = with c;
var d : data;
1`6
1`c
1`d
1`d
1`c
1`c
1`c
T[2]
color data = int;
color control = with c;
var d : data;
1`9
1`c
1`d
1`d
1`c
1`c
1`c


5
T[1]
S_{out}[1]
S_{in}[1]
T[2]
S_{out}[1]
S_{in}[1]
R[1]
P_{out}[1]
P_{in}[1]
C_{in}[1]
C_{out}[1]
P_{out}[2]
P_{in}[2]
C_{in}[2]
C_{out}[2]
P_{in}[1]
P_{out}[1]
R[3]
P_{in}[2]
P_{out}[2]
C_{in}[1] C_{out}[1]
S_{out}[1] S_{in}[1]
T[5]
R[2]
P_{out}[1]
P_{in}[1]
C_{in}[1]
C_{out}[1]
P_{out}[2]
P_{in}[2]
C_{in}[2]
C_{out}[2]
T[3]
S_{out}[1]
S_{in}[1]
T[4]
S_{out}[1]
S_{in}[1]

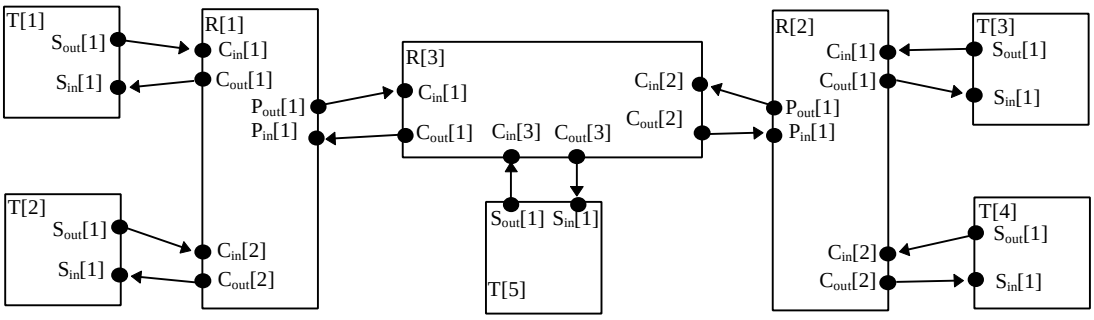
6

APPLICATION Distribution_Maximum_Tree;
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}: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];

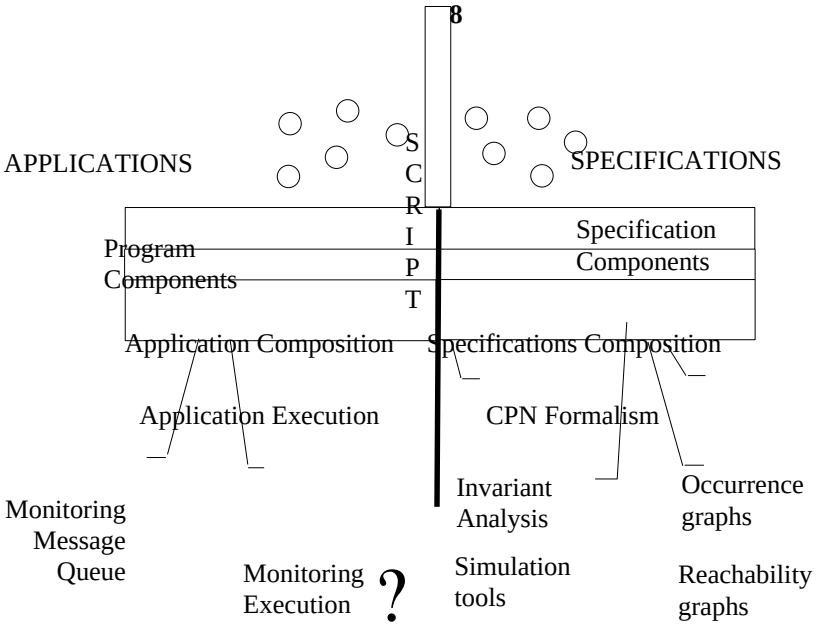
PCG Builder



7



8



9