Proc. of the Seventh Euromicro Workshop on Parallel and Distributed Processing, Funchal, Portugal, 1999,
IEEE Computer Society press, pp. 196-203

Testing and Debugging Message Passing Applications Based on the Synergy of Program and Specification Executions

Z. Tsiatsoulis, J.Y. Cotronis and E. Floros
Department of Informatics, University of Athens, Panepistimiopolis, 157 84 Athens, Greece
Tel.: +301 7275223, fax: +301 7275214, E-mail: {zack, cotronis, grad0162} @di.uoa.gr

Abstract

We outline Ensemble, a design and implementation methodology for composing message passing (MP) applications from program components. We also outline specification composition, directly associated with application composition. We present the integration of specification and implementation of program development. We particularly elaborate on testing and debugging of MP applications based on the synergy of tools for specification simulations with tools for program execution visualisation.

1 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 [10]. Software engineering aspects (e.g. processes, methods, tools) of parallel systems has been neglected. Tools and environments support later stages of development. Designing message passing (MP) applications, in particular, is a complex task and involves designing the combined behaviour of its processes, that is, their individual execution restricted by their communication and synchronisation interactions. Message Passing Environments (MPE), such as PVM [8] and MPI [14] provide a useful abstraction of underlying architectures, simplifying architecture resource management. The software-engineering step, however, from design to implementation is also a complex task, as it does not only involve programming of sequential processes, but also latent programming for management of process topologies and architecture resources.

The Ensemble methodology [2,3,4] was introduced for alleviating implementation complexities due to process management. Each MPE supports its own process model and consequently process topology management does not depend only on the design, but also on the target MPE. Implementation of the same design on distinct MPEs requires different techniques and the porting of applications from one MPE to another requires special effort. The differences in process management impose different structures to programs, to such a degree, that programs implementing the same design (e.g. a simple ring topology), using the same sequential language on two MPEs often look very different. Differences in communication operations cause only local syntactic variations. Furthermore, MPEs favour the management of regular process topologies (e.g. pipeline, ring, grid, torus) and some specific types of process topologies, those being closer to its process management model (e.g. PVM and tree-like process topologies). Designs of irregular process topologies are much more difficult to implement. Scaling is restricted to regular topologies by globally parameterising process position functions. As there are no implementation guidelines, programmers have their personal preferences and, quite often, use features of MPEs in unusual ways. For these reasons, designs are obscured in implementations and programs are difficult to develop, debug and modify.

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

processes. The loader performs all process and resource management, as specified in the script.

Although, Ensemble provides a productive framework for implementing and maintaining MP applications, it cannot guarantee absence of design or implementation errors. 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 have proposed a specification composition technique [5,6,16], which is directly associated with the MP program composition of Ensemble. We specify formal specifications of program components and compose the formal specification of applications. The application scripts, which direct composition of applications, also direct the composition of formal specifications. We have used the Petri net (PN) formalism and in particular Coloured Petri Nets [12] for expressing and composing specifications. PNs are well founded, have been widely used to specify parallel software systems and are supported by a number of tools.

The association of programs and specifications provides the basis for testing and debugging, particularly detecting errors supported by formal methods, using program and specification executions in synergy. This approach is in accordance with Agha [1] "... *the better way to think of formal methods is as techniques that help identify bugs rather than prove programs correct*...". We also strongly believe that the acceptance of formal methods will be facilitated when they are integrated with software engineering tools.

In the next two sections, we outline program and specification composition in Ensemble. In section 4, we introduce the integrated development methodology and we elaborate on the synergy between specification analysis and program executions. Finally, we present our conclusions and plans for future work.

2 The Ensemble Methodology

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

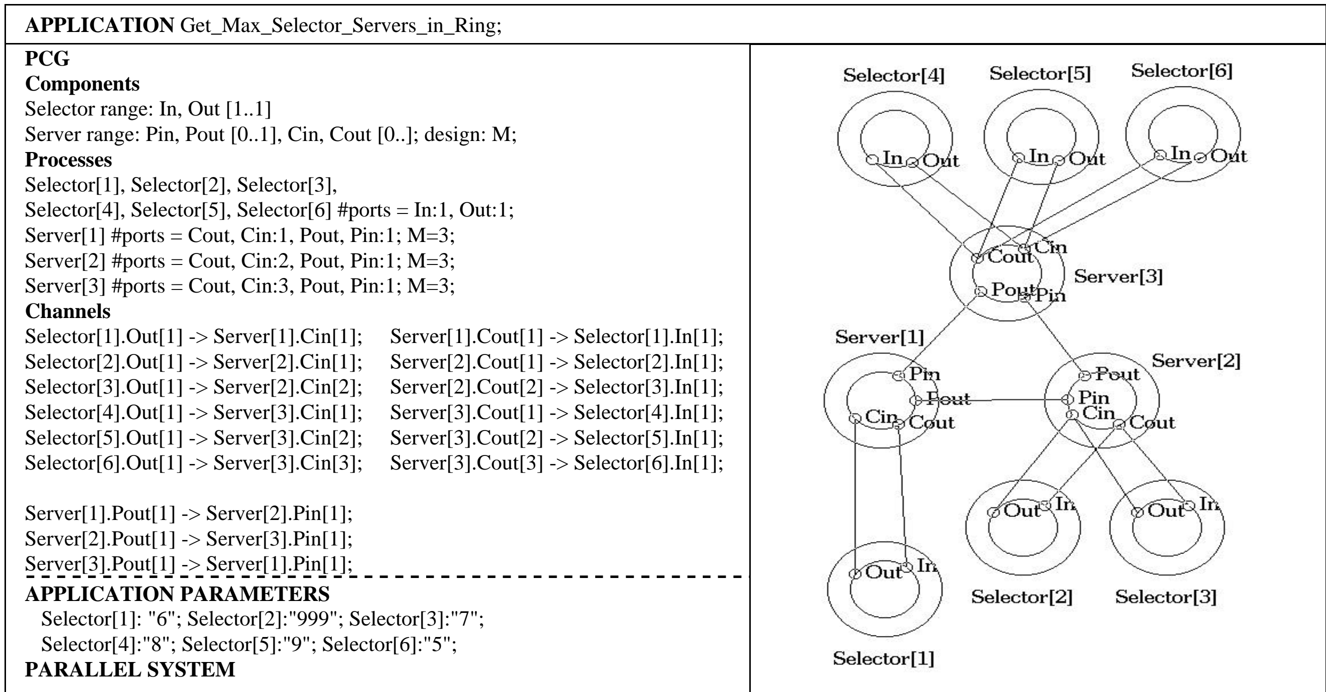
We shall implement a design, called Selector-Servers-in-Ring: Selectors are connected as client processes to associated Server processes. Each Selector sends (via port Out) its integer parameter to its Server and, eventually, receives (via port In) the required maximum. Servers receive (via their Cin ports) integer values from their client Selectors and find the local maximum. Servers are connected in a ring. They find the global maximum by sending (via Pout port) their current maximum to their next neighbour in the ring, receiving (via Pin port) the maximum of the previous server; they compare and select the maximum of these two values. Servers repeat the send-receive-select cycle M-1 times, where M is the size of the ring. Finally, Servers send (via Cout ports) the global maximum to their client Selector processes.

2.1 The Ensemble script

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

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

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



APPLICATION Get_Max_Selector_Servers_in_Ring;

PCG
Components
Selector range: In, Out [1..1]
Server range: Pin, Pout [0..1], Cin, Cout [0..]; design: M;
Processes
Selector[1], Selector[2], Selector[3],
Selector[4], Selector[5], Selector[6] #ports = In:1, Out:1;
Server[1] #ports = Cout, Cin:1, Pout, Pin:1; M=3;
Server[2] #ports = Cout, Cin:2, Pout, Pin:1; M=3;
Server[3] #ports = Cout, Cin:3, Pout, Pin:1; M=3;
Channels
Selector[1].Out[1] -> Server[1].Cin[1]; Server[1].Cout[1] -> Selector[1].In[1];
Selector[2].Out[1] -> Server[2].Cin[1]; Server[2].Cout[1] -> Selector[2].In[1];
Selector[3].Out[1] -> Server[2].Cin[2]; Server[2].Cout[2] -> Selector[3].In[1];
Selector[4].Out[1] -> Server[3].Cin[1]; Server[3].Cout[1] -> Selector[4].In[1];
Selector[5].Out[1] -> Server[3].Cin[2]; Server[3].Cout[2] -> Selector[5].In[1];
Selector[6].Out[1] -> Server[3].Cin[3]; Server[3].Cout[3] -> Selector[6].In[1];

Server[1].Pout[1] -> Server[2].Pin[1];
Server[2].Pout[1] -> Server[3].Pin[1];
Server[3].Pout[1] -> Server[1].Pin[1];

APPLICATION PARAMETERS
Selector[1]: "6"; Selector[2]:"999"; Selector[3]:"7";
Selector[4]:"8"; Selector[5]:"9"; Selector[6]:"5";
PARALLEL SYSTEM

Figure 1: The application script and (part of) the annotated PCG of Selector-Servers-in-Ring

The second part of the script, headed by APPLICATION PARAMETERS, specifies process parameters required in the application. In the example script, each Selector is given an integer parameter. Finally, the third part, headed by PARALLEL SYSTEM, specifies information required by specific MPEs and the underlying architectures (not present here).

2.2 The reusable components

The Ensemble components compute a result or provide a service and do not involve any process management or assume any topology in which they operate. Instead, they support a range of open ports for point-to-point communication with any compatible port of any process in any application and are reusable as executable library components. The ranges of their communication types are specified in the scripts. All send and receive operations in processes refer to their interface ports, identified by a communication type and a port index within the type. The underlying architecture and tools of Ensemble handle all the details regarding the initialisation of component interface.

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

```
            /* Server Code */
#include "ensemble.h"
void realMain(Interface, argc, argv)
      struct port_types *Interface;
      int argc; char **argv;
{  int Typecount=4;

   GetParam(M); Max=minint;
   for(j=1;j<=Interface[Cin].PortNo;j++)
   {  receive(Cin,j,V);if (V>Max) Max=V;}
   for (i=1;i<=M-1;i++)
   {  send(Pout,1,Max); receive(Pin,1,V);
      if (V>Max) Max=V;}
   for(j=1;j<=Interface[Cin].PortNo;j++)
   Send(Cout,j,Max);
}
```

Figure 2: Source code for the Server component

Typically the programmer will have to create the application script, either by hand or with the assistance of a graphical tool. From this script a skeleton code for each component is generated, denoted by bold the source codes of figure 2. The programmer adds the actual code

implementing application computations and builds the executables. Ensemble tools handle all the details regarding library linking, makefile generation, appropriate header files inclusion etc.

2.3 Composition of applications

An Ensemble tool called the Loader is in charge of spawning processes from component executables and establishing their interconnection scheme as this is described in the PCG part of the script. There is one universal Loader program for all applications in each MPE. In this section we outlined the basic aspects of Ensemble methodology and its tools, which are relevant in the context of this paper. Ensemble tools [3,2,4] respectively for PVM, Parix [15] and MPI have been developed. Information on Ensemble may be found on http://www.di.uoa.gr/~ensemble.

3. Specifications and their Composition

To reflect the Ensemble 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.

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

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

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

3.1 Specification Components

We have used the Coloured Petri Net (CPN) formalism for expressing and composing specifications. CPNs 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.

A component specification can be modelled on standard CPNs when its interface 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. Template CPNs are very close to the notion of pages in [12] as they are also "*flat*" structures (pages are non-hierarchical CPNs). A template CPN has a unique name, from which process specifications, called composable CPNs, can be instantiated (as a page having several page instances). Instantiation of composable CPNs from templates involves structural modification of the net, whilst the page instances are exact copies of the original page. We have studied two different approaches to composition of CPN: the place fusion approach [5,6], that models each communication channel explicitly and the environment unification approach [16], which we follow in this paper, that models communication channels implicitly, by coupling arc inscriptions.

We have defined a syntactic form for expressing template CPNs, which is more convenient for mechanical composition than graphical objects. Due to space limitations we will not elaborate on template CPNs but we will use graphical representations. In Figures 5 and respectively 4 of [17] in this volume, show the CPN representation of the Selector and Server components, respectively. One detail that requires consideration is the inscriptions of the arcs coming from and going to a components interface. The information that this inscription carries is the name of the outgoing ports, the data that is send, and the name of the recipient port respectively. Depending on the action (send or receive), in the template CPN one of the two ports is unbounded (declared with a "?"). This question mark will be later replaced, in the composition process, with the actual port.

3.2 Composition of Specifications

We now present the composition of specifications, directed by Ensemble scripts. Composition is performed in four steps. The composed CPN, corresponding to the script of figure 1 is illustrated in figure 6 of [17], in this volume, using the environment unification place approach.

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

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

values for design and application parameters are provided. If the interface is valid, we create process specifications. For each communication type, the actual number of arcs with inscriptions 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 channel section of the script. In this step all arcs are connected to the Environment place and all "?" in the inscriptions are replaced by actual ports, as directed by the channel section of the script. 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 specified in the script are actually connected.

4 Implementation and Testing Methodology

Ensemble supports the design and implementation phases of the software development life cycle. Gorton and Jelly present in [10] a number of challenges that must be addressed by a distributed systems designer. The architecture of Ensemble deals with a number of these challenges: scalability requirements, inter-component communications, design validation, choosing synchronisation and MP mechanisms, portability constraints. In this section, we propose the integration of formal methods and execution analysis which will cover challenges that apply to the testing and debugging phase of software development life-cycle. An overview of the framework of the proposed methodology is depicted in figure 3.

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

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

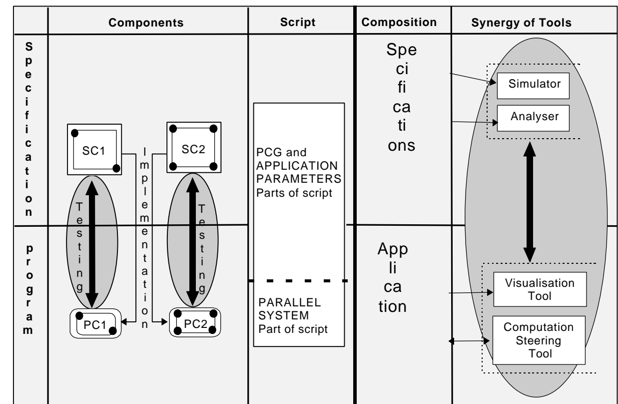


Fig. 3: Overview of the development methodology.

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

4.1 Implementation, Testing and Debugging of Program Components

Implementation of program components is guided by the associated specification components. The specification components may reflect various levels of abstraction. Their interface part must be represented in detail, even at the highest abstraction level. If we are interested only in the behaviour of the application that is related to its parallel nature, there is no need to use a detailed representation for the internal actions. The highest abstraction level that can be used, is the explicit modelling of the communication operations. All other operations may be represented "*abstractly*", since their participation in the behaviour of the application is restricted in the internal behaviour of the components. Heiner in [11] associates all usual program constructs, such as for loops, if-then-else statements etc. to specific PN constructs. If

required, we can model program constructs in detail by using these associations.

4.1.1 Implementation of Program Components

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

Implementation of RealMain actions is guided by the corresponding specification component's internal structure. With Heiner associations the implementation is mechanical.

4.1.2 Testing Individual Components

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

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

The testing environment along with the program component is in fact a downsized parallel program. The program components may be *"instrumented"* and monitoring, visualisation, and computation steering tools, e.g. [9] are used to analyse parallel aspects of program behaviour. Special breakpoints are inserted before and after each communication operation when producing the executable code of the program component. A computation steering tool may use these breakpoints to steer computations into states of particular interest.

We introduce a more advanced testing, based the synergistic error detection of specification and program components, which is depicted in figure 4 together with the environment components and the test script. Also depicted are analysis and simulation tools of specifications, as well as monitoring and debugging tools of programs. Specification and execution tools co-operate. On the one hand, tracing information of the composed application drives the simulation of the specification component. The simulator detects invalid events and gives the earliest possible warning. On the other hand, the specification simulator is used as an advanced computation steering tool to direct the execution of the program. The specification simulator may be used to derive valid (i.e. reachable) states or other properties of the system, driving the computation steering tool to reach corresponding program states. Any errors should be detected immediately, since the monitoring tool will report the failure of the execution to reach that state.

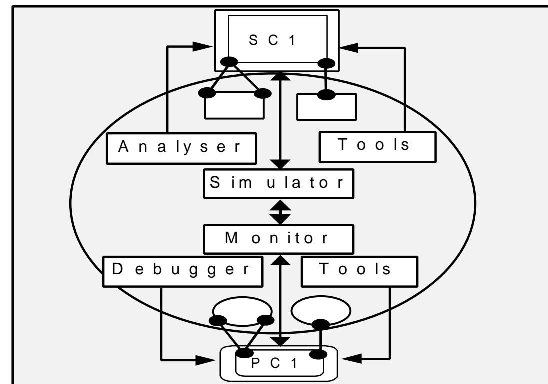


Figure 4: Synergistic testing of program and specification components

For the synergistic testing of specifications and program components the corresponding *specification testing environment* must be provided, consisting of *specification environment components*. The test scripts are the same. The specification environment components consist of an initial state (place), a single send or receive transition that will consume or produce tokens, a final state and an interface place. The type of the environment component, i.e. input or output, the colour of the tokens and the number of ports corresponding to the interface place must be specified.

Eisenstadt in [7] presents the results of a study on the reasons of the difficulty to trap errors, as well as the techniques used to locate the errors. The main reasons errors are difficult to locate fall in four categories. (i)*Cause/effect chasm* (ii) *Tools inapplicable or hampered*

(iii) *WYSIPIG (what you see is probably illusory governor)* (iv) *Faulty assumption/model or misdirected blame*. Eisenstadt reports also four major error detection techniques. (i) *Gather data* (ii) *Inspeculation (inspection-hand simulation-speculation)* (iii) *Expert recognised Clichés* (iv) *Controlled experiments*.

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

The difficulties to trap errors are also alleviated. The cause/effect chasm is reduced since the analysis of the specification components will detect the error when it occurs, even if its effect will appear much later in the execution. Thus, the notion of immediacy in debugging as presented by Ungar et al. in [18] is satisfied. The specification analysis and the monitoring tools do not interfere in the execution and do not alter program states. Furthermore, the WYSIPIG and faulty assumption cases are covered from the specifications as they modelling the actual behaviour of the system.

4.2 Testing Composed Ensemble Applications

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

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

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

Figure 5, depicts the proposed testing strategy of composed applications. If during execution an error occurs, we first check if the error can be detected in the specifications. In this case, we modify the initial design. If the error cannot be detected in the specifications, we first check if the error can be detected in component implementations. In this case the components are modified accordingly. If it cannot be detected, then we attribute it to execution environment factors. In order to correct an execution environment error, we may have to modify the specifications or the program component implementation, or the PARALLEL SYSTEM part of the script.

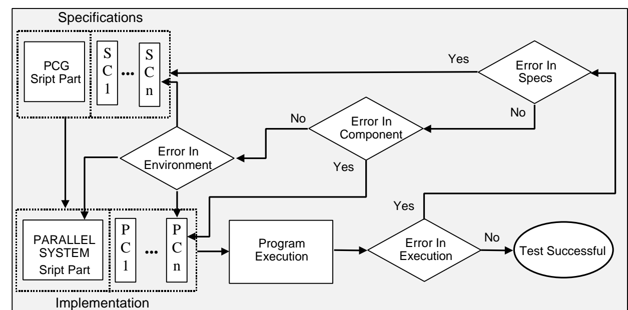


Figure 5 Testing Strategy under Ensemble

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

immediately detect that the error is due to this automatic conversion of the communication mode. Thus, the programmer should modify the design of the application, by using synchronous communication. An error that requires modification of the program component is an overflow of some data type, e.g. an assumed 64-bit integer implementation instead of an actual 32-bit one. Correction involves the variable definition to be modified from integer to long. Finally, an example of an error that must be corrected in the PARALLEL SYSTEM part of he script, is the case where a host specified is inoperable. The designer must modify the script, in order to use some other host.

5 Conclusions

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

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

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

6 References

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

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

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

[4] J.Y. Cotronis: Developing Message Passing Applications on MPICH under Ensemble. PVM/MPI 98 Workshop (to appear)

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

[8] 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*.

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

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

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

[12] 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.

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

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

[15] Parsytec Computer Gmbh, (1993) *Parix v. 1.9 Manual.*

[16] 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.

[17] Tsiatsoulis, Z, Dozsa, G, Cotronis, J.Y, Kacsuk, P, (1999) Associating Composition of Petri Net Specifications with Application Designs in Grade, in this volume..

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