Methods and Techniques for the Development of Parallel and Distributed Applications

Zacharias Tsiatsoulis[1]

Dept. of Informatics and Telecommunications, Univ. of Athens, 15784 Athens, Greece
zack@di.uoa.gr

Abstract. In this PhD thesis we present a methodology for the objective error detection for message passing applications composed by the Ensemble composition architecture. The methodology may be considered as a Parallel and Distributed Software Engineering process. It involves software development management, and links design, implementation and testing in an integrated development process. We propose the use of specifications of software components and a specifications composition technique, which is directly related to the applications composition of Ensemble, to produce the specifications of applications. We follow the concept of *lightweight formal methods*, which are applied during program execution as tools that provide assistance for *error detection*. We propose an extension to the coloured Petri net model, called template CPN, which is capable to model parametric interfaces of Ensemble software components. We model communication with Petri nets, covering all point to point communication operations for MPI and PVM. We finally propose the use of execution monitoring tools in synergy with the use of specifications simulation tools. This synergy provides the ability to derive objective conclusions on the behaviour of the application and on the detection of possible errors.

1 Introduction

In this thesis we are dealing with the problem of *testing and debugging parallel message passing applications*, which are developed following the software composition model. We propose a specifications component model, based on coloured Petri nets, to represent specifications of software components, able to model their open and scalable interfaces. We then propose a common framework to perform application and their specifications composition. We incorporate these techniques in a methodology, which provides an objective error detection capability. To achieve this, the methodology combines tools from the "*world* " of programs, with tools from the "*world* " of formal specifications. From the programs world we have used *visualization, monitoring and tracing* tools, while from the specifications world we have used a specifications *simulator*.

The methodology may be included in the framework of the *software life-cycle model* of Software Engineering, aiming at extending it in Parallel and Distributed Systems (*Parallel and Distributed Software Engineering*). We propose the integration of

[1] Supervisor: Assistant Professor J.Y. Cotronis

the methodology in the design and implementation phases, as well as in the testing and fine-tuning phase of the waterfall model, without limiting the methodology only to management of software development, but instead linking design, implementation and testing in an integrated development process.

We have followed the logic of *lightweight formal methods*, which are applied during program execution as tools that provide assistance for *error detection*. This logic does not invalidate the value of *heavyweight formal methods* but is supplementary to them.

The research performed during this thesis extends in three different fields of Computer Science: in the field of *Message Passing Parallel Programming*, in the field of *Modular Design and Component Software* and finally and most widely in the field of *Formal Methods*.

2 A Parallel and Distributed Software Engineering Approach

The message passing model is the most popular model of parallel programming. Message passing parallel programs create a number of processes, each with its own local data, a unique name, which communicate by sending and receiving messages to and from other processes. For the implementation of message passing parallel applications, integrated development environments (*Message Passing Environments-MPEs*) are used, which provide a useful abstraction of underlying architectures, thus simplifying architecture resource management. The most popular are PVM [7] and MPI [13].

Design and development of message passing parallel applications is a complicated process. Essentially a programmer has to design the combined behaviour of the processes that constitute an application, take into consideration the restrictions that are imposed in process execution by their interaction, implement correctly processes communication and synchronization and finally take into consideration additional restrictions that are related to the MPE used. The applications are developed following two models, the SPMD model (*Single Program Multiple Data*) and the MPMD model (*Multiple Program Multiple Data*). In the SPMD model, the application spawns a number of identical processes, each of which executes the same program on different data. In the MPMD model, the application spawns a number of different processes that execute different programs on different data.

The adaptation of sequential programming software engineering processes to parallel programming has specific difficulties regarding error detection [16]. Apart from general problems encountered in sequential programs [12], there also exist specific problems in detecting errors in parallel programs [6]. The solutions that have been proposed concern mainly the adaptation of methods that are used in sequential programs [1], to be applied to the individual sequential parts of parallel programs. The most common approach is to assign each process of a parallel program to a different sequential debugger.

In this thesis, we have coped with the problem of testing and objective error detection (i.e. as nearly as possible to the error itself rather than its observable consequences) in message passing parallel applications. Our approach lies in combining

techniques and tools from the "*world*" of programs, with techniques and tools from the "*world*" of formal specifications. From the world of programs such tools are *visualization, monitoring* and *execution tracing* tools. The visualization tools provide a visual chart of program execution, presenting in a graphic way process interactions, function calls, sending of messages etc. Monitoring tools combine features of debugger programs and visualization tools. Finally, execution *tracing* tools store in a file information relative with actions implemented during program execution, giving thus the possibility of replay of this execution, that is used in the so called *post mortem analysis*.

From the world of specifications we have used *specification simulation tools*, or simulators. The simulators are tools which, starting from an initial state of the specifications, follow one from all possible courses of program execution, moving from a state to one of its successor states, until the simulation is completed (i.e. there is no successor state available) providing thus conclusions on this specific course of program execution.

The direction that we followed was to include our methodology in the framework of the software life-cycle model of Software Engineering, in order to extend its application to Parallel and Distributed Systems (Parallel and Distributed Software Engineering). The various software life-cycle models determine the sequence of different activities and their relations in the software development process. There are three major life-cycle models, the linear phases or waterfall model, the spiral model, and the circular model [15].

The waterfall model represents the most widespread approach. Even though there exist a number of different variants, it always consists of similar phases, which are implemented in a similar "*chronological*" order. Our methodology may be included, on one hand in the Design and Implementation phases, on the other hand (and mainly) in the Testing and fine-tuning phase of the waterfall model. However, it is not limited only to management of the software development process, but instead design, implementation and testing are also connected and interacting in an integrated development process.

Our approach, follows the logic of *lightweight formal methods*, which are applied during program execution as tools that provide assistance for *error detection* [60], contrary to the logic of *heavyweight formal methods*, which are used to prove the correctness of programs before execution, such as *model checking* and *theorem proving*. Actually, we have tried to place our methodology in the framework of software engineering for parallel and distributed systems and not in the framework of the theoretical models of behaviour for these systems. Our approach does not invalidate the value of *heavyweight formal methods*, on the contrary it is proposed as supplementary to them.

Modular Design is a technique in which independent software units, that execute specific operations, are combined in such a way that the desired application is produced. In modern terminology the independent software units are called *software components* or simply *components*. The process of combining software components is called *composition.* Software composition is based either in object-oriented techniques, or in message passing with establishment of communication channels between components.

The basic advantage of modular design is that programs are not implemented in a monolithic unit, but in separate components, which have well defined operations and interfaces, increasing thus reliability and decreasing costs of program implementation, simplifying implementation itself, facilitating modification of programs so that they cover altered requirements and supporting the reusability of components in new programs.

The *Ensemble* application development methodology, which has been developed in the University of Athens, may be included in the framework of messages passing software composition. An application in Ensemble is an *ensemble* that consists of modular software components, executable programs and composition directives, which determine the application processes, their topology and their mapping in the architecture. The directives are interpreted and a composition program implements the application composition. Ensemble provides the option for MPMD programming and apart from various MPEs it has been extended to GRID environments.

Even though Ensemble provides a productive framework for implementation and maintenance of message passing applications, by supporting their correct design, it cannot guarantee the absence of design and implementation errors. Moreover, composition is prone to new types of errors, such as use of wrong components and undefined or incompatible binding of communication channels.

The most suitable solution to this problem is the use of formal methods, since they may, in case of heavyweight methods, provide formal verification of correct program behaviour before their implementation, or, in case of lightweight formal methods, improve special testing and monitoring techniques used in *performance monitoring* and in d*istributed debugging*.

For distributed systems various formal methods have been used, such as finite state machines, Petri nets, process algebras etc. The distributed system is modelled as a whole and then the model is executed in a simulation or analysis tool. The simulation may locate possible abnormalities in model execution, e.g. deadlocks and faulty synchronisations, which lead to non-deterministic behaviour of the system. The behaviour of a composed message passing application cannot, in general, be determined from the behaviour of its components. It is, however, possible to compose the formal specifications of components, to provide the formal specifications of the application, which can then be tested and verified.

The main part of our research, is related to methods and techniques for modelling the formal specifications of Ensemble software components, modelling the composition of these components into a single parallel application, as well as determining a way of interaction between the simulation of formal specifications and the analysis of execution of the associated programs. We also, dealt with the redesign of the Ensemble software components for PVM, in order to adopt, as much as possible, a common form and functionality with the Ensemble software components for MPI [5].

3 Specifications Composition with the Ensemble Methodology

We propose a specifications composition technique, which is directly related to the applications composition of Ensemble. The applications composition technique of

Ensemble is extended in order to cover the composition of the associated formal specifications, in a common way. Initially we define the specifications of software components, which in analogy are the *specifications components*. Consequently, the formal specifications of applications (*application specifications*) are composed from specifications components in analogy to the applications composition from software components. The composition directives that control the applications composition also control the specifications composition.

The use of specifications composition as a design methodology of Ensemble applications, leads to an integrated development process that follows the classic paradigm, where application design is followed by the implementation of specifications components and their composition, to produce the application specifications. At the same time the corresponding software components are being implemented or already implemented software components are reused. The world of specifications and the world of programs are actually disconnected. The single common part and essential advantage of Ensemble are the composition directives of applications and specifications, which are the same, a fact that gives the possibility to locate certain types of errors. The specifications composition is also used in the framework of an objective error detection methodology, in which we propose the use of tools from the world of programs (e.g. monitoring tools) in synergy with the use of tools from the world of specifications (e.g. specifications simulators). Essentially, in this way, the world of programs and the world of specifications are not disconnected anymore, since apart from the common composition directives they have now another common part, the interaction of specifications simulation tools with execution monitoring tools.

The synergy of these tools provides the possibility to derive objective conclusions on the behaviour of the application and on the detection of possible errors. On the one hand, program execution tracing information can "*guide*" the specifications simulation. The simulator detects invalid events in the trace file and gives the earliest possible warning. On the other hand, the simulator can be used to "*steer*" the program execution. The specifications simulator produces valid (i.e. feasible) states or other attributes of the system and steers program execution to the corresponding events. In this case too, we receive the earliest possible warning, i.e. possible errors can be detected in their point of origin and not in some later point of the execution (where their side effects can be observed), since the monitoring tool will depict the inability of the program execution to reach the corresponding event.

The formal model we have used is the Petri net model [14], a graphic formalism suitable to model systems that entail concurrency and resource sharing. This formalism is a generalisation of automata theory, which allows the expression of events that occur simultaneously. A Petri net consists of *places* (*states*), *transitions* (*actions*) and directed arcs. The arcs connect places with transitions and the reverse. There are no arcs between places or between transitions. The places can contain any number of *tokens*. The transitions are *fired*, i.e. tokens are consumed from input places and produced in output places. A transition is enabled if tokens are present in each of its input places. In the basic form of Petri nets, tokens are not distinguished. The most complicated models of Petri nets add colour in tokens, activation time in transitions and hierarchy in the network.

The choice of Petri nets was based in the fact that they have been used widely for modelling parallel and distributed systems, they are accompanied by abundance of ex-

tensions, practical and theoretical studies on modelling of distributed systems, as well as a big number of tools for simulation and analysis of various Petri net variants. An additional reason is that due to their graphic form, they become more easily comprehensible in comparison to various algebraic representations of other models, while the programmers accept them more easily as a tool integrated in a software engineering methodology. However, the more important reason is that the particular characteristics of the Ensemble message passing software components can be adequately modelled using Petri nets.

In particular, we have defined a new class of Petri nets, which is based on the coloured Petri nets model [11] and extends it by adding the ability to describe parametrical interfaces [17] and is called template CPN.

An important part of our proposal deals with modelling of communication with Petri nets. In the bibliography, the basic classification of process communication is synchronous and asynchronous. Modelling communication with Petri nets follows this general classification, even though, in most cases it is adapted to the particular requirements of the associated theoretical models. In the case of asynchronous communication, modelling is relatively simple, owed to the inherent characteristics of Petri nets, since it is modelled with "*fusion of places*" (two places), which represent communication ports (one input and one output) [30,32], corresponding to a form of communication called "*point to point communication*", which essentially models a communication channel between two processes. For synchronous communication, most approaches use a representation that corresponds to fusion of transitions.

In our work we propose another alternative, the "*unification of places*" (or fusion of all places that represent communication ports) into a "*total*" place, which is called "*environment*", since it represents all communication channels that are established by the application [3,17,18]. In this case, the distinction of channels is made by the messages themselves through information that is included in them (in the tokens), technique that approaches the *tuple space* of Linda and other *coordination languages* [8].

Special focus has been given in modelling the parametric interface of Ensemble message passing software components. In this case, the interface places and the arcs that connect them to communication transitions correspond to a communication type, which has a range for its number of ports. The actual number of ports is determined at the time of producing the process specifications from the specifications components. An approach for modelling the range of communication ports is the replication of interface places as well as of the arcs that connect them to the static net structure. In this case, composition is based on "*fusion of pairs of places* " and models point-to-point channels individually. This approach leads to an "*explosion*" of interface places. The second approach, which is based on our proposal to model communication by "*unification of places*", is modelling the range of number of communication ports of a communication type by maintaining a single interface place and reproducing inscriptions of the corresponding arc. In this case, when handling collective communication operations (*reduction or multicast-broadcast*) the arc inscription is one and includes all individual inscriptions that correspond to each port that participates in this communication, while when handling point to point communication operations, the inscription is parametric and the structure is nested in a for-while loop, where the inscription takes the appropriate form for each communication port.

Based on modelling of communication by "*unification of places*", we have modelled all point-to-point communication operations for MPI and PVM. We have developed specific Petri net modules, which, depending on the desired communication operation, can replace the *generic* transitions that correspond to communication operations, as well as the corresponding interface place.

In the sequel, we sought specific forms of representation of coloured Petri nets, since our aim was to use a representation that is supported by an existing tool. However, we preferred not to use directly a specific representation (e.g the graphic form of the design/CPN tool), but to define a parametric description language for Petri nets, based on our theoretical model of template CPN. In this way the choice of the simulation tool is not restrictive, and independence of any specific tool is achieved. The most difficult problem we have faced in modelling of components is the inherent static nature of Petri nets, which contradicts to our requirement for dynamic configuration of parametric interface of specifications components, i.e. places that correspond to communication ports.

For this reason, the description language for Petri nets maintains the static part of specifications components (i.e. the one that represents the internal operations and calculations of the component) and separates the dynamic part, so that it can be modified depending on the application. In practice this modification concerns only reproduction of inscriptions of arcs that connect communication ports and communication transitions. The template CPNs description files start with a header, which contains those elements whose values are assigned depending on the application and which control the replication of ports.

The next step was to formulate an algorithm that implements the composition of formal specifications of Ensemble message passing software components and produces the formal specifications of a single parallel application. Initially, all template CPN files that participate in an application are retrieved from a repository. Each template CPN has a unique name; process specifications are produced by instantiating accordingly the template CPN (i.e. setting actual values to interface parameters and indexing the template name by the instantiation index) and are called *composable CPNs*. Then composable CPNs are connected through unification of interface places, and the final Petri net of the application is produced, which is called *composed CPN* or *application CPN*. All information required for the production of composable CPNs from template CPNs as well as for their composition, is received from the same composition directives that also guide the composition of the associated programs and, actually, the process is precisely the same with the process of composition of software components. Thus, templates CPNs correspond to the software components, composable CPNs to the processes that are spawned from the software components and the application CPN to the composed application.

4 Using the Methodology

The methodology has two aspects; on the one hand it may be applied in application development within the classic framework and on the other hand it may be applied in the objective error detection. The methodology has the following phases:

(i) Design and Specifications Components Testing Phase

In this phase the programmer designs the application and develops the specifications components. The specifications can model any level of program detail. The feasible detail level is limited mainly by data representation of actual Petri net tools. The parallel behavior needs to be modeled in any detail level.

(ii) Composition Directives Testing Phase

In this phase the composition directives of Ensemble are developed and tested. The programmer may determine tests that are to be executed by the composed application and its specifications, based on information from application design. Testing is based on the fact that directives are common for specifications and programs.

(iii) Software Components Implementation and Application Composition Phase

Based on specifications the corresponding software components are developed. The complexity of the process is less than implementing monolithic code or specifications, since it is not required to incorporate in the code topology creation and management as well as interactions between processes or their specifications.

(iv) Individual Software Components Testing Phase

In this phase, "stub" processes and specifications test the interactions of components incrementally. A number of "minimized" applications are designed, which comprise the software component and the "stub" component and which test the possible configurations of the components interface. These applications as well as their specifications are then composed, the applications are executed and a trace file is produced for each. This file is used, for interaction with the specifications simulation tool. The communication operations that have been traced in the file as completed, are associated with corresponding transitions in the specifications, giving the possibility to test the compatibility of specifications and programs.

(v) Application Testing Phase

In this phase, after a complete application has been composed as well as its specifications, the procedure that was described in the previous phase is followed, but now interaction of program execution and specifications simulation concerns a complete application.

Under the aspect of application development, where components specifications and software components are developed separately, using features of the Ensemble methodology, certain types of errors are detected. The first type of errors that can be detected is the category related to the compatibility of specifications and programs.

The program that "*reads*" the files that contain the syntactic description of the corresponding specifications, detects an important number of errors. Some of them are simple syntactic errors, e.g. not declared identifier, while others are related to information of communication ports and channels binding between the components that are present in the composition directives:

- *Channels binding*. In this case we check whether all communication ports that are declared in the composition directives are used by the application, as well as whether they are declared also in the corresponding template CPN. Actually this test is the syntactic analysis (parsing) of the template CPN combined with the test of the corresponding composition directives.
- *Channel type*. This test is performed before composition, so that when the composition directives determine the channel binding between two incompatible ports

(e.g. that support different data types) the composition program detects it and terminates with an appropriate error message.

This aspect includes the (i) *Design and Specifications Components Testing Phase,* (ii) Composition *Directives Testing Phase* and (iii) *Software Components Implementation and Application Composition Phase* of the methodology. In these phases mainly static testing is performed based on syntactic and semantic information contained in the specifications components and in the composition directives.

The aspect of objective error detection includes the (iv) *Individual Software Components Testing Phase* and (v) *Application Testing Phase* of the methodology.

In order to achieve the desired objective error detection, the designer uses the execution monitoring and visualization tools in synergy with the simulation tool (which performs the simulation of the application CPN). Actually, in this way, the "*world*" of programs and the "*world*" of specifications are now connected through the interaction of specifications simulation tools with implementation monitoring tools. The synergy of these tools provides the ability to draw objective conclusions on the behavior of the application and on the detection of possible errors.

This test can locate:

- Errors in communication operations and other execution environment errors. This category includes normal termination of applications. This means that all send operations are associated one-by-one to receive operations. If such errors occur, in the case of synchronous communication a *deadlock* occurs, while in the case of asynchronous communication *left over messages* remain in the environment, a case which in PVM may not be detected since the application may terminate; in the specifications however *left over messages* correspond to tokens in the environment place that are not consumed by some receive operation.
- Correctness of other component elements, which are not related to communication operations. This type of errors may also be detected. The methodology supports the detection of errors in the sequential parts of the components, using the associations proposed by Heiner [9].

In Ensemble applications we may classify errors in the following categories: (i) *Execution Environment Errors*, (ii) *Composition Errors*, (iii) *Software Components Implementation Errors* and (iv) *Design Inadequacy Errors*. Error detection begins with execution environment errors and incrementally limits the search of errors to composition errors, component implementation errors and design errors.

5 Conclusions

Summarising, the most important contribution of this thesis in the field of testing and error detection for message passing parallel applications, consists in the definition of a systematic process that assists substantially in the earliest possible objective detection of probable errors, e.g. the detection of the place of the actual error appearance and not the detection of the place where the errors side-effects are becoming observable.

6 References

[1] Ball, T. and Eick, S.G. (1996) Software Visualization in the Large, *IEEE Computer*, **29(4)**, 33-43.

[2] 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, Chapman & Hall, 95-106.

[3] 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, vol.1, Εκδόσεις Νέων Τεχνολογιών, 299-312.

[4] Cotronis, J.Y. and Tsiatsoulis, Z. (1998) Specification Composition for the Verification of Message Passing Program Composition, *Microprocessors and Microsystems* **21**, Elsevier, 595-603.

[5] Cotronis, J.Y. and Tsiatsoulis Z. (2002), Modular MPI and PVM components, in *Proc. of PVM/MPI'02*, Springer, LNCS **2474**, 252-259.

[6] Foster, I. (1995) *Designing and Building Parallel Programs*, Addison-Wesley Publishing Company, **ISBN 0-201-57594-9**.

[7] 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*.

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

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

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

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

[12] Liebermann, H. (1997) The Debugging Scandal and What to Do About It, *Communications of the ACM*, **40(4)**, 26-29.

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

[14] Petri, C.A. (1962) Kommunikation mit Automaten, *Ph.D Thesis*, Schriften des Rheinish Westfaelishen Institutes fuer Instrumentelle Mathematik, Universitaet Bonn.

[15] Shatz, S.M. (1993) Development of Distributed Software: Concepts and Tools, New York, MacMillan Publishing Company, **ISBN 0-02-409611-3**.

[16] Tsai, J.J.P., Bi, Y.D. and Yang, S.J. (1996) Debugging for Timing-Constraint Violations, IEEE Software, 25(3), 89-99.

[17] Tsiatsoulis, Z. and Cotronis, J.Y. (2000): Testing and Debugging Message Passing Programs in Synergy with their Specifications, *Fundamenta Informaticae* **41**(3), 341-366.

[18] Tsiatsoulis, Z., Cotronis, J.Y. and Floros, E. (1999) Testing and Debugging Message Passing Applications Based on the Synergy of Program and Specification Executions, in *Proc. of the Seventh Euromicro Workshop on Parallel and Distributed Processing,* Funchal, Portugal, 1999, IEEE Computer Society press, pp. 196-203.