Proceedings Dapsys 98, Universitat Wien, pp. 39-46

Integration of Formal Specifications into GRADE

J.Y. Cotronis[1], P. Kacsuk[2], Z. Tsiatsoulis[3], G. Dózsa[4], E. Floros[5]

[1,3,5] Department of Informatics, Univ. of Athens, Panepistimiopolis, 157 71 Athens, Greece
Tel.: +301 7275223, fax: +301 7219561, E-mail: {cotronis[1], zack[3], grad0162[5] }@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: {kacsuk[2] , dozsa[4]}@sztaki.hu

Abstract **- We present GRADE, a development methodology for message passing (MP) applications extended with formal specifications. We present Grade, as a design and implementation tool for composing MP applications from program components. We also outline specification composition, directly associated with application composition. We outline the integration of specification and implementation of program development.**

Keywords: Message-Passing, Development Methodology, Program Composition, Specification Composition, Testing, Debugging, Formal Methods, Petri Nets.

I INTRODUCTION

Development of parallel applications is, to a large extend, an empirical process. The majority of research efforts in the field of parallel processing are focused on the advances in architectures and algorithms, languages, operating system interfaces or new application areas of parallel systems [12]. Software engineering aspects (e.g. processes, methods, tools) of parallel systems have been neglected. Tools and environments support later stages of development.

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 are usually modelled by virtual process topologies. The software-engineering step from design to implementation is also a complex task, as it does not only involve programming of sequential processes, but also latent programming for management of process topologies and architecture resources. Message Passing Environments (MPE), such as PVM [10] and MPI [19] provide a useful abstraction of underlying architectures, simplifying architecture 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 [15]. 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 helps in better understanding the parallel structure of the code.

?? 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.

?? 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 trace file.

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 [16]. 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 icons. 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 simply 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 information about the graphical editor and the different elements of the GRAPNEL language are available in [17][16].

In addition to the graphical editor, the GRADE environment provides integrated tools for correctness and performance debugging of the GRAPNEL that also use the above described graphical representation of the parallel program applications (the distributed debugger integrated into the environment is called DDBG [7] and it has been developed at the 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. Furthermore, a parallel architecture simulator developed at the University of Barcelona [20] 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 [8] environments is currently under investigation to extend the simulation capabilities of the system.

GRADE 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 code of the whole application could only be saved as one unit. However, this approach had the disadvantage that visual code of processes could not be reused through different application although, GRADE supports deliberately the re-use of sequential text code segments. To overcome this problem, we have decided to follow the concept of re-usable program components similarly to that of Ensemble methodology [2][3][4]. 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 the 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 [5][6][21], 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 two sections, we outline program and specification composition in GRADE. In section 4, we propose a development methodology in the integrated environment. Finally, we present our conclusions and plans for future work.

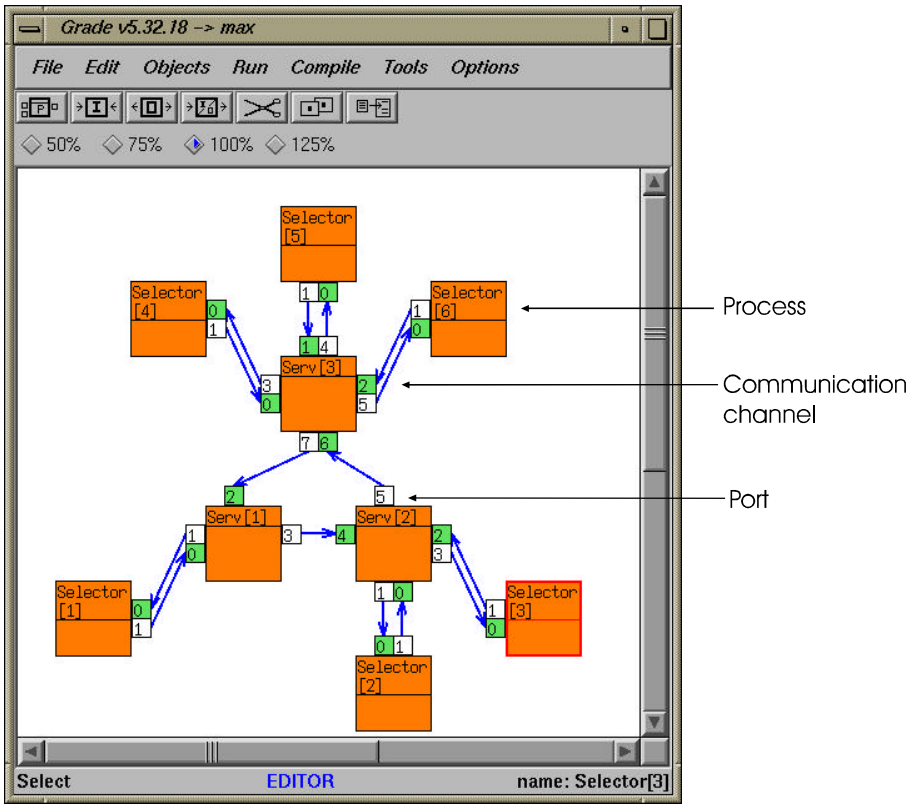
II Program Composition in Grade

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.

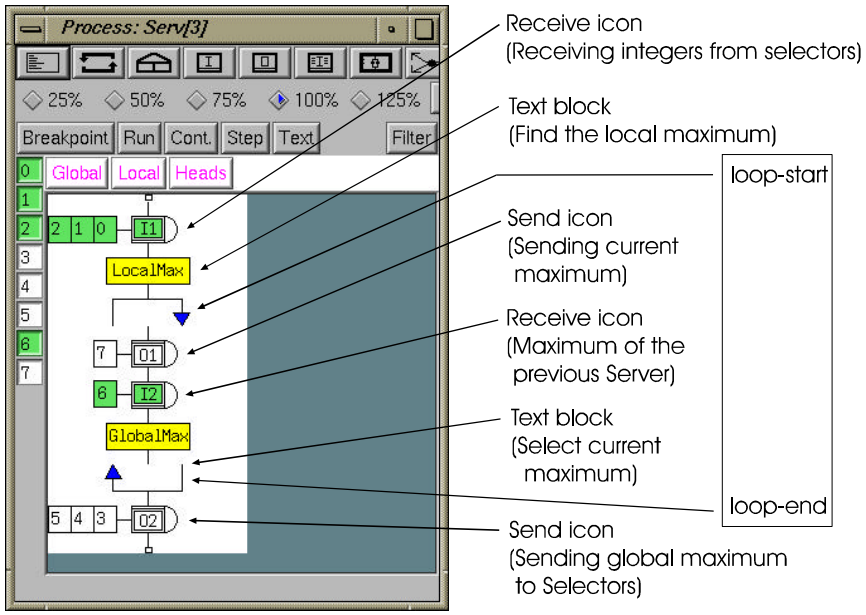
II.A Grade and Topologies

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.



I.A.1 Figure1: Process topology of the Get Maximum application

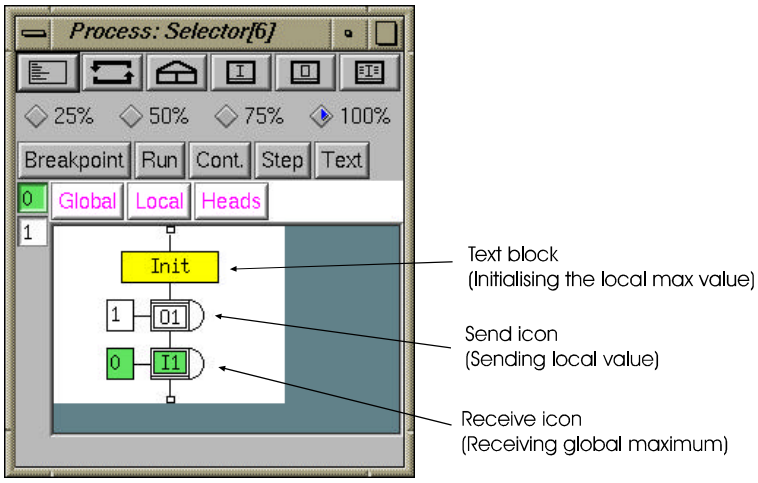
The GRADE components compute a result or provide a service and do not involve any process management or assume any topology in which they operate. Instead, they specify local ports for point-to-point communication with any compatible port of any process in any application. All send and receive operations in processes refer to these local ports, identified by a protocol and a port index.



? Figure2: Visual code of the Server component

II.B The reusable components in GRADE

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 defined, respectively.



I.A.2 Figure 3: GRAPNEL code of the Selector component

III Specifications and Composition

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

III.A Generation of Specification Components

Specification components are directly obtainable from graphical GRADE components in GRP files. Thus, we do not only generate PVM+C programs but also formal specifications and, in particular, Petri-Net specifications. The specification components reflect the level of abstraction of Grade components. Their communication interface is represented in detail. If we are interested only in the behaviour of the application, that is related to its parallel nature, there is no need to use a detailed representation for the internal sequential actions (computations), expressed in C code. The highest abstraction level that can be used is the explicit modelling of computations involving communication operations, represented graphically in Grade. All other operations may be represented "*abstractly*", since their participation in the behaviour of the application is restricted in the internal behaviour of the components. If more low level representations are required, we may use the association of all usual program constructs, such as for-loops, if-then-else statements, to specific PN constructs [13].

Specification components are themselves reusable, permitting the generation of process specifications, as required by the application design. Specification components have scalable interfaces, specifying the valid range of ports for each of their communication types, i.e. always fixed (as I1 of Selectors), or any positive integer (as I1 of Servers), or any non-negative integer (as O1 of Servers). They identify their input and output ports, the type of data that are sent and received through them as well as any application parameters.

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

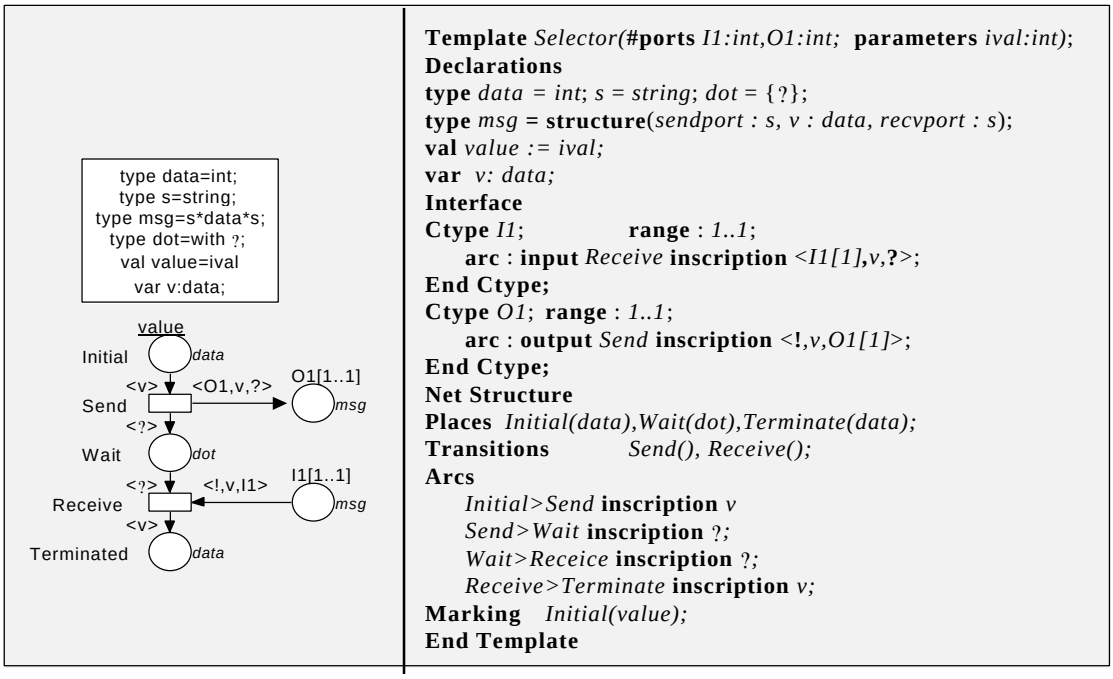
Specification composition is based on modelling point-to-point communication by binding interface ports. During composition, we check the compatibility of ports: binding output to input ports passing messages of the same type.

III.B Specification Components: template CPN

We have used the Petri net (PN) formalism 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 for validation, simulation, analysis and verification. We have studied two different approaches to composition of PN: the place fusion approach [5][6], that models each communication channel explicitly and the environment place approach [21], that models communication channels implicitly by coupling arc inscriptions.

A specification can be modelled directly with CPNs when the interface of a component is fixed, as for example in the Selector component (it has one port of types I and O), 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]. 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 the 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 4 we give the textual description of template Selector of figure 3 as well as its graphical equivalent.



I.A.3 Figure 4: The template CPN of selector component in textual and graphical form

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.

III.C Composition of Specifications

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

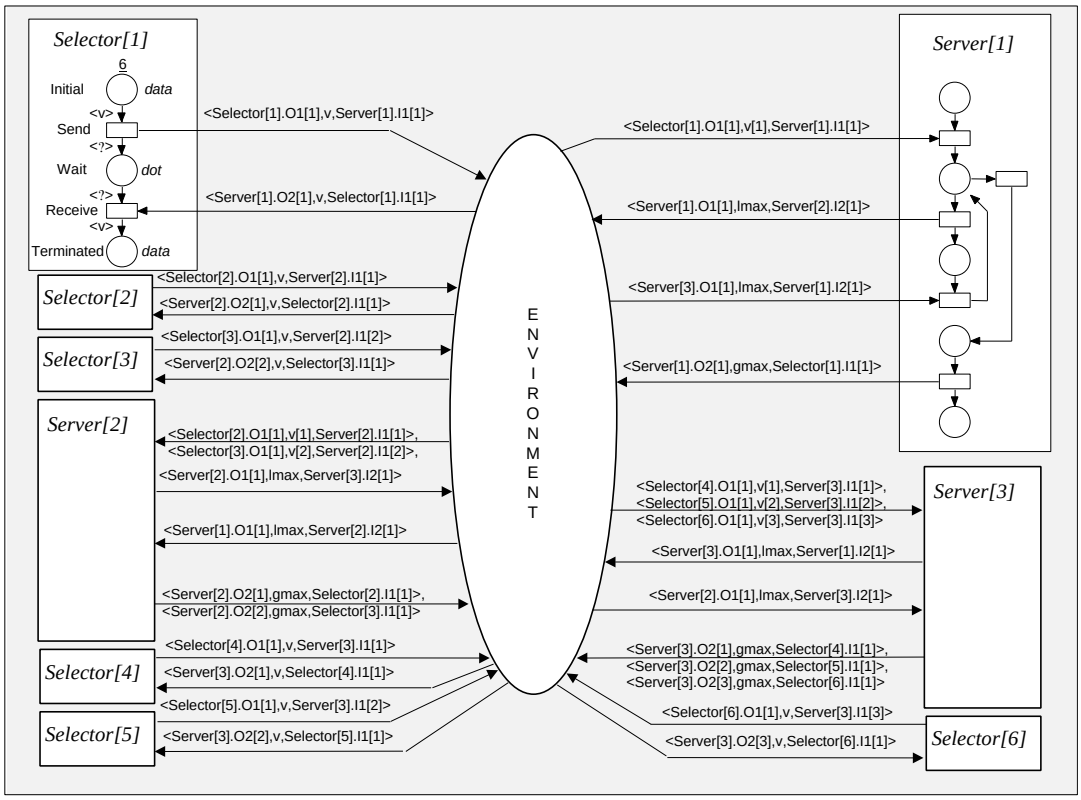
Step 1: Retrieve template CPNs. For each component in the application, we retrieve, from the template repository, the corresponding template CPN.

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 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 4 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 5 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.



I.A.4 Figure 5: The composed CPN of Get Maximum Selector Servers in ring

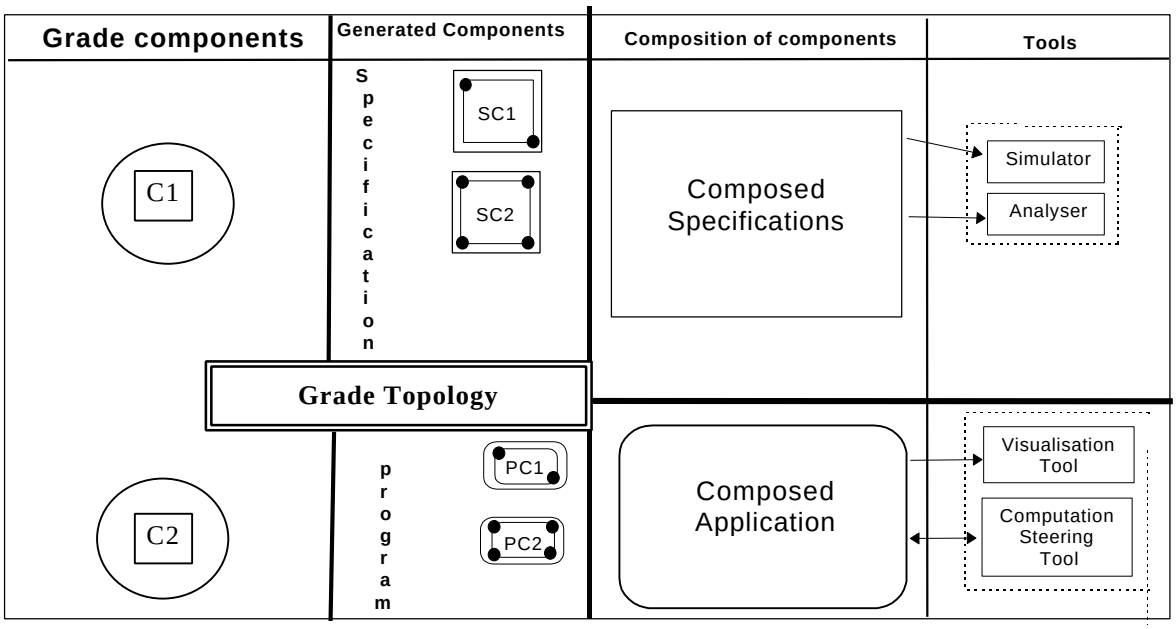
IV PROGRAM DEVELOPMENT METHODOLOGY

Grade supports the design and implementation phases of the software development life cycle. Gorton and Jelly present in [12] a number of challenges that must be addressed by a distributed systems designer. The architecture of GRADE deals with a number of these challenges: scalability requirements, inter-component communications, design validation, choosing synchronisation and MP mechanisms, portability constraints. In this section, we propose the integration of formal methods and execution environment which will cover challenges that apply to the testing and debugging phase of software development life-cycle. An overview of the framework of the proposed methodology is depicted in figure 6.

The leftmost part of figure 6, represents Grade application components, in this case C1 and C2. From these components we produce Petri-net specification components, SC1 and SC2, as well as program components, PC1 and PC2. We use the application topology designed in Grade to compose specifications of applications, as well as programs. The composed specifications may be validated by Petri-net tools using simulation or analysis. The composed programs may be executed on a parallel environment and by using tools, such as visualisation or computation steering, may inspect their behaviour.

The composed application may be tested to detect errors related to execution. The monitoring and visualisation tools of GRADE provide the developer with information of process interactions, message queues, tracing and replay mechanisms, etc. These tools, though, do not guide the developer to find errors but only provide tracing information requested by the programmer. The direct association of program and specification composition is a foundation for validating the design and implementation of applications. We propose the following testing strategy.

Having designed an application we generate specifications for the individual components and compose the specification of the application. We validate the specifications, either by formal analysis tools or simulations. Having corrected any design errors, we generate and execute the associated program. If an error occurs, we first check if the error can be detected in the specifications by “simulating” the behaviour of the program. If the error is observed in the specification, it means that we did not catch it in the previous step and that we have to modify the initial design and repeat step 1. We note however, that by simulating specifications we catch the first erroneous event that really occurred, which is not necessarily the initial execution error we observed. If the error cannot be detected in the specifications, then we attributed it to execution environment factors. In order to correct an execution environment error, we may have to modify the design, some individual components, or other parts of the design.



I.A.5 Figure 6: Overview of the application development methodology.

Let us exemplify the three cases of environment errors. Consider the situation where process P1 issues an asynchronous *send* of a long message to process P2 over a buffer smaller than the message. In some MPI implementations the system automatically switches to synchronous mode, in order to send the complete long message in smaller packets. This auto conversion may result into a deadlock, that could not be detected in any of the previous stages. Consider for example process P2 issuing a *send* to P1 before it receives the message form P1. In this stage, the specification analysis will show that the design is valid, the implementation monitoring will depict the specific problem, and the programmer may immediately detect that the error is due to this automatic conversion of the communication mode. Thus, the design of the application needs to be modified to use synchronous instead of asynchronous communication. We may also change the specifications to reflect the auto conversion of send thus, catching this behaviour in the specifications.

An error that requires modification of a program component is an overflow of some data type, e.g. an assumed 64-bit integer implementation instead of an actual 32-bit one. Correction involves the variable definition to be modified from integer to long. An example of another type error is the case where a specified host machine is inoperable. The designer must modify process allocation and re-run the program.

V Conclusions

In this paper we propose the integration of formal methods with software engineering methods, in order to improve testing and debugging of parallel applications. 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 after that run parallel programs using valuable resources. Specifications cannot catch all errors but they are useful anyway to identify the real cause of errors.

VI 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.

References

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

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[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] Geist, J.A., Kohl, J.A. and Papadopoulos, P.M. (1996) CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications.

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

[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] 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.

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

[17] 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 Netehrlands, 728-737.

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

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

[20] Suppi R. et al. (1997) Simulation in Parallel Software Design, in *Proc. Of Euro-PDS'97,* Barcelona, Spain, 51-59.

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