Proc. of the Seventh Euromicro Workshop on Parallel and Distributed Processing, Funchal, Portugal, 1999, pp. 204-211

Associating Composition of Petri Net Specifications with Application Designs in GRADE

Z. Tsiatsoulis[1], G. Dózsa[2], J.Y. Cotronis[3] and P. Kacsuk[4]

[1,3]Department of Informatics, University of Athens, Panepistimiopolis,
157 71 Athens, Greece, Tel.: +301 7275223, fax: +301 7219561
E-mail: {zack, cotronis}@di.uoa.gr

[2,4] Computer and Automation Research Institute, Hungarian Academy of Sciences
H1518 Budapest, P.O.Box 63. Hungary, Tel/Fax: +361 1297864
E-mail: {dozsa[2], kacsuk[4]}@sztaki.hu

Abstract

To provide high-level graphical support for developing message passing programs, an integrated programming environment (GRADE) is being developed. GRADE currently provides tools to construct, execute, debug, monitor and visualise message-passing based parallel programs. The paper describes the extension of GRADE with formal method support based on Petri nets composition. . We outline specification composition, directly associated with application composition as well as the integration of specification and implementation of program development.

1 Introduction

Parallel Processing has emerged as a means to cope with the computational power needed by the ever-increasing complexity of software applications. The most common paradigm of parallel processing is message passing (MP). Designing MP applications 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. Processes and their interactions (virtual communication channels) are modelled by virtual process topologies. Implementation of MP applications involves programming of sequential processes, as well as latent programming for management of processes and architecture resources. The emergence of Message Passing Environments (MPE), such as PVM [10] and MPI [21], provide a useful abstraction of the underlying architectures simplifying resource management.

Although, different MPEs can simplify several aspects of parallel program development, the lack of real user-friendly software tools for such development prevents many potential users from dealing with concurrent programming at all. To cope with the extra complexity of parallel programs arising due to inter-process communication and synchronisation, we have designed an integrated visual programming environment called GRADE [16]. GRADE stands for Graphical Application Development Environment and its major goal is to provide an easy-to-use, integrated set of programming tools for development of general message-passing applications that can run either on supercomputers or on heterogeneous workstation clusters. GRADE has the following main benefits:

?? Visual interface to define all parallel activities in the application (i.e. all process management and communication actions). Graphics help in better understanding the complex structure and run-time behaviour of the distributed program even for users not familiar with parallel programming.

?? Programmers are not required to know the syntax of the underlying message-passing system. GRADE generates all message-passing library calls automatically on the basis of the visual code. As a

result, GRAPNEL programs are portable between different MPEs provided that GRADE is able to generate code for those MP systems. Currently, GRADE can generate code only for PVM but MPI is to be supported very soon.

?? Compilation and distribution of the executables are performed automatically in the heterogeneous environment.

?? Debugging and monitoring information is related directly back to the user's graphical code during on-line debugging and post-mortem visualisation of the run-time behaviour of the application.

GRADE provides the GRED graphical editor for the programmer to construct the code of his/her parallel application according to the syntax and semantics of GRAPNEL language [17]. Furthermore, the environment offers integrated tools for correctness and performance debugging of the GRAPNEL applications which use the same graphical representation of the parallel program as GRED does. (The distributed debugger integrated into GRADE is called DDBG [8] and it has been developed at University of Lisbon). As a result, the programmer has the same high-level visual view of his/her application during the whole program development cycle. A parallel architecture simulator developed at University of Barcelona has also been integrated into an early version of the system and it is to be adopted soon for the current version. Possible co-operation between GRADE and EDPEPPS [9] environments is currently under investigation to extend the simulation capabilities of the system.

The GRED editor has originally supported only the top-down design of distributed applications, i.e. the programmer had to start his/her work by designing the topology of the program followed by elaborating more and more detailed codes of the processes. The visual code of the whole application could only be saved as one unit. However, this approach had the disadvantage that GRAPNEL code of processes could not be reused through different application. To overcome this problem, we have decided to follow the concept of re-usable program components similarly to that of Ensemble methodology [1][4][5]. In this way, the programmer can create and use processes individually as reusable program components which can be embedded into various applications using different inter connection schemes.

Although, GRADE provides a productive framework for implementing and maintaining MP applications, it cannot guarantee absence of design errors. 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, analyse or ideally verify the correct behaviour of the composed applications. To this end, we integrate into GRADE a specification composition technique [6][7][25], which is directly associated with program composition. We produce formal specifications of program components directly from GRADE's process graphical components. We then generate formal process specifications and compose them, directed by GRADE's application topologies, deriving formal specifications of applications. The topologies, which direct composition of applications, also direct the composition of formal specifications. We have used the Petri-net (PN) formalism [14] 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 direct association of formal specifications and implementations may improve the use of formal methods in the software engineering process, but only marginally. However, this direct association provides the basis for testing and debugging applications supported by formal methods, using specification and execution tools 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 section, we outline program composition in GRADE and discuss the requirements for associating specification composition with application composition. In section 3, we describe PN component specifications and present their composition followed by some conclusions.

2 Program composition in GRADE and specification requirements

In GRAPNEL programs there are three different design levels distinguished. Process topology related information (i.e. communication paths among processes) are defined at the application level as a graph (see Fig. 1). Nodes of this graph are processes and their codes are defined by graphical symbols at the process level of the program (see Fig. 2,3). However, not all instructions are described as individual icon. The point is that every send and receive operations must be defined graphically but arbitrary large and sophisticated code segments containing no send or receive operations are

represented by a single text block icon. The contents of graphical symbols can be defined as ordinary text program code at the lowest level of the program. The actual contents depend on the particular symbol. For example, in case of a receive icon the code must be simple the name of the program variable(s) where the data to be received are to be stored while, a text block icon may contain text code without any restrictions except the lack of inter-process communication. Detailed description of the GRED editor and that of different elements of the GRAPNEL language can be found in [18] and [17], respectively.

We briefly outline the design and implementation of applications in GRADE on an example 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 1) its integer parameter to its Server and, eventually, receives (via port 0) the required maximum. Servers receive integer values from their client Selectors and find the local maximum. Servers are connected in a ring. They find the global maximum by sending their current maximum to their next neighbour in the ring, receiving 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 the global maximum to their client Selector processes.

2.1 Process topologies in GRADE

Figure 1 shows the Application window of the graphical editor GRED in which the process topology of this simple application is defined. We have three Server processes (named "Serv[1]", "Serv[2]" and "Serv[3]"). There can be seen six Selectors around the Servers. Three of them connected to "Serv[3]" (named "Selector[4]", "Selector[5]" and "Selector[6]"), two of them connected to "Serv[2]" (named "Selector[2]" and "Selector[3]") and only one of them connected to "Serv[1]". The application is executed directly from this environment. At the end each selector process will have acquired the maximum of all integers initially stored in the selector processes. The topology shown can easily be scaled by adding new selector or server processes, and connecting them to the existing processes. This is performed with some simple mouse clicks and drag and drop sequences. The result of this simplicity is that the programmer can easily and quickly design and test a variety of topologies and different behaviours imposed by them.

2.2 The Reusable Components

Process icons shown in Figure 1 represent reusable program components as they can be inserted into various applications with different connection schemes. We use only two components in our example: Server and Selector. Figure 2 and 3 depict the Process window of the GRED editor in which the visual code of components Server and Selector are

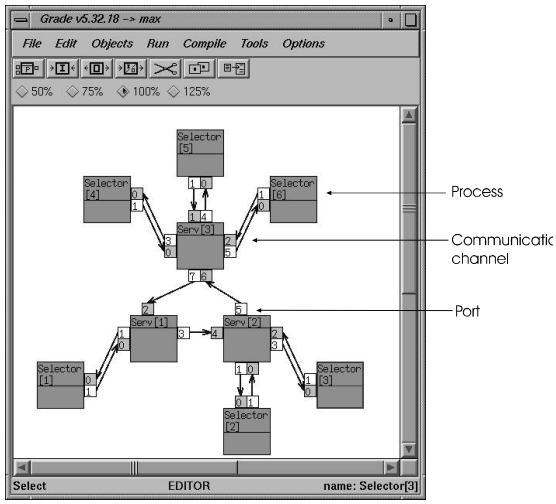

Figure1: Process topology of the Get Maximum example program

defined, respectively.

The GRADE 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 specify local ports for point-to-point communication with any compatible port of any process in any application. All send and receive operations in processes are defined graphically (as boxes) and they refer to these local ports represented also by different boxes. Port symbols available inside a component are listed at left edge of the process window of the GRED editor and the user can assign them to various send or receive icons by a single mouse click. Each port icon has its own protocol (defined as text code belonging to that icon) and a port index for identification purpose.

2.3 Program Generation in GRADE

The GRED editor creates the so-called GRP files from the GRAPNEL program. Topology and components related information are separated: there is always one individual GRP file containing the

description of the process topology while a separate GRP file is created to contain the code of each component.

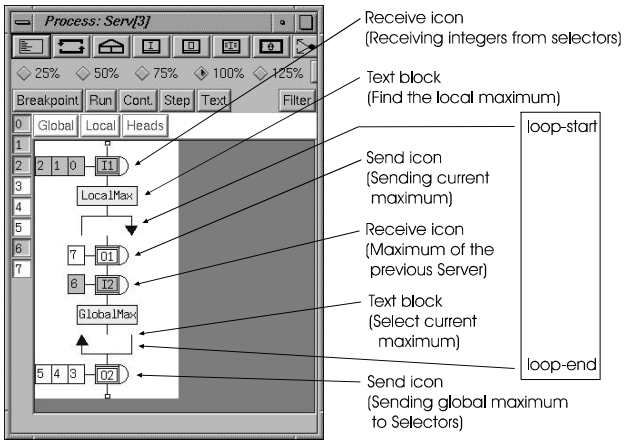

Figure2: Visual code of the Server component

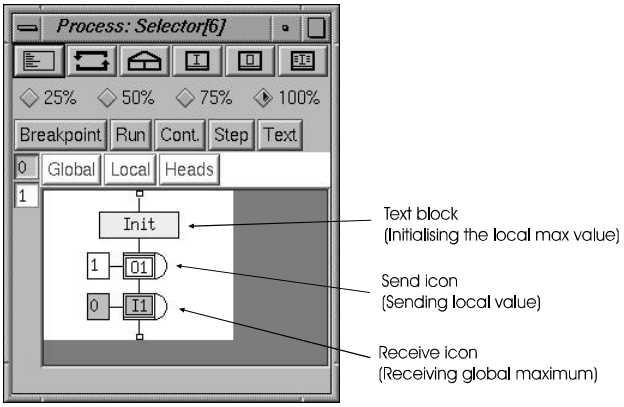

Figure 3: GRAPNEL code of the Selector component

Components can also be saved to and retrieved from a GRP file individually. GRP files contain all the information necessary to restore the program graph for further editing and to compile the GRAPNEL program into a C+PVM code. The latter is the task of the GRP2C pre-compiler, which additionally creates other auxiliary files including makefiles used by the UNIX 'make' utility for building the executables. C code distribution and compilation is fully automated on every host of the heterogeneous cluster of workstations. Having obtained the executables, the parallel program can be executed either in normal, debugging or trace mode. In debugging mode, the DDBG distributed debugger controls the execution of the program by providing commands to create breakpoints, step-by-step execution, animation, etc. In trace mode, a trace file is generated containing all the trace events defined by the user. These events are visualised by the PROVE graphical visualisation tool assisting the user in spotting performance bottlenecks in the GRAPNEL programs.

2.4 Requirements for specifications and their composition in GRADE

The essential idea is to compose programs and specifications directed by the same GRAPNEL code. To reflect the GRAPNEL architecture of parallel applications we need to define reusable specification components, process specifications (instantiations of specification components) and their composition, corresponding to reusable program components, processes and the composed application (i.e. process topology), respectively. We should also test and validate each step of the composition.

3 Composition of specifications supporting GRADE

We have adopted 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 suitable for the composition of specifications of message passing applications [19]. In particular, we use Colored Petri nets (CPNs) which allow the modeler to create simple and easily manageable descriptions, without losing the ability of formal analysis [14]. Furthermore, CPNs 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 [15], PEP [12], LOOPN [20], SYROCO [24]and others.

3.1 Composition by place unification

Heiner in [13] studies the association of the metanotions of a "reduced grammar for code statements" to PN constructs. We will use these associations in our specification components. As an example, figure 4 depicts the CPN Server component, satisfying the requirements posed in section 2.

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 4, transitions GatherfromClients and MultiCastToClients, model receive and, respectively, send operations from and to all client processes.

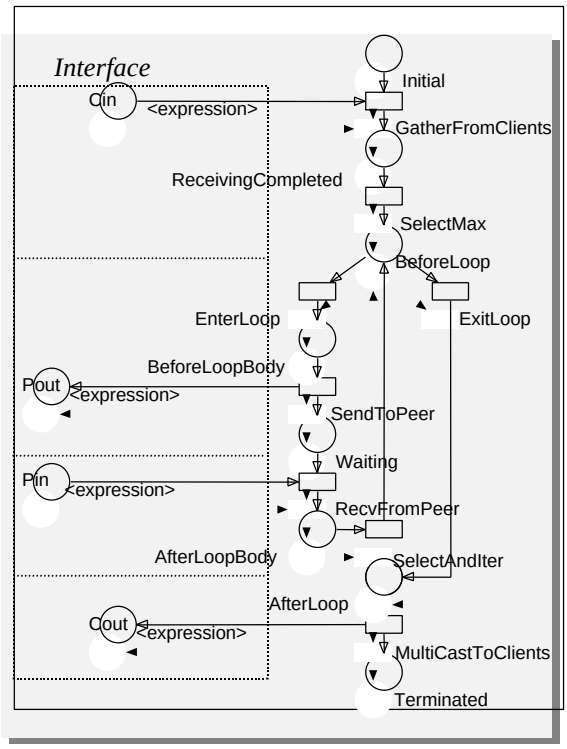

Figure 4: The CPN for the Server component

In this paper, we follow the approach in [25] modelling ranges of ports within a communication type by maintaining a single interface place and replicating inscriptions on the connecting arc, as depicted on figure 6. 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.

The unified environment place resembles modeling the tuple space of Linda [11]. 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.

In the sequel, we define specification components and the specification composition.

3.2 Specification Components: template CPN

A specification can be modeled directly with CPNs when the interface of a component is fixed, as for example in the Selector component (it has one port of types In and Out), Parametric interfaces cannot be directly modelled using CPNs. We extend CPNs by template CPNs, which contain additional information to specify open scalable interfaces parametrically.

Template CPNs are very close to the notion of pages in [14] and in the design/CPN tool [15]. They are also "*flat*" structures (pages are non-hierarchical CPNs). The template CPN is a parametric net-structure having a unique name, from which process specifications, called composable CPNs, may be instantiated (as a page having several page instances). Instantiation of composable CPNs from templates involves structural modification of the net, whilst the page instances are exact copies of the original page. We identify composable CPNs by unique indexing of template names.

We have defined a linguistic form for expressing template CPNs, because their mechanical composition is more manageable than that of graphical objects. Furthermore, by defining CPN templates closely to the new emerging standard, we remain independent of any actual Petri net tool, but also guarantee the possibility to use any of these tools. In figure 5 we give the textual description of template Selector of figure 3 as well as its graphical equivalent.

The template of each component is generated from the respective component GRP file. A *template generator* parses the GRP file, recognises the correspondence between GRAPNEL semantics and CPN structure (according to the associations in [13]), and produces the template CPN for this component. These correspondences are quite clear to identify since GRAPNEL mainly describes the communicational behaviour of each component. Finally the template is stored in a *template repository* where it will be later accessed during the specifications composition process.

3.3 Composition of Specifications

We now outline the composition of specifications. Composition is performed in four steps.

Step 1: Retrieve template CPNs. For each component in the application, we retrieve, from the template repository, the corresponding template CPN.

Step 2: Create composable CPNs. For each process in the application, we create the corresponding composable CPN. We check the validity of port interface parameters. Actual values

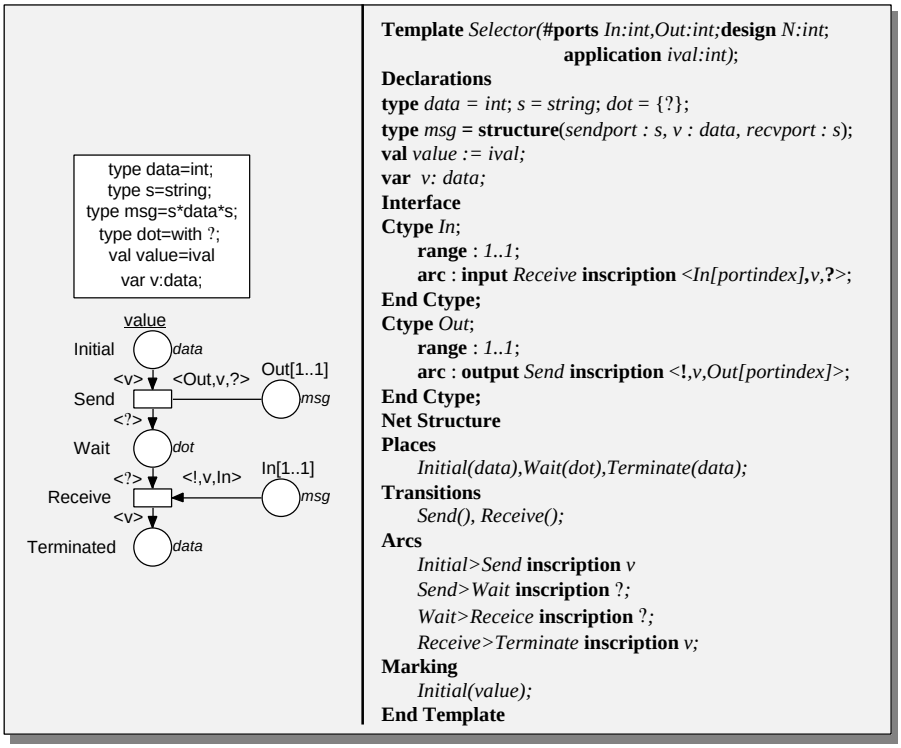

Figure 5: The template CPN of selector component in textual and graphical form

for application parameters are provided. If the interface is valid, we create process specifications. For each communication type, the actual number of ports is created.

Step 3: Merge composable CPNs. The individual composable CPNs will be merged (i.e. composed) into a single CPN. The composed CPN is constructed by merging the descriptions of composable CPNs into one, according to the application description. For example, the composed CPN, corresponding to application of figure 1 is illustrated in figure 6 using the environment unification place approach. 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.

Step 4: Validate composition. We check if all ports are actually connected.

The reader can easily observe the correspondences between figure 6 and the application graph as this is designed in GRADE (figures 1, 2, 3). A difference is the way interface ports are indexed. GRADE uses a global indexing schema for all ports of a single component, irrespective of its communication type (I1, O2 etc.), whereas in the CPN the indexing depends on the context. Thus port 0 of I1 in Server[3] corresponds to port I1[1] in server's specification, port 4 of O2 is O2[2] and so on.

6 Related works

A number of other visual programming environments have been developed for parallel applications (e.g. TRAPPER [23], EDPEPPS [9], HENCE [2], CODE [22]).

Most of them (e.g. HENCE, CODE) are based upon the idea that nodes represent parallel computation and arcs represent interactions (of some form) among nodes. The problem with the HENCE and CODE approaches is that they force computations to be split into separate processes when communications occur or when branching decisions control communications (i.e. some kind of data-flow approach). This can result in complicated, awkward and large process communication graphs.

This problem does not arise in TRAPPER and EDPEPPS which systems are very close to GRADE concerning both purpose and functionality. However, GRADE provides visual instructions to define communication operations while the others apply graphics only to define the process communication graph.

Furthermore, none of the above mentioned tools tries to provide formal method support to aid the complex process of parallel program development.

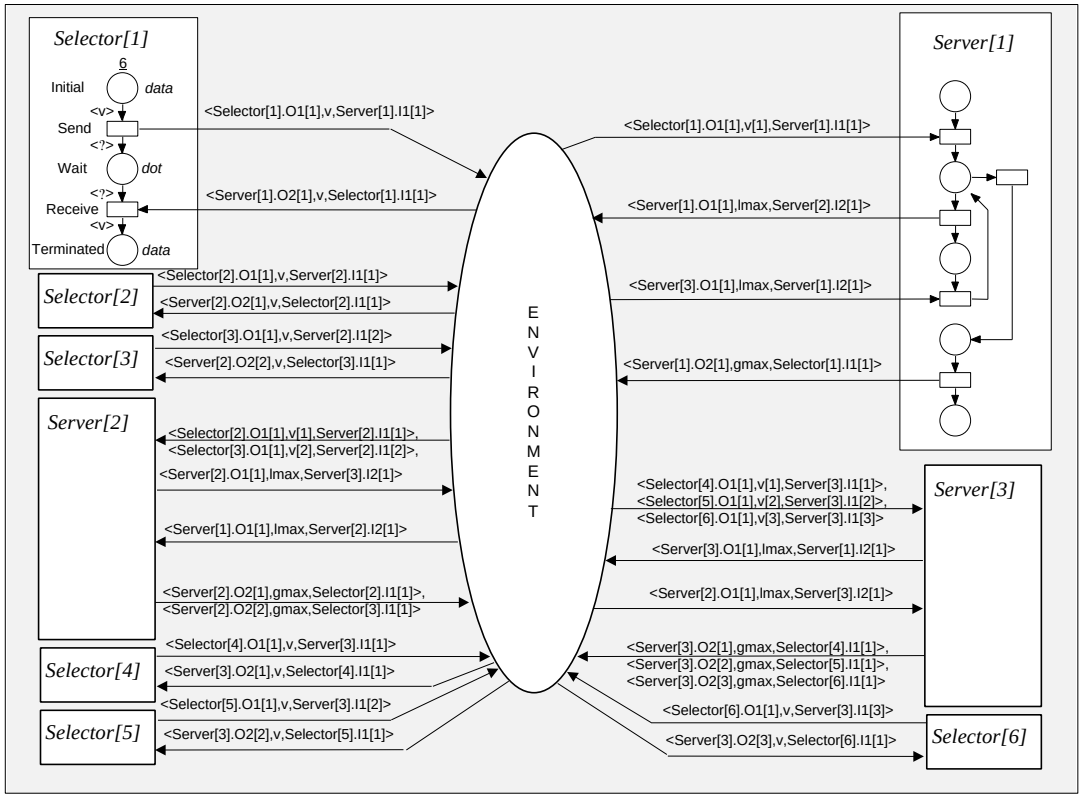

Figure 6: The composed CPN of Get Maximum Selector Servers in ring

7 Conclusions-Future Work

In this paper we propose the integration of formal methods with software engineering methods, in order to improve testing and debugging of parallel applications. Due to the lack of space, works related to the GRADE environment and composition of Petri nets are omitted here but they are discussed in [18] and [25], respectively. The proposed methodology takes advantage of the direct association of specification and program components with GRADE components. They are both composed from reusable components and their composition is directed by the Topology part of GRADE.

We have extended GRADE to permit the composition of message passing programs from generic process components. We have defined Petri net specification components directly associated with the process components. We have also defined the composition of specification components to derive the specification of the complete application. We may first test and verify that programs design are correct, and only then run parallel programs using valuable resources. Specifications cannot catch all errors, but they are useful anyway to identify the real cause of errors.

We intend to further extend our methodology for performance evaluation of the MP applications. Our long-term aim is to create an integrated software engineering support tool for the development of reliable message passing applications.

8 Acknowledgements

The work described in this paper is funded by the Hungarian Technological Development Committee (OMFB) in the framework of Hungarian-Greek TéT Project GR-25/96.and partly by the Hungarian National Science Research Fund (OTKA) Contract Num: F022105. It is also partly funded by the Greek Ministry of Development, General Secretariat of Research and Technology.

9 Bibliography

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

[2] Beguelin, A., Dongarra, J., Geist, G. and Sunderam, V. (1993) Visualization and debugging in a heterogeneous environment, *IEEE Computer*, Vol. **26(6)**.

[3] Cotronis, J.Y. (1996) Efficient Composition and Automatic Initialization of Arbitrarily Structured PVM Programs, in *Proc. of 1st IFIP International*

Workshop on Parallel and Distributed Software Engineering, Berlin, 74-85, Chapman & Hall.

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

[5] Cotronis, J.Y. (1997) Message Passing Program Development by Ensemble, in *Proc. of PVM/MPI'97*, Cracow, LNCS **1332**, 242-249, Springer.

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

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

[8] Cunha, J.C., Lourenco, J., and Antao, T. (1996) A Debugging Engine for Parallel and Distributed Environment, in *Proc. Of 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems*, Miskolc, Hungary, 111-118.

[9] Delaitre, T., Ribeiro-Justo, G., Spies, F. and Winter S. (1997) A graphical toolset for simulation modelling of parallel systems, *Parallel Computing*, **22(13)**, 1823-1836.

[10] 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.

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

[12] Grahlmann, B. (1997) The PEP Tool, in Proc. of Application and Theory of Petri Nets'97, Toulouse, LNCS 1248, Springer.

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

[14] Jensen, K. (1990) Coloured Petri Nets: A High Level Language for System Design and Analysis, in Advances in Petri nets 1990, LNCS 483, 342-416, Springer.

[15] Jensen, K. (1996) Design/CPN Reference Manual, Aarhus University-Metasoft.

[16] Kacsuk, P., Cunha, J.C., Dózsa, G., Lourenco, J., Fadgyas, T. and Antao T. (1997) A Graphical Development and Debugging Environment for Parallel Programs, *Parallel Computing*, **22**,1747-1770.

[17] Kacsuk, P., Dózsa, G. and Fadgyas, T. (1996) Designing Parallel Programs by the Graphical Language GRAPNEL, *Microprocessing and Microprogramming*, **41**, 625-643.

[18] Kacsuk, P., Dózsa, G., Fadgyas, T. and Lovas, R. (1998) The GRED Graphical Editor for the GRADE Parallel Program Development Environment, in *Proc. of HPCN98, International Conference on High-Performance Computing and Networking*, Amsterdam, The Netherlands, 728-737.

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

[20] Lakos, C.A. (1996) LOOPN++ User Manual, University of Tasmania, Department of Computer Science.

[21] Message Passing Interface Forum (1994) MPI: A Message Passing Interface Standard.

[22] Newton, P. and Browne, J. (1992) The code 2.0 graphical parallel programming language, *in Proc. of ACM International Conference on Supercomputing*.

[23] Scheidler, C. and Schafers, L. (1993) Trapper: A graphical programming environment for industrial high-performance applications, *in Proc. of PARLE'93: Parallel Architectures and Languages Europe*, Munich, Germany.

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

[25] Tsiatsoulis, Z. and Cotronis, J.Y. (1997) Associating Composition of Petri Net Specifications with Composition of Message Passing Applications, Report, Available from URL http://www.di.uoa.gr/~ensemble.