

***Enhanced Debugging Methods  
for Parallel and Metacomputing  
Applications Based on Macrosteps***

**PhD theses**

Written by Róbert Lovas

Advisor: Péter Kacsuk (MTA SZTAKI)

*Faculty of Electrical Engineering and Informatics  
Budapest University of Technology and Economics*

Budapest

2005

# CONTENTS

<b><u>1</u></b>	<b><u>PREFACE .....</u></b>	<b><u>9</u></b>
<b>1.1</b>	<b>INTRODUCTION .....</b>	<b>10</b>
1.1.1	CLASSIFICATION OF DISTRIBUTED DEBUGGERS.....	10
1.1.1.1	Interactive debugging of remote sequential processes .....	10
1.1.1.2	Trace, replay and debugging .....	11
1.1.1.3	Integrated testing, active control and debugging.....	11
1.1.1.4	Automated detection of global predicates, active control and debugging....	12
1.1.2	COMMERCIAL TOOLS .....	12
1.1.2.1	Distributed Debugging Tool (DDT).....	12
1.1.2.2	Totalview debugger .....	14
1.1.3	NON-COMMERCIAL DEBUGGERS .....	15
1.1.3.1	MAD environment.....	15
1.1.3.2	DDBG and STEPS .....	16
1.1.3.3	P2D2 debugger .....	18
1.1.3.4	Debuggers for JAVA.....	19
1.1.4	COMPARISON OF DEBUGGER TOOLS .....	20
<b>1.2</b>	<b>PROPOSED DEBUGGING CYCLE IN P-GRADE.....</b>	<b>21</b>
<b><u>2</u></b>	<b><u>MACROSTEP-BASED DEBUGGING TECHNIQUE FOR GRAPNEL</u></b>	
	<b><u>APPLICATIONS.....</u></b>	<b><u>24</u></b>
<b>2.1</b>	<b>THE FORMAL DESCRIPTION OF GRAPNEL PROGRAMS FOR DEBUGGING</b>	
	<b>PURPOSES.....</b>	<b>24</b>
2.1.1	PREFACE.....	24
2.1.2	INTRODUCTION TO GRAPNEL AND GRAPNEL* LANGUAGE .....	24
2.1.2.1	GRAPNEL language .....	24
2.1.2.1.1	The Process Model .....	25
2.1.2.1.2	The Communication Model.....	25
2.1.2.1.3	The Process.....	26
2.1.2.1.4	The Application Layer.....	26
2.1.2.1.5	The Process Layer .....	27
2.1.2.1.6	Data Declarations and Definitions for Processes .....	30
2.1.2.2	GRAPNEL* language .....	30
2.1.2.2.1	New language elements in GRAPNEL* .....	30
2.1.2.2.2	Limitations of GRAPNEL program for automated modelling.....	31
2.1.3	COLOURED PETRI-NET PATTERNS FOR GRAPNEL* PROGRAMS.....	32
2.1.3.1	Short overview of coloured Petri nets and hierarchical decomposition .....	32
2.1.3.2	Application level transformations .....	33
2.1.3.2.1	Process .....	33
2.1.3.2.2	Communication ports .....	34
2.1.3.2.3	Communication channels .....	35
2.1.3.3	Process level transformations .....	36
2.1.3.3.1	Processes.....	36
2.1.3.3.2	Variables.....	36
2.1.3.3.3	Sequential code.....	37
2.1.3.3.4	Conditional construct.....	38

2.1.3.3.5	Loop construct .....	39
2.1.3.3.6	Communication operations.....	40
2.1.4	BUILDING CPN MODEL AUTOMATICALLY .....	43
2.1.4.1	Traversal algorithm for CPN generation .....	44
2.1.5	RELATED WORKS .....	46
<b>2.2</b>	<b>FORMALISATION OF MACROSTEP-BASED DEBUGGING TECHNIQUE FOR GRAPNEL* APPLICATIONS .....</b>	<b>46</b>
2.2.1	PREFACE.....	46
2.2.2	MACROSTEP-BASED EXECUTION AND EXECUTION TREE .....	47
2.2.3	FORMAL DESCRIPTION OF MACROSTEP-BASED EXECUTION.....	50
2.2.4	RELATED WORKS .....	53
<b>2.3</b>	<b>CORRECTNESS OF MACROSTEP-BASED EXECUTION.....</b>	<b>53</b>
2.3.1	PREFACE.....	53
2.3.2	BASIC NOTATIONS: KRIPKE STRUCTURES .....	53
2.3.3	PARTIAL ORDERING TECHNIQUES .....	54
2.3.4	MACROSTEP-BASED EXECUTION: A PARTIAL ORDERING TECHNIQUE .....	55
2.3.4.1	Emptiness .....	55
2.3.4.2	Ample decomposition.....	56
2.3.4.3	Invisibility.....	57
2.3.4.4	Cycle closing condition .....	59
2.3.5	FURTHER REDUCTION: EXECUTION TREE .....	60
2.3.6	RELATED WORKS .....	61
<b>3</b>	<b><u>MODEL CHECKING METHODS IN PARALLEL DEBUGGING .....</u></b>	<b>62</b>
<b>3.1</b>	<b>RUN-TIME CHECKING OF TEMPORAL LOGIC SPECIFICATION DURING CONTROLLED EXECUTION.....</b>	<b>62</b>
3.1.1	PREFACE.....	62
3.1.2	INTRODUCTION .....	63
3.1.2.1	Theoretical background for temporal logic assertions .....	63
3.1.2.2	Macrostep Debugging in P-GRADE.....	65
3.1.3	MACROSTEP DEBUGGING WITH TEMPORAL LOGIC ASSERTIONS .....	66
3.1.3.1	Temporal logic specification .....	66
3.1.3.2	Atomic predicates .....	67
3.1.4	CHECKING TEMPORAL LOGIC ASSERTIONS WITH TLC.....	69
3.1.4.1	The evaluation protocol.....	70
3.1.4.2	Formal description of protocol .....	70
3.1.4.3	Messages types of protocol .....	72
3.1.5	RUN-TIME EVALUATION OF TL ASSERTIONS IN P-GRADE FRAMEWORK .....	73
3.1.5.1	Initialization phase.....	74
3.1.5.2	Assert detection .....	74
3.1.5.3	Evaluation of atomic predicates .....	75
3.1.5.4	Atomic predicate – debugger interface.....	75
3.1.6	OPTIMISATION ISSUES OF MODEL CHECKING BASED ON ASSERTIONS .....	76
3.1.7	SUMMARY: FUTURE WORK AND OPEN ISSUES .....	77
<b>3.2</b>	<b>FURTHER OPTIMISATION OF MACROSTEP-BASED DEBUGGING.....</b>	<b>78</b>
3.2.1	PREFACE.....	78
3.2.2	PETRI-NET SIMULATION FOR STEERING THE EXECUTION .....	79
3.2.2.1	Off-line simulation and steering.....	79

3.2.2.2	On-the-fly steering of debugging .....	80
3.2.2.3	Termination conditions for steering .....	80
3.2.2.4	Detection of discovered execution paths .....	82
3.2.2.5	Reduced CPN model .....	82
3.2.3	PARTITIONING, EXECUTION AND RELEASING OF GRAPNEL PROGRAMS .....	84
3.2.3.1	Partitioning of GRAPNEL programs for macrostep execution.....	84
3.2.3.1.1	Message-independent applications .....	84
3.2.3.1.2	Message-dependent applications .....	85
3.2.3.1.3	Summary.....	86
3.2.3.2	Execution on distributed platforms .....	86
3.2.3.3	Rayleigh model for estimation of error density.....	86
3.2.3.4	Controlled execution of released software .....	88
3.2.4	SUMMARY, RELATED AND FUTURE WORKS .....	89

## **4 DEBUGGING OF METACOMPUTING APPLICATIONS ..... 90**

### **4.1 METADEBUGGING: NEW DEBUGGING METHODS FOR METACOMPUTING**

<b>APPLICATIONS .....</b>	<b>90</b>
4.1.1 PREFACE.....	90
4.1.2 INTRODUCTION .....	90
4.1.3 THE HARNESS FRAMEWORK AND JPDA.....	91
4.1.3.1 Harness Metacomputing System .....	91
4.1.3.2 Java Platform Debug Architecture .....	93
4.1.4 METADEBUGGING: PRINCIPLES AND GOALS.....	93
4.1.5 ADAPTIVE DEBUGGING FRAMEWORK.....	94
4.1.5.1 Architecture .....	94
4.1.5.2 Start-up .....	96
4.1.5.3 Handling of Dynamic Behaviour.....	96
4.1.5.4 “Step-into” for Remote Method Invocations.....	97
4.1.5.5 Navigation and Visualization .....	98
4.1.5.6 Invocation of third-party debuggers .....	100
4.1.5.7 Programming of debugger.....	101
4.1.5.8 Breakpoint sets .....	102
4.1.6 RELATED WORK.....	102
4.1.7 SUMMARY .....	103

### **4.2 ENHANCED MACROSTEP-BASED DEBUGGING TOWARDS METACOMPUTING**

<b>APPLICATIONS .....</b>	<b>103</b>
4.2.1 PREFACE.....	103
4.2.2 INTRODUCTION .....	103
4.2.3 METADEBUGGER FOR HARNESS .....	104
4.2.4 MACROSTEP-BASED DEBUGGING IN HARNESS.....	105
4.2.5 ARCHITECTURE DEPENDENCIES .....	109
4.2.6 ARCHITECTURE OF MACROSTEP-BASED HARNESS DEBUGGER.....	110
4.2.7 CONCLUSION AND RELATED WORK.....	111

## **5 SUMMARY ..... 112**

## **6 ACKNOWLEDGEMENT ..... 114**

<b>7</b>	<b><u>REFERENCES .....</u></b>	<b><u>115</u></b>
----------	--------------------------------	-------------------

<b>8</b>	<b><u>APPENDIX .....</u></b>	<b><u>119</u></b>
----------	------------------------------	-------------------

<b>8.1</b>	<b>CASE STUDY: PRODUCER-BUFFER PROBLEM AND ITS CPN MODEL .....</b>	<b>119</b>
------------	--	------------

## FIGURES

Figure 1 – Classification of distributed debugging methodologies (source: [1]) .....	10
Figure 3 – Evaluation points in TotalView (source: [38]) .....	15
Figure 4 – Example event graph visualization of a program execution (source: [2]) .	16
Figure 5 – Level of analysis in STEPS (source: [68]).....	17
Figure 6 – Step by step replay of states from the information log (source: [68]) .....	18
Figure 7 – Sample debugging scenario with P2D2 (source: [39]) .....	19
Figure 8 – Proposed debugging cycle in P-GRADE framework .....	22
Figure 9 – Language elements of GRAPNEL and editor functions at application level .....	27
Figure 10 – Language elements of GRAPNEL and editor functions at process level	28
Figure 11 – CPN model of GRAPNEL process .....	34
Figure 12 – CPN model of GRAPNEL output communication port (OUTPORT) ....	34
Figure 13 – CPN model of GRAPNEL input communication port (INPORT) .....	35
Figure 14 – CPN model of GRAPNEL communication channel (with ports and processes) .....	35
Figure 15 – CPN model of GRAPNEL sequential code segment (SEQ).....	38
Figure 16 – CPN model of GRAPNEL conditional construct (COND) .....	39
Figure 17 – CPN model of GRAPNEL loop construct (LOOP) .....	40
Figure 18 – CPN model of GRAPNEL output communication action (CAO) .....	41
Figure 19 – CPN model of GRAPNEL input communication action (CAI).....	41
Figure 20 – CPN model of GRAPNEL alternative input communication action (CAIALT).....	42
Figure 21 – Consistent and inconsistent cuts .....	47
Figure 22 – Illustration for macrostep-based execution .....	48
Figure 23 – Execution Tree with navigation panel .....	49
Figure 24 – Occurrence Graph .....	51
Figure 25 – Stuttering equivalent paths .....	55

Figure 26 – Stuttering equivalent paths during macrostep based execution .....	55
Figure 27 – Invisible paths after reordering of transitions based on commutativity ...	58
Figure 28 – Illustration for cycle closing condition (source: [8]).....	60
Figure 29 – Temporal Logic Checker and DIWIDE applied on Buffer application ...	68
Figure 30 – Initialization and assert handling of temporal logic checker .....	74
Figure 31 – Evaluation of atomic predicates .....	75
Figure 32 – Meta-code of <i>SearchNodes</i> function.....	82
Figure 33 – Reduced CPN model of Buffer application .....	83
Figure 34 – Reduced occurrence graph .....	83
Figure 35 – Analysis of reduced occurrence graph .....	84
Figure 36 – Cumulative Distribution Function (example) .....	87
Figure 37 – Abstract Model of a Harness DVM with Message Box (MB) Service....	92
Figure 38 – HARNESS metadepbugger architecture .....	95
Figure 39 – Tools of HARNESS metadepbugger .....	99
Figure 40 – Debugging cycle for metacomputing applications.....	106
Figure 41 – Fundamental architecture of macrostep-based debugger in HARNESS	111
Figure 42 – Application level of Buffer program.....	119
Figure 43 – Process level of Buffer program .....	120
Figure 44 – CPN model of Buffer program: application level.....	120
Figure 45 – CPN model of Consumer process .....	121
Figure 46 – CPN model of Buffer process .....	123

## ABBREVIATIONS

<b>CAI</b>	Communication Action Input (GRAPNEL language element)
<b>CAIALT</b>	Communication Action Input <b>AL</b> ternative (GRAPNEL language element)
<b>CAO</b>	Communication Action Output (GRAPNEL language element)
<b>CDF</b>	Cumulative <b>D</b> istribution <b>F</b> unction
<b>CM</b>	Checker <b>M</b> odule
<b>COND</b>	<b>COND</b> itional construct (GRAPNEL language element)
<b>CP net</b>	Coloured <b>P</b> etri net
<b>CPN</b>	Coloured <b>P</b> etri <b>N</b> et
<b>Design/CPN</b>	a <b>DESIGN</b> and simulation tool for <b>Coloured Petri Nets</b>
<b>DIWIDE</b>	<b>D</b> istributed <b>W</b> indows <b>D</b> Ebugger (a part of P-GRADE environment)
<b>DVM</b>	<b>D</b> istributed <b>V</b> irtual <b>M</b> achine
<b>GRAPNEL</b>	<b>GR</b> APHical <b>NE</b> twork <b>L</b> anguage
<b>GRAPNEL*</b>	a modified version of <b>GR</b> APHical <b>NE</b> twork <b>L</b> anguage
<b>GRED</b>	<b>GR</b> APHical <b>ED</b> itor (a part of P-GRADE environment)
<b>GUI</b>	<b>G</b> raphical <b>U</b> ser <b>I</b> nterface
<b>HARNESS</b>	a metacomputing framework
<b>HDPI</b>	<b>H</b> arness <b>D</b> ebugger <b>P</b> lug- <b>I</b> n
<b>HDT</b>	<b>H</b> arness <b>D</b> ebugger <b>T</b> ool
<b>HMCPI</b>	<b>H</b> arness <b>M</b> onitor & <b>C</b> ontroller <b>P</b> lug- <b>I</b> n
<b>HMPI</b>	<b>H</b> arness <b>M</b> onitor <b>P</b> lug- <b>I</b> n
<b>HSDT</b>	<b>H</b> arness <b>S</b> ystematic <b>D</b> ebugger <b>T</b> ool
<b>ID</b>	<b>I</b> dentifier
<b>INOUTPORT</b>	<b>I</b> Nput/ <b>O</b> Utput communication <b>P</b> ORT (GRAPNEL language element)
<b>INPORT</b>	<b>I</b> Nput communication <b>P</b> ORT (GRAPNEL language element)
<b>IP</b>	<b>I</b> nternet <b>P</b> rotocol
<b>JDI</b>	<b>J</b> ava <b>D</b> ebug <b>I</b> nterface
<b>JDWP</b>	<b>J</b> ava <b>D</b> ebug <b>W</b> ire <b>P</b> rotocol
<b>JPDA</b>	<b>J</b> ava <b>P</b> latform <b>D</b> ebug <b>A</b> rchitecture

<b>JVM</b>	<b>Java Virtual Machine</b>
<b>JVMDI</b>	<b>Java Virtual Machine Debug Interface</b>
<b>LOOP</b>	<b>LOOP</b> construct (GRAPNEL language element)
<b>ML</b>	<b>Meta Language</b>
<b>MPI</b>	<b>Message Passing Interface</b>
<b>MTTD</b>	<b>Mean Time To Defect</b>
<b>OCC graph</b>	<b>OCC</b> urrence Graph
<b>OUTPORT</b>	<b>OUT</b> put communication <b>PORT</b> (GRAPNEL language element)
<b>P-GRADE</b>	<b>Parallel Grid Run-Time and Application Development Environment</b>
<b>PVM</b>	<b>Parallel Virtual Machine</b>
<b>RMI</b>	<b>Remote Method Invocation</b>
<b>SEQ</b>	<b>SEQ</b> uential or text block (GRAPNEL language element)
<b>TL</b>	<b>Temporal Logic</b>
<b>TLC</b>	<b>Temporal Logic Checker</b>

# 1 Preface

Correctness debugging of non-deterministic parallel programs is a time-consuming and tedious task, particularly in interactive way. In this case, the software engineers must face the probe effect, the irreproducibility, the completeness problem, and also the large state-space to be discovered during the debugging phase of software development cycle. Moreover, emerging high-performance applications require the ability to exploit heterogeneous and geographically distributed resources as well, which poses new challenges for application development tools.

While the importance of debugging (and testing) is highly accepted in the parallel software engineering domain, there is still a lack of widespread and user-friendly debugging methods and tools. This work attempts to overcome the limitation of existing debugging solutions and combines the traditional debugging methods with automated modelling and formal verification of parallel and metacomputing programs.

In these theses, a highly automated debugging methodology and an experimental toolset are presented relying on the automatic generation of successive consistent global states for parallel programs, called macrostep-by-macrostep execution. In order to improve the efficiency of macrostep-based debugging methodology, two well-funded model checking techniques are introduced in parallel debugging, such as simulation of program by its coloured Petri-net model, and program verification using temporal logic specification. Beside the adaptation of these formal methods into a parallel debugger in order to steer the debugging session towards suspicious situations and detect bugs automatically, adequate optimisation methods were also introduced. As a result, the elaborated parallel debugging framework provides facilities to find and eliminate programming bugs in parallel programs with significantly reduced user interaction both at the implementation and the testing stages.

Finally, the presented debugging methodology has been further developed for dynamically changing, reconfigurable, heterogeneous and distributed, so-called metacomputing systems. For these purposes, the presented new metadebugger is able to invoke third-party debuggers as well as to unify the debugging mechanism for local and remote method invocations.

## 1.1 Introduction

### 1.1.1 Classification of distributed debuggers

According to the literature [1], the distributed debugging methodologies can be categorised according to the level of support they provide to the software programmers concerning the global predicate specification and detection, and the search for the origin/cause of the special software bugs in the distributed program.

In this section, these approaches are briefly introduced based on [1]. The outlined methods begin from the most basic one, and the introduced four approaches are complementary to each other, i.e. each of them attempts to extend and also eliminate the barriers of the previous ones (see Figure 1).

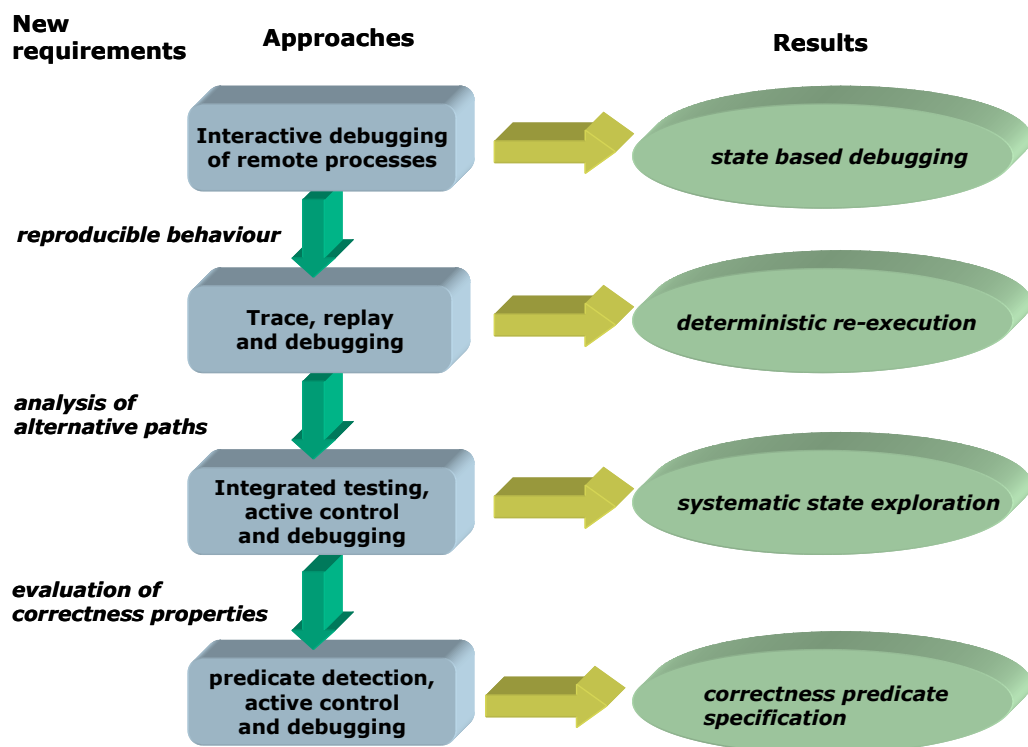


Figure 1 – Classification of distributed debugging methodologies

#### 1.1.1.1 Interactive debugging of remote sequential processes

In case of remote sequential processes the interactive debugging is heavily relying on (or a sort of extension of) the widespread sequential debugging functionalities. This approach enables the online observation as well as the control of the execution of sequential and remotely running processes one-by-one. This method can be used for examining only the local histories of individual processes; it is considered as its main drawback.

The local history describes only the progress of the related process (internal and interaction events), it remains the software developer's task to somehow get

the global “picture” on the entire and distributed computation. The above described remote debugging mechanisms are the most basic ones but required for the more enhanced approaches, and obviously included in all existing commercial and academic distributed debuggers (see Sections 1.1.2 and 1.1.3)..

The nondeterministic behaviour of distributed software is out of the scope in this approach.

### **1.1.1.2 Trace, replay and debugging**

In order to target the complex issue of non-reproducibility, the ‘trace, replay and debugging’ approach collects traces. The trace contains of a set of important (‘relevant’) events generated and collected during the first execution of the distributed program.

The collected trace represents a so-called computation path (i.e. a consistent run) that can be analysed after the execution; if one or more suspicious or erroneous situations are found, then the software developer can run the distributed program again but under the strict control of a supervisory mechanism. During the re-execution, the traced sequence of events is used to force the distributed computation to follow and repeat exactly the same computation path with the same timing conditions. This mechanism helps the user examine the behaviour of the suspicious or erroneous computation path within the usual cyclic interactive debugging session in a reproducible way, similarly to the sequential programs. In such a cycle, the programmer may invoke the remote observation and control facilities from the “*Interactive debugging of remote sequential processes*” approach. The related trace/replay techniques have been in focus of several research teams in the past decades; e.g. the minimisation of the unwanted perturbation (probe effect) and of the large volume of the recorded information (this approach is also used in monitoring systems.). Only a few commercial debuggers and some academic ones support the replay of distributed computations.

From the user’s point of view, there is an important drawback in the “*Trace, replay and debugging*” approach, since it does not provide support for the analysis of other computation paths, only for the actually inspected and monitored/traced path. Moreover, the first run, which is used to collect the trace, is not controlled (i.e. without supervisory mechanism). Therefore, the generated trace represents only a kind of random path from the potentially very large set of possible paths; this mechanism cannot ensure that the first run will be worth to consider for further checking and analysis.

### **1.1.1.3 Integrated testing, active control and debugging**

The “*Integrated testing, active control and debugging*” attempts to tackle the above-mentioned barrier of a simple and passive trace and replay approach. Several authors have described various solutions for the active execution control of distributed programs serving higher level debugging purposes; i.e. provide services to enforce the execution towards specific paths in case of distributed computation in order to increase the chance of finding the location of erroneous situations. The proposed solutions are different ones concerning the method they produce the desired computation path (i.e. consistent run), which is to be followed during a controlled execution. In the remaining part of this subsection, one of these approaches is outlined.

The approach has two distinguished phases during the distributed debugging activity. The first one is a sort of integration of static analysis and testing phases (*SAT* phase). The second one is the integration of dynamic analysis and debugging stage (*DAD* phase).

The main aim of the *SAT* phase is to help the user generate interesting computation paths (consistent runs) that e.g. cause deadlocks, erroneous situations, i.e. exhibit violations of correctness properties in general. Unfortunately the *SAT* phase cannot be automated in the most cases but exploiting the intuitions of the software developers through interactive testing tools can be useful. The interactive tool can cooperate with the user to define and fine-tune the conditions and code regions of distributed program that are to be considered for analysis. Later the *DAD* phase is used to generate a sequence of special commands that will be used to steer the distributed program execution in order to follow the paths specified by the above testing scenarios. Such a distributed program execution can be the subject of the ‘trace, replay and debugging’ approach, and also integrated in a cyclic debugging session.

The main benefit concerning this methodology is that it allows the software developer to switch freely between the *SAT* and *DAD* phases, and these phases can be repeated until the user becomes convinced about the satisfaction of the currently inspected correctness properties. Moreover, this approach combines the benefits of both static and dynamic analysis to assist the user to understand the complex behaviour of the investigated distributed program.

The main limitation of this approach is that the user may leave out relevant scenarios when he/she specifies and generates them, which will not be tested and analysed. Therefore, there is no full guarantee that no important situations were ignored or unrecognised.

#### **1.1.1.4 Automated detection of global predicates, active control and debugging**

The main goal of the “*Automated detection of global predicates, active control and debugging*” approach is to help the user increasing the confidence on the outcome of the previous approach. That is why, it allows the user to specify the correctness criteria using so-called global predicates. These global predicates are automatically evaluated by special detection algorithms.

The efficient evaluation of global predicates is limited to a subclass of predicates, but this approach can be considered as a useful complementary to the “*Integrated testing, active control and debugging*” approach. According to [1], “their integration seems a promising research direction to improve distributed debugging”. The presented work is a step towards this direction.

### **1.1.2 Commercial tools**

#### **1.1.2.1 Distributed Debugging Tool (DDT)**

The Distributed Debugging Tool (DDT) [65], developed by Allinea, is one of the most sophisticated graphical debuggers for parallel programs on the market. DDT can be used as a single-process or a multi-process (MPI) program debugger and basically follows the “*Interactive debugging of remote sequential processes*” approach but it also offers some advanced features.

Multi-process DDT allows the construction of user-defined groups to manage and apply debugging related operations to multiple processes at the same time. In multi-process mode, the user is able to set a breakpoint for every member of the current process group. The software developer may add a condition to any of the breakpoints by inserting an expression (local predicate) that evaluates to true or false; these are the so-called conditional breakpoints. Each time a process (in the group the breakpoint is set for) passes this breakpoint, the debugger evaluates the condition, and breaks the process only if it returns true.

Another useful feature of DDT is the synchronization of processes, i.e. if the processes in a process group are stopped at different points in the code and the user wants to re-synchronize them to a specific line of code. In this case, all processes in the selected group start running and DDT places a breakpoint at the line where the user wants to synchronize the processes, but ignoring any other breakpoints that the processes may reach before they have synchronised at the given line. (A major drawback of this solution is that if the software developer wants to synchronize the code at a point where all processes cannot reach then the processes, which cannot get to this point, will run to the end without any notification.)

By means of conditional breakpoint sets and synchronization capabilities, the software developer may control the parallel execution towards suspicious situation and replay the application manually but it may require much effort from the user. Hence, these features of DDT offer some limited and user-level support for the “Trace, replay and debugging” and “Integrated testing, active control and debugging” approaches.

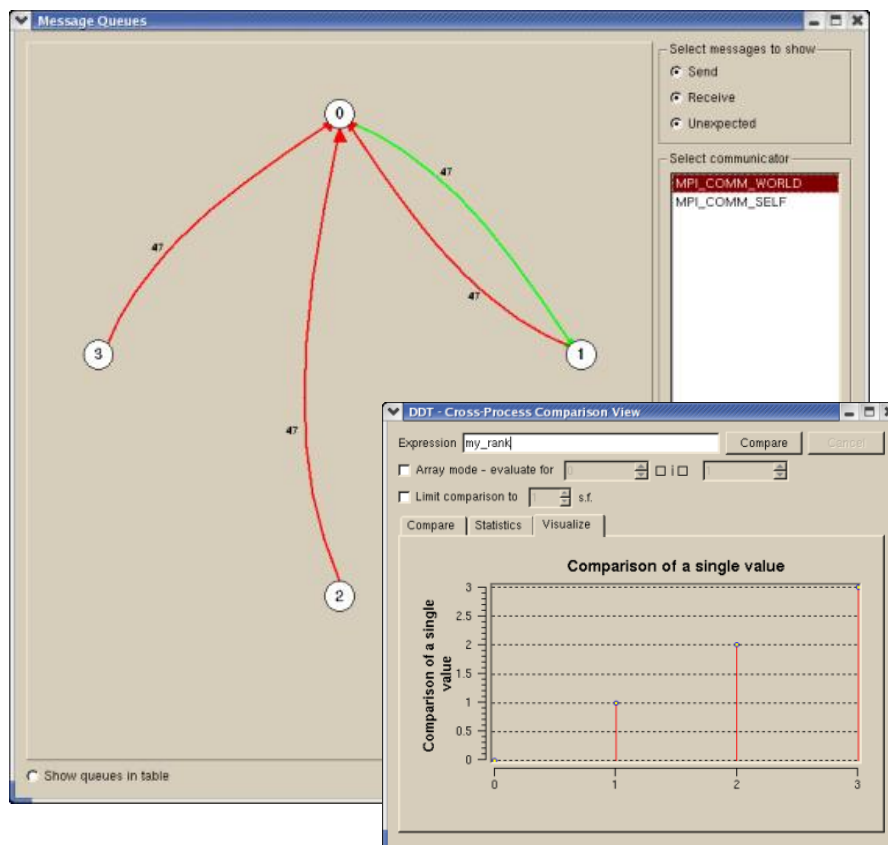


Figure 2 – Message Queue and Cross-process comparison in DDT

Message Queues (see Figure 2) can be also visualised but it stops all running processes (in asynchronous way) while DDT is obtaining data. The software developer may select an MPI communicator to focus the messages in that group, and the ranks displayed in the diagram related to the communicator. Different colours are used to display messages from three different message queue types according to the User's manual:

- Send Queue: represents all the outstanding send operations
- Receive Queue: represents all the outstanding receive operations
- Unexpected Message Queue: represents messages that have been sent to the current process but have not been received

The so-called *cross-process comparison* in DDT (see Figure 2) is a possible solution for the automatic detection of global predicates introduced in the “*Automated detection of global predicates, active control and debugging*” approach. The cross-process comparison window can be used to analyse expressions calculated on each of the processes in the current process group. The information is displayed in three optional ways according to the User's manual:

- Grouped by expression value
- Statistically – it means maximum, minimum, variance and similar with a plot which displays the maximum, minimum and interquartile range graphically
- A plot of values. In the case of a vector (one-dimensional array) expression, the plot of values will display a line graph of values for all processes.

### 1.1.2.2 Totalview debugger

One of the most widespread debugger tools at the market is Totalview debugger [38] from Etnus that provides similar functionalities to DDT; e.g. barrier points. Barrier points are similar to the traditional simple breakpoints, but differing in that the software developer can use them to synchronize a group of processes (or even threads). A barrier point holds *each* process (or thread) that reaches it until *all* processes (or threads) reach it.

The latest version of the TotalView debugger have been extended by graphical facilities including a Message Queue Graph (a snapshot of pending MPI messages) and a call tree diagram within a process. The independent TimeScan Multiprocess Event Analyzer seems to be useful for performance analysis by visualizing events. However, while TotalView and associated frameworks are very popular in parallel computing environments, these tools provide solutions only for the “*Interactive debugging of remote sequential processes*” approach and some limited facilities for “*Trace, replay and debugging*” by means of its checkpointing tool. On SGI IRIX and IBM RS/6000 platforms, the user can save the state of selected processes and then use this saved information to restart (i.e. resume) the processes from the position where they were saved. But processes running remotely that communicate by using sockets can have difficulties when they are checkpointed because IRIX will not checkpoint programs with open sockets.



Figure 3 – Evaluation points in TotalView

An evaluation point (see Figure 3) in Totalview debugger is a breakpoint together with a given code fragment. When a thread or process reaches such evaluation point, it executes the given code. The software developer can use evaluation points in several ways, including conditional breakpoints, thread-specific breakpoints, countdown breakpoints, and patching code fragments into and out of the program. Hence, this function in co-operation with barrier points can form a base of “*Integrated testing, active control and debugging*” and “*Automated detection of global predicates, active control and debugging*” approaches but the user is responsible for writing these code segments, and this is another way of inserting new programming bugs into the debugging cycle.

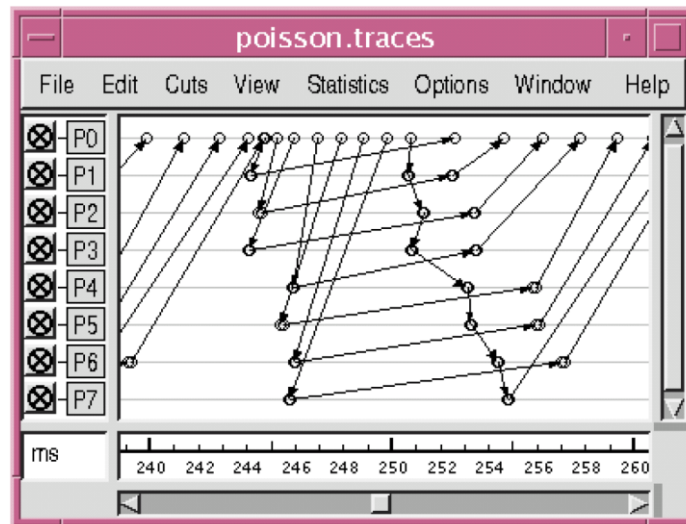
The Action Point Properties window (see Figure 3) in the foreground uses an evaluation point, a programming language statements and a built-in Totalview debugger function to stop a loop every 100 iterations (it also prints out the value of variable *i*). Another example in the background of Figure 3 stops the program execution every 100 times a statement gets executed.

### 1.1.3 Non-commercial debuggers

#### 1.1.3.1 MAD environment

GUP/Linz developed the record & replay tool NOPE (NONdeterministic Program Evaluator) [29] for testing and debugging of nondeterministic message passing programs within the MAD environment [2]. The operation of NOPE is split into a record (or trace) phase and a replay phase as it was described in “*Trace, replay and debugging*” approach. In this way, the initial tracing phase is responsible for

generating trace files containing only the ordering of critical events and later during replay these trace files are used to enforce exactly the same event ordering as occurred during the initial tracing phase, which ensures equivalent execution (see Figure 4).

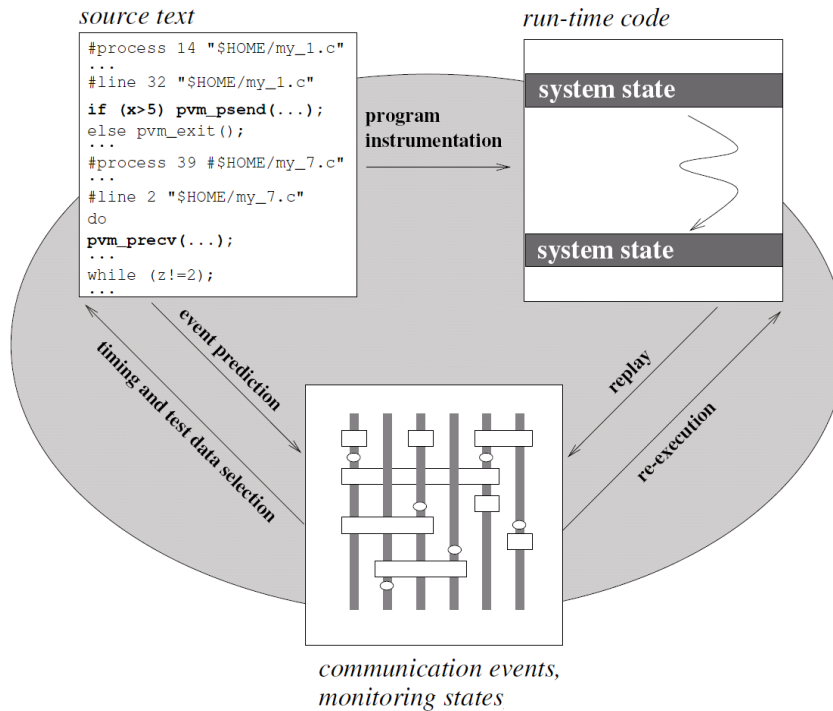


**Figure 4 – Example event graph visualization of a program execution**

Additionally, complete traces can be generated to serve as input for the post-mortem analysis tool [66] if necessary. In order to minimise the unwanted perturbation (probe effect) of monitoring code segments execution, the monitor instruments only receive operations and produces trace records only for wildcard receive events (i.e. for non-deterministic choice points). These data are sufficient to produce complete traces with exhaustive information about the execution during subsequent replay phases. Furthermore, the sophisticated automatic race condition detection and manipulation features in NOPE allows programmers to evaluate other possible program runs for the supplied input. This mechanism tries to find all occurrences of wildcard receive operations and evaluates ‘race candidates’ in the complete traces. The race candidates are messages that have chance to be accepted by a given wildcard receive. As the next step, additional replays have to be started in order to uncover previously hidden runs. The basic idea is the following: at each wildcard receive operation every possible race candidate must arrive first during one run. NOPE is able to change automatically the ordering with its event manipulation mechanism and to start a replay phase for each permutation. Then, the complete set of replays delivers all possible runs that are directly connected to the original program execution. The evaluation of possible race candidates has to be repeated iteratively, in order to detect indirectly connected runs as well. Hence, the MAD environment provides a solution for almost all debugging approaches except for “*Automated detection of global predicates, active control and debugging*” approach, which is under development in TLC [33].

### 1.1.3.2 DDBG and STEPS

DDBG [68] directly supports the “*Interactive debugging of remote sequential processes*”, but it can also support all other methodologies through tool integration; DDBG and the STEPS testing tool [67] has been integrated in order to explore systematically the state of C/PVM programs.



**Figure 5 – Level of analysis in STEPS**

As Figure 5 depicts in large, a structural testing methodology utilised by STEPS consists of three levels of representation of parallel programs: dynamic program states (by the program run-time code), decision statements (specified by the program source code), and communication events (in which corresponding processes may be involved).

In STEPS structural testing splits into complementary ‘streams’, which targeting their given issues at various representation levels, and they build a coherent framework together. The basic approach is to exploit not only static but also dynamic analysis to support each other, in order to make simpler the targeted particular activities of every stream. These streams are shown with arrows and connect the three program representation layers (see Figure 5).

In order to inspect a particular behaviour exhibited by a parallel program, the software developer needs assistance to detect communication actions during the program execution. The necessary event detection is based on log files and requires two basic steps: monitoring of communication actions and searching the entries in the event log files. STEPS is able to perform event prediction based on the program source text. In this way, it can identify special monitoring states requiring detection during runs. Transitions between monitoring states indicate at what communication actions the special monitoring probes (breakpoint traps) shall be installed.

STEPS is able to support interactive selection of paths through individual component processes and implements symbolic interpretation that mimics a real program execution. The software developer may determine timing (order of events) between selected paths and also select test data that satisfy entry conditions for selected paths. Each test scenario designed in this way is subsequently implemented by STEPS in a form of a so-called test scenario script. The executable

program code can be dynamically instrumented according to the predefined scenario — breakpoints are inserted and some testing procedures associated with them similarly to software interrupt handlers. These testing procedures can obtain processes with data and enforce the execution of communication actions with specific timing conditions.

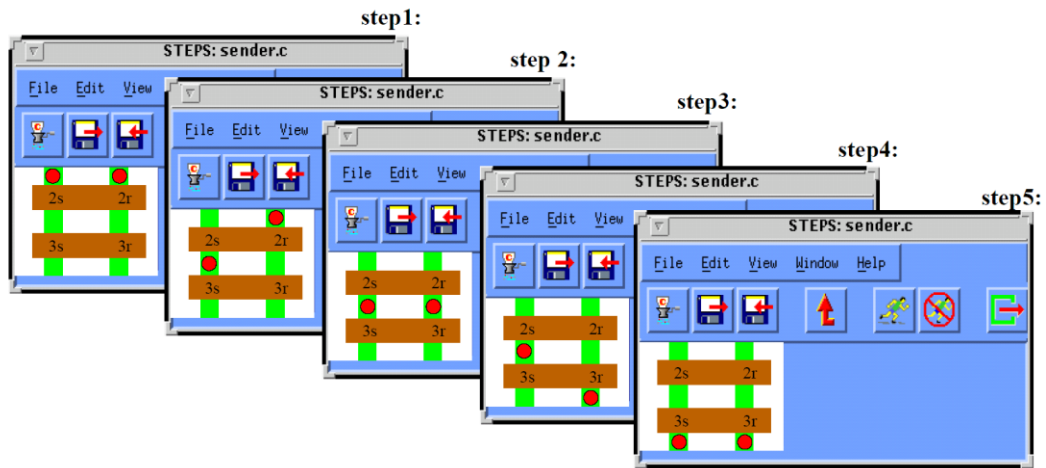


Figure 6 – Step by step replay of states from the information log

A program under test is launched and executed according to the test scenario specified by the software developer. The first scenario designed by the developer is very often only an initial one from the whole “cycle”. It usually a random execution of the instrumented program code (as it was described earlier) with any input data provided by the software developer. During execution, the program behaviour is traced to the event log for further inspection and analysis, and also serves as input for designing new test scenarios. Events are retrieved from log files and replayed visually on the screen. According to [68] the static analysis of the program source code and the dynamic analysis of its run-time code representation enable not only stepping back and forth through the program dynamic states and inspecting/modifying variables of individual processes but modifying the order of transitions in communication operations as well. These procedures involve re-execution of the program binary code from some initial state to a specific suspicious or interesting program state, and are performed interactively by the software developer. It also allows the developer to design new testing scenarios based on previously executed ones and even prototype these new scenarios before program execution.

According to described features, DDBG and STEPS give solution for almost all debugging approaches except for “*Automated detection of global predicates, active control and debugging*” approach, which must be done manually or using additional user-tailored tools for analysing the generated log files.

### 1.1.3.3 P2D2 debugger

P2D2 [39] from NASA is a portable debugger with a visualization tool and is appropriate for debugging Globus/MPICH applications. P2D2 supports the “*Interactive debugging of remote sequential processes*” and gives limited facilities for active control of processes by means of controls sets of processes (defined by elements/rows/columns of process grid) and focusing mechanism (maximum of 4

arbitrary processes). Process control operations allow the software developer to launch and stop processes and insert/remove breakpoints in each process of the controlled set. The focus mechanism is most useful when the software developer intends to exploit the built-in Data Viewer and the he or she wishes to compare values between two (or more) processes.

The most important feature for examining current state in a large number of processes is that of user configuration of the process grid (see Figure 7, upper left corner). This allows the developer to get an overview, a ‘big picture’ of all of the processes. By means of this feature, the software developer can specify how processes are to be depicted in the process grid. The first option selection is the corresponding icon, and the second one is the (local) predicate to be evaluated (see Figure 7, middle in the window). For some predicates such as `Eval()`, the user must also enter a string as a parameter in a text box (e.g. “`me % 2 == 0`”). Hence, P2D2 gives some facilities for detection of global predicates.

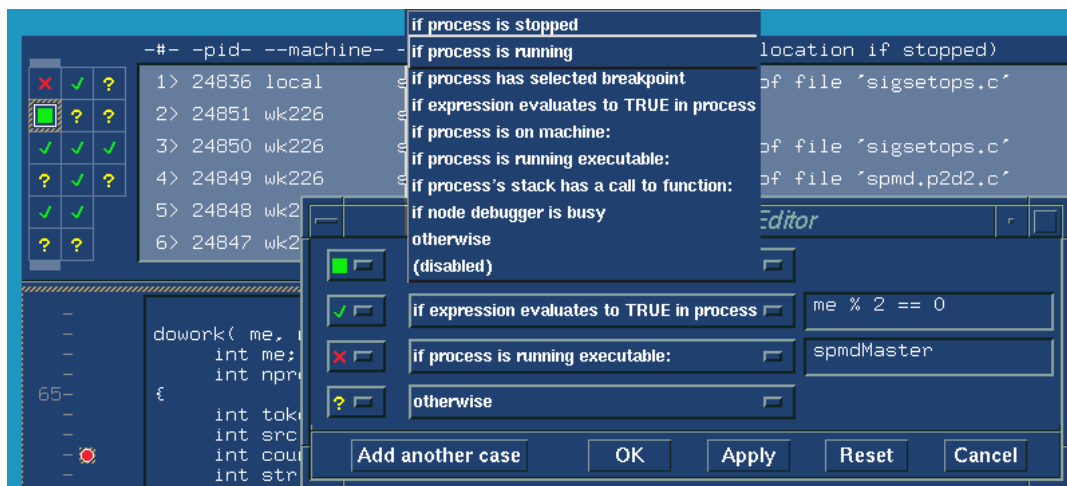


Figure 7 – Sample debugging scenario with P2D2

### 1.1.3.4 Debuggers for JAVA

Several debugging systems exist for Java-based environments; I briefly list a representative cross section and highlight the main features of each.

*Jdb* permits remote debugging (based on JPDA) and *Javadt* extends *Jdb* with a simple graphical user interface [43]. These tools permit non-local programs to be debugged, and they are typical representatives of “*Interactive debugging of remote sequential processes*” approach since they do not provide coordinated facilities for distributed debugging.

*DejaVu* for Jalapeno JVM [36] is a debugging tool that can deterministically replay the non-deterministic execution of multithreaded Java applications thus it a good example for “*Trace, replay and debugging*”. This tool also provides a GUI for visualizing program execution with nearly all the functionality of traditional sequential debuggers, but it cannot be used for distributed applications and runs only on IBM AIX.

*DJVM* [37] is a replay tool for distributed socket-based Java applications that could be a good basis for a distributed debugger. However, to the best of my knowledge, there is no available debugger tool using DJVM.

### 1.1.4 Comparison of debugger tools

<i>Tools</i>	<i>Approaches</i>			
	Interactive debugging of remote sequential processes	Trace, replay and debugging	Integrated testing, active control and debugging	Automated detection of global predicates, active control and debugging
<b>DDT</b>	✓	✍ conditional breakpoints and synchronization	✍ conditional breakpoints and synchronization	✍ cross-process comparison
<b>TotalView</b>	✓	✓ checkpoint on SGI/IRIS & IBM/AIX	✍ barrier and evaluation point	✍ evaluation point
<b>MAD</b>	✓	✓ with NOPE	✓ with NOPE	✗ under development
<b>DDBG &amp; STEPS</b>	✓ with DDBG	✓ with STEPS	✓ with STEPS	✍ evaluation functions, off-line analysis
<b>P2D2</b>	✓	✗	✍ control sets	✍ custom grid display editor
<b>JDB &amp; JAVADT</b>	✓	✗	✗	✗
<b>DejaVu</b>	✓	✓ multithreaded applications on IBM/AIX	✗	✗
<b>DJVM</b>	✗	✓ socket-based applications on IBM/AIX	✗	✗
<b>P-GRADE</b>	✓ with DIWIDE debugger	✓ with macrostep engine	✓ with GRSIM simulator	✓ with TLC temporal logic checker

Table 1 – Comparison of commercial and academic debuggers

Notes:

- ✓ Fully supported
- ✍ Limited support: the user is responsible for appropriate use of the enrolled basic features
- ✗ Not supported

The Table 1 compares some of the most sophisticated debugger tools according to their supports for the four main debugging approaches (see Section 1.1.1). At the end of the table the P-GRADE environment is also enrolled that tries to give user-friendly and automated solutions for all approaches by means of its integrated tools; such as DIWIDE distributed debugger, macrostep engine, GRSIM simulator and TLC temporal logic checker engine. The basic idea and the debugging cycle is described in details in 1.2.

## 1.2 Proposed debugging cycle in P-GRADE

The framework of P-GRADE parallel programming environment has been chosen for the development of an enhanced debugging methodology that basically follows the already presented “*Automated detection of global predicates, active control and debugging*” approach (see Section 1.1.1.4). One of the main advantages of P-GRADE framework [10][50] from debugging aspects was that the designed parallel applications are constructed based on the syntax and semantics of the GRAPNEL hybrid programming language. GRAPNEL provides language elements to express graphically the parallelism, the distribution, concurrency, and communication between processes at different hierarchical design levels, while the sequential code can be inherited from legacy sequential applications. Hence, the automatic generation of formal models and consistent global states for GRAPNEL programs are more feasible comparing to generic message-passing (or parallel) programs.

During the extension of the debugging capabilities of P-GRADE, the main goal was to support the following mechanism (see Figure 8).

First, when a *GRAPNEL program* is compiled successfully [22], the *GRP2cPN* tool generates the coloured Petri-net model (*cPN-model*) of the program based on its GRAPNEL code (see Section 2.1), and the *partitioner* is responsible for slicing the application into smaller parts (see Section 3.2.3.1) if possible. If the user specified the correctness properties (i.e. the expected program behaviour) of the GRAPNEL program with a temporal logic specification (*TL specification*), a transformation tool, *TL2Class*, separates it to a high-level specification with temporal logic operators (*SpecClass*), which can refer to the low-level atomic predicates (*PredLib*).

In the next phase, a distributed debugger tool, *DIWIDE*, in co-operation with the temporal logic model checker, *TLC engine*, compares (see Section 3.1) the specification with the observed program behaviour in each generated consistent global state, i.e. macrostep-by-macrostep (see Sections 2.2 and 2.3). Meanwhile, an

off-line or on-line Petri-net simulator, *GRSIM* steers the traversing of the state-space towards suspicious situations (see Section 3.2.2), which can be found by the help of its built-in analyser or by the invocation of *TLC engine* if the appropriate (simplified) specification of the program is available.

If an erroneous situation is detected, the user is able to inspect either the consistent global state of the application or the state of the individual processes as well (*Graphical User Interface*). Depending on the situation, the user may fix the programming bug by the help of *GREd editor* [21] (designed for GRAPNEL applications), or replay the entire application to get closer to the origin of the detected erroneous situation. Finally, the entire debugging cycle can be completed if the program's reliability achieves the satisfied level, which is estimated by the help of *Rayleigh-analyser* (see Section 3.2.3.3) based on the logged error reports collected in a database (*DB*).

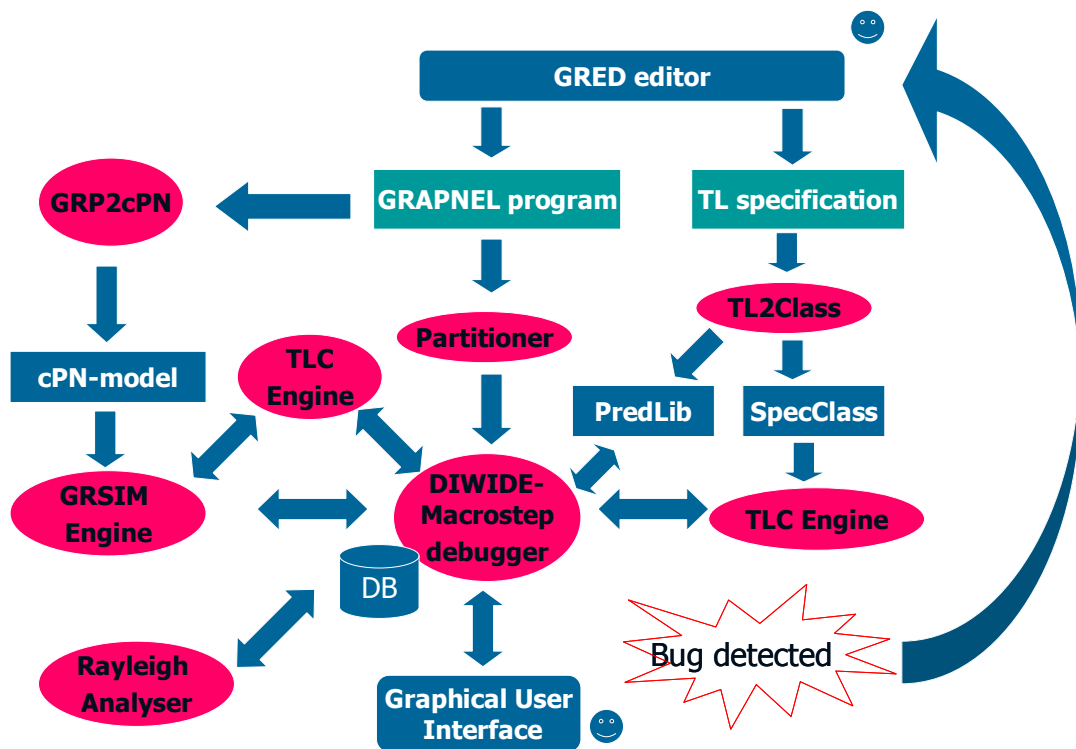


Figure 8 – Proposed debugging cycle in P-GRADE framework

The presented debugging framework attempts to give efficient facilities for the specification of desired correctness properties as well as for the automatic detection of erroneous or undesired situations corresponding to programming bugs, but they require the framework to support the following requirements:

1. A language to specify the correctness predicates.
2. Algorithms to evaluate the correctness predicates automatically.
3. Observation and control mechanisms to allow the user to observe the computation states corresponding to the detected erroneous situations.
4. Steering and optimisation mechanisms to reduce both the user interactions and the time of the debugging period.

The proposed solution is presented in details in Sections 2 and 3, and its generalisation towards metacomputing environments in Section 4. Please note that the following subsections do not reflect my own research achievements. They mostly serve as introductions to my theses:

- 1.1.1 Classification of distributed debuggers
- 2.1.2.1 GRAPNEL language
- 2.1.3.1 Short overview of coloured Petri nets
- 2.2.2 Macrostep-based execution and Execution Tree
- 3.1.2.1 Theoretical background for temporal logic assertions
- 4.1.2 Introduction (Metacomputing)
- 4.1.3.1 The Harness Metacomputing System
- 4.1.3.2 Java Platform Debugger Architecture

## 2 Macrostep-based debugging technique for GRAPNEL applications

### 2.1 The formal description of GRAPNEL programs for debugging purposes

#### 2.1.1 Preface

To provide high-level graphical support for the development of parallel programs, an integrated programming environment, P-GRADE [10][20][21][27][50] is being developed by MTA-SZTAKI.

P-GRADE provides tools to construct, execute, debug, monitor and visualise message-passing based parallel programs. P-GRADE also provides a hybrid graphical language and a user interface that hides the low-level details of the underlying message-passing system thus; it allows the developers to concentrate on the important aspects of parallel program development such as task decomposition.

In P-GRADE development environment, the parallel applications can be constructed by the GRED graphical editor based on the syntax and semantics of GRAPNEL hybrid programming language. GRAPNEL language provides language elements to express graphically the parallelism, distribution, concurrency, and communication between processes using three hierarchical design levels.

At the top level, called *application level* (or inter-process communication level) the outline of the whole application is described graphically with respect to communication connections among the processes. The *process internal level* is used for describing the inner structure of the individual processes graphically using a control flow-like technique, which describes the message passing related parts of the process. At the lowest level of GRAPNEL code, called *textual level*, the developer can define textual C code fragments representing the actual contents of the graphical icons defined at the process internal level.

In this thesis, I introduce new limitation rules and language elements in GRAPNEL hybrid language in order to automatically create hierarchical coloured Petri-net model [12][13] based on this description for debugging purposes. Then, my transformation methods for each element of this new GRAPNEL\* language are described by patterns of coloured Petri-net (i.e. CPN subgraphs). Based on these transformation steps, I prove that *a coloured Petri-net model exists and can be generated for any GRAPNEL\* programs for debugging purposes*.

#### 2.1.2 Introduction to GRAPNEL and GRAPNEL\* language

##### 2.1.2.1 GRAPNEL language

The GRAPNEL programming model is based on the message-passing (MP) paradigm [10]. The programmer can define processes, which perform the computation independently in parallel, and interact only by sending and receiving messages among themselves.

### ***2.1.2.1.1 The Process Model***

A process can be either a single unit or a member of a process group. Similarly to the Message Passing Interface (MPI), the process group is an ordered collection of processes, and each process is uniquely identified by its rank number within the group. Process groups can be used in two important ways. Firstly, they can be used to specify which processes are involved in a collective communication operation, such as a broadcast. Secondly, they can be used as an abstraction mechanism to support the structured design at the level of processes, i.e. processes which perform logically the same task, or a more or less independent subtask, can be put physically into a group to be managed together. Since the process groups can be nested (a group may contain further groups), they support the hierarchical design of the distributed application and facilitate to locate communication related errors during the testing and debugging phase. Recall that PVM [46] supports the process group concept and hence GRAPNEL was also designed to support this concept.

PVM supports dynamic process creation but MPI-1 does not. The Occam experience showed that static process creation is sufficient to program any parallel algorithm and hence for the sake of simplicity GRAPNEL was designed as a static language.

To manage the most often used regular process topologies, predefined topology templates are supported by GRAPNEL. In these regular arrangements only the representative process types of the topology must be defined by the user, the arrangement itself is generated automatically at runtime based on size parameter(s). This feature also supports to test and debug the application in a relatively small size and then the homogeneous topology<sup>1</sup> can be expanded without violating the correctness of the program.

### ***2.1.2.1.2 The Communication Model***

Communications among processes are either point-to-point or group communications, and can be synchronous or asynchronous ones. Communication operations always take place via communication ports which can belong to either processes or groups, and which are connected by channels. Every port of a process has its own protocol to ensure that the form of the transmitted data is the same at both the sender and receiver sides. The programmer must define the protocol explicitly by writing the appropriate list of data types to be delivered by the port.

Inside a process the Communication Input and Output Action nodes represent the two fundamental types of the communication operations which can be characterized by two features:

1. How many ports of the process are involved in the operation (one or more)
2. What type of operation is applied

The following different types of operations are available:

- a) Simple input or output action (CAI or CAO). One or several ports participate in the communication, so data can be received from or sent to one or several other processes (or process groups).

---

<sup>1</sup> Homogeneous means that processes which constitute the given regular topology have the same code (i.e. SPMD code)

- b) Alternative input action (CAIALT). Several ports participate in the input operation, so data can be received from several other processes, but only one message must be selected from the alternatives.

The first type corresponds to the usual point-to-point send-receive instruction pairs applied in general in the MP systems. The second type represent an extremely important higher level operation, it corresponds to the ALT construct of Occam and the wild-card message tag of PVM and MPI.

Collective communication operations can be performed by defining special compound ports to a process group. Via these group ports, all members in the group can be accessed by an outer process. Applying this technique the following types of collective communication operations become available:

- Broadcast ( $1 \rightarrow n$ ). The outer process can send the same data to each process in the group.
- Scattering ( $1 \rightarrow n$ ). The data sent by the outer process is scattered among the processes of the target group, i.e., each process in the group receives its own different data from the sender process.
- Gathering ( $n \rightarrow 1$ ). Every process in the group sends its own different data to the receiver process.
- Reduce ( $n \rightarrow 1$ ). A reduce port performs global computations like choosing the minimum from the values sent by the member processes, or computing the sum of them, etc.

#### **2.1.2.1.3 The Process**

In GRAPNEL the most fundamental unit of the language is the process which manifests at two levels:

- Application Layer (see Figure 9)
- Process Layer (see Figure 10)

Every process has two graphical views: one for describing the communication connections (ports and channels) to the other processes (Application Layer), and one for describing the internal structure of the process (Process Layer).

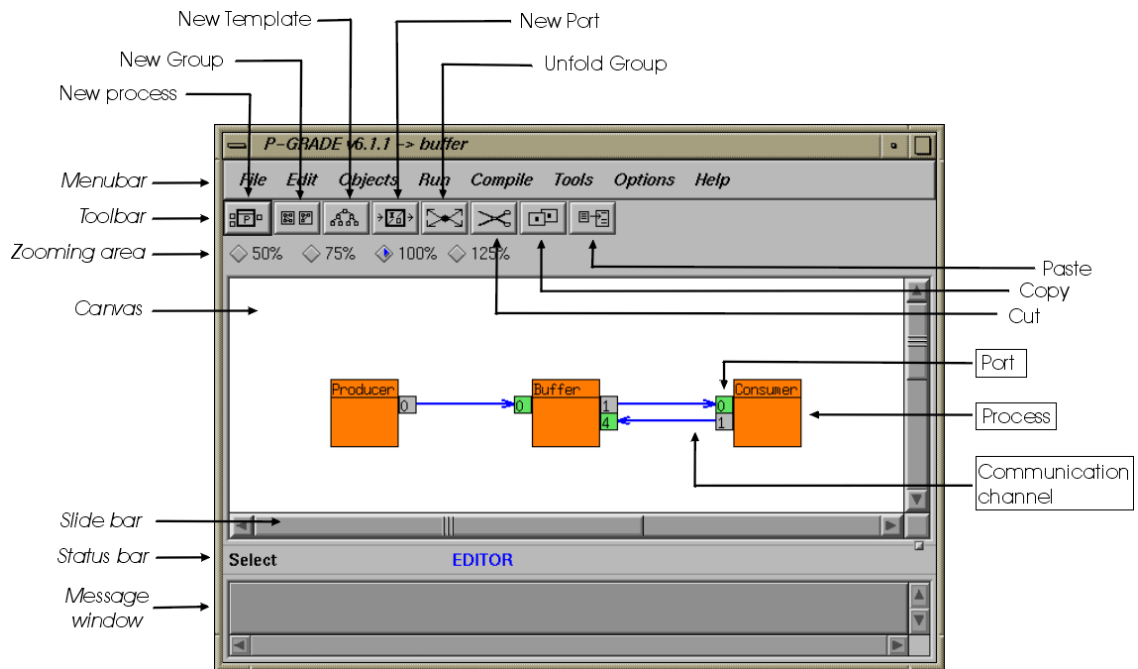
#### **2.1.2.1.4 The Application Layer**

At this level a process is represented by a rectangle icon with caption (see Figure 9). Attached to the edges of the process icon several small rectangles are used to represent the communication ports. Each port must be connected to a port of another process (or process group) by directed arcs (channels) representing the direction of message transfers. If there is an arc between two processes (or groups), they can communicate with each other during the execution of the program, otherwise, they cannot. There are three types of communication ports due to the direction of the data (see Figure 9):

- *INPORT*: input (represented by green boxes)
- *OUTPORT*: output (represented by grey boxes)

- *INOUTPORT*: inout<sup>2</sup> (represented by half green - half grey boxes)

Each port has its own *protocol*, which should ensure that the structure of the transmitted data always consistent concerning the sender and the receiver processes.



**Figure 9 – Language elements of GRAPNEL and editor functions at application level**

The protocol is defined by a list of data types. The data structure of the message transferred by the port should match to this list of data types. For example, the list `int[5];float[3];` represents a protocol in which first, five integer numbers and then three numbers should be packed in the message. If two ports are connected by a communication channel, their protocols should be the same and this equivalence is checked by the compiler.

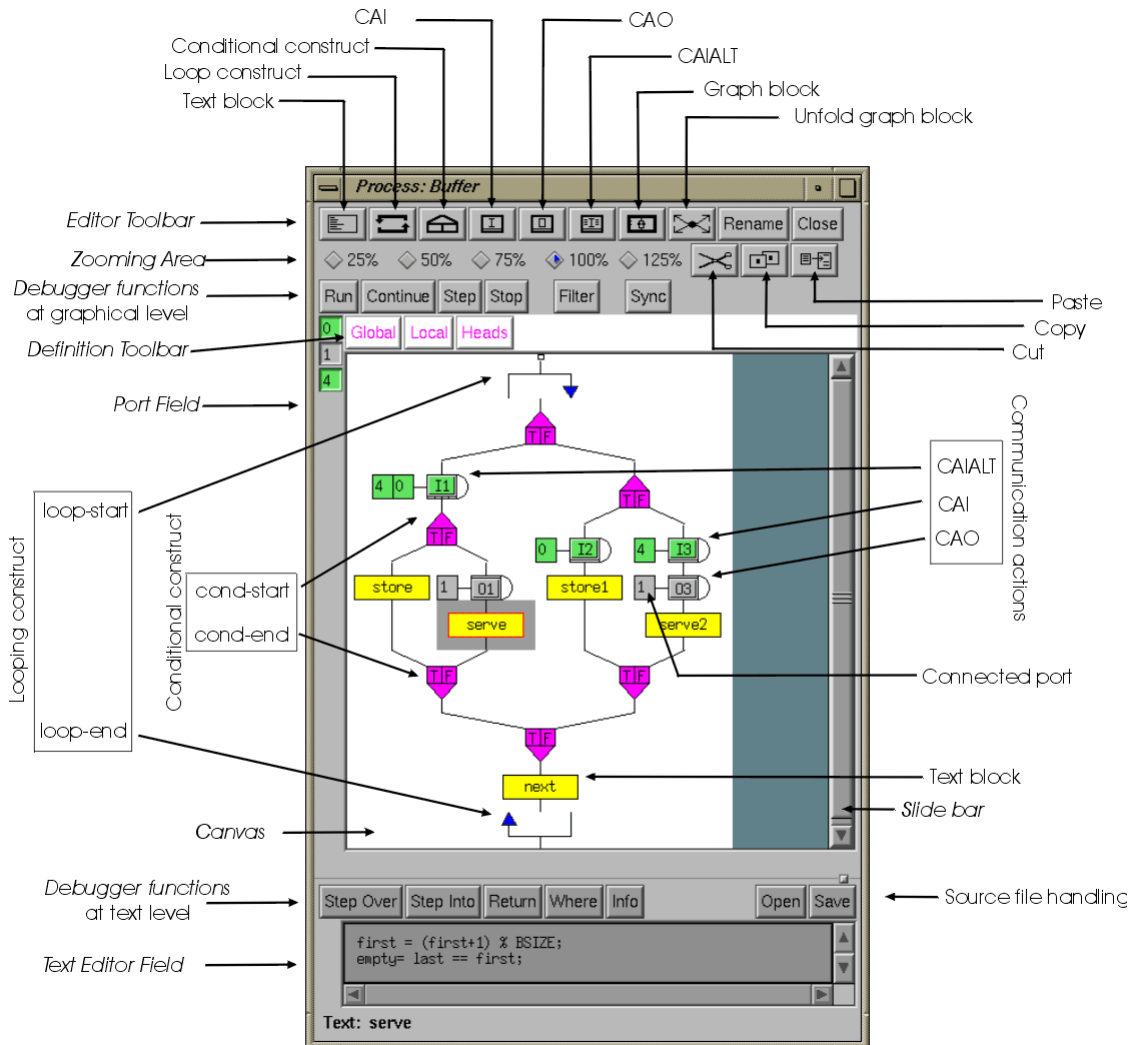
### 2.1.2.1.5 The Process Layer

The internal structure of any process can be described at the Process Layer using graphical symbols to represent the control. Only those parts of the control must be specified in a graphical way that are related to communication operations. The point is that all communication instructions are represented by icons thus, if communication occurs in a specific branch of a conditional construct than the whole conditional construct must be defined graphically. Similarly, if communication is needed inside a loop, the whole loop construct must appear in a graphical way. Graphical code of an example process is shown in Figure 10.

These drawings can seem to be unnecessary or uncomfortable for those who are experienced in ordinary textual programming, but it takes the great advantage of shortening dramatically the debugging phase of program development. Using these kinds of graphical views of processes the programmer

<sup>2</sup> The inout port is simple a graphical abbreviation of an input-output ports couple.

can track the whole application much more efficiently and can locate communication related errors faster than debugging the textual source code line by line.



**Figure 10 – Language elements of GRAPNEL and editor functions at process level**

Each graphical symbol possesses a corresponding piece of textual C/C++ code fragment to define the low level semantics of that symbol. The available icons to construct the graphical code of a process are listed as follows.

**Loop Construct.** This icon is used for any loop constructs in the program e.g. “for”, “while” and “do” structures (Figure 10). The icon consists of two arrows wrapping the body of the loop like parenthesis. The loop condition should be defined at the begin loop icon (*loop-start*) in the case of ‘for’ and ‘while’ loops and at the end loop icon (*loop-end*) for ‘do’ loops.

**Conditional Construct.** The icon of conditional constructs is used for any conditional structure like “if” or “switch” (Figure 10). The conditional expression should be defined on the upper triangle of this icon.

**Communication Action.** Inside a process, the Communication Action (CA) symbols are used to perform input and output operations via the appropriate ports. The CA symbols can be divided into two fundamental groups:

- Communication Action Input (CAI) and
- Communication Action Output (CAO).

Actually, the graphical symbols for input (CAI) and output (CAO) are very similar and - in the simplest case - consist of two icons (a main and a satellite one) (Figure 10). The main icon represents the sending or receiving actions of the communication, while the satellite icon represents a port of the process (which is defined at the Application Layer). The satellite icon (i.e. connected port) defines the communication protocol and the main icon defines when to use the protocol and on which data. It is possible to use both synchronous and asynchronous output communication actions.

In GRAPNEL it is possible to express input/output operations from/to several processes simultaneously. In this case, the main icon has more satellite icons denoting that the process receives data from or send data to all processes connected to the ports represented by satellite icons. In the text code belonging to the main icon the following expression must be defined for each port denoted by satellite icons:

```
PORT port-identifier : data-description;
```

So the port must be identified first – because there are more ports connected to the main icon – and then the variables are defined where the data should be stored or should come from (i.e. data-description).

GRAPNEL contains an additional special CA symbol, called Alternative Communication Input Action (CAIALT, see Figure 10). In the case of Alternative CAI the main icon has several satellite icons, so the process can receive data from several other processes but only one message must be selected from the alternatives. In the text window belonging to the main icon, the programmer must define a similar expression for each port denoted by satellite icons like in the case of an ordinary CAI, but now they can contain an additional field to define guard conditions:

```
PORT port-identifier : GUARD logical expression: data-description;
```

If there are several ports whose guard conditions are satisfied (the logical expression is evaluated to true) and on which a message appears, the selection will be *nondeterministic*.

**Blocks.** Blocks are used to support the structured program design at the Process Layer. They own code fragments, which can be two different types:

1. Text Block: The code fragment is written in textual way (in C/C++)
2. Graph Block: The code fragment is drawn using graphical symbols

If the code fragment, the user wants to put into a block, does not contain any CA (Communication Action) the Text Block<sup>3</sup> can be used (see Figure 10) otherwise, the Graph Block must be applied. Graph Blocks can be nested and they have the same satellite icons as it can be seen at the CA symbols because in this

---

<sup>3</sup> The Text Block is also called SEQ (sequential) block in GRAPNEL terminology.

case the Graph block itself denotes the sending and/or receiving operations at that level.

#### **2.1.2.1.6 Data Declarations and Definitions for Processes**

**Global** (Global Data): The programmer can declare and define global variables, types or constants according to the C syntax as text code belonging to Global section (see Figure 10). The term “global” means that data defined/declared here can be referred in every piece of text code belonging to the particular process, even if it is in a completely separate C file. Text code of the Global section may contain external declarations according to the standard C syntax if someone wants to refer to or use existing data or functions, respectively, which are defined in external C files or libraries.

**Local** (Local Data): In this case, the programmer can define local data in the text code of Local section (see Figure 10). “Local” means that the scope of these variables is restricted to the text code fragments belonging of the visual instructions defined of the particular process, and they can not be accessed in separate C files.

**Heads** (Include Files): The “Heads” section (see Figure 10) is used for defining C *#include* directives as text code in order to use predefined C code or data located either in libraries or in separate C files.

### **2.1.2.2 GRAPNEL\* language**

#### **2.1.2.2.1 New language elements in GRAPNEL\***

For the automatic model generation some new language elements are introduced in GRAPNEL\*. Without these new elements the model generator should parse the sequential C or C++ code to find the variables which steer the execution (i.e. which are referenced from loop and conditional constructs). Often this kind of parsing is not feasible in generic C/C++ code due to pointer references that cannot be always resolved statically before the actual execution of the program, etc. That is why, the user is in charge to separate the code in two parts using the following new language elements:

**Control** (control variables): The “Control” section is a specialisation of “Global” icon (see Section 2.1.2.1.6); by the help of this new language element the programmer can collect in one place the declaration and definition of variables, which are essential for steering the execution (accessed from loop or conditional constructions). The other declarations and definition can remain in “Global” icon.

**Control Text Block** (SEQ\_CONTROL): The Control Text Block can be used to separate the C code fragments into two parts; **1**) which are essential for steering the execution (accessed from loop or conditional constructions), and **2**) which have no effect to the parallel processing at control-flow level. In the first case, the sequential code fragments can be placed in Control Text Blocks, the other parts may remain in Text Blocks (see Section 2.1.2.1.5).

**Control protocol** (PROTOCOL\_CONTROL): This new language element is based on the protocol declaration for ports (see Section 2.1.2.1.4) but some (or all) of them can be marked as “control” protocol if the corresponding data will be accessed from conditional or loop constructs.

### **2.1.2.2.2 Limitations of GRAPNEL program for automated modelling**

The following limitations and restrictions of GRAPNEL programs are assumed for the automated model transformation:

*Limitation 1.* Execution of conditional and loop constructs must be dependent on globally declared and simple type variables only (i.e. direct reference to received messages and usage of C functions are not allowed). These variables will be referred as *control variables* and must be placed in the new section of GRAPNEL programs, called “*Control*” section.

*Limitation 2.* Generic text blocks (SEQ) are handled as atomic and error-proof operations. The code lines of a given SEQ block, which have effect to any global control variables, must be placed into a separated Control Text Block (SEQ\_CONTROL). The code of Control Text Block may refer to control variables only, and contain only those expressions, which have corresponding expressions in CPN ML [57] as well. (e.g. value-binding, if-then-else construct)

*Limitation 3.* A single message passing is handled as an atomic and an error-proof operation. A message can change the value of any global control variable, if and only if the message was passed via a dedicated protocol, called control protocol. Only simple types of variables are allowed in control protocols. The related Communication Actions may refer to control variables only.

*Limitation 4.* Every loop construct must contain at least one communication action.

*Limitation 5.* Either branch of the conditional construct must contain at least one communication action.

*Limitation 6.* Only blocked communication actions are allowed.

*Limitation 7.* Only one-way communication channels are allowed (inout port is not modelled yet).

*Limitation 8.* Process groups, pre-defined communication templates and collective communication operations are not supported (one communication channel must have two dedicated communication ports).

The GRAPNEL programs, which met all these requirements, are called GRAPNEL\* programs.

In generally, the first three limitations require some extra efforts from the user giving more information to debugger and analyser tools about program structure. On the other hand, the GRAPNEL\* program will be more readable since the control and the pure data processing are separated, and these limitations will enable the efficient automated modelling, simulation, and verification techniques on the model. Limitations 4 and 5 have been already suggested in [10] but they will be also required by the correctness proof of macrostep-based execution. The

remaining limitations reflect to the current state of the research, and they address some future work to be done.

### 2.1.3 Coloured Petri-Net Patterns for GRAPNEL\* programs

The formalism of coloured Petri net (CPN, CP net) was chosen for expressing and composing model for GRAPNEL\* programs since CP nets are well founded, have been widely used to specify parallel software systems, and CP nets are supported by a number of tools for simulation, analysis and verification. I have studied two different approaches to composition of CP net models for GRAPNEL\* programs: the place fusion approach [23], that models each communication channel explicitly and the environment place approach [15], that models communication channels implicitly, by coupling arc inscriptions. The main disadvantage of environment place approach is that the control of message orderings and the synchronisation dependences between processes cannot be expressed effectively using environment place but it will be essential in the modelling of new macrostep-based execution.

Following the place fusion approach the user can generate automatically the CP net model of GRAPNEL\* programs based on graph transformation. As a test environment, I worked with *Design/CPN* version 3.1.1 for Linux tool (and later with version 4.0.5) that is equipped with several facilities, such as simulation and analysis capabilities [14], or an extended meta-language (ML) for defining guards for transitions, compound tokens, etc. [57]

#### 2.1.3.1 Short overview of coloured Petri nets and hierarchical decomposition

. The primary components of a CP net are the followings.

- *Data*: CP nets define data types (colorsets), data objects (tokens), and variables that hold data values.
- *Places*: Locations for holding data objects; zero or more tokens (a multiset) of some particular colorset. A place is represented graphically as an ellipse. The tokens in a place, like any group of tokens, constitute a multiset. The multiset in a place is called the marking of the place. Taken together, the markings in all the places in a CP net constitute the state of the net.
- *Transitions*: Activities that transform data. CP net transitions can change the number and/or value of tokens in one or more places. A transition is represented graphically as a rectangle.
- *Arcs*: Connect place(s) to transition(s). Every arc has a direction, either from a place to a transition or vice versa. An arc inscription is a multiset expression associated with an arc.
- *Input Arc Inscriptions*: Specify data that must exist in order to an activity to occur.
- *Guards*: Boolean expressions that further define conditions that must be true for an activity to occur. It is associated with a transition and written inside square brackets. A transition's guard must evaluate to boolean “true” in order for the transition to occur.

- *Output Arc Inscriptions*: Specify data that will be produced if an activity occurs (i.e. when a transition fires).

A transition can fire whenever certain conditions are met. Three factors determine together whether a transition is enabled:

1. The multiset of tokens in each input place of the transition.
2. The input arc inscription on each input arc connected to the transition.
3. The transition's guard.

If each input place of a given transition contains the multiset specified by the place's input arc inscription (possibly in conjunction with the guard), and the guard is evaluated to true, the transition is enabled, and can be fired.

Effective CPN modelling requires the ability to distribute a net across multiple pages, so as to divide it into modules small enough to keep track of. Such a module is called a submodel. This **hierarchical decomposition** requires some mechanism for interconnecting the submodels on the various pages, so that their states can influence to each other. Design/CPN offers two mechanisms for interconnecting net structures on different pages: substitution transitions and fusion places.

A *substitution transition* (e.g. *ProcA* in Figure 11) is a transition that stands for a whole page of net structure. In order to add details to a model without losing overview, a transition may have associated with a separate page of CP net structure, called a subpage. This page contains a more detailed view of the activity that the transition represents. The page, which holds the transition, is called the superpage.

Another form of relationship is possible in a CP net; a *fusion place* (e.g. *Port1.SR* in Figure 12) is a special place that has been equated with one or more other places, so that the fused places act as a single place with a single marking, i.e. the places are then functionally identical. Such places are called fusion places, and a set of fusion places is a fusion set.

### 2.1.3.2 Application level transformations

The application level of GRAPNEL programs is the place where the inter-process communication takes part via communication channels and communication ports. In this section, the transformation of the application level is described as a Design/CPN page, called *MainPage*, which will be the highest level superpage of the generated CP net.

#### 2.1.3.2.1 Process

Every P-GRADE process is represented as one substitution transition and two fusion places; *ReadyToRun* fusion place, *Terminated* fusion place, and the necessary arcs connecting them together. Note that these two fusion places will also appear at the process level, and represent the first and last states of a given process.

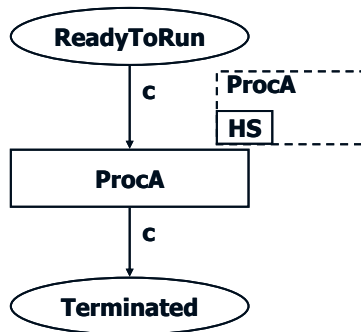


Figure 11 – CPN model of GRAPNEL process

### 2.1.3.2.2 Communication ports

Let us take an example, where *ProcA* is the sender process through port *Port1*, and process *ProcB* is the receiver through port *Port2*.

An input GRAPNEL\* communication port (*ProcA.port1*) is represented by three fusion places in the *Mainpage* of CPN model:

- *ProcA.port1.SR*: it holds a simple *ct* token, when the sender process is ready to send message
- *ProcA.port1.R*: it holds a simple *ct* token, when the receiver process is ready to receive message
- *ProcA.port1.D*: when the message passing is done, it will contain the message. The token can be a simple *ct* token if the corresponding port protocol is *not* control protocol, otherwise the colour of stored token is derived from the control protocol similarly to the case of control variables (see details in Section 2.1.3.3.2).

Note that these three fusion places will appear in process level, too.

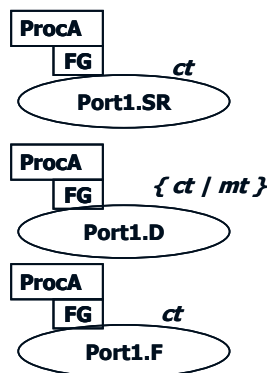


Figure 12 – CPN model of GRAPNEL output communication port (OUTPORT)

An output communication port of GRAPNEL\* program (*ProcB.port2*) is transformed into three fusion places in the *Mainpage* of CPN model:

- *ProcB.port2.SR*: it holds a simple *ct* token, when the sender process is ready to send message

- *ProcB.port2.F*: it holds a simple *ct* token, when the message has been received
- *ProcB.port2.D*: it will contain the message to be passed. The token can be a simple *ct* token if the corresponding port protocol is *not* control protocol, otherwise the colour of stored token is derived from the control protocol similarly to the case of control variables (see details in Section 2.1.3.3.2).

Note that these three fusion places will appear in process level, too.

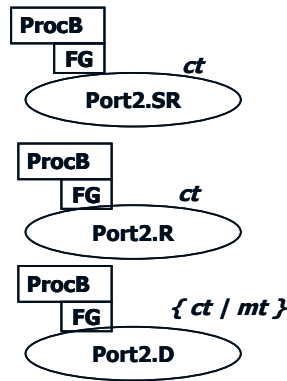


Figure 13 – CPN model of GRAPNEL input communication port (INPORT)

### 2.1.3.2.3 Communication channels

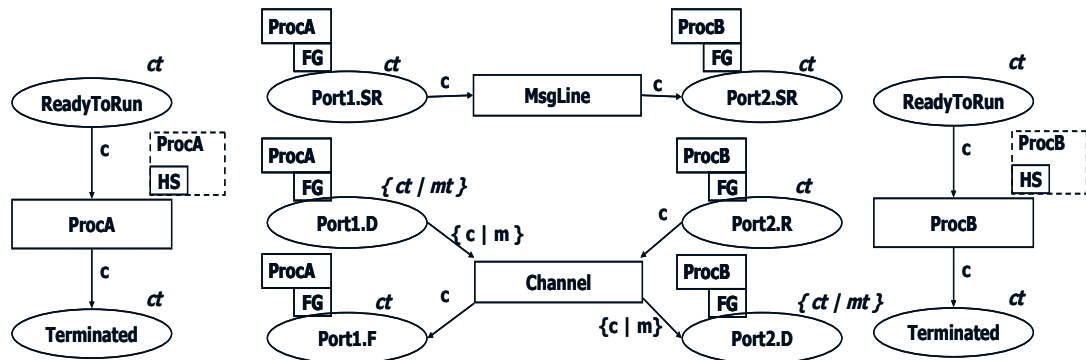


Figure 14 – CPN model of GRAPNEL communication channel (with ports and processes)

Two types of GRAPNEL communication channels can be defined applying the limitation rules of GRAPNEL programs; *synchronisation channel* or *control channel*. The synchronisation channel is not able to pass control messages via control protocols. On the contrary, the control channel is connected to ports, which uses control protocol.

The model of **synchronisation communication channel** consists of two simple transitions; *MsgLine* and *Channel*. All of the connected places can hold only *ct*

coloured tokens since; control messages cannot be passed via synchronous channel.

First, the sender process must inform the receiver process; the sender is ready to transmit the data. For this purpose, the sender process places a control token in *Port1.SR*, which can arrive to the receiver when transition *MsgLine* fires. This information is crucial when the receiver process must select the message source (i.e. a sender process) in case of alternative input communication action. These two places have not been integrated into one fusion place (in order to reduce the number of transitions) because of the future-proof design approach; in the more sophisticated models (including collective communication operations and process groups) one port will be connected to more ports. Thus, all of them should be fused, which may lead to a wrong model.

The model of synchronisation channel works similarly to the CPN model of *rendezvous primitive*: the transition is able to fire when both the *procA.port1.D* at the sender side and *procB.port2.R* at the receiver side places hold one *ct* token. Then, two tokens generated into the *procA.port1.F* and *procB.port2.D* that can be taken by the sender and the receiver processes for the detection of successful message passing.

The model of **control communication channel** is based on the model synchronisation communication channel but the *procA.port1.D* and *procB.port2.D* places can hold tokens not only *ct*-type tokens. The acceptable colours of these places are inherited from the control protocol in the same way as the transformation of types of control variables into colours (see details in Section 2.1.3.3.2).

### 2.1.3.3 Process level transformations

#### 2.1.3.3.1 Processes

Each P-GRADE process is represented on a separated Design/CPN subpage, which stands for the adequate substitution transition on the *MainPage*. The connection between these two pages is carried out using the *ReadyToRun*, *Terminated* fusion places, these will be the first and last places on the process subpage.

The *Initialize* transition is the first transition to be fired at subpage level after a token appears on the *ReadyToRun* place. It is responsible for all the necessary initializations on control variables.

#### 2.1.3.3.2 Variables

Two types of program variables can be distinguished from the aspect of our modelling; a part of the user-declared variables are referenced in the conditional expressions of conditional or loop constructs, while another part of the variables do not used in them. The CPN model contains only the variables, which are relevant in the controlling of program execution, i.e. the *control* variables.

At the beginning the first (*ReadyToRun*) places of each process contains one token, called *c*. This token works as an instruction pointer representing the currently executed piece of code. If a process does not declare any control variable, the token is declared as follows:

```
Color ct = with C;  
Var c : ct;
```

Otherwise, the control variables must be declared using either of the following syntax:

```
Variable_Type Variable_Name;
or
Variable_Type Variable_Name[Size_Of_Array];
or
enum Variable_Name = {"String1", "String2", ..., "Stringn"};
```

where

- Variable\_Type must be int, char, or string.
- Variable\_Name must be a valid variable name according to ANSI C syntax.
- Size\_of\_Array must be a positive integer between 1 and 2<sup>16</sup>.
- String<sub>x</sub> must be a valid string according to ANSI C syntax.

Each control variable is represented as a coloured token in the CPN model, referred to as *control token*:

```
Color Variable_Color = CPN_Type;
Var Variable_name : Variable_Color;
```

where

- Variable\_Name is remaining without changes.
- Variable\_Color is the Variable\_Name following “\_t” characters.
- CPN\_Type is a valid CPN colour, which represents the type of variable.

The global control variables types can be transformed into CPN colours in one of the following ways:

- CPN\_Type is the same as the Variable\_Type in case of simple types (int, char, or string).
- CPN\_Type is **Array of** Variable\_Type [**0..Size\_Of\_Array**]; in case of arrays.
- CPN\_Type is **With** “String<sub>1</sub>”, “String<sub>2</sub>”, ..., “String<sub>n</sub>”; in case of enumerations.

The control variables of a GRAPNEL\* process must be collected into one compound coloured token:

```
Color St = product Variable_Color1 * Variable_Color2 * ... * Variable_Colorn;
Var s : St;
```

During the simulation stage the St token also works as an instruction pointer showing the currently executed code region in the CPN model of GRAPNEL process.

Note: another approach can be also applied; a new place can be created for each control variable, which can store the actual value of a token but it is not feasible if too many control variables are declared.

### 2.1.3.3.3 Sequential code

The representation of Text Blocks (SEQ) are not necessary in the CPN model because each text block is handled as an atomic and error-proof operation from the view of modelling, and it has no effects to the control according to the

limitation rules (see Section 2.1.2.2.2). The sequential code can modify the value of any control variable in Control Text Blocks; it is modelled by a transition, and the changes of control variables are represented by the modification of dispatched compound coloured token.

There are two ways to modify a value of a token. One option is the use of output arc inscription of transitions substituting the token identifier with an expression (see Figure 15). On the other hand, each transition may have an attached code segment, which contains CPN ML code (see Figure 45 in the Appendix).

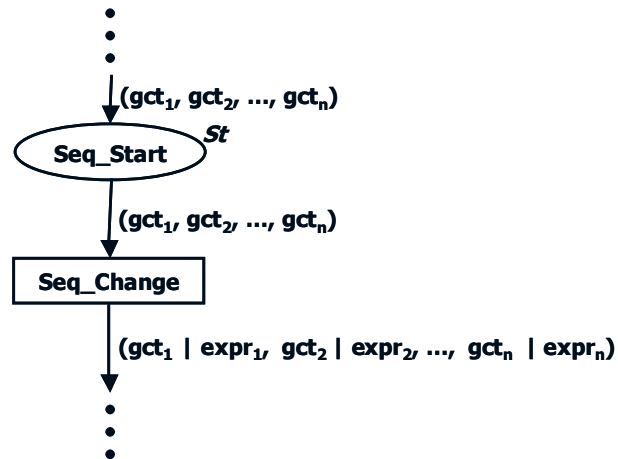


Figure 15 – CPN model of GRAPNEL sequential code segment (SEQ)

#### 2.1.3.3.4 Conditional construct

A conditional construction can be described by two guarded transitions. One guard, which belongs to the *true* branch, can be inherited from the C-like conditional expression of the related conditional construct. Obviously, another guard, which belongs to the *false* branch, must be the negated conditional expression of the related conditional construct. The conditional expressions may refer to global variables (stored in the compound token), but as a restriction, the invocation of arbitrary C function or C++ method is not allowed.

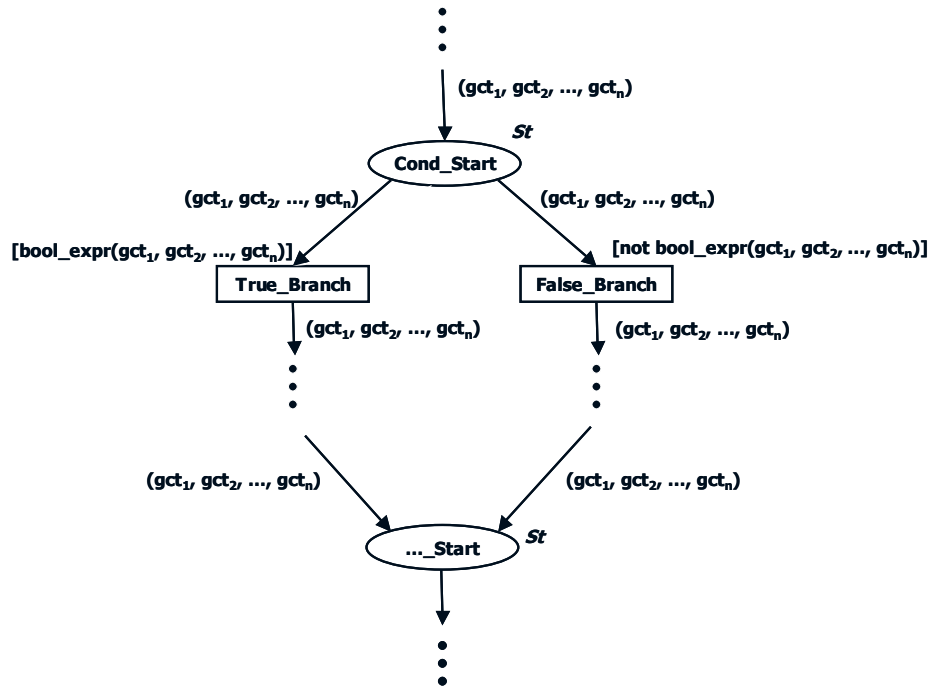


Figure 16 – CPN model of GRAPNEL conditional construct (COND)

The transformation of conditional expressions requires to replace the “==” signs to “=”, and Boolean operators, such as and  $\rightarrow$  andalso, or  $\rightarrow$  orelse according to the syntax of CPN ML language [57].

### 2.1.3.3.5 Loop construct

The CPN model of the loop construct relies on the previously described model of conditional branch construct.

In the case of FOR-type loop construct, four transitions are needed generally. The first transition (*Init\_Loop*) is used to set the control token to the given initial values according to the first part of C-like expression specified in loop-start icon. Then a guarded transition (*Enter\_Loop*) is placed at the beginning of the body of loop construct, and another guarded transition (*After\_Loop*), which represent the end of loop. The guard, which belongs to the body of loop, can be inherited from the C-like conditional expression (2<sup>nd</sup> part) of the related loop-start icon. Obviously, another guard, which belongs to the *After\_Loop* transition, must be the negated conditional expression.

By the end of each cycle the last transition (*End\_Loop*) modifies the value of the control token (i.e. global control variables) according to the 3<sup>rd</sup> part of loop expression specified in the loop-start icon.

The transformation of DO-type and WHILE-type loop constructs are similar to FOR-type loops but their CPN models are simpler and inherited easily from the FOR-type loop model.

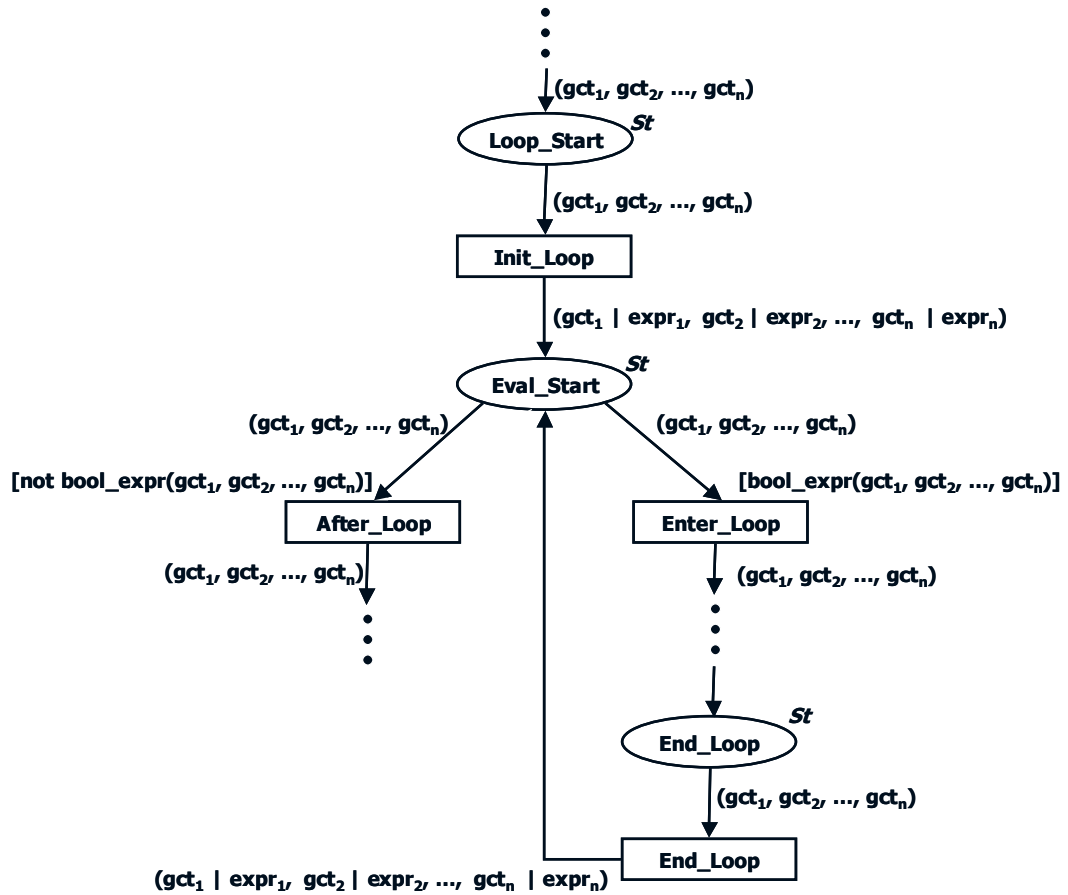


Figure 17 – CPN model of GRAPNEL loop construct (LOOP)

### 2.1.3.3.6 Communication operations

In GRAPNEL\* the communication operations (i.e. sending and receiving messages) always take place via communication ports, that belong to processes and are connected by communication channels. Every port must have its own protocol that determines the type and size of the messages to be transmitted via that port. The checking of protocol matching is the responsibility of GRED editor and GRAPNEL run-time library therefore; the CPN model does not deal with generic run-time protocol (data type) checking of GRAPNEL\* communication ports.

In the control flow graph of GRAPNEL\* process, Communication Actions (CA) are used for passing messages between processes. In the model, the different types of Communication Actions are represented basically by the CPN models of *fork and join primitives*.

At the beginning of an **output communication** (i.e. in the fork-phase), a *CAO\_SEND* transition places a new control token into the *ProCA.Port1.SR* place and another token into the *ProCA.Port1.D* place<sup>4</sup>. This token must be a ct token in case of a synchronisation channel. In case of a control communication channel, it can be a compound coloured token belonging to control message (*mt* token). The acceptable colours of *ProCA.Port1.D* place are inherited from the control protocol in the same

<sup>4</sup> It is similar to the real implementation of message passing: the process informs the underlying message passing library about the message to be send/received.

way as the transformation of types of control variables into colours (see Section 2.1.3.3.2).

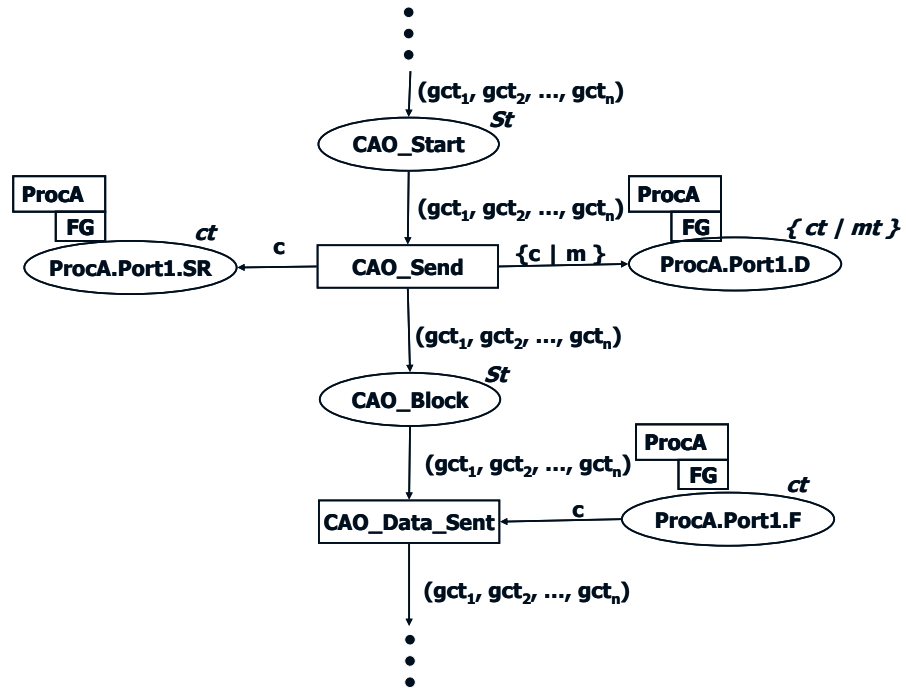


Figure 18 – CPN model of GRAPNEL output communication action (CAO)

The output Communication Actions must be blocked according to the limitation rules (see Section 2.1.2.2.2); the sender process must wait for the end of message passing in the join-phase. An additional transition, *CAO\_Data\_Sent* was introduced for this purpose, which can fire when the place *ProcA.Port1.F* holds a *ct* token at the end of the message transmission.

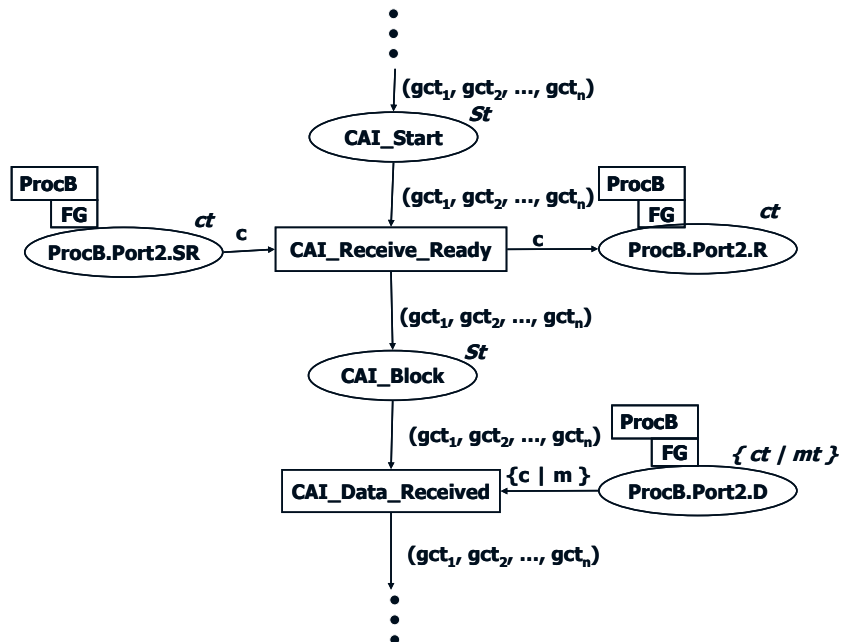


Figure 19 – CPN model of GRAPNEL input communication action (CAI)

The **input communication action** (CAI) also requires one transition in the fork-phase and another one in the join-phase, labelled as *CAI\_Receive\_Ready* and *CAI\_Data\_Received*. First, *CAI\_Receive\_Ready* transition is waiting for a control token from the sender. When it is arrived, the transition places a control token into *ProcB.Port2.R* in order to inform the sender, *ProcB* is ready to receive the message. Then, *ProcB* process has to wait for a token from the place labelled *ProcB.Port2.D*. The received token must be a single *ct* token in case of synchronisation channel. In case of control communication channel, there must be an *mt* token belonging to control variables, which were defined in the CAI icon. The acceptable colours of *ProcB.Port2.D* place are inherited from the control protocol in the same way as the transformation of types of control variables into colours (see Section 2.1.3.3.2). When the *CAI\_Data\_Received* transition fires, the contents of control messages are copied into coloured tokens (using its attached code segment [14]), which are belonging to the control variables defined in the data field of CAI icon.

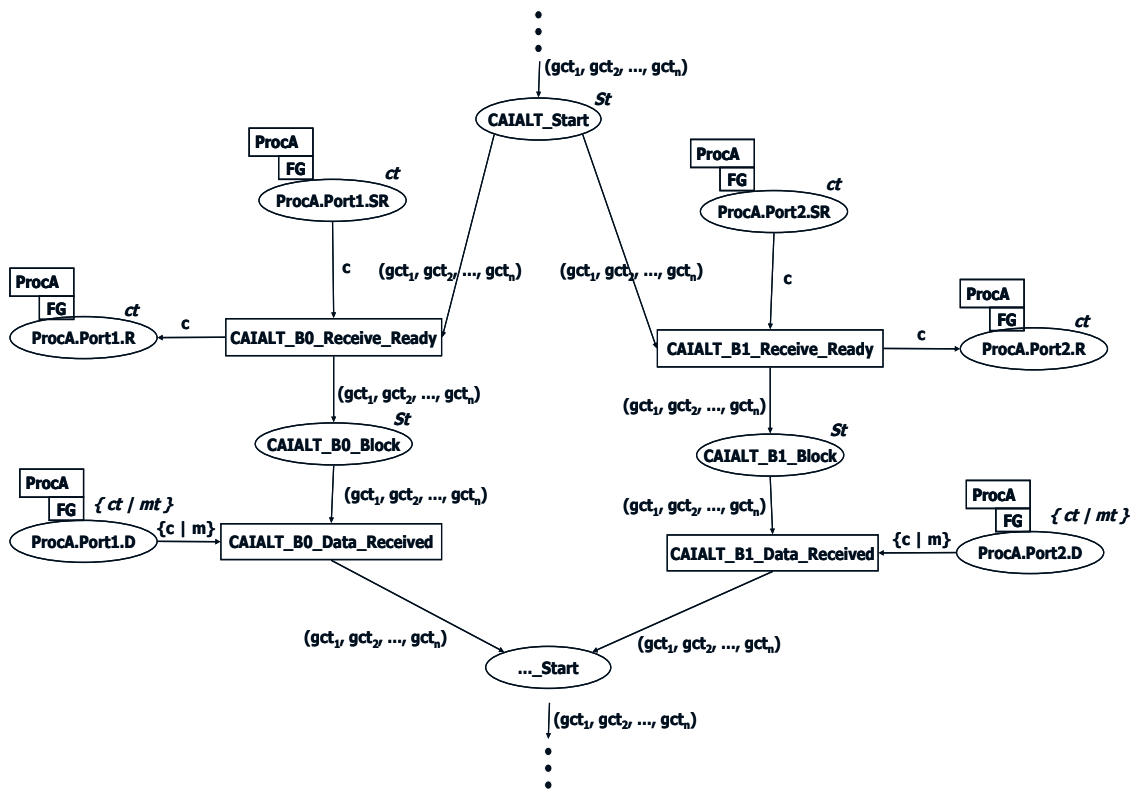


Figure 20 – CPN model of GRAPNEL alternative input communication action (CAIALT)

The **alternative input communication action** (CAIALT) can receive a message via any communication port, which is connected to the communication action. The process is blocked until a message is available from one of the sources. If more than one messages are available, the selection is non-deterministic.

The CPN model of CAIALT operation is based on the model of the simple input Communication Action (CAI) but both transitions are multiplied according to the number of possible sources but only one of them can fire at both stages of communication.

**Summarizing** the model of message passing; the way how data is exchanged between two processes in a CP net is the following in details:

1. The control of *Proc1* arrives to a CAO operation, this means that an *st* token (which represents the whole process local data image) is put on the *CAO\_Start* place. Then, the *CAO\_Send* transition fires, and a *ct* token (an “instruction pointer” without colour) appears on the *Proc1.PortA.SR* fusion place and an *mt* token (the token carrying the data to be sent) on the *Proc1.PortA.D* fusion place. If the partner’s execution is not pending on a CAI (belongs to this CAO), than *Proc1* will wait on the *CAO\_Block* place. Meanwhile the *MsgLine* (on the *MainPage*) transition freely fires, and a *ct* token is placed on *Proc2.PortB.SR*.
2. When the control of *Proc2* arrives to CAI (there is a token on *CAI\_Start*), it will block until a token does not arrive on *Proc2.PortB.SR* indicating that the partner is ready to send the data. When this happens, the *CAI\_Receive\_Ready* transition may fire, resulting a *ct* token on *Proc2.PortB.R* fusion place.
3. Keeping a close watch on the *MainPage*, the *Channel* transition may be executed, whenever there is a token on *Proc2.PortB.R*, and another on *Proc1.PortA.D*. When this occurs, the data is put on *Proc2.PortB.D*, and a control token is put on *Proc1.PortA.F*. These tokens enable the firing of *CAI\_Data\_Received*, and *CAO\_Data\_Sent* transitions, after all the control of these two processes may go further.

#### 2.1.4 Building CPN model automatically

Generally, the users develop GRAPNEL programs by the help of the built-in graphical editor of P-GRADE environment, and not by hand (i.e. applying simple text editors).

Three representation modes of GRAPNEL programs can be distinguished:

1. The basic representation level is the *textual description* in files (\*.grp) for storing and sharing GRAPNEL programs (see [58]).
2. Inside GRED editor the different elements of GRAPNEL language are represented by C++ classes (intermediate representation) in the operative memory. The GRP-IO module, which is a part of GRED editor, is responsible for writing GRAPNEL files (i.e. the textual description) directly, and also for reading them based on the facilities of a file parser designed for the GRAPNEL language.
3. Finally, the developers watch only the different types of GRED windows on the screen, where the language elements of GRAPNEL translated into displayable graphs and icons using *MOTIF widget set* (graphical representation).

The design and implementation issues of the two upper representation levels (intermediate/graphical representations) can be found in my M.Sc. thesis [59].

The automated generation of CPN model from GRAPNEL applications has been realised at the intermediate representation level extending the C++ class serialisation facilities of GRED editor. The following advantages of the intermediate representation level were taken into consideration when this was chosen as the input representation level of transformation into CPN model:

- The representation is already checked syntactically; the GRED editor does not allow to construct invalid graphs both the application and the process internal level [59]. On the other hand, at textual level the GRP2C pre-compiler and the built-in compiler tool are able to check the syntax of C and C++ source code related to graphical icons. The translation is launched if the code can be compiled successfully.
- The representation contains information about the graphical layout of language elements; for animation purposes the GRED editor computes and stores some particular graphical information (position, size) in the fields of C++ classes, which belong to graphically moveable/replaceable GRAPNEL language elements, such as processes, sequential icons, and control lines<sup>5</sup>. Based on the provided graphical information the transformation algorithm are able to place the transitions and places of CPN model in a conveniently readable way.
- A traversing/serialisation algorithm has been already implemented for the representation; the transformation based on a tested traversing algorithm but it extends the serialisation functionalities using predefined CPN patterns.
- The representation can be described in formal description; C++ classes of intermediate level itself and their relation to each other are described with widespread UML class diagrams.

#### 2.1.4.1 Traversal algorithm for CPN generation

Since the representation of GRAPNEL\* program must be syntactically verified before model generation [59], the coloured Petri-net model can be generated by using simple *embedding* graph transformation rules on each graphical programming item of the program. For this purpose, a new tool, *GRP2CPN* traverses the entire program; first, it generates the *Mainpage* of CPN model including application level elements, such as processes, ports and communication channels. In the second phase, the transformation tool generates one CPN subpage for each process, and a recursive algorithm (see below) ensures the traversing of all elements at process level, such as conditional and loop constructs, communication operations and text blocks.

```
Procedure ProcessElements_To_CPN(elem, term_elem)
begin
  while not End_Of_Process(elem)
  begin

    if elem = term_elem than return
```

<sup>5</sup> The Global, Local, Heads, and Control sections have fixed positions.

```

if elem.type = CODE_SEQ_CONTROLL than SEQ_CONTROLL_To_CPN(elem)
if elem.type = CODE_CAI than CAI_To_CPN(elem)
if elem.type = CODE_CAO than CAO_To_CPN(elem)
if elem.type = CODE_CAIALT than CAIALT_To_CPN(elem)

if elem.type = CODE_LOOP-START than
  begin
    LOOP-START_To_CPN(elem)
    LOOP-END_To_CPN(elem.get_loop-end)
    ProcessElements_To_CPN(elem.get_body, elem.get_loop-end)
  end

if elem.type = CODE_COND-START than
  begin
    COND-START_To_CPN(elem)
    COND-END_To_CPN(elem.get_cond-end)
    ProcessElements_To_CPN(elem.get_true_branch, elem.get_cond-end)
    ProcessElements_To_CPN(elem.get_false_branch, elem.get_cond-end)
  end

  elem = elem.get_next

end
end

```

Notes for the algorithm can be found in Table 2.

<i>elem</i>	the actual (or first) element of the subgraph to be transformed
<i>term_elem</i>	the terminal element of the subgraph to be transformed
<i>elem.type</i>	type of the given element (CODE_CAO, CODE_COND_START, etc.)
<i>elem.get_next</i>	next element in the subgraph: in case of start element of conditional or loop construct, it means the next element followed by the construct
<i>elem.get_true_branch</i> , <i>elem.get_false_branch</i>	in case of start element of conditional construct, it means the first element of its true/false branch
<i>elem.get_loop-body</i>	in case of start element of loop construct, it means the first element in the loop
<i>elem.get_cond-end</i> , <i>elem.get_loop-end</i>	in case of start element of conditional or loop construct, it means the last (terminal) element of the construct

**Table 2 – Notes for the traversal algorithm**

As it can be seen, by each GRAPNEL\* program element the introduced CPN pattern must be inserted into the model. Each GRAPNEL\* element has a corresponding CPN pattern in the model, and the number and directions of arcs between these CPN patterns always match.

In this way, I proved that *a coloured Petri-net model always exists and can be automatically generated for any GRAPNEL\* programs for debugging purposes*. As an illustration, an automatically generated model for the “Producer-Consumer” GRAPNEL application can be found in the Appendix.

### 2.1.5 Related works

There are existing approaches [6][16][17] to detect erroneous program behaviour based on Petri-net analysis, especially dead-locks, but these techniques developed mainly theoretically with less practical results, and not integrated into a high-level graphical development framework together with an automatic model generator.

Other advanced solutions [19] are based on UML models and can be used in the early phases of the system design to capture system dependability attributes like reliability and availability. This approach includes an automatic transformation from UML diagrams to Timed Petri Nets. In contrary of these, my method is to be used later in the debugging phase (after the implementation) and relying on the GRAPNEL\* language.

In [15], a CPN model is created for GRAPNEL programs using place fusion approach, but it is not suitable for our purposes as is. According to presented approach CPN model for generic GRAPNEL programs cannot be generated automatically (due to the missing detailed description of transformation steps), and has not been connected to any widespread CPN tool (since it uses its own description language).

## 2.2 Formalisation of macrostep-based debugging technique for GRAPNEL\* applications

### 2.2.1 Preface

The following issues make debugging of P-GRADE and other parallel programs much more difficult than traditional sequential debugging [48]:

- (1) To address the problem of the *large number of parallel processes* with dynamic interactions, the DIWIDE distributed debugger [49] in P-GRADE environment is able to observe computations both at a global level, to understand the interactions, and at the level of the individual processes.
- (2) The *non-deterministic behaviour* of GRAPNEL programs makes the actual execution behaviour dependent on actual process speeds (due to distinct processor speeds and to vary operating system scheduling effects) and unpredictable communication delays. It required the DIWIDE distributed debugger to provide facilities to detect those situations, and to evaluate program correctness properties for various possible execution timings during the automated debugging. It also required techniques to allow reproducible and coherent observation of such error situations [1].
- (3) The general problem of *constructing consistent global states* of distributed systems must be also considered because the evaluation of erroneous situations depends on accurate observations. In general, the accurate observation can only be approximately achieved in a distributed system, by

remote observation, based on message passing, so the user faced the difficulty of absence of a global system state. To solve this problem, DIWIDE distributed debugger provides strategies for the observation of consistent computation states by the help of macrostep-by-macrostep execution as it is described in this thesis.

- (4) *Probe-effect* due to the observation and control mechanisms is a well-known phenomenon: any observation affects the system under study, so the distributed debugger must rely on techniques that either ensure the lowest possible intrusion (and still allow user interaction, even knowing that this is a highly intrusive activity) or give full control on the occurring race conditions. The DIWIDE debugger applied the second approach because the debugger can gain detailed information about the structure of GRAPNEL program despite the traditional parallel debuggers.

In order to handle these problems for GRAPNEL\* programs in different debugging sessions, I present the formal description for a novel, automatic generation of successive global consistent states, called “macrostep-based” execution, in this thesis. The traversal of the state-space with macrosteps is also introduced, and my formal description is based on the Occurrence graph of the Petri-net model (see Section 2.2.3). I presented that *the traversal rules of Occurance graph can be formally defined as the selection of representative sets of state transitions, which result the macrostep-based execution of any GRAPNEL\* program.* Thus, the software developer can apply an execution mechanism for the GRAPNEL\* parallel programs, which is similar to step-by-step execution of traditional sequential programs.

### 2.2.2 Macrostep-based execution and Execution Tree

The main aim of macrostep-based execution is the generation of consistent cuts [1] (or global states) for GRAPNEL programs. For illustration purposes, two examples can be found in Figure 21; one consistent cut, and one inconsistent cut. Intuitively, a consistent cut incorporates all the past of its own events.

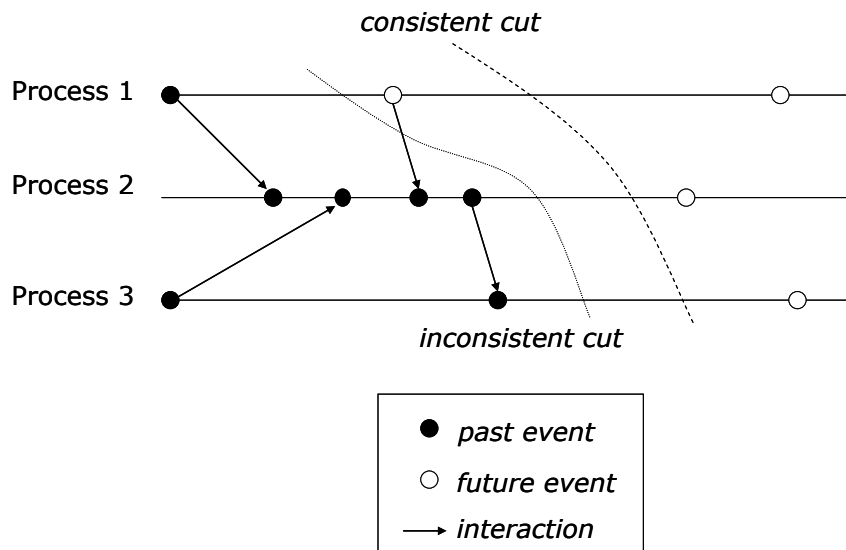


Figure 21 – Consistent and inconsistent cuts

The idea of macrostep is based on the concept of collective breakpoints<sup>6</sup> (such as  $CAO^2_1$ -  $CAO^2_2$ -  $CAO^2_3$ -  $CAO^2_4$ , see Figure 22), which are placed on the inter-process communication primitives (CAO, CAI or CAIALT actions) in each GRAPNEL process. These communication actions (see Figure 22) are indexed by the corresponding process number (lower index), and a serial number (upper index). The set of executed code regions between two consecutive collective breakpoints is called a **macrostep**. A detailed definition of macrostep is given in [49][48].

Assuming that sequential program parts between communication instructions are already tested, each sequential code region can be handled as an atomic operation. In this way, the automated debugging of a parallel program requires to debug the parallel program by pure macrosteps.

A single breakpoint of the collective breakpoint is called *active* if it was hit in a macrostep and its associated communication instruction can be completed (e.g. see  $CAO^2_2$  in Figure 22). On the other hand, a breakpoint is called *sleeping* if it was hit in a collective breakpoint but its associated communication instruction cannot be completed during the next macrostep thus, it will be a part of the next collective breakpoint. For example (see Figure 22), a send instruction ( $CAO^2_1$ ) of a given process (*Process 1*) wants to send a message to another process (*Process 4*), but it is communicating with a 3<sup>rd</sup> process (*Process 3*). That is why, the breakpoint placed at  $CAO^2_1$  operation is a sleeping breakpoint and can be found in the next collective breakpoint. Similarly to this,  $CAI^3_4$  is also a sleeping breakpoint, since it must wait for  $CAO^4_5$ .

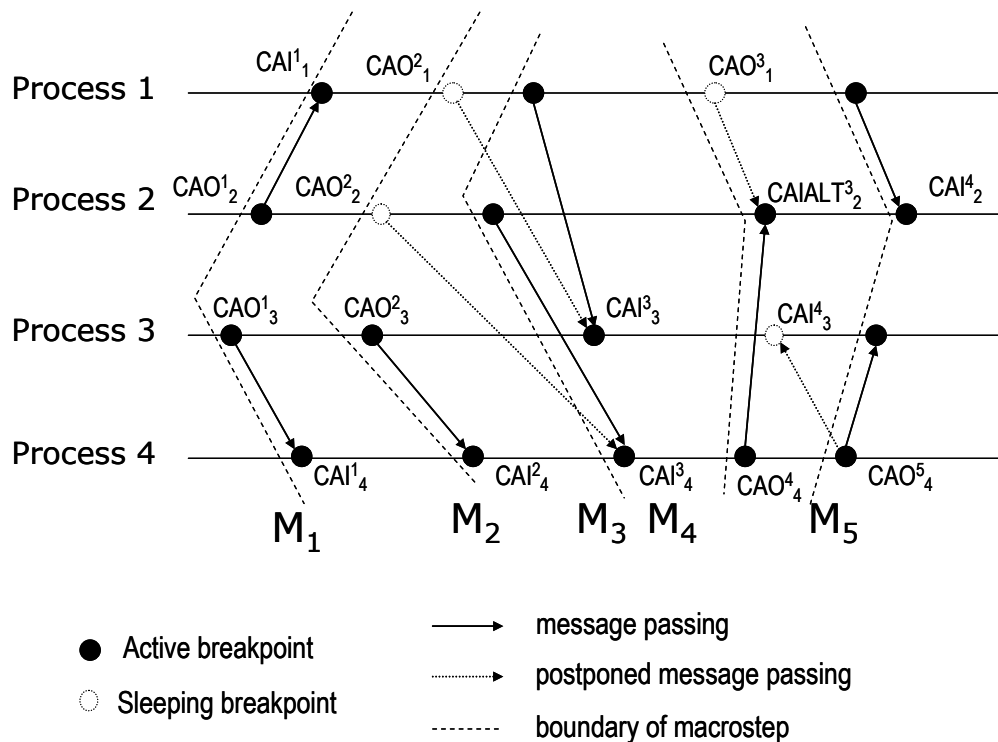


Figure 22 – Illustration for macrostep-based execution

<sup>6</sup> A collective breakpoint consists of a finite number of single breakpoints placed in different processes, and it was hit if (and only if) all the breakpoints belonging the collective breakpoint were hit.

The **macrostep-by-macrostep execution mode** of parallel programs can be defined as follows; in each macrostep every process is forced to run until a collective breakpoint is hit. Thus, the boundaries of the macrosteps (see Figure 22,  $M_1, M_2, \dots$ ) are defined by a series of global breakpoint set, and the consecutive consistent global states of parallel program are generated automatically. As it is described later, a checker engine is able to evaluate the specification in the consecutive consistent global states during the macrostep-by-macrostep execution (see Section 3.1), and a simulation engine is able to continue the program simulation starting from the given global state (see Section 3.2.2).

There is a clear analogy between the step-by-step execution mode of sequential programs realised by local breakpoints and the macrostep-by-macrostep execution mode of parallel programs. The macrostep-by-macrostep execution mode enables to check the progress of the parallel program at the points that are relevant from the point of view of parallel execution, i.e. at the message passing points. What we ensured is that the macrostep-by-macrostep execution mode works deterministically just as the step-by-step execution mode works in case of sequential programs. In order to ensure it, the debugger stores the history of collective breakpoints and the acceptance orders of messages at alternative receive instructions. For example (see Figure 22);  $CAIALT^2_3$  may receive either from *Process 1* ( $CAO^3_1$ ) or from *Process 4* ( $CAO^4_4$ ), but in the actual case only the message from *Process 4* is accepted in this race condition.

At replay, the progress of tasks is controlled by the stored collective breakpoints and the program is automatically executed again macrostep-by-macrostep as in the execution phase.

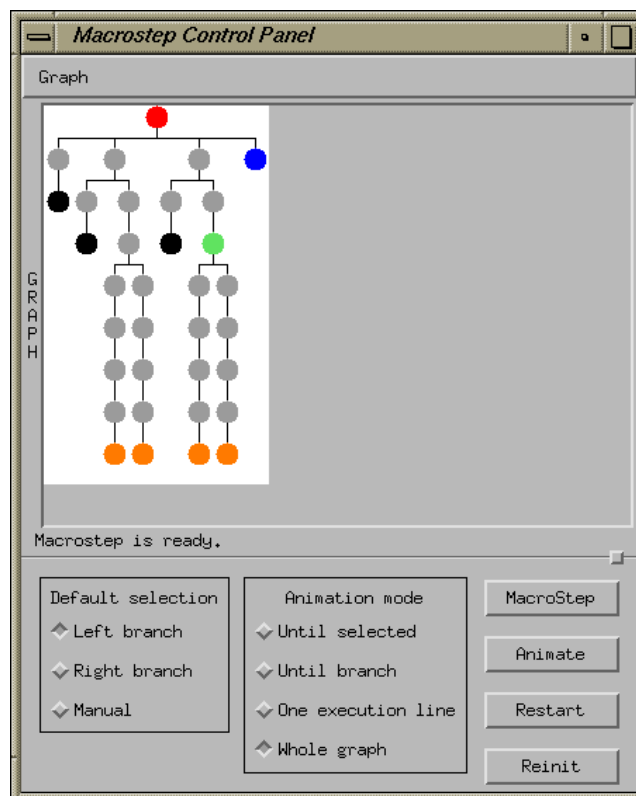


Figure 23 – Execution Tree with navigation panel

The execution path is a graph whose nodes represent the boundaries of macrosteps (i.e. consistent global states) and the directed arcs indicates the possible macrosteps (i.e. the possible state transitions between consecutive global states). The **execution tree** (see Figure 23) is a generalization of the execution path; it contains all the possible execution paths of a parallel program assuming that the non-determinism of the current program is inherited only from alternative (CAIALT) message passing communications. Nodes of the execution tree can be of four types: (i) Root node, (ii) Alternative nodes, (iii) Deterministic nodes.

The Root node represents the starting conditions of the parallel program. Alternative nodes indicate either a wildcard receive instructions which can choose a message non-deterministically from several processes.

Breakpoints can be placed at the nodes of the execution tree. Such breakpoints are called **meta-breakpoints**. The role of meta-breakpoints is analogous with the role of the breakpoints of sequential programs. A breakpoint in a sequential program means to run the program until the breakpoint is hit. Similarly, a meta-breakpoint at a node of the execution tree means to place the collective breakpoint belonging to that node, and run the parallel application until the collective breakpoint is hit. Replay guarantees that the collective breakpoint will be hit and the parallel program will be stopped at the requested node.

In *ideal* case, the task of automatic debugging (or testing) would be the exhaustive traversing of the complete execution tree with all the possible execution paths in it. Therefore, the execution tree would represent a search space that should be completely explored by the debugging method. Accordingly, the automatic debugging and testing of a parallel program would require (i) generation of its execution tree (ii) exhaustive traverse of its execution tree.

*In fact*, when the user tries to debug a real size program systematically, the user has to face the problem of combinatorial explosion of possible execution paths.

### 2.2.3 Formal description of macrostep-based execution

The Occurrence Graph (OCC graph) of a given CPN model (see Sections 2.1.3 and 2.1.4) is a directed graph where each  $s \in \mathbf{S}$  node represents a possible token distribution (i.e. marking), and each  $r \in \mathbf{R}$  arc represents a possible state transition between two consecutive token distributions [14].

A part of OCC graph related to a sample GRAPNEL\* application, Producer-Consumer (see Appendix 8.1) contains 1028 nodes and 2358 arcs. The first and the last four levels of the OCC graph are graphically presented in Figure 24.

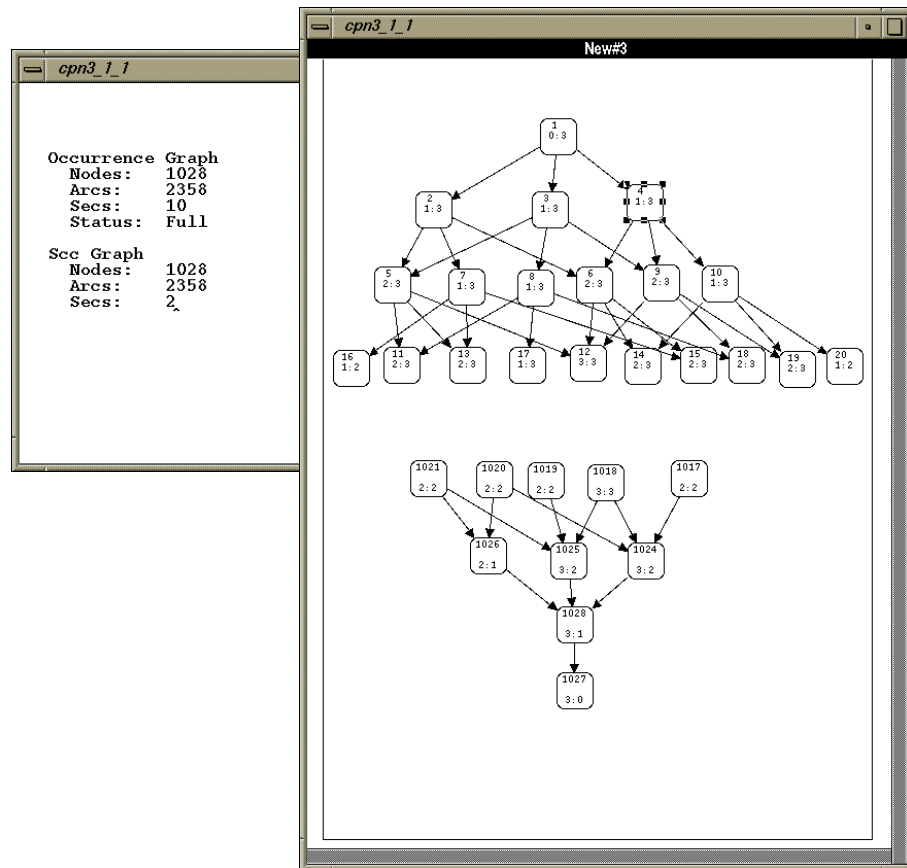


Figure 24 – Occurrence Graph

In order to formalise the macrostep execution the following notations are introduced:

$s_i$	$i^{\text{th}}$ node of OCC graph (i.e. possible marking or consistent global state)
$\text{enabled}(s_i)$	set of (globally) enabled transitions in the given $s_i$ marking. A transition $r \in \mathbf{R}$ is enabled in a state $s$ if there exists a state $s'$ such that $(s; s') \in r$ . Otherwise, $r$ is said to be disabled at $s$
$\text{locally\_enabled}(s_i)$	set of locally enabled transitions in the given $s_i$ marking. A transition $r \in \mathbf{R}$ is locally enabled if the corresponding process enables it (but another process may disable it thus, it may not be part of globally enabled transitions)
$\text{ample}(s_i) \subseteq \text{enable}(s_i)$	set of <i>representative</i> transitions in the given $s_i$ marking, only these transitions will be taken into consideration by the generation of state-space
$\text{succ}(s_i, j)$	the node of OCC graph where the $j^{\text{th}}$ ample transition leads from $s_i$ node. Note: $\text{succ}(s_i, 0) = s_i$

Table 3 – Notations for the description of macrostep-based execution

Three sets of transition can be distinguished in the CPN model: the *MsgLine* transitions (see Section 2.1.3.2.3) belonging to *communications channels* ( $T_{\text{comm}}$ ), the local transitions of individual *processes* ( $T_{\text{local}}$ ), and the first transitions *CAIALT\_Bx\_Receive\_Ready* of alternative input communications, denoted as  $T_{\text{CAIALT}}$ . The set of enabled transitions, *enabled(s)* may contain arbitrary types of transitions depending on the actual local conditions of processes as well as the actual communication interactions.

The macrostep execution tries to reduce the set of enabled transitions, which must be taken into consideration during the construction of global state-space (i.e. the execution tree). This subset of enabled transitions in OCC graph is noted as *ample(s)* in a given marking.

The requirements of generation of representative transitions can be summarised as follows:

- The debugging method must be unaffected by the skipped paths.
- The size of graph must be reduced.
- The calculation requirements of representative paths must be as low as possible.

The ample set generation of macrostep execution follows these rules:

**Rule 1.**  $\text{enabled}(s_i) = \emptyset \Rightarrow \text{ample}(s_i) = \emptyset$

**Rule 2.**  $(\text{enabled}(s_i) \cap T_{\text{local}}) \neq \emptyset \Rightarrow \text{ample}(s_i) = r : r \in (\text{enabled}(s_i) \cap T_{\text{local}})$

**Rule 3.**  $(\text{locally\_enabled}(s_i) \cap T_{\text{CAIALT}}) \neq \emptyset \Rightarrow$   
 $\text{ample}(s_i) = (\text{enabled}(s_i) \cap T_{\text{CAIALT}}) \cup$   
 $\forall r \in (\text{locally\_enabled}(s_i) \cap T_{\text{CAIALT}}) : \text{enabled}(s_i) \cap$   
 $T_{\text{partner\_process}(r)}$

**Rule 4.**  $(\text{enabled}(s_i) \cap T_{\text{comm}}) \neq \emptyset \Rightarrow (\forall j \in \mathbf{N}: 0 \leq j < |\text{enabled}(s_i) \cap T_{\text{comm}}|)$   
 $\text{ample}(\text{succ}(s_i, j)) = r : r \in (\text{enabled}(\text{succ}(s_i, j)) \cap T_{\text{comm}})$

In other words; (Rule 1) if there is not enabled transition, the ample set must be empty. (Rule 2) If any of the *local* transitions ( $T_{\text{local}}$ ) is enabled, the ample set must contain only one of these transitions (and nothing else). (Rule 3) If there is no local transition, all enabled *message selection* transitions ( $T_{\text{CAIALT}}$ ) must be included in the current ample set and the enabled (communication) transitions of the processes, which cannot send a message to any alternative input, however it is waiting for it. (4) If only message passing transitions ( $T_{\text{comm}}$ ) are enabled, they must be fired transition by transition.

For practical reasons, the boundaries of macrosteps are defined before the application of *Rule 4*, since *Rule 3* is the only one, which can generate an ample set including more transitions, and the selection of sender processes for CAIALT operations must be allowed to the user. Here, in the execution tree a new branch belongs to each combination of possible sender processes.

In this way, I presented that *the traversal rules of Occurance graph can be formally defined (Rule 1-4) as the selection of representative sets of state transitions, which result the macrostep-based execution of any GRAPNEL\* program.*

## 2.2.4 Related works

The NOPE (Non-deterministic Program Evaluator) deals with this problem by generating in a record phase partial traces which contain ordering information of critical events [29][28]. During replay these data are used to enforce the same event ordering as occurred during record; from these data, NOPE also allows to evaluate other possible program runs for the supplied input.

## 2.3 Correctness of macrostep-based execution

### 2.3.1 Preface

In this thesis, I present the correctness of the macrostep based debugging methodology based on the theoretical background of verification (model checking) of systems [24], and Kripke structures as the common way for the representation of OCC graph and Execution Tree (see Section 2.2).

As the first step, both the Execution Tree (the result of macrostep-based execution) and the Occurrence Graph (corresponding to the entire state space of GRAPNEL\* program) are transformed into Kripke structures;  $KS_e$  and  $KS_p$ . Then, I prove that *the macrostep-based selection algorithm of representative state transitions is a kind of partial ordering on the Kripke structure  $KS_p$ , and the Kripke structures  $KS_e$  and  $KS_p$  are stuttering equivalents to each other.* Therefore, the software developer gets successive global states (macrosteps) without losing any relevant information about the behaviour of observed system, and temporal logic formulas can be evaluated on the state-space generated by the macrostep mechanism (see Section 3.1)

### 2.3.2 Basic notations: Kripke structures

In order to find relationship between the Occurrence Graph (generated based on the CPN model) and the Execution Tree (generated by the macrostep debugger) both graphs were transformed to a common formal description using Kripke structures. Formally, a Kripke structure can be defined as follows:

$KS = (S, R, L)$ , where

$S = \{s_1, s_2, \dots, s_n\}$  a finite set of states

$R \subseteq S \times S$  state transitions (relation of states)

$L: S \rightarrow 2AP$  labels of states (atomic predicates)

In the case of Occurrence Graph the  $L$  labels can be derived from the actual states of (compound) coloured tokens. Due to performance reasons (see Section

2.3.4.3), only one selected process and its actual state are used to labelling the Kripke structure. The predicates can be simple derived from the colouring of tokens. On the other hand, during the macrostep-by-macrostep execution the  $L$  labels can be obtained from the address space of the given process relying on the variable evaluation facilities (provided by DIWIDE distributed debugger [49]). Therefore, the labels can be determined on-the-fly during the generation of Execution Tree as well as during the CPN simulation.

### 2.3.3 Partial ordering techniques

One of the main challenges in automatic verification of software products is related to the state space explosion. In case of many types of software products, the number of possible different states during program execution grows very rapidly, even exponentially with the program size and the number of its components. Partial order reduction is a special technique that addresses this state space explosion problem for concurrent asynchronous programs by resulting a smaller state space that is to be traversed by the verification (model checking) algorithms.

In general, asynchronous systems are defined using so-called interleaving model of computation. Concurrent events are modelled by allowing the occurrence of events in all possible orders, creating very often a huge number of possible states and related paths as well. However, the software specifications typically do not cover and make distinction between all different orders. The key concept of partial order reduction is to take into consideration only a limited set of behaviours of the program, but ensuring that the not inspected behaviours do not provide any new and relevant information for the model checking.

In the last decade, several researcher groups have developed methods by following different ways to exploit these reduction principles in model checking. These techniques include (among others) the stubborn sets method of Valmari [60], the persistent sets method of Godefroid and Wolper [61][62], and the ample sets method of Peled [63]. These works share similar ideas, although they differ concerning the details of the proposed reduction. I applied the ample sets method in this thesis because of its similar approach to macrostep based execution.

Before diving into the details, some key notions must be introduced according to the literature:

- Invisible transition: A transition  $r = (s, s')$  is called invisible, if  $L(s)=L(s')$  i.e. the labelling is not changed between the two transitions.
- Stuttering equivalence: In a path, there can be consecutive states with the same label due to invisible transitions. We can create so-called stuttering blocks from these states. Two paths,  $x1$  and  $x2$ , are *stuttering equivalent* ( $x1 \sim_{st} x2$ ) if the same stuttering blocks in the same order occur in both path. Two Kripke structures,  $K1$  and  $K2$ , are stuttering equivalent structures if every paths  $x1$  in  $K1$  belongs to a path  $x2$  in  $K2$ , where  $x1 \sim_{st} x2$ .

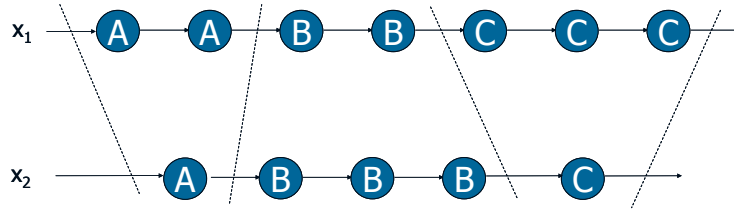


Figure 25 – Stuttering equivalent paths

### 2.3.4 Macrostep-based execution: a partial ordering technique

The core mechanism of macrostep-by-macrostep execution can be interpreted as the overlapped execution of independent state transitions. Hence, we can apply some partial ordering techniques and methods on  $KS_p$  Kripke structure (derived from the Occurrence Graph of coloured Petri net model) in order to get the  $KS_e$ , the Execution Tree built by the macrostep debugger.

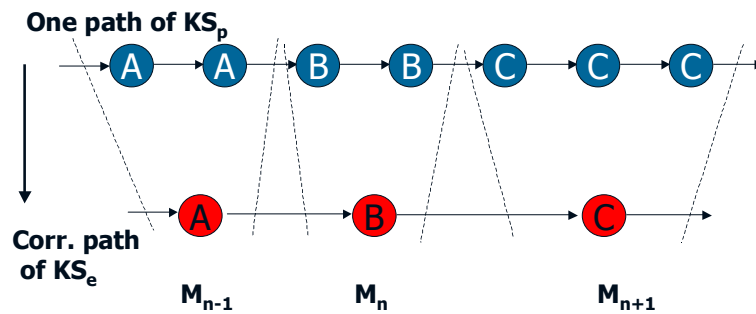


Figure 26 – Stuttering equivalent paths during macrostep based execution

According to the literature [8] four conditions must be satisfied by the calculation of ample sets in order to get stuttering equivalent graphs. The macrostep-based execution meets these requirements as it is proven in the following four sections.

#### 2.3.4.1 Emptiness

The first condition guarantees that the macrostep algorithm will make progress further if the normal search algorithm would also progress:

**C0 Emptiness:**  $ample(s) = \emptyset$  iff  $enabled(s) = \emptyset$

**Proof:** The emptiness condition in case of the macrostep algorithm can be proven in the following three steps:

*Step 1.* The macrostep algorithm generates an empty ample set if there is no enabled transition. It is ensured by Rule 1:  $enabled(s_i) = \emptyset \Rightarrow ample(s_i) = \emptyset$ , which must be applied in all states. Thus, this constraint must be true on the entire state space:

$$( enabled(s) = \emptyset ) \Rightarrow ( ample(s) = \emptyset )$$

*Step 2.* On the other hand, the transitions can be classified in three types in the CPN model, such as  $T_{\text{comm}}$ ,  $T_{\text{local}}$ , and  $T_{\text{CAIALT}}$ . Rules 2-4 guarantee that in each state:

$$( \text{enabled}(s) \neq \emptyset ) \Rightarrow ( \text{ample}(s) \neq \emptyset ),$$

since one of the rules must be always applied if the set of enabled transition is non-empty, and each rule generates a non-empty ample set.

In details; if at least one  $T_{\text{local}}$  is enabled, then the *Rule 1* is valid, which generates a one-element ample set. If at least one  $T_{\text{CAIALT}}$  is enabled, then the *Rule 2* must be applied, which generates an ample set with at least one element. If there is at least one  $T_{\text{comm}}$  transition, the *Rule 3* can be applied, which generates a sequence of one-element ample sets.

*Step 3:* Combining the constraints from *Step 1* and *Step 2* into one constraint, and after some basic transformations, we can get C0 as a result:

$$\begin{aligned} & ( \text{enabled}(s) = \emptyset \Rightarrow \text{ample}(s) = \emptyset ) \wedge ( \text{enabled}(s) \neq \emptyset \Rightarrow \text{ample}(s) \neq \emptyset ) = \\ & = ( \text{enabled}(s) = \emptyset \Leftrightarrow \text{ample}(s) = \emptyset ) = ( \text{ample}(s) = \emptyset \Leftrightarrow \text{enabled}(s) = \emptyset ) \end{aligned}$$

### 2.3.4.2 Ample decomposition

The ample decomposition constraint [8] has a crucial role in order to ensure that any path that is excluded in the reduced state-transition graph can be somehow transformed (based on the properties of independent transitions) into an existing path in the reduced model. Therefore, in other words; the reduction does not leave out accidentally any paths, which are essential for the model verification.

**C1 Ample decomposition:** In the full state graph, on any path starting from some state  $s$ , a transition dependent on a transition from  $\text{ample}(s)$  cannot appear before some transition from  $\text{ample}(s)$  is executed.

More details of the construction and a proof for its correctness are given by Clarke et al. [24].

#### **Proof.**

In general, it is more difficult to check condition C1 than C0 because this condition deals with a property of ample sets concerning the execution sequences of the *full* state-transition graph. However, the most important goal of the state reduction technique is exactly that to avoid generating this full state-transition graph.

Another problem is that the execution sequences on which condition C1 are to be evaluated can extend arbitrarily until the occurrence of the first ample transition. In principle, checking condition C1 is usually at least as difficult as performing reachability analysis for the full state transition graph, as it described in [24].

In practice, algorithms for verifying condition C1 for an arbitrarily chosen set of ample transitions might be complex and expensive. That is why, partial order

based verification techniques exploit the given software structure to produce the necessary ample sets of state transitions that can meet easier the condition C1. The ample set selection becomes simpler in our typical case as well when the inspected application is a composition of concurrent processes, i.e. using GRAPNEL\*.

Following this approach, we can define ample(s) as the set of all transitions enabled at  $s$  in some set of processes  $\mathbf{P}$  with the following property: no process  $P_i \in \mathbf{P}$  has a communication transition locally enabled in  $P_i$  with any process outside of  $\mathbf{P}$ . This rule can be implemented in a simple way; first include only one individual process as a member of set  $\mathbf{P}$  and then adding the processes to  $\mathbf{P}$  that communicate with any process in  $\mathbf{P}$  in a repetitive way. If  $\mathbf{P}$  includes all processes by the end, the state is fully expanded and no reduction can be achieved at the current state.

The partitioning of the concurrent processes into these two subsets ensures that by executing transitions not from the ample set a transition dependent on an ample transition cannot be globally enabled, i.e. performed before a transition in the ample set. This is exactly the constraint set by condition C1.

The macrostep-based execution meets this condition since every rule satisfies it as follows:

*Rule 1:* There is no enabled transition at all.  $\mathbf{P}$  must be empty.

*Rule 2:*  $\mathbf{P}$  has always one member process with one enabled local transition, which will form the one-element ample set. This way of ample set generation meets condition C1, since this process do not intend to communicate to other processes.

*Rule 3:* In this case,  $\mathbf{P}$  contains all processes, which have alternative input communication action (CAIALT) as well as all the possible sender process belonging to these alternative input communications. All the enabled CAIALT transitions are the member of ample set. Moreover, the ample set contains the enabled transitions belonging to the sender processes, which are actually not able to send a message to the CAIALT constructions. Thus, condition C1 is guaranteed.

*Rule 4:* In this case, we have only pairwise processes, which are ready to exchange their messages and they do not intend to communicate to other processes. Therefore,  $\mathbf{P}$  always contains the two communicating processes, and the ample set contains only one transition, which is a member of  $T_{\text{comm}}$ , so condition C1 is guaranteed.

### 2.3.4.3 Invisibility

After applying the conditions C0 and C1 during the calculation of ample sets, we need to deal with two remaining types of the paths [8]:

- $q_1, q_2, \dots, q_n, r$ , when transition  $q_1, q_2, \dots, q_n$  are independent from the transitions of ample(s) set and  $r \in \text{ample}(s)$
- $q_1, q_2, \dots, q_i$  infinite path, when transition  $q_1, q_2, \dots, q_i$  are independent from the transitions of ample(s) set

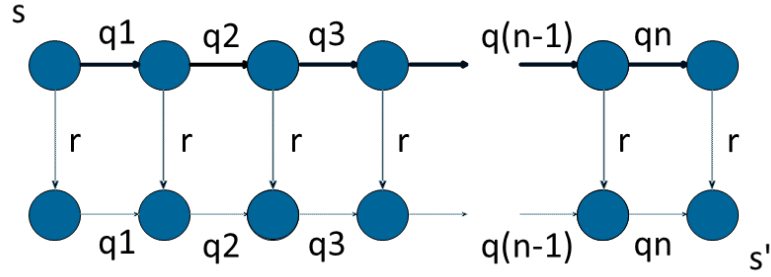


Figure 27 – Invisible paths after reordering of transitions based on commutativity

It must be guaranteed that the specification is not affected, by ensuring that the generated path is stuttering equivalent to the original one, i.e.  $q_1, q_2, \dots, q_n, r \sim_{st} r, q_1, q_2, \dots, q_n$ . Generally, this aspect is covered by the following condition C2:

**C2 Invisibility:** If a state  $s$  is not fully expanded, every transition  $r \in ample(s)$  has to be invisible.

In practice, we have to take into consideration a fact that influences the effectiveness of the reduction [8]. Assuming two independent transitions both of them can change the truth-value of predicates that are in the checked property (the transitions are visible), i.e. the execution order between the two transitions becomes crucial, even if they are independent. In details, condition C2 enforces that the visible transitions must be involved into an ample set only with all the other visible transitions. As a result, if an execution is not presented because of the reduction, then another execution with the same order of visible transitions will be presented in the reduced state graph. Based on the experiments, the effectiveness of the reduction decreases dramatically with the number of desired predicates used in the specification. As a thumb rule, the simplified checked properties are more effective, e.g. checking separately  $\diamond p$  and  $\diamond q$ , rather than  $\diamond p \wedge \diamond q$ .

On the other hand, some experimental results of using partial order reduction are presented in [8] with the *Spin* model checking system [64]. The checked algorithms were as follows:

- *sieve*: The distributed Sieve of Eratosthenes algorithm (used to find prime numbers)
- *dtp*: A data transfer protocol
- *snoopy*: A cache coherence protocol
- *pftp*: A file transfer protocol

For each of these algorithms, the property that was checked asserted that some variable, initialized with 0, eventually becomes 1.

That is why, it is a reasonable restriction if a user must focus on only one GRAPNEL\* process and the checked properties must refer to control variables of only one selected process  $P_s$ . Obviously, the address spaces of other processes are also available for manual checking during the real debugging. Note: the user can be

responsible for avoiding these transient situations by selecting the appropriate properties.

**Proof.**

I introduced a modified version of C2 to guarantee that the generated path is stuttering equivalent to the original one, i.e.  $q_{1r} q_{2r} \dots, q_{nr} r \sim_{st} r, q_{1r} q_{2r} \dots, q_n$ .

**C2' Invisibility:** If a state  $s$  is not fully expanded, every transition, which is **independent from** any transition  $r \in ample(s)$ , has to be invisible.

According to C2' transition  $q_{1r} q_{2r} \dots, q_n$  must be invisible. Hence,  $q_{1r} q_{2r} \dots, q_{nr} r \sim_{st} r, q_{1r} q_{2r} \dots, q_n$  is also guaranteed. Therefore, we can apply either C2 or C2' in order to proof the invisibility condition by each macrostep Rule:

*Rule 1:* There is no ample set.

*Rule 2:* The ample set always consists of one local transition  $r \in T_{local}$  of a process. If  $r$  is not visible (i.e. *not* belonging to the selected process  $P_s$ ), then C2 is true. On the other hand, if transition  $r$  is visible (i.e. belonging to the selected process  $P_s$ ), all the other independent transitions  $q_{1r} q_{2r} \dots, q_n$  must belong to other (not selected) processes. Thus, transitions  $q_{1r} q_{2r} \dots, q_n$  must be invisible, they can not change the labelling according to introduced labelling rules thus, C2' is true.

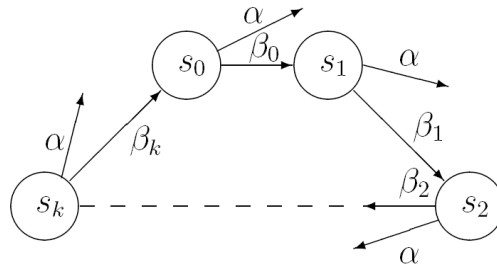
*Rule 3:* The ample set may consist of transition either  $r \in T_{CAIALT}$  or  $r \in T_{comm}$ . Transition  $r \in T_{CAIALT}$  is always invisible, they cannot change the state of control tokens. So if there is not any transition  $r \in T_{comm}$ , then C2 is true. If there is at least one transition  $r \in T_{comm}$ , we can distinguish two cases:

- If all transitions  $r \in T_{comm}$  are invisible (not belonging to the selected process  $P_s$  or communication without CONTROL\_PROTOCOL), then C2 is again true.
- If one of the transitions  $r \in T_{comm}$  belongs to the selected process  $P_s$  and communication with CONTROL\_PROTOCOL, then this transition will be visible. All the other independent transitions  $q_{1r} q_{2r} \dots, q_n$  must belong to other (not selected) processes, which cannot change the labelling, then C2' is true

*Rule 4:* The ample set always a one-element set containing a transition  $r \in T_{comm}$ . If one of these transitions  $r \in T_{comm}$  belongs to the selected process  $P_s$  and communication with CONTROL\_PROTOCOL, then this transition will be visible (otherwise C2 can be applied). The independent transitions  $q_{1r} q_{2r} \dots, q_n$  must belong to other (not selected) processes thus, C2' is true.

### 2.3.4.4 Cycle closing condition

**C3 Cycle closing condition:** A transition, which is enabled in every state of a cycle in the reduced state space, belongs to the ample set of some state on the cycle.



**Figure 28 – Illustration for cycle closing condition**

The origin of a cycle in the state-space is obviously the LOOP construct of GRAPNEL\* programs. The condition C3 can be fulfilled relying on Limitation 4, i.e. every LOOP construct must contain at least one communication action. Consequently, at least one successful communication action, i.e. one macrostep must be done successfully in each cycle involving the application of Rule 4. . Rule 4. ensures that there must be a state, where are only message passing transitions ( $T_{\text{comm}}$ ) enabled in the model. Later, these transitions will form one-element ample sets and will be fired one by one. Therefore, there is no transition, which is always enabled in a cycle, and it is not an element of any ample set.

### 2.3.5 Further reduction: Execution Tree

Basically – based on the earlier described macrostep-based traversal algorithm – the macrostep **Execution Tree** can represent the stuttering blocks of the reduced state-space but some other aspect can be also taken into consideration during the generation of Execution Tree.

Control variables of the selected process (i.e. labelling in the Kripke structure) can be modified in

- CONTROL\_SEQ block
- CONDS
- LOOPS (in case of FOR or WHILE type loop construct)
- LOOPE (in case of DO-type loop construct)
- CAI or CAIALT instructions (if the message is arrived from CONTROL\_PROTOCOL)

Moreover, every communication action means a chance of deadlock or other synchronization errors, that is why, the CAO boxes are also enrolled in this list as well. Therefore, the boundaries of stuttering blocks would be these elements, and the macrostep debugger should place breakpoints on these language elements.

According to the limitations (see Section 2.1.2.2.2), we can assume that the sequential parts of GRAPNEL\* processes are well tested and error-proof code segments, itself. So in practice the boundaries of stuttering blocks are extended to communication actions when the process intends to interact with other process. (Note: there is also an option to execute a process graphical icon-by-icon if

necessary but this is a sequential debugging issue.) In this way, between two consecutive communication actions, any modification in the state-space (i.e. in the labelling) can be considered as a transient state, only the two states belonging to the communication actions are represented in the Execution Tree.

Thus, *the macrostep-based selection algorithm of representative state transitions is a kind of partial ordering on the Kripke structure  $KS_p$ , and the Kripke structures  $KS_e$  (belonging to the Execution Tree) and  $KS_p$  (belonging to the OCC graph) are stuttering equivalents to each other.*

### **2.3.6 Related works**

Partial order reduction is a very successful technique for handling the state explosion problem that is inherent to explicit state model checking of asynchronous concurrent software products (systems). It exploits the commutativity of concurrently executed transitions in interleaved system runs in order to reduce the size of the explored state space, as it was presented in [7][8][9].

## 3 Model checking methods in parallel debugging

### 3.1 *Run-time checking of temporal logic specification during controlled execution*

#### 3.1.1 Preface

The behaviour of sequential programs can be described with classical logic by a predicate (i.e. output condition) that must hold after the execution of the program. Furthermore, the output condition can be translated (by the technique of weakest preconditions) into conditions that must hold at every step of the program. Thus, such a condition can be considered as an assertion that must hold at a particular program step. If a system focuses on a particular subclass of formulas, such an assertion can be checked at runtime. Annotating a program by *runtime assertions* is a simple but very effective way of increasing the code's reliability and thus, the developers' confidence in a program's correct behaviour.

The fact that a particular assertion holds does not prove the correctness of a program. Besides, if an assertion fails, it uncovers an error in the behaviour of program. Assertions play an important role in the development of sequential programs, but currently their usage in parallel programming and debugging is far less dominant. One reason is that a program may exhibit different executions for the same input due to non-determinism; furthermore, the most relevant properties describe the state of the complete system (and not the state of a single process). Another reason is that the scopes of properties are usually not defined by specific code locations but by temporal relations to other properties. These problems are difficult to overcome in production runs of parallel programs and with classical logic. Therefore, my attention turned to program runs controlled by parallel debugger and to assertions expressed in the language of temporal logic.

The particular motivation of the presented work is to support the development of correct and reliable parallel programs by runtime assertions that are derived from temporal logic formulas, which describe the expected program behaviour. This thesis presents that the temporal logic formulas can be used as runtime assertions in a macrostep-based parallel debugging environment.

In this approach, the program developer asserts in a message-passing program by one or more of such formulas regarding the expected system behaviour. The debugger allows by generating consistent global states (macrosteps) to interactively elaborating the Execution Tree (i.e., the set of possible execution paths) which arises from the use of non-deterministic communication operations. In each macrostep, a temporal logic checker verifies that the once asserted temporal formulas are not violated by the current program state. This approach thus introduces effective runtime assertions into parallel debugging by incorporating ideas from the model checking of temporal formulas.

In this thesis, relying on the above outlined approach, I describe in details a method for the integration of a new debugging framework, where the actual GRAPNEL\* program runs controlled by the macrostep-based debugger as the universe in which the user-defined temporal logic formulas are checked. My proposed method includes the initialisation phase of the framework, the way of

insertion and detection of run-time temporal logic assertions, the support for the evaluation of atomic predicates referenced by temporal logic formulas, and the communication protocol with a general purpose temporal logic checker using state machine description. Based on these achievements, I present that *a particular class of temporal logic expressions (LTL<sub>x</sub>) can be evaluated on the paths of Execution Tree during macrostep based execution*. In this way, the required user-interaction (to detect erroneous situations) can be radically reduced by temporal logic assertions.

### 3.1.2 Introduction

Since debugging and testing parallel programs is an important but difficult task, many projects have been developing tools to support the user in this area; for surveys see [32] or the more recent [28].

A particular challenge is the handling of non-determinism which arises in message passing programs from a wildcard receive operation, i.e., a receive operation that non-deterministically accepts messages from different communication partners. The DIWIDE debugger [52] of the P-GRADE environment [27] applies the technique of macrostep, which allows the developer testing all branches of an application in a concurrent manner; as it was described in Section 2.2.

Temporal logic has proved as an adequate framework for describing the dynamic behaviour of a system (program) consisting of multiple asynchronously executing components (processes) [30][31]. A temporal logic formula can be considered as the specification of a parallel program; in linear time temporal logic (LTL) a program is correct if every possible execution satisfies the formula. If a program itself is described in a formal framework, the technique of model checking can be applied to decide about the correctness of temporal specifications provided that the program only exhibits a finite number of states [24]. There exist tools for the validation of concurrent system designs based on temporal logic [25] and for the generation of test cases from temporal specifications [26]. In the system presented here, the actual program runs controlled by a debugger as the universe in which a temporal formula is checked. Thus, the developed framework combines ideas and methods from parallel debugging and from model checking as well.

#### 3.1.2.1 Theoretical background for temporal logic assertions

This section sketches the formal basis of using temporal formulas as runtime assertions. For sake of brevity, only the core of the formal definitions are showed.

Any system can be described by a tuple  $\langle is, ns \rangle$  where  $is$  is the set of initial states of the system and  $ns$  is next state relation of the system. A temporal formula  $F$  is valid for such a system, written as  $\mathbf{T}[[F]]is\ ns$ , if for every (finite or infinite) state sequence  $s$  induced by  $\langle is, ns \rangle$ ,  $F$  holds at position of  $s$ . Thus it suffices to define the truth value of a temporal formula  $F$  at position  $i$  of  $s$ , written as  $\mathbf{T}[[F]]s\ i$ :

$$\mathbf{T}[[p_n(t_0, \dots, t_{n-1})]]s\ i = [[p_n]]([[t_0]]s_i, \dots, [[t_{n-1}]]s_i)$$

$$\mathbf{T}[[\Box F]]s\ i = \text{true iff } \mathbf{T}[[F]]s\ j = \text{true for all } j \text{ with } i \leq j < |s|$$

$$\mathbf{T}[[\Diamond F]]s\ i = \text{true iff } \mathbf{T}[[F]]s\ j = \text{true for some } j \text{ with } i \leq j < |s|$$

The corresponding definitions for connective formulas and quantifier formulas as usual. Now let us introduce a “next step” formula  $\circ_v F$

$$\mathbf{T}[[\circ_v F]]_s \text{ } i = \text{if } i + 1 = |s| \text{ then } v \text{ else } \mathbf{T}[[F]]_s (i+1)$$

which is true, if  $F$  holds in the next step, and if no such step exists (because  $i$  denotes the last position of a finite sequence  $s$ ), takes the truth value  $v$ . Then a formula translation  $\mathbf{G}[[F]]$  can be defined

$$\mathbf{G}[[p_n(t_0, \dots, t_{n-1})]] = p_n(t_0, \dots, t_{n-1})$$

$$\mathbf{G}[[\Box F]] = \mathbf{G}[[F]] \wedge \circ_{\text{true}} \Box F$$

$$\mathbf{G}[[\Diamond F]] = \mathbf{G}[[F]] \wedge \circ_{\text{false}} \Box F$$

such that in the result  $G := \mathbf{G}[[F]]$  the operators  $\Box$  (“always”) and  $\Diamond$  (“eventually”) are always guarded by the  $\circ_v$  operator. It is easy to show that the translation preserves the semantics, i.e., that  $\mathbf{T}[[F]]_s \text{ } i = \mathbf{T}[[G]]_s \text{ } i$ . Therefore, there is a way to reduce the validity of a temporal formula  $F$  in a state sequence  $s$  at position  $i$  to the validity of atomic formulas in state  $s(i)$  and to the validity of temporal formulas in  $s$  at  $i+1$ .

However, while the above definition is based on state sequences, in assertion checking we only have access to the “current” state of the system. Therefore a set of state trees<sup>7</sup>  $T(is, ns)$  is introduced induced by  $\langle is, ns \rangle$ . Each node in such a tree  $t$  holds a state  $t_{state}$ , has a link  $t_{prev}$  to its predecessor node (i.e.,  $ns(t_{prevstate}, t_{state})$  holds), and a set of successor nodes  $t_{next}$  (such that for each  $x$  in this set  $x_{prev} = t$ ). The root  $r$  of these trees (the node with  $r_{state} \in is$ ) have  $r_{prev} = \top$ ; the leaves  $l$  of these trees (for which no state  $s$  exists such that  $ns(l_{state}, s)$ ) have  $l_{next} = \{\top\}$ . We can now define the semantics  $\mathbf{T}[[G]]t$  of a guarded formula  $G$  with respect to such a tree  $t$  as follows:

$$\mathbf{T}[[p_n(t_0, \dots, t_{n-1})]]t = [[p_n]]([[t_0]]t_{state}, \dots, [[t_{n-1}]]t_{state})$$

$$\mathbf{T}[[\circ_v F]]t = \text{true iff } \mathbf{T}(\mathbf{G}[[F]])x = \text{true for every } x \in t_{next}$$

$$\mathbf{T}[[\circ_v F]]\top = v$$

If we define validity as

$$\mathbf{T}[[G]]T = \text{true iff } \mathbf{T}[[G]]t = \text{true for every } t \in T$$

it is easy to show that the relationship to the original semantics is preserved, i.e.,  $\mathbf{T}[[F]]is \ ns = \mathbf{T}[[G]]T(is, ns)$ . Thus, the validity of a temporal formula is reduced in a system to the validity of a guarded temporal formula in the set of state trees induced by the system.

However, during assertion checking, there is access to only a part of the tree referenced by a “current” state node whose children (the nodes of the successor states) may not yet be completely (or not at all) evaluated. Such “partial trees” can be represented by trees that contain “unknown subtrees” denoted by  $\perp$ . In order to extend the semantics  $\mathbf{T}$  on complete trees to a semantics  $\mathbf{T}_3$  on partial trees, the 2-valued logic can be replaced by a 3-valued logic with an additional value  $\perp$  (“unknown”)

$$\neg_3 \perp = \perp, \quad \mathbf{T} \wedge_3 \perp = \perp, \quad \mathbf{F} \wedge_3 \perp = \mathbf{F}, \quad \mathbf{T} \vee_3 \perp = \mathbf{T}, \quad \mathbf{F} \vee_3 \perp = \perp$$

<sup>7</sup> I speak of “trees” instead of “directed graphs” since we will not utilize the fact that the graph may contain loops.

and introduce the additional rule

$$\mathbf{T}_3 [[G]] \perp = \perp$$

The semantics compatible with original one and monotonic with respect to a partial ordering  $\subseteq$  of trees according to their information content (i.e.  $\perp \subseteq t$ , for every  $t$ ):

$$t \text{ does not contain } \perp \Rightarrow \mathbf{T}_3 [[G]]t = \mathbf{T} [[G]]t$$

$$s \subseteq t \Rightarrow \mathbf{T}_3 [[G]]t = \mathbf{T}_3 [[G]]s$$

So the semantics of a guarded temporal formula  $G$  can be defined on the set of partial state trees induced by a system. While above explanation only describes temporal operators  $\square$  and  $\diamond$  which refer to the “future” of a state, the presented framework also supports corresponding operators which talk about the “past”. Based on this semantics, a temporal logic checker, TLC, has been defined by RISC/LINZ in a joint work that has been integrated into the DIWIDE debugger as described in the following sections.

### 3.1.2.2 Macrostep Debugging in P-GRADE

DIWIDE is a distributed debugger [52], which is part of the visual parallel programming environment P-GRADE [50]. This debugger implements the macrostep method, which gives the user the ability to execute the application from communication point to communication point [49].

In brief, a macrostep is essentially the set of executed code regions between two consecutive collective breakpoints. A collective breakpoint is a set of local breakpoints, one for each process, that are all placed on communication instructions such that a macrostep contains communication instructions only as the last instructions of its regions. In the macrostep execution mode, DIWIDE generates from the current collective breakpoint the next collective breakpoint and then runs the program until the new collective breakpoint is hit. In this fashion, the program is executed macrostep by macrostep. At replay, the progress of processes are controlled by the stored collective breakpoints and the program is automatically executed again macrostep by macrostep as in the execution phase.

When a communication operation in a collective breakpoint is alternative receive operation, this collective breakpoint splits macrostep execution into multiple possible execution paths. Each path represents one possible selection of sender/receiver pairs for all wildcard receive operations in the originating collective breakpoint.

The set of all possible execution paths can be represented by a tree whose nodes represent collective breakpoints and whose arcs represent macrosteps. The macrostep control panel of the DIWIDE debugger visualizes the Execution Tree and allows the user to control its further elaboration. The developer may select particular branches in the tree or let the system traverse the tree according to some strategy (always choose leftmost respectively rightmost branch). The developer may also set a metabreakpoint in some node and let the system replay execution along the corresponding branch until the selected node is hit. The system therefore gives the developer very powerful means to control the non-deterministic behaviour of a parallel program in the debugging process.

The following sections describe the macrostep-based debugging model extended by checking the validity of temporal formulas in the branches of the execution tree that are elaborated according to the manual choice of the user respectively to the automatic selection strategy.

### 3.1.3 Macrostep Debugging with Temporal Logic Assertions

In order to illustrate the integrated macrostep-temporal logic model checker methodology, a case study is presented based on the well-known producer-consumer problem.

Take a parallel program, *Buffer* with three processes: a *Producer* process, which generates a finite number of items and sends them to a Buffer process. The *Buffer* process can receive items from the Producer and eventually forwards them to a Consumer process. The *Consumer* process receives these items from the Buffer and processes them. The Buffer has a finite capacity; depending on its full state (full, empty, not full and not empty) it is waiting for requests either from one or both processes. See details in Appendix 8.1.

Therefore, its behaviour is –in general– non-deterministic and may be investigated by the macrostep debugger.

#### 3.1.3.1 Temporal logic specification

Assuming synchronous message-passing (see Limitation 6 in Section 2.1.2.1), a fundamental property, which the user expects from the Buffer application, is that the number of items stored in the rounded buffer always equals the difference of the number of items sent by the Producer process and the number of items received by the Consumer process. The left side of the conjunction ( $\square$  NoLostItem) and the corresponding predicate (NoLostItem, see Section 3.1.3.2) express the above constraint. Another property concerning the progress of Buffer application: if the buffer is non-empty, it will eventually get empty in the future (see the right side of the conjunction). In case of finite running, these constraints also guarantee implicitly that the buffer must be empty as well as all the items are consumed at the termination.

In the notation of temporal logic [31][30], these properties can be written as

**BufferSpec.**  $\square$  NoLostItem  $\wedge$   $\square$  (  $\neg$  BufferEmpty  $\Rightarrow$   $\diamond$  BufferEmpty )

where NoLostItem expresses the core of the first property and BufferEmpty the core of the second property. The temporal operator  $\square$  reads as ‘always’ and the temporal operator  $\diamond$  as ‘eventually’.

The BufferSpec property is asserted at the beginning of example application (see the first icon of Buffer process, labeled assert in Figure 29) by a new GRAPNEL language element, called ASSERT box. It contains the name of specification (BufferSpec) implemented as a Java class (inherited from the checker.Specification class) whose method getFormula returns an object that encodes formula:

```
package checker;
import checker.*;

class BufferSpec extends Specification
{
    public BufferSpec() { }
}
```

```

public Formula getFormula()
{
    return new Conjunction
        (new Always
         (new Atomic("NoLostItem", null)),
         new Always
         ( new Implication
           (new Negation( new Atomic("BufferEmpty", null) ),
            new Eventually ( new Atomic("BufferEmpty", null) )
           )
         )
        );
}
}

```

When the debugger encounters the assert statement, it instructs the temporal logic checker (TLC), which is implemented in Java, to dynamically load this class. TLC is called by the debugger after every subsequent macrostep to verify whether the state of the current collective breakpoint violates the asserted formula or not. The user can follow the checking process in a window that displays the status of the formula in every collective breakpoint: false means that the stated assertion has been violated by the current execution, true means that the assertion cannot be violated any more, unknown means that the assertion may be still violated in the future.

### 3.1.3.2 Atomic predicates

The formula BufferSpec refers to two atomic predicates NoLostItem and BufferEmpty, which are the names of C-functions located in a separate library that is dynamically loaded by the debugger. Whenever the TLC asks the debugger for the value of an atomic formula, the debugger executes the corresponding function, which returns the value of the predicate (true or false) in the current globally consistent system state.

```

#include "chk-ext/IFpred2diw.h"

int NoLostItem() {
    long first, last, nBufferedMsgs, bufsize, nProdMsgs, nConsMsgs;

    int iBuffer=getProcessIndex("Buffer"); // We will use it often.

    // Number of messages in the rounded buffer
    first = getVarLongInt("first", iBuffer);
    last = getVarLongInt("last", iBuffer);
    bufsize = getVarLongInt("BUFFER_SIZE", iBuffer);

    if (last>=first) nBufferedMsgs = last - first;
    else nBufferedMsgs = bufsize-last + first;

    // Number of produced/consumed messages?
    nProdMsgs = getVarLongInt("i", getProcessIndex("Producer"));
    nConsMsgs = getVarLongInt("i", getProcessIndex("Consumer"));

    return (nProdMsgs - nConsMsgs) == nBufferedMsgs ;
}

int BufferEmpty() {
    long first, last;
    int iBuffer=getProcessIndex("Buffer");

    first = getVarLongInt("first", iBuffer);
    last = getVarLongInt("last", iBuffer);
    return first == last;
}

```

The `chk-ext/IFpred2diw.h` header contains the prototypes of functions, which can obtain state information concerning the P-GRADE application. At the beginning of function `NoLostItem` the index of Buffer process is stored, it will be used three times in the following codelines. Then the number of actually stored items is calculated based on the size of rounded buffer (`BUFFER_SIZE`) and the values of first and last integers. In the Buffer process the first variable stores the index of the first element in the rounded buffer, and the last shows the index where the Buffer process has to put the next item. The loop variable `i` represents the number of produced/consumed items in the Producer/Consumer process. The function `BufferEmpty` compares the value of first and last variables. If and only if they equal, the Buffer is empty.

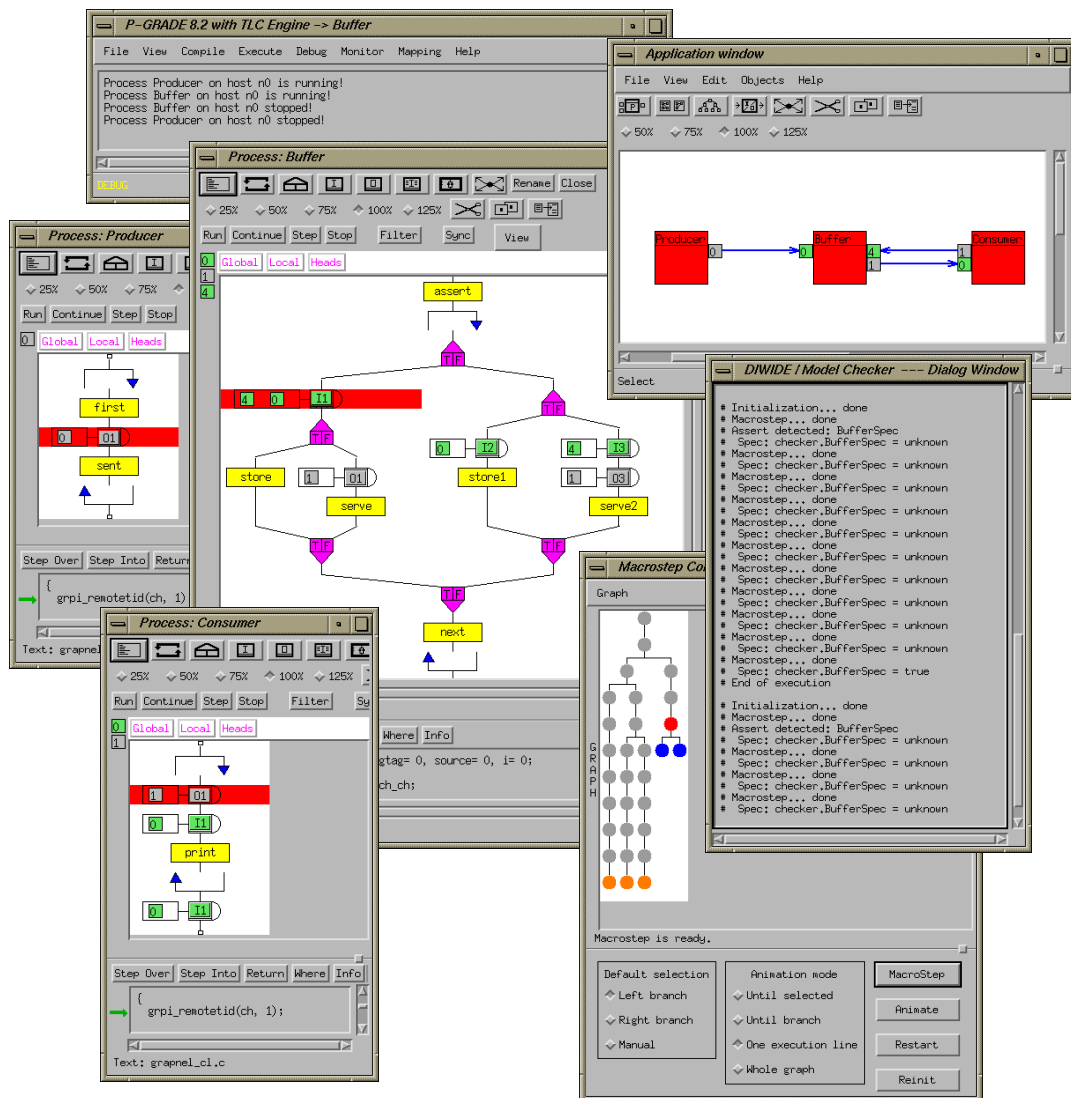


Figure 29 – Temporal Logic Checker and DIWIDE applied on Buffer application

For inspecting the system state, the atomic predicates use an interface to the debugger. For instance, the function `getVarLongInt(var, proc)` returns the value of the program variable `var` in process `proc` as a value of C type long. In this way, the predicate function `NoLostItem` checks the number of messages in the buffer process with respect of the values of two counter variables in the producer process and in

the consumer process. Summarizing, for using temporal formulas as runtime assertions, the programmer needs to

1. annotate the program to be debugged by the assertions,
2. provide a Java encoding of the temporal formulas,
3. provide C functions for the atomic predicates used in the temporal formulas.

This only reflect the current state of the system; in later versions, I plan to develop a meta-language where the Java encoding and the C functions are automatically generated from a high-level specification language.

As Figure 29 depicts, three different paths of the execution tree (see Macrostep Panel) are traversed and actually, the application is running for the fourth times. At this moment, each process is stopped after a successfully performed macrostep. The communication actions, where the processes are actually stopped, are marked with red background by the debugger. In the Model Checker Dialog Window the current state of the assertions can be inspected. Actually, the state of the specification is still unknown (due to the always operator used in the sample specification) but the previous execution met the requirements defined in the specification (see Figure 29, the upper part of Model Checker Window).

### 3.1.4 Checking Temporal Logic Assertions with TLC

The temporal logic checker TLC is an engine (implemented in Java), which interacts with an external partner (i.e. with a debugger) by a specified message protocol. The protocol operates in a sequence of rounds that correspond to the states of an execution sequence. In each round,

1. TLC may receive from its partner a new (additional) temporal formula whose validity is to be checked in the subsequent state,
2. TLC may ask its partner questions about the truth values of atomic formulas in the current state,
3. TLC announces its knowledge about the truth of the set of temporal formulas it has received up to now (true, false, unknown).

When a round has ended, the external partner informs TLC about the beginning of a new round (when a new consistent global state in the current execution sequence is available). TLC evaluates in each round the truth of all formulas with respect to the round in which the corresponding formula has been submitted by the external partner. If a formula refers to the future, the result will be frequently unknown. However, if more and more rounds are performed, the added knowledge may let the knowledge about such a formula change to true (the formula cannot be falsified any more after the current round) respectively false (the formula has been falsified by the current round). If a formula has been falsified, the corresponding assertion has been violated.

TLC does not repeatedly evaluate (sub)formulas whose final value (true or false) has been already determined. Such results are cached such that only those formulas are re-evaluated whose values are required to determine the value of the overall formula and whose status is still unknown.

In order to support temporal ‘past operators’, TLC prefetches in each round the values of all those atomic predicates whose results may be required in the future to evaluate a ‘past formula’. Whenever a formula including past operators is submitted to TLC, TLC records the atomic predicates in the scope of such operators in order to start prefetching the corresponding values. The ‘past’ of a temporal formula therefore extends only up to the round where the formula was submitted to TLC.

Currently, the only external partner of TLC is the DIWIDE debugger, which interacts with TLC as illustrated in Figure 29.

### 3.1.4.1 The evaluation protocol

The section outlines the protocol between TLC Engine and DIWIDE debugger as well as the type of messages. The description is given as a state machine where seven different states can be distinguished: *InitialState(1|2)*, *MainLoop*, *EvalLoop*, *EvalTermLoop* and *EndState*. In the *InitialState(1|2)* the checker and the debugger tell each other the protocol they know (see description of protocol message), and the debugger tells how many processes the P-GRADE program contains (see description of startup message).

In the *MainLoop* the debugger may assert new assertions, or tells whether a new program state (or the last program state) is available, see description of *assert*, *next* and *done* messages. If a new program state is available (i.e., checker receives a *next* message) and there is at least one assertion to be checked, then the *EvalLoop* takes place (for each assertion one *EvalLoop*). In the *EvalLoop* the checker asks the debugger what are the value of atomic predicates needed to evaluate the specification, see description of *eval* and *truth* messages.

When the checker needs to know the value of a term in order to evaluate the specification, then the *EvalTermLoop* takes place. The *EvalTermLoop* is similar to *EvalTerm*, the difference is the usage of *evalterm* and *value* messages, see description of them. The *EndState* is the final state of the state machine.

### 3.1.4.2 Formal description of protocol

In this section the formalised description of the outlined protocol (see Section 3.1.4.1) between temporal logic checker and macrostep-based DIWIDE parallel debugger tool are presented in table format. In the first column (*State*), the possible states are enrolled. The second column (*Input Message*) shows the possible input messages or message classes, while the corresponding output message and following state belongs to the the input message can found in the third (*Output message*) and the last (*Next-State*) columns, respectively.

State	Input Message	Output Message	Next-State
<b>InitialState</b>		<b>Protocol</b>	<b>InitialState1</b>
<b>InitialState1</b>	<b>protocol</b>	<b>Okay</b>	<b>InitialState2</b>
	<b>protocol</b>	<b>error failed</b>	<b>EndState</b>
	<b>not_expected_m</b>	<b>error failed</b>	<b>EndState</b>

	unregistered_m		EndState
<b>InitialState2</b>	startup	Okay	MainLoop
	Startup	error failed	EndState
	not_expected_m	error failed	EndState
unregistered_m		EndState	
<b>MainLoop</b>	assert	Okay	MainLoop
	Assert	error failed	MainLoop
	next {answer}	Eval	EvalLoop
	next {answer}	evalterm	EvalTermLoop
	next {answer}	Okay	MainLoop
	next {answer}	error failed	MainLoop
	done {answer}	Okay	EndState
	abort		EndState
	not_expected_m	error failed	MainLoop
	unregistered_m		EndState
<b>EvalLoop</b>	truth	{answer} eval	EvalLoop
	truth	{answer} evalterm	EvalTermLoop
	truth	answer {answer} okay	MainLoop
	truth	answer {answer} error failed	MainLoop
	abort		EndState
	not_expected_m error	Failed	EvalLoop
	unregistered_m		EndState
<b>EvalTermLoop</b>	value	Eval	EvalLoop
	value	evalterm	EvalTermLoop
	value	error evalterm	EvalTermLoop
	abort		EndState
	not_expected_m error	Failed	EvalTermLoop
	unregistered_m		EndState
<b>EndState</b>			

Note: {} means 0 or more items.

### 3.1.4.3 Messages types of protocol

This section describes in details the message types used by the proposed protocol (see 3.1.4.2). The table summarizes for each message types (first column) the possible sender party (i.e. the direction of the message), a short description, and its parameters.

MESSAGE TYPES	Sender	Description	Parameters
<b>abort[]</b>	D	TLC aborts when it receives this message.	
<b>answer[<i>arg1</i>, <i>arg2</i>]</b>	C	It tells the truth value of an assertion in the current state. This message is sent by TLC as a response to an assert, next or done message. After an assert message TLC sends one answer message. After a next or done message TLC sends as many answer messages as the number of unknown assertions enrolled in the previous state.	<i>arg1</i> : <i>String</i> Name of the specification.  <i>arg2</i> : <i>{true, false, unknown}</i> Truth value of the specification.
<b>assert[<i>arg1</i>]</b>	D	DIWIDE asks TLC to check an assertion beginning from the current state. TLC will respond with an answer message (if there was no error) and an okay message.	<i>arg1</i> : <i>String</i> Name of the specification to be checked.
<b>done[]</b>	D	It tells TLC the last program state is available.	-
<b>error[<i>arg1</i>]</b>	C	It is sent by TLC, if an error occurs.	<i>arg1</i> : <i>String</i> Explanation of the error.
<b>eval[<i>arg1</i>, <i>arg2</i>, ...]</b>	C	TLC asks DIWIDE to evaluate a predicate described by the given arguments. DIWIDE responds by a truth message. It is a varying size message.	<i>arg1</i> : <i>1..MaxInt32</i> Number of parameters of this (varying size) message.  <i>arg2</i> : <i>String</i> Name of the predicate.  <i>... : Int32</i> Parameters of the predicate. Note that <i>arg1</i> is 1 + number of parameters of the predicate.
<b>evalterm[<i>arg1</i>, <i>arg2</i>, ...]</b>	C	TLC asks DIWIDE to evaluate an application function described by the arguments. DIWIDE responds by a value message. It is a varying size message.	<i>arg1</i> : <i>1..MaxInt32</i> Number of parameters of this (varying size) message.  <i>arg2</i> : <i>String</i> Name of the application function.

MESSAGE TYPES	Sender	Descripton	Parameters
			<i>... : Int32</i> Parameters of the application function. Note that arg1 is 1 + number of parameters of the application function.
<b>failed[]</b>	C	It is sent if the message sent by DIWIDE was unsuccessful.	-
<b>next[]</b>	D	It tells TLC there is a new state available.	-
<b>okay[]</b>	C	It is sent if the message sent by DIWIDE was successful.	-
<b>protocol[arg1]</b>	C	The sender tells what protocol it uses.	<i>arg1: String</i> The protocol which is used by the sender.
<b>startup[arg1]</b>	D	DIWIDE tells the startup parameters to TLC.	<i>arg1: 0..MaxInt32</i> Number of processes of the debugged parallel program.
<b>truth[arg1]</b>	D	DIWIDE tells TLC the truth value of the predicate that is described by the eval message sent by TLC.	<i>arg1: {true, false, unknown}</i> Truth value of the predicate described by the eval.
<b>value[arg1]</b>	D	DIWIDE tells TLC the truth value of the application function that is described by the evalterm message sent by TLC.	<i>arg1: Int32</i> Value of the application function described by the evalterm.
<b>not_expected_m</b>	N	Not expected message. Can stay for any message that is not listed in the state in the Input Message column. For example; the value message in the MainLoop state.	-
<b>unregistered_m</b>	N	Unregistered message. Can stay for any string, which doesn't correspond to a message.	-

Note: in the *Sender* column the following acronyms are used:

C: temporal logic Checker Engine

D: Diwide distributed debugger

N: Not specified

### 3.1.5 Run-time evaluation of TL assertions in P-GRADE framework

The integration of TLC engine and DIWIDE distributed debugger required the introduction of a new component in the P-GRADE framework, called Checker Module (CM). The main task of Checker Module is allowing the collaboration between the macrostep debugger and the temporal logic model checker (TLC)

engine, on the other hand, the Checker Module is responsible for loading dynamically and also for handling the predicate libraries that can be referenced from the specification. The current implementation of Checker Module provides also a simple graphical user interface where the programmer/tester can inspect the state of temporal logic specifications macrostep by macrostep.

### 3.1.5.1 Initialization phase

In details; the connections between the different components during the start-up phase (initialization and assert handling) as well as the major interactions in order of time can be seen in Figure 30.

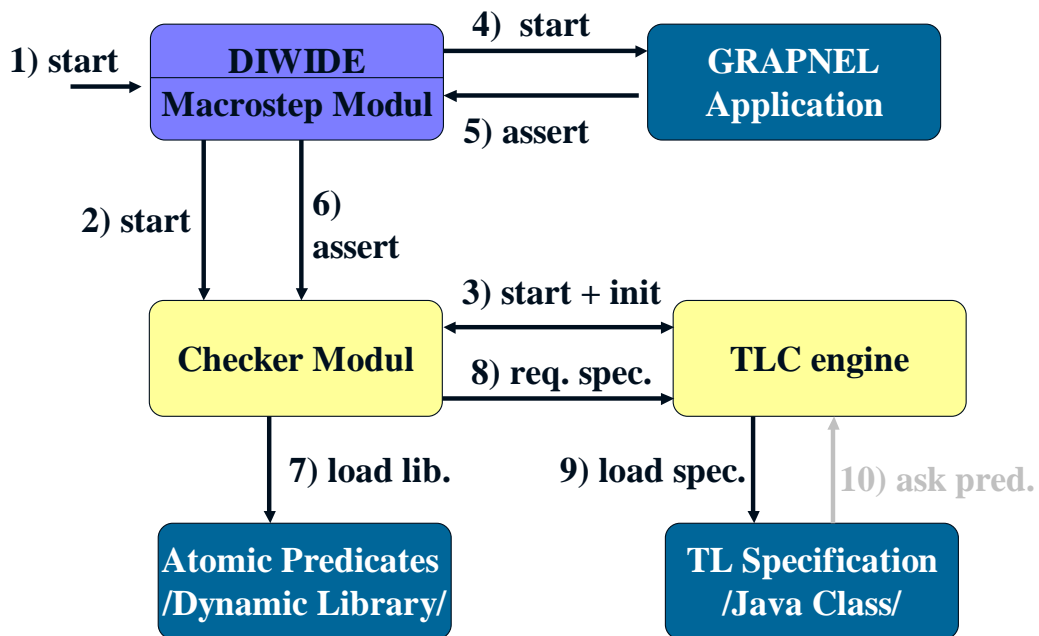


Figure 30 – Initialization and assert handling of temporal logic checker

First of all, the macrostep module launches a new Checker Module that is implemented as a C++ class. As the second step, the Checker Module spawns a new JAVA application, the TLC engine as a separate process. The Checker Module wraps into the newly launched JAVA application; redirects both the input and the output of the engine, and communicates with it via pipes. This is one of the easiest/fastest way of communication with a JAVA process running on the same host. As the last step, the Checker Module and the TLC engine has to negotiate the actually used protocol and the number of processes that will be started in the current P-GRADE application. (It is required for the initialization of TLC engine).

### 3.1.5.2 Assert detection

After the initialization phase, the user can start the P-GRADE application and usually the program reaches the end of the first macrostep after a certain period of time. Then, Checker Module scans through each process whether a run-time assertion has been activated (by a GRP\_ASSERT icon placed in the P-GRADE application) during the last executed macrostep or not. The names of activated assertions are stored separately in the processes address space (arrays of strings) and the Module Checker can obtain the information via the expression evaluation

facilities of DIWIDE debugger. If a new assertion is encountered, Checker Module passes the name of the assertion formula to the TLC engine. The TLC engine then loads the corresponding Java class and starts the evaluation of the formula. At the same time, the CM also loads the corresponding predicate library that must be implemented as a dynamic library by the developer.

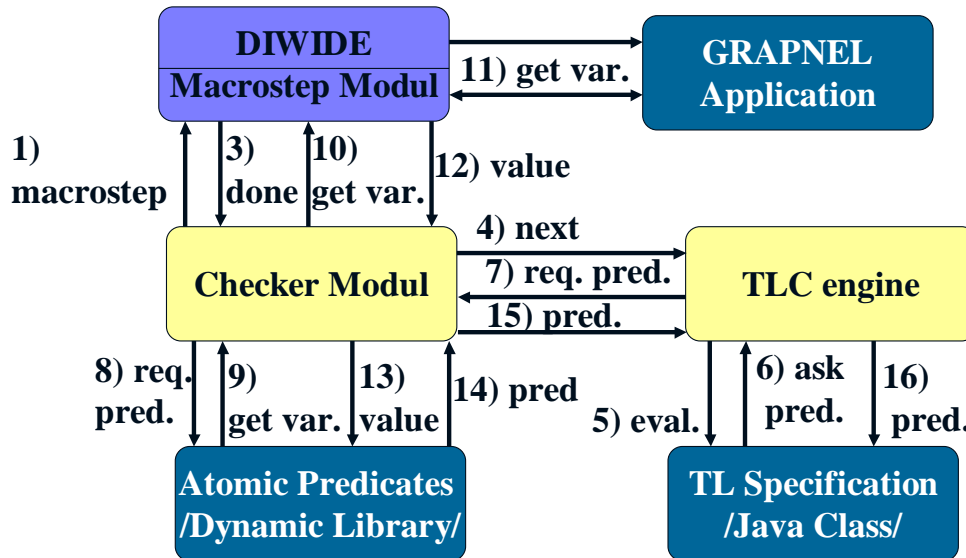


Figure 31 – Evaluation of atomic predicates

The above described pooling mechanism is repeated each time when a macrostep is done in order to find the user's assertions.

### 3.1.5.3 Evaluation of atomic predicates

As it described in the protocol (see Section 3.1.4) in any macrostep the TLC engine may require the value of an atomic predicate from the Checker Module. For the evaluation of temporal logic expressions (written in the specification) and user-defined atomic predicates (referenced from the specification) must be also evaluated. The atomic predicates are placed separately in the dynamically loaded predicate libraries. Basically, the evaluation of atomic predicate means the invocation of the corresponding C function from the predicate library that function represents the specified atomic predicate. In a given atomic predicate different kinds of function calls can be used for obtaining the necessary information about the state of processes, such as the current value of a specified program variable or the labels of actually executed communication actions, etc. These commands are usually executed via the CM relying on the expression evaluation facilities of DIWIDE. The return value (true or false) is passed to the TLC engine via the described protocol (see Section 3.1.4). Of course, the engine can also request the value of any program variable.

### 3.1.5.4 Atomic predicate – debugger interface

Functions supported in predicates (where the procindex is a unique identifier for processes):

```

char* CM_getVar(char* varname, int procindex);
    // get the value of the given variable, return with a gdb formatted string
long CM_getVarLongInt(char* varname, int procindex);
    // get the value of the given variable, return with a long integer
int CM_atLabel(char* label, int procindex);
    // return TRUE, if the process stopped at the communication action, labeled with 'label'

int CM_getProcNumber();           // get the number of all processes
int CM_getProcessIndex(char* procname); // get the index of the given process
int CM_getMessageNumber(int, int); // get the number of messages
char* CM_getMessage(int, int, int); // get the contents of the given message

```

### 3.1.6 Optimisation issues of model checking based on assertions

As it was described earlier, TLC does not repeatedly evaluate (sub)formulas whose final value (true or false) has been already determined. Such results are cached, only those formulas are re-evaluated whose values are required to determine the value of the overall formula and whose status is still unknown.

One of the most time consuming task is the evaluation of program variables, which referred from the temporal logic specification. To obtain a single program variable requires numerous communication steps among the different modules of DIWIDE debugger tool, TLC engine, and the executed program via the interconnecting network. Hence, I focused on the elimination of the extra overhead caused by the evaluation of program variables.

First, I proposed a *caching technique* for program variables and values. The variable-value pairs, which are already evaluated in a consistent global state, are stored in a cache separately for each process. Often, only a few process pairs are able to communicate with each other during a macrostep, and the remaining processes are blocked on a sleeping breakpoint (see Section 2.2.2) without any changes in their local process states. Before the execution of the following macrostep, every cache has to be cleaned which belongs to a non-blocked process, i.e. stopped on an active breakpoint (see Section 2.2.2). At the next meta-breakpoint the TLC engine usually asks the same variables (as in the previous consistent global state) from a “sleeping” processes. Obviously, these variables can be obtained much faster from the cache, which belongs to the process. In this way, the debugger can save valuable time macrostep-by-macrostep.

In fact, TLC engine or Checker Module asks the referred program variables separately from each other during the evaluation of temporal formulas. As a second step, I introduced a *pre-fetching technique* for frequently referred program variables, which belong to the same process. It must be taken into consideration due to the high communication overhead (particularly the long latency time) when the debugger evaluates a single variable. In the current implementation, I proposed a simple algorithm: a variable is marked as “frequently used” if it has been referred from the specification (1) at each of three consecutive meta-breakpoints or (2) at

least of the certain percent of meta-breakpoints. Then, the debugger automatically pre-fetches all the marked variables *together* at the beginning of the actual evaluation phase. At a sleeping breakpoint the debugger can obtain the values of the frequently used variables from the cache, as it was described earlier.

### 3.1.7 Summary: Future work and open issues

In this thesis, I presented the way in which *a particular class of temporal logic expressions (LTL-x) can be evaluated on the paths of Execution Tree during macrostep based execution.*

Concerning the future work; the current external interface of TLC has been targeted towards the current functionality of the DIWIDE debugger and does not yet make full use of all facilities of TLC and its underlying framework. In particular,

1. TLC could inform its external partner not only about the overall value of an assertion but about the value of each subformula with respect to the state in which it is required to determine the overall result. This information can be used in a graphical user interface in order to browse through states to determine, which formulas are evaluated in which state and what the current knowledge about the value of the formula in which state is. This will give the user much more information about the overall behaviour of the program with respect to the asserted formula than the current interface.
2. TLC need not operate in rounds and evaluate the formula in a single sequence of states. TLC actually maintains a tree of states as suggested by the theoretical framework sketched in Section 3.1.2.1. It is at any time possible to add a child to any node in the tree and to re-evaluate the truth of a formula with respect to the whole tree. TLC could therefore interact with a debugger which maintains more than one system state and gives the user the possibility to further execute the program from any of these states.

With the described framework, it is possible to assert temporal formulas and have their validity checked in (manually or automatically) selected runs of a parallel program. However, from a practical point of view, an adequate graphical user interface is not yet provided that allows the programmers in an intuitive way to determine the fundamental reason why an assertion has failed respectively is not yet satisfied (i.e., which subformula in which particular state is the core of the problem).

Second, there is a lack of a high-level specification language from which the low level encodings of temporal formulas (as Java objects) and atomic predicates (as C functions) are automatically generated. In the experimental framework, it is the responsibility of the programmers.

Third, and most important, we need to evaluate by larger program examples with interesting properties to which extent in practice the use of temporal assertions actually helps to improve the understanding of program behaviours and detect errors in them. In any case, the presented system will serve as a good starting point for these investigations on the usefulness of extending a parallel debugger with model checking capabilities in real-size applications.

Concerning the long-term plan, the future work should deal with generalisation of the results, e.g. using Unified Modelling Language (UML). Statechart Diagrams provide a graphical notation for describing dynamic aspects (e.g. concurrency, non-determinism, and priority) of system behaviour within the UML. In [41] a translation method is presented from a subset of UML Statechart Diagrams into PROMELA, the specification language of the SPIN model checker. SPIN is one of the most advanced analysis and verification tools, and this translation allows for the automatic verification of UML Statechart Diagrams. According to [41] the translation itself was simple, proven correct, and efficient in terms of state space representation. Thus, its further investigation and combination with the techniques presented in this thesis could be a promising approach to provide a more efficient way for debugging.

## **3.2 Further optimisation of macrostep-based debugging**

### **3.2.1 Preface**

The macrostep-based execution of parallel program with model-checking (see Section 3.1) is not optimal; some feasible optimisation methods can be applied in order to accelerate the macrostep-based debugging cycle. The most important approach is the integration of a coloured Petri-net (CPN) simulation engine into the debugging framework relying on the CPN model of GRAPNEL\* programs. In this thesis, I present that the *CPN simulation engine is able to steer the macrostep-based traversal of state-space* (the building of the Execution Tree) *towards erroneous situations during the debugging phase, and to detect the already traversed execution paths*. The presented method (see Section 3.2.2) can assist the software developer to find programming bugs by its simulation & steering techniques. As a part of my work, some other techniques are also proposed and investigated in order to enhance the debugger framework. In the rest of this section, they are outlined together with the CPN simulation method from the users' point of view.

The first optimisation methods is based on the partitioning of GRAPNEL applications to get smaller code segments, which must be tested and debugged as one unit (see Section 3.2.3.1). After partitioning, the simultaneous execution of debugging sessions, i.e. execution paths can be performed in parallel utilising the available distributed computational resources (see Section 3.2.3.2) while the on-the-fly or off-line Petri-net simulation of the program steers the execution towards the erroneous situations (see Sections 3.2.2.2, 3.2.2.2). The Petri-net simulation can be shortened by automatic termination in case of the violation of given temporal logic specification (see Section 3.2.2.3) and the number of required sessions can be reduced if the analyser is able to recognise the earlier discovered states of execution paths (see Section 3.2.2.4). Finally, the entire debugging cycle can be completed if the program's reliability achieves the satisfied level, which is estimated by the help of Rayleigh-model (see Section 3.2.3.3). I also presented a method, which may improve the reliability of the released program (in certain cases) forcing its states remaining inside the previously tested state-space (see Section 3.2.3.4).

### 3.2.2 Petri-net simulation for steering the execution

The temporal logic checker (see Section 3.1) deals with the history and the current states of individual processes but the checker has no knowledge on the future of the running program. To improve efficiency and capabilities of the macrostep-based debugging methodology, a coloured Petri-net simulation, GRSIM engine can be integrated into the debugger framework that is able to simulate, and also to analyse the possible future states of the executed program as it described in Sections 3.2.2.1, 3.2.2.2. Based on the simulation and analysis the macrostep-based execution can be steered towards to the suspicious directions assisting the users to focus on the erroneous situations.

The pure Petri-net simulation and analysis of entire program is usually not feasible due to the combinatorial explosion of program states, and the simulation is based on the model of the program that neglects numerous physical constraints and real programming bugs. On the other hand, the simulation can traverse the different program states much faster than the real execution by orders of magnitude, and the advantage of this fast simulation can be taken during the idle periods of macrostep-by-macrostep or in off-line mode.

Taking into consideration to above introduced issues, in the experimental framework two scenarios are proposed to get use of OCC graph:

- Off-line simulation and steering
- On-the-fly simulation and steering

#### 3.2.2.1 Off-line simulation and steering

In this case, the OCC graph for a coloured Petri-net is constructed by the simulator independently from the actual debug session. The standard reporting facilities of the simulator (e.g. Design/CPN simulator) can be utilized in order to generate a standard report providing information about:

- Statistics (e.g. size of Occurrence Graph)
- Boundedness Properties (integer and multi-set bounds for place instances)
- Home Properties (home markings)
- Liveness Properties (dead markings, dead/live transition instances)
- Fairness Properties (impartial/fair/just transition instances)

The contents of the report file can be interpreted and translated automatically to GRAPNEL program behaviour properties, especially keeping a close watch on suspicious situations.

One of the main goals is to detect deadlocks, which are equivalent of dead markings in the OCC graph. For all dead markings (*ListDeadMarkings*) the GRSIM calls the *Reachable* function that determines whether there exists an occurrence sequence from the marking of the first node (the actual or initial marking) to the marking of the second node. It means the search in OCC graph to find a directed path between these nodes. When this search is finished, GRSIM gains information about the paths leading to deadlock situations.

The syntax of the output of our queries (the paths) is defined by Design/CPN. The GRSIM uses these paths by *converting* them to a standard form (specified by the input/output interface of macrostep debugger) that allows the user to replay the execution-path from an input file. During this conversation, GRSIM traverses the nodes the OCC path and also converts the proper states into the standard file form of execution tree. For this purpose, the path is segmented into series of nodes, which are corresponding to a macrostep, taking into consideration the transitions, which represent message passing (particularly where an alternative input receives a message). Relying on the cross-reference file, which is generated during the Petri-net model generation, the segments of OCC path (the reachable coloured Petri-net markings) are translated back and stored as the nodes of execution tree (reachable states of executed program). While the user replays the execution macrostep-by-macrostep through the path ending in deadlock searching for the cause of deadlock, it is possible to inspect the actual values of variables, the composition of stack, the instruction pointer in every process with DIWIDE debugger.

### 3.2.2.2 On-the-fly steering of debugging

During the execution of each macrostep the simulation engine has to build up an undiscovered part of the OCC graph based on the Petri-net model of GRAPNEL program. On the other hand, using OCC graph analysis the simulation engine can steer the traversing of Execution Tree and can direct the user's focus to deadlocks, wrong terminations, and other erroneous situations that may occur in the future. The starting point of the Petri-net simulation (the first marking from where the simulation starts) is always related to the current consistent global state, i.e. the current node of the Execution Tree that is discovered by the macrostep engine using a depth-first searching strategy<sup>8</sup>. Then the simulation is running concurrently with the real program execution until the next macrostep starts. During the simulation an undiscovered sub-graph of OCC graph is generated automatically applying a breadth-first searching strategy since it cannot be predicted easily, which are the most possible timing conditions (occurring in the future). The simulator is able to detect some simple classes of erroneous situations that require low-cost analysis, such as deadlocks or wrong process terminations. Meanwhile, the analyzer is trying to find other erroneous situations (which require deeper analysis) in the OCC subgraph generated during the *previous* macrostep. When either the simulator tool or the analyzer tool detects an erroneous situation, the macrostep engine gets a message on the type of error and the list of timing constraints (using the conversation technique described in 3.2.2.1 ) that lead to the erroneous situation. Thus, the macrostep engine can steer the program execution towards the erroneous node of Execution Tree, and the user can uncover the reasons of the error deploying the distributed debugging facilities of DIWIDE debugger.

### 3.2.2.3 Termination conditions for steering

The Coloured Petri-net model of GRAPNEL applications simulates the changes of global control variables by global control tokens. Hence, they are good candidates to be refereed in termination conditions, where the simulation must

---

<sup>8</sup> The breadth-first search would require a tremendous cost due to the difficult recovering of previous (global) program states.

stopped because of the violation of specification. Two options are investigated for this aim:

- Invocation of an enhanced temporal logic checker as an external tool
- Evaluation of TL specification relying on the built-in facilities of the simulation tool

In the *first case*; the PN simulation engine can interact a temporal logic checker similarly to the integration of TLC and DIWIDE (see Section 3.1). During the simulation, the advantages of the facilities of TLC engine can be utilised but some requirements must be taken into consideration.

The state-space, where the temporal logic expression is evaluated, includes the OCC graph nodes, which are related to consistent global states defined by the macrostep execution. Obviously, the assertions may refer only to the values of global control tokens, which are defined in the coloured Petri-net model. By each assertion a simple parser can check, whether the temporal logic expression of the user-defined specification meet the above described requirements or not. If the expression is passed, the CM dispatches the assertion to the GRSIM engine for evaluation at each macrostep.

Regarding the *second case*; several PN simulation tools (e.g. Design/CPN [14]) allow the user to define global predicates as termination conditions using traditional or sometimes temporal logic.

The expressiveness of temporal logic is more powerful (see Section 3.1) for debugging and testing purposes but the computing and memory demands of the evaluation of temporal logic expressions are higher. That is why, some temporal expressions can be selected, which apply only the *always* operator. Thus, the user gets traditional logic expressions<sup>9</sup>, which can be transformed to the specified format of PN simulation engine but it is not a straightforward direction.

Another option is to create custom queries derived from the user's TL specification (see Section 3.1.3.1). In the rest of this section, the meta-language and built-in functions of Design/CPN [14] are used for illustration purposes.

The base function to take into consideration is *SearchNodes* (see Figure 32), which traverses the nodes of the OCC graph. At each node, the specified calculation is performed and the results of these calculations are combined, in the specified way, to form the final result. The converted form of the negated temporal logic expression must be evaluated to *true* as the *Pred* parameter of *SearchNodes*. The conversion is needed because the introduced TL specification (see Section 3.1.3.1) is higher description level of program expected behaviour than the approach followed by Design/CPN.

The *SearchNodes* function can be invoked with the following parameters: *SearchNodes(EntireGraph, Pred, NoLimit, fun Eval(n) = n, [], fun Comb(new,old) = new::old)*. As the result of this method, we get all the places, where TL specification fails. From this stage, the way of steering is the same as it was described in Section 3.2.2.1.

---

<sup>9</sup> Another requirement is that the assertions may refer only to global control tokens, which tokens are defined in the Petri-net model.

```

SearchNodes (Area, Pred, Limit, Eval, Start, Comb)
begin
  Result := Start; Found := 0
  for all n ∈ Area do
    if Pred(n) then
      begin
        Result := Comb(Eval(n), Result)
        Found := Found + 1
        if Found = Limit then stop for-loop
      end
    end
  end
end.

```

**Figure 32 – Meta-code of *SearchNodes* function**

### 3.2.2.4 Detection of discovered execution paths

In practice, the most crucial disadvantage of execution trees is the lack of facilities for detection of already discovered and traversed execution paths.

The goal of this optimisation method is the detection of those situations, when a program state (or a sub-graph starting from the given state) has been already discovered and traversed (at least partially) in another execution path. At the design stage of the original macrostep concept this problem was out-of-scope, but now it can be managed with the introduced Petri-net simulation; during the generation of OCC graph, the Petri-net engine detects this kind of situations and also merges the different paths if the markings (program states) are exactly the same ones. Using this information by the building of execution tree, the *Execution Graph* (derived from the execution tree, see Section 2.2.2) can be introduced, and the number of states can be reduced radically that must be traversed and inspected by the macrostep debugger. Assuming the Petri-net model of GRAPNEL program is not available, the Execution Graph can be generated by the extension of already described pre-fetching technique (see Section 3.1.6). In this case, the debugger obtains the current values of *all* the relevant program variables (enrolled as CONTROL variables, see Section 2.1.2.1) and stores them in an associative memory structure at each meta-breakpoint. Later, when an already discovered program state occurs again, the debugger detects this situation and merges the different execution paths relying on this information.

### 3.2.2.5 Reduced CPN model

In general, the two upper levels of GRAPNEL hybrid language can be transformed easily to CPN but the transformation of the lowest level, which contains C source code, is much more difficult without users' interaction due to the differences between the C programming language and the meta-language of CPN model.

We can get a (reduced) CPN model if we left the transformation steps of C-related parts of the GRAPNEL program, i.e. in case of conditional branches the guards of transitions are missing, the global control token has no type, etc.

Taking into account the above described rules, and applying flat model instead of the hierarchical approach, the Buffer application can be represented by a simpler “model” (see Figure 33).

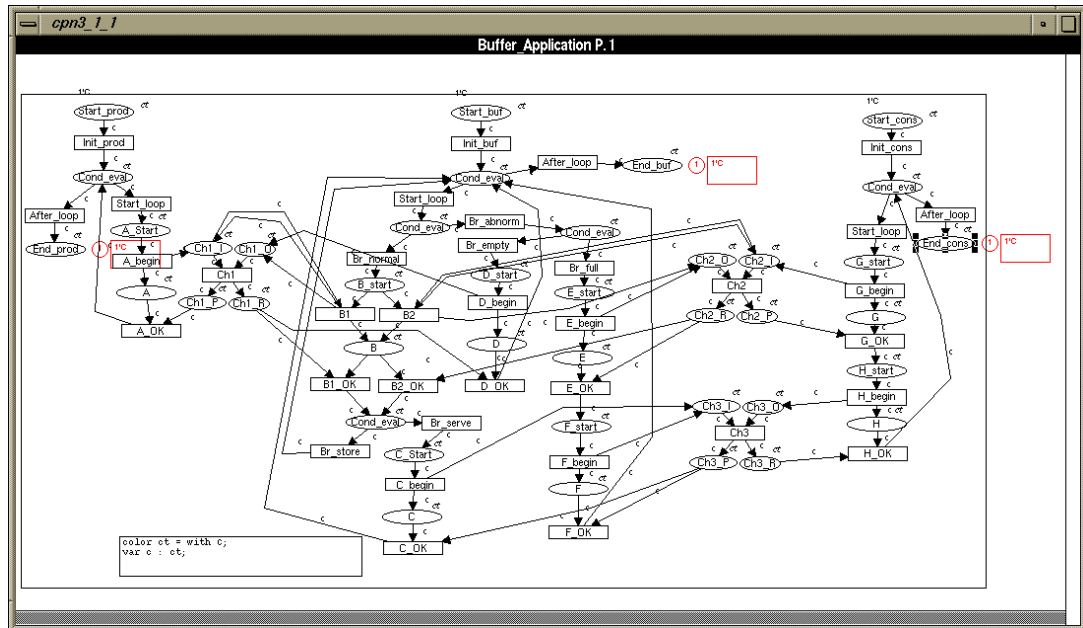


Figure 33 – Reduced CPN model of Buffer application

It can be seen, we get a traditional Petri net, where the execution can be continued on either branch of conditional constructs, and the number of iterations are not defined. Hence, the given model contains *all* the possible states, which are allowed by the *graphically* described program structure, *without* taking into account the actual values of program variables. (The correctness of the generated code based on the graphical parts is itself proven []). So the OCC graph will contain “real” and “fake” states as well, where real states have corresponding program states, while the fake states are the results of the reduced model without corresponding program state.

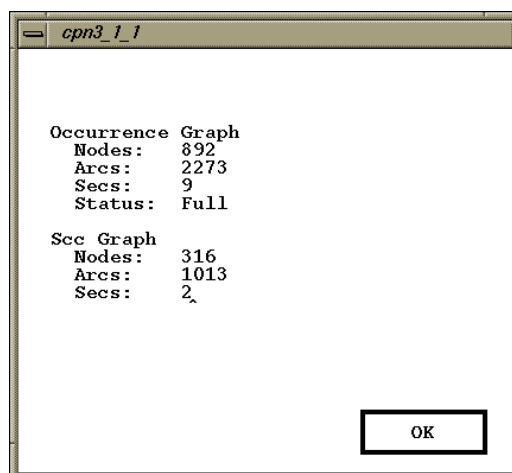


Figure 34 – Reduced occurrence graph

By inspecting the OCC graph of Buffer application (see Figure 34), it is turned out, the number of the possible token distributions less than in case of original model since the loops and conditional constructs are not represented, and the number of fake states do not increase significantly the number of nodes in OCC graph.

Based on the generated report, 19 dead markings were found, one of them occurs in case of proper termination (119<sup>th</sup> node of OCC graph). In the remaining 18 cases, the system gets in deadlock situation sometimes caused by early process termination. By the help of the following query (see Figure 35) relying on the *SearchNodes* and the *length* functions the number of token distributions can be determined from which the proper termination are not reachable. Currently, we got 182 such nodes.

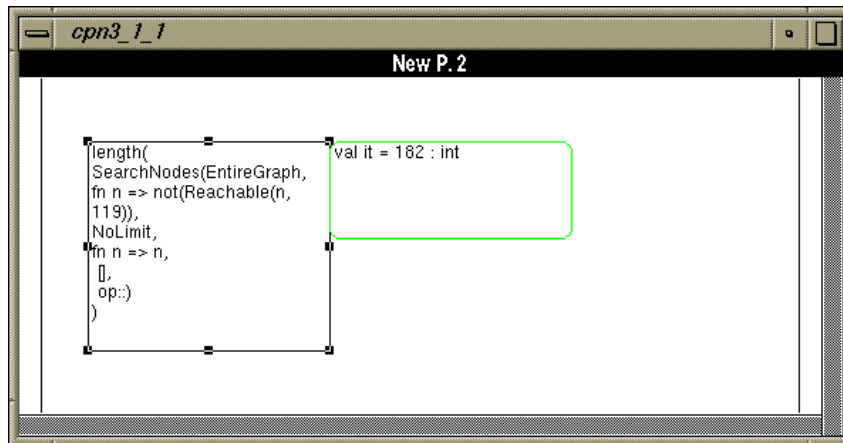


Figure 35 – Analysis of reduced occurrence graph

The advantages of this simple model and analysis can be taken during the steering of macrostep-based execution. If the current state during the real execution of the program belongs to a node of OCC graph, which leads to surely deadlock situation, the system can warn the user in advance saving up valuable time since complex sequential code segments can take a long time during the real execution of the program.

### 3.2.3 Partitioning, execution and releasing of GRAPNEL programs

#### 3.2.3.1 Partitioning of GRAPNEL programs for macrostep execution

GRAPNEL programs can be message-dependent and message-independent applications depending on their control-flow graph and exchanged messages.

##### 3.2.3.1.1 Message-independent applications

Definition. A GRAPNEL program is called *message-independent* iff the usage of global control protocols are permitted. In other words, the actual values of global control variables can be derived from their initial values, and they do not depend on the contents of the received messages.

For example, a message independent application is the already presented *Buffer* application; the control-flow is independent from the value of produced, buffered, and consumed items.

Definition. A GRAPNEL process in a message-independent application can behave in *deterministic* or *non-deterministic* manner. The process is deterministic if its control flow graph does not contain any CAIALT operation, and non-deterministic when at least one CAIALT can be found in the process.

For example, deterministic processes are the *Producer* and *Consumer* processes, while the *Buffer* process is non-deterministic in the *Buffer* application.

A deterministic process sends and receives always the same messages in the same order. Thus, we can cut the application into two disjunctive process sets:  $PS_d$  (set of deterministic processes) and  $PS_{nd}$  (set of non-deterministic processes). The communication channels, between the two sets of processes, are called as interface channels.

After partitioning, one successful execution of GRAPNEL application is needed to store the messages passed via the interface channels towards the processes of  $PS_{nd}$ , and also the message requests from the processes of  $PS_{nd}$ . Later, the macrostep controller launches only the processes of  $PS_{nd}$ , and feeds the processes with the stored messages.

An erroneous situation may arise if there is no available message in the message queue of an interface channel contrary to a CAI or CAIALT operation. On the other hand, if a not expected message is sent from any process towards the set of  $PS_d$ , another synchronisation error occurs. By the end of execution the message queues in both directions must be empty, otherwise a termination error occurs.

### 3.2.3.1.2 *Message-dependent applications*

Definition. A GRAPNEL program is called *message-dependent* iff the usage of global control protocols is allowed. In other words, the actual values of global control variables cannot be derived directly from their initial values, since they may depend on the contents of the received messages.

For example, a message dependent GRAPNEL application is the slightly modified version of the already presented *Buffer* application; the number of items are not given, only the last item is marked by a special sign (e.g. zero) and the processes must terminate after the consuming/buffering/processing of this item.

Definition. A GRAPNEL process in a message-dependent application can behave in *deterministic* or *non-deterministic* manner. The process is deterministic if its control flow graph does not contain any CAIALT operation nor CAI operation, which can receive message from non-deterministic process via global control protocol. A process is non-deterministic when at least one CAIALT can be found in the process or a CAI operation, which can receive message from non-deterministic process via global control protocol.

For example, the only deterministic processes is the *Producer* process, while the *Buffer* and *Consumer* processes are non-deterministic in the modified version of *Buffer* application since the *Consumer* process gets messages from a non-deterministic source; from the *Buffer* process.

Since the definition of non-deterministic processes is given in recursive way, the selection method is recursive, too.

Similarly to the previous case, a deterministic process send/receive always the same messages in the same order, and the application can be split up two disjunctive process sets:  $PS_d$  (set of deterministic processes) and  $PS_{nd}$  (set of non-deterministic processes). The communication channels, between the two sets of processes, are called as interface channels, again. Contrary to the message-independent applications, the interface channels have only one direction, from  $PS_d$  to  $PS_{nd}$ .

After partitioning, one successful execution of GRAPNEL application is needed to store the messages passed via the interface channels towards the processes of  $PS_{nd}$  (the message requests from the processes of  $PS_{nd}$  are missing from this partitioning method). Later, the macrostep controller launches only the processes of  $PS_{nd}$ , and feed the processes with the stored messages.

An erroneous situation may arise if there is no available message in the message queue of an interface channel contrary of a CAI or CAIALT operation. By the end of execution the message queues must be empty, otherwise a termination error occurs.

### **3.2.3.1.3 Summary**

In summary, the described partitioning mechanisms are feasible if the developer wants to save up some computing power and testing time, but the mechanisms require more disk space to store the messages passed between the partitions. Moreover, at least one successful execution is also needed therefore; the partitioning mechanisms cannot be applied from the beginning of the testing/debugging stage in general.

### **3.2.3.2 Execution on distributed platforms**

The introduced debugging framework can be executed on a dedicated workstation/PC cluster as a well-balanced concurrent system, itself. The temporal logic assertions are evaluated *between* two consecutive macrosteps, and the Petri-net simulation and analysis are executed *during* the macrosteps in case on-the-fly steering. That is why, it is recommended to map the Petri-net simulator and TLC engine together on a dedicated workstation separately from the processes of the user's application. When the user's processes communicate via the network (during a macrostep), the Petri-net simulation does not require massive communication. On the other hand, between macrosteps the processes are blocked on a meta-breakpoint (communication is not allowed between the processes), and the Checker module can obtain the program variables efficiently via the network without sharing the network with user's activities.

The time requirements of the tests can be decreased by magnitude of orders taking the advantage the large number of resources included in a metacomputing environment [45] by starting more test scenarios at the same time.

### **3.2.3.3 Rayleigh model for estimation of error density**

Practically, the complete exhausted testing and the elimination of all defects of a real size program is almost impossible. Hence, the debugging and testing phase, which performed over the implementation and testing phases of

software development life-cycle, must be stopped at a certain point, where the quality of the software is reliable enough to release as a beta or a final release.

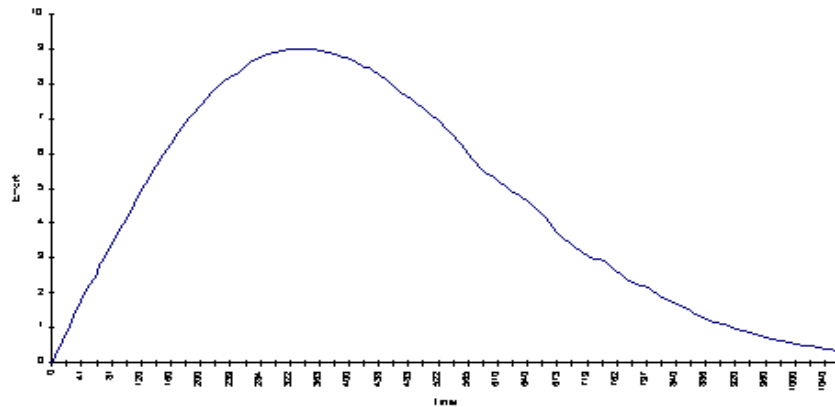
To make correct estimations the macrostep-based debugging methodology integrates some widely accepted metrics and estimation methods into the macrostep-based methodology.

The integrated defect monitoring subsystem uses the Rayleigh model [18] to forecast the discovery rate of defects as a function of time during the proposed software development lifecycle (see Figure 8). Empirical evidence shows the Rayleigh model is able to approximate closely the actual profile of defect data collected from software development efforts. The generic form (see below) of the Rayleigh model is tuned (see  $d$  and  $\alpha$  parameters) using the actual collected defect data reported and stored in a database. The model then forecasts the defects that remain in software product and the key dates when specific levels of reliability will be very likely achieved<sup>10</sup>.

The general equation, Cumulative Distribution Function for the model is

$$\text{CDF: } R(t) = d(1 - e^{-\alpha t^2})$$

where:  $t$  = time,  $d$  = the scale factor of the distribution,  $\alpha$  = the shape parameter  
 $R(t)$  = the total effort expended on the project (see Figure 36).



**Figure 36 – Cumulative Distribution Function (example)**

Simple further extensions of the model can provide other useful information for the developers. For example, defect priority classes can be specified as percentages of the total. This allows the model to predict software defects by severity classes over time. The tuning for the defect classes can be made relying on the actual reported defects and adjusted as the software development progresses. The reported software defect categories are:

1. Critical
2. Major
3. Minor

---

<sup>10</sup> Where very high reliability is required this is the point in time when 99.9 % of the theoretical software defects have been discovered.

Purchasing or QA department can set the (contractual) quality targets in terms of the software mean time to defect (MTTD). The MTTD is simply average time between software failures, and calculated as the reciprocal of the average number of defects within a given period<sup>11</sup>.

Defects are tracked and forecasted throughout the development and macrostep-based debugging by the help of Rayleigh analyser. Product release acceptance begins when the release meets the MTTD acceptance criteria. Quality continues to be improved by the e.g. supplier after the first delivery.

In practice, usually three MTTD quality targets are set:

1. RFA: Ready for Acceptance, marks the first delivery to the purchaser to start a so-called factory acceptance test (typically this is a MTTD of 5 days)
2. RFP: Ready for Production, is the point at which the release is introduced at one single location as a pilot project (MTTD 15 days)
3. RFM: Ready for Maintenance, during the pilot the last crucial defects are uncovered and fixed. This marks the start of issuing the new more reliable release. (MTTD 30 days)

#### **3.2.3.4 Controlled execution of released software**

During the testing phase, the successfully traversed paths are logged into a database. The core mechanism of macrostep execution and the generated database can remain the part of the released the software. After the release of the product the macrostep engine tries to force the program to stay within the already tested state-space based on the execution tree stored in the database.

The drawbacks of the method:

- The size of released software becomes larger due to the bundled database and the integrated macrostep engine.
- The execution speed can be slower since the *original* macrostep execution causes a kind of extra synchronisation between processes. Moreover, the macrostep controller sometimes does not allow the program to follow an untested execution path, and the alternative input operation must wait for a message from another source.

The advantages of the method:

- The controlled execution of released software is able to increase the reliability of the software product.
- The bug reports and feedbacks to developers can deliver more information since the controlled execution always knows the last global state and the execution path, which results the arised bug.

To increase the speed of controlled execution, application of optimisation technique can improve the efficiency of macrostep execution. It can be achieved, since the users do not want to get a closer look on the inner working mechanism of

---

<sup>11</sup> For instance if 5 defects are found each working month of 10 days then on average the MTTD is two day between each defect.

the program; they are interested only in the correct result, which is generated by the program. Thus, the released version the macrostep controller deals with only the alternative input communication actions, and the acceptance order of messages.

In this case, the necessary functions of macrostep controller can be minimised. The database of execution tree is also reduced because the database must contain only the records of alternative nodes, and the deterministic nodes can be eliminated. Thus, the first disadvantage (size problem) of the controlled execution can be improved.

The second disadvantage (speed problem) is also improved, since the introduced extra synchronisation of macrostep execution is dropped, the execution speed is decreased only by the following two factors.

- (1) Before receiving a message by an alternative input communication action, a query to the database of reduced execution tree is required; whether the source is tested or not, i.e. the receiving of actual message drives the program into untested state or the following state will be a part of the traversed and tested execution tree.
- (2) If the actual source is not tested, the controlled must ignore the message acceptance and the alternative input must wait for another message from another source.

### 3.2.4 Summary, related and future works

As the main achievement, I presented that *the CPN simulation engine is able to steer the macrostep-based traversal of state-space (the building of the Execution Tree) towards erroneous situations during the debugging phase, and to detect the already traversed execution paths.*

In this thesis I followed the active control approach, similarly to other state-of-the-art debugging frameworks [1][3][4][5]. There are existing approaches [6][16][17] to detect erroneous program behaviour based on Petri-net analysis, especially dead-locks, but these techniques developed mainly theoretically with less practical results, and not integrated into a high-level unified framework.

The presented attempt is an extension of a parallel programming environment providing automated and optimised support as much as possible preventing the user from the unnecessary interactions; such as automatic steering based on Petri-net simulation and analysis. However, the presented experimental debugging framework is strongly tied to the GRAPNEL graphical language thus; it cannot be applied directly for real-size legacy parallel application contrary to MAD environment [1]. Other solutions, such as FIDDLE [5], address the flexibility of the debugging environment making an open framework for debugging.

As a future work, the partly elaborated tools and experimental prototypes described in Sections 3.2.2.2-3.2.2.4, and 3.2.3 must be completed, and the presented optimisation methods must be investigated deeply in real-size applications. For example, the hardware acceleration can be a feasible solution concerning the performance issues in case of complex applications; in [40] an FPGA-based accelerator is proposed for reachability analysis of Petri-nets. Since the simple components of Petri-nets can be easily realized in high-density FPGAs, the complete problem can be solved applying an *in-silico* solution that is faster than the traditional software-based simulators.

## 4 Debugging of metacomputing applications

### 4.1 *Metadebugging: new debugging methods for metacomputing applications*

#### 4.1.1 Preface

Emerging high-performance applications require the ability to exploit diverse, geographically distributed resources. These applications use high-speed networks to integrate supercomputers, large databases, archival storage devices, advanced visualization devices, and/or scientific instruments to form *networked virtual supercomputers* or *metacomputers*. While the physical infrastructure to build such systems is becoming widespread, the heterogeneous and dynamic nature of the metacomputing environment poses new challenges for developers of system software, parallel tools, and applications [45].

In this section, an adaptive and extensible graphical debugger is presented for metacomputing applications both from theoretical and practical aspects. The introduced metadebugger has been developed in the context of the Harness metacomputing framework, a dynamic and reconfigurable software infrastructure for distributed computing. A major distinguishing feature of the new metadebugger is its support for dynamic addition and deletion of resources in the distributed virtual computing environment at runtime. The metadebugger also includes several further novel features, including an enhanced *step-into* mechanism, i.e. a kind of automatic context management for remote method invocations (RMI) that unifies the debugging mechanism for local and remote calls in the dynamically changing distributed virtual machine. In addition, an online visualization tool, a navigation tool, a breakpoint manager, and a simple script language interpreter are integrated into the metadebugger as the basis of its further development towards the already introduced macrostep-based reply technique (see Section 2). The metadebugger is also capable of invoking third-party debuggers that might implement additional, useful debugging facilities in heterogeneous computing environments. This facility required new methods in order to “export” and “import” the consistent process states of multi-threaded applications between debuggers.

Therefore, in this thesis, I present *an adaptive debugger framework (metadebugger) for HARNESS metacomputing applications that is able handle the dynamic behaviour of metacomputing systems during the debugging phase*.

#### 4.1.2 Introduction

Metacomputing and grid computing are rapidly gaining in functionality, performance, robustness, and are consequently finding acceptance as standard platforms for HPC [45]. However, debugging of metacomputing applications executing on loosely coupled heterogeneous distributed systems continues to remain a cumbersome and tedious process. The level of complexity in debugging such environments is considerably more complex than traditional sequential or even distributed programs for a number of reasons, including:

- *Heterogeneity* – when multiple architectures and operating systems are involved, standard debugging actions are difficult to generalize
- *Distributed nature* – multiple threads of control within separate address spaces and CPU's must be coordinated.
- *Scalability* – since debuggers intercede between applications and the hardware/OS, multiple instances incur increasing amounts of coordination overhead and therefore do not scale well.
- *Dynamic behaviour and run-time reconfiguration* – in metacomputing, both processes and hardware resources are subject to dynamic joining and leaving, complicating the debugging process.
- *Security* – since debuggers are privy to the inner details of programs, in distributed settings they may pose a security hazard.
- *Non-deterministic execution* – distributed programs can follow different trajectories due to different message and execution orderings.

Although subject to the difficulties listed above, debugging is a crucial phase in the deployment of metacomputing applications and should be streamlined and facilitated to the maximum extent possible. The goal of HARNESS metadepbugger described herein was to attempt to address as many of the above issues as possible, by defining and prototyping an extensible *metadepbugger* for distributed computing environments. The primary architectural innovation is the notion of a meta-level framework for debugging that is adaptive and extensible. Within this meta-level framework, several novel features such as support for dynamically changing virtual machines and remote step-into facilities are implemented. The debugging system is also augmented with a visualization service that permits the graphical depiction of common distributed system events.

### 4.1.3 The Harness Framework and JPDA

#### 4.1.3.1 Harness Metacomputing System

Harness [42] is a metacomputing system that attempts to overcome the limited flexibility of traditional distributed computing software frameworks by defining a simple but powerful architectural model based on the concept of a software backplane. The Harness model consists primarily of a kernel that is configured, according to user or application requirements, by attaching so-called *plug-in* modules that provide various services. Some plug-ins are provided as part of the Harness system, while others might be developed by individual users for special situations, while yet other plug-ins might be obtained from third-party repositories.

By configuring a Harness virtual machine using a suite of plug-ins appropriate to the particular hardware platform being used, the application being executed, and resource/time constraints, users are able to obtain functionality and performance that is well suited to their specific circumstances. Furthermore, since the Harness architecture is modular, plug-ins may be developed incrementally for emerging technologies such as faster networks or switches, new data compression algorithms, visualization methods, or resource allocation schemes — and these

may be incorporated into the Harness system without requiring a major re-engineering effort.

The fundamental abstraction in the Harness metacomputing framework is the *Distributed Virtual Machine* (DVM) (see Figure 37, Level 1). Any DVM is associated with a symbolic name that is unique in the Harness name space but has no physical entities connected to it. *Heterogeneous Computational Resources* may enroll into a DVM (see Figure 37, Level 2) at any time however, at this level the DVM is not ready yet to accept requests from users. To get ready to interact with users and applications the heterogeneous computational resources enrolled in a DVM need to load 'plug-ins' (see Figure 37, Level 3). A plug-in is a software component implementing a specific *service*. By loading plug-ins a DVM can build a consistent service baseline (see Figure 37, Level 4). Users may reconfigure the DVM at any time (see Figure 37, Level 4) both in terms of computational resources enrolled by having them join or leave the DVM and in terms of services available by loading and unloading plug-ins.

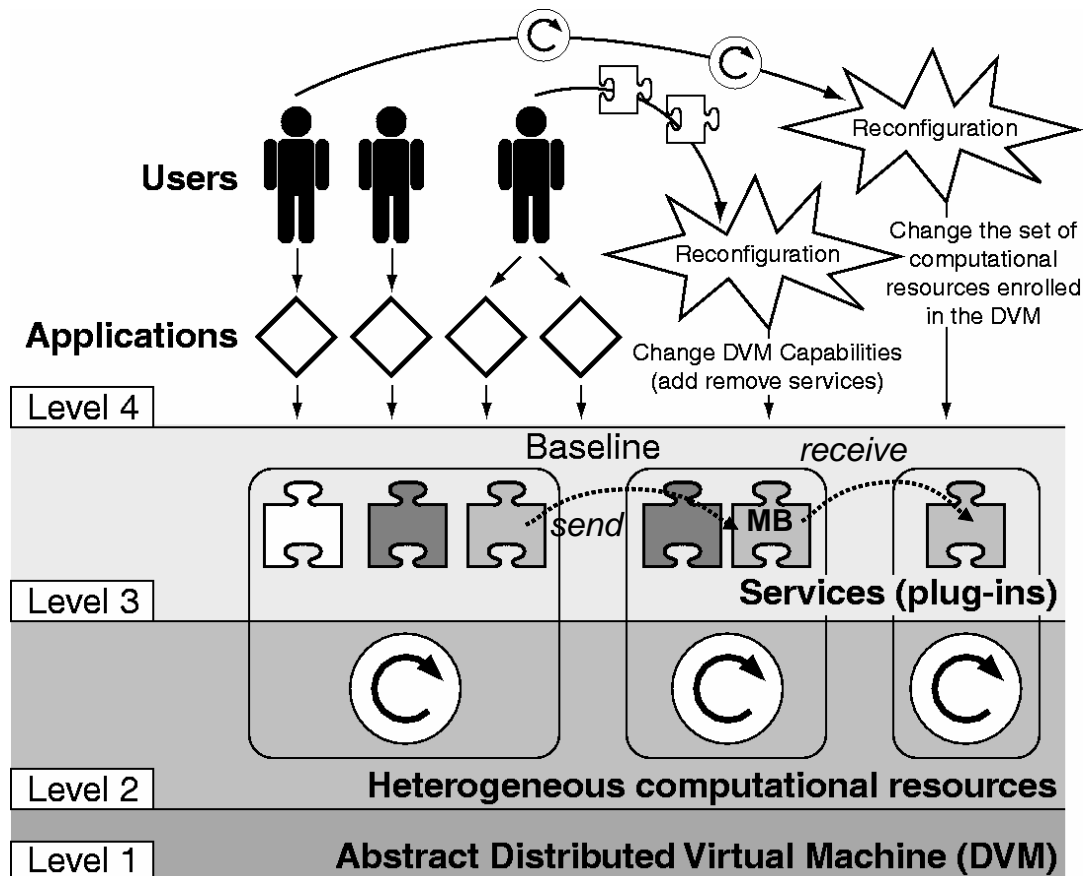


Figure 37 – Abstract Model of a Harness DVM with Message Box (MB) Service

The availability of services to heterogeneous computational resources derives from two different properties of the framework: the portability of plug-ins and the presence of multiple searchable plug-in repositories.

The Harness system implements its model mainly by leveraging two different features of Java technology. These features are the capability to layer a homogeneous architecture such as the Java Virtual Machine (JVM) over a large set of heterogeneous computational resources, and the capability to customize the

mechanism used to load and link new objects and libraries. Harness has evolved into a flexible but robust framework for metacomputing. While Harness can be programmed in *native* mode, using a distributed component model, its reconfigurability also permits plug-ins implementing other programming models to be deployed, thus enabling legacy applications to be executed within the Harness framework. To date, programming environment plug-ins for MPI and PVM have been developed [46], and others are being explored.

#### 4.1.3.2 Java Platform Debug Architecture

As the Harness backplane and a number of core services are implemented in Java, debugging the Harness system and applications that execute within it are closely intertwined with frameworks that support Java. Sun Microsystems, Inc. released the Java Platform Debug Architecture (JPDA) [43], initially with JDK 1.2.2, but JPDA is now widely available as part of Java 1.3. JPDA provides a high-level remote debugging interface from the view of debuggers called the Java Debug Interface (JDI). For the purpose of out-of-process debugging, JPDA opens the Java Virtual Machine Debug Interface (JVMDI) for the debugged/target JVM. Between the JDI and the JVMDI, the Java Debug Wire Protocol (JDWP) is responsible for transporting requests and debug events.

Since the JPDA architecture is modular and based on a separate interface/transport, it is an ideal platform on which remote debuggers can be built. However, although the remote debugging infrastructure is very useful, the JPDA architecture does not offer any support for distributed debugging or the debugging and monitoring of metacomputing applications. Moreover, JPDA does not provide true debugging support for Remote Method Invocations (RMI); rather, it only supports facilities for generating logfiles containing client-IP-address and timestamp information about RMI calls. To use these logs effectively, a debugger-provided identification method to uniquely match RMI calls at the client and server sides is required. Further, the granularity provided by the system is insufficient for accurate visualization of program behaviour. Therefore, I was compelled to develop our own algorithms and techniques to overcome these shortcomings and to be able to construct a robust and effective distributed debugging system.

#### 4.1.4 Metadebugging: Principles and Goals

The HARNESS metadepbugger described in this chapter was undertaken to fulfill the need for debuggers in distributed and metacomputing systems that are based on Java and related technologies. Although developing a true debugger might prove to be somewhat more efficient in operation, such an effort would require a tremendous amount of resources, and further, would have to react closely to changes and updates in Java virtual machines. A more elegant scheme is to develop a “meta” debugging system, viz. a software framework that operates above existing debuggers but provides the scaffolding and additional functionality to enable the coherent and integrated debugging of distributed applications. HARNESS debugger is therefore a metadepbugger in that respect. Prior to describing its internal structure and operational aspects, I briefly list the design goals that guided this project. The three major design objectives that guided our system and implementation choices were:

- (1) permitting adaptability to the debugged metacomputing applications,
- (2) the ability to integrate the system with program visualization, programming and testing tools/techniques, and
- (3) enabling extensibility in order to take advantage of third-party debugger tools in the heterogeneous computational environment.

The first goal implies that the metadepbugger itself must be designed as a real metacomputing application, so that the metadepbugger can be closely adapted to the debugged application and avail of all features of the metacomputing environment, such as support for fault-tolerance and handling of dynamic behaviour and heterogeneity. At the same time, some program visualization techniques with event filtering facilities are required to be integrated in the metadepbugger tool, including features to support navigation through the computational resources in the metacomputing framework and dynamic support for visualizing the history of the debugged application. Furthermore, the debugger should permit users to write their own scripts within the debugger tool for initializing a debug session, for finding different timing errors during the initialization phase or for testing non-deterministic execution. Finally, extensibility is an important requirement. Since the metadepbugger operates over existing debuggers, which, in a heterogeneous environment can be different from each other, it must be general enough to present an interface whose individual commands can be mapped to the underlying debuggers. On the other hand, only supporting the common functions can be restrictive, since special features of individual debuggers that could be very useful might be excluded. To avoid such a situation, extensions to the metadepbugger must be able to invoke third-party debuggers that might implement some architecture dependent debugging facilities on a specified host/pool in the heterogeneous computational environment. The core of the metadepbugger tool can also be based on existing debugger technologies whose capabilities can be extended by the metadepbugger. In this way the user can always choose the best tool for each phase of debugging.

#### **4.1.5 Adaptive Debugging Framework**

The debugging system described herein is an attempt to embody the principles discussed above in a practical toolkit and framework. In this section, some of the crucial design decisions, implementation strategies, and features are described that were incorporated into the prototype version of our tool.

##### **4.1.5.1 Architecture**

A metadepbugger based on the general principles outlined in the previous section needs to address three important aspects:

- The choice of a sequential debugger for monitoring information from a processing element
- The distributed coordination layer to manage the (possibly) multiple distributed components of the application being debugged.
- An appropriate graphical user interface to control and view the debugged system.

The **sequential debugger** that is used as the basis for metaddebugging is the most crucial component of any such system. To use such debuggers without modifications, a software “wrapper” can be used that serves as an interface layer with the remainder of the metaddebugging framework and by redirecting its standard input and output [49]. In our situation, a well-supported debugger architecture with a high-level debugging interface, the JPDA, was also available as an alternative to the option of placing a software wrapper around the *jdb* debugger. Using *jdb* is appealing because it can be used on all platforms, but this strategy would result in increased resource usage because each target JVM requires its own instance of *jdb* for debugging, and the maintenance of the proposed wrapper can be quite difficult due to the existence of multiple and changing *jdb* versions. On the other hand, the JPDA framework is currently not available for the SGI/Irix platform but (unlike *jdb*) it can support multiple attachments to different JVMs, thereby requiring less resource. Based on these considerations, I chose the JPDA framework as the basis for HARNESS metaddebugger, noting that JPDA will likely soon be available for all platforms in the near future.

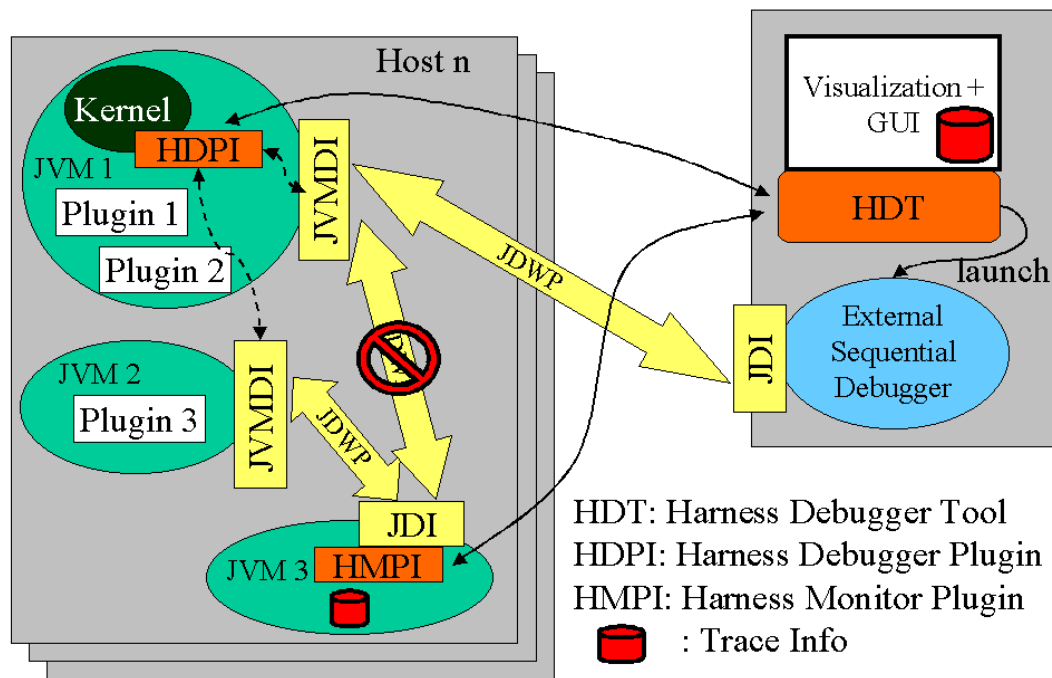


Figure 38 – HARNESS metaddebugger architecture

Two options were considered with regards to the **coordination and monitoring layer**: a Harness-independent layer and a plug-in based approach. With the first option, we would get a more widely usable system but a replicated controlling/communication mechanism would be necessary. With the second option, HARNESS metaddebugger depends on the Harness system and the debugger plug-ins could possibly interfere with the debugged application since they would share some Harness resources. On the other hand, the metaddebugger would be able to leverage the advantages of numerous pre-built Harness services. For the prototype implementation of HARNESS metaddebugger, I decided to adopt the

second approach. Since the metadepbugger is intended for use in debugging application-level components and not Harness system plug-ins, the possibility of interference is reduced if not eliminated. From a pragmatic point of view, the choice of a Harness-based implementation provides a real metacomputing software system in which the principles and concepts underlying the metadepbugger can be tested. In particular, the notion of using distributed components to implement metadepbuggers would be thoroughly tested. Finally, the Harness system itself would become more usable owing to the existence of a debugging framework.

#### 4.1.5.2 Start-up

The newly introduced components of the HARNESS metadepbugger architecture (see Figure 38) and their main functions are as follows.

- *Harness Debugger Plug-In* (HDPI) is responsible for some special metadepbugging-related functions, such as controlling JAVA threads during the invocation of third-party debuggers, gathering status information during start-up or clock-synchronization.
- The distributed monitoring and tracing functions are placed in the *Harness Monitor Plug-In* (HMPI). The monitored events can be either traditional debugger events (e.g. breakpoint hit) or special events for visualization (e.g. RMI method entry event).
- *Harness Debugger Tool* (HDT) dispatches the user's requests to the appropriate monitor/debugger plug-ins and also collects information for the GUI

To debug applications with HARNESS metadepbugger, the user launches the Harness kernels (see Figure 38) with a special debug flag to enable the JVMDI interfaces and turn the JIT compiler off. As Figure 38 shows, on each host in the distributed virtual machine two different types of plug-ins can be found for debugging purposes: one instance of HDPI - a "threadable" plug-in (i.e. loaded in the same JVM as the kernel), and one instance of HMPI - a "non-threadable" plug-in (i.e. loaded in a separate JVM). Both of these plug-ins are loaded by HDT (using the same authorization keys as the user plug-ins) during the initialization phase, but they are assigned different tasks as described later. In the first step of initialization, the HDPI plug-in gathers information about the enabled JVMDI interfaces and passes the information to the HDT. Thus, in the second step the HDT is able to load the HMPI plug-in and to attach the loaded HMPIs to the kernel JVMs (using the facilities provided by JDWP).

#### 4.1.5.3 Handling of Dynamic Behaviour

Unlike some existing distributed debuggers, the HARNESS metadepbugger, in the context of Harness, must adapt to the dynamic behaviour of debugged applications. In other words, when a new plug-in is loaded by the user's application anywhere in the metacomputer, HARNESS metadepbugger must load and activate the HDPI and HMPI plug-ins on the target host for debugging/monitoring purposes (using the same authorization keys as the loaded plug-in). To handle the dynamic behaviour, re-configurability, and plug-in substitution facilities inherent in Harness distributed virtual machines (DVMs), special breakpoints are used. During the initialization of HMPI, the monitor plug-in places so-called "system breakpoints" in the Harness kernel to detect changes in the DVM in advance. System

breakpoints are placed within different methods of the Harness kernel that relate to plug-in loading/unloading and host addition/removal.

In this way, we can avoid the need to build a new and unnecessary communication structure between the Harness kernels and HDT. When a system breakpoint is hit, the HMPI plug-in evaluates the appropriate fields of the kernel objects depending on the type of system breakpoint and also refreshes both the inner database of HMPI and the GUI. When loading a non-threadable plug-in, HMPI has to suspend the progress of plug-in loading between the JVM spawning and the real plug-in loading. During this period the HMPI is able to place system breakpoints and to request notification of method entry and exit events (described later) in the new JVM.

#### 4.1.5.4 “Step-into” for Remote Method Invocations

Several Harness plug-ins are implemented as pure JAVA, object oriented and RMI-communication-based classes, and thus exploit the easy-to-use RMI concept while retaining the advantages of modular and component oriented development. Harness also provides strong support for RMI-based plug-ins with a specialized name service. To provide efficient debugging support for RMI-based plug-ins, HARNESS metad debugger offers some unique debugging capabilities for RMI communication. One of these is the newly introduced *step-into* facility for RMI during step-by-step execution. Essentially, this feature permits the debugger to intervene at two points — once when the RMI client invokes a method of the RMI server object, and again when the remote method returns to the calling client. The metad debugger GUI automatically switches contexts at these two points. Upon RMI invocation, when the GUI switches to the new context (Host, JVM, Plug-in, Thread), the user is able to debug the remote method by evaluating its fields, by executing it step-by-step, and examining variable values. An example of this can be seen in Figure 39 when a user plug-in (identified by //tuborg/myDVM.helloHarness.HmasterImpl.3) has invoked the *ping* remote method of another plug-in (named //cssun/myDVM.helloHarness.HslaveImpl.2).

This newly contributed step-into feature for RMI invocations allows the user to debug an RMI call in the same manner as a traditional method invocation. Handling the step-into for RMI call works as follows HARNESS metad debugger listens for method entry events of RMI stubs belonging to Harness plug-ins. If any method entry event occurs during a user requested step-into, HARNESS metad debugger obtains the following information:

- (i) IP address of the server host,
- (ii) the RMI ID for the remote object on the current host,
- (iii) the unique ID for RMI call.

This information is obtained by analyzing certain private fields belonging to the sun.rmi.transport.LiveRef and the java.rmi.server objects. The client-side monitor sends this information to the HDT and suspends the current thread until the server-side has received the actual remote object ID and unique ID of RMI call from HDT. Then the client-side monitor issues an extra step out request ensuring that the execution will stop on the appropriate line in the user's source code when the RMI call has completed. Meanwhile, when the server-side monitor detects the current

(server-side) RMI method entry event with the already provided IDs, the monitor suspends the current thread and sends the location information to the GUI in the same way as would happen on a breakpoint or a step event. Based on the obtained location information, the GUI switches to the new context (Host, JVM, Plugin, Thread) and at this stage, the user is able to debug the remote method by evaluating its fields, executing it step-by-step, inspecting values and so on.

One aspect remains: return to the client side. On the GUI, the user may select either the *resume thread* or the *step out* functions. In the first case, the RMI method on the server side finishes its work and the previously issued client-side step-out request will stop the client just after the current RMI call. In the second case, execution will be stopped in the server-side skeleton instead of returning to the client side. When the server-side monitor detects a method exit event during a step-out request within an RMI call, the monitor resumes the current thread automatically and everything continues as in the first case. The only drawback of this algorithm is the usage of a `sun.*` class and some private fields of RMI related classes that might be changed in the future by SUN. As a result, small portions of the HARNCESS metad debugger may require maintenance when a new JAVA SDK version is released.

It should be noted that while any thread blocked on synchronized remote method can be found with the help of the integrated navigation tool (see Figure 39), the current version of HARNCESS metad debugger is not able to detect deadlocks caused by synchronization errors.

#### **4.1.5.5 Navigation and Visualization**

One of the most difficult tasks during debugging a Harness application (or an application in any other distributed or metacomputing system) involves navigation across multiple levels of abstraction and the large numbers of entities at each level. The typically large size and complexity of such applications also adds an additional level of difficulty to the debugging process. From the point of view of debugging at least five different layers can be distinguished in Harness, including:

1. Distributed Virtual Machine (DVM),
2. each Host in the DVM,
3. Java Virtual Machine (JVM) on each Host,
4. Harness Plug-in within a JVM, and
5. Threads in Harness Plug-ins.

Obviously, a DVM may contain several hosts or plug-ins. Without efficient debugging support and a graphical user interface equipped with filtering facilities, it can be a very exhausting task to switch between the different instances of threads and plug-ins. In our system a tree structure is used to conveniently represent the different elements of different abstraction layers of the DVM as Figure 39 (Navigation Tool) depicts. To avoid the use of too many levels in the tree, the JVM level was merged into the plug-in level in the graphical representation. This is possible because each “non-threadable” plug-in — loaded in a separate JVM — has its own JVM (having only one leaf from a branch of tree can be redundant) and it is sufficient to represent such plug-ins by another icon. Additionally, it is often

not sufficient to only know the current state of a debugged application. Frequently the programmer would like to get an overview what has happened during the program execution. Reading of thousands messages in different log files generated in *printf*-style debugging can be very tedious and unproductive. The solution is an integrated visualization tool (see Figure 39), a concept that has already been implemented in several systems, but there has not yet been an implementation for JAVA/RMI-based metacomputing environments.

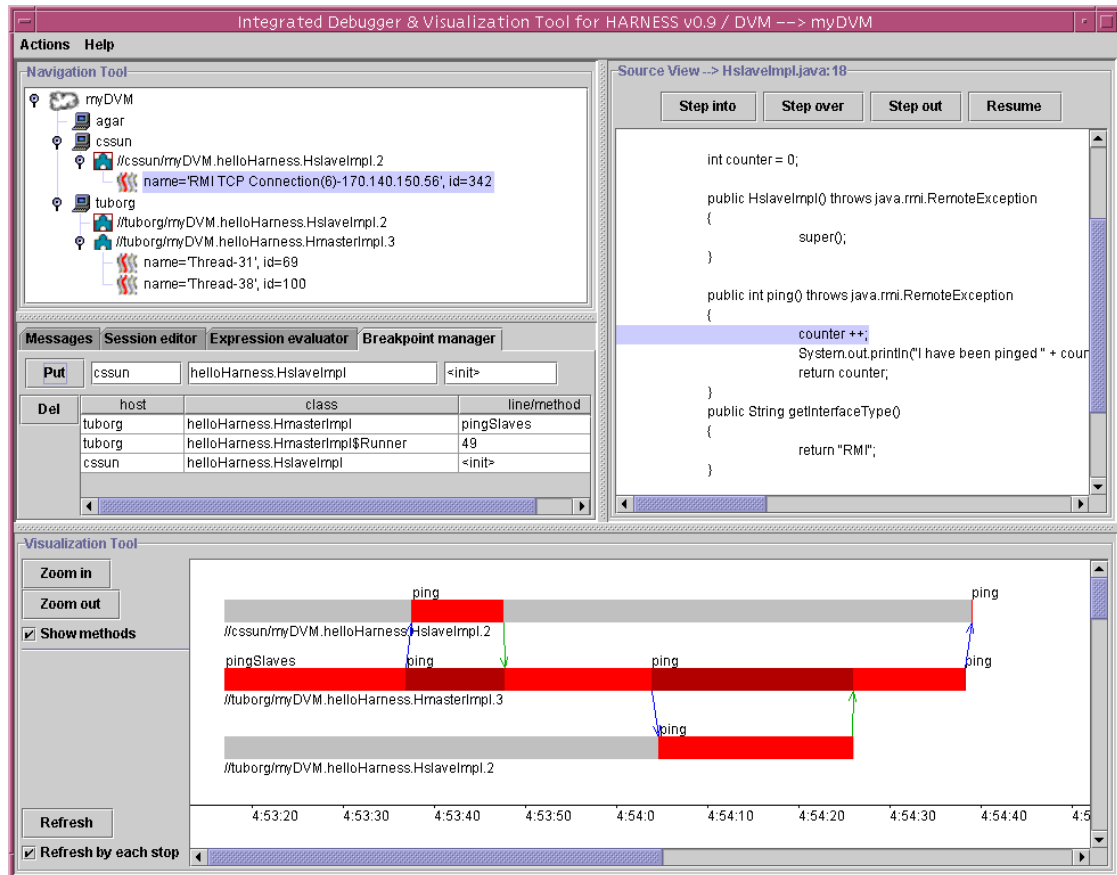


Figure 39 – Tools of HARNESS metadbugger

Visualization facilities in HARNESS metadbugger are based on the monitoring of method entry/exit events provided by JPDA but enabling these features necessitated the solution of several issues. For example, debug events issued by JPDA do not contain timestamps. Therefore, the HARNESS metadbugger must be responsible for generating timestamps for events. For this purpose, the monitoring tool has to be as close to the debugged JVM as possible to avoid measurement errors that may be caused by unpredictable network traffic. For this reason, there are local/remote monitors (see HMPI in Figure 38 on each host in the HARNESS metadbugger architecture). Further, JPDA does not offer debugging support for RMI as mentioned earlier. Therefore, we had to implement our own identification procedure to determine the appropriate client-server method pairs of RMI calls. The following events are monitored separately for each JVM during execution:

- (i) Plug-in load/unload,
- (ii) RMI client-side call/return, and
- (iii) RMI server-side method entry/exit.

A trace event indicating client-side RMI call must contain the IP address of the server host, the hash code of the remote object, and a unique id for the remote call, in order to be able to determine appropriate client-server pair matching as described in Section 3.3. The monitoring tool also traces the generated timestamp and the handler of current plug-in (as shown below the plug-in bars in the Visualization tool, see Figure 39).

Matching executing threads to their corresponding plug-ins is not a simple task. During an RMI call, the main thread can be identified easily by its corresponding remote object, but the generated threads do not contain information about the related remote object/plug-in. One possible solution is to monitor thread creations and to record the handlers of related plug-ins, but this scheme can result in high overheads for the monitoring subsystem. In the current prototype implementation of HARNESS metadepbugger we overcame this issue by introducing a new thread type called a *H\_Thread*. The usage of *H\_Thread* is the same as traditional Java threads. Only the creation syntax is different; as an extra parameter the user has to provide the hashcode of the current (parent) object. This solution is also useful for thread handling when a breakpoint is hit: we were able to change the default *suspend all threads* policy with the more acceptable *suspend all H\_Threads* policy. Thus, threads belonging to the Harness kernel (within the same JVM as threadable Harness plug-ins) can be distinguished easily from user threads and can be suspended separately, and the kernel is able to continue system-level tasks, such as answering heartbeat signals sent by the Harness server.

Our time synchronization for visualization is based on a simple *ping-pong* algorithm assuming the equality of average network latencies in both directions between two hosts. Based on these architectural solutions and algorithms, the central part of the debugger (HDT in Figure 38) can collect trace files generated by HMPs at each step event/breakpoint hit, or when the space-time diagram is explicitly refreshed by the user. This implies a “semi-realtime” visualization behavior; updates occur periodically but there is minimal perturbation to the debugged application. Using the above scheme, the visualization tool draws the space-time diagram (see Figure 39) where each plug-in belongs to a red horizontal bar. RMI calls are represented by blue arrows, and green arrows show the returns. If more than one RMI call is active in a Harness plug-in at the same time, the visualization tool uses increasingly dark shades of coloring. There are options to show the method names and threads, to resize/scroll the current diagram or to filter events/plug-ins. The filtering facilities are very useful when a large number of resources are enrolled in the DVM.

#### **4.1.5.6 Invocation of third-party debuggers**

It is a well-accepted fact that most application developers prefer to use their favorite and/or widespread tools whenever possible. However, frequently, architecture dependent debugging tools may be installed in heterogeneous computational environments. To take advantage of such situations, our debugger is designed to be able to invoke an existing external debugger tool and attaching to a target JVM. Obviously, in this instance the user will not be able to take full advantage of HARNESS metadepbugger features (e.g. *step-into* for RMI), but there is an option to re-attach the HARNESS metadepbugger to any JVM if the user wishes to do so at another stage in the debugging process. A software engineering

advantage of being able to switch between debuggers is that the built-in sequential debugger does not need to provide the all functionalities of architecture-specific, special and sophisticated sequential debuggers. The latter may be used when sequential portions of plug-ins need to be debugged very carefully and deeply, while the JPDA framework may be used in conjunction with HARNESS metadepbugger for debugging aspects related to the metacomputing portion of the application.

Another issue concerns a strict limitation in JPDA: only one connection is allowed to a target JVM at a time (see Figure 38). Moreover, JPDA releases the all suspended threads when a debugger is detached from the target JVM. In our solution this thread-controlling problem is handled by the help of HDPI plug-ins that are also responsible for suspending the appropriate threads during switching to and back. When an external debugger is needed for the further inspection of the sequential parts of Harness plug-ins, I propose the following solution. Before switching to an external debugger, HDPI looks up the all threads suspended by the user and suspends them again in the traditional way (i.e. the *Thread.suspend* method) and not via JPDA. Following the switch, HDPI lists these specially handled threads on the display, and the user is responsible for suspending them with the help of the external sequential debugger. Finally, the user has to resume (using the *Thread.resume* method) this set of threads by the HDPI. Through this mechanism, the originally suspended threads are exported under the newly launched sequential debugger. Switching back requires another sequence. First, the user suspends the requested threads through HDPI (in the traditional way). As the next step the external debugger can be detached from the target JVM. At the end of switching back, the user can re-attach the HARNESS metadepbugger that automatically suspends the current set of threads using JPDA and also releases them in the traditional way through HDPI.

#### **4.1.5.7 Programming of debugger**

It can be a very tedious and exhausting process for users to load the same plug-ins, set the same breakpoints, and follow the same routine when for each new debug session in a typical development cycle. To overcome this drawback, as the very first step of debugging an entire Harness application, HARNESS metadepbugger allows the user to load, save and edit scripts for initializing a debug session. Scripts may contain debug commands, such as GRAB\_HOST, LOAD\_PLUGIN, EVAL and so on. In addition, control structures such as DELAY, WHILE *<expr>*, IF *<expr>*, and RUN\_TO\_NEXT\_RMI have been introduced for convenience and control purposes. The DELAY command combined with different ordering sequences of LOAD\_PLUGIN commands can be used for testing the startup of Harness applications. Using IF, WHILE, and RUN\_TO\_NEXT\_RMI commands, users can force their applications to run with varying timing conditions. This feature is expected to be very valuable but the next thesis deals with these problems in details (see Section 4.2).

#### 4.1.5.8 Breakpoint sets

In addition to supporting traditional breakpoints, the HARNESSE metadepbugger extends breakpoint description with two extra parameters, namely the target host and target plug-in identifier. This feature is useful in debugging only selected instances of plug-ins on selected hosts. Both of these can be undefined (denoted by “\*” or an empty field); if undefined, the breakpoint will be set on each enrolled host and in each instance of the specified plug-in. The Breakpoint Manager (see Figure 39) is used for editing breakpoint sets: adding a new breakpoint or deleting an existing breakpoint, and loading/saving the entire breakpoint set. The breakpoint manager can also provide status information about breakpoints, such as: deferred, resolved or invalid.

#### 4.1.6 Related Work

The HARNESSE metadepbugger described in this thesis is oriented specifically at metacomputing environments. Our experimental implementation of HARNESSE metadepbugger contributes several novel facilities but also leverages certain characteristics of the Java language and system; therefore, in this section, we compare and distinguish our work in the context of Java debuggers. Several debugging systems exist for Java-based environments; we briefly list a representative cross section and highlight the main features of each. *Jdb* permits remote debugging (based on JPDA) and *Javadt* extends *Jdb* with a simple graphical user interface. While these tools permit non-local programs to be debugged, they do not provide coordinated facilities for distributed debugging, visualizing program execution, and convenient debugging of remote method invocations. *DejaVu* for Jalapeno JVM [36] is a debugging tool that can deterministically replay the non-deterministic execution of multithreaded Java applications. This tool also provides a GUI for visualizing program execution with nearly all the functionality of traditional sequential debuggers, but it cannot be used for distributed/metacomputing applications and runs only on IBM AIX. DJVM [37] is a replay tool for distributed socket-based Java applications that could be a good basis for a distributed debugger or metadepbugger. However, to the best of our knowledge, there is no available debugger tool using DJVM. An upcoming version of the TotalView [38] debugger will be extended by graphical facilities that include a Message Queue Graph (a snapshot of pending MPI messages) and a call tree diagram within a process. The independent TimeScan Multiprocess Event Analyzer can be used for performance analysis by visualizing events. However, while TotalView and associated frameworks are very popular in parallel computing environments, these tools from Etnus support neither the JAVA language/RMI concept nor the invocation of external tools<sup>12</sup>. Similarly, P2D2 [39] from NASA is a portable debugger with a simple visualization tool and is appropriate for debugging Globus/MPICH application but P2D2 has no JAVA/RMI support and it does not support invoking an external sequential debugger.

---

<sup>12</sup> According to Totalview user’s documentation (2001). Meanwhile, the debugger tool has been extended towards this direction.

### 4.1.7 Summary

Motivated by the dearth of effective debugging systems for metacomputing, I presented *an adaptive debugger framework (metadebugger) for HARNESS metacomputing applications that is able handle the dynamic behaviour of metacomputing systems during the debugging phase.*

The current prototype of HARNESS metadebugger has been tested on a Linux cluster with 60 Pentium III processors and it is currently undergoing testing on a SUN HPC 10000 equipped with 96 processors. Our experiences have been positive and have shown that the metadebugger approach is viable for the debugging and visualization of Java-based systems and applications. Although the prototype was developed for Harness, the solutions introduced in this chapter can be generalized to other metacomputing systems, by replacing the Harness plug-in mechanisms and name service schemes with framework-dependent mechanisms. The metadebugging approach presented herein provides flexibility and extensibility, while requiring only minimal adaptation and support from the underlying distributed or metacomputing framework.

It should be noted that while the JPDA architecture simplified certain aspects of HARNESS metadebugger (such as remote debugging and heterogeneity), more than 80% of the metadebugger design and implementation addresses non-JPDA related theoretical, architectural, and technological issues such as dynamic distributed virtual machines, 3<sup>rd</sup> party debuggers, step-into for RMI, distributed monitoring, online visualization, special thread handling or testing.

## 4.2 *Enhanced macrostep-based debugging towards metacomputing applications*

### 4.2.1 Preface

This thesis focuses on the non-deterministic behaviour and architecture dependencies of metacomputing applications from debugging point of view. For this purpose, I present that the *macrostep-based debugging methodology* [48] *can be generalised towards HARNESS metacomputing applications* [45].

The introduced methodology is based on modified collective breakpoints and macrosteps furthermore, I introduced host-translation tables generated automatically for exhaustive testing. The experimental prototype is developed under the Harness metacomputing framework for message-box communication based applications.

The main principal issues as well as the generic architecture of the macrostep-based metadebugger are also described as the further development of HARNESS metadebugger [54].

### 4.2.2 Introduction

Debugging of metacomputing applications can be much more exhausting task contrary to debugging of sequential or even parallel programs. This problem comes from the following features of metacomputing [45]:

- (i) heterogeneity,
- (ii) dynamic behaviour of computational environment,
- (iii) large amount of computational resources
- (iv) authorisation/authentication on different administration domains, and
- (v) non-deterministic execution of metacomputing applications.

In the previous thesis (see Section 4.1) I have already given solutions for (i)-(iv) in the Harness framework but the *systematic handling of non-determinism* was out of scope of that work. In this thesis, I focused on the issues of the non-deterministic behaviour of metacomputing applications caused by the varying relative execution speeds of tasks as well as the architecture dependent failures. For instance, it seems a given metacomputing application always generates correct results on a particular architecture or a combination of architectures (where the programmers originally developed their application) *but* often fails on other architectures. Mostly, the reason for this behaviour is the varying relative speeds of tasks together with the hazardous and untested race conditions. Besides, these different timing conditions might be occurred more frequently in metacomputing environment than in case of dedicated clusters or traditional supercomputers because of the different implementation of the underlying operating systems/communications layers and the unpredictable network traffic, CPU loads or other dynamical changes. By metacomputing applications the above described phenomenon can be very crucial because we cannot ensure that our metacomputing application always runs on the same nodes with almost the same timing conditions.

The only way to prove the ‘metacomputing-enabled’ feature of an application is the usage of *systematic* testing methods in order to find the timing/architecture dependent failures in the implemented code. For this purpose I applied and also extended the macrostep systematic debugging methodology that has been introduced originally for message passing parallel programs developed by P-GRADE graphical programming environment [50]. The experimental prototype is under development in the Harness metacomputing framework [42] and the work is relying on the achievements of the earlier developed HARNNESS metad debugger [54].

The Harness framework, the Java Platform Debugger Architecture (JPDA) and the HARNNESS metad debugger as the basis of our prototype have been already described in details in Section 4.1.3.

### 4.2.3 Metad debugger for HARNNESS

In order to solve the emerging debugging issues in the field of metacomputing we already defined the fundamental principles of an extendible, programmable and integrated debugging & visualization tool [54]. The next target was to design and implement a prototype; HARNNESS metad debugger applying the defined principles and relying on the Harness framework as well as the above described Java Platform Debugger Architecture.

In order to illustrate briefly the novelty of this work, the main features of the current HARNNESS metad debugger prototype can be summarized as follows; HARNNESS metad debugger was designed as a *real metacomputing application* itself

hence, the debugger tool can adapt totally to the debugged application and also can take all advantages of the metacomputing environment, such as fault-tolerance, dynamic behaviour, support for heterogeneous computational environment and authorization. When a plug-in is loaded by the user's application anywhere in the metacomputer, HARNESS metadepbugger can load and activate some system plug-ins on the target host for debugging/monitoring purposes (using the same authorization keys as the loaded plug-in). Moreover, for providing efficient debugging support for RMI-based plug-ins, HARNESS metadepbugger offers some unique debugging capabilities for RMI communication. Firstly, during step-by-step execution HARNESS metadepbugger is able to hide the differences between the traditional and remote method invocations from user's point of view. Basically, it means two automatic context switches during an RMI call (client to server/server to client side). On the other hand, HARNESS metadepbugger combines some program visualization techniques with debugging methods. Hence, the user can get a big picture about the history of plug-ins with the help of an integrated semi-online visualization tool depicted the communication interacts among Harness plug-ins.

Another significant feature of the system is the *extendibility*. HARNESS metadepbugger can invoke external sequential debuggers that might implement some other architecture dependent debugging facilities on a specified host/pool in the heterogeneous environment. In this way, the user can choose the best tool in every phases of debugging procedure. Additional tightly integrated graphical tools are responsible for the navigation through the distributed/Java virtual machines and threads (equipped by filtering options for handling of scalability), management of breakpoint sets and establishment of new debug sessions.

Finally, HARNESS metadepbugger is *programmable* with a simple macro language particularly for testing purposes. Thus, the programmer can test the start-up of his application, and can force the metacomputing application to run with vary timing conditions.

#### **4.2.4 Macrostep-based debugging in HARNESS**

As it was described above, HARNESS metadepbugger was designed originally for Harness applications built on RMI-based plug-ins. During an RMI-based interaction the invoked remote methods are executed in separated threads on the server side but the macrostep debugging methodology [48] cannot be applied in case of multithreaded applications (which might use shared objects). Thus, we had to take into consideration two options:

- (i) attempt to extend the macrostep debugging methodology with multithreaded/shared objects support or
- (ii) provide systematic debugging support for other types of Harness plug-ins, e.g. which are based on message passing paradigm.

As the first stage of this work, the macrostep debugging methodology has been applied on Harness applications, which can communicate with each other via message box. Based on these experiences and achievements I will try to solve the systematic debugging issues of multithreaded/RMI-based metacomputing applications as the next stage of this work.

In Harness the message box plug-in provides a generic send/receive/scatter/gather message passing service for Harness plug-ins via a simple interface:

- public void *send*(String senderID, String destination, Object message)
- public void *sendToAny*(String senderID, Object message)
- public H\_Envelope *receive*(String myID, String senderID)
- public H\_Envelope *receiveFromAny*(String myID)
- public H\_Envelope *receiveAsync*(String myID, String senderID)
- public H\_Envelope *receiveFromAnyAsync*(String myID)

In details, the *send* and *sendToAny* operations are always executed asynchronously but each type of *receive* operation can be either asynchronous or synchronous. As a first step, I reduced these communication possibilities in order to get a similar message passing interface as in P-GRADE system where the macrostep debugging methodology has been successfully implemented. Thus, the asynchronous send operations turned to synchronous send and also removed both asynchronous receive operations in the new message box\* plug-in.

The proposed debugging cycle of metacomputing applications (see Figure 40) is similar to the debugging cycle of GRAPNEL programs but here the HARNESS metadepbugger is situated in the middle as the base of debugging infrastructure instead of DIWIDE parallel debugger. On one hand, the macrostep engine helps handle the non-deterministic behaviour of application, on the other hand, 3<sup>rd</sup> party debuggers can be also invoked (see Section 4.1.5.6) if necessary.

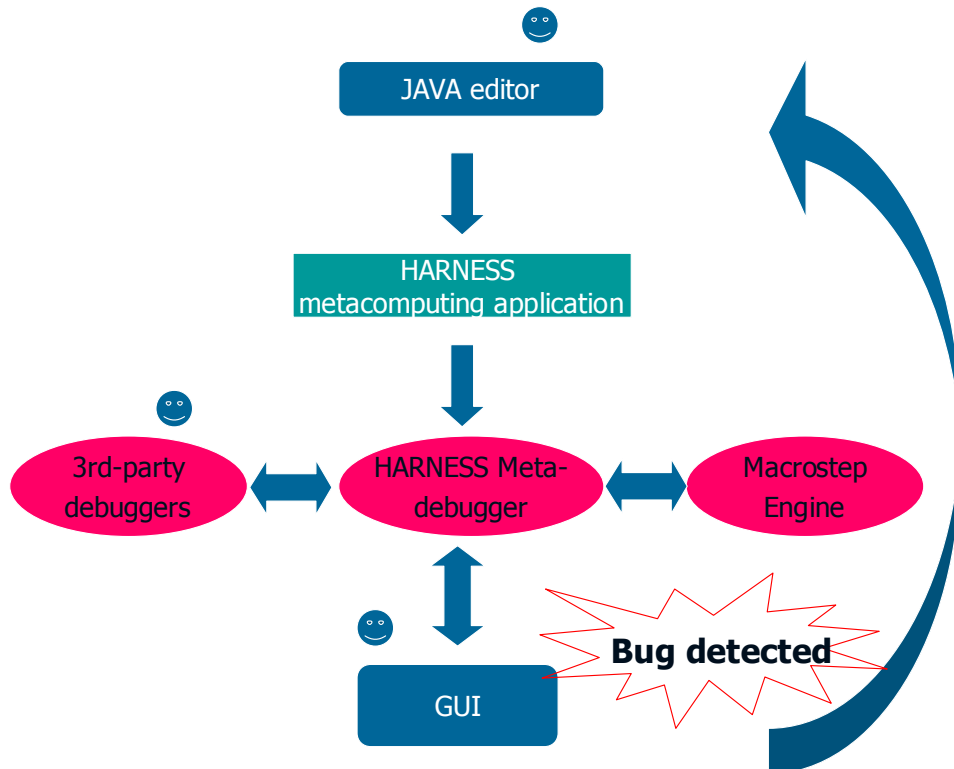


Figure 40 – Debugging cycle for metacomputing applications

The main ideas of the further developed macrostep debugging methodology can be summarized by the following concepts:

- (i) enhanced collective breakpoints,
- (ii) modified macrosteps,
- (iii) extended macrostep-by-macrostep execution mode,
- (iv) execution tree,
- (v) meta-breakpoints,
- (vi) host-translation table.

In the rest of this section, these concepts are described as well as some implementation issues.

In [48], a restriction was introduced on the global breakpoint sets and introduced a special version of them called **collective breakpoints**. When all the breakpoints of the global breakpoint set are placed on communication instructions, the global breakpoint set is called collective breakpoint. If there is at least one breakpoint for each alternative execution path of every process, the collective breakpoint is called strongly complete. In practice, we were able to implement the strongly complete collective breakpoints by placing breakpoints on each method entries of message box interface. It means only a couple of permanent breakpoints for each message box thus; we might achieve good performance that can be crucial in case of communication intensive metacomputing programs.

Two problems have risen during the design phase:

- (i) RMI communication between plug-ins and the message box,
- (ii) dynamically created message boxes.

In details; the message box service was implemented as a plug-in, according to the Harness concept, and the sender/receiver plug-ins have to communicate with the message box plug-in *via RMI*. As it is described in [54] JPDA does not provide debugging support for RMI but the system has to find out which plug-in wants to send or receive a message (the *myID* and *senderID* string arguments can be defined without any restrictions by plug-ins). Hereby, I had to deal with issues of RMI debugging and to apply some RMI-related functions of HARNESS metadbugger in spite of our original plans. On the other hand, any Harness plug-in can create dynamically *new message boxes* therefore; our debugger tool must be also responsible for detecting when a new message box plug-in is loaded.

The set of executed code regions between two consecutive collective breakpoints is called a **macrostep**. Precise definition of macrostep is given in Section 2.2. If sequential program parts between communication instructions are already tested, a systematic debugging of a metacomputing program requires to debug the metacomputing program by pure macrosteps, i.e. instrumenting all the communication instructions by global breakpoints. A breakpoint of the collective breakpoint is called active if it was hit in a macrostep and its associated instruction can be completed. A breakpoint is called sleeping if it was hit in a macrostep but its associated instruction cannot be completed (for example, receive instruction

waiting for a message). Those breakpoints that were either active or sleeping in a macrostep are together called effective breakpoints as it was described in Section 2.2.2.

After the definitions given above we can define the **macrostep-by-macrostep execution mode** of metacomputing programs. In each step either the user or the debugger runs the program until the collective breakpoint is hit. Under these conditions the metacomputing program will be executed by macrostep-by-macrostep. The boundaries of the macrosteps are defined by a series of effective global breakpoint sets. In such cases, the user is interested only in checking the program state at the well-defined boundary conditions.

There is a clear analogy between the step-by-step execution mode of sequential programs realised by local breakpoints and the macrostep-by-macrostep execution mode of metacomputing programs. The macrostep-by-macrostep execution mode enables checking the progress of the metacomputing program at the points that are relevant from the point of view of parallel and distributed execution, i.e. at the message passing points. What we should ensure is that the macrostep-by-macrostep execution mode should work deterministically just as the step-by-step execution mode works in case of sequential programs. In order to ensure it, according to the original macrostep concept the debugger should store the history of collective breakpoints, the acceptance order of messages at receive instructions and the result of input operations. Additionally, in a metacomputing environment we should also store the events about the reconfiguration; when a new plug-in is loaded, unloaded or failed anywhere in heterogeneous computational environment, new host is grabbed/released or a new message box is started by the user's application. Therefore, the debugger tool must be able to adapt to the dynamic behaviour of debugged application as well as its fault tolerance. As it was mentioned in Section 4.1.3.1, the enrolled computational resources as well as the DVM itself can be reconfigured. To handle the dynamic, reconfigurable and fault tolerant behaviour of DVM, the basic concept was the following. During the initialisation the Harness Monitor/Control Plug-In (HMCPI) places some so-called 'system breakpoints' in the Harness kernel (see Figure 41) in order to detect the changes/reconfiguration of DVM in advance. Then, HMCPI can report these events to Harness Systematic Debugger Tool (HSDT) that is responsible for storing these reconfiguration events in a trace file (see Figure 41). Basically, the fault tolerance of HARNESSES metad debugger has been inherited from the Harness Framework itself.

At replay, the progress of tasks are controlled by the stored collective breakpoints and reconfiguration events and the program is automatically executed again macrostep-by-macrostep as in the execution phase. The debugger is also responsible for loading/unloading/killing the plug-ins, grabbing/releasing hosts and starting new message boxes during each macrostep (if it is needed). Obviously, during the replay phase it is not guaranteed that a host can be grabbed again for the distributed virtual machine or a given host is able to load the required plug-in (resource limitations, etc.). The solution is a new *host-translation table* maintained by the debugger, in that each host enrolled in the original DVM can be associated to a substitute host (independently for each plug-in) where the appropriate plug-in actually run during the replay phase. The relative speed of the substitute host is unessential because the macrostep-by-macrostep execution can handle this phenomenon. Only the architecture of the substitute host can be important if the

current plug-in uses some architecture dependent features (e.g. via Java Native Interface). In this case, the system must check whether both architectures of the reference and the substitute hosts are the same ones.

In Harness the introduced host-translation table is used by the systematic debugger tool as well as the RMI communication core during the replay/control phases.

The execution path is a graph whose nodes represent macrosteps and the directed arcs connect the consecutive macrosteps. The **execution tree** is a generalization of the execution path; it contains all the possible execution paths of a metacomputing program assuming that the non-determinism of the current program is inherited from (wildcard) message passing communications. Nodes of the execution tree can be of four types: (i) Root node, (ii) Alternative nodes, (iii) Deterministic nodes.

The Root node represents the starting conditions of the metacomputing program. Alternative nodes indicate either a wildcard receive instructions which can choose a message non-deterministically from several sources or –as an extension of the original macrostep concept– wildcard send instructions which can send a message non-deterministically to any target. Only alternative nodes can create new execution paths in the execution tree, deterministic nodes cannot create any new execution path.

Breakpoints can be placed at the nodes of the execution tree. Such breakpoints are called **meta-breakpoints**. The role of meta-breakpoints is analogous with the role of the breakpoints of sequential programs. A breakpoint in a sequential program means to run the program until the breakpoint is hit. Similarly, a meta-breakpoint at a node of the execution tree means to place the collective breakpoint belonging to that node and run the metacomputing application until the collective breakpoint is hit. Replay guarantees that the collective breakpoint will be hit and the metacomputing program will be stopped at the requested node.

The task of systematic debugging or testing is to traverse exhaustively the execution tree with all the possible execution paths in it. Therefore, the execution tree represents a search space that should be explored completely by the debugging method. Accordingly, systematic testing and debugging of a metacomputing program require (i) generation of its execution tree (ii) exhaustive traverse of its execution tree. With the help of the extended macrostep-by-macrostep concept both of these issues can be solved and implemented in a very similar way as they have been implemented in DIWIDE [49]. Some minor changes are required by the wildcard send operations as well as the event tracing and replaying.

#### 4.2.5 Architecture dependencies

Often a Harness plug-in does not require a particular architecture for its execution. Despite of this, the system always has to inspect whether each plug-in has been implemented *architecture independently* if the developer wants to get an error-prone metacomputing application. For testing the architecture independency of plug-ins or whole applications *systematically* generated host-translation tables are needed. It means that each architecture independent plug-in must be tested on each significantly different architecture (by exhaustive traverse of the execution tree).

The debugger tool can test several plug-ins (from the aspect of architecture dependency) in one exhaustive traverse of an execution tree. In the best case, we need only as much traversing of the execution tree as the number of significantly different architecture we have in the metacomputing environment. Our solution contains four steps:

(1) the debugger looks for a host with a new and untested architecture for each architecture independent and not fully tested plug-in and the debugger also registers the found host into the new host-translation table,

(2) if step 1 was not successful by any plug-in (after a timeout), we allowed the debugger to look for *any* host for the unsuccessfully mapped plug-ins

(3) if there was at least one successfully mapped plug-in among the not fully tested plug-ins the debugger starts exhaustive traverse of execution tree, else go to Step 1,

(4) if there is any not fully tested plug-in, go to Step 1.

The time and resource requirements of the exhaustive tests can be decreased by magnitude of orders in two ways. On one hand, we can take the advantage the large number of resources included in metacomputing environment by starting more (hundreds or thousands) test scenarios at the same time. On the other hand, we can try to reduce the complexity/size of metacomputing application as much as possible without losing the relevant parts of the application.

#### **4.2.6 Architecture of macrostep-based HARNESS debugger**

The first version of proposed architecture Figure 38 required some changes. Obviously, at least one message box plug-in is needed in the distributed virtual machine (see Figure 41). The HMPI (Harness Monitor Plug-in) got controlling facilities locally over the user's plug-in during the macrostep-based execution. It has been renamed to HMCPI (Harness Monitor & Controller Plug-in), which is able to

- (i) place the system breakpoints in the Harness kernel in order to detect the reconfiguration of DVM,
- (ii) monitor the accesses to the message boxes,
- (iii) control the execution of plug-ins.

The local HMCPIs are supervised by HSDT (Harness Systematic Debugger Tool), which was inherited from the class of HDT (Harness Debugger Tool). The new HSDT is responsible for the macrostep-based related functions and management of host-translation tables.

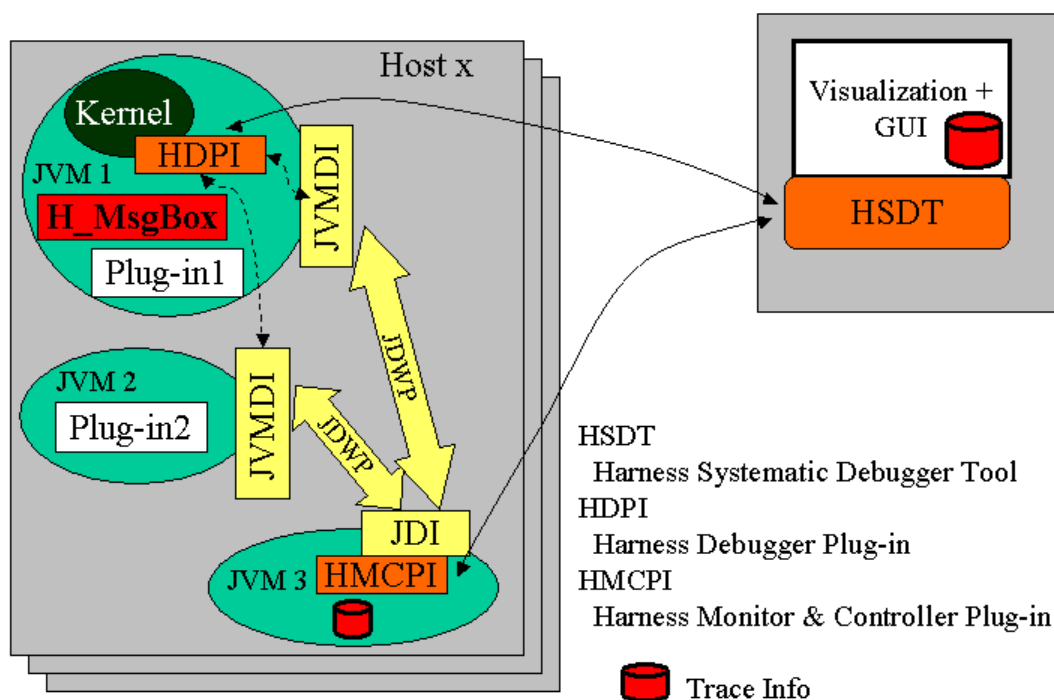


Figure 41 – Fundamental architecture of macrostep-based debugger in HARNESS

#### 4.2.7 Conclusion and related work

As we studied the related work (such as DejaVu [36], DJVM [37], DIWIDE [52] integrated in P-GRADE [50], Macrostep-by-macrostep concept [48], TotalView [38], P2D2 [39]) we realized that there is a lack of an integrated graphical systematic debugger for metacomputing applications equipped by visualization techniques. Even the most relevant grid/metacomputing projects such as Globus [45], Condor [53] and Legion [47] do not give efficient solution to the emerging debugging issues. That was the main reason for the further development of the macrostep debugging methodology, and I presented that *the macrostep-based debugging methodology can be generalised towards HARNESS metacomputing applications*. The current prototype is partly implemented under the Harness Framework as an extension of HARNESS metadepbugger.

I also plan to investigate the efficiency issues of the described methodology as well as the feasible optimisation techniques (see Section 3) that could be applied in order to reduce the complexity of testing.

## 5 Summary

The presented debugging framework provides facilities for

- Observation and control mechanisms (*macrostep based execution*) to allow the user to observe easily the computation states corresponding to the detected erroneous situations in parallel (*P-GRADE*) and metacomputing (*HARNCESS*) environments
- Methods to evaluate the correctness predicates automatically (*DIWIDE with Temporal Logic Checker*)
- Steering and optimisation mechanisms to reduce both the user interactions and the time of the debugging period (*CPN simulation, partitioning algorithm, etc.*)

The first aim of my work was to prove the correctness of the macrostep-based execution of GRAPNEL\* programs in P-GRADE environment during the debugging phase applying formal methods from the area of model verification.

In the first thesis, the formalism of coloured Petri-nets (CPN) was chosen for modelling GRAPNEL\* programs from debugging aspects. The transformation to CPN is based on the class representation of GRAPNEL\* programs following its hierarchical design concept. The generated CPN model is specified by the XML description of a widespread CPN simulation tool.

The formal description of the macrostep-based execution relies on the state-space (Occurrence graph) of the introduced Petri-net model. Then, the correctness of macrostep concept has been proven formally by the help of partial ordering Kripke-structures, which are derived from the state-space of the CPN model in case of uncontrolled running as well as macrostep-based execution.

As the second goal, the macrostep-based debugging methodology was improved, where further model checking techniques have been utilized in the area of parallel debugging. The introduced support for run-time evaluation of temporal logic specifications has been defined by state machine description. The Petri-net simulation tool can steer and optimise the traversal of state-space during the macrostep-based execution, a static analyser (a partitioning algorithm) classifies the processes into subclasses, and the Rayleigh error-model is applied for the estimation of fault density in GRAPNEL\* applications.

Finally, the macrostep-based execution was generalised towards metacomputing applications. The described approach followed the novel design methods of the HARNCESS metacomputing framework, and an adaptive and open architecture has been introduced for debugging of metacomputing applications. After the investigation of the available debugging tools and the requirement analysis for debugging of metacomputing applications, new debugging mechanisms have been developed for the unification of local and remote method calls, for transferring the consistent global states of individual processes between arbitrary debugging tools, and for the macrostep-based execution of metacomputing applications.

The results described in the theses are published in conference proceedings, journals, and demonstrated at several scientific forums and exhibitions.

On the other hand, P-GRADE development environment is gaining more and more attention from universities and IT companies from the EU and USA. P-GRADE has been installed in several educational and research institutes for using it in European and in bilateral R&D projects, e.g

University of Vienna	New University of Lisbon
Technical University of Munich	Kyushu University, Fukuoka
University of Athens	Technical University of Gdansk
Univ. of Westminster, London	Polish-Japanese Institute of Information Technology, Warsaw
University of Miskolc	Institute of Informatics, Slovakian Academy of Sciences
Autonoma Univ. of Barcelona	Institute of Computer Science, Polish Academy of Sciences

P-GRADE is also used for education in the parallel programming curricula, e.g. at the University of Miskolc, University of Westminster (London), University of Vienna and in the Polish-Japanese Institute of Information Technology (Warsaw).

The most important application of P-GRADE is the parallel version of a software package for ultra-short range weather prediction created by the Hungarian Meteorological Service. MEANDER (Mesoscale Analysis Nowcasting and DEcision Routines) can forecast extreme weather conditions using measurement data and time consuming complex simulations, thus, enabling weather alarm in time (at Lake Balaton, or for aviation service e.g.). The correctness of the core control mechanism of the package had been analyzed with the macrostep debugger and the Petri-net simulator [20].

P-GRADE has been also applied successfully in engineering for simulation of urban traffic (University of Westminster), and in chemistry for modelling of reaction-diffusion systems (Eötvös Loránd University of Budapest) [70][71].

Most of the presented scientific results have been already implemented in P-GRADE environment, and the software developers are able to take the advantages of these new debugging methods in order to increase the reliability of their software products. Recently, P-GRADE has been further developed to support the seamless migration from traditional parallel and distributed platforms towards grid environments [72] and workflow-based complex applications [73][74][75]. Therefore, the designed and debugged application can be deployed on these new platforms as well providing even more new opportunities for the end-users. The results, related to metacomputing applications, can be applied in other metacomputing or grid computing frameworks, which are finding acceptance as standard platforms for high performance and data intensive applications.

The presented work is strongly tied to two software development frameworks; P-GRADE parallel programming environment (developed by MTA SZTAKI), and HARNESS metacomputing system (developed by Emory University, Oak Ridge National Laboratory, and University of Tennessee). The future works are described at the end of each section but, as the most important one, I would like to generalise my achievements, e.g. using UML and other widespread modelling methods. The partly elaborated tools presented in Sections 3.2.2.2-3.2.2.4, 3.2.3 and 4.2.4 must be also completed, and this will address new issues to be solved.

## 6 Acknowledgement

This dissertation would not have been completed without the advice and help of my mentors, colleagues, friends, family and the financial support of several organizations. I am sincerely grateful to all of them for their constructive contributions to this work.

First of all, I would like to thank Prof. Péter Kacsuk (MTA SZTAKI) and dr. Ruth Bars (BME) for their advisory work as well as Prof. Ferenc Vajda (BME) and dr. István Vajk (BME) for their valuable support.

I would like to thank Prof. Yiannis Cotronis (University of Athens), Floros Vangelis (University of Athens), and Bertalan Vécsei (ELTE/MTA SZTAKI) for their assistance with various aspects of Petri nets, modelling and simulation with Design/CPN, and also dr. Gábor Dózsa (MTA SZTAKI/IBM J. Watson) for his valuable help concerning GRAPNEL language. The work described in Section 2.1 is funded in part by the Hungarian Technological Development Committee (OMFB) in the framework of Hungarian-Greek Tét Project GR-25/96, and by the Hungarian National Science Research Fund (OTKA) Contract Num: F022105.

Concerning the Section 2.2, I would like to thank José C. Cunha (Universidade Nova de Lisboa), dr. Joao Lourenco (Universidade Nova de Lisboa), and József Kovács (MTA SZTAKI) for their assistance with various aspects of state-of-the-art debugger methods and tools. The work presented in this section was supported in part the Hungarian-Portuguese Intergovernmental Tét project, No. P-12/97.

I would like to thank dr. István Majzik (BME) for his excellent post-graduate course on formal methods, and also dr. Dániel Varró (BME) for his valuable comments, which motivated and assisted the work presented in Section 2.3 and partly in Section 3.

Concerning Section 3.1 and 3.2, I would like to thank Prof. Wolfgang Shreiner (RISC-LINC), Gábor Kusper (RISC-LINZ) for their assistance with various aspects of the TLC system, as well as dr. Charaf Hassan (BME) for his valuable post-graduate courses on software engineering, which motivated and assisted the presented work, funded in part by the ÖAD-WTZ Project A32/2000.

I would like to thank Prof. Vaidy Sunderam (Emory University), dr. Mauro Migliardi (Universita' di Genova) and Dawid Kurzyniec (Emory University) for their assistance with various aspects of the Harness metacomputing system. The work presented in Section 4 was supported in part by U.S. DoE grant # DE-FG02-99ER25379, NSF grant ACI-9872167 and National Research Grant (OTKA) registered under No. T-032226.

Last but certainly not least; I would like to thank my family; my parents, my brother, and my wife for their support and patience.

## 7 REFERENCES

- [1] José C. Cunha, João Lourenço, Vitor Duarte: Debugging of Parallel and Distributed Programs. In the book: *Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments* (Chapter 5), pp. 101-136, Nova Science Publishers New York, 2001
- [2] Dieter Kranzlmüller, Axel Rinnac: Parallel Program Debugging with MAD - A Practical Approach. *International Conference on Computational Science 2003*, pp. 201-212
- [3] A. Tarafdar and V. K. Garg: Predicate control for active debugging of distributed programs, *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98)*, pages 763-769, Los Alamitos, March 30-April 3 1998
- [4] M. Frey and M. Oberhuber: Testing and Debugging Parallel and Distributed Programs with Temporal Logic Specifications, *Proc. of Second Workshop on Parallel and Distributed Software Engineering 1997*, pages 62-72, Boston, May 1997
- [5] João Lourenço, José C. Cunha: Fiddle: A Flexible Distributed Debugging Architecture, *ICCS 2001*, San Francisco, CA, USA, 2001, pp. 821-830
- [6] E. Bruneton and J.-F. Pradat-Peyre: Automatic verification of concurrent Ada programs, *Ada-Europe'99 International Conference on Reliable Software Technologies (Santander, Spain)*, pp. 146—157
- [7] Alberto Lluch-Lafuente, Stefan Leue and Stefan Edelkamp: Partial Order Reduction in Directed Model Checking, In: *Proceedings of the 9th International SPIN Workshop on Model Checking Software*, Springer LNCS, Grenoble, April 2002
- [8] E. M. Clarke, O. Grumberg, M. Minea, D. Peled. State space reduction using partial order techniques. *Software Tools for Technology Transfer*, vol. 3, no. 1, Springer Verlag, 1999, pp. 279-287.
- [9] R. Kurshan, V. Levin, M. Minea, D. Peled, H. Yenigün. Static partial order reduction. *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lisbon, Portugal, March/April 1998, pp. 345-357
- [10] P. Kacsuk, G. Dózsa, R. Lovas: The GRADE Graphical Parallel Programming Environment, In the book: *Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments* (Chapter 10), pp. 231-247, Nova Science Publishers New York, 2001
- [11] A. Bäcker, D. Ahr, O. Krämer-Fuhrmann, R. Lovas, H. Mierendorff, H. Schwamborn, J. G. Silva, K. Wolf: WINPAR, Windows-Based Parallel Computing, In: *Parallel Computing: Fundamentals, Applications and New Directions*, Series of Advances in Parallel Computing, Vol. 12, pp. 495-502, Elsevier Science, 1998
- [12] K. Jensen: *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*. Monographs in Theoretical Computer Science, Springer-Verlag, 1992
- [13] K. Jensen: *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods*. Monographs in Theoretical Computer Science, Springer-Verlag, 1994
- [14] M. Software “Desgin/CPN. A Tool Package Supporting the Use of Colored Petri Nets” Tech. Rep., Meta Software Corporation, Cambridge, MA, USA, 1991
- [15] Z. Tsiatsoulis, G. Dozsa, Y. Cotronis, P. Kacsuk: Associating Composition of Petri Net Specifications with Application Designs in Grade, *Proc. of the Seventh Euromicro Workshop on Parallel and Distributed Processing*, Funchal, Portugal, pp. 204-211, 1999
- [16] P. Rondogiannis, M.H.M Cheng. Petri-net-based deadlock analysis of Process Algebra programs. *Science of Computer Programming*, 1994. Vol. 23 (1), pp. 55-89
- [17] Matthew B. Dwyer, Lori A. Clarke, and Kari A. Nies. A compact petri net representation for concurrent programs. Technical Report TR 94-46, University of Massachusetts, Amherst, 1994
- [18] Jim Greene: Ensuring Delivery of Highly Reliable, Complex Software Releases, QSM White Paper, October 2003
- [19] I. Majzik, A. Pataricza and A. Bondavalli: Stochastic Dependability Analysis of System Architecture Based on UML Models. In R. de Lemos, C. Gacek and A. Romanovsky (eds.): *Architecting Dependable Systems*, LNCS-2667, Springer Verlag, Berlin, 2003, pp 219-244

- [20] R. Lovas, P. Kacsuk, A. Horvath, A. Horanyi: Application of P-GRADE Development Environment in Meteorology, *Journal of Parallel and Distributed Computing Practices*, Special issue on DAPSYS 2002, Nova Science Publishers (accepted for publication)
- [21] P. Kacsuk, G. Dózsa, T. Fadgyas and R. Lovas: The GRED Graphical Editor for the GRADE Parallel Program Development Environment, In: *High-Performance Computing and Networking*, Lecture Notes in Computer Science, Vol. 1401, pp. 728-737, Springer Verlag, 1998
- [22] G. Dózsa, D. Drótos, R. Lovas: Translation of a High-Level Graphical Code to Message-Passing Primitives in the GRADE Programming Environment, In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Science, Vol. 1908, pp. 258-265, Springer-Verlag, 2000
- [23] B. Vécsei, R. Lovas: Debugging method for parallel programs based on Petri-net representation, *Proceedings of MicroCAD 2004*, Miskolc, Hungary, pp. 413-420, 2004
- [24] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.
- [25] D. Drusinsky. The Temporal Rover and the ATG Rover. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, 7th International SPIN Workshop, volume 1885 of Lecture Notes in Computer Science, pages 323-330, Stanford, CA, August 30 - September 1, 2000. Springer.
- [26] J. Hakansson. Automated Generation of Test Scripts from Temporal Logic Specifications. Master's thesis, Uppsala University, Sweden, 2000.
- [27] P. Kacsuk, G. Dózsa, T. Fadgyas, and R. Lovas. The GRED Graphical Editor for the GRADE Parallel Program Development Environment. *Future Generation Computer Systems*, 15:443-452, 1999.
- [28] D. Kranzlmüller. Event Graph Analysis for Debugging Massively Parallel Programs. PhD thesis, Johannes Kepler University, September 2000.
- [29] D. Kranzlmüller and J. Volkert. NOPE: A Nondeterministic Program Evaluator. In P. Zinterhof et al., editors, *Parallel Computation*, Proceedings of ACPC'99, 4th International ACPC Conference, volume 1557 of Lecture Notes in Computer Science, pages 490-499, Salzburg, Austria, February 16-18, 1999. Springer, Berlin.
- [30] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872-923, May 1994.
- [31] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems Specification*. Springer, Berlin, 1992.
- [32] C. Pancake and R. Netzer. A Bibliography of Parallel Debuggers, 1993 Edition. *SIGPLAN Notes*, 28(12):169-186, 1993. Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, CA, May 1993.
- [33] J. Kovacs, G. Kusper, R. Lovas, W. Shreiner: Integrating Temporal Assertions into a Parallel Debugger, In: *EuroPar 2002 Parallel Processing*, Lecture Notes in Computer Science, vol. 2400, pp. 113-120, Springer-Verlag, 2002
- [34] R. Lovas, B. Vécsei: Integration of formal verification and debugging methods in P-GRADE environment, In: *Distributed and Parallel Systems: Cluster and Grid Computing*, Kluwer International Series in Engineering and Computer Science, Vol. 777, pp. 83-92, 2004
- [35] R. Lovas, P. Kacsuk: Enhanced Macrostep-based Debugging Methodology for Parallel Programs, *The Third Conference of PhD Students in Computer Science*, pp. 72, Szeged, Hungary, 2002
- [36] Bowen Alpern, Jong-Deok Choi, Ton Ngo, Manu Sridharan, John Vlissides. A Perturbation-Free Replay Platform for Cross-Optimized Multithreaded Applications. IBM Research Report, RC 21864, 22 September 2000.
- [37] Ravi Konuru, Harini Srinivasan, and Jong-Deok Choi. Deterministic Replay of Distributed Java Applications. *14<sup>th</sup> International Parallel & Distributed Processing Symposium*, pages 219-228, May 2000.
- [38] Etnus, LLC. TotalView debugger. Available online at <http://www.etnus.com/Products/TotalView/index.htm>

- [39] Robert Hood. The p2d2 project: building a portable distributed debugger. Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, May 22 - 23, 1996, Philadelphia, PA USA
- [40] Gy. Csértán, I. Majzik, A. Pataricza, S. C. Allmaier: Reachability Analysis of Petri-nets by FPGA Based Accelerators. Proc. Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS'98), 1998, Szczyrk, Poland, September 2-4, pages 307-312
- [41] D. Latella, I. Majzik, M. Massink: Automatic Verification of UML Statechart Diagrams using the SPIN Model-Checker. Formal Aspects of Computing, 1999, volume 11, number 6, pages 637-664, Springer Verlag
- [42] M. Migliardi, V. Sunderam, A. Geist, J. Dongarra. Dynamic Reconfiguration and Virtual Machine Management in the Harness Metacomputing System, Proc. of ISCOPE98, pp. 127-134, Santa Fe', New Mexico (USA), December 8-11, 1998.
- [43] Sun Microsystems, Inc. Documentation of Java Platform Debugger Architecture. Available online at <http://java.sun.com/products/jpda/doc/>. Dec 2000.
- [44] Jong-Deok Choi and Harini Srinivasan. Deterministic Replay of Java Multithreaded Applications. ACM SIGMETRICS Symposium on Parallel and Distributed Tools, pages 48-59, August 1998.
- [45] I. Foster and C. Kesselman. Globus: a Metacomputing Infrastructure Toolkit. International Journal of Supercomputing Application and High Performance Computing, Vol. 11, Number 2, pp. 115-128, 1997.
- [46] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Mancheck and V. Sunderam. PVM: Parallel Virtual Machine a User's Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, MA, 1994.
- [47] A. Grimshaw, W. Wulf, J. French, A. Weaver and P. Reynolds. Legion: the next logical step toward a nationwide virtual computer, Technical Report CS-94-21, University of Virginia, 1994.
- [48] P. Kacsuk. Systematic Macrostep Debugging of Message Passing Parallel Programs. Future Generation Computer Systems, Vol. 16, No. 6, pp. 609-624, 2000.
- [49] P. Kacsuk, R. Lovas, and J. Kovács. Systematic Debugging of Parallel Programs in DIWIDE Based on Collective Breakpoints and Macrosteps. In P. Amestoy et al., editors, 5<sup>th</sup> Euro-Par Conference, Vol. 1685 of Lecture Notes in Computer Science, pages 90-97, Toulouse, France, August 31 - September 3, 1999. Springer.
- [50] P. Kacsuk, G. Dózsa, T. Fadgyas, R. Lovas. GRADE: A Graphical Programming Environment for Multicomputers. Computer and Artificial Intelligence. 17 (5) :417-427. (1998)
- [51] G. Dózsa: Visual Programming to Support Parallel Program Design, In: Parallel Program Development for Cluster Computing, Methodology, Tools and Integrated, Nova Science Publishers, Inc. pp. 17-44, 2001
- [52] J. Kovacs, P. Kacsuk. The DIWIDE Distributed Debugger on Windows NT and UNIX Platforms, Distributed and Parallel Systems, From Instruction Parallelism to Cluster Computing, Eds.: P. Kacsuk and G. Kotsis, Cluwer Academic Publishers, 2000.
- [53] M. J. Litzkow, M. Livny and M. W. Mutka. Condor – A Hunter of Idle Workstations, Proc. of the 8<sup>th</sup> International Conference on Distributed Computer Systems, pp. 104-111, IEEE Press, June 1998.
- [54] R. Lovas, V. Sunderam: Debugging of Metacomputing Applications, Proc. of the 16th International Parallel and Distributed Processing Symposium (IPDPS-JPDC), pp. 119 + CD-ROM, Fort Lauderdale, FL, USA, 2002
- [55] R. Lovas, V. Sunderam: Extension of macrostep debugging methodology towards metacomputing applications, In: Computational Science - ICCS 2001, Lecture Notes in Computer Science, Vol. 2074, p. 263-272, Springer Verlag, 2001
- [56] R. Lovas, V. Sunderam: A Metadebugger Prototype for the HARNESS Metacomputing Framework, Proc. of the Tenth IEEE International Symposium on High Performance Distributed Computing, pp. 427-428, San Francisco, CA, USA, 2001
- [57] Søren Christensen and Torben Bisgaard Haagh: Design/CPN, Overview of CPN ML Syntax, Version 3.0

- [58] Daniel Drotos, Gabor Dozsa, Peter Kacsuk: GRAPNEL to C translation in the GRADE environment. In: *Parallel Program Development for Cluster Computing, Methodology, Tools and Integrated*, Nova Science Publishers, Inc. pp. 17-44, 2001
- [59] Robert Lovas: GRED grafikus editor a GRADE integrált vizuális programozási környezethez, MSc Theses, Budapest University of Technology and Economics, 1998
- [60] Valmari, A.: A stubborn attack on state explosion. *Proc. Second Workshop on Computer-Aided Verification*, New Brunswick, NJ, June 1990. LNCS 531. Berlin, Heidelberg, New York: Springer-Verlag, 1990, pp. 156-165
- [61] Godefroid, P., Pirottin, D.: Refining dependencies improves partial-order verification methods. *Proc. Fifth Workshop on Computer-Aided Verification*, Elounda, Greece, June 1993. LNCS 697. Berlin, Heidelberg, New York: Springer-Verlag, 1993, pp. 438-449
- [62] Godefroid, P., Wolper, P.: A partial approach to model checking. *Proc. Sixth Annual IEEE Symposium on Logic in Computer Science*, Amsterdam, July 1991, pp. 406-415
- [63] Peled, D.: Combining partial order reductions with on-the-fly model-checking. *Proc. Sixth Workshop on Computer-Aided Verification*, Stanford, CA, June 1994. LNCS 818. Berlin, Heidelberg, New York: Springer-Verlag, 1994, pp. 377-390
- [64] Holzmann, J.: *Design and Validation of Computer Protocols*. Prentice Hall, 1991
- [65] Allinea Software Ltd.: *Distributed Debugger Tool v1.8, User Guide*, 2004
- [66] D. Kranzlmüller, S. Grabner, J. Volkert. Event Graph Visualization for Debugging Large Applications. *Proc. SPDT'96, ACM SIGMETRICS Symp. on Parallel and Distr. Tools*, Philadelphia, USA, pp. 108-117, 1996
- [67] Henryk Krawczyk, Piotr Kuzora, Marcin Neyman, Jerzy Proficz and Bogdan Wiszniewski: STEPS - a Tool for Structural Testing of Parallel Software. In the book: *Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments* (Chapter 16), pp. 334-354, Nova Science Publishers New York, 2001
- [68] José C. Cunha, João Lourenço and Vitor Duarte: The DDBG Distributed Debugger. In the book: *Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments* (Chapter 13), pp. 292-303, Nova Science Publishers New York, 2001
- [69] Lovas R., Horváth Á.: Ultrarövidtávú meteorológiai előrejelző rendszer párhuzamosítása a P-GRADE fejlesztőkörnyezettel, *Proc. of NETWORKSHOP '2002*, pp. 56 + CD-ROM, Eger, Hungary, 2002
- [70] R. Lovas, P. Kacsuk, I. Lagzi, T. Turányi: Unified development solution for cluster and grid computing and its application in chemistry, In: *Computational Science and Its Applications – ICCSA 2004: International Conference, Assisi, Italy, Lecture Notes in Computer Science*, Vol. 3044, pp. 226-235, Springer-Verlag, 2004
- [71] I. Lagzi, R. Lovas, T. Turányi: Development of a grid enabled chemistry application, In: *Distributed and Parallel Systems: Cluster and Grid Computing*, Kluwer International Series in Engineering and Computer Science, Vol. 777, pp. 137-144, 2004
- [72] P. Kacsuk, R. Lovas, J. Kovács, F. Szalai, G. Gombás, N. Podhorszki, A. Horváth, A. Horányi, I. Szeberényi, T. Delaitre, G. Terstyánszky, A. Gourgoulis: Demonstration of P-GRADE job-mode for the Grid, In: *Euro-Par 2003 Parallel Processing, Lecture Notes in Computer Science*, Vol. 2790, pp. 1281-1286, Springer-Verlag, 2003
- [73] P. Kacsuk, G. Dózsa, J. Kovács, R. Lovas, N. Podhorszki, Z. Balaton, G. Gombás: P-GRADE: a Grid Programming Environment, *Journal of Grid Computing*, Volume 1, Issue 2, 2003, Pages 171 – 197
- [74] R. Lovas, G. Dózsa, P. Kacsuk, N. Podhorszki, D. Drótos: Workflow Support for Complex Grid Applications: Integrated and Portal Solutions, In: *Grid Computing – Second European AcrossGrids Conference, AxGrids 2004*, Nicosia, Cyprus, *Lecture Notes in Computer Science*, Vol. 3165, pp. 129-138, Springer-Verlag, 2004
- [75] Cs. Németh, G. Dózsa, R. Lovas and P. Kacsuk: The P-GRADE Grid portal, In: *Computational Science and Its Applications – ICCSA 2004: International Conference, Assisi, Italy, Lecture Notes in Computer Science*, Vol. 3044, pp. 10-19, Springer-Verlag, 2004

## 8 Appendix

### 8.1 Case study: *Producer-buffer problem and its CPN model*

Take a P-GRADE parallel program, the Buffer application (see Figure 42), which consists of three processes, namely *Producer*, *Consumer*, and *Buffer* processes (see Figure 42).

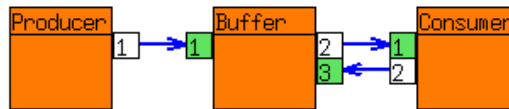


Figure 42 – Application level of Buffer program

- The *Producer* process (see Figure 43) generates a finite number of items, and sends them to the Buffer process item-by-item.
- The *Consumer* process (see Figure 43) sends requests periodically to the Buffer process for consumable items, and gets them item-by-item.
- The *Buffer* process (see Figure 43) can receive items from the Producer and also eventually forwards them to the Consumer process (when it is ready to receive a new item). Thus, the Buffer must store temporarily the produced but not consumed elements.

The Buffer process has a finite capacity, since its rounded buffer may have three distinguished states: full, empty, not full and not empty. Three different execution paths can be followed in the Buffer process depending on the state of its rounded buffer: If it is empty –obviously– no requests for items can be served, only the new incoming items should be stored (only the communication port connected to the Producer is watched, the Consumer must wait). Oppositely, if the buffer is full no more items can be stored, and the requests for storing new elements should be hanged until the consumer takes an item from the store (only the input port connected to the Consumer is watched). If the buffer is neither empty nor full it can serve both kinds of request (see the alternative input communication action, labelled 'I1' in Buffer process). In general, the behaviour of Buffer application is non-deterministic, due to the alternative communication operations.

The Figure 43 shows the structure of the Buffer process in the middle. The left branch of the topmost condition contains the code for the general status of the buffer when it can either receive or store items. The right branch of the condition contains the second conditional branch: left for the empty buffer, right for the full one.

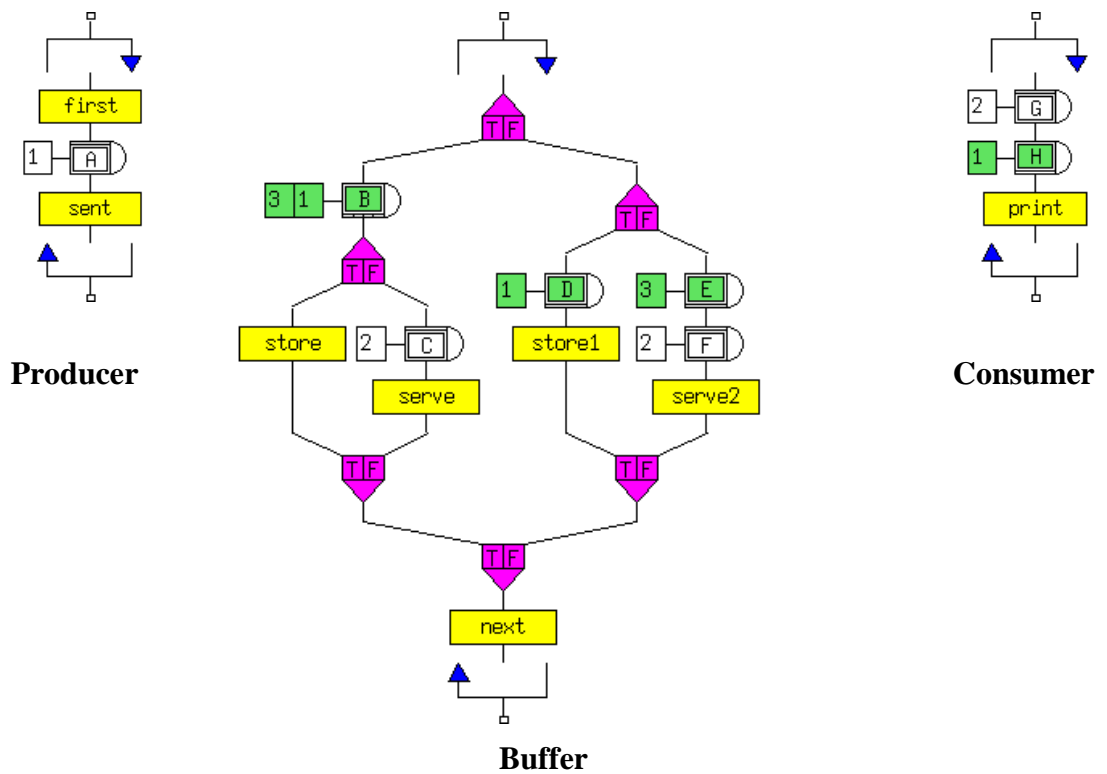


Figure 43 – Process level of Buffer program

The generated CPN model based on the transformation rules (see Section 2.1) can be found in the following three figures (see Figure 44, Figure 45, and Figure 46).

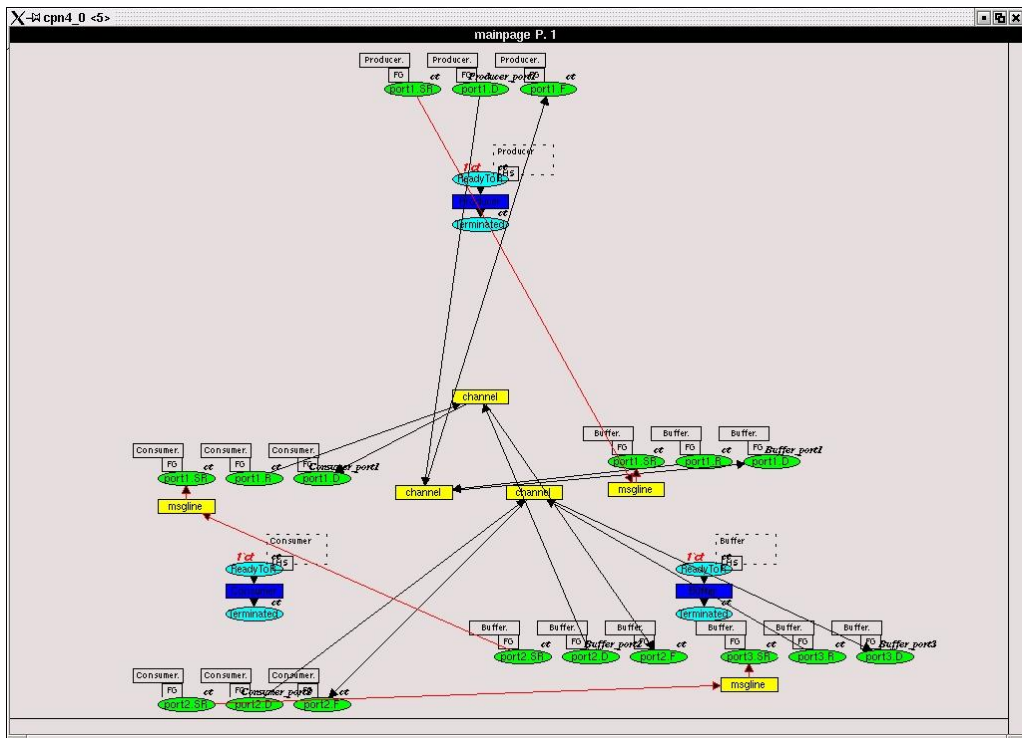


Figure 44 – CPN model of Buffer program: application level

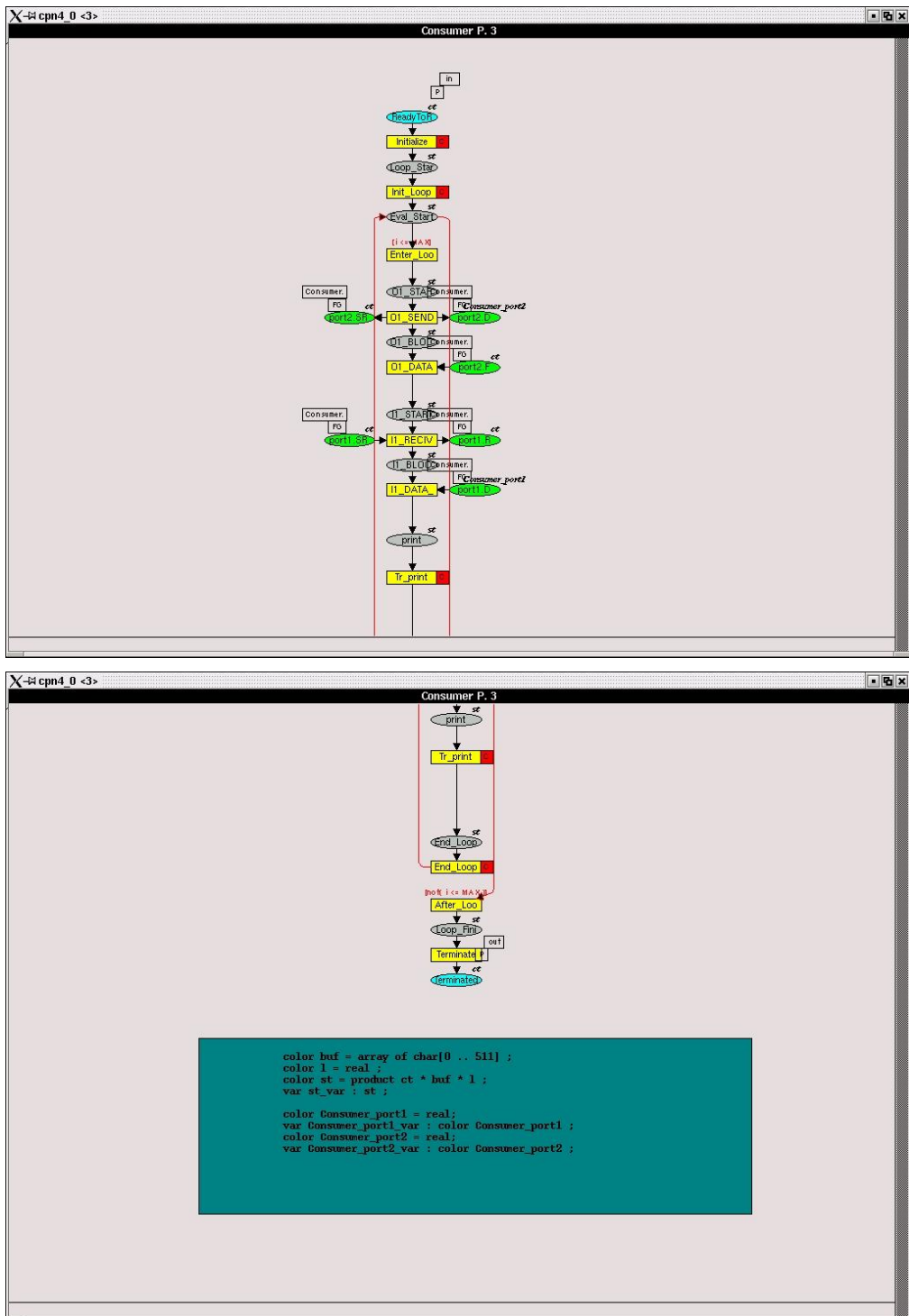


Figure 45 – CPN model of Consumer process



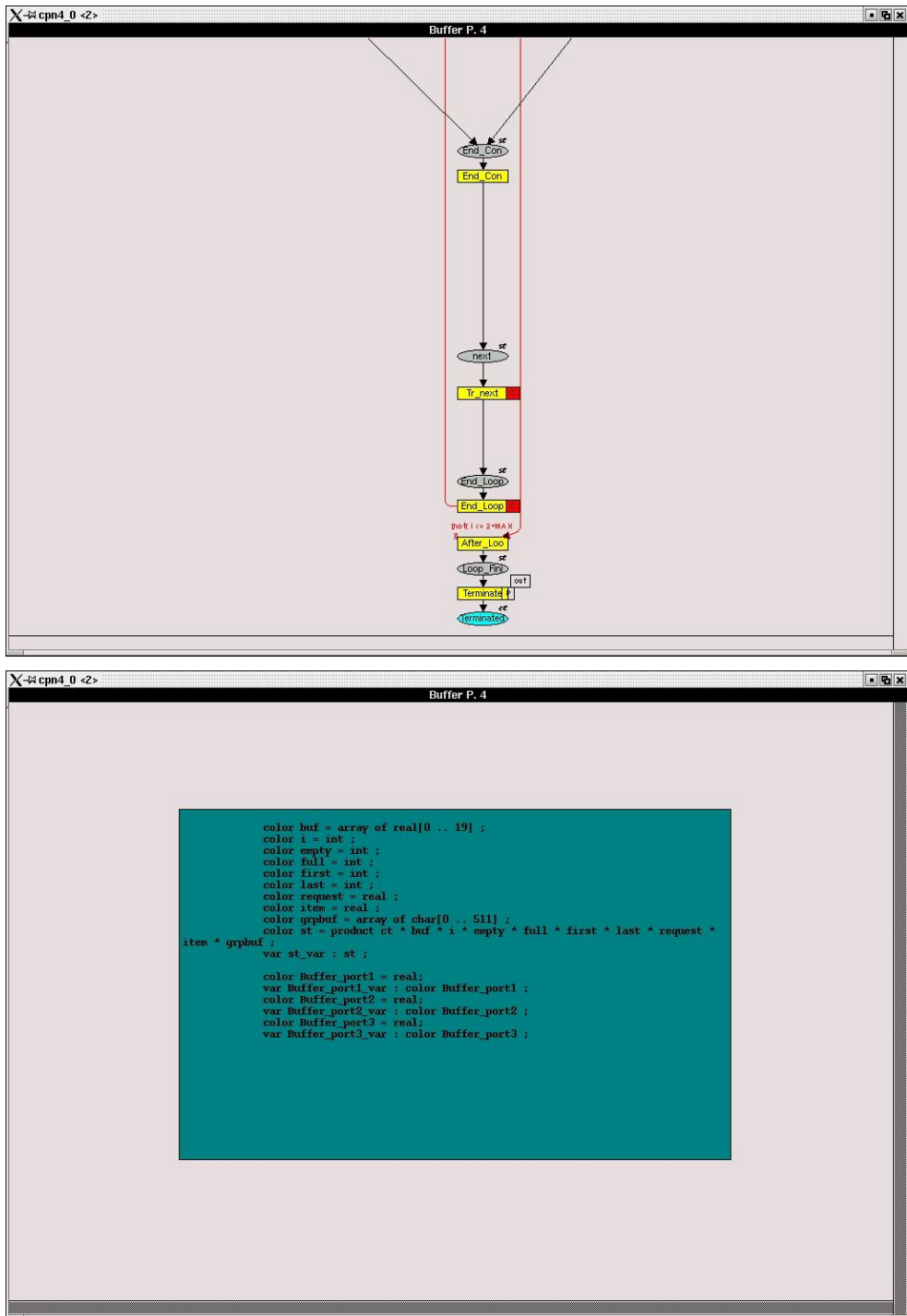


Figure 46 – CPN model of Buffer process