

Multilevel Debugging of Parallel Message Passing Programs

by

Jan Bækgaard Pedersen

Cand. Scient. (M.Sc.)
Department of Computer Science
Institute of Mathematics
University of Århus
Århus, Denmark, 1997

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)

~~We accept this thesis as conforming
to the required standard~~

The University of British Columbia

July 2003

© Jan Bækgaard Pedersen, 2003

In presenting this thesis/essay in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying for this thesis for scholarly purposes may be granted by the Head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

21 July 2003

Date

Department of Computer Science
The University of British Columbia
2366 Main mall
Vancouver, BC
Canada V6T 1Z4

Abstract

"I am not young enough to know everything"

— James M. Barrie

"Errare humanum est" - To err is human (Hieronymus, Epistle 57, 12); this fact has been known throughout time, and inevitably this means that humans writing computer programs are bound to introduce errors. With computers operating in Frankenstein's Igor mode, *'Your wish is my command'*, executing instructions without questioning their validity, errors introduced by humans are carried out. When adding parallel programming with message passing an error in one process can spread like a virus through message passing to other processes.

Much research has been done on debugging sequential programs, and most of these theories and results apply directly to parallel programs, but the set of potential errors dramatically increases in size when introducing parallelism and message passing. Not only can one process fail, but sets of processes can deadlock, computational errors can be propagated from process to process, thus infecting otherwise correct programs. Correct programs can stop working because of the underlying implementation of the message passing system.

We propose a framework for debugging parallel message passing programs: a multilevel approach that divides errors into separate groups at various levels from the well known sequential errors, such as stray pointers and array out of bound, to deadlock caused by incorrect message passing code, protocol errors and buffer allocation problems. We show the validity of this approach by developing new debugging techniques and analyses, and by implementing these in Millipede, a prototype multilevel debugger written for C programs that use the PVM message passing system.

Contents

Abstract	ii
Contents	iii
List of Figures	vii
List of Tables	x
Acknowledgements	xi
1 Introduction	1
1.1 The Debugging Problem	2
1.2 Problem Definition	4
1.3 Thesis Statement	6
1.4 Contributions	6
2 Background and Related Work	10
2.1 Background and Rationale	10
2.1.1 The Parallel Programming Domain	11
2.2 The Debugging Process	15
2.2.1 Iterative debugging	15
2.2.2 The Why, How and What of Errors	16
2.3 Related Work	20
2.3.1 Program Development Environments	20
2.3.2 Visualization Tools	22
2.3.3 Extension of Sequential Debuggers	23
2.3.4 Replay Tools/Debuggers	25
2.3.5 Relative Debuggers	26
2.3.6 Language Support for Communication	26
2.3.7 Summary of Related Work	26

2.4	Top-down versus bottom-up debugging	27
2.5	Multilevel Debugging	28
2.6	Error Classification	29
2.7	Tool Development	31
2.7.1	Automation	33
2.8	Tool Support for Parallel Program Development	34
3	Millipede - A Prototype Multilevel Parallel Debugger	36
3.1	Design Criteria	36
3.2	The legs of Millipede	37
3.2.1	Overview	38
3.2.2	Implementation	39
3.3	The Sound of Little Legs Running	40
4	Sequential Debugging of Parallel Processes	43
4.1	The Sequential Debugging Module	44
4.2	Limitations	45
4.3	Examples	47
4.3.1	Division by Zero Error	47
4.3.2	Memory Errors	48
4.4	Implementation Details for the Sequential Debugging Module	50
4.5	Summary	50
5	Message Debugging	51
5.1	Interactive Message Debugging	51
5.2	Message Queries	54
5.3	User Defined Queries	54
5.4	Built-in Message Queries	57
5.5	Discussion	60
5.6	Summary	61
6	Deadlock Detection and Correction	62
6.1	Deadlock Detection and Correction	62
6.2	Description of Problem	63
6.3	The Algorithm	63
6.4	Algorithm accuracy	67
6.5	Message tags	82
6.6	Summary	83

7	Protocol Conformance Checking	84
7.1	Between Testing and Verification	84
7.2	Protocol Checking and Verification	86
7.3	Protocol Constraint Specification	87
7.3.1	Protocol Contents	87
7.4	The PCSL Grammar and Semantics	88
7.5	Examples	91
7.5.1	The Simplest Protocol	92
7.5.2	Pipe-and-Roll Matrix Multiplication	93
7.5.3	A Partial Differential Equation Solver	95
7.6	Online Checking	98
7.6.1	Strictness	99
7.7	Offline Checking	99
7.8	Protocol Prediction	100
7.9	Implementation	100
7.10	Discussion	101
7.10.1	State Dependent Communication	102
7.11	Summary	103
8	Buffer Allocation in Message Passing Programs	104
8.1	Motivation and Related Work	105
8.2	Buffer Allocation Problems	107
8.3	The Nonblocking Buffer Allocation Problem	109
8.4	Approximations of BAP using NBAP	112
8.5	Discussion	116
8.6	Summary	117
9	Conclusion and Future Work	118
9.1	Conclusion	118
9.1.1	The Sequential Level	119
9.1.2	The Message Level	119
9.1.3	The Protocol Level	120
9.1.4	Summary	122
9.2	Future Work	123
9.2.1	Further Development	123
	Bibliography	125

Appendix A	A Complete Example of a Millipede Session	132
Appendix B	The PCSL Grammar and Semantics	134
B.1	The PCSL Grammar and Semantics	134
B.2	A Complete Example Using PCSL/MOPED	135
Appendix C	The MQL Grammar	137
C.1	The Millipede Query Language Grammar	137
Appendix D	Millipede Screen Shots	139
Appendix E	Theoretical Framework for The Buffer Allocation Problems	141
E.1	Definitions	141
E.1.1	The Graph Based Framework	142
E.1.2	Colouring the Communication Graph	143
E.2	Useful Lemmas	145
E.3	Buffer Allocation in Systems with Receive Side Buffers	147
E.3.1	The Buffer Allocation Problem	147
E.3.2	The Buffer Sufficiency Problem	149
E.3.3	The Nonblocking Buffer Allocation Problem	153
E.4	Proof of Correctness of the Nonblocking Buffer Allocation Algorithm	155
E.5	Buffer Allocation in Systems with Send Side Buffers	157
E.6	Buffer Allocation in Systems with Send and Receive Side Buffers	158
E.7	Buffer Allocation in Channel Based Systems	162
E.7.1	The Buffer Allocation Problem	163
E.7.2	The Buffer Sufficiency Problem	165
E.7.3	The Nonblocking Buffer Allocation Problem	169
E.8	Summary	169

List of Figures

2.1	Using default buffers	12
2.2	Explicitly allocated buffers	14
2.3	The sequential versus the parallel programming domain	15
2.4	Top-down versus bottom-up debugging	27
2.5	Message passing widens the cause/effect chasm	30
3.1	Implementation of Millipede	39
3.2	Examples of redefined PVM functions	40
4.1	Example of the importance of logging PVM function return values	45
4.2	An application with communication library and Millipede	46
4.3	Sequential code with divide by zero	47
4.4	Using Gdb for sequential debugging	48
4.5	Source code with a memory error	49
4.6	Using Purify to locate a memory error	49
5.1	Missing value in a log file	52
5.2	Inspecting and changing message content	53
5.3	Executing the <code>match</code> query	56
5.4	MQL code for the <code>match</code> query	57
5.5	Executing the <code>locate</code> query	58
5.6	MQL code for the <code>locate</code> query	58
5.7	Executing the <code>status</code> query	59
5.8	Executing the <code>dump</code> query	60
6.1	A simple Error	64
6.2	Executing the matching algorithm	67
6.3	All valid communication configurations in \mathcal{C}_2	68
6.4	The set $\overline{B}(v_1, 1)$	69

6.5	Example of increasing \mathcal{B} sets	70
6.6	Example of overlapping \mathcal{B} sets	71
6.7	Configurations reachable in k steps from the configuration 0011	72
6.8	Intersections in \mathcal{C}_2	77
6.9	Failure rate for the deadlock correction algorithm	82
6.10	Introducing message tags	83
7.1	Adding elements to the symbol table	88
7.2	How to check a message against a protocol line	91
7.3	Semantics for a PCSL line	91
7.4	Example of $\beta[\]\{ \}(\) \rightarrow \beta[\]\{ \}(\)$;	92
7.5	Example of $\beta[\]\{i\}(\) \rightarrow \beta[\]\{(i+1)\%n\}(\) :: \forall i : 0 \leq i \ \&\& \ i \leq n-1$;	93
7.6	Algorithm for the master of the pipe-and-roll matrix multiplication	93
7.7	Algorithm for the slave of the pipe-and-roll matrix multiplication.	94
7.8	The pipe-and-roll part of the matrix multiplication algorithm	95
7.9	Algorithm for the master algorithm for a differential equation solver	96
7.10	Algorithm for slave algorithm for a differential equation solver	96
7.11	\mathcal{P}_1 – Version 1 of the protocol specification	96
7.12	\mathcal{P}'_1 – Extended version 1 of the protocol specification	97
7.13	Graphical representation of \mathcal{P}_2	98
7.14	\mathcal{P}_2 – Version 2 of the protocol specification	98
7.15	\mathcal{P}_3 – Version 3 of the protocol specification	98
7.16	\mathcal{P}'_3 – Extended version 3 of the protocol specification	99
7.17	The four different stages of the pipe operation	102
8.1	An unsafe communication graph	108
8.2	A general t-ring	109
8.3	Communication dependences	110
8.4	The NBAP algorithm	110
8.5	Communication graph for a 2×2 worker system	111
8.6	Executing the nbap command in Millipede	112
8.7	Detailed information about buffer requirements using nbap	112
8.8	Examples of worst and best case approximations	113
8.9	Introduction of epochs into a communication graph	114
8.10	Shortening the arrival intervals using epochs	115
8.11	Sub-epochs	116
8.12	Barriers using asynchronous communication	117

A.1	A complete sequential debugging session	133
B.1	The PCSL BNF grammar	134
B.2	Semantics for the PCSL grammar	135
B.3	A complete example using MOPED	136
C.1	The MQL grammar	138
D.1	The Millipede startup screen	139
D.2	Screen shot illustrating interactive message debugging	140
D.3	Screen shot showing the status monitor	140
E.1	Order of execution can cause deadlock	141
E.2	A communication graph with a 2-ring	143
E.3	Dependency cycle in $G(S)$	143
E.4	Construction of G	148
E.5	The construction of the components	150
E.6	The disperser component	151
E.7	$v_{i,c+k}$ is communication dependent on $v_{i,c}$	154
E.8	Algorithm for nonblocking buffer assignment	154
E.9	A 2×2 worker process mesh	156
E.10	Nullifying send side token pools	159
E.11	Reduction from 3SAT to NBAP _{sr}	160
E.12	The clause representation in epoch j	164
E.13	Simulating m tokens by m components	166

List of Tables

2.1	Dimension 1. Why is an error hard to find?	17
2.2	Dimension 2. How is an error found?	18
2.3	Dimension 3. What is the root cause of the error?	20
5.1	The Senders relation.	54
5.2	The Receivers relation.	55
5.3	The SentMessages relation.	55
5.4	The ReceivedMessages relation.	55
5.5	The built-in queries	60
6.1	Examples of the size of $B(v, e)$	73
6.2	Distances between valid configurations in $\mathcal{V}_3 \subset \mathcal{C}_3$	74
6.3	The rate of growth of c_i	75
6.4	Number of valid configuration at different distances in \mathcal{V}_n	76
6.5	Intersection sets for various values of k and k' in \mathcal{C}_2	77
6.6	Sizes of intersecting B sets	78
7.1	The MOPED Strictness levels	99
7.2	Prediction table for the \mathcal{P}'_3 protocol specification	100
9.1	Results for various buffer placements schemes	122
E.1	Results for various buffer placement schemes	170

Acknowledgements

“Kind words can be short and easy to speak, but their echoes are truly endless.”

— *Mother Theresa of Calcuta*

I would like to thank Alex Brodsky for taking such an active interest in the buffer allocation aspect of message passing programming. The work presented in chapter 8 is done in close collaboration with him—many thanks for the hard work on the proofs. I would also like to thank my supervisor, Professor Alan Wagner, for his persistence in pushing the work on the buffer allocation problems, and for his financial support and ability to balance practice and theory in my dissertation. In addition, many thanks go to Professor Peter Welch from the University of Canterbury at Kent, Professor Dyke Stiles at Utah State University in Logan, Utah, and WOTUG for conference support on numerous occasions. I would also like to thank Doctor Bettina Speckmann for her help on proof reading the material presented in section 6. In addition, many thanks are extended to Yvonne Coady for always telling me that I was on the right track and to Chamath Keppitiyagama, Joon Suan Ong and Dmitry Brodsky for everyday support in the DSG. Also thanks to Professor Norm Hutchinson and Professor Kris De Volder for being on my committee. For financial support, acknowledgments and thanks go to Forskningsrådet (The Research Academy) in Denmark, and to Randers Reb. Finally, I want to thank the most important people in my life, my parents, Karen and Finn, and my sister Lisa, for always supporting me, never questioning my decisions to remain in school, for what seems like forever, and for all those airline tickets home for Christmas, and the motorcycle loan.

Jan Bækgaard Pedersen

The University of British Columbia
July 2003

Chapter 1

Introduction

"If debugging is the art of removing bugs,
then programming must be the art of inserting them."

– *Unknown*

One of the most interesting and fastest growing fields in parallel and distributed computing is the field of grid computing—to provide on-demand computing. Tom Hawks, grid computing general manager for IBM, states that businesses can improve utilization of their technology infrastructures by 30 percent or more by taking advantage of Grid technologies to enable on-demand computing [IBM02].

Hawk identifies five industries where Grids are likely to have the biggest near-term impact: Financial Services, which can harness Grids for derivative analysis, statistical analysis and portfolio risk analysis; Life Sciences, for cancer research, new drug discovery, protein folding and protein sequencing; Energy, for seismic analysis and reservoir analysis; Manufacturing, for mechanical design, process simulation, finite element analysis, and failure analysis; and entertainment, for digital rendering. In addition, Hawk adds: "Each of these industries, while different, shares similar business challenges that can be addressed by the unique benefits that Grids can deliver, including on-demand computing, business transformation, data sharing and IT optimization."

Linking computers through Grids to aggregate their power promises to deliver the immense processing capabilities of supercomputers to new venues. For instance, a financial institution could use Grid computing to offer higher levels of service to their best customers for risk management or portfolio analysis. A pharmaceutical company could amalgamate the power of several supercomputers and make the data available to researchers, who access the Grid to collaborate in the development of new drugs. The use of Grid computing as an on-demand utility promises to deliver computing power on a pay-as-you-go basis, as accessible as electricity.

A similar system, the NetSolve client-server system [Net] enables users to solve complex scientific problems remotely. This system allows users to access both hardware and software computational resources distributed across a network. NetSolve searches for computational resources on a network, chooses the best one available, solves the problem using retry for fault tolerance, and returns the answers to the user.

If we turn our attention to more tightly coupled systems, such as large clusters, yet another set of problems become tractable. In 1991, the US Congress passed the High Performance Computing Act of 1991 (Public Law 102-194), which authorized The Federal High Performance Computing and Communications (HPCC) Program. One class of problems developed in conjunction with the HPCC Program was dubbed 'Grand Challenge Problems' by Dr. Ken Wilson of Cornell University, a physicist and Nobel laureate. Since then, various committees and government agencies have added others to the original list. These problems are characterized as 'fundamental problems in science and engineering that have broad economic and/or scientific impact and whose solution can be advanced by applying high performance computing techniques and resources'. They address issues of great societal impact, such as biomedicine, the environment, economic competitiveness, and national security.

One of the Grand Challenge problems includes weather prediction, a task that relies heavily on the availability of computing power; tomorrow's weather report must be available before tomorrow arrives. To obtain such computing power many systems make use of large clusters tightly coupled by a fast network. If we look at the Top 500 list of the fastest computers in the world [Top], an interesting trend appears: many of the fastest machines in the world today are indeed clusters made up of (fairly) ordinary PCs connected by a fast network.

Whether dealing with a system of loosely coupled home PCs or a tightly coupled cluster of high performance PCs in a research lab, the same important question arises: how do we program these large parallel systems, since they do not share any memory, and information must migrate from one process to another through the network? The most widely used technique for exchanging data on such systems is message passing. Message passing involves a sender explicitly sending a message to a receiver containing the data to be exchanged. Some of the most used message passing systems include PVM (Parallel Virtual Machine) [Gei94] and MPI (Message Passing Interface) [Don94]. Many of the fastest computers in the world today use some dialect of MPI, combined with Fortran or C.

1.1 The Debugging Problem

The need for debugging is present in any software development project. Programs have errors and bugs, and these need to be located and corrected. Many approaches have been suggested in the literature, but in practice, the following two approaches are most used. The first is the

traditional and well known debugging-by-print-statements. This approach involves inserting print statements, which display information to the screen or write to a file, into the program being debugged.

This approach is still widely used. In [Pan93c], it is estimated that up to 90% of programmers still rely exclusively on print statements in their code as the only means of debugging.

Logging, or tracing is a more pragmatic approach; instead of printing to the screen, log files are created. The programming language Java has classes for creating such log files or traces. The tracing can be switched on for debugging purposes, and when debugging has finished, can be turned off; thus the programmer does not have to go through the source code to delete the print statements. The same effect can be achieved with C/C++ by using `#ifdef ... #endif` constructions, and setting flags on the compiler command line.

The second approach uses debugging tools. Traditional sequential debugging tools are designed to easily and efficiently provide needed information to debug a sequential program: they provides key views into the different components of such programs and allows the programmer to alter the state of the running program. Examples of such views are stack traces, variable values, and break points. One of the key features of any sequential debugger is the ability to view and alter variable values. That is, the state of the program; the variables of a sequential program can be considered the key players—the core—of the program. Much time is spent verifying whether variables have the correct values, and that the correct branches of `if` and `switch` statements are taken based on expressions that contain variables among others.

Research such as [Eis97] has shown that one important reason errors are hard to find is because the cause and effect are often separated by great distance in the code; the errors are most often found through print statements or similar debugging techniques, and the root cause often turns out to be a memory problem, that is, pointer errors or other forms of memory corruption. This research is based on sequential programs, but as we discuss later, the cause/effect chasm widens significantly when introducing message passing.

A number of tools for parallel debugging have been proposed throughout the years, but in general, these tools are not widely used [Pan93b, Pan93a, Pan93c]. Some complain that the tools are hard and tedious to use, and fail to provide the information users really need and want. One major problem is the information overload that many of these tools suffer from [Pan99]. Too much information is presented to the programmer, thus making the debugging task difficult. Often, this information overload is caused by the tool trying to give the user a global view of the program. That is, taking a top-down approach to debugging without providing the correct views and filtering functions for the user to easily find the information necessary to complete the debugging task.

All the problems and issues mentioned so far arise in the sequential programming domain.

Once parallelism is introduced, not only do these problems become even more prevalent, but new problems are introduced. Not only does the programmer have to deal with asynchronously executing processes on multiple machines, but the message passing increases the distance between the cause of an error and its effect, that is, the location where the error is exhibited. Incorrect variable values may be communicated between processes, making an otherwise correct process behave incorrectly. This increase is not only in the distance in the code as experienced in the sequential domain with, for example, function or procedure calls. When multiple processes execute simultaneously, the cause of an error can originate in one process but the effect manifests in another, thus increasing the distance in the code. Another aspect is time; when data values propagate from one process to another, the amount of time that passes from the cause to the effect of an error also increases.

One example of the cause/effect chasm in time is deadlock. Consider a number of processes in a ring topology, each receiving from its left neighbour and sending to its right. If one of these processes sends its message to a wrong process, eventually all the processes block in receive calls, and the system deadlocks. Not only is the cause and the effect spread far apart in the code, but the time that elapses from the wrong send to the deadlock might be significant.

In summary, the amount of information available to the programmer when debugging n concurrently running processes is magnified, thus making the debugging task even more daunting.

1.2 Problem Definition

In this dissertation we show the following:

- By decomposing the debugging task of parallel message passing programs according to a number of different levels, each specifically concerned with one type of error (sequential, message content, protocol etc.), it is possible to provide tools specifically tailored to finding these types of errors. At the same time, the amount of useless information given to the user is reduced. We refer to this technique as Multilevel Debugging.
- By extracting information about the messages and the protocol from the parallel program, we can focus on specific debugging problems. This has led to a number of new techniques, analyses and algorithms, that can assist the programmer greatly. Many of these can be incorporated into a multilevel debugging tool, such as Millipede.

We have implemented Millipede, a prototype multilevel debugger, that implements the algorithms and analyses described in this thesis. In addition, a multilevel debugging strategy provided in Millipede is closely coupled with the development cycle of a parallel program, whether it is written from scratch or adapted from an existing sequential program.

Millipede follows the model 'You must crawl before you can walk', which in the parallel programming world translates into 'You must fix your pointer errors before you can pass messages'. In general, Millipede considers the various parts of a parallel message passing program as separate levels of the program. Each level has a specific type of information needed to find errors and correct them. The information useful to the programmer to locate and correct errors in the sequential code might not be the most useful for tracking down errors in the communication patterns of the overall program. The three major levels that we propose, to which the multi-level debugging technique should adhere, arise from our definition of a parallel message passing program: A parallel message passing program consists of a number of *sequential processes* bound together by *messages* according to a *protocol*. Thus the three levels are these:

- Sequential code – the code executed by each process. This can be different pieces of code (e.g., a data driven pipeline computation where data passes from process to process) or a number of instances of the same piece of code (e.g., an SPMD program—Single Program Multiple Data, a number of processors executing the same program—such as a processor farm, for example).
- Messages – the individual messages passing from one sender to one receiver. A typical message passing program consists of a number of such messages; each message originates at a send event (or in some cases a multicast or broadcast event) and ends up at a receive event in another (possibly the same) process.
- Protocols – The collection of messages and the pattern in which they are exchanged makes up the protocol of the program.

Each level can be decomposed into a number of sublevels. As we illustrate, the protocol level contains several interesting analyses and algorithms that all deal with the protocol level of a parallel message passing program.

The levels can be viewed as a broadened view of the program, as we 'climb up' the levels. The main focus of the sequential level is the sequential code executed within one process. The message level is concerned with the communication between two processes, while the protocol level focuses on the entire program, and the overall pattern according to which messages are exchanged. While the levels may overlap, the type of information needed at each level expands to include more and more processes as the view expands.

In addition to correcting errors after they have occurred and potentially crashed the program, the added information that can be collected about the protocol and the message passing patterns can also be used for preemptive debugging purposes. An example of such extra information is the protocol and message passing pattern, which can be used to predict deadlocks due to insufficient buffers.

The multilevel debugging strategy is a bottom-up approach; that is, the programmer uses tools specifically tailored to finding sequential errors in the straight line code of his program before moving on fixing message content errors and protocol errors.

1.3 Thesis Statement

The points from the previous sections can be summarized in the following four points:

1. A decomposition of the parallel programming domain into three levels (sequential, messages, and protocol) leads to the multilevel methodology for debugging parallel message passing programs. This methodology also provides a guide and framework for developing and integrating new tools into the debugging system.
2. Extracting the information present in parallel message passing programs, such as message content and protocol information, facilitates the design of tools tailored to specific error types at different levels of the parallel programming domain, which further allows automation of a number of analyses that can not easily be performed otherwise.
3. Such tools map errors back to the source code, and sometimes suggest how to correct them.
4. Such tools can be implemented with a simple command-line interface and require minimal configuration. In addition, no translation or rewriting of the source code is necessary, which makes them directly applicable to the source code.

1.4 Contributions

The literature suggests that tools are not widely used in the parallel programming community. Reasons for this include wrong abstraction, complicated interfaces, and lack of focus on the problem at hand. This means that many programmers still rely on print statements as their only debugging tool. After reviewing many of the faults and shortcomings of tools in general and more specifically tools for debugging parallel message programs we formulate a bottom-up debugging strategy, referred to as multilevel debugging. This strategy not only offers a methodology for debugging parallel message passing programs, but also serves as a design and implementation framework for new tools.

Using details about the three level decomposition of the parallel domain (sequential, messages, and protocol) and error types, we derive a number of general design goals for tools. Examples of these goals are navigation tools at different levels, views for key players, state displayed on request, and relations computed by the tool. In addition a number of specific goals for debugging tools are derived. Examples include the support for finding and correcting specific

types of errors, applicability to source code with a mapping back to the error in the source, as well as providing tools that do not require any rewriting or transformation of the code.

We present the Millipede Debugging System, a prototype of a multilevel debugger for parallel message passing systems, and show how such a system can be implemented for message passing systems such as PVM and MPI. With the basic framework in place, we implement a number of specific tools, all tailored in accordance to our design goals. These tools target different levels of the parallel programming domain, and are specifically tailored to an error type at one of the three levels of the domain. We show how a specific error types can be located and mapped back to the source code.

One of the most important criticisms of existing tools is that the wrong granularity or abstraction often makes the tool either useless for a specific debugging task or create information overload. Information overload is an excess amount of information presented to the user at one time, which in turn makes the debugging task daunting and unnecessarily more complicated. The granularity and level of abstraction of each of the tools we present is set in accordance with the specific error type that the tool is tailored for, thus eliminating much of the excess information not related to the specific debugging task.

We start at the lowest level, the sequential level, where we show how a sequential process can be extracted and debugged using existing sequential debugging tools. This allows the programmer to debug the sequential code of one process at a time, and thus not have to worry about a number of processes running at the same time. In addition, it enables the use of existing debuggers like Gdb [Gdb], and other tools such as purify [Pur] and program profilers. By providing the user with the ability to use well known sequential tools on the sequential code of a parallel program, we provide a way of matching the abstraction of the tool to the abstraction of the debugging task. In addition, no rewriting of the code is needed to make use of these sequential tool. We demonstrate the usefulness of this tool by showing how a number of different sequential errors are located and corrected by extracting the process and using Gdb and Purify. We have provided a translation of the sequential level of the parallel programming domain into the sequential domain, thus covering all types of sequential errors by allowing the user to deploy any sequential tool at this level.

At the message level, we present two techniques related to debugging messages and their content. The first, referred to as interactive message debugging, allows the user to inspect and change the value of messages as they are sent or received. This technique, coupled with the sequential debugging module, allows for the debugging of one process from a parallel system while having the ability to inspect and correct the content of the messages, and also allows the user to perform unit testing of each of the processes during the development cycle. This means that separate parts of the system can be tested independently, which allows for hypothesis testing

that includes the message content to take place, without the rest of the parallel system running.

The second technique is a query language referred to as MQL (Millipede Query Language) that allows the user to write queries using a simple database language and a number of relations containing information about the messages. This provides structured access to the messages and their content, and allows for the computation of relations. Access to messages is considered important as these are 'key players' at this level. In addition, these relations contain information that map the content of the messages back to the source code, that is, lines where data was packed/unpacked and messages sent and received. We demonstrate how to write simple queries to compute a number of useful relations.

At the last level, the protocol level, we implement three different tools. The first tool is an implementation of an algorithm for correcting deadlocks in message passing systems. We show that if a deadlock is caused by a small number of typographical errors in the send or receive calls, the presented algorithm can, with high probability, suggest the correct way of removing the deadlock. We formally argue for the validity and the accuracy of the algorithm by proving an upper bound for the error rate. By focusing on the 'deadlock error type' we have raised the level of abstraction such that an automatic analysis can be performed, and in addition, the algorithm will provide a way to correct the program to remove the deadlock. Again, information mapping the error back to the source is provided, as well as information about which lines to change and what changes to make.

The second tool at this level is a protocol conformance checking tool. We introduce a tool that allows the programmer to specify a number of constraints on a communication protocol of a parallel system in as much detail as he wants. The tool is used to specify constraints on the message passing pattern of the communication protocol, as opposed to both the temporal and spatial aspects. The runtime system of Millipede reads the protocol specification and checks each message against the specification, and reports any errors. The constraint specification can start out very general, and become increasingly complex as the program, or as the debugging effort evolves. This tool serves as a debugging tool that can be used in connection with iterative hypothesis testing as well as a tool that can be deployed during the development cycle. One main argument for this tool is the ability to bridge the gap between a protocol specification (verified or not) and an actual implementation; even if a protocol has been verified using verification tools, when implementing it, the risk of making mistakes still prevails. This tool provides a way to subject an implementation of a protocol to a number of constraints that are automatically checked by the runtime system.

The motivation for the last tool at this level is the problem of guaranteeing k -safety, (i.e., the problem of determining the buffer requirements for message passing systems). We investigated k -safety for four buffering schemes, and in all cases showed that the problem remained intractable.

We showed that the related problem of computing the number of buffers needed to avoid deadlock and blocking sends is tractable. This algorithm provides an upper bound that can be used in combination with techniques for inserting synchronization points into the code to ensure k -safety for any k .

We show a number of results for different buffer placement schemes (sender side buffers, receiver side buffers, etc.), some of which are proven intractable, and some are proven tractable. We develop an algorithm for computing the number of buffers needed to avoid deadlock and blocking in a system with receive side buffers only. The original k -safety problem for all buffer schemes is intractable. Using the algorithm we developed to avoid blocking, we describe techniques to approximate solutions to the k -safety problem.

The decomposition of the parallel programming domain into three levels and the study of error types induces a multilevel debugging strategy; a bottom-up approach where the error type determines what type of tool should be applied. Furthermore, it states that errors at lower levels should be attended to before turning to errors at higher levels. That is, pointer errors and array-out-of-bound errors should be corrected before turning to debugging the protocol. In addition, this decomposition, based on levels, and error types provides a framework for developing new tools, which again drives the debugging process.

We showed that it is possible to develop tools according to the multilevel debugging methodology. As described in the following chapters we have provided a number of such tools at all three levels. The tools all have simple user interfaces, and no rewriting of the source code is necessary to deploy the tool. In addition each tool maps the errors in question directly back to the source code, and in some cases suggest how to remove the error.

The work in Chapters 4, 6, 7 and 8 has been published as conference papers in the parallel programming community (the sequential debugging module, presented in Chapter 4, has been published in [BW00]). The deadlock detection and correction work, described in Chapter 6, appears in [BW01a]. The protocol checking tool has been published in [BW01b] and finally, parts of the joint work with Alex Brodsky on the buffer allocation problems, found in Chapter 8, has been published in [BBW02].

Chapter 2

Background and Related Work

“The power of accurate observation is commonly called cynicism
by those who have not got it.”

— *George Bernard Shaw*

“If we begin with certainties, we will end in doubt.
But if we begin with doubts and bear them patiently,
we may end in certainty.”

— *Francis Bacon*

2.1 Background and Rationale

Debugging sequential programs can be a tedious and time consuming task. The time required can be greatly reduced by using some of the many different tools developed for this task. Some of the more well known debugging tools include Gdb [Gdb] and purify [Pur], and various integrated development environments accompanying programming languages. Unfortunately, these tools are not as readily available in the parallel programming domain.

To better understand the lack of tools and the limited use of existing tools, the next section briefly introduces some of the problems encountered when working in the parallel programming domain and some of the observations made about debugging in general.

2.1.1 The Parallel Programming Domain

Parallel programming involves a set of components that must each be considered when developing a parallel system. This set, which we regard as the parallel programming domain, includes, among others, the following aspects of the code: sequential code, interprocess communication, synchronization, and processor utilization. Understanding the issues involved with the components of this domain makes understanding the source and manifestation of errors easier. This understanding is useful for determining the approach needed to efficiently debug parallel programs. In addition, it helps determine where to focus the debugging effort, depending on which component of the domain the programmer looks for errors in.

In [Fos95] a four stage model for constructing a parallel program, referred to as PCAM, representing the parallel programming domain is suggested. The four components are:

1. **Partitioning.** The computation to be performed and the data which it operates on are decomposed into small tasks.
2. **Communication.** The communication required to coordinate task execution is determined, and the appropriate communication structures and algorithms are defined.
3. **Agglomeration.** The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs.
4. **Mapping.** Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs.

The two last components, agglomeration and mapping, are mostly concerned with performance issues which, while important, are outside the scope of this dissertation.

For the first two components, partitioning and communication, we propose the following breakdown:

1. **Algorithmic changes.** Most parallel programs begin life as a sequential program. If parallel algorithms are based on, or derived from, existing algorithms and/or programs, then a transformation from the sequential to the parallel domain must occur. The transformation of a sequential program into a parallel program typically consists of inserting message passing calls into the code and changing the existing data layout; for example, shrinking the size of arrays as data is distributed over a number of processes. However, if the sequential algorithm is not suitable for parallel implementation, a new algorithm must be developed. For example, for the pipe-and-roll matrix multiplication algorithm [FJL⁺88] does not have a sequential counterpart.

2. **Data decomposition.** When a program is re-implemented, the data is distributed according to the algorithm being implemented. Whether it is the transformation of a sequential program or an implementation of a parallel algorithm from scratch, data decomposition is a nontrivial task that cannot be ignored when writing parallel programs, as not only correctness, but efficiency also greatly depends on it.
3. **Data exchange.** As parallel programs consist of a number of concurrently executing processes, the need to exchange data inevitably arises. This problem does not exist in the sequential world of programming where all the data is available in the process running the sequential program. However, in parallel programs, the need for data exchange is present. On a shared memory machine, the data can be read directly from memory by any process. There is still the problem of synchronized access to shared data to consider, but no sending and receiving of data is needed. When working with a cluster of processors, each having a separate memory, message passing becomes necessary.

When message passing systems like PVM and MPI are used, the programmer is responsible for a number of different tasks: specifying the correct IDs of the involved processes, packing messages into buffers, using the correct functions to pack the data depending on the type, and assigning tags to the message. In part, the difficulty of using a message passing library like PVM is the low level of the interface of the message passing system. Figure 2.1 shows an example of the minimal number of steps that are needed to perform a send and a receive of 2 integers (stored in variables *a* and *b*), respectively.

```
Sender:
1:  pvm_initsend(PvmDataDefault);
2:  pvm_pkint(&a, 1, 1);
3:  pvm_pkint(&b, 1, 1);
4:  pvm_send(Receiver, 22);

Receiver:
5:  pvm_recv(Sender, 22);
6:  pvm_upkint(&myA, 1, 1);
7:  pvm_upkint(&myB, 1, 1);
```

Figure 2.1: Simple PVM program that exchanges two values using default buffers.

As Figure 2.1 illustrates a number of low level message passing library function calls must be performed to send a message. Line 1 initializes the message passing system to use the default send buffer with default data encoding. Line 2 packs the integer *a* into the send buffer according to the encoding scheme specified in the `pvm_initsend` call. `pvm_pkint` can be used to pack arrays of integers as well; the second parameter is the number of

integers to pack, and the third is the stride. In this example, only one value is packed, thus, the number of values to pack and the stride are 1. Finally, line 4 sends the message to a process with task ID `Receiver` with message tag 22. The message tag is an integer number that allows the message passing system to differentiate message types. A typical use of the message tag is to use different values for different message types; this also allows the receiver to specify a specific type of message in receive calls, thus allowing messages to overtake each other in the message buffer queues at their destination.

The second part of Figure 2.1 is the code necessary to receive the message. Line 5 issues a receive call, which requests a message from a sender with the task ID `Sender` and message tag 22. This receive call is blocking. If no message is available from the correct sender with the correct tag, the receive call simply blocks until such a message arrives. The sender task ID and the tag can be specified as wild cards that match anything, but for reasons, such as the ability to read and understand a program, wildcards should be used with care. More importantly, the quality of some of the analyses we introduce later will increase when the use of wild cards is reduced.

There exist nonblocking, or timer controlled, versions of the receive call, but these add further complications to the code. Once the message has been passed to the process by the underlying message passing system, it must be unpacked into the destination variables. The `pvm_upkint` is the exact opposite of the packing call, it unpacks values into variables. Again, the number of values and the stride can be specified. The low level nature of a message passing library, such as PVM or MPI (MPI does support packing complex data structures in one call, but sender, receiver, tags, and variables must still be specified), increases the risk of introducing errors: a wrong sender or receiver may be specified, wrong variables are packed, or values are packed or unpacked in the wrong order.

These are problems that can occur when the default buffers are used. If the programmer allocates buffers explicitly, instead of using the default buffer, the issues of buffer management arise. One single send buffer is not always sufficient, the prototype debugger that we implemented in connection with this work makes heavy use of allocated buffers. Figure 2.2 shows an example using an allocated buffer.

Only one send buffer can be active at any time during execution. This means that if a process uses more than one send buffer, explicit buffer handling is necessary. When creating a new send buffer, the old one must be saved so it can be restored later. Line 1 creates the new send buffer, line 2 stores the old buffer and activates the new. Line 3 and 4 pack the data stored in variables `a` and `b`, and line 5 sends the message to `Receiver` with tag 22. Line 6 restores the previous send buffer and finally, line 7 deletes the newly allocated send buffer. The receiver in Figure 2.2 is identical to the one in Figure 2.1. The `pvm_recv` call returns

```

Sender:
1:  newSendBuffer = pvm_mkbuf(PvmDataDefault);
2:  oldSendBuffer = pvm_setsbuf(newSendBuffer);
3:  pvm_pkint(&a, 1, 1);
4:  pvm_pkint(&b, 1, 1);
5:  pvm_send(Receiver, 22);
6:  pvm_setsbuf(oldSendBuffer);
7:  pvm_freebuf(newSendBuffer);

Receiver:
8:  pvm_recv(Sender, 22);
9:  pvm_upkint(&myA, 1, 1);
10: pvm_upkint(&myB, 1, 1);

```

Figure 2.2: PVM code for sending and receiving two values using explicitly allocated buffers.

the identifier of the new active receive buffer; this is a buffer created by the underlying system. If the receiving process works with multiple buffers simultaneously, `pvm_setrbuf` can be used in a manner similar to `pvm_setsbuf`. The added complexity of managing send or receive buffers naturally increases the risk of introducing errors, and further complicates the use of the message passing libraries. Data exchange is concerned with the point to point communication of data between two processes, and not the overall communication structure of the entire program. Thus, for every data exchange, there is one send operation and at least one receive (if broadcast or multicast is used there can be multiple receivers).

4. **Protocol specification.** The protocol for a parallel system is defined as the content, order, and overall structure of the message passing between communicating processes. Along with the data exchange, the communication protocol of the program is a new concept that has been introduced by parallelizing the algorithm.

Figure 2.3 shows a stylized representation of a sequential and a parallel program. As shown, a sequential program is depicted as a single box, representing the sequential code of the program. The parallel program is represented as a number of boxes, each consisting of three nested boxes. The innermost of these boxes represents the sequential program that each process in the parallel program executes. The sequential code of the parallel program can either be an adaption of the existing sequential program, or a completely rewritten piece of code. The middle box represents the messages being sent and received in the system (the data exchange), and the outer box represents the protocol that the communicating processes must adhere to.

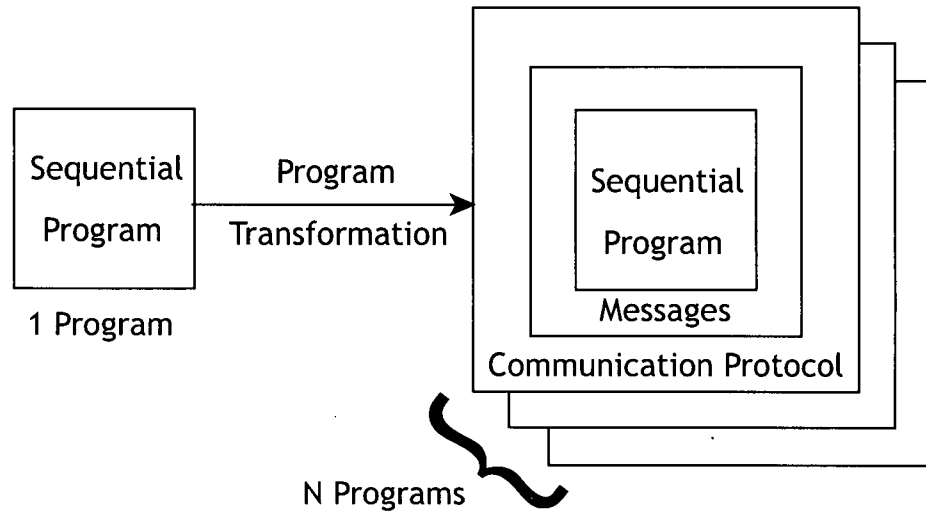


Figure 2.3: In the sequential programming domain we work with one (sequential) program, whereas in the parallel domain we encounter a number of parallel processes making up a parallel program. These parallel processes each execute a sequential program, but in addition, send messages to other processes while adhering to a communication protocol. The messages and the protocol are represented as boxes enclosing the sequential program as new levels.

2.2 The Debugging Process

In this section we introduce the debugging problem/process, briefly present ideas about how to debug in general, describe the problems with current approaches, look at the purpose of our research, and explain how it differs from existing systems.

2.2.1 Iterative debugging

A well known approach to debugging was proposed by Araki, Furukawa and Cheng [AFC91]. They describe debugging as an iterative process of developing hypotheses and verifying or refuting them. They proposed the following four step process:

1. **Initial hypothesis set.** The programmer creates a hypothesis about the errors in the program, including the locations in the program where errors may occur, as well as a hypothesis about the cause, behaviour, and modifications needed to correct them.
2. **Hypothesis set modification.** As the debugging task progresses, the hypothesis changes through the generation of new hypotheses, refinement, and the authentication of existing ones.

3. **Hypothesis selection.** Hypotheses are selected according to certain strategies, such as narrowing the search space and the significance of the error.
4. **Hypothesis verification.** The hypothesis is verified or discarded using one or more of the four different techniques: static analysis; dynamic analysis (executing the program); semi-dynamic analysis (hand simulation and symbolic execution) and program modification.

If the errors have not been fixed after step four, the process is repeated from step two. In the above model, step four, hypothesis verification, is the focus of our research.

2.2.2 The Why, How and What of Errors

M. Eisenstadt describes in [Eis97] a 3-dimensional space in which sequential errors are placed according to certain criteria. This classification shows some interesting results, which we briefly summarize. 51 programmers were asked to participate in a study in which programming errors are placed into a 3-dimensional space. The 3 dimensions are:

- **Dimension 1:** Why is the error difficult to find?
- **Dimension 2:** How is the error found?
- **Dimension 3:** What is the root cause of the error?

We briefly describe the results of the survey, with respect to each of the dimensions.

Dimension 1: Why is an error hard to find?

This first dimension is concerned with the difficulty of locating the problem, and is further divided into 5 subcategories:

1. **Cause/effect chasm.** Often the symptom of the error is far removed in space and time from the root cause, and this makes the cause hard to detect. Specific instances can involve timing or synchronization problems, bugs which are intermittent, inconsistent, or infrequent, and bugs which materialize 'far away' (e.g., thousands of iterations) from the actual place they were spawned.
2. **Tools inapplicable or hampered.** These are the so called 'Heisen bugs' [Gra86]. This covers bugs that go away when switching on the debugging tool. Another type of bug in this category are referred to as 'context precludes', and covers the cases where memory constraints or other configuration issues make it impractical or impossible to use the debugging tool.

3. **WYSIPIG (What You See Is Probably Illusory, Guv'nor).** A piece of code is misconceived; it does not give the result that it looks like it should produce. An example could be the number 010 which in Tcl does not equal the decimal value 10 (ten); but the octal value 8 (eight). The preceding 0 makes the Tcl interpreter treat the value as octal.
4. **Faulty assumption/model.** The programmer does not understand the underlying system, model or the environment. An example is assuming the stack grows up rather than down.
5. **Spaghetti (unstructured) code.** The code is hard to read. This is typically reported as a reason when programmers work with code they did not write themselves.

Table 2.1 shows how the 51 answers are placed in the above categories for dimension 1.

Category	No. of answers	Percentage
Cause/effect chasm	15	29.4%
Tools inapplicable or hampered	13	23.5%
WYSIPIG	7	13.7%
Faulty assumption/model	6	11.8%
Spaghetti code	3	5.9%
No answer	8	15.7%

Table 2.1: Dimension 1. Why is an error hard to find?

It is notable that over 50% of the cases are caused by the two first categories. As we see later on, the first category, the cause/effect chasm is greatly amplified in the parallel programming domain, and the second category is, as we have already pointed out, one of the problems we are researching.

Dimension 2: How is an error found?

This dimension is concerned with how an error can be found, and it is divided into four categories:

1. **Gathering data.** The programmer discovers more using methods such as print statements and breakpoints. This category includes a number of subcategories:
 - Step-and-study, which includes single stepping through the code using a debugger.
 - Wrap-and-profile, where profiling information is collected by enclosing the suspect function inside another function that does the information collecting.
 - Print-and-peruse, the most well known of the sub categories, involves inserting print statements and observing the output.
 - Dump-and-diff involves comparing different versions of large amounts of information gathered (e.g., a true core dump).

- Conditional break and inspect, which includes the use of breakpoints.
 - Specialist profile tool, which includes using standard tools, such as *purify* to locate memory leaks.
2. **'Inspection'.** This term covers inspection of the code, hand simulation, and speculation. Speculation involves leaving the code to think about the problem, then later returning to try to correct it.
 3. **Expert recognized cliché.** The programmer receives assistance from other people.
 4. **Controlled experiments.** Once the cause of the error is better understood, specialized tools or approaches can be applied.

The placement of the answers in dimension 2 can be seen in Table 2.2.

Category	No. of answers	Percentage
Gather data	27	53%
'Inspection'	13	25.5%
Expert recognized cliché	5	9.8%
Controlled experiments	4	7.8%
No answer	2	3.9%

Table 2.2: Dimension 2. How is an error found?

An interesting, but not surprising, result is that data gathering (e.g., print statements) and hand simulation account for almost 78% of the techniques reported in locating errors. This result corroborates the result of Pancake [Pan94]: up to 90% of all debugging is done using print statements.

While the use of print statements is straightforward when working with sequential programs, their use in parallel programs is often more complicated. Often, processes run on remote processors, which makes redirecting output to the console difficult. Even when output can be redirected to the console, all processes are writing to the same window, thus making the interpretation of the output a challenging task. This is an example of the information overload theory mentioned earlier. Furthermore, the order of the output (i.e., the debugging information from the concurrently executing processes) is not the same for every run, as the processes execute asynchronously and only synchronize through message passing. A possible solution is to have each process write its output to a disk file. However, this introduces the problem of nonflushed file buffers; if a process crashes, the buffer might not be flushed, thus missing output written by the program. Of course this can be solved by inserting calls to flush the I/O buffers, but if these are missing, the programmer ends up spending time on debugging the code he added for debugging purposes!

In the worst case this can lead the programmer to believe that the process crashed somewhere between the last print statement that appears in the file, and the first one that does not. A lot of time can then be wasted looking for an error in a place where no error can be found.

Dimension 3: What is the root cause of the error?

This last dimension contains nine categories:

1. **Memory:** Memory is 'clobbered' or used up. This includes overwriting a reserved portion of the memory causing the system to crash, and array subscripts out of bounds.
2. **Vendor:** Compilers generate wrong code or the hardware is faulty (logic boards do not adhere to specifications or are broken).
3. **Design logic:** The logic design of the algorithm is wrong. Examples include cases forgotten or overlooked by the programmer.
4. **Initialization:** Covers wrong types, redefinition of the meaning of system keywords, or incorrectly initialization of a variable.
5. **Variables:** Wrong use of operators or variables.
6. **Lexical:** Lexical problem, bad parse or ambiguous syntax. These are trivial problems such as typographical errors.
7. **Unsolved:** As yet undetermined.
8. **Language:** Language, semantic ambiguities or misunderstandings. For example, 250K is not 250,000, but rather 256,000 ($250 \times 1,024$).
9. **Behaviour:** Unanticipated behaviour by the user that makes the program behave in a unanticipated way.

Table 2.3 shows that nearly 50% of the errors are caused by the first two categories. This also perfectly agrees with previous studies where tools and runtime systems are described as a source of errors [Pan94]. The classification used in dimension 3 is a mixture of deep plan analysis [Joh83, SSP85] and phenomenological analysis [Knu89]. Deep plan analysis states that many bugs can be accounted for by analyzing the high level abstract plans underlying specific programs, and by specifying both the possible fates that a plan component may undergo (i.e., missing or misplaced). An alternative phenomenological taxonomy can be found in [Knu89] where the root causes are divided into nine categories, all very similar to the ones in Table 2.3.

Category	No. of answers	Percentage
Memory	13	25.5%
Vendor	9	17.7%
Design logic	7	13.7%
Initialization	6	11.8%
Variables	4	7.8%
Lexical	3	5.9%
Unsolved	3	5.9%
Language	2	3.9%
Behaviour	2	3.9%
No answer	2	3.9%

Table 2.3: Dimension 3. What is the root cause of the error?

2.3 Related Work

In this section we describe some of the existing approaches to parallel debugging and parallel system development. We try to point out any shortcomings these tools might have, and compare them with the theory of errors and debugging presented earlier.

2.3.1 Program Development Environments

One approach to writing programs is to use an integrated development environment (IDE). Some well known examples in the sequential world include Visual C/C++, Visual Basic from Microsoft, and the Eclipse and the NetBeans IDEs for Java. Not only do these environments offer support for program development, but they come with built-in debuggers. The idea of developing programs and complicated systems through a development environment also extends to the parallel programming domain.

One of the most important tasks of a program development tool is to allow the user to develop programs in a structured way using some high-level abstraction, such as graphs. An important side effect of the structure and high-level abstraction is a lowered risk of introducing certain error types. For instance, certain tools, such as the PVMbuilder tool [BB97], always create deadlock free message passing code by ensuring that all send calls are matched with receive calls and that the corresponding communication graph does not have cycles. This abstraction allows the user to concentrate on higher levels of the program design, for example, function or control, or data decomposition of the program, depending on which abstraction is adopted by the environment. However, this high level of abstraction restricts the user in which types of programs he can develop using PVMbuilder. Programs with dynamic communication cannot be implemented.

Often, the structure and abstraction level offers relatively easy debugging of certain types

of errors within the environment, which of course, is a very desirable quality from a debugging point of view. Unfortunately, the concepts that make development environments desirable also have their disadvantages. An environment structured around a high level of abstraction is a good tool for program development, only if the abstraction of the task at hand matches that of the environment. For example, if the environment is structured around the data flow model and the program being implemented is structured according to the control flow model, then implementing such an algorithm becomes complicated and cumbersome. A concrete example of this problem emerges when trying to write a program that makes use of explicit message passing using a tool that supports the data flow model. That is, entire blocks of data flow between functional units in the program representation of the tool. Even if it is possible to implement the program, it will be conceptually difficult and artificially structured. The problem with the structure and the abstraction of the tool not fitting that of the program being implemented is one of the most common reasons for not using such tools [Pan94].

Another reason is that users are often conservative and hesitant to learn new environments [Pan94]. The nature of the generated code can contribute to a programmer's hesitation. If the tool supports a source to source transformation, for example, from the tool abstraction to C source code, this code can be hard to read or illogically structured because of the automated code generation. Even worse, sometimes no source code is available, and that limits a programmer's ability to apply other tools for further development and maintenance. This problem is apparent with the PVMbuilder tool. Since the tool generates all the communication code when the user compiles the application, this code is often extremely hard to read. Debugging such code, or code produced by rewriting tools, is a daunting task. In the worst case, the generated code is virtually unreadable. Furthermore, data structures and functions not implemented by the user might be used by the generated code, adding yet another level of complication to the debugging task.

Some examples of environments specific to developing parallel applications are described in the following. Examples of environments that have adopted the data flow model as a main abstraction are Code [NC92, NY93] and HeNCE [Don]. The abstraction is based on data flowing between functional units or entire processes of the system. Trapper [Hei97] is a CSP based tool—a Trapper procedure contains channel communication calls to read and write to the channels. The underlying implementation is hidden, however, there is still a need for the programmer to directly specify what to send and receive. It is the graphical user interface's task to link channels together, thus making sure there are no disconnected channel ends. Trapper is comparable to programming libraries like JCSP [WAF02] without a graphical user interface. However, such interfaces are currently being developed for JCSP

Like Code, HeNCE, and Trapper, graphs are also used to represent programs by PVMbuilder

and VPE [ND94]. However, these two tools both allow explicit message passing. In these cases the abstraction adopted leans more towards the control flow model than the data flow model.

Tools like Enterprise [SSS90] and Frameworks [SSG91] take a template-based approach to generating distributed applications. Programs are written as sequential procedures enclosed in templates. The templates hide all the distributed computing implementation details, such as communication and synchronization. The procedures themselves contain only a small amount of information as to how they interact with the rest of the system. Most of it is specified separately via templates.

All of the tools strive to make parallel programming easier, that is, to reduce the number of errors, and take away much of the work with respect to explicit message passing from the user. Unfortunately, when this is the goal, the user's freedom and expressiveness is reduced; the safer the development tool is required to be the more restrictive it becomes, and the more limited the expressiveness becomes. In other words, the higher the level of abstraction and the more rigid the structure of a tool, the smaller the set of easily implementable programs. The greater the set of programs the user wishes to implement, the more general the environment must be. At one end of the spectrum, tools are specifically designed for a certain type of program with a very rigid structure. At the other end, 100% manually coded programs exist where the programmer himself supplies message passing calls using, for example, PVM or MPI. Many of the tools described fall in between these two extremes. However, the problem of picking the correct tool or trying to utilize familiar tools to solve the problem at hand remains. Picking the right tool might involve having to learn a new abstraction and a new tool.

2.3.2 Visualization Tools

One class of tools that can be used not only for performance tuning, but also for debugging is the family of visualization tools. Visualization tools are categorized by their ability to provide the user with information about a program's behaviour.

A typical tool offers a fixed set of views, each displaying different information about the system in various ways, such as graphs and charts. A visualization tool that supports message passing views can be used when the programmer searches for errors involving stray messages or simply erroneous protocol specifications. Such tools are also excellent if the programmer is trying to obtain a global view of the entire system.

However, often global views are much too vast for a programmer to easily locate errors. These types of tools are faced with the very difficult task of providing a vast amount of information in an easy to understand way. This problem has been addressed by Pancake [Pan99], and some of the more serious issues pointed out include not only the difficulty with presenting large amounts of data, but also the inability to zoom into views, to extract lower level information, and to map

these displays/views back to the source code which created the error.

This is similar to having gathered data as in dimension 2, but the data is not directly understandable, which is caused by the inapplicability of the tool (dimension 1). In addition, there are also problems associated with the amount of data that can be displayed, the type of displays, and at least for tuning, the problem of perturbing the execution of the program. The last problem can appear in any tool based on a software monitoring and runtime collection of information; however, it is more critical in the case of performance tuning and identifying performance problems.

Another problem is the lack of user defined views. Many visualization tools support a limited preprogrammed set of views. We believe that this directly contradicts one of the design goals set by Eisenstadt in [Eis97]: the possibility of a tool that offers a view of what the user wants, when she wants it, not just the information that the programmer of the tool thinks might be useful to the user.

Examples of such tools are Paradyn [MHC94], Vampir [NAW⁺96], and ParaGraph [HE93]. One visualization tool specific to PVM is XPVM [KG96], which uses the tracing facilities available in PVM 3.4, and offers a graphical user interface to dynamically visualize network status, utilization, message queues, and much more. These tools all use graphical representations to display program behaviour.

2.3.3 Extension of Sequential Debuggers

In this section we look at the family of debuggers that are extensions of well known sequential debuggers. We divide this class of tools into two categories: debugging environments and *N*-version sequential tools.

Debugging Environments

A number of debugging environments exist that support parallel debugging. These environments are typically extensions of sequential debuggers. As a result, the set of operations available in these tools are well known because of familiarity with standard sequential debuggers. These include stepping, breakpoints and variable inspection. The biggest difference—and greatest strength—is that these tools operate on several processes at a time, thus allowing collective breakpoints over multiple pieces of source code and macro stepping, which allow several processes to step through one line of program code at the same time. The strength of these tools is their ability to control multiple processes at the same time. This can be a problem for the user; keeping track of a large number of processes simultaneously blurs the focus of the debugging task. Even though these tools support the common set of debugging activities, they all require the user to learn a new environment with its own graphical interface. Furthermore, the focus is on the sequential code, not on the entire parallel system. That is, the granularity cannot be

varied, but is set to 'fine grained'; only the sequential debugging task is supported. In a sense, these parallel debugging environments can be said to suffer from the opposite problem as the visualization tools: the granularity is too fine and the focus is always on the source code. Such tools, therefore, get placed into the 'tool inapplicable or hampered' category of dimension 1.

Examples of such debugging environments include DIWIDE [KLK99] and TotalView [Pal99]. The DIWIDE debugger is a parallel debugger that implements collective breakpoints and macro steps (collectively stepping over program parts). It allows the programmer to treat a collection of processes as one, and allows the user to easily issue global commands and set global breakpoints. TotalView, a commercial product, is a multiprocess, multithreaded tool for online source code debugging.

N-version Debuggers

A naïve approach to parallel debugging involves the use of N copies of a sequential debugger like Gdb—one for each process. The disadvantage of providing N versions of a sequential tool is the overwhelming amount of information. In addition, the way in which this vast amount of information is presented to the user is often inappropriate for the task at hand [Pan99]; it is not as easy for the user to focus on one particular process in the system when attending to all of them. The complexity of the program development process alters drastically when parallelism is introduced, and the problems are heightened by the relative instability of current parallel runtime environments [Pan94]. This suggests that debugging a parallel program while all the processes are running concurrently may be too difficult and a tool tailored more to specific processes in the system is more effective. In addition, the granularity cannot be varied and the user is left with the functionality of a sequential tool, which might not be applicable for a parallel debugging task.

N-version debuggers also lack the ability to supply different views of whatever information the user might want. Although all variables and program texts are available (which can be a great advantage when debugging low level sequential code), this information is spread over N windows and not readily available for queries. It would take an overwhelming amount of time for the programmer to extract, collect and interpret the information available. In addition, if the focus is on one single process, the debugging views are not needed for the rest of the processes.

An example is pdbx [Pdb] for the SP/2, which is a front end for multiple instances of the UNIX debugger dbx running on multiple nodes on an IBM SP system. As well, p2d2 [Hoo96] is a graphical front end for multiple instances of the Gdb sequential debugger, and has been used successfully to debug systems with as many as 128 concurrent processes. It is possible to design a script for PVM to allow users to execute Gdb on every process spawned. However, that would require a script for each sequential tool being supported, and it is more difficult, in PVM, to conditionally

spawn the debugger for a given set of processes, without having to rewrite the code in the original program.

2.3.4 Replay Tools/Debuggers

Another major class of tools is the family of replay tools which allows the user to animate or replay the execution of a program [XWXS96, TSS96, KV97, Arv92, CFR95]. Replay tools collect information about the system as the program executes: messages are collected, time stamped and saved on secondary storage for replay. When the tool replays the execution, information about message content and program state is retrieved from the disk. A replay tool is typically considered an offline tool, deployed when the program has finished executing. Many of these systems have a set granularity, thus focusing on, for example, the source code level, leaving the user helpless if debugging on a higher level is needed. The opposite can of course also be the case: when the focus is on the higher level of the system, mapping the error back to the lower level is difficult because the tool does not readily support debugging at a lower level.

BUSTER [XWXS96] is one such 'post mortem' replay system; it allows the user to reexecute the debugged program in different modes, depending on the amount of control needed, without having to run the message passing system. A system like PVaniM [TSS96] is another PVM based graphical tool that supports both online debugging and post mortem visualization of a parallel execution.

Other examples of tools that combine the online and offline strategies of debugging in a more integrated environment include MAD [KV97], PDT from the Annai toolset [CFR95], and PDM [Arv92]. These tools provide more integrated environments for debugging while providing higher level tools for finding and correcting specific errors, such as communication errors. MAD is a debugging environment based on event graphs and their manipulation. It allows debugging on various levels, from pattern of processes (groups of event graphs), to control flow graphs and source code. PDT is an interactive distributed source-level debugger for distributed memory parallel processes in the Annai toolset, and allows both online debugging and offline replay. The PDM system is a framework for detecting communication-related errors in concurrent Occam programs running on a Transputer network.

The major disadvantage of these tools is the massive amount of information involved and the need to learn a new environment. The replay mechanism is extremely useful. However, unless other tools can be used in connection with a replay, it becomes virtually impossible to accomplish a specific debugging task, unless the tool specifically supports it. Again, if the replay mechanism is merged with some of the techniques described in earlier subsections of this chapter, a stronger and more flexible tool results.

2.3.5 Relative Debuggers

For the sake of completeness we wish to mention the concept of relative debugging. Relative debugging is a technique often used when porting programs to different architectures, thus allowing the execution of two different versions of the same program on two different machines at the same time. Guard [SA97, WA98] is such a debugger. It executes two different instances of the same program on two different machines, thus allowing the programmer to compare the contents of variables and more while the programs execute. Relative debugging is useful for the programmer who ports existing programs to different architectures or operating systems. The technique focuses on comparing two different instances of the same algorithm, where one is known to produce the desired result. Thus, this technique is not directly applicable when it comes to general (parallel) debugging.

2.3.6 Language Support for Communication

A different approach altogether involves writing programs using languages with built-in support for parallelism. A well known example of this is the μ C++ language [BS95] from the University of Waterloo. μ C++ extends C++ with new language constructs to express parallelism and provides a runtime system that runs programs concurrently or in parallel, when appropriate hardware is available. However, the need to debug is still present.

The work in [BK95, Kar95] describes debugging and performance tools for μ C++ in greater detail. A debugging session for a program written in μ C++ compares to those found in DIWIDE and TotalView: a front end to a number of instances of well known sequential debuggers, such as Gdb, attached to each process being debugged.

Other well known examples of languages that support communication include CML [RWZ88] (Concurrent ML), and Facile [GMP89a, GMP89b] (ML with higher order concurrent processes based on CCS), both functional languages. Often such languages are not considered to be platforms for implementing parallel applications. One of the strongest arguments is lack of speed, a well known side effect of functional programming languages. This is an unfortunate tradeoff, as functional programs are more easily verified by program verification tools, thus reducing the need for debugging.

2.3.7 Summary of Related Work

The following points summarize the problems with many of the existing tools:

- Restrictive interfaces that support only a number of predefined tasks.
- The data gathered need to be interpreted by the user to map the error back to the cause, which often renders the tool less useful. In other words, the cause/effect concept is not

well supported. In [KV97], it is argued that the original source code is a good basis for debugging activities, since it contains the cause of the wrong behaviour.

- A fixed, often small grained, number of tasks are supported. Fixed granularity in connection with restrictions on the interface makes debugging at higher levels almost impossible.
- Information overload: the amount of information presented can be so large that time needed to find the information becomes unmanageable.

What the user wants is not always available in any of the reviewed systems. Each of the systems have strong points and can be very useful for certain tasks. Unfortunately, applying different selections of tools from different toolsets is an impossible task; different user interfaces, different representations, different formats and so on make changing between tools for different debugging tasks difficult. This means that the user must choose only one, or at best a small number of tools, which might not be preferred for the debugging task.

2.4 Top-down versus bottom-up debugging

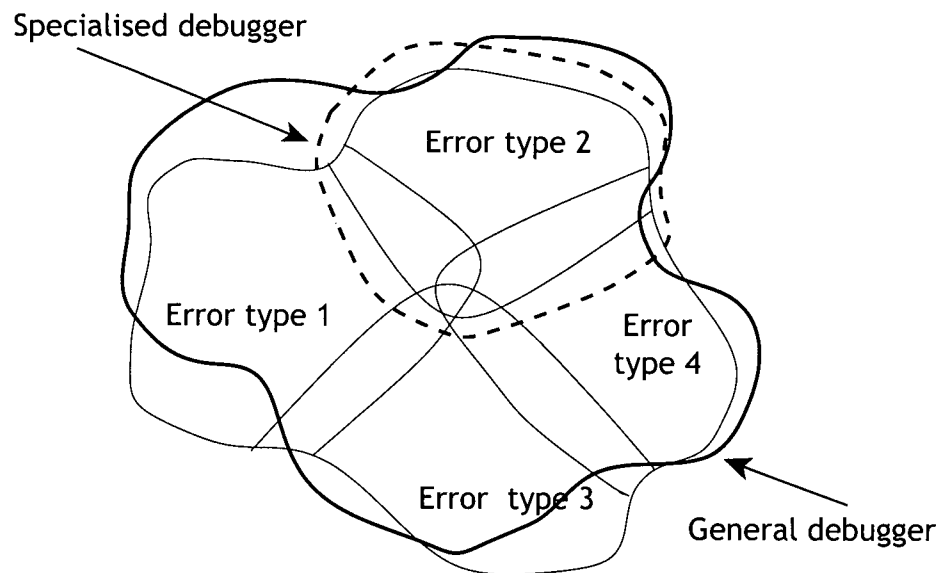


Figure 2.4: The top-down debugging strategy versus the bottom-up. Thin solid lines represent various error types. The thick solid line represents a typical top-down debugging tool that presents all the information available to the user. The dotted thick line illustrates a bottom-up approach—the tool is specifically designed to finding and correcting errors of type 2.

Many debugging tools and environments offer a global view of the entire program and leave it to the programmer to narrow the search space, including specializing the formation and testing of hypotheses, and searching for errors. This approach poses one of the greatest problems with existing tools and environments. The set of visualization tools and environments *do* support a global problem identification and hypothesis-making process, however, they do not readily support the process of localizing the error and mapping it back to the source code. We believe that this is due to the information overload theory presented earlier and supported in [Pan99]. We refer to this method of debugging using a global view tool as a top-down approach to debugging.

In Figure 2.4 the inner shape, containing various error types, represents the potential errors in a program. The figure has been divided into a number of parts, one for each type of error that can occur. The outer bold shape represents the typical method of debugging when using debugging tools: the top-down approach.

A different approach is obtained by turning this well known method upside down. Instead of providing a global view of a program and allowing the user to look for any kind of error using just one tool, we propose that a bottom-up approach be adopted.

Assume the user has made some hypothesis about the type of error, typically based on the report obtained when the error occurred. We propose the application of a tool specifically tailored to supporting hypothesis creation, verification and error search of the specific type of error, using the extra information that can be gathered from a parallel application. As mentioned, this includes information about messages, their content as well as protocol information, such as message exchange patterns.

2.5 Multilevel Debugging

The purpose of this research is to examine a bottom-up approach, which we refer to as 'multilevel debugging', over the more conventional top-down approach described earlier. The focus is closely tied to the major points of the multidimensional analysis described earlier and the description illustrated in Figure 2.3. In addition, we strive to develop tools and techniques that make use of the extra debugging information that can be extracted from the parallel program, and we show that new useful analyses can be done based on this information. To succeed in this task we believe that the following three points must be understood and shown to be manageable tasks:

1. **Error classification.** We wish to determine the various types of errors involved in parallel message passing programming and develop a methodology for efficient debugging of parallel message passing programs. A number of new types of errors arise when dealing with parallel message passing systems. We still believe that dimensions 1 and 2 can be applied as is, whereas dimension 3 must be extended to contain error types caused by message passing.

2. **Tool development.** Understanding these error types makes it possible to write specific tools that can greatly assist the programmer to more easily debug parallel message passing programs.
3. **Automation.** It is possible that some of these tools can be semi-automated to remove part of the burden of debugging from the programmer. By focusing on different error types in an isolated way, tasks that might have been intractable become tractable, and in some cases, it is possible to automate the debugging or correction process.

If these three tasks can be accomplished, they will promote the writing of parallel message passing programs by allowing easy-to-use debugging tools that users will find useful.

2.6 Error Classification

For studying the task of debugging parallel programs, Figure 2.3 illustrates a good starting point. An error in a parallel program can occur at any of the three different levels shown in Figure 2.3. The data decomposition can contain errors as well, but this thesis is not concerned with these types of errors. Data decomposition is a large separate subject that has been described in detail in books such as [Fos95, FJL⁺88].

We briefly discuss some of the types of errors that can be encountered at the three different levels. The errors at the sequential level have already been described in the previous sections, but as mentioned, many of these errors occur in the parallel domain as well. In particular, it is worth noting that the cause/effect chasm mentioned in [Eis97] further widens as the possibility for even greater distance between cause and effect arises when message passing is involved.

When messages propagate from one piece of code to another through message passing, an incorrect value can occur and be used in a piece of otherwise correct sequential code. The distance between cause and effect in a sequential program is limited to the one process and its source code. In a parallel program, the distance can potentially be much greater as processes communicate. This increases the distance in both dimensions: space and time. The spatial distance increases as the cause and effect can now occur in different processes. With respect to time, the distance can potentially increase as well. When data is transmitted from one process to another, the time gap between the cause and the effect of an error become larger as it takes substantially more time to pack, transmit, buffer and unpack data than it does to retrieve it from local memory. Combining a large spatial distance with an increased temporal distance further complicates locating the cause of the error. Figure 2.5 illustrates this situation. Process A computes a bad value (0) for variable *a* and sends it to process B. Process B uses the value of *a* as a divisor and hence crashes. It immediately looks as if the error is caused by faulty division, but instead it is caused by a wrong computation by function *f* in process A. This error then

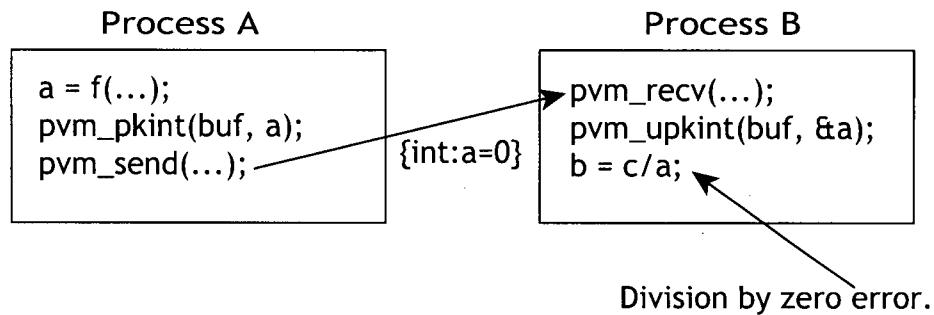


Figure 2.5: An illegal value 0 for the variable `a` was propagated from process A to process B, where it caused the program to crash. This example shows how the cause/effect chasm is widened by message passing.

propagates to process B through message passing. This example is typical of how the cause and effect chasm negatively affect message passing programs, because now errors can be propagated from one process to another through the network. Furthermore, the chasm widens when an error propagates through a number of processes before being detected.

The problem with wrong values in messages carries directly into debugging at the message level. Some of the errors that occur at this level are as follows:

- Wrong values (variables) sent/received. This is an example of one of the types of errors Knuth classifies as a type T (Trivial typo) error in [Knu89].
- Too little/much data sent/received. This fits nicely into the M category (mismatch between modules); the programmer is unaware of the mismatch between packs and unpacks.
- Variables packed/unpacked in the wrong order.

The highest level of debugging is in the communication protocol. One well known type of error is deadlock. Deadlock can occur for a number of different reasons:

- If a process crashes due to an illegal computation, and if another process is blocked in a receive call, waiting for a message from the crashed process, a deadlock occurs. Alternatively, if the process does not crash but sends a message to a wrong or a nonexistent receiver, the same receiver blocks, waiting for a message that never arrives.

This scenario with missing messages can easily happen in a master/slave configuration, where the slaves communicate with each other. For example, in a ring pattern, if the process IDs are stored in an array and each slave uses a wrong index to access this array and thus sends its message to the wrong receiver, all the processes eventually block and

create a system wide deadlock, because none of the messages are delivered to the correct receivers.

- A safe program is defined in [BD95a] as a program that does not require any buffers to complete. That is, communication is synchronous. Often asynchronous message passing is utilized to overlap computation with communication. Such programs are no longer safe, and can deadlock due to an insufficient number of buffers. We investigate this in detail in Chapter 8 and Appendix E.
- Messages delivered to the wrong receiver. Depending on the implementation of the program, this can lead to deadlock, or if wild cards are used, the message can end up being delivered to the wrong receiver and potentially cause errors in that process.
- Not only can messages be sent to the wrong receiver, but receivers can also attempt to receive a message from a process that is not sending. This can cause the process to block indefinitely, and at worst, cause a deadlocked process.

All these points are potential pitfalls in the parallel programming domain. All these errors require debugging. Some of them are fairly easy to correct, while others are more problematic.

2.7 Tool Development

In the previous section, we gave examples of the types of errors that can occur, and divided them into the three categories associated with the breakdown in Section 2.1.1: sequential errors, message errors, and protocol errors.

These errors are conceptually different; sequential errors are errors found in the straight line code. Message errors are errors caused by or associated with messages: a message can contain wrong data, which can affect otherwise correct sequential code (this is an example of an overlap between errors at different levels). The message can be received by the wrong receiver, or at the wrong place in the correct process, which can more easily happen when wild cards are used extensively. The overall structure of the messages, the protocol, can contain errors as well, which can result in messages being sent to wrong receivers.

Given the difficulty in having users adopt tools, we believe that in order to increase the usage of a new tool, it must be designed with the following goals in mind:

- It is vital that the tool can be used directly on the source code. When using development tools, part of the final program code is generated or inserted by the tool. This means that no complete source code exists for the entire program. If the development tool does not support the debugging task, debugging becomes cumbersome and complicated. Some tools generate a complete source, but machine generated code is typically hard to read and

understand. An error in the generated code, rather than the user's code, is very difficult to find. If the user can simply recompile or relink with a debugging library to utilize the tool, the likelihood that the user will adopt the tool is higher.

- To promote the usability of the tool it must be easily executable, either from the command line or within a simple interface that does not require the user to learn a new environment.
- Finally, the tool should enable users to find and correct specific types of errors, depending on their manifestation. We believe correct tailoring of the tool is one of the most important goals. Not only does this reduce information overload, but also makes certain complicated and time-consuming debugging tasks easier. This is achieved by targeting a specific type of error, using the information extracted from the program and the messages. Examples include the ability to extract one process from a parallel system and debug it sequentially. We return to this issue in Chapter 4.

These three goals are supported by the design goals proposed by Eisenstadt in [Eis97]. The most important ones are these:

- Computable relations should be computed on request by the tool, not left to the user to deduce on his own. Examples of violations of this design criteria include the often limited number of views found in many visualization tools [MHC94, NAW⁺96, HE93].
- Displayable state should be displayed on request, not left to the user to draw or visualize. This design goal is parallel to the previous one, and many of the tools that do not meet the previous goal inherently do not meet this goal either.
- Views for 'key players' (important pieces of information) other than variables should be provided. This design goal is the most frequently violated when considering *N*-version debuggers [Pdb, Hoo96]. These tools are designed for the sequential programming domain, and thus, do not offer easy access to information not indigenous to this domain.
- A variety of navigation tools should be provided at different levels of granularity. Instead of locking the focus on the sequential code, or the code of a number of sequential processes in parallel, the user should be able to change the level at which the debugging takes place. Many replay tools/debuggers [XWXS96, TSS96, KV97, Arv92, CFR95] focus on one level, namely the sequential code, making it virtually impossible to change the focus during the debugging session to, for example, the protocol, or the messages.

Thus, we propose a multilevel tool whose modularity (levels) closely follows the error classification mentioned in Section 2.1.1 and the above design goals. (See Figure 2.3). The last of the above points can be expanded into the two following design goals for such a tool:

- **Conceptual modularization.** Depending on which type of error the user is trying to correct, an appropriate tool should be applied. As a result, certain parts of the tool are tailored specifically to finding and correcting errors of a specific type, which reduces the amount of extraneous information reported by the specific debugging task.
- **Extensibility.** The overall debugging tool should allow for easy extension as new tools are implemented and need to be added.

2.7.1 Automation

If a certain task in a debugging session can be automated, then the tool should do so. This follows directly from the design goals in the previous section. For example, when trying to resolve deadlocks, it is possible to automate the search for a way to change the program to avoid a deadlock; at a higher level it is possible to automate the verification of the protocol of the system at runtime.

Protocols can be specified in process algebras such as CSP [Hoa78, Ros93, Ros94], and verified using tools like FDR [For] from Formal Systems. However, in order to use these tools, the user must have a strong background in theory as the specification of a protocol is a complicated task. For CSP models, the protocol is checked for deadlocks, livelocks and fairness constraints. Even if the programmer uses a tool like FDR to check the protocol, the potential for errors is still present. The protocol specification is not an implementation of the protocol. When the program is developed, the protocol must be implemented as well. Problems can arise when the implementation of the protocol does not match the specification. This problem is a specialization of a problem known in software engineering: guaranteeing that the implementation of a system adheres to the specification.

We propose a protocol testing module, where the specification is much easier to write, and where the system simply checks all messages against this specification of the communication protocol. That is, the protocol is not verified but all the messages in the system are checked against the specification.

Many problems in the sequential domain are NP-hard or undecidable, hence the need for heuristics. The (debugging) problems considered in the parallel domain are no easier.

We believe that by focusing on a particular type of error and developing heuristics directed towards this error type, it might be possible to raise the limit of what can be computed, and even automated. This means that by narrowing the search space, as shown in Figure 2.4, we can increase the size of the set of problems that can be solved (or semi-solved). One such example is the deadlock correction algorithm, described in Chapter 6.

2.8 Tool Support for Parallel Program Development

In previous sections we described the parallel programming domain, showed some of the numerous places where errors can occur, and described some of the errors. This discussion shows the importance of good tools for programmers working within the parallel programming domain.

In this subsection we briefly describe some of the problems that exist with tools for programming, debugging or development.

A parallel programming environment is an obvious tool to use when developing parallel programs. In addition, these environments can greatly reduce the number of errors programmers make.

A number of these tools have been developed over the years, and in Section 2.3 some of them are presented. Though many of these tools restrict the user, to avoid certain types of errors, the risk of errors in user code remains. These errors can cause subsequent errors in the generated code as well. Even when using tools or programming languages with built-in support for parallel programming, the problem of locating and correcting errors persists. Despite the obvious advantages found in many of these systems, Pancake argues that not many are widely adopted. In fact, it is claimed that “often only the developers of the tools end up using them in the end [Pan93b]”. A number of reasons for this paradox is given:

Steep learning curve. Many of the tools are advanced and offer a wide variety of functionality; they can be quite difficult and time consuming to learn.

Difficult abstraction. The abstraction adopted by a tool, for example the way a program is represented, the way communication is specified, and its limitations, can be difficult to understand and familiarize oneself with.

Restrictiveness. Many tools are so restrictive that they work against the programmer. One example is a tool that assures that any code created is deadlock free. This apparent advantage has a drawback: programs with dynamic communication can not be expressed using this tool.

Conservatism. There tends to be general skepticism towards new tools or languages, especially if they require the user to learn a new language, or a new integrated tool.

Given the difficulty with tool adaptation and the inherent conservatism that perpetuates the use of well known methods and tools, debugging is still unavoidable.

Debugging tools can suffer from the same problems as the development environments. So when developing tools for debugging, the above points should be kept in mind as part of the design goal. That is, the tool should be easy to learn and use, and the abstraction adopted by the tool should not restrict one’s ability to perform a specific debugging task.

We attempt to avoid the four disadvantages mentioned earlier in the following ways:

- By designing the tools to have a simple user interface; that is, without many different windows, menu bars and buttons.
- The abstractions in our approach closely follow the natural abstraction found in a parallel program, namely, the three levels proposed.
- As with any tool a certain restrictiveness is unavoidable, but by satisfying the design goals mentioned in Section 2.7, we believe that the tool becomes less restrictive and rigid.
- General conservativeness is difficult to counter; learning any new tool requires some effort by the user, but we believe that by addressing the previous three subjects we can reduce the amount of inherited conservatism. Whether the user now wants to use the tool or not is a question of personal taste, not so much relying on steep learning curves, restrictiveness, or complicated abstractions.

With a good understanding of error types, design goals, and the problems with existing tools, we have formulated a multilevel debugging strategy and specified a number of design goals such a tool must satisfy. In the next chapter we describe the design of Millipede, a prototype multilevel debugging tool.

Chapter 3

Millipede - A Prototype Multilevel Parallel Debugger

“The point I have been patiently trying to make,” Godwin said impatiently, “is that you expect far too much of a first sentence. Think of it as analogous to a good country breakfast: what we want is something simple, but nourishing to the imagination. Hold the philosophy, hold the adjectives, just give us a plain subject and perhaps a wholesome, nonfattening adverb or two.”

— *Godwin to Danny Deck, Some Can Whistle*

In this chapter we present Millipede, a prototype multilevel debugger designed in accordance with the ideas and methodologies presented in the previous chapters. We show how it is possible to write debugging and analysis tools specifically tailored to the different levels mentioned earlier. These tools are modules that are incorporated into Millipede. However, many of these can be executed outside of Millipede, given the information about message history and the communication protocol extracted by Millipede.

3.1 Design Criteria

An aim of this dissertation is to show that debugging can be decomposed into several tools, each tailored to a specific error type, thus working on different levels of the program structure. These levels are sequential code, message passing, and communication protocol. As argued in Section 2.4, we propose a bottom-up technique referred to as ‘multilevel debugging’ (illustrated in Figure 2.4) and develop tools to support debugging according to this methodology.

A number of such tools are useful for locating and correcting a number of different errors; this is illustrated by the overlapping error types within the specialized debugging tools in Figure 2.4. On the other hand, even though this approach might prove to be useful, situations could arise where such a strategy is not applicable. This could happen if an error occurs that is not covered by any of the tools.

Given the problems with existing tools (see Section 2.3.7), as well as the apparent lack of tool usage, we formulated a multilevel debugging strategy. Using this as the foundation for the Millipede prototype, we can summarize a number of important design goals.

1. Access to source code debugging is vital as errors may be located in the sequential code. It should be possible to apply the user's favourite sequential debugging tool to debug the sequential part of the parallel program.
2. Access to delivered messages, as well as messages still in the message queues. Not only do many of the modules make extensive use of this information, but it might also be useful for new modules that perform other forms of analyses on the unmatched (unreceived) messages.
3. Automation of complex parts of the debugging process. This technique, combined with relation-on-demand computations, reduces information overload.
4. Though not a direct design goal, extensibility is another important issue in the implementation of a tool like Millipede. Extensibility of the message passing calls should be developed, thus allowing the user to add new functionality to existing message passing calls. This could become necessary when other modules are developed by other users of Millipede. In Chapter 6, we show how extensibility plays an important role in a tool like Millipede: The analysis described in Chapter 6 relies on the ability to extract information about messages. This information can easily be extracted from the runtime system's internal relations and tables, which is further described in Chapter 5. Using these relations and tables simplifies the implementation of other modules in Millipede.

3.2 The legs of Millipede

The Millipede prototype debugging system is written for the PVM message passing system, and consists of the following main parts:

- A core system built into the communication system consisting of wrapper functions for all the communication calls. These new functions (`_PVM_XXX`) are added to the original communication library. They execute the Millipede debugging code and then call the original

message passing functions (`pvm_XXX`). Figure 3.2 shows a few examples of these new functions. Functionality, such as writing and reading log files, and logging messages and protocol information, is implemented in these new functions.

- A runtime system that consists of several separately executing processes, which allows the user to interact with the debugging system.
- A number of analysis tools/modules which are described in greater details in the following chapters. These tools are typically invoked by the runtime system as a result of a query or a command sent to the system by the user.

The current implementation of Millipede supports debugging parallel message programs that use PVM. Millipede uses PVM to communicate internally as well. However, an MPI port of Millipede to MPI and to also work for MPI programs is straightforward.

3.2.1 Overview

As stated, Millipede is a prototype implementation of a tool that utilizes a multilevel debugging strategy. This implies that the implementation and design closely follows the layered approach proposed in the methodology.

Figure 3.1 shows a graphical representation of the Millipede debugging system and how it interfaces with the application being debugged and the message passing system. The parallel application has the message passing library linked into its executable. The current version of Millipede uses PVM. The Millipede Core System is a re-implementation of all the functions in the PVM library, that is, when a message passing function is called from the application, the Millipede version of that function is called. Any information that must be written to or read from log files is handled here, and the original PVM functions are called. In general, code is wrapped around the original call; some is executed before the call and some after. This code can be thought of as a logging aspect (as in aspect oriented programming) [Asp03], that is, it is executed before and after the original message passing code.

Depending on whether log files are written or read in a subsequent debugging session, different parts of this code are executed. The core system communicates with the runtime system, informing it about messages sent and received. The information about the messages are kept in relations in the runtime system.

The various modules and analysis tools are separate functions/processes that can obtain information from the runtime system about the messages and the protocol. These modules can then perform the analyses or tasks and send the results back to the runtime system.

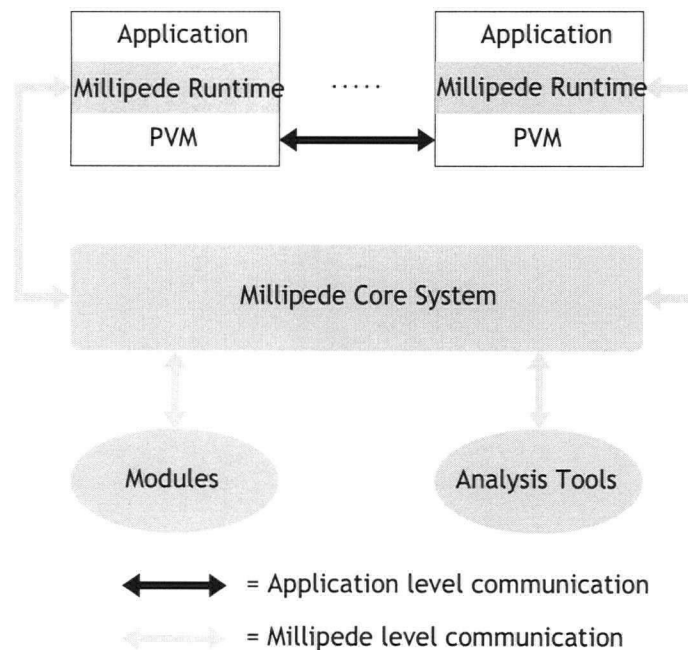


Figure 3.1: The implementation of Millipede. The gray arrows represent communication within Millipede, and the black arrows represent communication between applications. The Application/Millipede Runtime/PVM boxes represent one process.

3.2.2 Implementation

In the following subsections we briefly describe the implementation of the wrapper functions and the runtime system. Details about the implementation of the modules can be found in the respective chapters.

Wrapper Functions

By redefining the PVM functions, as shown in Figure 3.2, the C compiler substitutes all PVM calls in the user code with calls to the equivalent underscore functions (e.g., `_PVM_pkint` instead of `pvm_pkint`). These functions then perform the Millipede debugging code (informing the runtime system about changes, prompting for input, printing output, writing log files, etc.). The implementation of these functions is linked into the original PVM library (`libpvm3.a`). When a PVM program is compiled, the redefinition is only included if the `MILLIPEDE` flag is set during compilation. If this flag is not set, the program executes like a normal PVM program, but if the flag is set, the Millipede version of the functions is executed. This way of switching between normal and debugger execution is easy to manage, and does not require any rewriting of the program, just recompilation and re-linking. Even if the program is compiled with the `-DMILLIPEDE` option the user can choose a normal execution by setting an environment variable.

```

:
#define pvm_initsend(X)  _PVM_initsend(X, __FILE__, __LINE__)
#define pvm_recv(X,Y)    _PVM_recv(X,Y, __FILE__, __LINE__)
#define pvm_upkint(X,Y,Z) _PVM_upkint(#X,X,Y,Z, __FILE__, __LINE__)
#define pvm_pkint(X,Y,Z) _PVM_pkint(#X,X,Y,Z, __FILE__, __LINE__)
:

```

Figure 3.2: Examples of redefined PVM functions.

The Runtime System

The core of the runtime system consists of the following three main parts:

1. A centralized message number administrator process is responsible for assigning unique numbers to all messages. These are not timestamps as defined by Lamport in [Lam78], but rather a unique marker for each message in the system. The Lamport timestamps impose a partial ordering of the messages using a happens-before relation, whereas the message numbers are merely used to identify messages. This is necessary for matching the sending process of a message with the receiving process. It also makes it possible for the user to easily distinguish messages when working with the Message Debugging Module.
2. A number of status windows where the system reports information about events and additional information requested by the user.
3. A driver/interface process in which the user interacts with the runtime system. It also maintains information about the number of running processes in order to report termination.

When a parallel program is running and log files are generated, all the information linking log files with program files is collected in a project file. This file contains all the information required for a module in Millipede to locate the needed log files. Refer to appendix A for an example of a project file.

Within the Millipede runtime system each process has a message queue. In this queue all the information about the messages sent or received by the process is maintained. The modules described in the remaining chapters make heavy use of this information.

Examples on how the modules use this information can be found in Chapters 4, 5, 6, 7, and 8.

3.3 The Sound of Little Legs Running

In accordance with the multilevel debugging strategy, Millipede has the following levels:

The Sequential Level – The module at this level facilitates the application of sequential debuggers, as well as other sequential analyses or profiling tools, to one single process extracted from the parallel application. Chapter 4 describes the Sequential Debugging Module in greater detail.

The Message Level – Here, we are concerned with messages sent between two processes. The current tool in Millipede at this level allows for the interactive inspection and debugging of messages, and supplies the user with a query language (MQL) for querying messages. Examples and an in depth discussion about the Message Debugging Module can be found in Chapter 5.

The Protocol Level – This is the last of the three levels and also the level that provides an overview of the entire application with respect to the protocol. This level contains three modules.

- The first module is the Deadlock Detection and Correction Module. This module is an example of using automation to reduce the amount of information presented to the user. When an application deadlocks, this module can be applied. The messages are analyzed and a suggestion for altering the source code to remove the deadlock is provided. More detail of the theory behind this analysis is given in Chapter 6.
- The second module is the Protocol Conformance Checking Module. This module allows the user to specify a number of constraints on the protocol and have the runtime system check all messages against these constraints and report any violations. This module can be used to advantage in the program development cycle when implementing the protocol from a possibly verified specification. Examples, along with more detail on the implementation, are provided in Chapter 7.
- The last of the three modules at the protocol level is the Buffer Allocation Analysis Module. This module performs an analysis on the message history; a graph based on messages is created and analyzed to determine the number of buffers needed to ensure efficient execution. By efficient we mean an execution that does not have any blocking send calls due to a lack of buffers. The algorithm is described in Chapter 8, along with a more detailed description of the general problem of determining buffer allocation in systems with different buffer allocation schemes. Theoretical results are derived for three problems with four different buffer allocation schemes.

Although we present the multilevel debugging strategy and the implementation of Millipede as having three distinct levels, a certain amount of overlap is present. For example, when using the Sequential Debugging Module to extract and execute one process from the parallel system,

if the log file belonging to that process is either corrupt or incorrect, or if the user specifically requests it, the runtime system will prompt for valid data values for unpacked data. This feature is conceptually part of the Message Debugging Module, but has proven to be a useful addition in the Sequential Debugging Module as well. Millipede currently contains a total of five tools, but it provides a general infrastructure for incorporating more. Millipede uses a simple command line interface, and Appendix D contains screen dumps of the Millipede interface windows.

Chapter 4

Sequential Debugging of Parallel Processes

“A computer lets you make more mistakes faster than any invention in human history—with the possible exceptions of handguns and tequila.”

— *Mitch Ratliffe*

Following the multilevel debugging methodology outlined in Chapter 2, errors should be fixed at the lowest levels before moving on to higher levels. The sequential code of each process constitutes the lowest level, so we first consider debugging code at the sequential level.

The two main problems we address in this chapter are:

- Providing the ability to extract a single process from a parallel system.
- Allowing the user to take advantage of existing sequential debugging tools on the sequential code of the parallel program.

The sequential debugging module allows the user to extract one process from the parallel system and replay the execution. It facilitates using any sequential tool to analyze or debug the process. The messages that the process would have received in the parallel execution are provided by the underlying Millipede runtime system. Extensive research and numerous tools have been developed in the area of sequential debugging. Instead of providing new tools that might be inadequate, the sequential debugging module enables the user to use existing sequential tools specifically tailored to finding and correcting the type of errors that arise in sequential code.

4.1 The Sequential Debugging Module

To use the sequential debugging module the user must first compile the program and link the Millipede runtime system to her code. The program is then executed in parallel until an error occurs. The runtime system collects all the messages that were sent throughout the execution and stores them in log files, one for each process of the parallel program.

The runtime system writes the total content of the messages along with the return value of the communication call. The messages and their content as well as the return values of the message passing function, are captured to assure that the sequential re-execution is exactly the same.

Handling nondeterminism is a problem for any debugging tool that attempts to locate errors by re-executing the code. In the case of the Millipede tool for executing a single process, we can classify nondeterminism into two types; the nondeterminism within the sequential code that does not affect message history and the nondeterminism that can affect either messaging.

Nondeterminism which does not affect the messaging can be handled with the same techniques used for handling nondeterminism within a sequential program. For example in case of random numbers the seed to random number generator can be fixed. As long as it does not affect the message history the message history provided by Millipede will produce the same execution and hence exhibit the same error.

It is possible however for the nondeterminism in the program to affect the messaging. It may affect the ordering of messages or the even the contents of messages to be received. If this cannot be fixed, then it can result in an invalid replay of the message history and the tool itself may fail. As in the previous case, it can be avoided by removing the nondeterminism to ensure the re-execution runs with the same message history. One particularly difficult type of error occurs when the nondeterminism is due to some type of timing measurements. Although messages are replayed in the correct order, it is impossible to ensure identical timings.

In conclusion, what distinguishes these two types of nondeterminism is whether the replay is unsuccessful in exhibiting the error, or whether the tool itself fails because of changes to the expected message history.

The message passing, and in particular, the order of the message receipt, is fixed. Therefore, it is not necessary to timestamp messages. It is necessary to capture the return values of all the message passing function calls, as the program behaviour may depend on these values. For example, nonblocking receive calls can return without receiving any data, and in order to replay the execution when the process is executed sequentially, these values must be stored as well.

Consider the code in Figure 4.1. If no message is ready in the message system the process will execute the else part of the if-statement. This can happen a number of times, and each time the state of the program might change. In order to replay this behaviour when the program is

```
arrived = false;
while (!arrived) {
    arrived = pvm_nrecv(tid, msgtag);
    if (arrived)
        pvm_upkint(array, 10, 1);
    else
        // Do something else
}
```

Figure 4.1: If the `arrived` variable is not logged there is no way to tell how many times the loop is executed, thus perhaps putting the process in a state that does not match the state it was in when it logged the messages.

re-executed with messages supplied from the log files, the value of the `arrived` variable must be stored each time `pvm_nrecv` is called, and then returned to the process. This ensures an execution that matches the original execution.

Figure 4.2 illustrates how an application's executable contains the original PVM functions when linked with the PVM runtime system only, and how it contains both the PVM and the Millipede runtime systems when the `MILLIPEDE` flag is set at compile time. The Millipede runtime system is compiled into the PVM library, but the replacement of the message passing calls only occurs when the `MILLIPEDE` flag is specified. Since the Millipede runtime system is not compiled with the `MILLIPEDE` flag, the calls to the original PVM function are called from the `_PVM_send` function.

Appendix A shows a complete example from compiling an application to extracting one process using Millipede, and debugging it sequentially using Gdb.

4.2 Limitations

One of the problems with this straightforward logging approach is that log files can potentially be very large. One solution is to try to reproduce the error with a smaller data set, however, this approach is not always feasible.

A different approach is to use checkpointing. The idea is to save an image of the running process and then purge the log files. If the process needs to be debugged, it can be restarted from the checkpoint and the execution replayed from there using the messages in the log files. The problem with checkpointing is restoring state; the kernel state associated with the process must be consistent after restoration.

There has been work done on tools that allow for the checkpointing of the sequential state of a process. A lightweight library called 'save_world' was developed by Bennet Yee [Yee96] from the University of California in San Diego. Another well known system is Condor [LLM88], which

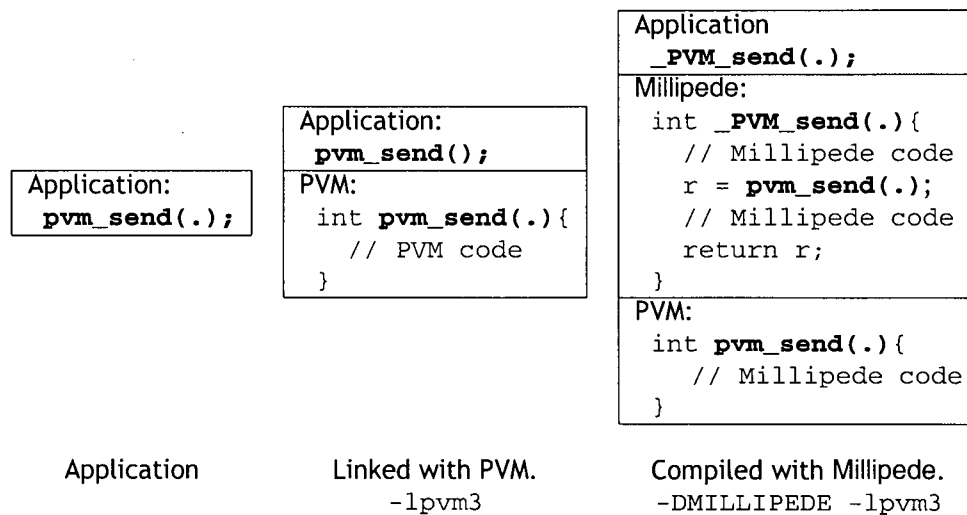


Figure 4.2: An example of the different parts of an application; first, the application; second, the application linked with the PVM library, and third, the application and communication library with the Millipede runtime system added.

also supports process migration. If such a checkpoint library were incorporated into Millipede, it would be possible to save and resume an execution at any given time, and at the same time purge log files and reduce their size considerably. Two important benefits arise from checkpointing: first, a reduction in the disk space needed to save log files as these are purged at checkpoints, and second, the time it takes to restart the process and re-execute it to the state of failure is reduced by using the most recent checkpoint as a starting point for debugging.

One remaining problem occurs if the cause and effect chasm spans the checkpoint. That is, if the source of the error is before a checkpoint, and if the manifestation is after, then all evidence of the source of the error will have been removed because the process was checkpointed and the log files purged. This could be solved by storing the purged log files and all checkpoint images on secondary storage until the debugging process has finished, if enough space is available.

The sequential debugging module is tailored to extract the messages of a parallel system and support the debugging of one sequential process based on the log files captured while the parallel system executes. However, there are classes of errors that are difficult to find using this approach. For example, if an error does not occur every time the program is run, and if log files are not generated during the execution that encounters the error, the Sequential Debugging Module is of no use.

Millipede is not thread safe; if it is used with a multithreaded program, there may be problems with the communication library and nondeterminism due to thread scheduling. In general, debugging multithreaded parallel programs introduces yet another type of concurrency that further

complicates the debugging process. The scheduling of threads can result in nondeterminism that cannot be controlled by the user, and checkpointing with threads could potentially become a problem as well. Thus, the MPI standard is not defined as thread safe; the two major public domain implementations of MPI, LAM and MPICH, are not thread safe.

4.3 Examples

The following examples illustrate the types of errors that can be found and corrected by using Millipede.

As a basis for these tests we used a master/slave implementation of an iterative hyperbolic differential equation solver [FJL⁺88], which we seeded with two different errors. The first error is a division by zero caused by a variable value of zero propagating through a message. The second error is an out-of-bound indexing error; the array indexed is not big enough. The results show that the errors that occurred in the parallel execution are faithfully reproduced in the sequential execution of the process containing the error. Figures 4.3 and 4.5 show the relevant parts of the slave program containing the errors.

4.3.1 Division by Zero Error

Consider the code shown in Figure 4.3. The assignment causes a division by zero if the variable `nproc` equals zero. When the program is executed in parallel the slave process executing the illegal division encounters an arithmetic exception and terminates.

In this case, Gdb is used in combination with the sequential code and the messages that were extracted by Millipede. As shown in Figure 4.4, the error is easily located using Gdb. In contrast, finding this error using N versions of Gdb online, the programmer would have to single step each process to a point where the communication has occurred and the division by zero executed.

```
pvm_upkint(&nproc, 1, 1);  
pvm_upkint(tids, nproc, 1);  
pvm_upkint(&n, 1, 1);  
.   
e = n % nproc;  
.
```

Figure 4.3: Sequential code with divide by zero.

Tools, such as DIWIDE, that allow macro stepping or tools, such as p2d2 or TotalView, that allow control of all debuggers at the same time, can be used in a similar way to locate the error. Such tools are good for master/slave or processor farm computations as a number of similar

```
(gdb) step
45  e = n % nproc;

Program received signal SIGFPE Arithmetic
exception 0xef4a86a8 in ()

(gdb) print n
4
(gdb) print nproc
0
```

Figure 4.4: Using Gdb to locate the error in the sequential code of a process from a parallel program

processes are controlled as a unit, but this is not as easy for a pipeline computation or a parallel program which is functionally decomposed; the different processes execute different code, thus making it difficult to control all the processes in a collective manner.

4.3.2 Memory Errors

If a process in a parallel system contains a memory leak or memory error, often, especially in C, the most likely manifestation is that the process terminates because of an illegal memory reference. When executed in parallel, even if one process terminates, the others continue to execute until they crash, deadlock, or finish incorrectly. In sequential programming there are tools, such as Purify [Pur], that are effective for discovering memory reference errors. Purify links a runtime library to the process that tracks memory references and reports any illegal ones. Unfortunately, no parallel version of Purify exists. The sequential version does not easily lend itself to finding memory leaks in a parallel program. The only way to apply Purify to a parallel program is to apply one instance to each process.

By using Millipede to extract the process that crashed, along with the corresponding log file, it becomes possible to use Purify to find the offending code that contained the illegal memory reference.

If a parallel system consists of a number of instances of the same program, for example a master/slave or a processor farm application, it suffices to apply Purify to a single instance of the slave processes in order to catch memory errors. This approach reduces the amount of information the programmer needs to consider during the debugging process, and may reduce the time needed to complete the debugging task.

Figure 4.5 shows a code fragment that indexes an array out of bounds. The `x` array is too

small, and at index `nodes+1` an index out of bound error occurs. This error can result in two different program behaviours: an incorrect result, or a segmentation fault where the process terminates abnormally. This error is easily detected by using a tool like Purify. Note, this error was introduced into the program when the parallel version was developed. Since the data must be distributed across a number of processes, this error might not have been present in the sequential version.

```
x = calloc(nodes, sizeof(double));
.
.
for (i=1; i <=nodes; i++)
    x[i] = (1*(start+i-1))/(n-1);
.
```

Figure 4.5: Source code with a memory error: the `x` array is one element too short.

Figure 4.6 shows the output from running Purify on the extracted process. The output clearly marks the problematic array and specifies that the problem is an attempt to write past the end of the array. In addition, the line in which the array is allocated is printed out. It is now a simple problem to correct the error by either allocating a bigger array or changing the condition.

```
ABW: Array bounds write. This is occurring while in:

main      [Wave_slave.c:57]

      for (i=1; i<=nodes; i++)
==>    x[i] = (1*(start+i-1))/(n-1);

Writing 8 bytes to 0xdc630 in the heap. Address 0xdc630
is 1 byte past end of a malloc'd block at 0xdc5a8 of
136 bytes. This block was allocated from:

malloc    [rtlib.o]
calloc    [rtlib.o]
main      [Wave_slave.c:50]

==>  x = calloc(nodes, sizeof(double));
```

Figure 4.6: Using Purify in combination with Millipede to locate memory errors.

The advantage of having the extracted process is the ability to use tools like Purify that

have not been ported to the parallel domain. These tools can be more effectively used with one process rather than trying to coordinate the use of N versions of them running at the same time. It is possible to extract one process and its corresponding log file, and then debug it as a sequential program.

4.4 Implementation Details for the Sequential Debugging Module

In this section we briefly explain some of the details of the implementation endemic to the Sequential Debugging Module. The Sequential Debugging Module consists of two parts:

1. The collection phase that intercepts the values and names of variables being packed (calls to the `pvm_pkXXX` functions) and return values of all the message passing functions.
2. The replay phase that reads the log files written in the collection phase, when `pvm_upkXXX` functions are called.

The collection part is straightforward. When packing functions are called the names and the values of the variables are written to the corresponding log file. All message passing functions also write their return values to the log file. During replay, the log files are read, that is, instead of performing a call to the PVM library the log files are read, and the values are returned to the caller as if a message had arrived from the network.

The majority of the code for this part of the tool is for checking that the values read from the log files are consistent with the message passing calls in the code. This is done by comparing the names of the variables in the `pvm_upkXXX` call with the names of the variables in the log file. If a mismatch between variable names or types is found, Millipede prompts the user for a value for those variables that were unsuccessfully unpacked.

4.5 Summary

We have shown how it is possible to extract one process from a parallel system and debug it sequentially. After extraction, it can execute as a standalone UNIX process and be debugged using sequential debugging tools. This allows the user to correct errors in the sequential code of a parallel program as if it were a purely sequential program.

Debugging message content, and obtaining an overview of messages and message queues in the message passing system, are not supported by the Sequential Debugging Module. These problems belong to the message level of Millipede, and in the following chapters we introduce tools that provide the user with debugging capabilities at this level.

Chapter 5

Message Debugging

If we knew what it was we were doing, it would not be called research, would it?

— *Albert Einstein*

Once errors are corrected in the sequential code, the focus turns to process interaction. This interaction happens through message passing. The purpose of the Message Debugging Module is to support the location and correction of errors which cause incorrect messaging and message content. We provide two tools to help in correcting the messaging:

- The first allows the user to interactively inspect and debug messages as the program runs. That is, as messages are delivered to the process by the message passing system, the user can inspect and change parts or all of the message without the need for a debugger, such as Gdb.
- The second is a simple query language to aid in querying the message history maintained by the runtime system.

5.1 Interactive Message Debugging

In the Sequential Debugging Module the focus is on the key players of sequential programming, for example, the variables and the control flow. When moving up the hierarchy, the next level is the one concerned with messages passed between two processes. As illustrated in Figure 4.3, an otherwise correct piece of sequential code can produce faulty results or even terminate with an error due to the cause/effect chasm widening through message passing; a wrong value can be sent from one process to another and cause an error.

The idea of interactive message debugging, or inspection, is to allow the user to choose one or more processes, and while the parallel program executes, give the user separate views/windows for each of these processes. These views show the messages that are sent to the process and allow these to be changed. This is particularly useful if the user notices a message that contains a wrong value, but does not want to terminate the execution to correct the problem. The value is simply changed before it is delivered to the process and execution can continue uninterrupted.

At first, this might seem like a typical application of the *N*-version strategy. However, the difference is that in Millipede, the user chooses which processes he wishes to interact with at any time during the debugging process. These views can be turned on and off at will. This corresponds to the idea of computing relations on demand, and also attempts to reduce information overload.

Naturally, the possibility of information overloading exists, especially if the user turns on too many views at once. However, by starting out by displaying a small amount of information in a few windows (or even no windows to begin with), and allowing the user to extend the view by opening new windows and closing down those he does not need anymore, it differs from the typical use of an *N*-version tool. One difference between the tool here and the *N*-version type of tools is that in the latter case, the user adds views rather than removes them, which allows him to focus on the activities at hand.

If a receiving process attempts to unpack more data than the sender sent, Millipede will issue a warning with the line and the variable name that cannot be unpacked. In addition, in order not to terminate the debugging session, Millipede will prompt for a value for the variable. Figure 5.1 shows an example where the message contained too little data. Figure 5.1 represents a window for one process; for each process observed (i.e., where the user has decided to perform interactive message debugging), a separate window appears. Figure D.2 in Appendix D shows a screen shot of an actual debugging session.

Receiving New Message:

```
Line 4: pvm_recv(4,0) <ok>
Line 9: pvm_upkint(&a,1,1) = [2] <ok>
Line 12: pvm_upkint(&b,1,1) = [?] <error>
No value available for int: b.
      Please specify a value. int: b = 78
```

Figure 5.1: No value for *b* was sent or read from the log files, so Millipede prompts the user for a value. Debugging can continue once a value has been specified.

This technique can also be utilized in conjunction with the sequential debugging module for testing purposes: if a replay file does not exist, the runtime system will prompt the user to supply

values for all the incoming messages. This allows for the testing of a single process without running, or even having written, the code for the other processes of the parallel system. The user simply runs the program with the Millipede runtime system turned on, specifies no log file and debugs as usual, with the exception of having to specify values for unpacked variables for all incoming messages and return values for calls to procedures such as `pvm_nrecv`. It also provides a technique for interactively testing the program with extreme or incorrect values. In addition, this technique is particularly useful when prototyping a program, and it matches the idea of iterative program development proposed by Araki et al. [AFC91]. This is especially good for testing master/slave, processor farm and pipeline code.

Figure 5.2 shows an example of how to use the interactive message debugging module. Note, Millipede not only shows the unpacked values, but also the names of the variables that they are unpacked into as well as the line number of the PVM communication call. Providing the user with information about the line numbers and variable names addresses the issue of mapping the display back to the source code. An interesting problem with interactive message debugging is that it can exhibit the opposite of information overload; when a receive call expects a large message, it requires the user to type an unreasonable amount of data. The user has the option of replying `f` and the system prompts for the name of a file containing the data to be used.

Receiving New Message:

Line 78: `pvm_nrecv(-1,0)` <ok>

Line 81: `pvm_upkint(&nproc,1,1)` = [2] <ok>

Do you want to change this [y/n] ? **y**

int: nproc = 2. New value = **3**

Line 82: `pvm_upkint(tids,3,1)` = [262151,262152,262153] <ok>

Do you want to change this [y/n/f] ? **f**

Filename: **tids.txt**

Read: [262150, 262151, 262152] <ok>

Figure 5.2: A message with tag 0 is received from anyone (-1 is a wild card in the receive call), and two unpacking instructions are executed. One that unpacks one integer and one that unpacks three. If the messages are supplied from a replay file and an error or some type of inconsistency appear the <ok> will be replaced with an error message and the user will be asked to provide a value that is compatible with the destination variables for the unpacking call.

5.2 Message Queries

We now focus on the second part of the Message Debugging Module: the message querying tool along with the Millipede Query Language (MQL) allows the user to write SQL-like queries for an internal database of messages maintained by the Millipede runtime system.

One problem with existing tools is that only a fixed number of views are provided. The views, or queries, provided by tools are typically those that the developer of the tool considers useful. Since the developer and the user might have different foci or views, this is one of the contributing factors as to why tools are primarily used only by their developers [Pan93b]. Three of the design goals for the Millipede debugging system mentioned earlier are as follows:

- Displayable states should be displayed on request.
- Computable relations should be computed on request.
- Information overloading should be avoided.

Not only can state and computable relations be displayed on request, but queries can reduce the complexity of information gathering from the message history; the alternative would require filtering through a large quantity of data by hand.

The problem with a fixed number of views is easily solved for the Message Debugging Module by giving the user a higher degree of freedom within the tool, that is, allowing users to define their own queries. If the message history is viewed as a large database, it is natural to develop an SQL-like language to facilitate queries.

5.3 User Defined Queries

The Millipede runtime system organizes the messages into four relations: *Senders* and *Receivers* as shown in Tables 5.1 and 5.2, which contain the overall information about the messages, and *SentMessages* and *ReceivedMessages* as shown in Tables 5.3 and 5.4, which contain the details about the messages.

MsgNo	Size	STID	SLine	SFile	STag
:	:	:	:	:	:
7	154	262152	118	slave.c	5
:	:	:	:	:	:

Table 5.1: The *Senders* relation.

MsgNo	RTID	RLine	RFile	RTag
⋮	⋮	⋮	⋮	⋮
7	262150	86	master.c	5
⋮	⋮	⋮	⋮	⋮

Table 5.2: The Receivers relation.

MsgNo	No	SLine	SType	SVarName	SCount	SValue
⋮	⋮	⋮	⋮	⋮	⋮	⋮
6	1	107	int	me	1	...
6	2	108	int	nodes	1	...
6	3	114	double	y[1]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 5.3: The SentMessages relation.

MsgNo	No	RLine	RType	RVarName	RCount	RValue
⋮	⋮	⋮	⋮	⋮	⋮	⋮
6	1	87	int	who	1	...
6	2	88	int	result_length	1	...
6	3	94	double	y[index]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 5.4: The ReceivedMessages relation.

Queries are implemented using the four relations and the query language. To illustrate the use of MQL, we implement two useful queries: `locate`, which locates messages sent between two program lines, and `match`, which matches the packing functions with the corresponding unpacking functions of a message. Figure C.1 in Appendix C shows the grammar for MQL.

The `match` Query

Since many variable values can be packed into one message, one of the easiest errors to make is to unpack the message in the wrong order or into wrong variables. This means that values can potentially be swapped in the variables they are unpacked into. By querying the message passing system it is easy to verify in which order the values were packed, and in which order the values were unpacked. In addition, the name of the variables on both the sender and receiver side, and the line numbers are shown. This query is performed by executing the `match` query.

Figure 5.3 shows the wanted output of the `match` query on message number 6. The upper half of the figure shows the packing routines and their line numbers, and the lower half shows the unpacking routines and their line numbers.

```
(0)MILLIPEDE> match(6)
Message number: 6
----- Sender -----
File:      slave.c
Line 107   pvm_pkint(&me, 1, 1);
Line 108   pvm_pkint(&nodes, 1, 1);
Line 114   pvm_pkdouble(&y[1], nodes, 1);
Line 118   pvm_send(262150, 5);
----- Receiver -----
File:      master.c
Line 86    pvm_recv(262151, 5);
Line 87    pvm_upkint(&who, 1, 1);
Line 88    pvm_upkint(&result_length, 1, 1);
Line 94    pvm_upkdouble(&y[index], result_length, 1);
-----
```

Figure 5.3: By executing the `match(6)` query, Millipede will query the message queues for all packing and unpacking commands for message number 6.

The MQL code for the `match` query is shown in Figure 5.4.

The locate Query

If the user believes that a message has been delivered to the wrong receiver, it is useful to search the message database for messages that match a specific sender and receiver line number. Such a query, `locate`, can easily be implemented using MQL. Figure 5.5 shows the expected output from the query `locate(86, 118)`, that is, it lists the messages sent from line 86 in some process and received in line 118 in a different process (in practice, the sender and the receiver could be the same process).

The `Senders` and the `Receivers` relations have the field `MsgNo` in common. By joining these two relations and selecting the tuples where both the sender and the receiver line match the values 118 and 86, we obtain a new relation containing the result. Figure 5.6 shows the implementation of the `locate` query.

The `match` and the `locate` queries are two examples of the use of MQL. The user can write his own queries, ranging from very simple to arbitrarily complicated. In addition to the four mentioned relations, there are a few more bookkeeping relations. One such relation is the `TIDS` relation that maps process IDs to message queue numbers. Its purpose is to allow the user to develop queries that contain more information and have better formatted output.


```

define match(mno) as
begin
  print("Message number: %\n",mno);
  let SenderInfo be
    project
      select from Senders
      where (MsgNo == mno)
      over (SFile, STID, STag);
  let ReceiverInfo be
    project
      select from Receivers
      where (MsgNo == mno)
      over (RFile, RTID, RTag);
  print("\t----- Sender ----- \n");
  display SenderInfo
  using "File:\t%\nLine %\tpvm_send(%,%);\n";
  display
    project
      sort
        select from SentMessages
        where (MsgNo == mno)
        by (No)
        over (SLine, SType, SVarName, Scount)
      using "Line %\tpvm_pk(%,%,1);\n";
  print("\t----- Receiver ----- \n");
  display ReceiverInfo
  using "File:\t%\nLine %\tpvm_recv(%,%);\n";
  display
    project
      sort
        select from ReceivedMessages
        where (MsgNo == mno)
        by (No)
        over (RLine, RType, RVarName, Rcount)
      using "Line %\tpvm_pk(%,%,1);\n";
  print("\t----- \n");
end

```

Figure 5.4: The implementation of the match query in MQL.

5.4 Built-in Message Queries

To keep MQL small, certain things such as arithmetic, tuple insertion and complex formatting has been excluded. Sometimes certain queries can be cumbersome to implement if formatting the output is important.

```
(0)MILLIPEDE> locate(118,86)
Messages sent from line 118 to 86:
```

Sender					Receiver				
No	Tid	File	Line	Tag	Tid	File	Line	Tag	Size
6	262229	slave.c	118	5	262228	master.c	86	5	152
7	262230	slave.c	118	5	262228	master.c	86	5	144

Figure 5.5: The `locate(118,86)` query queries the message passing system for messages sent from line 118 and received by a (different) process at line 86.

```
define locate(sl,rl) as
begin
    print("Messages sent from line % to %:",sl,rl);
    print("\n\n\t\t\t\t\tSender\t\t\tReceiver\n");
    print("          -----");
    print("-----\n");
    print(" No\tTid \tFile \tLine \tTag \tTid \t");
    print("File \tLine \tTag \tSize \n");
    print("-----");
    print("-----\n");
    let Msgs be join Senders with Receivers;
    display
        project
            select from
                select from Msgs
                    where (SLine == sl)
                        where ( RLine == rl)
                            over (MsgNo, STID, Sfile, SLine, STag, RTID,
                                RFile, RLine, RTag, Size);
    using "%\t%\t%\t%\t%\t%\t%\t%\t%\n";
end
```

Figure 5.6: The query code for computing the `locate` query.

The status Query

Often it is useful to query the message passing system for its status, that is, obtain a list of the messages that have been delivered, the ones that are still in the system and any outstanding receive calls. An outstanding receive call is a process that is blocked in a call to `pvm_recv()`, but has not yet received any data. Such a listing can be obtained by issuing the `status` query. Figure 5.7 shows an example of using the `status` query. Millipede matches each send to a receive, and shows the file names and the line numbers of the message passing calls. For a

pvm_send the first argument is the ID of the receiver, and the second argument is the message tag. For a **pvm_recv** the first argument is the ID of the sender, and the second argument is the message tag. Both the sender and the message tag in a receive call can be specified as -1, a wild card value, which matches any sender or message tag.

(0)MILLIPEDE> status()			
Msg No.	Command	Line	File
1	pvm_bcast(262152, 0)	78	master.c
<->	pvm_recv(262150, 0)	22	slave.c
1	pvm_bcast(262151, 0)	78	master.c
<->	pvm_recv(262151, 0)	22	slave.c
2	pvm_send(262152, 11)	75	slave.c
<->	pvm_recv(262151, 11)	89	slave.c
3	pvm_send(262151, 22)	80	slave.c
<->	pvm_recv(262152, 22)	85	slave.c
4	pvm_send(262152, 11)	75	slave.c
<->	pvm_recv(262151, 11)	89	slave.c
5	pvm_send(262151, 22)	80	slave.c
<->	pvm_recv(262152, 22)	85	slave.c
6	pvm_send(262150, 5)	118	slave.c
<->	pvm_recv(262151, 5)	86	master.c
7	pvm_send(262150, 5)	118	slave.c
<->	pvm_recv(262152, 5)	86	master.c

Figure 5.7: Executing the `status` query produces a listing of matched and outstanding messages. In this example no messages or receive calls are outstanding, that is, all messages that were sent were received and the message system is “empty”. Note, message number 1 occurs twice, implying that the sending process issued a multicast or a broadcast.

This query could, with a little effort, be implemented using MQL, but we believe that the `status` query is a query that the user might use often. So to provide an easy to read output we have implemented it as a built-in query. The problem with an MQL implementation of this query is trying to list the outstanding receives and the unreceived messages in the same relation as the messages that are already delivered. Since the final relation contains information about both a sender and a receiver for each message, but the outstanding receives do not have a sender part, and the unreceived messages do not have a receiver part yet, implementing the query in MQL is not straightforward, but still possible (possibly with a different structure of the output). Thus, for convenience the `status` query is built-in.

We presented MQL code for the `match` and the `locate` queries in the previous subsection. Table 5.5 shows a list of the currently available built-in queries, and in Appendix D, a number of screen shots of actual query sessions are shown.

status	Displays all the messages that are delivered, that are still in the system, and all outstanding receive calls.
locate	Locates all messages sent between two specified line numbers.
match	Matches up packing and unpacking routines for a specific message number.
dump	Displays on a per process basis, in reverse order, all the messages ever sent.

Table 5.5: The built-in queries of the Message Debugging Module of Millipede.

The dump Query

The last query listed in Table 5.5 is for convenience, provided as a built-in. The `dump` query allows users to obtain a complete listing of all messages sent and received as well as undelivered.

Figure 5.8 shows an example of the `dump` query.

```
(0)MILLIPEDE> dump()
```

Queue: 0	Filename: master.c	Tid: 262150	
Msg.No	Send.Tid	Recv.Tag	Send.Tag
6	262151	5	5
7	262152	5	5

```
=====
```

Queue: 1	Filename: slave.c	Tid: 262152	
Msg.No	Send.Tid	Recv.Tag	Send.Tag
4	262151	11	11
2	262151	11	11
1	262150	0	0

```
=====
```

Queue: 2	Filename: slave.c	Tid: 262151	
Msg.No	Send.Tid	Recv.Tag	Send.Tag
5	262152	22	22
3	262152	22	22
1	262150	0	0

```
=====
```

Figure 5.8: The `dump` query shows all messages, both the ones that have been delivered and the ones that are still in the message passing.

5.5 Discussion

The idea of making public the information gathered by the runtime system about the messages, and representing them as relations in a database is a different approach to debugging. By allowing the user to compute relations when needed, we fulfill one of the important design goals for a debugging environment, namely computing relations on demand.

However, there are certain limitations. Since everything is represented as relations, only queries that use the query language can be performed. However, the internal structure of the

Millipede runtime system can make use of these relations as well. We have seen two examples of built-in queries that use these relations. In particular, one of these is the `status` command that uses these relations to extract information but presents it in a way that is not easily implemented as a user query.

Another important advantage of the relation/MQL part of the Message Debugging Module is that the proficient user should be able to extend Millipede himself by adding built-in queries into the runtime system if necessary.

One possibility that would address the above mentioned limitation is to extend MQL to allow the creation of user defined relations and support an explicit tuple insertion, thus making it a fully functional database language. However, this extension requires the ability to do other forms of computations, for example, arithmetic, which in turn would seriously increase the size and complexity of the query language.

An interesting technique for further investigation is the idea of message breakpoints. In sequential debugging, breakpoints are often used; by setting breakpoints the user can let the program run until the line containing the breakpoint is reached. This idea is extended to include collective breakpoints, that is, the ability to set a number of breakpoints in multiple processes. This could be further extended by abstracting away the line numbers; a receive statement might be called a number of times, but instead of using the line number as a breakpoint, the message number could be used. This would allow for greater flexibility for controlling the program execution during a debugging session. Combined with collective stepping and collective breakpoints, expressions such as `break p0:186, p2:m45` could be allowed. `p1:186` being a breakpoint in line 186 of process `p1`, and `p2:m45` being a breakpoint that is activated when message number 45 is delivered to process `p2`.

5.6 Summary

We provided a simple expressive query language that can compute a large number of relations, and we believe that this shows that providing such services in a debugger is not only possible but also very useful. In addition, we introduced interactive message debugging which allows the user to inspect and change the message content during execution.

In connection with the sequential debugging module, the interactive message debugging allows for unit testing of single parts of a system. This means, that these tools can be used in the development phase as well.

In this chapter and the previous, we introduced the two lowest levels in the multilevel debugging hierarchy. In addition, for each level we presented examples to illustrate the usefulness of the tools at these levels.

Chapter 6

Deadlock Detection and Correction

“Problems cannot be solved at the same level of awareness that created them.”

— *Albert Einstein*

In the previous chapter we demonstrated a number of techniques that are useful for locating and correcting errors in messages sent between two processes. We now focus on the next—and final—level in the multilevel debugging hierarchy: the protocol level. At this level the focus shifts to include not only messages, but the communication protocol as well.

In this chapter, and in Chapters 7 and 8, we present three different tools and techniques for performing debugging at the protocol level. Here, we present a tool for locating deadlocks, and suggest corrective measures to remove them.

In Chapter 5 we introduced a number of relations that the programmer could use in conjunction with the Millipede Query Language. These relations are the foundation for the analysis presented here and in the following chapter. Millipede extracts protocol information from these relations; in this chapter we describe how such information can be used to perform a deadlock correction analysis.

6.1 Deadlock Detection and Correction

Detecting and correcting communication errors in message passing programs is a difficult problem. Even simple communication errors are difficult to debug in a parallel environment with multiple processes exchanging large numbers of messages. Although there are visualization tools [KG96, KV97] to help users visualize the communication patterns of parallel programs, they do not directly support the detection and correction of errors based on the user’s source code.

In this chapter, we present an algorithm for correcting communication errors using delivered

and undelivered messages. The algorithm is used to suggest corrective measures for removing communication errors introduced by users as typographical errors in message passing systems, such as PVM and MPI.

This work focuses on the validity of the algorithm by proving that for a nontrivial number of errors the algorithm can suggest changes to correct these errors. The majority of this chapter is devoted to theoretically justifying the validity of using this algorithm for correcting errors. We use a counting argument to show that for less than $n/2$ errors, where n is the number of processes involved in the deadlock, the algorithm is able to identify a few potential corrections. This demonstrates the usability of the algorithm for debugging these types of communication errors.

The algorithm we present not only works for statically specified communication, but can also be applied when the sender or receiver is specified through an index into an array or by a function call. It is then the programmer's job to go back and correct the array or function, to return what the algorithm suggests. We assume that these errors are independent and infrequent. The effectiveness of the technique decreases as the number of errors increases.

6.2 Description of Problem

The basic structure of send and receive calls in PVM and MPI are as follows:

```
send(buffer, receiver_node_ID, tag)
recv(buffer, sender_node_ID, tag)
```

Mistyping the `node_ID` or `tag` value results in a message that is either undelivered, or a message that is received by the wrong process.

For example, consider the simple case of a single error, as shown in Figure 6.1. There is an error in the send call of process B in Figure 6.1. B attempts to send a message to A, but incorrectly sends it to someone else. Depending on whether the communication is synchronous or asynchronous, process B either blocks, eventually hanging the system, or terminates; in either case the result is an undelivered message in the system. Using the message queues in Millipede, it is possible to extract both undelivered and recently delivered messages from the system, which then can be used in the analysis to correct deadlocks.

6.3 The Algorithm

For the sake of simplicity we do not consider message tags or wild cards in our initial analysis, however, we return to these cases later. We start this section with a number of definitions that are used in the next section.

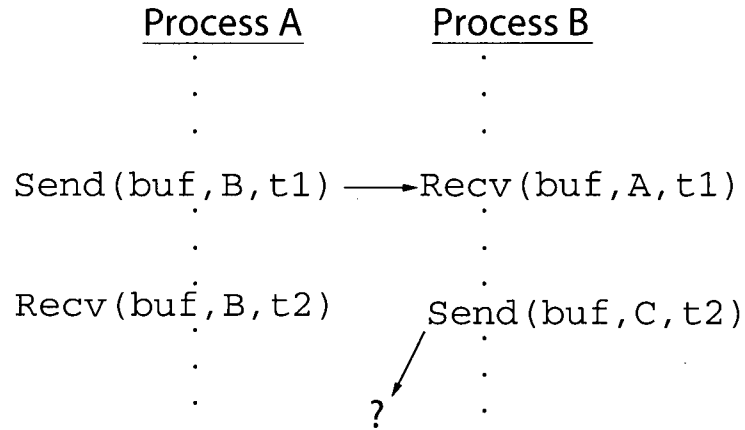


Figure 6.1: A simple error.

Definition 6.3.1 Let $S = (s_0, s_1, \dots, s_{n-1})$ be an ordered list of senders where each $s_i = (a, b)$ and a, b are integer process identifiers (ranks in MPI). Let $\mathcal{R} = (r_0, r_1, \dots, r_{n-1})$ be an ordered list of receivers where each $r_i = (a, b)$, and again, a, b are process identifiers. For $s_i = (a, b) \in S$, a is fixed as the ID of the sending process, and for $r_i = (a, b) \in \mathcal{R}$, b is fixed as the ID of the receiver.

Definition 6.3.2 A match between a sender $s_i = (a_i, b_i)$ and a receiver $r_j = (a_j, b_j)$ occurs when $(a_i = a_j) \wedge (b_i = b_j)$.

The rationale behind the algorithm is as follows: Find a set of permutations $M = \{\pi_s, \pi_r\}$ where the number of fields that need to be changed in order to obtain a system without any unmatched sends and receives is minimal. This is always possible as there is a finite number of senders and receivers, and thus a finite number of different permutations. This means that one or possibly more permutations yield a minimum distance.

Therefore, applying these changes induced by the permutations to the sends and receives in the program results in a program where all the messages are matched (assuming it does not deadlock for other causes as well). This means that all remaining undelivered messages can be delivered to a receiver and the program is deadlock free (we assume that the deadlock is not caused by insufficient buffer space in the buffering process). For an in depth analysis about deadlocks due to buffer insufficiencies please refer to Chapter 8.

It is possible to reduce the problem to a bipartite matching problem [Pre92]. The approach is as follows: Let $G = (V, E)$ be a directed graph with weighted arcs as follows:

- $V = V_s \cup V_r$, where V_s represents processes sending a message, and V_r represents processes receiving a message. An element in V is composed of the process' ID and the call point of either the send or the receive function (e.g., the line number of the call).

- E is constructed as follows:
 - For each unmatched (undelivered) messages m , do the following:
 - ★ If $m = (s, r)$ is an outstanding send (i.e., the message sent by sender s has not been received by receiver r), add arc (s, r) where $s \in V_s$ and $r \in V_r$ to E with weight 2.
 - ★ If $m = (r, s)$ is an outstanding receive, add arc (r, s) where $s \in V_s$ and $r \in V_r$ to E with weight 2.
 - Iterate backward through all successfully delivered messages (from newest to oldest) (u, v) and add arcs (u, v) and (v, u) with weight 2 to E if (u, v) or (v, u) does not already exist in E . The addition of arcs based on messages already delivered is done in the opposite order they were delivered. The order is defined using the $<$ operator on the message numbers assigned to each message by the message number process (see page 40). This ordering is chosen in an attempt to involve only processes that communicated close in time to the occurrence of the deadlock.
 - Add arcs with weight 1 to E to make G a complete bipartite graph.

Now consider the induced undirected weighted graph $\overline{G} = (\overline{V}, \overline{E})$ constructed in the following way:

- $\overline{V} = V$.
- Each pair of directed arcs (u, v) and (v, u) in E is replaced by one undirected arc (u, v) to \overline{E} with weight equal to the sum of the two arcs (u, v) and (v, u) .

\overline{G} is the complete bipartite graph $K_{n,n}$, where $n = |V|$. The maximum bipartite graph matching algorithm can be used to obtain a maximal matching in \overline{G} [CLR90]. This matching represents a system without a deadlock as \overline{G} is a complete bipartite graph, and all nodes are involved in the matching. Since all senders are matched to a receiver, no messages are undelivered.

Furthermore, this matching can be obtained by changing a minimum number of fields in the senders and receivers. This is because arcs representing actual messages and outstanding receive calls are favored with weight 2 over added arcs with weight 1. Since a maximum flow is computed, as many of the weight 2 arcs as possible are in the result, and each node on the left is matched to exactly one node on the right, thus resolving the deadlock. The following lemmas show that a maximum matching implies a minimal change.

Lemma 6.3.3 *Let G be as described above, and let f^* be a maximal matching to the induced undirected graph \overline{G} . The total number of changes, needed for the system represented by G to resolve the deadlock is $4n - |f^*|$.*

Proof: We have edges in \overline{G} with three different weights: 2, 3, and 4. We need to consider the number of changes needed for each weight.

- For edges with weight 4 no changes are needed. These edges represent delivered messages.
- Edges with weight 3 represent an arc with weight 2 and one with weight 1. The arc with weight 2 represents either a posted send or an outstanding receive. When this edge is included in the matching, a change to the source of the arcs with weight 1 is needed to make it match the source of the arc with weight 2. Thus $4 - 3 = 1$ change is needed.
- Edges with weight 2 represent two arcs with weight 1 each. These arcs are both added to make G complete. This implies that the sender did not attempt to send to the receiver, and the receiver did not attempt to receive from the sender, so a change in both sender and receiver is needed; a total of $4 - 2 = 2$ changes must be made.

This shows that the number of changes per send/receive pair is equal to 4 minus the weight of the edge. Since the sum of the edges is equal to $|f^*|$, and the total number of send/receive pairs is n , the result follows. ■

Lemma 6.3.4 *Let G be as described above. A maximal matching f^* on the corresponding graph \overline{G} determines a minimum number of changes to the parallel system represented by G to resolve the deadlock.*

Proof: Since the minimal number of changes needed is $4n - |f^*|$, the minimal number of changes occurs when the second term of this expression is maximal. This second term is the size of the matching, which means that the expression is minimal when the matching is maximal. ■

The time complexity of this max flow algorithm is $O(|\overline{E}| \cdot |f^*|)$ [CLR90] where $|f^*|$ is the size of the matching. Since $\overline{G} = K_{n,n}$, $|\overline{E}| = n^2$ and $|f^*| = n$. Therefore, the time complexity is $O(n^3)$.

Example 6.3.5 *Consider a simple example with three senders (S_1, S_2 and S_3) and three receivers (R_1, R_2 and R_3). Assume that S_1 has sent a successfully delivered message to R_1 . Now assume that S_2 is trying to send a message to R_2 , but R_2 is expecting a message from S_1 . S_3 is attempting to send to R_2 , and R_2 has posted a receive for a message from S_3 . This results in a deadlock of R_2 and R_3 , as no messages are ever sent to these processes. We can represent this scenario as a bipartite graph with senders on the left and receivers on the right. The graph labelled (a) in Figure 6.2 shows this deadlocked system.*

The algorithm requires a complete bipartite graph; the graph labelled (b) in Figure 6.2 illustrates the $K_{3,3}$, where the weights are shown as pairs (x, y) , and where x is the weight of the arc from the sender to the receiver, while y is the weight from the receiver to the sender.

We now construct the graph \overline{G} by replacing the arc pairs and adding their weights. The graph is labelled (c) in Figure 6.2. With three senders and three receivers there is a total of 6

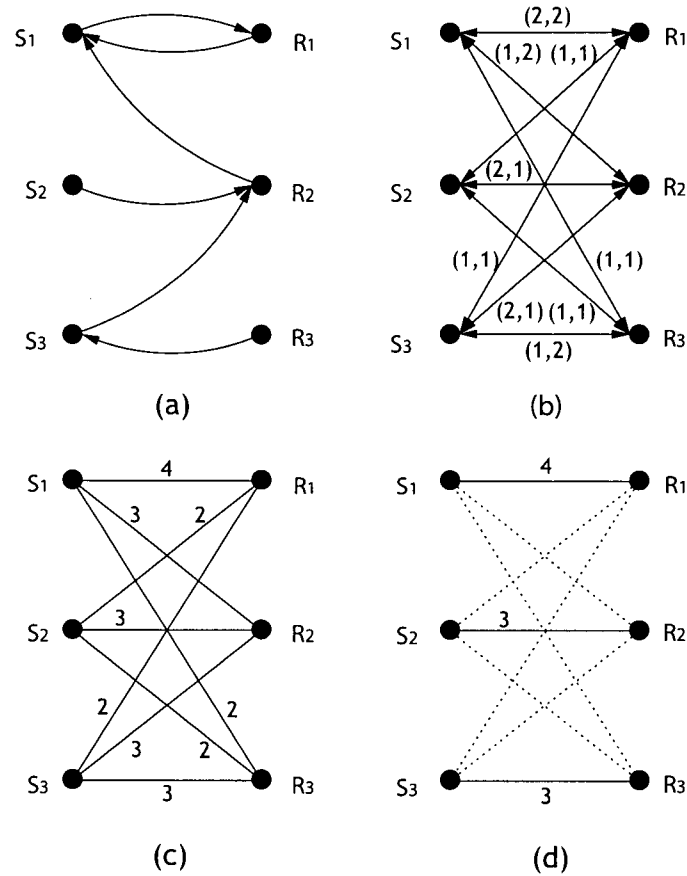


Figure 6.2: (a) depicts a deadlocked configuration: R_2 and R_3 are not receiving any messages. (b) illustrates the complete directed bipartite graph G . (c) shows the induced complete undirected bipartite graph \bar{G} . (d) gives the maximal matching in \bar{G} .

different ways to combine the senders and receivers so all senders are matched with a receiver. A maximal matching with weight 10 is given by matching S_1 with R_1 , S_2 with R_2 , and S_3 with R_3 . The graph labelled (d) in Figure 6.2 shows the result of applying the max-flow algorithm to the complete bipartite graph, where the maximal matching is shown as solid lines.

6.4 Algorithm accuracy

In this section, we evaluate the effectiveness of the techniques by showing that the algorithm does not frequently return an incorrect, or even more than one, answer. There could be more than one way to correct a deadlock with a minimum number of field changes. To do this we need to introduce a model that describes a system of senders and receivers equivalent to the one used in the previous section. In the following, let n denote the number of senders and receivers, and

k the number of errors in the system. First we define the following:

Definition 6.4.1 A communication configuration is a pair (S, \mathcal{R}) (see Definition 6.3.1). Let \mathcal{C}_n denote the set of all communication configurations with n senders and n receivers.

Definition 6.4.2 A send $s = (a, b)$ is unmatched if for $r = (c, b)$, $c \neq a$. Equivalently a receive $r = (a, b)$ is unmatched if for $s = (a, d)$, $d \neq b$. We call a communication configuration valid if it has no unmatched sends or receives. The set of valid configurations in \mathcal{C}_n is denoted by \mathcal{V}_n .

Given a configuration $(S, \mathcal{R}) = (\{s_0, \dots, s_{n-1}\}, \{r_0, \dots, r_{n-1}\})$ in \mathcal{C}_n , $s_i = (a_i, b_i)$ and $r_j = (a_j, b_j)$. The associated directed bipartite graph $G = (V, E)$ is defined by the following:

$$V = \left(\bigcup_{s_i \in S} a_i \cup \bigcup_{r_j \in \mathcal{R}} b_j \right)$$

$$E = \left(\bigcup_{s_i \in S} (a_i, b_i) \right) \cup \left(\bigcup_{r_j \in \mathcal{R}} (b_j, a_j) \right).$$

This graph is a subgraph of G . More specifically, it has the same node set, but the arc set contains only arcs of weight 2.

Example 6.4.3 For a system with two senders and two receivers, Figure 6.3 shows the only two valid configurations.

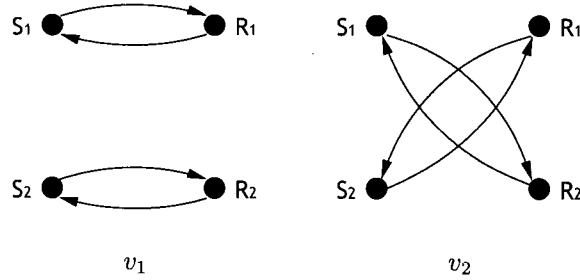


Figure 6.3: Only two of the 16 configurations in \mathcal{C}_2 are valid: all sends and receives are matched in both v_1 and v_2 , thus making them valid configurations.

Definition 6.4.4 The valid communication configuration $v \in \mathcal{V}_n$, where $s_i = r_i, \forall i : 0 \leq i < n$ is called the correct configuration. There is only one correct configuration in \mathcal{V}_n and we denote it by v_c .

The correct communication configuration is the configuration that the programmer intended to write. Lemma 6.4.15 shows that all valid configurations are equivalent, and without loss of generality, we can choose one of them to represent the correct configuration.

Definition 6.4.5 Let $v \in \mathcal{C}_n$. $\mathcal{B}(v, i)$ is the set of all communication configurations obtainable by first removing $k_1 + k_2 = j \leq i$ arcs from v (k_1 arcs oriented from \mathcal{S} to \mathcal{R} and k_2 arcs in the opposite direction), and then adding k_1 new arcs oriented from \mathcal{S} to \mathcal{R} and k_2 in the opposite direction. The set of configurations that can be obtained by removing exactly i , and then adding exactly i new arcs is denoted as $\overline{\mathcal{B}}(v, i)$. This set can be computed in the following way: $\overline{\mathcal{B}}(v, i) = \mathcal{B}(v, i) \setminus \mathcal{B}(v, i - 1)$ and $\overline{\mathcal{B}}(v, 0) = \mathcal{B}(v, 0) = \{v\}$.

Example 6.4.6 Figure 6.4 shows $\overline{\mathcal{B}}(v_1, 1)$ for v_1 from Figure 6.3.

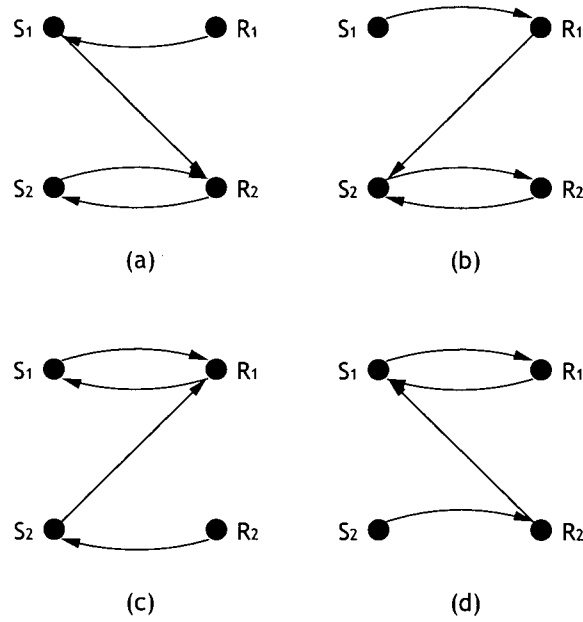


Figure 6.4: The set $\overline{\mathcal{B}}(v_1, 1)$ for configuration v_1 as shown in Figure 6.3.

Example 6.4.7 In Figure 6.5 consider an invalid configuration $v \in \mathcal{C}_n \setminus \mathcal{V}_n$. The boldface \times marks v . The boxes mark valid configurations and the rest, marked by \times , are other invalid configurations $v' \in \mathcal{C}_n \setminus \mathcal{V}_n$. The solid line marks $\mathcal{B}(v, 0)$, the dashed line $\mathcal{B}(v, 1)$, and the dotted line $\mathcal{B}(v, 2)$. In order to correct v , such that it becomes a valid configuration by making a minimal number of changes—that is, moving as few arcs as possible to transform v into a valid configuration—we choose the first valid configuration found in the series of increasing sets: $\mathcal{B}(v, 1), \mathcal{B}(v, 2), \dots$. In the example in Figure 6.5, a valid configuration is found in $\mathcal{B}(v, 1)$.

We show, that for any invalid communication configuration in \mathcal{C}_n , the probability that the first encountered valid communication configuration in the series of increasing sets $\mathcal{B}(v, 1), \mathcal{B}(v, 2), \dots$ is the correct communication configuration is high. In other words, if we introduce k errors into a

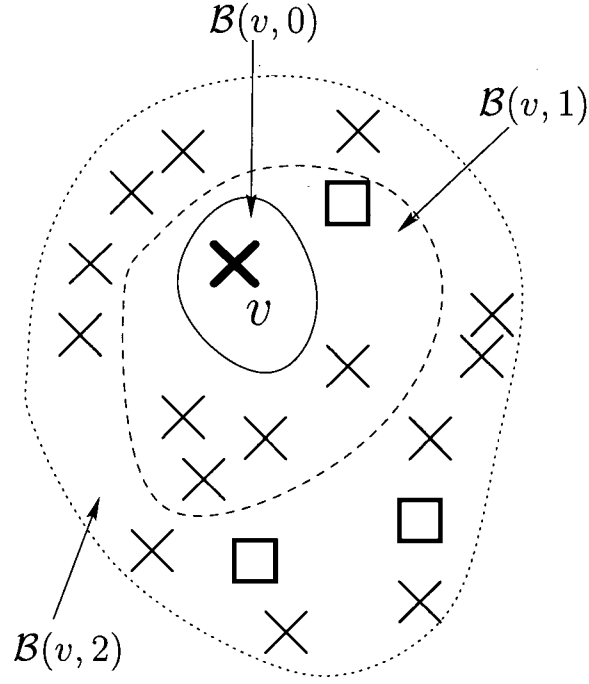


Figure 6.5: The set of elements bounded by the solid line is $B(v, 0)$ (This set always contains only one element, namely v itself). The set bounded by the dashed line (including $B(v, 0)$) is $B(v, 1)$, and $B(v, 2)$ contains all the elements.

valid communication configuration $v \in \mathcal{V}_n$, then the algorithm with a high probability will propose \times to correct the error.

Lemma 6.4.8 *The number of valid configurations in \mathcal{C}_n , that is, the size of the set \mathcal{V}_n , is $n!$.*

Proof: For a configuration to be valid, each sender must send to a distinct receiver, and this receiver must receive from this sender. If $s_i = (a_i, b_i)$, then a receiver $r_j = (a_j, b_j)$ where $b_i = b_j$ must exist. It is therefore, sufficient to determine the number of different ways to order n senders. There are $n!$ such ways. ■

In the following, we consider the set of configurations on n senders and n receivers \mathcal{C}_n , and the corresponding set of valid configurations $\mathcal{V}_n = \{v_0, \dots, v_{n!-1}\}$. Let $k \leq n/2$ to be the number of errors in the system.

Example 6.4.9 *Consider a system with 1 error; we need to consider the configurations obtainable by introducing one error to all $v_i \in \mathcal{V}_n$. This is a set of sets like this:*

$$\mathcal{B}_1 = \{B(v_0, 1), B(v_1, 1), \dots, B(v_{n!-1}, 1)\}.$$

If we know for every system with one error that the following is true:

$$\bigcap_{i=0}^{n!-1} \mathcal{B}(v_i, 1) = \bigcap_{b \in \mathcal{B}_1} b = \emptyset,$$

then $\mathcal{B}(v, 1)$ contains only one valid configuration, which must be the correct configuration.

Consider Figure 6.6. If two errors are introduced into the configuration v_1 , then we have a configuration that is in the intersection of $\mathcal{B}(v_1, 2)$ and $\mathcal{B}(v_2, 2)$, and this non valid configuration can be corrected to either v_1 or v_2 by moving two arcs. Since there are two valid configurations, either may be the correct configuration.

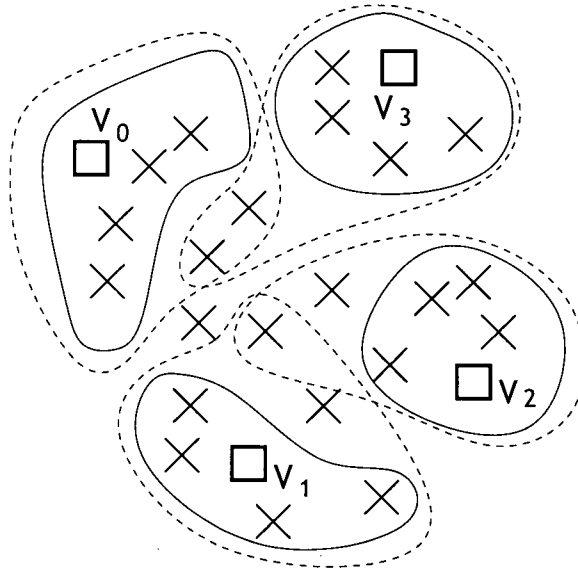


Figure 6.6: $\bigcap_i \mathcal{B}(v_i, 1) = \emptyset$, which means the correct configuration will always be found if only one error is present in the system. However, $\bigcap_i \mathcal{B}(v_i, 2) \neq \emptyset$, which means that if two errors are present, then a wrong valid configuration might be suggested as the correction to the deadlock.

In order to argue that the configuration obtained by moving a minimal number of arcs in any invalid configuration is the correct valid configuration, we must therefore show the following for all $v_i, v_j \in \mathcal{V}_n, i \neq j$:

$$\frac{|\mathcal{B}(v_i, e) \cap \mathcal{B}(v_j, e)|}{|\mathcal{B}(v_i, e)|} \quad \text{and} \quad \frac{|\mathcal{B}(v_i, e) \cap \mathcal{B}(v_j, e)|}{|\mathcal{B}(v_j, e)|} \quad \text{are small } \forall e \leq k, \quad (6.1)$$

where small means an acceptably low fraction of wrongly proposed corrections. This is equivalent to showing the following:

$$\frac{|\mathcal{B}(v_c, e) \cap \mathcal{B}(v_i, e)|}{|\mathcal{B}(v_c, e)|} \quad \text{is small } \forall e \leq k, \forall v_i \in \mathcal{V}_n \setminus \{v_c\}, \quad (6.2)$$

The following lemmas are needed to prove Equation 6.2.

Lemma 6.4.11 *The number of configurations that can be obtained by moving i or less arcs in v , denoted by $|B(v, i)|$, is as follows:*

$$\sum_{j=0}^i \binom{2n}{j} (n-1)^j$$

Proof:

$$|B(v, i)| = \left| \bigcup_{j=0}^i \bar{B}(v, j) \right| \quad (6.3)$$

$$= \sum_{j=0}^i |\bar{B}(v, j)| \quad (6.4)$$

$$= \sum_{j=0}^i \binom{2n}{j} (n-1)^j, \quad (6.5)$$

where Equation 6.3 follows from Definition 6.4.5, Equation 6.4 follows from the fact that $\bar{B}(v, j) \cap \bar{B}(v, i) = \emptyset$ if $i \neq j$, and Equation 6.5 follows from the observation that $\bar{B}(v, j)$ is the set of configurations where we move exactly j arcs: we must choose j out of the $2n$ arcs to move. Each of these arcs can be moved to any of the either n senders or receivers except for the one it pointed to originally, leaving $n-1$ choices. This is done for a total number of j times. ■

Example 6.4.12 Table 6.1 shows a few examples of $|B(v, e)|$. Recall that $B(v, e)$ is the set of configurations that can be obtained by moving a maximum of e different arcs in configuration v .

	errors (e)						
n	0	1	2	3	4	5	6
2	1	5	11	15	16		
3	1	13	73	233	573	665	729

Table 6.1: $|B(v, e)|$ is the size of the sets that can be obtained by moving e or less arcs in a valid configuration v .

Definition 6.4.13 *The distance between two valid configurations in \mathcal{V}_n , denoted as $d(v_i, v_j)$, is defined as follows:*

$$d(v_i, v_j) = \sum_{l=0}^{2n} [v_{i_l} \neq v_{j_l}],$$

where $v_i = v_{i_1} v_{i_2} \dots v_{i_{2n}}$, $v_j = v_{j_1} v_{j_2} \dots v_{j_{2n}}$, and

$$[S] = \begin{cases} 1 & : \text{if } S \text{ is true.} \\ 0 & : \text{otherwise.} \end{cases}$$

and S is a relational expression.

Example 6.4.14 The valid configurations in \mathcal{C}_3 are as follows

$$\mathcal{V}_3 = \{001122, 110022, 002211, 220011, 221100, 210210\}.$$

Table 6.2 shows the distances between these different valid configurations.

	001122	110022	002211	220011	221100	210210
001122	0	4	4	6	4	6
110022	4	0	6	4	6	4
002211	4	6	0	4	6	4
220011	6	4	4	0	4	6
221100	4	6	6	4	0	4
210210	6	4	4	6	4	0

Table 6.2: Distances between valid configurations in $\mathcal{V}_3 \subset \mathcal{C}_3$. The maximum distance between configurations in \mathcal{V}_3 is 6, and the number of valid configurations is also 6.

Lemma 6.4.15 For any system \mathcal{C}_n , a distance k , and a valid configuration $v \in \mathcal{V}_n \subset \mathcal{C}_n$, the number of valid configurations in \mathcal{V}_n with distance k does not depend on the choice of v .

Proof: Permutations are automorphisms. ■

Lemma 6.4.16 The possible distances between valid configurations in $\mathcal{V}_n \subset \mathcal{C}_n$ are $4, 6, \dots, 2n - 2, 2n$.

Proof: A necessary condition for a configuration to be valid is that $\{s_1, \dots, s_n\} = \{r_1, \dots, r_n\} = \{0, \dots, n - 1\}$. Since all valid configurations are equivalent, consider $v_c = s_1 r_1 s_2 r_2, \dots, s_n r_n$. A minimum of two send/receive pairs must be switched to obtain a different valid configuration. This gives a minimum distance of 4. Now choose two send/receive pairs a, b to switch. There are three cases to consider:

1. Both pairs are of the form $s_i r_i = ii$, which means that either they have not been switched before, or that they have been switched back to their original state. When these pairs are switched, the distance increases by 4.
2. One of the pairs, say a , is of the form $s_i r_i = ii$, and the other one, b , is not. When a and b are switched a contributes distance 2 to the total distance, and b already contributed distance 2, so the total distance only increases by 2.
3. Neither a nor b are of the form $s_i r_i = ii$. Neither contribute further to the total distance by being switched.

■

Definition 6.4.17 Let $\mathcal{D}(v, m)$ be the set of valid configurations exactly distance m from the valid configuration v , that is, the set $\overline{B}(v, m) \cap \mathcal{V}_n$.

Lemma 6.4.18 The size of $\mathcal{D}(v, m)$ for $m = 2k$ is the following:

$$|\mathcal{D}(v, 2k)| = \binom{n}{k} c_k$$

where

$$c_k = k! \sum_{i=0}^k \frac{(-1)^i}{i!}$$

Proof: Since $\mathcal{D}(v, 2k)$ is the set of configurations exactly distance m away from v , we start by choosing k of the n send/receive pairs to move; this can be done in $\binom{n}{k}$ different ways. Since we are only interested in permutations that result in configurations exactly distance $2k$ away, we must multiply by the number of permutations of k elements that permute all k elements. We need to determine the number of permutations of k elements which have no fixed points. This problem was first proposed by the French mathematician Pierre Rémond de Montmort in 1713 [Mon13]. The answer is the alternating sum $k! \sum_{i=0}^k \frac{(-1)^i}{i!}$. This series of numbers is known as *recontres numbers* or *derangements*. ■

Example 6.4.19 c_i is a fast growing series. Table 6.3 illustrates this by computing the first 11 values of c_i . It should be clear, that the rapid growth in the number of valid configurations makes correcting systems with a large number of errors virtually impossible.

i	c_i
0	1
1	0
2	1
3	2
4	9
5	44
6	265
7	1,854
8	14,833
9	133,496
10	1,334,961

Table 6.3: The rate of growth of c_i .

Example 6.4.20 Table 6.4 gives an example of the number of valid configurations at given distances from another valid configuration. The columns for distance 0 and 2 are omitted as they are always 1 and 0, respectively.

Number of valid configurations at different distances.								
n	4	6	8	10	12	14	16	18
2	1							
3	3	2						
4	6	8	9					
5	10	20	45	44				
6	15	40	135	264	265			
7	21	70	315	924	1,855	1,854		
8	28	112	630	2,464	7,420	14,832	14,833	
9	36	168	1,134	5,544	22,260	66,744	133,497	133,496
10	45	240	1,890	11,088	55,650	222,480	667,485	1,334,960

Table 6.4: The number of valid configurations at different distances in \mathcal{V}_n . Distances 0 and 2 are omitted as they are always 1 and 0, respectively. Note, the last number in each row corresponds to c_i for $i = n$.

Consider the following two configurations: $v_1 = 001122$ $v_2 = 002211$. These two configurations differ in the last four positions, thus having a distance of four. To compute the intersection $\mathcal{B}(v_1, 2) \cap \mathcal{B}(v_2, 2)$, we must find the configurations that can be reached from both v_1 and v_2 by changing at most two positions in each. Since the distance between the two configurations is four, and we can change at most two positions in each configuration, it follows that we must change exactly two in each. Choose two fields in v_1 , say v_{1_i} and v_{1_j} . Change these two positions to have the values of v_{2_i} and v_{2_j} , and obtain v'_1 . We know that $d(v'_1, v_2) = 2$. Now change the two positions in v_2 that differ from v'_1 , say v_{2_l} and v_{2_m} to have the values of $v_{1_l} = v'_{1_l}$ and $v_{1_m} = v'_{1_m}$, and obtain v'_2 . We now know that $d(v'_1, v'_2) = 0$. The original distance is four and we must change two fields in each configuration. The number of different ways this can be done is $\binom{4}{2} = 6$. The six configurations are as follows:

$$11\underline{33}\overline{33}, \quad 11\underline{22}\overline{22}, \quad 11\underline{23}\overline{23}, \quad 11\underline{32}\overline{33}, \quad 11\underline{32}\overline{32}, \quad 11\underline{23}\overline{32}.$$

The underlined positions are the fields changed in v_1 and the overlined fields are the ones changed in v_2 . According to Lemma 6.4.15, all valid configurations are equivalent. Therefore, we can simply study the properties of the correct valid configuration v_c of \mathcal{V}_n .

Example 6.4.21 Figure 6.8 illustrates the overlapping sets, the intersections can easily be seen by comparing the coloured areas with the solid lines. Table 6.5 shows the intersection sets for various values of k and k' . Note, the size of the intersecting sets shown in Table 6.5 can be found on the diagonal in Table 6.6 from the lower left to the upper right.

Table 6.6 shows the sizes of the various intersections depending on different values of k and k' . All these values can be read from Figure 6.8. The upper left part is mainly zeros because

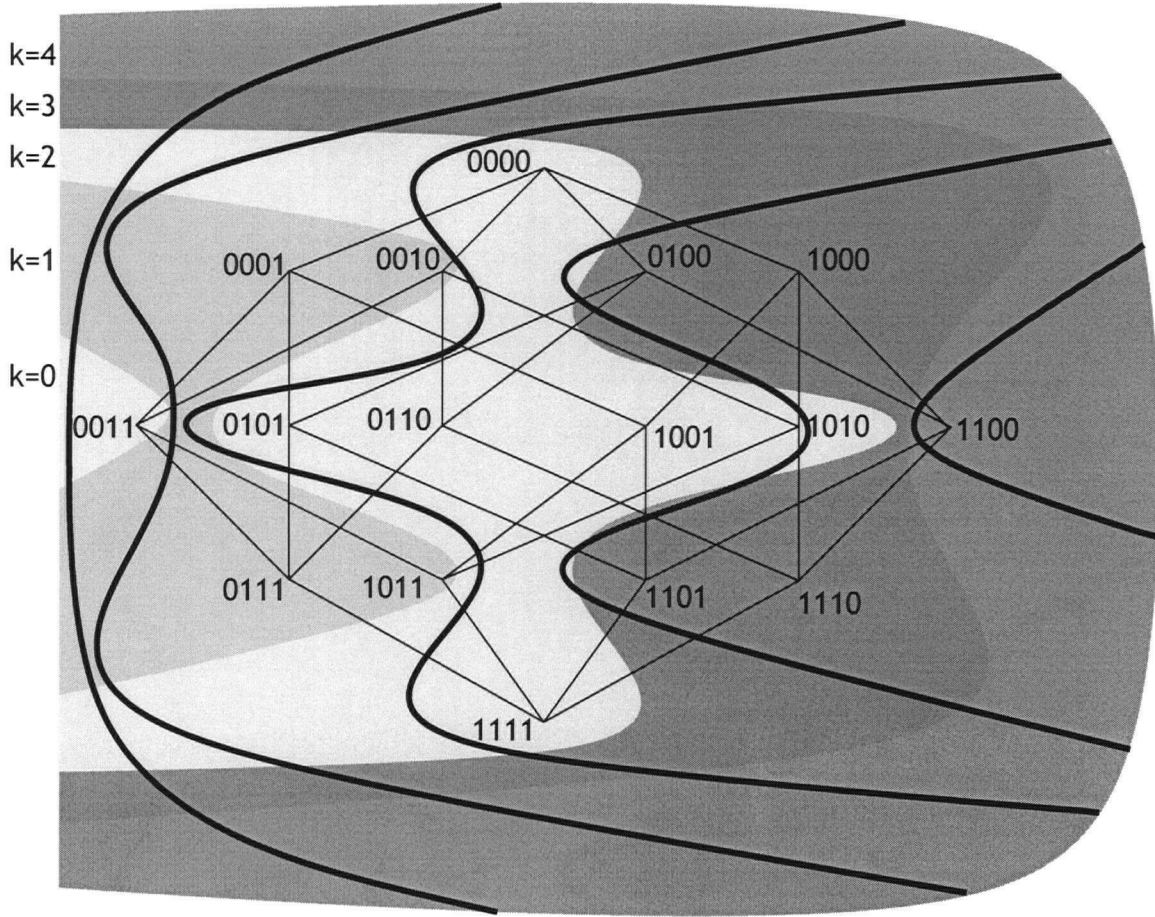


Figure 6.8: An illustration of which configurations are in which intersections when considering all the valid configurations in the \mathcal{C}_2 system.

Constraint	$\mathcal{B}(1122, k) \cap \mathcal{B}(2211, k')$
$(k \leq 4) \wedge (k' \leq 0)$	2211
$(k \leq 3) \wedge (k' \leq 1)$	1211, 2111, 2212, 2221
$(k \leq 2) \wedge (k' \leq 2)$	2222, 1212, 1221, 2112, 2121, 1111
$(k \leq 1) \wedge (k' \leq 3)$	1112, 1121, 1222, 2122
$(k \leq 0) \wedge (k' \leq 4)$	1122

Table 6.5: An example of the intersection $\mathcal{B}(1122, k) \cap \mathcal{B}(2211, k')$, that is, the configurations that can be transformed into 1122 by moving at most k arcs, and into 2211 by moving at most k' arcs.

$d(1122, 2211) = 4$. For example, it is impossible to find a configuration that can be turned into 1122 by moving two arcs, and into 2211 by moving one.

	$k' \leq 0$	$k' \leq 1$	$k' \leq 2$	$k' \leq 3$	$k' \leq 4$
$k \leq 0$	0	0	0	0	1
$k \leq 1$	0	0	0	4	5
$k \leq 2$	0	0	6	10	11
$k \leq 3$	0	4	10	14	15
$k \leq 4$	1	5	11	15	16

Table 6.6: The size of $B(1122, k) \cap B(2211, k')$ for various values of k and k' .

We can now determine the number of elements in the intersections of the B sets in 6.2.

Theorem 6.4.22 *Let e be the number of errors in a communication system C_n . The number of configurations with e errors for which the maximum matching either suggests a wrong valid configuration or a set of valid configurations where the correct one is included is as follows:*

$$\left| \bigcup_{v_j \in \mathcal{V}_n} B(v_c, e) \cap B(v_j, e) \right| \leq \sum_{i=2}^e \binom{n}{i} c_i \sum_{b=0}^e \sum_{a=\max\{b, e-b\}}^e \sum_{c_o=0}^{\frac{a+b+2i}{2}} \mathcal{O}(i, a, b, c_o) \quad (6.6)$$

where

$$\mathcal{O}(i, a, b, c_o) = \binom{2i}{c_x} \binom{2i - c_x}{c_y} (n-2)^{c_{xy}} \binom{2(n-i)}{c_o} (n-1)^{c_o}$$

and

$$\begin{aligned} c_{xy} &= (a - c_o) + (b - c_o) - 2i \\ c_x &= 2i - a + c_o \\ c_y &= 2i - b + c_o \end{aligned}$$

under the constraints $(c_{xy} \geq 0 \wedge c_x \geq 0 \wedge c_y \geq 0)$.

Proof: We wish to compute the total number of configurations in the intersections of the set $B(v_c, e)$ with the sets $B(v_j, e)$, and then compare that with the total number of configurations in the set $B(v_c, e)$ (cf. Equation 6.1). We start with Equation 6.7.

$$\left| \bigcup_{v_j \in \mathcal{V}} B(v_c, e) \cap B(v_j, e) \right|. \quad (6.7)$$

Equation 6.8 follows from Equation 6.7; since the maximum distance between a configuration $v \in B(v_c, e)$ and a configuration $v' \in B(v_i, e)$ for $v_i \in \mathcal{V}_n \setminus \{v_c\}$ is $2e$, we can thus restrict ourselves to consider valid configurations at distances $4, 6, \dots, 2e$ from v_c . $\mathcal{D}(v_c, i)$ is the set of valid configurations exactly distance i from v_c .

$$\left| \bigcup_{i \in \{4, \dots, 2e\}} \bigcup_{v_j \in \mathcal{D}(v_c, i)} B(v_c, e) \cap B(v_j, e) \right| \quad (6.8)$$

From Definition 6.4.5 we get the following:

$$\mathcal{B}(v_c, e) = \bigcup_{a=0}^e \overline{\mathcal{B}}(v_c, a) \quad \text{and} \quad \mathcal{B}(v_j, e) = \bigcup_{b=0}^e \overline{\mathcal{B}}(v_j, b).$$

This gives the following result in Equation 6.9:

$$\left| \bigcup_{i \in \{4, \dots, 2e\}} \bigcup_{v_j \in \mathcal{D}(v_c, i)} \left(\bigcup_{a=1}^e \overline{\mathcal{B}}(v_c, a) \right) \cap \left(\bigcup_{b=1}^e \overline{\mathcal{B}}(v_j, b) \right) \right|. \quad (6.9)$$

Since the \mathcal{D} sets are disjoint, we can exchange the unions with sums, and arrive at Equation 6.10:

$$\sum_{i=2}^e \sum_{v_j \in \mathcal{D}(v_c, 2i)} \left| \left(\bigcup_{a=1}^e \overline{\mathcal{B}}(v_c, a) \right) \cap \left(\bigcup_{b=1}^e \overline{\mathcal{B}}(v_j, b) \right) \right| \quad (6.10)$$

For a configuration $v \in \mathcal{B}(v_c, a) \cap \mathcal{B}(v_j, b)$ with $b = d(v_j, v)$ and $a = d(v_c, v)$, if $a < b$, then v_c will be reported as the correct communication configuration. Since we are counting the valid communication configurations that are incorrect corrections, we only consider cases where $a \geq b$. Additionally, if $a + b < e$ then the intersection between $\mathcal{B}(v_c, a)$ and $\mathcal{B}(v_j, b)$ is empty. These three observations combined, yield Equation 6.11.

$$\sum_{i=2}^e \sum_{v_j \in \mathcal{D}(v_c, 2i)} \left| \bigcup_{b=0}^e \bigcup_{a=\max\{b, e-b\}}^e \overline{\mathcal{B}}(v_c, a) \cap \overline{\mathcal{B}}(v_j, b) \right| \quad (6.11)$$

Finally, by summing the sizes of the intersections $\overline{\mathcal{B}}(v_c, a) \cap \overline{\mathcal{B}}(v_j, b)$, we obtain the quadruple sum shown in Equation 6.12, which is an upper bound for Equation 6.7.

$$\sum_{i=2}^e \sum_{v_j \in \mathcal{D}(v_c, 2i)} \sum_{b=0}^e \sum_{a=\max\{b, e-b\}}^e |\overline{\mathcal{B}}(v_c, a) \cap \overline{\mathcal{B}}(v_j, b)| \quad (6.12)$$

We now calculate the value of

$$|\overline{\mathcal{B}}(v_c, a) \cap \overline{\mathcal{B}}(v_j, b)|$$

for values a and b where $j = 2i$. Let $x = v_c = x_1 x_2 \dots x_{2n}$ and $y = v_j = y_1 y_2 \dots y_{2n}$ with $d(x, y) = j = 2i$. We want to find configurations $z = v = z_1 z_2 \dots z_{2n}$ such that $d(x, z) = a$ and $d(y, z) = b$. Assume, without loss of generality, that $x_k = y_k$ for $j+1 \leq k \leq 2n$ such that we get the following:

$$\begin{array}{rcl} & \overbrace{}^j & \overbrace{\phantom{x_{j+1} \dots x_{2n}}}^{2n-j} \\ x & = & x_1 \quad x_2 \quad \dots \quad x_j \quad \bigg| \quad x_{j+1} \quad \dots \quad x_{2n} \\ y & = & z_1 \quad z_2 \quad \dots \quad z_j \quad \bigg| \quad z_{j+1} \quad \dots \quad z_{2n} \\ z & = & y_1 \quad y_2 \quad \dots \quad y_j \quad \bigg| \quad x_{j+1} \quad \dots \quad x_{2n} \end{array}$$

Now define the following:

$$\begin{aligned}
 C_y &= \{z_k : 1 \leq k \leq j : z_k = x_k \wedge z_k \neq y_k\} \\
 C_x &= \{z_k : 1 \leq k \leq j : z_k \neq x_k \wedge z_k = y_k\} \\
 C_{xy} &= \{z_k : 1 \leq k \leq j : z_k \neq x_k \wedge z_k \neq y_k\} \\
 C_o &= \{z_k : j+1 \leq k \leq 2n : z_k \neq x_k \wedge z_k \neq y_k\} \\
 c_x &= |C_x| \\
 c_y &= |C_y| \\
 c_{xy} &= |C_{xy}| \\
 c_o &= |C_o|
 \end{aligned}$$

A z -configuration must satisfy the following:

$$\begin{pmatrix} a \\ b \end{pmatrix} = c_{xy} \begin{pmatrix} 1 \\ 1 \end{pmatrix} + c_x \begin{pmatrix} 1 \\ 0 \end{pmatrix} + c_y \begin{pmatrix} 0 \\ 1 \end{pmatrix} + c_o \begin{pmatrix} 1 \\ 1 \end{pmatrix},$$

since $d(x, z) = a$ and $d(y, z) = b$. This results in the following 3 equations:

$$\begin{aligned}
 a &= c_{xy} + c_x + c_o \\
 b &= c_{xy} + c_y + c_o \\
 j &= c_{xy} + c_x + c_y
 \end{aligned} \tag{6.13}$$

The last equation follows from the fact that $x_k \neq y_k$ for $1 \leq k \leq j$. Setting $a' = a - c_o$ and $b' = b - c_o$ we obtain 3 equations with 3 unknowns represented by the following matrix equation:

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} c_{xy} \\ c_x \\ c_y \end{pmatrix} = \begin{pmatrix} a' \\ b' \\ j \end{pmatrix}$$

By inverting the matrix we can compute values for c_{xy} , c_x , and c_y in the following way:

$$\begin{pmatrix} c_{xy} \\ c_x \\ c_y \end{pmatrix} = \begin{pmatrix} 1 & 1 & -1 \\ 0 & -1 & 1 \\ -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} a' \\ b' \\ j \end{pmatrix} = \begin{pmatrix} a' + b' - j \\ j - b' \\ j - a' \end{pmatrix} = \begin{pmatrix} a + b - j - 2c_o \\ j - b + c_o \\ j - a + c_o \end{pmatrix}$$

where the number of fields where z differs from x , but not from y , is c_x ; the number of fields where z differs from y , but not x , is c_y ; and the number of fields where z differs from both x and y is c_{xy} .

We now look at the different ways of choosing fields in z that satisfy these constraints. Of the j fields where $x_k \neq y_k$, we must choose c_x where z differs from x but not y . Of the remaining $(j - c_x)$, we must choose c_y fields. The rest of the c_{xy} fields are prechosen. Of the $(2n - j)$ fields where x and y do not differ, we must choose c_o fields where z differs.

The values of the C_x and C_y fields are chosen to be the values of the opposite string, that is, for the c_x chosen fields the value is that of y and vice versa, for the c_y chosen ones. The remaining C_{xy} fields can take any value except those of the corresponding fields in x and y , which $(n-2)$ choices for these c_{xy} fields. The C_o fields can take any value except that of the corresponding fields of x and y , which are the same. There are $(n-1)$ different choices for these values.

By substituting these values in Equation 6.12 on page 79, and summing over all valid values of c_o (i.e., values for c_o that produce nonnegative values for c_{xy}, c_x and c_y) we get the number of configurations with distance a to x , and distance b to y . The constraint $(c_x \geq 0 \wedge c_y \geq 0 \wedge c_{xy} \geq 0)$ assures valid values of c_{xy}, c_x and c_y . By subtracting the two last equations of 6.13 from the first, we get $c_o \leq (a + b + 2i)/2$. Taking this into consideration, we arrive at the following result, as stated in Equation 6.6:

$$\begin{aligned}
 & \left| \bigcup_{v_j \in \mathcal{V}_n} \mathcal{B}(v_c, e) \cap \mathcal{B}(v_j, e) \right| \\
 & \leq \sum_{i=2}^e \sum_{v_j \in \mathcal{D}(v_c, 2i)} \sum_{b=0}^e \sum_{a=\max\{b, e-b\}}^e |\overline{\mathcal{B}}(v_c, a) \cap \overline{\mathcal{B}}(v_j, b)| \\
 & = \sum_{i=2}^e |\mathcal{D}(v_c, 2i)| \sum_{b=0}^e \sum_{a=\max\{b, e-b\}}^e \sum_{c_o=0}^{\frac{a+b+2i}{2}} \mathcal{O}(i, a, b, c_o) \\
 & = \sum_{i=2}^e \binom{n}{i} c_i \sum_{b=0}^e \sum_{a=\max\{b, e-b\}}^e \sum_{c_o=0}^{\frac{a+b+2i}{2}} \mathcal{O}(i, a, b, c_o) \tag{6.14}
 \end{aligned}$$

where c_i are the constants from Lemma 6.4.18, and

$$\mathcal{O}(i, a, b, c_o) = \binom{2i}{c_x} \binom{2i - c_x}{c_y} (n-2)^{c_{xy}} \binom{2(n-i)}{c_o} (n-1)^{c_o}$$

and c_{xy}, c_x, c_y are given as follows:

$$\begin{aligned}
 c_{xy} &= (a - c_o) + (b - c_o) - 2i \\
 c_x &= 2i - b + c_o \\
 c_y &= 2i - a + c_o
 \end{aligned}$$

under the constraints $(c_{xy} \geq 0 \wedge c_x \geq 0 \wedge c_y \geq 0)$.

■

Using Equation 6.14 we can now compute an upper bound for the fraction in Equation 6.2. Figure 6.9 shows the estimated failure rate for the algorithm. An ambiguous correction is when more than one valid configuration is at the minimum distance.

	Number of errors (e)									
	$e = 1$		$e = 2$		$e = 3$		$e = 4$		$e = 5$	
n	W	A	W	A	W	A	W	A	W	A
2	0.00	0.00	0.00	54.55						
3	0.00	0.00	0.00	24.66						
4	0.00	0.00	0.00	13.00						
5	0.00	0.00	0.00	7.38	4.74	32.93				
6	0.00	0.00	0.00	5.26	2.67	23.86				
7	0.00	0.00	0.00	3.75	1.64	17.91	7.76	53.79		
8	0.00	0.00	0.00	2.80	1.07	13.88	5.10	42.08		
9	0.00	0.00	0.00	2.17	0.74	11.05	3.51	33.67	11.33	88.85
10	0.00	0.00	0.00	1.73	0.53	8.99	2.52	27.49	8.02	71.39

Figure 6.9: The failure rate for the algorithm in percents—incorrect suggested corrections (labelled W) and ambiguous corrections (labelled A).

6.5 Message tags

We now consider message passing that includes message tags. We do not formally analyze this case, but argue that the chances of the algorithm being able to predict the correct solution increases proportionally to the number of different message tags used.

We do not have a polynomial time algorithm for the case where tags are considered, but the desired permutations can be obtained by computing a Hamming distance between all possible combinations of permutations of senders and receivers, and choosing the one or ones that give the smallest Hamming distance. This is an exhaustive search that is only feasible for small values of n . This algorithm has time complexity $O(n!)$.

Figure 6.10 is a copy of Figure 6.3, where we have introduced message tags. The S_1 and R_1 both send/receive with tag 11, and S_2 and R_2 both send/receive with tag 22. It is obvious that v_2 is no longer a valid configuration as there is a tag mismatch between S_1 and R_2 , and between S_2 and R_1 . In fact, v_2 is now a configuration with at least 2 errors.

By introducing message tags into a communication system, and by choosing them carefully, that is, in a meaningful way with respect to the message they are associated with, the risk of the algorithm predicting a wrong solution is greatly reduced.

As an example, consider C_2 . As seen in the previous example the 54.55% ambiguity rate has disappeared as v_2 is no longer a valid configuration. This holds true if two errors are introduced in the sender or receiver IDs. The correct valid configuration v_1 is distance 2 away, where v_2 is distance 4 away. Similarly, if two errors are introduced into the tags, the correct valid configuration is distance 2 away, whereas the wrong valid configuration is distance 4 away.

A wild card or an 'any' value is a special value that matches any other value. These are often used when a receiver does not know the identity of the sender or the tag of the package. They

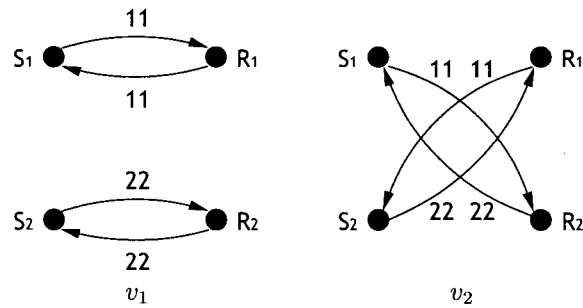


Figure 6.10: Introducing message tags.

are often used for dynamic communication in client/server type applications or for convenience, instead of the process ID. When introducing wild cards into a communication system, the degree of freedom with respect to field values increases. This significantly decreases the success rate of an algorithm, such as the one presented.

6.6 Summary

In this chapter, we presented an algorithm that proposes changes in message passing systems that have deadlocked due to a small number of typographical errors. If a small number of errors occur in an otherwise working message passing system, then we can correct these errors with a high probability.

Many programmers make extensive use of wild cards in receive calls. This does not only increase the risk of a message being accepted by a receiver that is not supposed to receive it, but also complicates the problem of discovering the source of the error. In contrast, by carefully choosing message tags, and by associating different tags with different types of communication, the risk of wrong sends going through is substantially reduced. Furthermore, the ability to predict the correct communication configuration is greatly increased.

We do not have a polynomial time algorithm for the case where message tags are considered; we believe that the problem can be reduced to a 3 dimensional matching problem, which is NP-complete. Introducing message tags make the problem more complex but on the other hand reduces the size of the overlapping B sets. The $O(n!)$ algorithm can easily be implemented such that message tags are taken into account.

Chapter 7

Protocol Conformance Checking

“The pure and simple truth is rarely pure and never simple”

- *Oscar Wilde*

In the previous chapter, we described the Deadlock Detection and Correction Module, which is the first tool at the protocol level of the Millipede multilevel debugger. In this chapter, we investigate a technique referred to as Protocol Constraint Conformance Checking. A protocol constraint specification is an assertion-like specification of a protocol's behaviour that specifies a number of constraints that the protocol must conform to when executed. It is not a verification tool like SMV [CLM89, McM92] or FDR [For], nor is it directly comparable to assert statements in C, but rather a technique that allows the user to automate checking the behaviour of the protocol of a running parallel system by writing a specification file containing a number of constraints. These constraints are checked against the messages at runtime. We present the Protocol Constraint Specification Language (PCSL) and a Millipede tool Millipede Online Protocol Error Detection (MOPED) which executes the constraint conformance checking at runtime.

7.1 Between Testing and Verification

The idea behind constraint conformance checking is to allow the user to write a specification of the behaviour of the protocol, and then, using information about actual messages, automatically check that the messages satisfy the constraints. It is important to distinguish the protocol constraint specification from the well known concept of constraint programming [Lel88]. A program written in a constraint programming language is a set of equations that are given to a constraint-satisfaction system which in turn, returns the values satisfying the constraints. Our approach does not generate a list of messages that satisfy the constraint system, but rather, using

message information, checks that the constraints are valid. In later sections we explain in detail how constraints are instantiated using message information and checked.

We now argue why protocol constraints are useful in the debugging and development cycles of a parallel program. The communication protocol of a parallel message passing program starts as a specification; this specification can be anything from written prose to a detailed CSP description. One of the goals of such a specification is to serve as a starting point for the implementation of the protocol using, for example, C and PVM. A second goal is to serve as a specification that can be used for testing purposes.

A number of different paths can be taken from the specification to the actual running implementation. The most straightforward one is to simply implement the protocol. This leaves the user with the daunting task of having to test a protocol implementation that might contain errors and deadlocks. If the specification is more rigorous than just plain English, perhaps written in some verification language, using a verification tool to check that the protocol does not have deadlocks, livelocks, and race conditions is a natural choice. In Section 7.2, we briefly describe some of the advantages and disadvantages of this technique. Once a protocol specification is verified, it must still be implemented in the target language and message passing system. Errors may be introduced into this implementation as well; this means that testing the implementation is still necessary.

Whenever the translation of the protocol from specification to implementation is done by hand, as with any implementation, the risk of introducing errors exists. We believe this is particularly true when the source (specification or verified protocol in some verification language) and the target (e.g., a C program) domains differ greatly. One of the closest relationships between a specification language and a programming language is between CSP [Hoa78] and Occam [May83], but even here the difference is still substantial.

No matter which approach is taken, a substantial amount of testing is necessary. This is where protocol constraints can help; as a mixture of asserts and constraints, we allow the user to specify relationships between processes (through a constraint-like specification), and have the system check that the messages conform to the constraints through assert-like checks.

We believe that the collection of constraint specifications in one file (rather than assert like statements associated with each message passing call) gives the user a faster and more complete overview of the entire constraint specification, as well as more tightly coupling the sending process with the receiving process.

Since program development is often an iterative task, another main goal of the constraint system is to provide the ability to use it in connection with such a program development strategy. This means that the initial specification can be very general, and as the implementation becomes more complex, or as discoveries are made about the protocol, the specification can be refined as

well.

Naturally, there is no guarantee that the transcription of the protocol specification to the constraint specification language is correct, but this language is not large nor complicated. Whether a protocol is formally verified or not, protocol constraints can aid in testing the implementation of a protocol. Many users are not familiar with CSP or other complex specification languages or verification tools. If this is the case, protocol verification is virtually impossible. However, though not constituting verification in the typical sense of the word, constraints enable users unfamiliar with verification tools to write simple protocol constraint specifications as the program is being developed, and MOPED checks messages against these when the program runs. We believe that this can assist the user in correcting errors in the implementation that might otherwise be difficult to find.

7.2 Protocol Checking and Verification

For completeness, we include some information on protocol verification in this chapter. A common denominator for the tools mentioned in this section is the ability to check and verify protocols and perform model checking. Being able to check a protocol for deadlocks and fairness constraints is an important part of developing and debugging parallel programs. However, most existing tools require the protocol/model to be specified or implemented separately in the language of the tool, which means that the protocol must be re-implemented in the source language the application is written in.

Some well known approaches to protocol specification include CSP [Hoa78], CTL/ μ -calculus [CE81] and coloured Petri nets [Jen92]. Specifications written in CSP can be verified and checked using the FDR model checking tool [For]. FDR (Failures-Divergence-Refinement) allows for the checking of many properties of finite-state systems and the investigation of systems which fail these checks. CSP allows a wide range of correctness conditions, including deadlock and livelock freedom, as well as general safety and liveness properties to be encoded and checked using FDR.

A different approach to model checking is using CTL (Computational Tree Logic); systems using this abstraction include VIS [The96], Mur ϕ [Dil96] and SMV [CLM89, McM92]. The specification is typically translated into a BDD (Binary Decision Diagram), and various algorithmic techniques can be applied in order to verify statements about the model. All of these systems accept specifications written in different languages, none of which are compatible with standard C or C++. The SPIN [Hol97] system also falls into this category of tools, although it is based on LTL (Linear Temporal Logic) and not CTL. In [San99] two important problems are pointed out with these techniques; these are as follows: the specification languages are fairly low level, and the state space explosion problem is present.

The approach to model checking with coloured Petri-nets is slightly different; the user has

to specify a graphical representation of the protocol and annotate it with code written in ML. A number of analyses can then be performed on the model by constructing a state space for the net. To transcribe a Petri-net model to C requires implementing the protocol based on a graphical representation and translating ML code to C. The risk of introducing errors is increased as the translation from a graphical representation and a functional specification must be performed manually.

7.3 Protocol Constraint Specification

Before we start defining protocols, we introduce a few concepts and definitions. A **group** of processes is an ordered set of processes all spawned from the same `pvm_spawn` call. There can be several groups of the same program depending on the number of spawn calls. An **instance** is one process from a group. Each process in a group is given an instance number, starting at 0, each time a group is spawned.

A **line number** is either a concrete line number containing a `pvm_send`, a `pvm_recv` or an identifier. If an identifier is used, Millipede will search the appropriate source file for comments of the form `/* ((line-label)) */` where `line-label` is the identifier used in the specification of the protocol.

7.3.1 Protocol Contents

To use the PCSL/MOPED, the user first writes a file containing a constraint specification that she wishes to check her program against. We refer to a protocol constraint specification as simply a protocol specification, or just as a specification. A protocol specification file consists of a number of lines that specify which sends can send to which receives. One of the powerful features of the PCSL/MOPED module is the ability to start out by specifying a very general version of the protocol and checking it; as errors are detected and corrected, or as more knowledge about the protocol is gained, the specification can be specialized step by step. A protocol consists of a number of lines of the following form:

$$pgname_1[e_1]\{e_2\}(e_3) \rightarrow pgname_2[e_4]\{e_5\}(e_6)$$

Each line can be followed by a number of **quantifiers** of the following form:

$$\forall id : RelExpression;$$

The first part states that a process created from program `pgname1` with instance number `e2` in group `e1` may send from a send call in line `e3` to a receive call in a process created from a program `pgname2` with instance number `e5` in group `e4` with a receive call in line `e6`. Values for `e1`, `e2` and `e3` can either be omitted, or be a number or an identifier. If `e3` is the identifier `xyz`, and `pgname1.c`

contains a **pvm_send** followed by a `/* ((xyz)) */` comment, e_3 will be substituted with the actual line number of the send call in the source file.

If e_1 or e_2 are identifiers, or if e_3 is an identifier that does not match any `/* (...) */` line in the source file, then these are bound to the group, the instance number, or the line number of the process who sent the message. If any or all of e_1, e_2 , or e_3 are omitted, no check is done for the missing expression. This is equivalent to a wild card match.

Values for e_4, e_5 , or e_6 can either be expressions, identifiers, or be omitted. Again, if omitted, a wild card match is performed. If an expression is given, this expression is evaluated and matched to the actual values of the group, instance, and line number of the process that received the message. If a new identifier is introduced in any of e_4, e_5 or e_6 , it is bound to the actual group, instance, or line number of the receiver of the message. If e_6 is an identifier, a similar replacement, as described for e_3 , will take place.

A quantifier introduces constraints on an identifier used in the e_1, \dots, e_6 . These can be qualified by both lower and upper bounds or bound by other expressions.

A message (sent from a sender to a receiver) is a tuple as follows:

$$\mathbf{M} = (P_s, P_r, (G_s, I_s, L_s), (G_r, I_r, L_r), N_s, N_r)$$

where P_s and P_r are the program names of the sender and receiver processes, G_s, I_s , and L_s denote the group, instance, and line of the sender, and G_r, I_r , and L_r denote those of the receiver. N_s and N_r are the total number of processes in group G_s and G_r . N_s and N_r are reserved names in PCSL; at check time they contain the values of N_s and N_r of \mathbf{M} .

7.4 The PCSL Grammar and Semantics

For completeness, the BNF grammar of the protocol constraint language (PCSL) can be found in Appendix B.1. The expressions and the relational expressions of the grammar are a subset of the grammar for expressions in the C programming language with the square root function added.

In Appendix B.1, the semantics for computing expressions and relational expressions are shown. The Greek symbol σ denotes a symbol table that associates variables with values. Variable/value pairs can be added to the symbol table using the Σ function defined in Figure 7.1.

$$\Sigma[e](v)\sigma = \begin{cases} \sigma \cup \{e = v\} & \text{if } e \text{ is an identifier, and } e \text{ is not bound in } \sigma. \\ \text{error} & \text{if } e \text{ is an identifier, and } e \text{ is already bound in } \sigma. \\ \sigma & \text{otherwise.} \end{cases}$$

Figure 7.1: Adding elements to the symbol table.

With the symbol table σ in place, we now turn our attention to the semantics of a single PCSL

line. Recall the following appearance of a specification line L :

$$\beta[e_1]\{e_2\}(e_3) \rightarrow \delta[e_4]\{e_5\}(e_6) :: Q$$

where Q is a list of quantifiers. Such a line (referred to by L) is always checked with respect to a message M . The semantic function we create is named B . We say that a message M does not violate a specification line L , if

$$B[\![L]\!]M = \text{true}.$$

We briefly explain how a message is checked against a protocol line in the following. Remembering that e_1, e_2 and e_3 can either be left blank, a number (constant) or an identifier. We perform the following for each of these:

- If e_i is a number (c_i), e_i is replaced by α_i , and the quantifier $\forall \alpha_i : \alpha_i = c_i$ is added to Q .
- If e_i is an identifier, replace all of its occurrences by α_i . This step is not necessary, but it clarifies the following explanation.
- If e_i is left blank, replace the blank with α_i , and add the quantifier $\forall \alpha_i : \text{true}$ to Q .

This transformation is applied to each protocol line such that any quantifiers associated with the sender side of a protocol line can be checked separately from the rest of the quantifiers (by looking up quantifiers that bind α_i).

For e_4, e_5 , and e_6 , apply the following transformation: if e_i is left blank, replace e_i with γ_i , and add the quantifier $\forall \gamma_i : \text{true}$ to Q . This transformation is done in order to avoid comparing numbers to empty expressions.

The table in Figure 7.2 summarizes the following in depth explanation of how to check a message against a protocol line. If any of the checks past step 2 fail, the protocol is violated by the message.

Before any checking can be performed, we need to add information from the message M to the symbol table. Recall that e_1, e_2 and e_3 are all replaced by α_1, α_2 and α_3 , respectively. We now use the Σ function to add the bindings $\alpha_1 = G_s$, $\alpha_2 = I_s$, and $\alpha_3 = L_s$ to the symbol table σ . We are now ready to check the protocol line against the actual message M .

- The first step in checking a line against a message is to determine if the actual sender and receiver of the message match the program names specified in the line. The actual sender and receiver are P_s and P_r , and the sender and receiver specified in the protocol line are β and δ . Thus, the first check that must be performed becomes the following: $(P_s = \beta \wedge P_r = \delta)$. If this evaluates to *true* the sender may send a message to the receiver.
- Recall the transformation performed on the protocol line, that is, replacing or adding α_1 , α_2 , and α_3 for the expressions e_1, e_2 , and e_3 . This transformation may result in up to three

quantifiers $\alpha_i : r_i$, which must be checked as well. Here, checked means that the values are within the boundaries of their definitions. r_i is a relational expression, so the semantic function \mathcal{R} is used in the following way: $\bigwedge_{Q \ni q = \forall \alpha_i : r_i} \mathcal{R}[[r_i]]\sigma$. If this expression evaluates to true we know the sender part of the message matches the line.

- If the first two steps of the check are true, the protocol line matches the sender; now we need to check if the receiver of the message matches the receiver part of the protocol line. First, if any of e_4 , e_5 or e_6 are identifiers, add these to the symbol table with the bindings of the receiver group, instance or line number, respectively, (any line number identifiers that existed in a `/* (. . .) */` comment will have been replaced already). All other quantifiers are checked also using the \mathcal{R} function as follows: $\bigwedge_{Q \ni q = \forall v : r} (v, \cdot) \in \sigma \wedge \mathcal{R}[[r]]\sigma$. We can restrict the conjunction to only consider quantifiers where $v \neq \alpha_i$, but to keep it simple we do not bother with this restriction as checking the sender quantifiers one more time does not change anything.
- We now need to check that the actual receiver of the message may indeed receive it according to the specification. That entails checking three properties: The group, the instance, and the line number.
 - The expression e_4 is evaluated using the \mathcal{E} semantic function for evaluating expressions. Note, if e_4 is an identifier, it would have been inserted into the symbol table with the value G_r . The resulting value is then compared to the actual group number of the receiver, G_r : $\mathcal{E}[[e_4]]\sigma = G_r$.
 - A similar check involving the use of \mathcal{E} is performed on the instance number in the following way: $\mathcal{E}[[e_5]]\sigma = I_r$.
 - Finally, the line number of the actual receiver is compared to the value we get by evaluating e_6 : $\mathcal{E}[[e_6]]\sigma = L_r$.

Note, if for example e_4 is left blank, it then gets replaced by γ_4 , and the quantifier $\forall \gamma_4 : true$ is added to Q . This quantifier always evaluates to true; however, since $e_4 = \gamma_4$ is an identifier, the value G_r is associated with it; that is, before performing the quantifier check on the receiver part, the binding $\gamma_4 = G_r$ is inserted into σ . Now, when the check $\mathcal{E}[[\gamma_4]]\sigma = G_r$ is performed, it becomes trivially true because of the binding of γ_4 in the symbol table.

If any of the checks after the second check fail, an error must be reported, as the receiver should not have received the message, or the sender should not have sent the message to the receiver. Figure 7.2 shows the six steps of the checking algorithm. We can summarize the check by the semantic function \mathcal{B} , which is shown in Figure 7.3.

Step	Check	Comment
1	$(P_s = \beta \wedge P_r = \delta)$	If <i>false</i> move on to the next line. If <i>true</i> continue.
2	$\bigwedge_{Q \ni q = \forall \alpha_i : r_i} \mathcal{R}[r_i] \sigma$	This checks if the sender part of the message matches the sender part of the protocol line. If <i>false</i> move on to the next line. If <i>true</i> continue.
3	$\bigwedge_{Q \ni q = \forall v : r} ((v, \cdot) \in \sigma \wedge \mathcal{R}[r] \sigma)$	Check the rest (all) of the quantifiers. If <i>false</i> report a quantifier error. If <i>true</i> continue.
4	$\mathcal{E}[e_4] \sigma = G_r$	Check if the receiver group may receive this message. If <i>false</i> report a group error. If <i>true</i> continue.
5	$\mathcal{E}[e_5] \sigma = I_r$	Check if the receiver instance may receive this message. If <i>false</i> report an instance error. If <i>true</i> continue.
6	$\mathcal{E}[e_6] \sigma = L_r$	Check if the receiver line may receive this message. If <i>false</i> report a line error. If <i>true</i> protocol line is not violated by the message with respect to the semantics of Figure 7.3

Figure 7.2: Checking a protocol line takes six steps. If the check fails in the first 2 steps, it is because the line did not match the sender, so move on to the next. If any of the checks in the subsequent steps fail, it constitutes an error; the sender is matched to the protocol line, but the receiver did not match the line.

$$\mathcal{B}[\mathbf{L}]\mathbf{M} = (P_s = \beta \wedge P_r = \delta) \wedge \left(\bigwedge_{Q \ni q = \forall v : r} ((v, \cdot) \in \sigma \wedge \mathcal{R}[r] \sigma) \right) \wedge (\mathcal{E}[e_4] \sigma' = G_r) \wedge (\mathcal{E}[e_5] \sigma' = I_r) \wedge (\mathcal{E}[e_6] \sigma' = L_r)$$

where

$$\begin{aligned} \sigma &= \Sigma[e_3](L_s)(\Sigma[e_2](I_s)(\Sigma[e_1](G_s)\emptyset)) \\ \sigma' &= \Sigma[e_6](L_r)(\Sigma[e_5](I_r)(\Sigma[e_4](G_r)\sigma)) \\ Q &= \forall v_0 : r_0; \dots; \forall v_n : r_n; r_i \text{ is a relational expression.} \end{aligned}$$

Figure 7.3: Semantics for a PCSL line.

7.5 Examples

In this section we present a number of examples of how to specify protocol constraints.

7.5.1 The Simplest Protocol

As stated in the previous section, a protocol constraint specification can start out being very general. In Figure 7.4 the simplest possible protocol is shown.

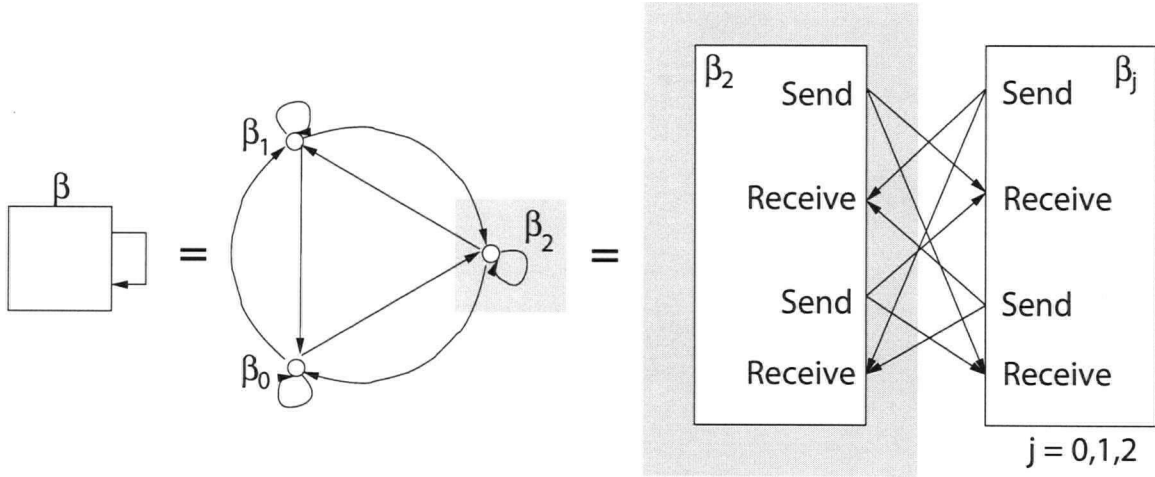


Figure 7.4: Example of $\beta[]\{\}\{()\} \rightarrow \beta[]\{\}\{()\};$

This protocol consists of only the following one line (we use the Greek symbol β as a shorthand notation to represent a program name):

$$\beta[]\{\}\{()\} \rightarrow \beta[]\{\}\{()\};$$

This line states that any β process can send to any other β process regardless of the group, instance or line number. The first part of the picture in Figure 7.4 shows that β processes communicate among themselves. The second part shows that any β process can communicate with any one β process. Lastly, the rightmost part shows that all sends in any β process may send to any receive in any β process, including itself.

We can specialize this very simple specification to represent a system where β process number i can send to another β process with instance number $i + 1$, and where process number $n - 1$ sends to instance number 0. In summary, we have the following:

$$\begin{aligned} \beta[]\{0\}() &\rightarrow \beta[]\{1\}(); \\ \beta[]\{1\}() &\rightarrow \beta[]\{2\}(); \\ &\vdots \\ \beta[]\{n-1\}() &\rightarrow \beta[]\{0\}(); \end{aligned}$$

Alternatively, in short notation using a quantifier, we arrive at the following:

$$\beta[]\{i\}() \rightarrow \beta[]\{(i+1)\%n\}() :: \forall i : 0 \leq i \ \&\& \ i \leq n-1;$$

In Figure 7.5 this protocol is shown graphically with a fully quantified PCSL line.

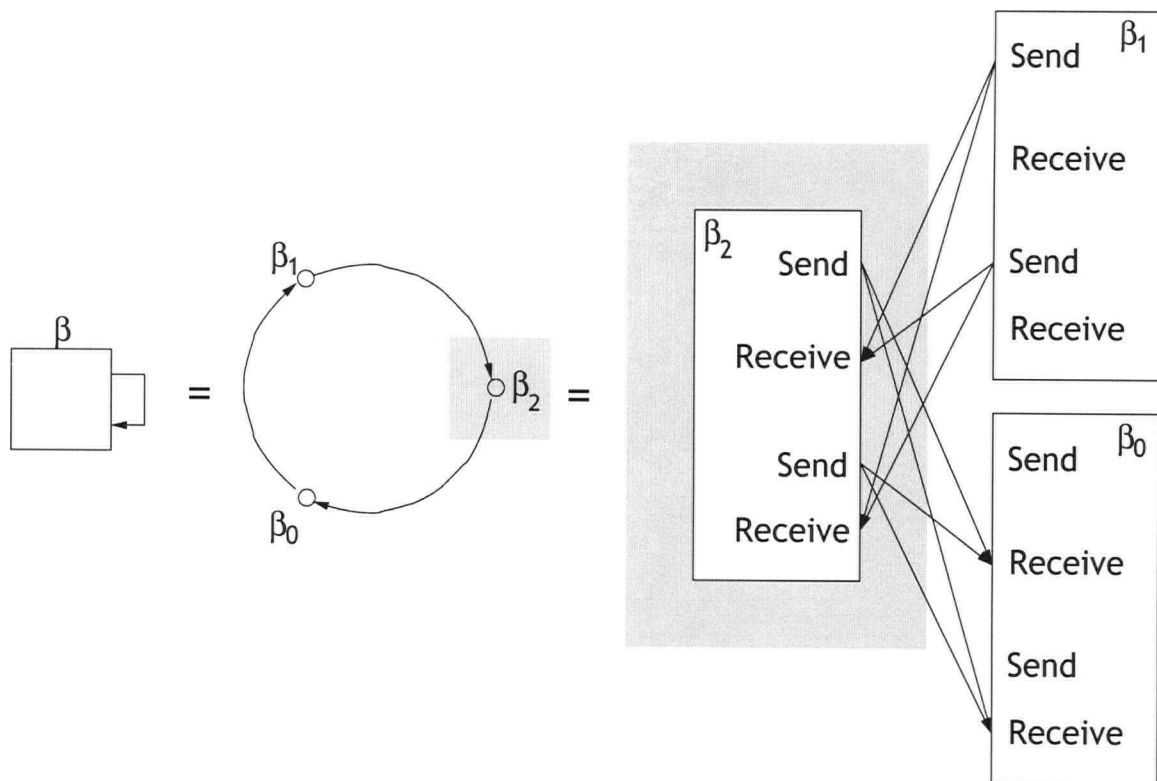


Figure 7.5: Example of $\beta[i](\) \rightarrow \beta[(i+1)\%n](\) :: \forall i : 0 \leq i \ \&\& \ i \leq n-1$;

7.5.2 Pipe-and-Roll Matrix Multiplication

Consider a more complex example that also includes the use of line numbers. The pseudo code for the pipe-and-roll matrix multiplication algorithm [FJL⁺88] is shown in Figure 7.6 (the master's code) and in Figure 7.7 (the slave's code). Processes communicate subblocks of a matrix in a two-dimensional grid, sending up and right to neighbor processes. (A graphical illustration of this protocol can be seen in Figure 7.8.)

```

Let  $N*N$  be the number of processors
Map concurrent computer on to array of  $N*N$  processors
Distribute subblocks of  $A$  and  $B$  to processors
Await subblock results in matrix  $C$ 

```

Figure 7.6: Pseudo code for the master of the pipe-and-roll matrix multiplication algorithm.

As we can see from the 2 functions **Pipe_A** and **Roll_B**, a process executing a pipe call can

```

Initialize subblock matrix C to 0
Receive subblocks A and B
for i=0 to N-1 do {
    T = Pipe_A()
    C = C + T*B
    Roll_B()
}
Send subblock C to master

Pipe_A() {
    Determine the source processor of the pipe
    Determine the last processor of the pipe.
    if (this processor is the source processor) then
        Copy A to T
    else if (processor is not the source processor) then
        Receive T from processor on the left
    if (processor is not the last processor in pipe) then
        Send T to processor on the right
    return T
}

Roll_B() {
    Send B to processor above (with wrap around)
    Receive B from processor below.
}

```

Figure 7.7: Pseudo code for the slave of the pipe-and-roll matrix multiplication algorithm.

only send to the process to the right of it and receive from the process to the left of it, and when executing a **Roll_B**, it can only send to the process above it and receive from the process below it (assuming the processes are arranged in a grid of size $N \times N$). Let us assume, for simplicity, that $N = 4$ in the following; that is, we are working with a 4×4 grid of processes.

A process with instance j performing a **Pipe_A** operation can send to process $(j+1)\%4$, and a process j performing a **Roll_B** operation can send to process $(j+12)\%16$. This can be expressed by the following two PCSL lines:

$$\begin{aligned}
 \text{Matrix}[j](\text{SendPipe}) &\rightarrow \text{Matrix}[(j+1)\%4](\text{ReceivePipe}) :: \forall j : j < 16; \\
 \text{Matrix}[j](\text{SendRoll}) &\rightarrow \text{Matrix}[(j+12)\%16](\text{ReceiveRoll}) :: \forall j : j < 16;
 \end{aligned}$$

The graphical representation can be seen in Figure 7.8. This only includes the communication between the worker processes (called *Matrix*).

To add protocol specification lines to check communication between the master (*Master*) and

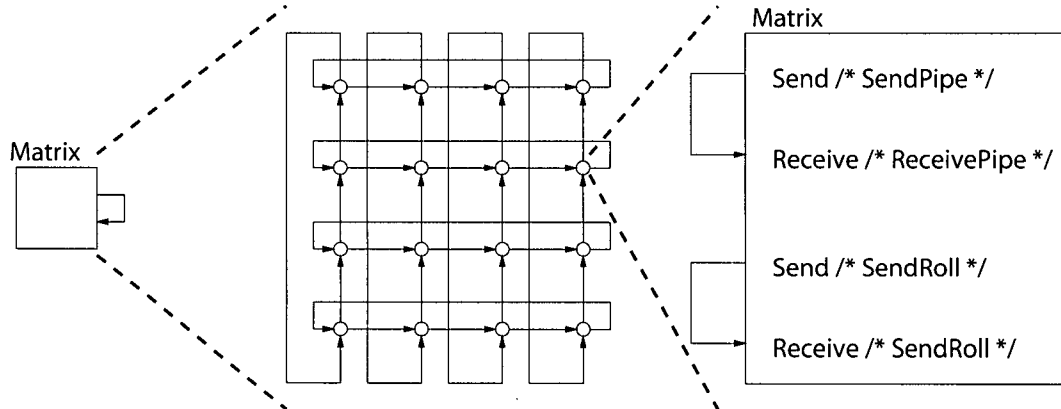


Figure 7.8: The pipe-and-roll part of the matrix multiplication algorithm.

the slaves, add the following two lines to the specification file:

$$\begin{aligned} Master[]\{0\}(SendParams) &\rightarrow Matrix[]\{ \}(ReceiveParams); \\ Matrix[]\{ \}(SendResult) &\rightarrow Master[]\{0\}(ReceiveResult); \end{aligned}$$

Also, note that the group numbers are left out to simplify the description of the protocol.

Limitations

By inspecting the communication pattern in the program pseudo code, it becomes clear that the pipe communication does not need to wrap around, that is, the last processor in the pipe does not need to send anything to the source processor. The source processor of each round of pipes differs from the one in the previous round. It is not directly possible to specify a protocol that reflects such a communication pattern that depends on the state in the application. In Section 7.10 we describe a way to resolve this problem and expand the set of protocols that can be specified.

7.5.3 A Partial Differential Equation Solver

Let us consider a parallel master/slave program to solve a hyperbolic differential equation. There is one master process and n slave processes. Figure 7.9 shows the algorithm for the master, and Figure 7.10 for the slaves.

Version 1 of the Protocol Constraint Specification

The most general protocol, \mathcal{P}_1 (covering all sends) that we can specify for the master/slave system is illustrated in Figure 7.11.

The \mathcal{P}_1 protocol contains 3 lines:

```

Send parameters to slaves 0,...,N-1          /* ((MS)) */

Repeat N times {
  Receive result from slave                  /* ((MR)) */
}

```

Figure 7.9: Pseudo code for master algorithm for a differential equation.

```

Receive parameters from master                /* ((SR)) */

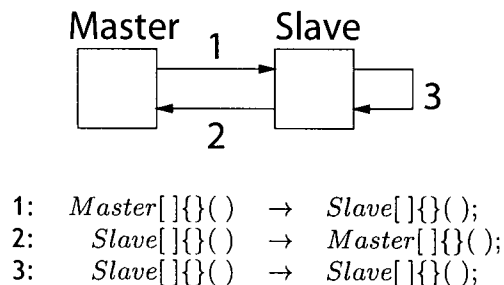
Repeat n times {
  if (id > 0) then
    Send to slave id - 1                    /* ((S1)) */
  if (id < N-1) then
    Send to slave id + 1                    /* ((S2)) */

  Calculate

  if (id > 0) then
    Receive from slave id - 1               /* ((R1)) */
  if (id < N - 1) then
    Receive from slave id + 1               /* ((R2)) */
}
Send result to the master                    /* ((SS)) */

```

Figure 7.10: Pseudo code for slave algorithm for a differential equation solver.

Figure 7.11: \mathcal{P}_1 – Version 1 of the protocol specification.

1. Any master program can send to any slave program regardless of group, instance, or line number.

2. Any slave program can send to any master program regardless of group, instance, or line number.
3. Any slave program can send to any other slave program regardless of group, instance, or line number.

\mathcal{P}_1 is not very useful; it does not specify anything about the communication between the slaves. First, we extend \mathcal{P}_1 for master group 0 (only one group of master programs is spawned, and this group contains only one process with instance 0). This changes the left part of the first line and the right part of the second line in Figure 7.11 to $Master[0]\{\}\{0\}()$. Likewise, for the slaves, there is only one group of slaves spawned, so lines 1, 2, and 3 can be changed to $Slave[0]\{\}\{0\}()$. Let \mathcal{P}'_1 denote this version of the protocol specification, as shown in Figure 7.12.

```

1:   $Master[0]\{\}\{0\}()$    $\rightarrow$    $Slave[0]\{\}\{0\}()$ ;
2:   $Slave[0]\{\}\{0\}()$    $\rightarrow$    $Master[0]\{\}\{0\}()$ ;
3:   $Slave[0]\{\}\{0\}()$    $\rightarrow$    $Slave[0]\{\}\{0\}()$ ;

```

Figure 7.12: \mathcal{P}'_1 – Extended version 1 of the protocol specification.

Version 2 of the Protocol Constraint Specification

By inspecting the code in Figure 7.10 we see that slave number i can send to slave number $i + 1$ if $i < N - 1$ (assuming the system has N slave processes), and slave number i can send to slave number $i - 1$ if $i > 0$. Figure 7.13 shows the protocol as a graphical representation. We can incorporate this into the protocol specification and arrive at the second version, which is shown in Figure 7.14.

Note, that line 3 is split into line 3a (i sends to $i + 1$) and line 3b (i sends to $i - 1$). Also note the use of the two quantifier expressions following these lines.

Version 3 of the Protocol Constraint Specification

Looking closer at the lines 3a and 3b in Figure 7.14, and comparing these with the pseudo code in Figure 7.10, we see that the \mathcal{P}_2 protocol specification does not specify that the send marked $S1$ always sends to the receive marked $R1$, and that the send marked $S2$ always sends to the receive marked $R2$. If, by mistake, a message were delivered to the wrong receive, there will be a violation of the communication protocol, so we need to add this information to the specification. Thus, line 3a represents the message passed between send $S1$ and receive $R1$, and line 3b represents the message passed between send $S2$ and receive $R2$. Adding this to the specification we obtain the third version, as shown in Figure 7.15. For completeness, we added line information about the parameter and result messages sent to and from the master.

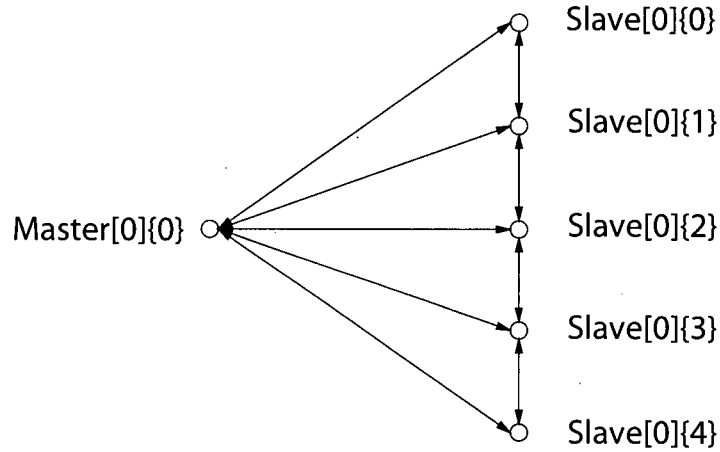


Figure 7.13: Graphical representation of \mathcal{P}_2 – the second version of the protocol specification.

- 1: $Master[0]\{0\}() \rightarrow Slave[0]\{()\};$
- 2: $Slave[0]\{()\} \rightarrow Master[0]\{0\}();$
- 3a: $Slave[0]\{i\}() \rightarrow Slave[0]\{i+1\}() :: \forall i : i < n-1;$
- 3b: $Slave[0]\{i\}() \rightarrow Slave[0]\{i-1\}() :: \forall i : 0 < i;$

Figure 7.14: \mathcal{P}_2 – Version 2 of the protocol specification.

- 1: $Master[0]\{0\}(MS) \rightarrow Slave[0]\{()\}(SR);$
- 2: $Slave[0]\{()\}(SS) \rightarrow Master[0]\{0\}(MR);$
- 3a: $Slave[0]\{i\}(S1) \rightarrow Slave[0]\{i+1\}(R1) :: \forall i : i < n-1;$
- 3b: $Slave[0]\{i\}(S2) \rightarrow Slave[0]\{i-1\}(R2) :: \forall i : 0 < i;$

Figure 7.15: \mathcal{P}_3 – Version 3 of the protocol specification.

Figure 7.16 shows an extended version of \mathcal{P}_3 , where we added information about the instance of the slaves in lines 1 and 2. Furthermore, we added an upper bound for i in line 3b, and a lower bound for i in line 3a. All these changes do not change the protocol in any way, but allow the system to predict which sends/receives are legal. \mathcal{P}_3 can only be checked, not predicted (see Section 7.8 for more information on protocol prediction).

7.6 Online Checking

MOPED can be used in two different modes: online or offline. The online mode checks the specification as the communication takes place; each message in the system is captured by Millipede and checked against the constraint specification. If an error occurs, that is, if a message

- 1: $Master[0]\{0\}(MS) \rightarrow Slave[0]\{i\}(SR) :: \forall i : (0 \leq i) \ \&\& \ (i < n);$
- 2: $Slave[0]\{i\}(SS) \rightarrow Master[0]\{0\}(MR) :: \forall i : (0 \leq i) \ \&\& \ (i < n);$
- 3a: $Slave[0]\{i\}(S1) \rightarrow Slave[0]\{i + 1\}(R1) :: \forall i : (0 \leq i) \ \&\& \ (i < n - 1);$
- 3b: $Slave[0]\{i\}(S2) \rightarrow Slave[0]\{i - 1\}(R2) :: \forall i : (0 < i) \ \&\& \ (i < n);$

Figure 7.16: P'_3 — Extended version 3 of the protocol specification.

violates the protocol specification, a message is displayed in the Millipede status window.

When developing programs, this approach can be used incrementally, as shown in the example in Section 7.5.3. The first version of the specification can be very general, and then gradually refined until errors are discovered. Once the error is corrected, the specification can be further refined if the program still does not function correctly.

7.6.1 Strictness

A protocol specification can be checked using different levels of strictness. When using the refinement technique, that is, starting out with a simple specification, some messages might not match any lines, thus violating the protocol. The user might not perceive this as a violation as the protocol is not fully specified; if this is the case, a lower level of strictness can be adopted. Table 7.1 shows the 3 different levels of strictness that MOPED currently supports.

Level	Description
1	0 or more protocol specification lines may match with respect to program name and sender quantifiers.
2	At least one protocol specification line must match with respect to program name and sender quantifiers.
3	Exactly one protocol specification line must match with respect to program name and sender quantifiers.

Table 7.1: The MOPED Strictness levels.

Strictness level 1 should be used when the protocol has not yet been fully specified, level 2 when the protocol is fully specified, but not uniquely (i.e., a message can match more than one protocol line), and level 3 if a full specification is given.

7.7 Offline Checking

As described in the previous section, Millipede can check messages against the protocol specification, while the program is running. However, if Millipede is generating log files while the program executes, the checking can also be performed offline. All the information needed to check the protocol can be extracted from the set of log files and the corresponding project file.

7.8 Protocol Prediction

As mentioned earlier, if all constraint lines are fully quantified with bounds for each variable, Millipede can generate a list of all possible valid send/receive combinations. For the example in Figure 7.16, the prediction table is shown in Table 7.2 (for $n = 4$).

Sender	Receiver	Line
$Master[0]\{0\}(MS)$	$\rightarrow Slave[0]\{0\}(SR)$	1
	$\rightarrow Slave[0]\{1\}(SR)$	1
	$\rightarrow Slave[0]\{2\}(SR)$	1
	$\rightarrow Slave[0]\{3\}(SR)$	1
$Slave[0]\{0\}(SS)$	$\rightarrow Master[0]\{0\}(MR)$	2
$Slave[0]\{0\}(S1)$	$\rightarrow Slave[0]\{1\}(R1)$	3a
$Slave[0]\{1\}(SS)$	$\rightarrow Master[0]\{0\}(MR)$	2
$Slave[0]\{1\}(S1)$	$\rightarrow Slave[0]\{2\}(R1)$	3a
$Slave[0]\{1\}(S2)$	$\rightarrow Slave[0]\{0\}(R2)$	3b
$Slave[0]\{2\}(SS)$	$\rightarrow Master[0]\{0\}(MR)$	2
$Slave[0]\{2\}(S1)$	$\rightarrow Slave[0]\{3\}(R1)$	3a
$Slave[0]\{2\}(S1)$	$\rightarrow Slave[0]\{1\}(R2)$	3b
$Slave[0]\{3\}(SS)$	$\rightarrow Master[0]\{0\}(MR)$	2
$Slave[0]\{3\}(S1)$	$\rightarrow Slave[0]\{2\}(R2)$	3b

Table 7.2: Prediction table for the \mathcal{P}'_3 protocol specification.

A prediction table can help determine if the protocol specified matches what the user had in mind. Naturally, there is always a risk that an error is present in the protocol specification; this is similar to the risk mentioned in Section 7.2 of introducing errors into the implementation of a protocol that is verified using a model checker. However, we believe that the number of errors introduced here should be considerably smaller than in the implementation stage.

7.9 Implementation

As with all other Millipede modules, the MOPED module is a separate process that runs the protocol checking algorithm. MOPED parses the constraint specification file using a parser generated from the BNF grammar in Appendix B.1. This parser is generated using Flex [Pax98] and Bison [CSH02]. A parse representing the specification is returned by the parser, and messages can be checked against this specification by evaluating the tree using the information about the sender, receiver, group, and instance number. Messages are provided by the Millipede runtime system; when the module is run offline (i.e., the application is not currently running) the messages are extracted from the log files. The protocol specification lines are checked against a message one at a time, and depending on the strictness level, errors are reported to the user.

7.10 Discussion

A number of interesting extensions should be added to PCSL and the MOPED checking module in order to strengthen the quality of the checks performed. We briefly describe some of these in this section.

Since we are working with message passing systems, such as PVM and MPI, where all sends are annotated with a message tag, it should be possible to add information about message tags to a PCSL line. This means expanding the following:

$$\beta[\{\}](\) \rightarrow \delta[\{\}](\)$$

to the following:

$$\beta[\{\}](\) < \ > \rightarrow \delta[\{\}](\) < \ >$$

where $< \ >$ represents an expression that determines the message tag.

In order to ease the possibility of choosing from a small number of values, another useful functionality would be to allow set expressions of the following form:

$$e \in \{v_1, v_2, \dots, v_n\}.$$

So far, the focus has been on the senders and receivers of messages. However, errors also occur because the content of messages is incorrect. Another useful extension is to allow each PCSL line to be associated with one or more templates describing the structure of the message being sent and received.

This can be achieved easily in Java by defining messages as objects, and using the reflection mechanism (`instanceof` function) to determine the type of the incoming object. In Occam [May83] the notion of typed channels assures that the correct type of data is always received on a channel. However, in PVM and MPI, the notion of channels does not exist, and when using message passing static analysis and type checking are not always possible. A possible solution can be specifying message content using a specification language like XDR [Sri95] or XML [XML98] to define data types, or using the `MPI_Datatype` function, which specifies an internal message data type for all calls.

The above mentioned extensions to PCSL/MOPED not only allow for a more refined protocol specification, which results in more rigorous checks, but also make it possible to check that messages contain the correct type of data. This last point again shows an example of an overlap between levels; at the protocol level we are also concerned with the content of the messages, which theoretically should be included in the Message Level.

7.10.1 State Dependent Communication

As the last part of this section we briefly return to the problem stated in Section 7.5.2. The problem is defining a protocol that depends on values stored in the program at runtime. The example at hand is more clearly illustrated in Figure 7.17 (the roll part of the protocol is left out for clarity). Depending on the program variable k , a number of processes do not send anything; this set of nonsending processes varies according to the row in which the process is located, as well as the number k .

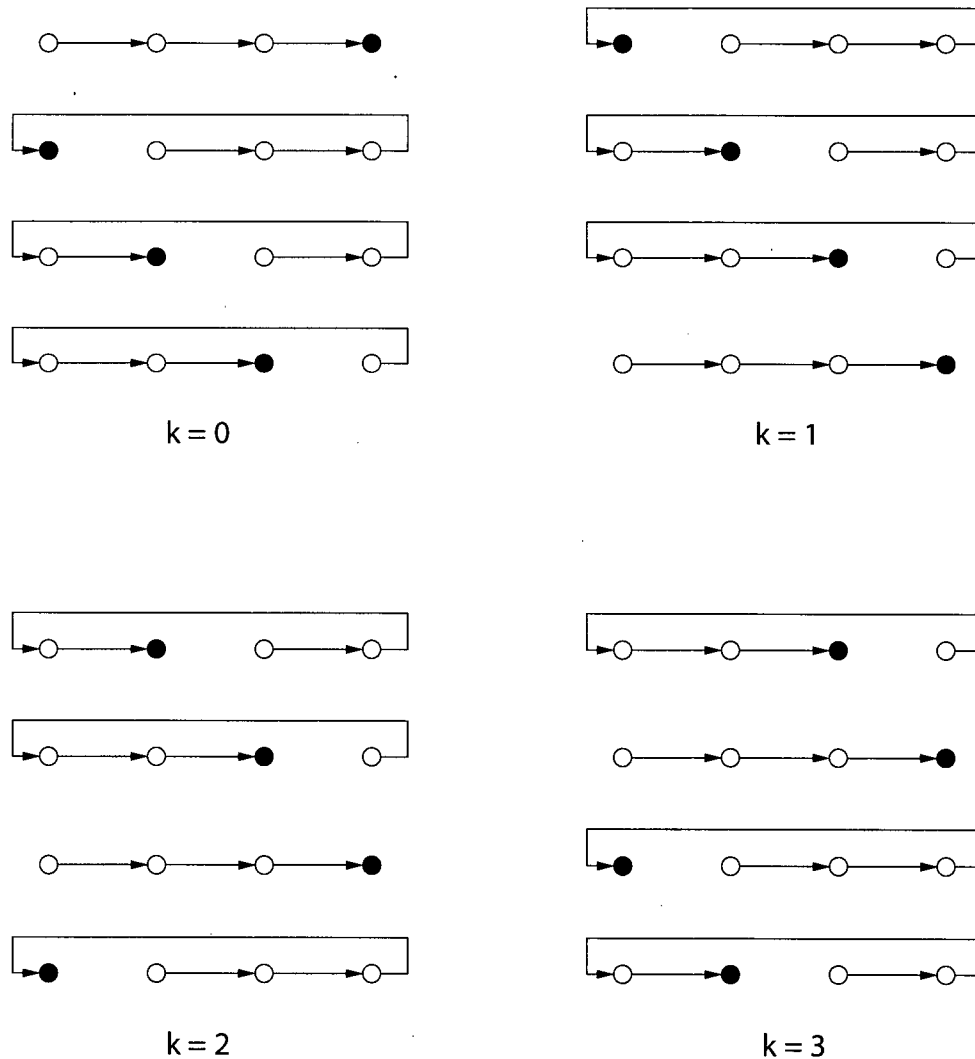


Figure 7.17: The four different stages of the pipe operation.

A simple formula that determines which instance numbers should not send, given a value k

and a row number, is the following:

$$(row + k + N_s - 1) \% N_s + row * N_s$$

where N_s is the group size. We can write the following protocol specification line:

$$Matrix[\{j\}(SendPipe) \rightarrow Matrix[\{(j+1)\%4\}(ReceivePipe) :: \\ \forall j : j \leq 0 \ \&\& \ j < N_s \ \&\& \ j \neq (row + k + N_s - 1) \% N_s + row * N_s;$$

This requires the values of the program variables k and row (row can be computed as $N_s/4$). These can be obtained by adding lines to the program in the following way:

```
protocol_sym(row);
protocol_sym(k);
pvm_send(...);    /* ((SendPipe)) */
```

The program variables row and k are then packed and sent to the protocol checking module (and added to the log files), and inserted, along with N_s , into the symbol table σ at check time.

Appendix B.2 shows an entire debugging session using the protocol specification language and the MOPED checking tool.

7.11 Summary

In this chapter we presented a tool at the protocol level that can assist the user in checking that the messages in a system adhere to a protocol specification. The specification of the protocol is comparable to assertions in C; if a message violates a line in the specification, the user is notified about the line as well as the offending message.

We have designed PCSL to be small in comparison to other specification languages to avoid having the user learn a complete new language. We have also avoided adding temporal constraints to the language. The reason is to reduce its complexity to the more simple task of matching message patterns. Adding temporal constraints is possible, however, the added expressiveness complicates the language when compared to simpler task of message matching.

A number of interesting extensions are proposed; these extensions provide an even more flexible tool to aid the user when developing and debugging protocols.

Chapter 8

Buffer Allocation in Message Passing Programs

“How extremely stupid not to have thought of that.”
– *Thomas Huxley* on reading ‘Origin of the Species’

“Just because a problem is NP-complete doesn’t mean we can’t try to solve it; as a matter of fact, those problems are the only interesting ones.”
– *My M.Sc. supervisor Peter Møller-Nielsen, The University of Aarhus*

In the previous two chapters we described two tools at the protocol level of the Millipede tool. In this chapter, we present a number of theoretical results concerned with buffer allocation in message passing systems, plus a tool at the protocol level of Millipede.

The motivation behind the work presented in this chapter is the simple question ‘can we determine the minimum number of buffers needed for an asynchronous message program to be guaranteed not to deadlock because of an insufficient number of buffer?’ This question leads to a detailed investigation of several buffer related questions under different buffer placement schemes [BBW02];

An example showing the importance of this problem is the following: assume a program has been developed and tested on a cluster of machines using a small problem set. Now, when the program is executed on a larger problem on a production machine, it deadlocks due to lack of buffers. If the production machine has fewer buffers due to the bigger problem size, an otherwise working program might deadlock.

However, the answer to the original question turns out to be ‘no, not for systems like MPI and

PVM,' but other useful discoveries and results emerge in the process. One in particular, referred to as the Nonblocking Buffer Allocation Problem, turns out to be tractable, and thus a useful tool to be included in Millipede. In addition, our ability to solve this problem enables us to compute approximations for the original, intractable, problem.

8.1 Motivation and Related Work

The multiprocess system that we consider is a collection of simultaneously executing independent asynchronous processes that compute by interspersing local computation and point-to-point message passing between processes; these are referred to as A-computations in [CMT96]. Such a system is equivalent to one with three different events, such as the one defined by Lamport [Lam78]: send events, receive events and internal events. As well, we only consider programs that are repeatable [CL94a, CL94b] when executed in an unrestricted environment, that is, programs with static communication patterns. While this narrows the class of programs we consider, the class of applications with static communication patterns is still considerable.

The message passing primitives considered are the traditional, asynchronous, buffered communications: the nonblocking send and the blocking receive, which are the standard primitives used in MPI and PVM. Cypher and Leu formally define the former as a POST-SEND, immediately followed by a WAIT-FOR-BUFFER-RELEASE, and the latter as a POST-RECEIVE immediately followed by a WAIT-FOR-RECEIVE-TO-BE-MATCHED [CL94a, CL94b]. Informally, the send blocks until the message is copied out of the process into a send buffer; the receive blocks until the message has been copied into the receive buffer.

One aspect of portability introduced in the MPI standard [Don94] is that of a safe program. As defined in the standard, a program is safe if it requires no buffering, that is, if it is synchronous. Safe programs can be ported to machines with differing amounts of buffer space. Determining whether a system is buffer independent—the system is 0-safe—was investigated in [CL94a, CL94b]. However, to demand that the program execute correctly with no buffering is restrictive. Buffering reduces the amount of synchronization delay and also makes it possible to offload communication to the underlying system or network components, thus overlapping communication and computation.

The notion of safety, as introduced in the MPI standard, underscores the concern that, when buffer resources are unknown, asynchronous communication can potentially deadlock the system. This notion is extended to k -safety, in order to better characterize the buffer requirements of the program, thus making it safe to take advantage of asynchronous communication. The definition of k -buffer correctness is introduced by Bruck et al. [BDH⁺95] to describe programs that complete without deadlock in a message passing environment with k buffers per process. Similarly, Burns and Daoud [BD95b] introduce guaranteed envelope resources into LAM [GBV94], a public domain

version of MPI. Guaranteed envelope resources—a weaker condition than k -safety—is used in LAM to reserve a guaranteed number of message header slots on the receiver side.

In our model, the interesting systems are buffer dependent, and require an unknown number of buffers to avoid deadlock. More recently in modern clusters, greater overlap of computation and communication is possible by offloading communication onto network interface cards. Unfortunately, most NICs have orders of magnitude less memory than the average host, which makes message buffers a limited resource. Thus, programs that use asynchronous message passing, and that execute correctly otherwise, might deadlock when executing on a system where parts of the message passing system have been offloaded to the NIC. These issues have been investigated in several papers [Don94, DHHW93, FBH⁺92, KW01].

Unfortunately, the value of k , for determining k -safety, is usually not known a priori. We have investigated the complexity of determining the minimum value of k for programs using asynchronous buffered communication with static communication patterns. A program is said to be k -safe if k buffers are enough to guarantee that the program never deadlocks due to insufficient buffers.

In this chapter, we consider the following three problems, all related to determining buffer requirements for asynchronous message passing programs:

BAP—the Buffer Allocation Problem: What is the minimum number of buffers required to ensure deadlock free execution (i.e., determine k for k -safety)?

BSP—the Buffer Sufficiency Problem: Given a buffer assignment, can we determine whether or not the assignment is sufficient to avoid deadlock?

NBAP—the Nonblocking Buffer Allocation Problem: What is the minimum number of buffers needed to allow for an asynchronous execution (i.e., no send calls block)?

The complexity of these questions depends as well on the type of buffers provided by the system. We consider the following types of buffering schemes. In the first three schemes, the buffers are either allocated on the send side only, the receive side only, or mixed and allocated on both sides. Finally, we also consider schemes that allocate buffers on a per channel basis.

In the following section, we present the results of our investigation for the different buffer allocation schemes; we also present a tool in Millipede to assist the user to solve the Nonblocking Buffer Allocation Problem. The solution for this problem is an upper bound for the Buffer Allocation Problem, and we later return to how the user can use this tool to reduce buffer requirements for a system by inserting barrier synchronizations.

Variations of these problems have been investigated by the operations research community [Ana89, Rei87, She75]. In these models, events or products are buffered between various stations in the production process, however, the arrival of these events is governed by probability

distributions, which are specified a priori. In our model, since processes are asynchronous, the time for a message to arrive is nondeterministic; that is, a message may take an arbitrarily long time to arrive and a process may take an arbitrarily long time to perform a send or a receive.

To determine the minimum number of buffers, the execution of a system can be modeled using a (coloured) Petri net [Jen92]. In order to determine whether the system can reach a state of deadlock, the Petri net occurrence graph [HJJ85] is constructed, and a search for dead markings is performed. However, the size of the occurrence graph is exponential in the size of the original Petri net.

8.2 Buffer Allocation Problems

We now introduce a number of definitions to formalize the model we will work with. Let S be a multiprocess system with n processes and E_i communication events occurring in process i ; a communication event is either a send or a receive call. A communication graph of S is a directed acyclic graph $G(S) = (V, A)$ where the set of vertices $V = \{v_{i,c} \mid 1 \leq i \leq n, 0 \leq c \leq (E_i + 1)\}$ corresponds to the communication events and the arc set A consists of two disjoint arc sets: the computation arc set P and the communication arc set C . Each vertex represents an event in the system: vertex $v_{i,0}$ represents the start of process i , vertex $v_{i,c}$, $1 \leq c \leq E_i$, represents either a send or a receive vertex, and finally, vertex $v_{i,(E_i+1)}$ represents the end of a process. An arc $(v_{i,c}, v_{i,c+1}) \in P$, $0 \leq c \leq E_i$, represents a computation within process i and an arc $(v_{i,s}, v_{j,t}) \in C$ represents a communication between different processes, i and j , where $v_{i,s}$ is a send vertex, and $v_{j,t}$ is a receive vertex. Note, process arcs are drawn without orientation, but are always oriented downward. These communication graphs are comparable to the time-space diagrams—without internal events—noted in [Lam78].

A multiprocess system S is *unsafe* if a deadlock can occur due to an insufficient number of available buffers; if S is not unsafe, then S is said to be *safe*. Figure 8.1 shows an example of a system that is unsafe; with no buffers this system always deadlocks.

A per-process buffer assignment is an n -tuple $B = (b_1, b_2, \dots, b_n)$ of nonnegative integers representing the number of buffers for each process. Similarly, a per-channel buffer assignment is a q -tuple $B = (b_1, b_2, \dots, b_q)$, $q = \binom{n}{2}$, representing the number of buffers allocatable by the application; ideally, as few buffers as possible should be allocated.

Two natural decision problems arise from this optimization problem. Given a communication graph $G(S)$ and a nonnegative integer k , the Buffer Allocation Problem (BAP) is deciding whether there exists a buffer assignment B such that S is safe and $\sum b_i \leq k$. In order to solve this problem we need to solve a simpler one. Suppose we are given a buffer assignment B and a communication graph $G(S)$, the Buffer Sufficiency Problem (BSP) is deciding whether the assignment is sufficient to make S safe.

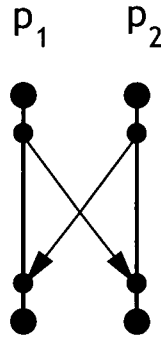


Figure 8.1: An unsafe system—without buffers this communication graph always deadlocks.

In addition, we can require that no process in the system S should ever block on a send. Given a communication graph $G(S)$ and a nonnegative integer k , the Nonblocking Buffer Allocation Problem (NBAP) is deciding whether there exists a buffer assignment B , such that no send in S ever blocks, and $\sum b_i \leq k$. As we see later, this problem plays a key role in the buffer reduction technique we describe in Section 8.4.

In Section E.1 in Appendix E, we formally present the graph framework that we use to prove our results. The first result that we prove is Theorem 8.2.1 concerning the Buffer Allocation Problem.

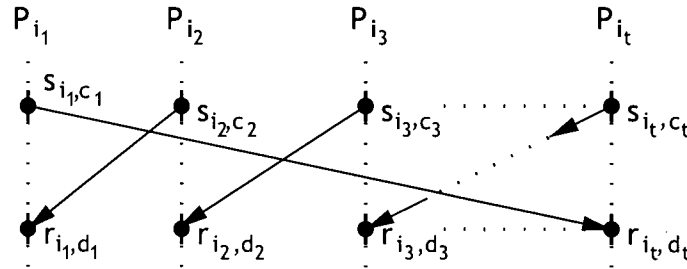
Theorem 8.2.1 *Assuming buffers are allocated on the receiver side, the Buffer Allocation Problem (BAP) is NP-hard.*

Proof: See page 147 in Appendix E. ■

Theorem 8.2.1 shows that determining the minimum number of buffers needed for a program to be k -safe, that is, determining the value k , is intractable. Thus, there is likely no polynomial time algorithm to solve this problem.

To illustrate this problem, consider the graph shown in Figure 8.2. To assure deadlock free execution, such a graph must have at least one buffer. The buffer can be placed in either of the processes; however, the choice of the process might affect future buffer requirements. The graph in Figure 8.2 is an example of a graph used throughout the proofs of both BAP and BSP in Appendix E; we refer to such a graph as a t -ring. A t -ring is a subgraph of a communication graph $G(s)$, consisting of $t > 1$ processes, such that in each of the t processes there is a send vertex s_{i_j, c_j} and a receive vertex r_{i_j, d_j} , $c_j < d_j$, $1 \leq j \leq t$ such that the arcs $(s_{i_1, c_1}, r_{i_t, d_t})$ and $(s_{i_{j+1}, c_{j+1}}, r_{i_j, d_j})$, $1 \leq j < t$ are in A .

The next step in our investigation is to consider a potentially easier problem, referred to as the Buffer Sufficiency Problem.

Figure 8.2: A general t-ring in $G(S)$.

Theorem 8.2.2 *Assuming buffers are allocated on the receiver side, the Buffer Sufficiency Problem (BSP) is coNP-complete.*

Proof: See page E.3.2 in Appendix E. ■

As Theorem 8.2.2 states this problem is as hard as BAP, that is, in polynomial time we cannot verify that a buffer assignment is sufficient to avoid deadlocks.

These two results mean that adding a polynomial time automated analysis to Millipede to determine the exact minimum number of buffers to assure k -safety is difficult. However, as the opening quote of this chapter states, just because a problem is NP-hard does not mean we cannot provide approximations and heuristics. In Section 8.4, we present a technique that utilizes barriers to reduce buffer requirements; by inserting barriers or making certain communications synchronous, the buffer requirements for NBAP is reduced. Since the result of NBAP is still an upper bound for BAP, this is a way of approximating the number of buffers required to assure safe execution in the original program, augmented with the barriers. We now turn to the Nonblocking Buffer Allocation Problem.

8.3 The Nonblocking Buffer Allocation Problem

We have shown the Buffer Allocation Problem to be intractable; interestingly, the problem of determining the minimum number of buffers needed to assure nonblocking sends for an asynchronous message passing program, referred to as the Nonblocking Buffer Allocation Problem (NBAP), is tractable. This means we have a trivial upper bound for BAP. However, examples exist where the result of the NBAP algorithm used as an approximation for BAP results in an unbounded overestimation of the optimal solution for BAP. In Section 8.4 we return to such an example.

The NBAP problem is stated as follows for a multiprocess system S : does there exist a buffer assignment B , such that no send in S ever blocks, and $\sum b_i \leq k$?

To explain the algorithm we need to introduce a few definitions. Given a communication graph $G(S)$ and two vertices, $v_{i,c+k}$ and $v_{i,c}$ in $G(S)$. Vertex $v_{i,c+k}$ is communication dependent

on vertex $v_{i,c}$ if $v_{i,c}$ is the start vertex, or if there exists a vertex $v_{j,d}$, $j \neq i$, such that there exists a path from $v_{i,c}$ to $v_{j,d}$ and the arc $(v_{j,d}, v_{i,c+k})$ is in A . See Figure 8.3. Vertex $v_{i,c+k}$ is terminally communication dependent on vertex $v_{i,c}$ if $v_{i,c+k}$ is communication dependent on $v_{i,c}$ and is not communication dependent on the vertices $v_{i,c+l}$, $0 < l < k$.

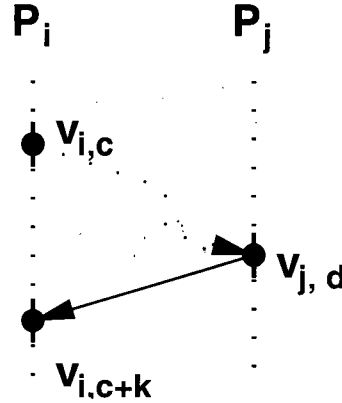


Figure 8.3: $v_{i,c+k}$ is communication dependent on $v_{i,c}$.

The algorithm for computing the minimum buffer assignment to assure nonblocking sends is shown in Figure 8.4. Section E.4 in Appendix E contains the proof for the correctness of the NBAP algorithm.

1. For each receive vertex $v_{i,t}$ determine its terminal communication dependency, vertex $v_{i,c}$, where $t > c$.
2. Set $I_{i,t} = [c, t]$ to be the interval between vertex $v_{i,c}$ and vertex $v_{i,t}$.
3. For each process component G_i , compute b_i , the maximum overlap over all intervals $I_{i,t}$.
4. $B = \{b_1, b_2, \dots, b_n\}$ is the optimal nonblocking buffer assignment.

Figure 8.4: Algorithm for computing an optimal nonblocking buffer assignment.

The time between when a message can arrive at process i and when it is received by the receive call corresponding to vertex $v_{i,c}$ is represented by the interval $I_{i,c}$. Each of these intervals must have a buffer to ensure nonblocking sends. Hence, the minimum number of buffers, b_i , is the maximum overlap over all intervals within process p_i . See page 155 for a detailed description on the specific techniques for computing the maximum overlap in polynomial time.

To illustrate the algorithm, consider an implementation of the parallel pipe-and-roll matrix multiplication algorithm as described in Figures 7.6 and 7.7 in Chapter 7. In this instance, let us consider a system with one master process and four slave processes arranged in a 2×2 grid.

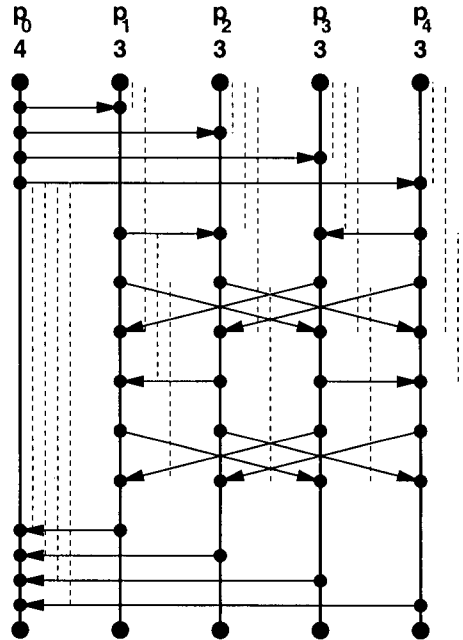


Figure 8.5: Communication graph with buffer intervals for a 2×2 worker configuration of the pipe-and-roll matrix multiplication algorithm

Figure 8.5 illustrates the communication graph created based on the messages sent when the algorithm is executed. P_0 is the master process, and P_1, \dots, P_4 are the slave processes. The dotted vertical lines represent the I intervals computed by the NBAP algorithm. Remember, the beginning of an interval signifies the earliest time during that process when the message, received by the receive vertex at the end of the dotted line, can arrive in the communication system in that process. Thus, a buffer must be available for this message during this interval.

Figures 8.6 and 8.7 illustrate the use of this algorithm in Millipede. Millipede maintains information to construct a communication graph based on the messages. This information can be extracted from the relations *Senders* and *Receivers*, or from a set of message log files.

By executing the command `nbap`, the current program loaded into Millipede is analyzed. The corresponding communication graph is built, and the NBAP algorithm is applied. The output, as seen in Figure 8.6, is a line for each process with its buffer requirements. To further investigate the requirements of a single process, the user can execute the `nbap` command with the process ID of the process in question. Figure 8.7 shows the output from the command `nbap 242167`. The output contains a number of intervals; these are equivalent to the space between two communication nodes on the communication graph. The number in the *Line* column represents the line number in which that interval ends, that is, the line number of either a send or a receive call.

```
(0) |MILLIPEDE> nbap
Master.c
Group / Instance / Tid / Buffers
-----
0 / 0 / 242165 / 4
-----
Slave.c
Group / Instance / Tid / Buffers
-----
0 / 0 / 242167 / 3
0 / 0 / 242169 / 3
0 / 0 / 242171 / 3
0 / 0 / 242173 / 3
-----
```

Figure 8.6: The result of executing the nbap command in Millipede.

```
(0) |MILLIPEDE> nbap 242167
File: Slave.c
Interval Line No Buffers
-----
1 78 2
2 36 1
3 64 2
4 66 3
5 36 2
6 64 1
7 66 1
8 117 0
9 end 0
-----
```

Figure 8.7: By passing the nbap command a process identifier, detailed information about its buffer requirements is displayed.

Another important aspect of the NBAP algorithm is that if the system can provide the number of buffers suggested by the algorithm, all sends become nonblocking, which optimise's the ability to overlap communication and computation. In order to make efficient use of asynchronous message passing, it is important to ensure that no send operations block.

8.4 Approximations of BAP using NBAP

As shown the Buffer Allocation Problem is NP-hard for all four buffer placement schemes. As mentioned, an obvious approximation to BAP is the result computed by the NBAP algorithm; not only does it guarantee that a program with that many buffers does not contain any blocking sends, but a side effect is that no deadlocks due to insufficient buffers ever occurs. Unfortunately, this approximation is not always sufficient. Consider the two communication graphs shown in Figure 8.8. For the left graph labelled (a), the NBAP algorithm suggests a number k , where k is the number of messages, in this case 8. However, the correct number of buffers for this graph to avoid deadlocks is 0; the graph represents a 0-safe program, which can be executed synchronously, thus requiring no buffers at all. For the graph labelled (b), the algorithm computes the value 1.

Again, this communication graph represents a programs that is 0-safe, but the approximation of 1 buffer for (b) is a much tighter upper bound than 8 is for (a).

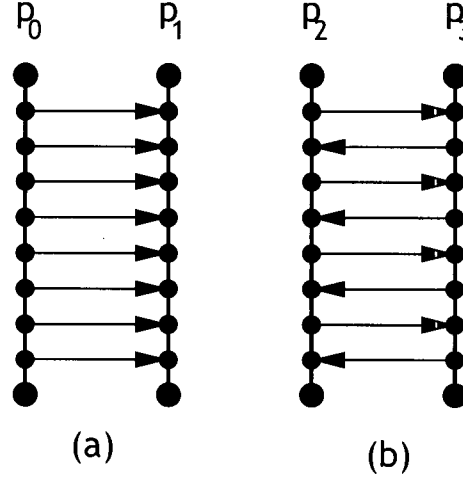


Figure 8.8: (a) shows an example for which the NBAP output is a worst case approximation. The graph in (b) is an example where NBAP is within 1 from the optimal result.

We attempt to counter this problem by introducing epochs. Intuitively, an epoch in a communication graph is any set of vertices that can be separated from the rest of the graph by two horizontal lines cutting the graph in three parts, such that exactly one vertical arc in each process is crossed by each line, and no communication arcs are crossed. Actually, all three parts, above the top line, between the lines and below the bottom line, are epochs.

Consider a communication graph $G(S)$ with n processes. A set e of n pairs of vertices (v_{i,c_i}, v_{i,k_i}) , one pair for each process in $G(S)$, represents an epoch if the following holds. Define $\bar{E}(e)$ as follows:

$$\bar{E}(e) = \{v_{i,l} \mid (v_{i,c_i}, v_{i,k_i}) \in e, i \in [0, n), l \in [c_i, k_i]\}$$

such that for all $v \in \bar{E}(e)$, if v is a send event, the corresponding receive event v_r is in $\bar{E}(e)$, or if v is a receive event, the corresponding send event v_s is in $\bar{E}(e)$. The vertices in the epoch are exactly the vertices in $\bar{E}(e)$.

Figure 8.9 shows an example of dividing the communication graph from Figure 8.5 into three epochs. The significance of an epoch is that it is self contained, that is, no communication within an epoch involves any communication events outside that epoch.

Assume that we have computed the minimum number of buffers needed to assure nonblocking sends, that is, the result of the NBAP algorithm. We know this is an upper bound for BAP. An example of parts of such a graph is shown in the left part of Figure 8.10. Consider a process arc

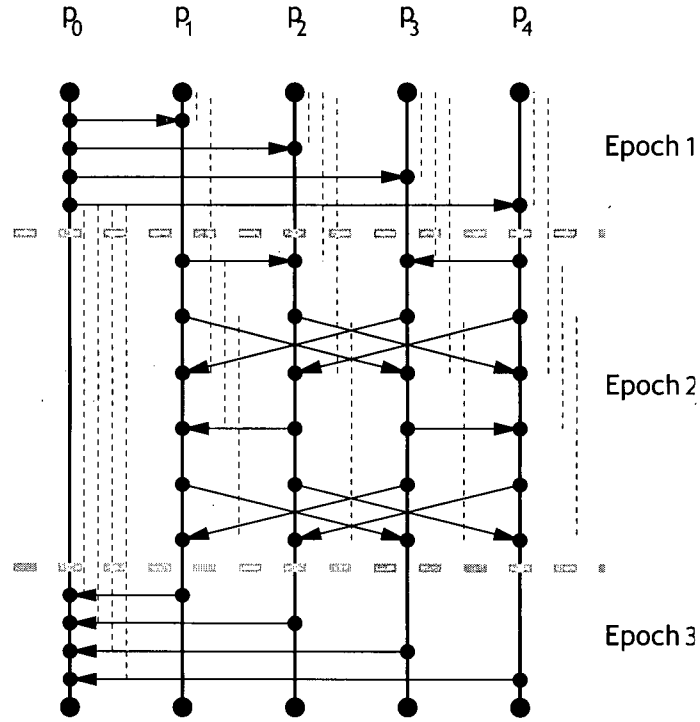


Figure 8.9: The communication graph for the matrix multiplication system with three epochs introduced.

$(v_{i,c}, v_{i,c+1})$, and an epoch e where $v_{i,c} \notin \bar{E}(e)$ and $v_{i,c+1} \in \bar{E}(e)$, that is, the process arc crosses an epoch boundary.

If the line dividing the communication graph (the epoch line) is replaced by a barrier synchronization, the intervals, representing buffer requirements, that start before the barrier can now be shortened to start at the barrier. This is illustrated in the right side of Figure 8.10. Thus, the more epochs the communication graph is divided into, the better the chances of reducing buffer requirements. However, barriers do force the processes to synchronize, which can lead to a significant slow down, with respect to execution speed. This means there is a trade off between the number of buffers required and the number of barriers added; safety is traded for execution speed.

The barriers we suggest are equivalent to the epoch boundaries, and involve all the processes in the system. This might be an unnecessarily conservative approach. If a number of the processes require more buffers than others, it might be sufficient to focus on these processes.

We can define sub-epochs as self contained sets, like $\bar{E}(e)$, that do not involve all n processes, but still ensure that both ends of a communication arc are included in the epoch for the involved processes. The boundaries of such sub-epochs can be replaced by barrier synchronization between

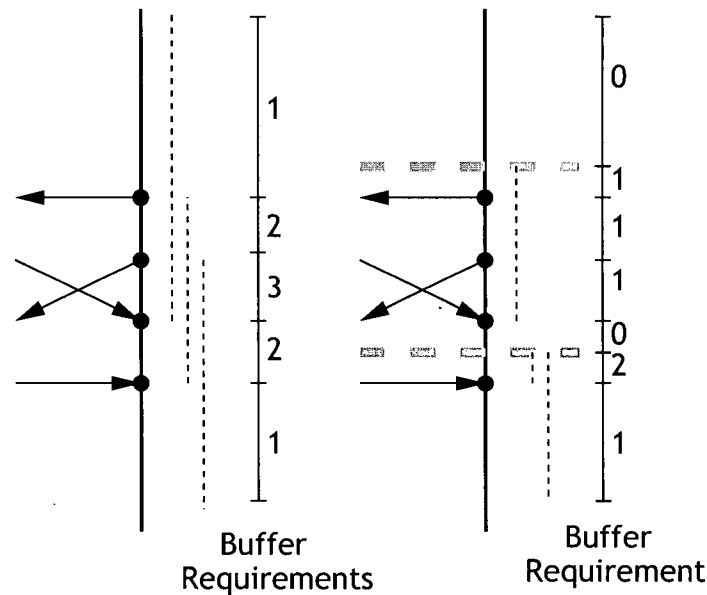


Figure 8.10: By creating epochs, buffer requirements are reduced. Intervals no longer cross epoch boundaries.

the processes included in the epoch. Figure 8.11 shows an example of a sub-epoch.

An even less restrictive approach is to add synchronization points between just two processes. In practice, this is equivalent to making a message passing call synchronous. This is easily achieved in MPI by using the synchronous message passing calls rather than the asynchronous ones.

Barriers are by definition synchronous, but our model assumes asynchronous communication. However, barriers can be simulated using asynchronous communication, as shown in Figure 8.12.

Simulating a synchronous message passing call in the corresponding asynchronous communication graph can be done by adding an 'ack-like' message in the opposite direction of the original message, thus making the call and the added 'ack' look like a mini barrier between two processes only.

One of the important design goals for a tool like Millipede is to easily allow the user to map a problem back to the source code. Millipede provides this mapping through the `nbap` command. When `nbap` is executed with a process identifier, as seen in Figure 8.7, the buffer requirements for each interval are shown with corresponding line numbers. The intervals 2 through 6 are equivalent to the left side of Figure 8.10. By adding a barrier immediately before the line `T = Pipe_A()` in Figure 7.7, and re-executing the `nbap` algorithm, the buffer requirements change; the maximum is now 2 instead of 3, as previously.

With the information about the buffer requirements for each interval, it is easy to search the list for the intervals with the largest numbers, return to the source code for examination, and

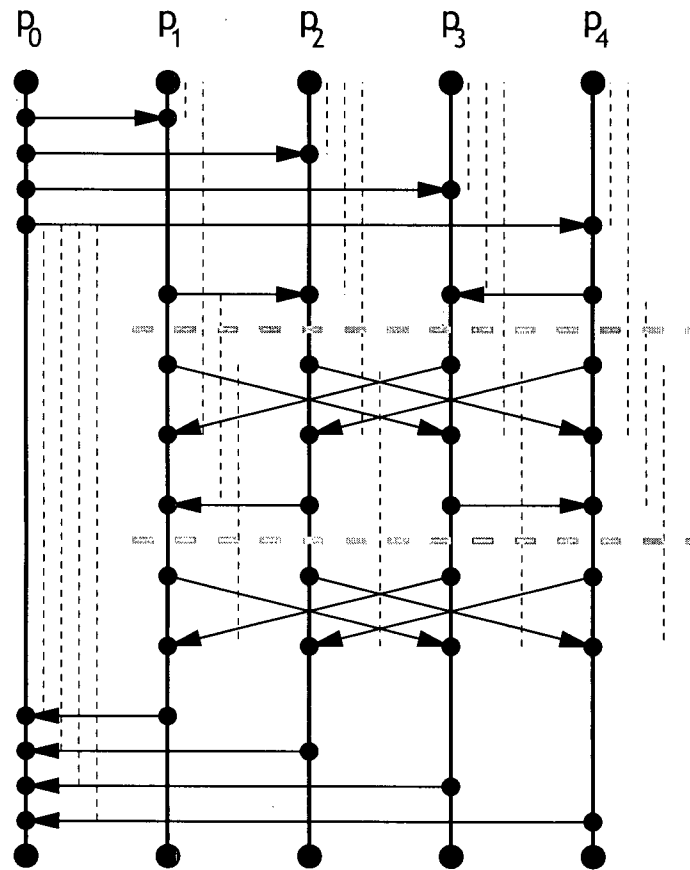


Figure 8.11: An illustration of the use of sub-epochs; epochs that do not involve all processes. Processes $P_1, P_2, P_3,$ and P_4 all participate in a sub-epoch. Barrier synchronizations between these four processes can then be inserted at the epoch boundaries.

possibly insert barriers, or make some calls synchronous.

8.5 Discussion

A number of techniques can be amalgamated into a new analysis tool for the BAP approximation algorithm. Since the user might not want to work with the communication graph, certain automated tasks can be implemented:

- Communication in loops should be automatically recognized in the communication graph, which then could be 'rolled up'. Often, buffer requirements within loops only slightly differ, so by rolling up loops, it becomes clear where barriers can be placed so that they do not interfere with loops.
- Automatic recomputation of buffer requirements after inserting barriers or synchronous

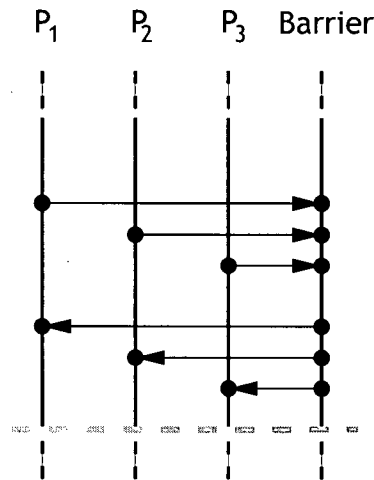


Figure 8.12: Implementation of barrier synchronization using asynchronous communication. No process can cross the dashed line before every process is ready.

communication without re-executing the program would make the development/debugging cycle much more efficient and shorter.

8.6 Summary

In this section we presented three problems related to buffer allocation in asynchronous message passing systems, the Buffer Allocation Problem, the Buffer Sufficiency Problem, and the Non-blocking Buffer Allocation Problem. We considered them under four different buffer placements schemes (send side buffers, receive side buffers, mixed send and receive side buffers, and buffers placed on communication channels), and proved that the most general problem of determining the minimum number of buffer needed to assure deadlock free execution is intractable. We presented a polynomial time algorithm for computing the minimum number of buffers needed to assure that no send ever blocks. In addition, we showed how to use the NBAP algorithm along with barriers or synchronous communication calls to approximate the solution to the Buffer Allocation Problem.

The majority of the work concerning buffer allocation is found in Appendix E, and while most of the problems presented are intractable (see Table 9.1), the results themselves are interesting and form a solid starting point, and offers insight into the problem for further research into heuristics and approximations.

Chapter 9

Conclusion and Future Work

“If you have an important point to make, don’t try to be subtle or clever. Use a pile driver. Hit the point once. Then come back and hit it again. Then hit it a third time with a tremendous whack.”

— *Winston Churchill*

“Always do one thing less than you think you can do”

— *Bernard Baruch*

9.1 Conclusion

In this dissertation we described a debugging strategy for parallel message passing programs, called multilevel debugging. To validate the thesis, we develop a number of tools and analyses to support it. These tools are realized as modules in a prototype implementation of a multilevel debugger, called Millipede(Multilevel Interactive Parallel Debugger). Millipede is implemented for C programs that use the PVM message passing library. The multilevel debugging strategy is based on a number of observations about parallel programming, debugging, and the shortcomings of existing tools.

A parallel system decomposes naturally into three parts: sequential code, messages, and the overall communication protocol. Therefore, it is natural to tailor debugging of such systems to this structure; errors are classified according to the three levels in which they occur. We show that by providing tools specifically tailored to each level, certain tasks and analyses, which are otherwise tedious, error prone, and time consuming, become easier to accomplish. The

narrower focus enables the automation of certain tasks, such as deadlock correction and protocol conformance checking. Additionally, the design goals include avoiding information overload, computing relations on request, and providing views for 'key players' at each level. One main problem with existing tools, which we address with the multilevel debugging strategy, is the set granularity, or lack of support for certain debugging tasks. An example of this is the application of N versions of a sequential debugger, such as Gdb, to N different processes in a parallel system.

We show that a bottom up approach with a number of specially tailored tools is useful for debugging parallel message programs. We verify this thesis by implementing a variety of tools that support the multilevel debugging strategy at each of the three levels.

9.1.1 The Sequential Level

At the sequential level—the lowest level—we reason that since a number of very good tools, such as Gdb and Purify already exist, the missing functionality is the ability to apply these tools to a single process of a multiprocess system. To facilitate the use of existing debugging tools, we add message logging functionality to the runtime system of Millipede; these log files contain information about the messages in the system. By intercepting all message passing calls, the runtime system replays messages read from log files rather from the network, facilitating the sequential debugging of one process of a parallel system—as if it were a stand-alone program. We successfully validated this approach by demonstrating how a number of sequential debugging tools can be applied to one process of a parallel system. One of the key points in the design of this tool is the reduction in the amount of information presented to the user, as well as assuring that the abstraction of the tool is correct. Nondeterminism in the sequential code limits this tool's usefulness. If the sequential code does not receive the messages in the same order every time the code is executed the order of the messages in the log files will be incorrect. However, similar problems exist in the sequential domain; errors can be hard to reproduce with nondeterministic code.

9.1.2 The Message Level

At the message level—the second level—it is difficult to perform debugging tasks that involve controlling and keeping track of messages. We counter this problem by providing two tools at this level. The first is an extension of the well known idea of inspecting and changing variables in a sequential program. We extend that idea to the Message Level, by providing functionality in the runtime system to allow the user to chose a number of processes to perform interactive message inspection on; as messages are delivered to the process, the user can choose to intercept them for inspection. In addition, if errors occur during unpacking, the user is automatically informed and can focus his attention on correcting the problem. This technique can be used in connection with

the Sequential Debugging Module as well; if no log files exist, a program can still be debugged sequentially—messages can be provided manually by the user or from a file. This provides a way to test a single processes of a multiprocess system; even if the rest of the system is not yet implemented. An important result of this tool is the ability to view and manipulate messages a unit; if the need arises, the user can investigate the source and destination variables of the values in the message. The level of abstraction can be tuned to match the user's needs. In addition, the tool will automatically notify the user about errors caused by unpacking too much data, immediately enabling him to identify the packing and unpacking routines that were involved with the message. One potential draw back with interactive message debugging is the potentially large amount of data the user must manually supply if the tool is being used for unit testing. We have provided the user with the ability to read values from a file; this avoids errors due to typing, but the problem of producing the data in the file still remains.

The second tool at the message level is a query language, the Millipede Query Language (MQL), which allows the user to form queries about the messages. If the user is interested in discovering certain facts about messages, it is an almost impossible task if he has to manually browse through a large set of messages. Using MQL he can easily form a query that selects the messages that he is interested in, thus computing relations when needed and reducing the information overload. Relations containing information about the messages are maintained by the runtime system, and at any time the user can form queries related to the messages and their content. We show a number of different useful queries as an example of the expressiveness of MQL. This tool is based on the design goal of 'computing relations on request'. It follows that this technique reduces the information overload that could otherwise appear when perusing the large relations containing information about the messages and their content. MQL is extensible, thus implementing support for more complex queries is straightforward. Queries in MQL are limited by the the language itself, but also by the operations provided by the underlying database system. One possibility is to replace MQL with an implementation of the complete SQL specification and use a state-of-the-art relational database.

9.1.3 The Protocol Level

At the protocol level—the third of the three levels—we develop three different tools and perform an in-depth analysis of the problem of determining the minimum number of buffers needed to ensure that an asynchronous message passing program does not deadlock due to buffer insufficiencies.

The first tool is the Deadlock Detection and Correction Tool. When a parallel system deadlocks, a global overview is often required to gain the knowledge needed to remove the deadlock. The tool automatically computes a possible solution, which is a number of changes to the source code

that will remove the deadlock. We present an algorithm that does this and is based on computing a maximal matching in a bipartite graph, where the nodes represent the senders and receivers in the parallel system. We show that the probability of the algorithm suggesting an incorrect way to resolve the deadlock is small. This tool allows the user to query the system for potential solutions. Again, a relation is computed automatically. Computing this relation by hand can be time consuming and error prone. We show that the number of times the algorithm suggests a wrong solution, when the number of errors in the system is less than half the number of processes, is sufficiently small (between 0% and 11% for the number of processes less than or equal to 10). This tool illustrates how certain tasks can be automated when the level of abstraction is raised. In addition to the automation of the task, the tool suggests a solution to remove the error, which is something made possible by the level of abstraction and automation as well.

The second tool at the protocol level is the Protocol Conformance Checking Tool. The idea we investigated is how to provide an easy way for the user to automate checking that the messages adhere to the specification of the intended protocol. The tool allows the user to write a simple specification of a protocol, and then have all messages checked against that specification as the program is running. This tool reduces the gap between a protocol specification (even a protocol specification that has been verified using verification tools) and implementation, by aiding the user in verifying that the implemented protocol matches the specification. An obvious downside with such a tool is that it is impossible to guarantee that the constraint specification the user provides matches the protocol. On the other hand, the specification language is small, and if used in an iterative fashion along with the program development, we have shown that it is a feasible approach to take to automatically verify that messages adhere to the provided specification. One important issue is the tool's ability to map the error back to the source code. That is, once a message has violated the specification, information about the sender and the receiver, as well as the lines involved in packing and unpacking the message are reported to the user. One of the limiting factors is that the tool does not take any temporal or timing issues into account; it is concerned with spatial issues only, that is, the message passing pattern.

The last tool at this level originates from this theoretical question: "in a system that uses asynchronous buffered nonblocking sends and blocking receives, can we determine the minimum number of buffers required by a message passing program such that it never deadlocks due to lack of buffers?" This problem is known as the k -safety problem, and it led to an in-depth investigation of three related questions: one, "can we compute a minimal buffer assignment needed to avoid deadlock?"; two, "can we verify that an assignment is sufficient?"; and three, "can we compute the minimal buffer assignment needed to avoid blocking sends?" We determine the complexities for all three questions; the first two are NP-hard, whereas the last one has a known polynomial time solution. We investigate these three problems, the Buffer Allocation Problem (BAP), the

Buffer Sufficiency Problem (BSP), and the Nonblocking Buffer Allocation Problem (NBAP), respectively, with four different buffer allocation schemes: buffers placed on the receiver's side, on the sender's side, on both sides, and on channels (under the assumption that we use channel-like communication). The result of this investigation is as follows:

- The Buffer Allocation Problem is intractable under all four buffer allocation schemes.
- The Buffer Sufficiency Problem is intractable for the receive side buffer and for the mixed buffer allocation schemes, tractable for the channel scheme and conjectured to be tractable for sender side buffers.
- The Nonblocking Buffer Allocation Problem is tractable for all buffer placement schemes, except the mixed send and receive scheme.

Table 9.1 summarizes these results.

Problem	Buffer Placement			
	Receive	Send	Send & Receive	Channel
BAP	NP-hard	NP-hard	NP-hard	NP-complete
BSP	coNP-complete	(P)	coNP-complete	P
NBAP	P	P	NP-hard	P

Table 9.1: Results for the three problems under the four different buffer placements schemes.

In addition, we provide an implementation of the NBAP algorithm in Millipede, and show how this algorithm, combined with a number of techniques for inserting synchronization points into the code, can be used to reduce the number of buffers required to ensure k -safety. Since the number of buffers needed to ensure nonblocking sends (the result of the NBAP algorithm) is an upper bound for the original BAP problem, reducing the number of buffers necessary for NBAP provides an improved approximations for BAP. This work was done in collaboration with Alex Brodsky [BBW02].

9.1.4 Summary

The decomposition of the parallel programming domain into three levels led to a bottom-up approach to debugging, referred to as multilevel debugging. We have implemented a number of tools in accordance with this strategy. These tools are tailored to a specific error type at one of the three levels of the parallel programming domain. In addition to serving as a debugging framework the multilevel strategy provides a guide for implementing new tools within the framework.

By implementing various tools we have shown that it is possible to utilize the extra information that can be extracted from a parallel program to raise the level of abstraction within each tool; in turn, this allows for automation and analyses that otherwise would be impossible.

Furthermore, the decomposition and the narrowed focus on one specific error type for each tool has reduced the information overload that is eminent in many existing tools, and allowed us to meet a number of design goals for tools in general, and for debugging tools specifically. These include views for 'key players', automatic relation computation, automated analyses, and automatic computation of possible solutions to errors.

All these tools have been implemented as modules in Millipede, a prototype multilevel debugger with a simple command line interface. Millipede does not require any rewriting or transformation, and is thus applicable directly to the source code, which is another important design goal.

9.2 Future Work

During the design and implementation of Millipede, a number of interesting issues and suggestions for improvements have arisen. The following sections describe some of the research directions future work can take, and also describe issues that should be addressed to make Millipede fully functional.

9.2.1 Further Development

The current version of Millipede is written for PVM, and all PVM functions have not yet been implemented. A first step is to complete the implementation of the Millipede runtime system to support all PVM functions. A natural next step is to implement a version of the runtime system for MPI.

Furthermore, providing a simple GUI for interacting with Millipede has certain advantages. One of the more important advantages is its ability to easily manage a number of windows through the use of tabs, or cascading panes. A simple GUI can be implemented using Tcl/Tk, which provides call back functionality to C.

In Chapters 4 through 8 we describe in detail improvements and future work for each tool. We highlight some of the more important ones here.

Since log files can become quite large, and since replaying a process from the start can take too long, one of the most important improvements for the runtime system is to add check pointing and log file truncation. Netzer and Xu describe an efficient way to checkpoint and maintain log file consistency in [NX93]. Implementing a similar scheme for Millipede will improve its ability to debug long running applications with large message data sets.

The database system used in connection with MQL is written in C and is part of Millipede. By using existing databases that provide access through C functions, not only do we get a better database, but it also becomes easier to extend MQL with new functionality that might already be found in the native SQL dialect of the chosen database.

Breakpoints are well known in the sequential debugging domain; we believe that this concept can be extended to messages as well. Message breakpoints abstracts away line numbers, and lets the user control program execution, based on the messages.

The algorithm for correcting deadlocks, described in Chapter 6, does not take message tags into account. An interesting extension to this work is to develop an algorithm that also considers message tags. In addition, a new analysis of the quality of such an algorithm should be performed.

For the Protocol Conformance Checking Tool, a number of simple improvements have been suggested in Section 7.10. Two examples are adding message tags to specification lines and allowing the user to write specification lines where communication is dependent on program state. An interesting extension to this module would be a graphic display showing the flow of the messages as the program executes. Such an interface can be combined with the message relations described in Chapter 5 to provide a graphical replay of the protocol.

Finally, the Buffer Sufficiency Problem for the mixed buffer allocation scheme should be investigated. If we can show that the problem is tractable, an algorithm can be developed and added to Millipede. Further research should be done on an approximation heuristic for the Buffer Allocation Problem. We only suggest simple measures, but we believe that more work can be done to provide better approximations.

Bibliography

- [AFC91] K. Araki, Z. Furukawa, and J. Cheng. A General Framework for Debugging. *IEEE Software*, pages 14-20, May 1991.
- [Ana89] V. Anantharm. The optimal buffer allocation problem. *IEEE Transactions on Information Theory*, 35(4):721-725, 1989.
- [Arv92] D. K. Arvind. On the detection of communication related errors in parallel programs. *Parallel computing*, 18:1381-1392, 1992.
- [Asp03] The AspectJ Team, Xerox Corporation, Palo Alto Research Center. The AspectJTM Programming Guide, 2003.
- [BB97] B. B. Blendstrup and J. B. Pedersen. PVMbuilder - et grafisk værktøj til parallel programmering. Master's thesis, Aarhus Universitet, January 1997.
- [BBW02] A. Brodsky, J. B. Pedersen, and A. Wagner. On the Complexity of Buffer Allocation in Parallel Message Passing Systems. In *Communicating Process Architectures 2002*. IOS Press, September 2002.
- [BD95a] G. Burns and R. Daoud. Robust MPI Message Delivery with Guaranteed Resources. MPI Developers Conference at the University of Notre Dame, June 1995.
- [BD95b] G. Burns and R. Daoud. Robust MPI Message Delivery with Guaranteed Resources. MPI Developers Conference at the University of Notre Dame, June 1995.
- [BDH⁺95] J. Bruck, D. Dolev, C. Ho, M. Rosu, and R. Strong. Efficient message passing interface (mpi) for parallel computing on clusters of workstations. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 64 - 73, July 1995.
- [BK95] P. A. Buhr and M. Karsten. *μC++ Monitoring, Visualization and Debugging Annotated Reference Manual*, Preliminary draft edition, November 1995.
- [BS95] P. A. Buhr and R. A. Strooboscher. *μC++ Annotated Reference Manual, Version 4.4*, Available via ftp from `plg.uwaterloo.ca` in `pub/uSystem/uC++.gz`, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1 edition, September 1995.
- [BW00] J. B. Pedersen and A. Wagner. Sequential Debugging of Parallel Programs. In *Proceedings of the international conference on communications in computing, CIC'2000*. CSREA Press, June 2000.

- [BW01a] J. B. Pedersen and A. Wagner. Correcting Errors in Message Passing Systems. In F. Mueller, editor, *High-Level Parallel Programming Models and Supportive Environments, 6th international workshop, HIPS 2001 San Francisco, CA, USA*, volume 2026 of *Lecture Notes in Computer Science*, pages 122-137. Springer Verlag, April 2001.
- [BW01b] J. B. Pedersen and A. Wagner. Protocol Verification in Millipede. In *Communicating Process Architectures 2001*. IOS Press, September 2001.
- [CE81] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *Logic of Programs: Workshop, Yorktown Heights, NY*, 131, May 1981.
- [CFR95] C. Cl  men  on, J. Fritscher, and R. R  hl. Visualization, Execution Control and Replay of Massively Parallel Programs within Annai's Debugging Tool. In *Proceedings of High-Performance Computing Symposium*, pages 393-405, July 1995.
- [CL94a] R. Cypher and E. Leu. Repeatable and portable message-passing programs. In *Proc. of The Symposium on the Principles of Distributed Computing (PODC)*, pages 22-31, 1994.
- [CL94b] R. Cypher and E. Leu. The semantics of blocking and nonblocking send and receive primitives. In *Proceedings of 8th IEEE International parallel processing symposium (IPPS)*, pages 729-735, 1994.
- [CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 353-362. IEEE Computer Society Press, June 1989.
- [CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT press, 1990.
- [CMT96] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous, asynchronous, and causally ordered communication. *Journal of Distributed Computing*, 9(4):173-191, 1996.
- [Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing*, pages 151-158, 1971.
- [CSH02] R. Corbett, R. Stallman, and W. Hansen. Bison 1.35, May 2002. <http://www.gnu.org/manual/bison/index.html>.
- [DHHW93] J. Dongarra, R. Hempel, A. Hey, and D. Walker. A proposal for a user-level, message-passing interface in a distributed memory environment. Technical Report TM-12231, ORNL, June 1993.
- [Dil96] D. L. Dill. The Mur   Verification System. In *8th International Conference on Computer Aided Verification*, pages 390-393, July/August 1996.
- [Don] J. Dongarra et al. HeNCE: A Users' Guide. Version 2.0. <http://www.netlib.org/hence>.
- [Don94] J. Dongarra. MPI: A Message Passing Interface Standard. *The International Journal of Supercomputers and High Performance Computing*, 8:165-184, 1994.

- [Eis97] M. Eisenstadt. My hairiest bug war stories. In *The Debugging Scandal and What to Do About It - Communication of the ACM*. ACM Press, April 1997.
- [FBH⁺92] D. Frye, R. Bryant, H. Ho, R. Lawrence, and M. Snir. An external user interface for scalable parallel systems. Technical report, IBM highly parallel supercomputing systems laboratory, November 1992.
- [FJL⁺88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving problems on concurrent processors. General techniques and regular problems*, volume 1. Prentice Hall International, 1988.
- [For] Formal Systems. FDR2. <http://www.fsel.com>.
- [Fos95] I. Foster. *Designing and Building Parallel Programs: Concepts and tools for parallel software engineering*. Addison Wesley, 1995.
- [GBV94] R. Daoud G. Burns and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Supercomputing Symposium '94*, June 1994.
- [Gdb] Gdb - GNU Debugger. <http://www.gnu.org/directory/gdb.html>.
- [Gei94] A. Geist et al. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*. Prentice Hall International, 1994.
- [GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.
- [GMP89a] A. Giacalone, P. Mishra, and S. Prasad. Facile: A Symmetric Integration of Concurrent and Functional Programming. *Proceedings of the 1989 TAPSOFT Conference*, 352, 1989.
- [GMP89b] A. Giacalone, P. Mishra, and S. Prasad. Facile: A Symmetric Integration of Concurrent and Functional Programming. *International Journal of Parallel Programming*, 18(2), 1989.
- [Gra86] J. Gray. Why do Computers Stop and What Can be Done About it? *Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems*, pages 3-12, January 1986.
- [HE93] M. T. Heath and J. A. Ethridge. ParaGraph: A Tool for Visualizing Performance of Parallel Programs. Technical report, Technical Report Oak Ridge National Laboratories, 1993.
- [Hei97] F. Heinze et al. Trapper, Eliminating Performance Bottlenecks in a Parallel Embedded Application. *IEEE Concurrency*, pages 28-37, July-September 1997.
- [HJJJ85] P. Huber, A. M. Jensen, L. O. Jepsen, and K. Jensen. Reachability trees for high-level Petri nets. *Theoretical Computer Science*, 45:261-292, 1985.
- [Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666-677, August 1978.
- [Hol97] G. J. Holzmann. The Spin Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279-295, May 1997.

- [Hoo96] R. Hood. The p2d2 Project: Building a Portable Distributed Debugger. In *Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'96)*, pages 127-136, May 1996.
- [IBM02] IBM Press Release. IBM Executive Says Grids Will Be A Breakthrough For Managing IT Efficiency, June 2002. <http://www-916.ibm.com/press/prnews.nsf/jan/F23B8EA466B5569085256BDC0064024B>.
- [Jen92] K. Jensen. *Coloured Petri nets. Basic Concepts, Analysis Methods and Practical use*, volume 1. Springer Verlag, 1992.
- [Joh83] W. L. Johnston. An Effective Bug Classification Scheme Must Take the Programmer into Account. *Proceedings of the workshop of High-level debugging*. Palo Alto, California, 1983.
- [Kar95] M. Karsten. *A Multi-Threaded Debugger for Multi-Threaded Applications*, Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Mannheim, Deutschland edition, August 1995.
- [KG96] J. A. Kohl and G. A. Geist. The PVM 3.4 Tracing Facility and XPVM 1.1. *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, pages 290-299, 1996.
- [KLK99] P. Kacsuk, R. Lovas, and J. Kocács. Systematic Debugging of Parallel Programs in DIWIDE Based on Collective Breakpoints and Macrosteps. In *Proceedings of the 5th International Euro-Par Conference (Euro-Par'99)*, volume 1685 of *Lecture Notes in Computer Science*, pages 90-97. Springer Verlag, August 1999.
- [Knu89] D. E. Knuth. The Errors of \TeX . *Software - Practise and Experience*, 19(7):607-685, July 1989.
- [KV97] D. Kranzlmüller and J. Volkert. Using Different Levels of Abstraction for Parallel Programming Debugging. In *Proceedings of the 1997 IASTED (International Conference on Intelligent Information)*, pages 523-529, 1997.
- [KW01] C. Keppitiyagama and A. Wagner. Asynchronous MPI messaging on myrinet. In *Proceedings 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*. IEEE, 2001.
- [Lam78] L. Lamport. Time, clocks and the orderings of events in a distributed system. *Communications of the ACM*, 21:558-565, 1978.
- [Lel88] Wm Leler. *Constraint Programming Languages — Their Specification and Generation*. Addison-Wesley, 1988.
- [LLM88] M. Litzkow, M. Livny, and M. Mutka. Condor: A Hunter of Idle Workstations. *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104-111, June 1988.
- [May83] D. May. OCCAM (language). *ACM SIGPLAN Notices*, 18(4):69-79, April 1983.
- [McM92] Ken McMillan. *Symbolic Model Checking: An Approach to the State Space Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.

- [MHC94] B. P. Miller, J. K. Hollingsworth, and M. D. Callaghan. The Paradyn Parallel Performance Measurement Tools and PVM. *Environments and Tools for Parallel Scientific Computing*, 1994.
- [Mon13] P. de Montmort. On the game of thirteen. 1713. Reprinted in *Annotated Readings in the History of Statistics*, ed. H. A. David and A. W. F. Edwards, Springer Verlag, 2001, pp. 25-29.
- [NAW⁺96] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer* 63, XII(1):69-80, January 1996.
- [NC92] P. Newton and J. C. Browne. The CODE 2.0 Graphical Parallel Programming Language. *Proceedings of the ACM International Conference on Supercomputing*, July 1992.
- [ND94] P. Newton and J. Dongerra. Overview of VPE: A Visual environment for Message-Passing, 1994. <http://www.cs.utk.edu/newton/vpe/vpe.html>.
- [Net] NetSolve. <http://icl.cs.utk.edu/netsolve>.
- [NX93] R. H. B. Netzer and J. Xu. Adaptive message logging for incremental replay of message-passing programs. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 840-849. ACM Press, 1993.
- [NY93] P. Newton and S. Y. Khedekar. *CODE 2.0 User Manual*, March 1993.
- [Pal99] Pallas GmbH. TotalView, 1999. <http://www.pallas.de/pages/totalv.htm>.
- [Pan93a] C. M. Pancake. Graphical Support for Parallel Debugging. *Software Support for Parallel Computation*, pages 216-228, 1993.
- [Pan93b] C. M. Pancake. Why Is There Such a Mis-Match between User Need and Parallel Tool Production? Keynote address, 1993 Workshop on Parallel Computing Systems: A Dialog between Users and Developers, April 1993.
- [Pan93c] C.M. Pancake et al. Results of User Surveys Conducted on Behalf of Intel Supercomputer Systems Division, Two Divisions of IBM Corporation, and CONVEX Computer Corporation, 1989-1993.
- [Pan94] C. M. Pancake. What Users Need in Parallel Tool Support: Survey Results and Analysis. Technical Report CSTR 94-80-3, Oregon State University, June 1994.
- [Pan99] C. M. Pancake. Applying Human Factors to the Design of Performance Tools. In *Proceedings of the 5th International Euro-Par Conference (Euro-Par'99)*, volume 1685 of *Lecture Notes in Computer Science*, pages 44-60. Springer Verlag, August 1999.
- [Pax98] V. Paxson. Flex - a scanner generator, November 1998. <http://www.gnu.org/manual/flex-2.5.4/flex.html>.
- [Pdb] pdbx and pedb: Parallel Program Debuggers. <http://www.tc.cornell.edu/UserDoc/Software/PTools/pdbx>.
- [Pre92] O. Pretzel. *Error-Correcting Codes and Finite Fields*. Clarendon Press, 1992.

- [Pur] Rational Purify for UNIX. <http://www.rational.com/products>.
- [Rei87] M. Reiman. The optimal buffer allocation problem in light traffic. In *IEEE Conference on Decision and Control*, 1987.
- [Ros93] A. W. Roscoe. Developing and verifying protocols in CSP. *Proceeding of the protocol verification workshop, Mierlo, The Netherlands*, March 1993.
- [Ros94] A. W. Roscoe. Model-Checking CSP. *A classical mind, essays in honour of C.A.R. Hoare*, 1994.
- [RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. CML: a higher-order concurrent language. *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, January 1988.
- [SA97] R. Sasic and D. Abramson. Guard: a Relative Debugger. *Software - Practise and Experience*, 27(2):185-206, February 1997.
- [San99] A. A. Sane. *Techniques for Developing Correct, Fast and Robust Implementation of Distributed Protocols*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.
- [She75] T. Sheskin. Allocation of interstage storage along an automatic production line. *AIEE Transactions*, 8(1), 1975.
- [Sri95] R. Srinivasan. XDR: External Data Representation Standard. RFC 1832, Sun Microsystems, August 1995.
- [SSG91] A. Singh, J. Schaeffer, and M. Green. A template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations. *IEEE Transactions on parallel and distributed systems*, 2(1):52-67, January 1991.
- [SSP85] J. C. Spohrer, E. Soloway, and E. Pope. A Goal/Plan Analysis of Buggy Pascal Programs. *Human-computer Interaction*, 1(2):163-207, 1985.
- [SSS90] A. Singh, D. Szafron, and J. Schaeffer. Experience with parallel programming using code templates. *Concurrency, Practise and Experience*, 10:91-120, March 1990.
- [The96] The VIS Group. VIS: A system for Verification and Synthesis. *Proceedings of the 8th International Conference on Computer Aided Verification*, 1102, July 1996.
- [Top] The top 500 fastest computers. <http://www.top500.org>.
- [TSS96] B. Topol, V. Sunderam, and J. Stasko. PVaniM 2.0, 1996. <http://www.cc.gatech.edu/gvu/softviz/parviz/pvanimOL/pvanimOL.html>.
- [WA98] G. Watson and D. Abramson. Finding Errors in Data Parallel Programs: A Case Study. May 1998. <http://www.rdt.monash.edu.au/~greg/papers/sc98.html>.
- [WAF02] P. H. Welch, J. R. Aldous, and J. Foster. CSP networking for Java (JCSP.net). *Lecture Notes in Computer Science*, 2330, 2002.
- [XML98] Extensible markup language (XML) 1.0. Technical Report REC-xml-19980210, W3C, February 1998.

- [XWXS96] J. Xiong, D. Wang, W. Xheng, and M. Shen. BUSTER: An Integrated Debugger for PVM. *Proceedings of 1996 IEEE Second International Conference on Algorithms and Architectures for Parallel Processing, ICAPP '96, Singapore*, pages 124-129, June 1996.
- [Yee96] Bennet Yee. A Portable save_world Process Checkpointing Package, 1996. <http://www.cs.ucsd.edu/users/bsy/fun.html>.

Appendix A

A Complete Example of a Millipede Session

In this section, we illustrate a complete example of how to use Millipede to extract a single process from a parallel program, and how to debug such a process sequentially. We consider a master/slave application and extract one of the slave processes. The steps are as follows, and Figure A.1 shows how this session looks in Millipede:

1. First, we compile both the master program and the slave program with the `-DMILLIPEDE` option set. This ensures that the Millipede versions of the message passing calls execute when the parallel program executes.
2. If the environment variable `MILLIPEDE_RCM` is set, log files are generated when the program executes. The program can execute as it is normally, or through Millipede. In this example we show an execution through Millipede.
3. Millipede is started and the name of the master program is passed as a parameter.
4. Using the Millipede command `project <name>`, we specify a project file, which contains all the information about the execution for future debugging purposes.
5. The parallel program can now be executed using the `run`.
6. After exiting Millipede, we can unset the `MILLIPEDE_RCM` environment variable and set `MILLIPEDE_REM`. This instructs the Millipede runtime system that instead of writing log files when executing the message passing calls, log files are read.
7. We can now apply any sequential debugging tool; in this case we execute Gdb with one of the slave processes.

8. When the first PVM call executes, the Millipe runtime system prompts the user for the name of a log file from which the messages are supplied.
9. Sequential debugging now commences as if the program were a sequential program. The Millipe runtime system reads the log file each time a message passing call is made in the code, and supplies the program with values for the variables received through the messages. The programmer can debug, recompile, and re-execute the process with the message log until the errors have been corrected. If the programmer wishes to debug the same process with another set of messages, the program can be restarted with a different log file.

```

(1) gcc -g -DMILLIPEDE -I. -L$PVM_ROOT/lib/$PVM_ARCH/ -o Master Master.c -lpvm3
(2) gcc -g -DMILLIPEDE -I. -L$PVM_ROOT/lib/$PVM_ARCH/ -o Slave Slave.c -lpvm3
(3) setenv MILLIPEDE_RCM
(4) pvm
pvm> quit
pvmd still running
(5) Millipe Master
                                W E L C O M E   T O   T H E
#      #
##    ##      #      #      #      #      #      #      #      #      #      #
# # # #      #      #      #      #      #      #      #      #      #      #
# # # #      #      #      #      #      #      #      #      #      #      #
# # # #      #      #      #      #      #      #      #      #      #      #
# # # #      #      #      #      #      #      #      #      #      #      #
# # # #      #      #      #      #      #      #      #      #      #      #
                                M U L T I - L E V E L   D E B U G G I N G   S Y S T E M
(0) |MILLIPEDE> project Master-slave.prj
Project file is 'Master-slave.prj'
(0) |MILLIPEDE> run
Program 'Master' terminated normally; 0 messages in 0 queues.
(0) |MILLIPEDE> project
----- Project -----
Project name: Master-slave.prj
1 Master.c process(es):
  Group / Instance /      Tid
-----
    0 /           0 / 242165
4 Slave.c process(es):
  Group / Instance /      Tid
-----
    0 /           0 / 242167
    0 /           1 / 242169
    0 /           2 / 242171
    0 /           3 / 242173
----- End of Project -----
(0) |MILLIPEDE> exit
(6) unsetenv MILLIPEDE_RCM; setenv MILLIPEDE_REM
(7) gdb Slave
GNU gdb 5.2
(gdb) break main
Breakpoint 1 at 0x15af8: file Slave.c, line 19.
(gdb) run
19      mytid = pvm_mytid();
(gdb) next
Replay file name: Slave-242171.rpf
.

```

Figure A.1: A complete example of a debugging session using a sequential debugging tool on an extracted process.

Appendix B

The PCSL Grammar and Semantics

B.1 The PCSL Grammar and Semantics

This section contains the grammar and the semantics for the Protocol Constraint Specification Language (PCSL).

Figure B.1 shows the BNF grammar for the PCSL grammar. Note, the symbol ϵ is not a symbol

Protocol	::=	Commlist
Commlist	::=	ϵ Commlist Comm
Comm	::=	LeftClass '->' RightClass Quantifiers ";"
LeftClass	::=	Identifier "[" Index "]" "{" Index "}" "(" Index ")"
RightClass	::=	Identifier "[" ClassExpression "]" "{" ClassExpression "}" "(" ClassExpression ")"
Quantifiers	::=	ϵ ":" QuantifierList
QuantifierList	::=	Quantifier Quantifier "," QuantifierList
Quantifier	::=	forall Identifier ":" RelExpression
Index	::=	ϵ Number Identifier
ClassExpression	::=	ϵ Expression
Expression	::=	Expression "*" Expression Expression "/" Expression Expression "+" Expression Expression "-" Expression Expression "%" Expression Expression "^" Expression "-" Expression "(" Expression ")" "sqrt(" Expression ")" Identifier Number
RelExpression	::=	Expression "<" Expression Expression "<=" Expression Expression ">" Expression Expression ">=" Expression Expression "=" Expression Expression "!=" Expression RelExpression "&&" RelExpression RelExpression " " RelExpression "(" RelExpression ")" "true" "false"

Figure B.1: The PCSL BNF grammar.

$\mathcal{E}[\text{Number}] \sigma$	=	Number	$\mathcal{R}[\text{true}] \sigma$	=	true
$\mathcal{E}[\text{Identifier}] \sigma$	=	$\sigma(\text{Identifier})$	$\mathcal{R}[\text{false}] \sigma$	=	false
$\mathcal{E}[e_1 * e_2] \sigma$	=	$\mathcal{E}[e_1] \sigma * \mathcal{E}[e_2] \sigma$	$\mathcal{R}[e_1 < e_2] \sigma$	=	$\mathcal{E}[e_1] \sigma < \mathcal{E}[e_2] \sigma$
$\mathcal{E}[e_1 / e_2] \sigma$	=	$\mathcal{E}[e_1] \sigma / \mathcal{E}[e_2] \sigma$	$\mathcal{R}[e_1 > e_2] \sigma$	=	$\mathcal{E}[e_1] \sigma > \mathcal{E}[e_2] \sigma$
$\mathcal{E}[e_1 + e_2] \sigma$	=	$\mathcal{E}[e_1] \sigma + \mathcal{E}[e_2] \sigma$	$\mathcal{R}[e_1 \leq e_2] \sigma$	=	$\mathcal{E}[e_1] \sigma \leq \mathcal{E}[e_2] \sigma$
$\mathcal{E}[e_1 - e_2] \sigma$	=	$\mathcal{E}[e_1] \sigma - \mathcal{E}[e_2] \sigma$	$\mathcal{R}[e_1 \geq e_2] \sigma$	=	$\mathcal{E}[e_1] \sigma \geq \mathcal{E}[e_2] \sigma$
$\mathcal{E}[e_1 \% e_2] \sigma$	=	$\mathcal{E}[e_1] \sigma \bmod \mathcal{E}[e_2] \sigma$	$\mathcal{R}[e_1 = e_2] \sigma$	=	$\mathcal{E}[e_1] \sigma = \mathcal{E}[e_2] \sigma$
$\mathcal{E}[e_1 \wedge e_2] \sigma$	=	$\exp(\mathcal{E}[e_1] \sigma, \mathcal{E}[e_2] \sigma)$	$\mathcal{R}[e_1 \neq e_2] \sigma$	=	$\mathcal{E}[e_1] \sigma \neq \mathcal{E}[e_2] \sigma$
$\mathcal{E}[(e)] \sigma$	=	$\mathcal{E}[e] \sigma$	$\mathcal{R}[r_1 \&\& r_2] \sigma$	=	$\mathcal{R}[r_1] \sigma \wedge \mathcal{R}[r_2] \sigma$
$\mathcal{E}[-e] \sigma$	=	$-\mathcal{E}[e] \sigma$	$\mathcal{R}[r_1 \mid\mid r_2] \sigma$	=	$\mathcal{R}[r_1] \sigma \vee \mathcal{R}[r_2] \sigma$
$\mathcal{E}[\text{sqrt}(e)] \sigma$	=	$\sqrt{\mathcal{E}[e] \sigma}$	$\mathcal{R}[(r)] \sigma$	=	$\mathcal{R}[r] \sigma$

Figure B.2: Semantics for the PCSL grammar

in the grammar, but simply means that the left hand side of the production can be substituted with nothing, that is, it can be left blank.

Figure B.2 contains a natural semantics for the PCSL grammar.

B.2 A Complete Example Using PCSL/MOPED

In this section we illustrate the use of the MOPED module in Millipede. Figure B.3 shows how to activate MOPED within Millipede. The protocol specification is the same as used in Chapter 7.

A protocol specification file can be loaded when Millipede is in MOPED mode by using the command `load`. The `list` command displays the content of the specification currently loaded. Once the program executes, any violations of the protocol are reported. If the user wishes to know more about the message that violated the protocol, such information can be obtained through a query.

```

(0) |MILLIPEDE> moped
(0) |MILLIPEDE\MOPED> load "protocol.pcs"
(0) |MILLIPEDE\MOPED> list
1: master[0]0(MS) -> slave[0]i(MR)  :: forall i : (0<=i && i<=7);
2: slave[0]i(SS) -> slave[0]0(MR)  :: forall i : (0<=i && i<=7);
3: slave[0]i(S1) -> slave[0]i+1(R1) :: forall i : (0<=i && i<7);
4: slave[0]i(S2) -> slave[0]i-1(R2) :: forall i : (0<i && i<=7);
(0) |MILLIPEDE\MOPED> exit
(0) |MILLIPEDE> run
MOPED: Protocol violation:
  Quantifier error:
    (0<i && i<7)
  violated by
    i == 0
    [(0<0 && 0<7)]
  in
    file.... : slave.c
    group... : 0
    instance : 0
    line.... : 34
    TID..... : 242167
(5) |MILLIPEDE>
.
.

```

Figure B.3: A complete example using MOPED to check all messages against a protocol specification written in PCSL.

Appendix C

The MQL Grammar

C.1 The Millipede Query Language Grammar

Figure C.1 shows the BNF grammar for the Millipede Query Language.

All MQL definitions are stored in an internal function table in Millipede, and retrieved when needed. During execution of a query, a local environment is maintained, containing any intermediate relations created using **let**. This local environment, along with all the relations created during evaluation of a statement of application, is then removed. It is currently not possible to add any permanent relations to the runtime system.

Command	::=	Application Definition Query
Application	::=	Name "(" [Arguments] ")"
Arguments	::=	Value Value "," Parameters
Definition	::=	"define" Name "(" [Parameters] ")" "as" Query
Parameters	::=	Name Name "," Parameters
Query	::=	QueryElement "begin" [QueryList] QueryElement "end"
QueryList	::=	QueryElement QueryElement ";" QueryElement
QueryElement	::=	Query "print" "(" String { "," Value } ")" "display" Relation "using" String "let" Name "be" Relation
Op	::=	"==" "!=" "<" "<=" ">" ">="
AttributeList	::=	Name AttributeList "," Name
Relation	::=	Name "select from" Relation "where" Name "Op" ("#" Name Value) "project" Relation "over" AttributeList "rename" Name "in" Relation "to" Name "join" Relation "with" Relation "union of" Relation "and" Relation "difference between" Relation "and" Relation "sort" Relation "by" AttributeList

Figure C.1: The Millipede Query Language BNF grammar.

Appendix D

Millipede Screen Shots

In this chapter we show a number of the most common windows in Millipede.

Figure D.1 shows the start up screen of Millipede. This is the main window for interacting with the debugging system. Commands for controlling the execution are issued here. MQL scripts are loaded from here, and protocol specifications for the protocol assertion module can be controlled from this window as well. Figures D.2 and D.3 show an example of interactive debugging and the status monitor, respectively.

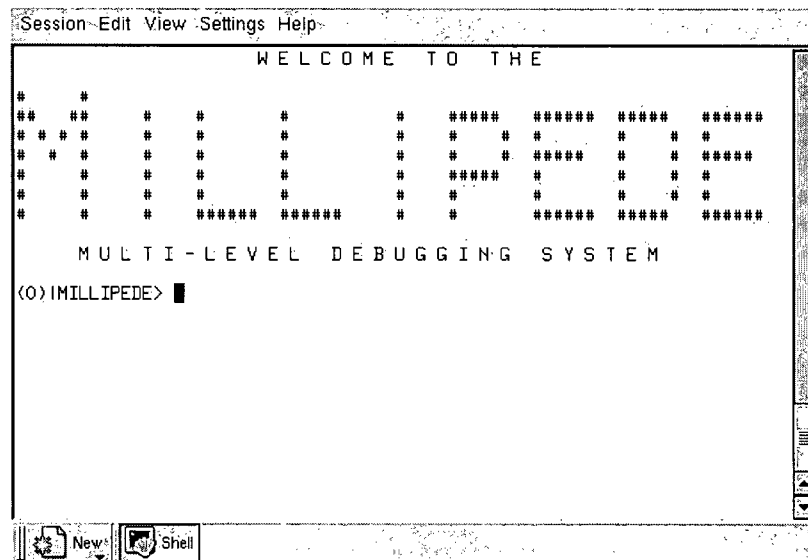
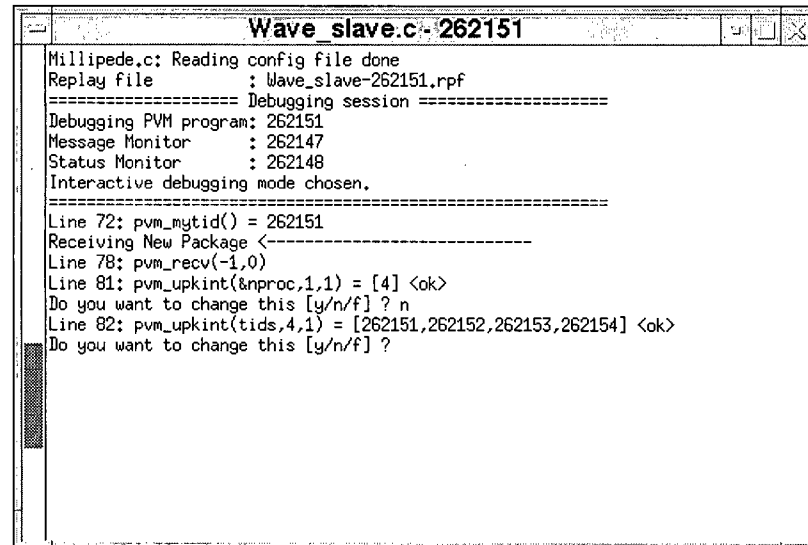


Figure D.1: The Millipede startup screen.

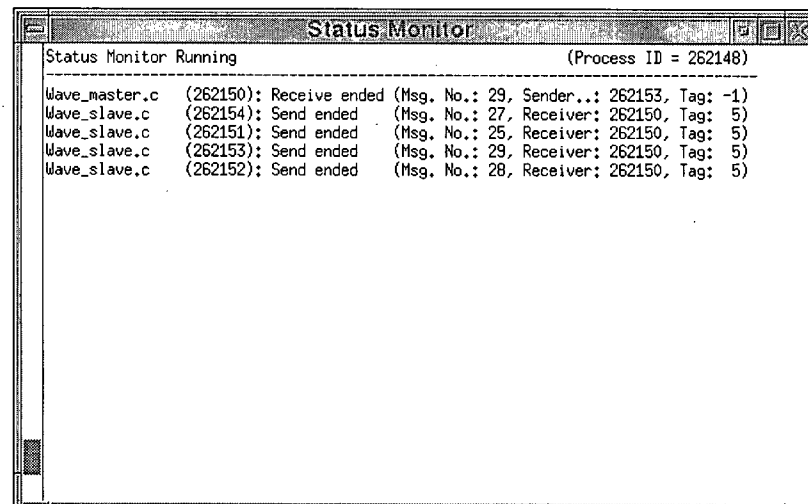


```

Wave_slave.c - 262151
Millipede.c: Reading config file done
Replay file      : Wave_slave-262151.rpf
===== Debugging session =====
Debugging PVM program: 262151
Message Monitor  : 262147
Status Monitor   : 262148
Interactive debugging mode chosen.
=====
Line 72: pvm_mytid() = 262151
Receiving New Package <-----
Line 78: pvm_recv(-1,0)
Line 81: pvm_upkint(&nproc,1,1) = [4] <ok>
Do you want to change this [y/n/f] ? n
Line 82: pvm_upkint(tids,4,1) = [262151,262152,262153,262154] <ok>
Do you want to change this [y/n/f] ?

```

Figure D.2: Screen shot illustrating interactive message debugging.



```

Status Monitor
Status Monitor Running (Process ID = 262148)
Wave_master.c (262150): Receive ended (Msg. No.: 29, Sender...: 262153, Tag: -1)
Wave_slave.c (262154): Send ended (Msg. No.: 27, Receiver: 262150, Tag: 5)
Wave_slave.c (262151): Send ended (Msg. No.: 25, Receiver: 262150, Tag: 5)
Wave_slave.c (262153): Send ended (Msg. No.: 29, Receiver: 262150, Tag: 5)
Wave_slave.c (262152): Send ended (Msg. No.: 28, Receiver: 262150, Tag: 5)

```

Figure D.3: Screen shot showing the status monitor. This window shows what each process in the system is doing with respect to communication.

Appendix E

Theoretical Framework for The Buffer Allocation Problems

In this section we define the three buffer allocation problems formally, present the theoretical graph frame work, and the proofs.

E.1 Definitions

Let S be a multiprocess system with n processes and E_i communication events occurring in process i ; a communication event is either a send or a receive. A multiprocess system S is **unsafe** if a deadlock can occur due to an insufficient number of available buffers; if S is not unsafe, then S is said to be **safe**. Figure E.1 is an example of an unsafe system. The numbers above the graph in Figure E.1 represent the buffer assignment.

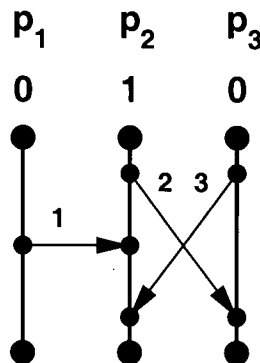


Figure E.1: Order of execution can cause deadlock.

A per-process **buffer assignment** is an n -tuple $B = (b_1, b_2, \dots, b_n)$ of nonnegative integers representing the number of buffers that can be allocated by each process. Similarly, a per-

channel buffer assignment is a q -tuple $B = (b_1, b_2, \dots, b_q)$, $q = \binom{n}{2}$, representing the number of buffers each channel in the system can allocate. Since buffers take up memory, which may be needed by the application, ideally, as few buffers as possible should be allocated. However, allocating too few buffers results in an unsafe system.

Buffer utilization is the nondeterministic phenomena of interest in the system. Making the choice of when to use a buffer affects future choices. For example, in Figure E.1, using a buffer for communication 1 before communication 3 completes results in deadlock.

Two natural decision problems arise from this optimization problem. Given a system S and a nonnegative integer k , the **Buffer Allocation Problem** (BAP) is to decide if there exists a buffer assignment B such that S is safe and $\sum b_i \leq k$. In order to solve this problem we need to solve a simpler one. Suppose we are given a buffer assignment B and a system S ; the **Buffer Sufficiency Problem** (BSP) is then to decide whether the assignment is sufficient to make S safe.

Additionally, we can require that no process in system S should ever block on a send. Given a system S and a nonnegative integer k , the **Nonblocking Buffer Allocation Problem** (NBAP) is to decide whether there exists a buffer assignment B , such that no send in S ever blocks, and $\sum b_i \leq k$.

We model systems by using communication graphs, and executions of systems by colouring games on these graphs. Communication graphs can be derived from execution traces of a program. The following subsection defines the graph based framework used throughout this section.

E.1.1 The Graph Based Framework

A **communication graph** of S is a directed acyclic graph $G = G(S) = (V, A)$ where the set of vertices $V = \{v_{i,c} \mid 1 \leq i \leq n, 0 \leq c \leq (E_i + 1)\}$ corresponds to the communication events and the arc set A consists of two disjoint arc sets: the computation arc set P and the communication arc set C . Each vertex represents an event in the system: vertex $v_{i,0}$ represents the start of process i , vertex $v_{i,c}$, $1 \leq c \leq E_i$, represents either a send or a receive event, and vertex $v_{i,(E_i+1)}$ represents the end of a process. An arc, $(v_{i,c}, v_{i,c+1}) \in P$, $0 \leq c \leq E_i$, represents a computation within process i and an arc $(v_{i,s}, v_{j,t}) \in C$ represents a communication between different processes, i and j , where $v_{i,s}$ is a send vertex, and $v_{j,t}$ is a receive vertex (e.g. Figure E.2). Note, the process arcs are drawn without orientation for clarity; they are always oriented downward. Communication graphs are comparable to the time-space diagrams—without internal events—noted in [Lam78].

The i th **process component** G_i of G is the subgraph $G_i = (V_i, A_i)$ where $V_i = \{v_{i,c} \in V \mid 0 \leq c \leq (E_i + 1)\}$ and $A_i = \{(v_{i,c}, v_{i,c+1}) \in A \mid 0 \leq c \leq E_i\}$. The process component corresponds to a process in S . We construct communication graphs by connecting process components with arcs. Hence, it is more intuitive to treat a process component as a chain of send and receive vertices bound by a start and an end vertex. A channel is represented by a **channel pair** (G_i, G_j)

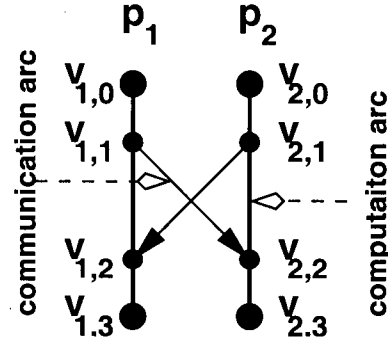
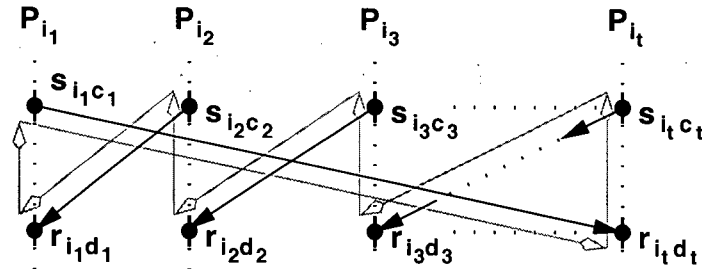


Figure E.2: An example of a communication graph with a 2-ring.

of process components.

A **t-ring** is a subgraph of a communication graph $G(S)$, consisting of $t > 1$ process components, such that in each of the t process components there is a send vertex s_{i_j, c_j} and a receive vertex r_{i_j, d_j} , $c_j < d_j$, $1 \leq j \leq t$ such that the arcs $(s_{i_1, c_1}, r_{i_t, d_t})$ and $(s_{i_{j+1}, c_{j+1}}, r_{i_j, d_j})$, $1 \leq j < t$ are in A . This definition is equivalent to the definition of a *crown* in [CMT96].

A t-ring represents a circular dependence of alternating send and receive events; see the example in Figure E.3. The shaded arcs in Figure E.3 show how each receive event depends on the preceding send event and each send event depends on the corresponding receive event. Thus, without an available buffer, there is a circular dependency that results in the system deadlocking.

Figure E.3: Dependency cycle in $G(S)$.

To model the execution of a system S , we define a colouring game that simulates the execution of the system with respect to $G(S)$.

E.1.2 Colouring the Communication Graph

Given a communication graph $G(S)$, an execution of a corresponding system S is represented by a colouring game where the goal is to colour all vertices green; a green vertex corresponds to the completion of an event. We use three colours to denote the state of each event in the system: a red vertex indicates that the corresponding event has not yet started, a yellow vertex indicates

that the corresponding event has started but not completed, and a green vertex indicates that the corresponding event has completed. Hence, a red vertex must first be coloured yellow before it can be coloured green; this corresponds to a traffic light changing from red, to yellow, to green.¹

We use tokens to represent buffer allocations. A buffer assignment of a process (or channel) is represented by a pool of tokens associated with the corresponding process component (respectively, the channel component). An instance of buffer utilization is represented by removing a token from a token pool and placing it on the corresponding communication arc.

The colouring game represents an execution via the following rules. Initially, the start vertices of G are coloured green and all remaining vertices are coloured red; this is called the **initial colouring**.

- | | |
|----------------------------------|---|
| $send \rightarrow yel$ | A red send vertex may be coloured yellow if the preceding vertex is green—the send is ready. |
| $recv \rightarrow yel$ | A red receive vertex may be coloured yellow if the corresponding send vertex is yellow, and the preceding vertex (in the same process component) is green—both the send and the receive are ready. |
| $recv \xrightarrow{\bullet} yel$ | A red receive vertex may be coloured yellow if the corresponding send vertex is yellow, and a token from the corresponding token pool is placed on the incident communication arc—the send is ready and a buffer is used. |
| $send \rightarrow grn$ | A yellow send vertex may be coloured green if the corresponding receive vertex is coloured yellow—the communication has completed from the sender's perspective. |
| $recv \rightarrow grn$ | A yellow receive vertex may be coloured green if both of its preceding vertices are green. If the incident communication arc has a token, the token is returned to its token pool—a receive completes after the send completes. |
| $end \rightarrow yel$ | A red end vertex may be coloured yellow if the preceding vertex is green. |
| $end \rightarrow grn$ | A yellow end vertex may be coloured green. |

Buffer utilization is represented by placing a token from the token pool on the selected arc, and colouring the corresponding receive vertex yellow. If no tokens are available, the rule cannot be invoked.

A **colouring** of G , denoted by χ , is a colour assignment to all vertices, which can be obtained by repeatedly applying the colouring rules, starting from the initial colouring. A **colouring sequence** $\Sigma = (\chi_1, \chi_2, \dots)$ is a sequence of colourings such that each colouring is derived from the preceding one by a single application of one of the colouring rules. An execution of a multiprocess system

¹Naturally, we refer to a European traffic light.

S with buffer assignment B is represented by a colouring sequence on $G(S)$. Each transition, from one colouring to the next, within a colouring sequence, corresponds to a state change of an event in the corresponding execution. Assuming that all events in the system are ordered, there is a correspondence between the colouring sequences on $G(S)$ and the executions of system S . Using the correspondence between colouring sequences on $G(S)$ and executions of system S , we reason about system S by reasoning about colouring sequences on $G(S)$.

We say that a colouring sequence **completes** if and only if the last colouring in the sequence comprises only green vertices. A colouring sequence **deadlocks** if and only if the last colouring in the sequence has one or more nongreen vertices and the sequence cannot be extended via the application of the colouring rules. A system S is safe if and only if every colouring sequence on the graph $G(S)$ completes.

We say that a colouring sequence **blocks** if there exists a sequence on $G(S)$, ending with a colouring containing a yellow send vertex, that cannot be extended by applying rule $recv \rightarrow yel$ to the corresponding receive vertex. A colouring sequence is **block free** if every prefix of the sequence does not block; a communication graph G , is block free if all colouring sequences on it are also block free. If $G(S)$ is block free, then no send in S will ever block during an execution.

A **token assignment**, also denoted by B , is a list of nonnegative integers, denoting the number of tokens assigned to each token pool; the token assignment is the abstract representation of a buffer assignment. The number of tokens required depends on the number of times that rule $recv \rightarrow yel$ may be invoked. If a token pool is empty, this means all buffers are in use.

E.2 Useful Lemmas

The following lemmas are used throughout our proofs. Lemma E.2.1 characterizes the conditions under which a colouring sequence will deadlock. Lemma E.2.2 characterizes conditions under which a single colouring sequence may represent all possible colouring sequences. Finally, Lemma E.2.3 characterizes a class of communication graphs on which no colouring sequence will deadlock.

Lemma E.2.1 (The t-Ring Lemma) *Let G be a communication graph comprising a single t-ring. Any colouring sequence on G completes if and only if rule $recv \rightarrow yel$ is invoked at least once.*

Proof: Assume by contradiction that there exists a complete colouring sequence Σ that does not make use of rule $recv \rightarrow yel$. Consider the first colouring in Σ where one of the send vertices is green; call the vertex s_i . Let r_j be the corresponding receive vertex. According to rule $send \rightarrow grn$, the vertex r_j must be yellow. Since rule $recv \rightarrow yel$ has not been applied, rule $recv \rightarrow yel$ must have been invoked earlier in the sequence. By the definition of a t-ring, the send vertex s_j must be the predecessor of r_j . Since the rule $recv \rightarrow yel$ was applied to r_j , s_j must be green. Hence, there is an earlier colouring in Σ with a green send vertex. This is a contradiction.

In the other direction, if rule $recv \rightarrow yel$ is invoked on receive vertex r_j , then rule $send \rightarrow grn$ can be invoked on the corresponding send vertex s_j , breaking the circular dependency. ■

Define the dependency graph of communication graph $G = (V, A)$ to be $H = (V, E)$ where all process arcs in A are reversed in E and all communication arcs in A are bidirectional in E . Define the depth $d(v)$ of a vertex $v \in V$ to be the length of the maximum length path in H from v to a start vertex.

Lemma E.2.2 *Let G be communication graph with a token assignment of 0. For any vertex v in G , if there exists a colouring sequence that colours vertex v green, there does not exist a colouring sequence that deadlocks before colouring v green.*

Proof: Proof by contradiction. Assume that there exist two colouring sequences, such that one colouring sequence colours a vertex green and the other deadlocks and does not colour the vertex green. Let $v \in V$ be such a vertex of minimum depth; that is, all vertices of lesser depth will be coloured green eventually by any colouring sequence on G . In order for a vertex to be coloured green, its component predecessor must be green. Since the depth of the predecessor is less than the depth of v , it can always be coloured green. Furthermore, since a send and its corresponding receive vertex are adjacent to each other, their depths differ by at most 1.

Since v must be a communication vertex, by rules $send \rightarrow grn$ and $recv \rightarrow grn$, the adjacent communication vertex t must be coloured yellow before v can be coloured green. If vertex t is of a lesser depth than v , then t must be green colourable in all colouring sequences; hence, v must also be green colourable. If t is at the same depth as v , then its component predecessor is at a lesser depth and must be green colourable, hence t is yellow colourable, and v is green colourable. If t is at a greater depth than v , the component predecessor of t , say u , is at the same or a lesser depth than t . If the latter, then u is green colourable and t is yellow colourable, otherwise, we apply the same argument to u first. Since there is no path from u to v in H —because $d(u) \leq d(v)$ —we need only recurse a finite number of times. ■

Lemma E.2.3 *If G is a communication graph whose dependency graph is acyclic, then no colouring sequence on G will deadlock.*

Proof: Proof by contradiction. Assume that a colouring sequence deadlocks on G . Let v be the vertex of minimum depth that cannot be coloured green. If v is a send (receive) vertex, let u be the corresponding receive (send) vertex. Let vertex t be the component predecessor of vertex u and let vertex w be the component predecessor of vertex v . Since the dependency graph is acyclic, the depths of both t and w are less than the depths of u and v . Hence, both t and w may be coloured green based on our minimality assumption. However, then both u and v may be coloured green; this is a contradiction! If v is an end vertex, then it has only one predecessor, which is of a lesser depth, which leads to the same contradiction. ■

E.3 Buffer Allocation in Systems with Receive Side Buffers

In systems with receive side buffers, messages are buffered only by the receiver. Buffers are allocated by the receiving process when a message arrives, but cannot be received, and are freed when the message is received by the application. Analogously, when colouring a receive vertex of the corresponding communication graph, only a token belonging to the same process component may be used. We call this the **receive side allocation scheme**.

E.3.1 The Buffer Allocation Problem

In order to prevent deadlock in distributed applications, the underlying system needs to allocate a sufficient number of buffers. Ideally, it should be the minimum number required. Unfortunately, determining the number of buffers required to make the system safe is intractable.

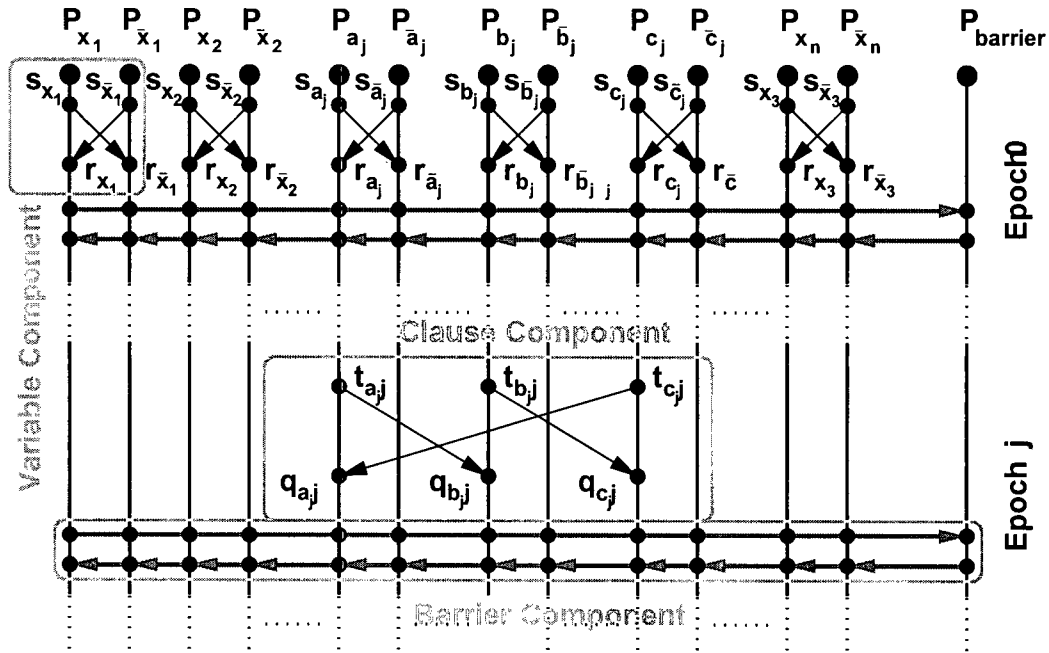
The corresponding graph-based decision problem is this: given a communication graph G and a positive integer k , determine if there is a token assignment of size k such that no colouring sequence deadlocks on G . We show that BAP_r is NP-hard by a reduction of the well known 3SAT problem [Coo71] to BAP_r . Recall the definition of 3SAT: determine if there exists a satisfying assignment to $\bigwedge_{i=1}^n (a_i \vee b_i \vee c_i)$, where a_i , b_i , and c_i are Boolean literals in $\{x_1, \bar{x}_1, x_2, \bar{x}_2, \dots, x_n, \bar{x}_n\}$.

Theorem E.3.1 *The Buffer Allocation Problem (BAP_r) is NP-hard.*

Proof: Proof by reduction of 3SAT to BAP_r . For any 3SAT instance F we construct a corresponding communication graph G such that for a token assignment of size n , any colouring sequence completes on G if and only if the corresponding variable assignment satisfies F .

Let F be an instance of 3SAT with n variables and c clauses; the variables are denoted x_1, x_2, \dots, x_n , and the j th clause is denoted $(a_j \vee b_j \vee c_j)$, where $a_j, b_j, c_j \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$. The corresponding communication graph G comprises $2n + 1$ process components: $2n$ of the components—called **literal** components—are labeled P_{x_i} and $P_{\bar{x}_i}$, $i = 1 \dots n$, and correspond to the literals of F . The last component—called the **barrier** component—is labeled P_{barrier} .

Each process component is divided into $c + 1$ epochs, where each epoch is a consecutive sequence of zero or more vertices within the component. All epochs are synchronized, that is, the vertices of one epoch must be coloured green before any of the vertices in the next epoch may be coloured. To ensure this we use a barrier component; the j th epoch of the barrier component, $j = 0, \dots, c$, comprises $2n$ receive vertices, labeled $q_{l,j}$, and $2n$ send vertices, labeled $t_{l,j}$, $l \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$. At the end of each epoch there is an arc from each of the literal components P_l , $l \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$, to the barrier component. Each arc emanates from vertex $s_{l,j}$, called a barrier send vertex, and is incident on vertex $q_{l,j}$, where $l \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$ and $j = 0 \dots c$. These arcs are followed by arcs emanating from the barrier component to the literal components; the arcs emanate from vertices $t_{l,j}$ and are incident on vertices $r_{l,j}$, called barrier

Figure E.4: Construction of G .

receive vertices. The barrier component has no cyclic dependencies. Hence, by Lemma E.2.3, no colouring sequence will deadlock on a barrier component.

Epoch 0 fixes a token assignment corresponding to a variable assignment in 3SAT. Each pair of process components, P_{x_i} and $P_{\bar{x}_i}$, $i = 1 \dots n$, forms a variable component, which corresponds to a variable. The two process components of a pair share a 2-ring; see Figure E.4. By Lemma E.2.1, at least one token must be assigned to either process component P_{x_i} or $P_{\bar{x}_i}$ to prevent all colouring sequences from deadlocking on G . Since only n tokens are available, each component pair can be assigned exactly one token. Finally, assigning the token to process component, P_{x_i} or $P_{\bar{x}_i}$, corresponds to fixing variable x_i to true or false. The epoch terminates with a barrier send vertex $s_{l_i,0}$, followed by a barrier receive vertex $r_{l_i,0}$, $l_i \in \{x_i, \bar{x}_i\}$.

Epoch j of each process component corresponds to the j th clause of F . The epoch of a process component P_l , $l \neq a_j, b_j, c_j$ —not labeled by a literal of the j th clause—contains only two vertices: the barrier send vertex $s_{l,j}$ and the barrier receive vertex $r_{l,j}$. The three process components, $P_{a_j}, P_{b_j}, P_{c_j}$, whose labels correspond to the literals in the j th clause share a 3-ring in the j th epoch; see Figure E.4. By Lemma E.2.1, to avoid deadlock, at least one of the three process components must have a token. If none of the components are assigned a token, all literals in the j th clause are false. The epoch is terminated by the barrier send and the barrier receive vertices.

A satisfying assignment on F satisfies at least one literal in every clause. A corresponding token

assignment assigns a token to the corresponding process component in each 3-ring—corresponding to the true literal. Hence, by Lemma E.2.1 none of the colouring sequences will deadlock on any of the clause component and any colouring sequence on G will complete.

For a falsifying assignment of F , there exists at least one clause comprising false literals. The corresponding token assignment fails to assign any tokens to the process components that share the corresponding 3-ring. Thus, by Lemma E.2.1 all colouring sequences will deadlock in that clause component.

Hence, for a token assignment of size n , any colouring sequence on G will complete if and only if the corresponding assignment satisfies F . Since finding a token assignment of size n such that no colouring sequence on G deadlocks is as hard as finding a satisfying assignment for F , BAP_r is NP-hard. ■

E.3.2 The Buffer Sufficiency Problem

A potentially simpler problem involves verifying whether a given buffer assignment is sufficient to prevent deadlock. Formally, given a graph G and a token assignment on G , determine if none of the colouring sequences on G deadlock. This problem turns out to be intractable as well.

We show that BSP_r is coNP-complete by a reduction from the TAUTOLOGY problem [GJ79, Page 261] to BSP_r . Given an instance of a formula in disjunctive normal form (DNF), $\bigvee_{i=1}^t \bigwedge_{j=1}^{l_i} a_{i,j}$ where $a_{i,j} \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$, the formula is a tautology if it is satisfied by all assignments. An assignment that falsifies F is a concise proof that the formula is not a tautology. We shall restrict our attention to 3DNF formulas, where each term has three literals: $\bigvee_{i=1}^t (a_i \wedge b_i \wedge c_i)$.

Theorem E.3.2 *The Buffer Sufficiency Problem (BSP_r) is coNP-complete.*

Proof: Let G be a communication graph along with a token assignment. If there exists a deadlocking colouring sequence on G , then the sequence itself is a certificate. The sequence is at most twice the number of vertices in G . Hence, BSP_r is in coNP.

Let F be a 3DNF formula with t terms where each term has three literals. For any 3DNF formula F , we construct a communication graph G and fix a token assignment such that there is a colouring sequence on G that deadlocks if and only if the corresponding assignment falsifies F . The construction consists of four types of components that correspond to fixing an assignment, a term in the disjunction, the disjunction, and the interconnects between components.

Each variable in F is represented by a variable component comprising three process components that are labeled P_{x_i} , $P_{\bar{x}_i}$, and Q_i . The latter, called the **arbitrator** component, comprises three receive vertices, labeled q_i , r_{x_i} , and $r_{\bar{x}_i}$. The former two process components, called **variable** components, comprise two send vertices each. The first, labeled s_{x_i} ($s_{\bar{x}_i}$), is adjacent to the corresponding receive vertex r_{x_i} ($r_{\bar{x}_i}$) in the arbitrator component. The second, labeled t_{x_i} ($t_{\bar{x}_i}$), is adjacent to receive vertices in components called dispersers, described later. The

vertex q_i in the arbitrator component is similarly adjacent to a vertex in a disperser component. The corresponding token assignment for each variable component assigns one token to Q_i and no tokens to the other two components; see Figure E.5. The component has the following property:

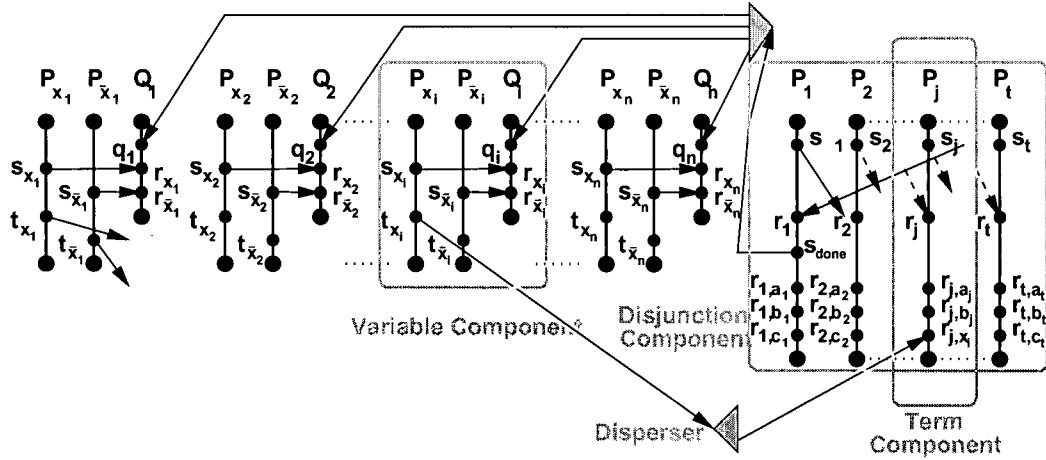


Figure E.5: The construction of the components.

Property E.3.3 Let G be a communication graph that contains a variable component. Any colouring sequence on G may colour exactly one of the two vertices t_{x_i} or $t_{\bar{x}_i}$ yellow before vertex q_i is coloured green.

Proof: By rule $\text{send} \rightarrow \text{yel}$, in order for t_{x_i} ($t_{\bar{x}_i}$) to be coloured yellow, vertex s_{x_i} ($s_{\bar{x}_i}$) must be coloured green. Hence, by rule $\text{send} \rightarrow \text{grn}$, vertex r_{x_i} ($r_{\bar{x}_i}$) must first be coloured yellow. Since vertex q_i is red, vertex r_{x_i} ($r_{\bar{x}_i}$) can only be coloured yellow via rule $\text{recv} \rightarrow \text{yel}$. However, there is only one token assigned to process component Q_i , hence rule $\text{recv} \rightarrow \text{yel}$ may only be invoked once. ■

The j th term in the disjunction is represented by a term component comprising a process component, which is called the **term component** and labeled P_j . The first part of each term component consists of a send vertex s_j and a receive vertex r_j ; these vertices are part of a t -ring. In the first term component, P_1 , there is an additional send vertex labeled s_{done} ; these are described in the next paragraph. The second part of each term component consists of three receive vertices labeled r_{j,a_j} , r_{j,b_j} , and r_{j,c_j} , where $a_j, b_j, c_j \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$ correspond to the literals in the j th term; see Figure E.5. These receive vertices are adjacent to send vertices in components called **dispersers**, which are described later. The term components are used to construct a disjunction component.

The disjunction component comprises t term components, where the first two vertices, s_j and r_j , are part of a t -ring spanning all t components. Specifically, each send vertex s_j , $j < t$, is

adjacent to receive vertex r_{j+1} and vertex s_t is adjacent to receive vertex r_1 ; see Figure E.5. Each term component is assigned one token. The disjunction component has the following property.

Property E.3.4 *Let G be a communication graph that contains a disjunction component. Any colouring sequence on G can colour r_j , $j \in [1, t]$, green if and only if at least one of r_k , $k \in [1, t]$, is coloured yellow before any r_{k,a_k} , r_{k,b_k} , or r_{k,c_k} are coloured yellow.*

Proof: By Lemma E.2.1, vertex r_j can be coloured green, if and only if rule $recv \rightarrow yel$ is invoked, colouring one of the receive vertices r_k , $k \in [1, t]$, yellow. The rule may only be invoked if and only if a token is available. Since each term component only has one token assigned and since vertex r_k precedes vertices r_{k,a_k} , r_{k,b_k} , and r_{k,c_k} , a token is available if and only if none of the vertices r_{k,a_k} , r_{k,b_k} , and r_{k,c_k} , are coloured yellow via rule $recv \rightarrow yel$, before vertex r_k is coloured yellow. ■

Once vertex r_k , $k \in [1, t]$, is coloured yellow, all r_j , $j = 1 \dots t$ may be coloured green, and vertex s_{done} may be coloured yellow. We now describe how the components are connected together using disperse components. Let s be a send vertex and R be a set of receive vertices. An (s, R) -dispenser comprises $|R| + 1$ process components: one master component, labeled M_s , and $|R|$ slave components labeled S_r , $r \in R$. The master component comprises one receive vertex labeled r_s , followed by $|R|$ send vertices labeled s_r , $r \in R$. Each send vertex is adjacent to the receive vertex on the corresponding slave component S_r . Each slave component has two vertices: a receive vertex q_r , followed by a send vertex t_r ; see Figure E.6. The latter vertex is adjacent to the receive vertex r in some other component. None of the components are assigned any tokens. The following property of a dispenser follows from Lemma E.2.3.

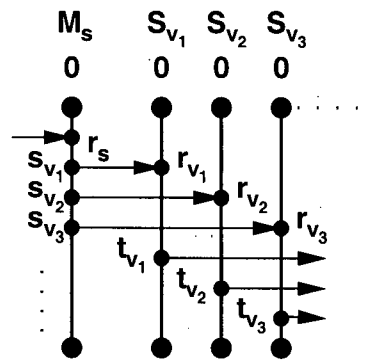


Figure E.6: The dispenser component.

Property E.3.5 *Let G be a communication graph containing an (s, R) -dispenser. If a colouring sequence colours vertex r_s yellow, then the colouring sequence can be extended to colour all vertices t_r , $r \in R$ yellow.*

Let R_{x_i} , $i = 1 \dots n$, be the set of receive vertices labeled $r_{j,x_i} \in P_j$, $j \in [1, t]$, and let $R_{\bar{x}_i}$ be similarly defined; recall that a_j, b_j, c_j are simply literal place holders in the vertex labels $r_{j,a_j}, r_{j,b_j}, r_{j,c_j}$. Hence, a (t_{x_i}, R_{x_i}) -dispenser connects send vertex $t_{x_i} \in P_{x_i}$ to vertices in R_{x_i} —belonging to the term components. Furthermore, let Q be the set of receive vertices q_i (in the variable components), $i = 1 \dots n$; a (s_{done}, Q) -dispenser connects vertex s_{done} to all variable components via receive vertices q_i . The construction of G comprises n variable components and one disjunction component, composed of t term components; these are connected together by a (s_{done}, Q) -dispenser, and $2n$ (t_a, R_a) -dispensers, where $a \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$. We claim that there exists a colouring sequence that deadlocks on G if and only if there is a falsifying assignment for formula F , that is, F is not a tautology.

Suppose that F has a falsifying assignment x , that is every term in the disjunction is false because each term has a literal x_i or \bar{x}_i , which is false. To construct a colouring sequence on G that deadlocks, we construct a set of vertices U . The first half of the colouring sequence is a maximal colouring sequence involving only the vertices of U . The second half of the sequence may involve all vertices in G . The resulting colouring sequence will always deadlock.

Let $X = \{a \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\} \mid a|_x = 0\}$, which is the set of literals that are false, and let $Z = \{s_a \in P_a \mid a \notin X\} \cup \{s_j \mid j = 1, \dots, t\}$, which contains the set of send vertices from the variable components that are labeled by a true literal and the numbered send vertices in the disjunction component; the set Z contains the vertices which may not initially be coloured. Let $U = V \setminus Z$ be the rest of the vertex set.

Consider a colouring sequence involving only vertices in U . By property E.3.3 any maximal colouring sequence will colour the vertices t_a yellow (in the variable component), where $a \in X$. Hence, by property E.3.5 the vertices t_r (in the dispensers) will be coloured yellow, where $r \in \bigcup_{a \in X} R_a$ —the send vertices t_r in the dispensers are adjacent to the receive vertices in R_a . Since x is a falsifying assignment, every term contains a literal, which is falsified by x . Without loss of generality, let a_j denote a literal that is false in the j th term; therefore, process component P_j contains a receive vertex r_{j,a_j} , which is adjacent to the yellow send vertex $t_{r_{j,a_j}}$ (in the dispenser). Since none of the vertices of the t -ring (in the disjunction component) are not in U —they are still coloured red—the token belonging to component P_j is used to apply rule $\text{recv} \rightarrow \text{yel}$ to colour vertex r_{j,a_j} yellow. Since every term has a false literal, the colouring sequence colours a receive vertex r_{j,a_j} , $j = 1 \dots t$ in every term component P_j . After the sequence cannot be extended, allow all vertices to be coloured; since vertices r_{j,a_j} (in the term components), $j = 1 \dots t$, have been coloured yellow before vertex r_j (in term component P_j), according to property E.3.4, the sequence will deadlock.

If a colouring sequence on G deadlocks, according to property E.3.4, deadlock occurs only if there is a yellow vertex labeled r_{k,a_k} , r_{k,b_k} , or r_{k,c_k} in each of the term components. Their

predecessors—vertices t_l , $l \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$, in the dispersers—must be green. Since the colouring sequence is maximal, by property E.3.3 exactly one of t_{x_i} or $t_{\bar{x}_i}$ is red, thus this corresponds to a valid assignment: setting $x_i = 0$ if t_{x_i} is green, or $x_i = 1$ if $t_{\bar{x}_i}$ is green yields an assignment that falsifies F .

Thus, a colouring sequence on G deadlocks if and only if the corresponding assignment falsifies F . Hence, BSP_r is coNP-complete. ■

Therefore, just determining whether a buffer assignment is sufficient is intractable, even one as simple as in the preceding example. Intuitively, the buffers of a process are assigned based on the behaviour of other processes; thus, buffer utilization is not locally decidable. Further, the order in which buffers are assigned is nondeterministic, exploding the search space of possible buffer utilizations. This phenomena, which our proofs rely on, is what we call *buffer stealing*. For example, in a system corresponding to the variable component (see Figure E.5), the first process to send its message gets the buffer, and the other process remains blocked until the arbitrator performs the receives. This stealing corresponds to fixing a value of a variable. Similarly, the system corresponding to the disjunction component allocates buffers for each of the term processes. However, if the buffer is stolen in all terms, corresponding to a falsifying assignment, then the system will deadlock within the ring.

For completeness, we note the following corollary:

Corollary E.3.6 *The Buffer Allocation Problem (BAP_r) is in $\Sigma_2\text{P}$.*

Proof: By Theorem E.3.2, verifying that a token assignment is sufficient to prevent deadlock (BSP_r) is coNP-complete. Since we can nondeterministically guess a sufficient token assignment, the result follows. ■

E.3.3 The Nonblocking Buffer Allocation Problem

In addition to the system being safe, we can require that no sending process ever blocks due to insufficient buffers on the receiving process. The Nonblocking Buffer Allocation Problem (NBAP_r) is to determine the minimum number of buffers needed to achieve nonblocking sends.

Formally, the corresponding decision problem is this: given a communication graph G and an integer k , determine if there exists a token assignment of size k such that no colouring sequence on G blocks. Recall that a colouring sequence does not block if, whenever a send vertex is coloured yellow, rule $\text{recv} \rightarrow \text{yel}$ may be applied to the corresponding receive vertex.

Let P_i and P_j , $j \neq i$, be two process components. Given two vertices, $v_{i,c}$ and $v_{i,t}$, in P_i , $t > c$, vertex $v_{i,t}$ is **communication dependent** on vertex $v_{i,c}$ if $v_{i,c}$ is the start vertex or if there exists a vertex $v_{j,d} \in P_j$, such that there is a path from $v_{i,c}$ to $v_{j,d}$ and the arc $(v_{j,d}, v_{i,t})$ is in A (see Figure E.7). Vertex $v_{i,t}$ is **terminally communication dependent** on vertex $v_{i,c}$ if $v_{i,t}$

is communication dependent on $v_{i,c}$ and is not communication dependent on the vertices $v_{i,l}$, $c < l < t$. The algorithm depicted in Figure E.8 computes an optimal token assignment such that no colouring sequence on G can block.

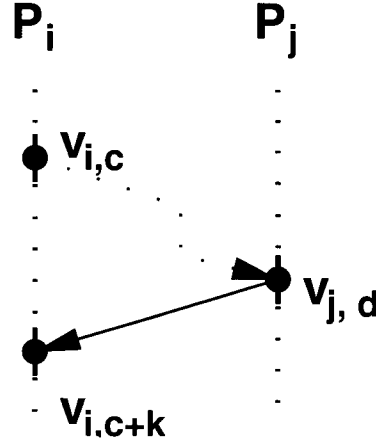


Figure E.7: $v_{i,c+k}$ is communication dependent on $v_{i,c}$.

1. For each receive vertex $v_{i,t}$ determine its terminal communication dependency, vertex $v_{i,c}$, where $t > c$.
2. Set $I_{i,t} = [c, t]$ to be the interval between vertex $v_{i,c}$ and vertex $v_{i,t}$.
3. For each process component G_i , compute b_i , the maximum overlap over all intervals $I_{i,t}$.
4. $B = \{b_1, b_2, \dots, b_n\}$ is the optimal nonblocking token assignment.

Figure E.8: Algorithm for computing an optimal nonblocking buffer assignment.

Remark E.3.7 In a system corresponding to communication graph G , the time between a message arriving at process i and its receipt corresponds to the interval $I_{i,t}$. Each interval must have a buffer to ensure nonblocking sends. Hence, the minimum number of buffers, b_i , is the maximum overlap over all intervals within process p_i .

Computing the terminal communication dependencies for G can be done via dynamic programming in $O(|V|n)$ time, where V is the vertex set of G and n is the number of process components. If there exists a path from vertex $v_{i,c}$ to $v_{j,d}$, then there exists a path from $v_{i,c}$ to all vertices $v_{j,d+k}$, $k > 0$. Associate with each vertex $v_{i,c}$ an integer vector $a_{i,c}$ of size n ; $a_{i,c}[j] = d$ means that there exists a path from $v_{i,c}$ to $v_{j,d}$, and thus to $v_{j,d+k}$, $k > 0$. The vector $a_{i,c}$ is computed by taking the element-wise minimums over the vectors of the adjacent vertices $v_{i,c}$; this is simply

a depth-first traversal of G . Since the number of arcs is bounded by $3|V|/2$ and the pairwise comparison takes n steps, the traversal takes $O(|V|n)$ time.

Next, computing the $O(|V|)$ intervals, $I_{i,t}$, requires one table lookup per interval. To compute the maximum overlap we sort the intervals and perform a sweep, keeping track of the current and maximum overlap; this takes $O(|V| \log |V|)$ time. Thus, the total complexity is $O(|V|n + |V| \log |V|)$ time. In the worst case, where $n \approx |V|$, this algorithm is quadratic. However, in practice n is usually fixed, in which case the $|V| \log |V|$ term dominates.

E.4 Proof of Correctness of the Nonblocking Buffer Allocation Algorithm

Lemma E.4.1 *Let G be a communication graph. For all vertices $v_{i,c}, v_{j,d} \in G$; if $v_{j,d}$ is a send vertex and there exists a path from the vertex $v_{i,c}$ to vertex $v_{j,d}$, then vertex $v_{j,d}$ cannot be coloured yellow until vertex $v_{i,c}$ is coloured green.*

Proof: By rule $\text{send} \rightarrow \text{yel}$, the predecessor of $v_{j,d}$ must first be coloured green before $v_{j,d}$ can be coloured yellow. Since rules $\text{send} \rightarrow \text{grn}$, and $\text{recv} \rightarrow \text{grn}$ imply that the predecessors of a green vertex must be green, the result follows. ■

Corollary E.4.2 *Let G , $v_{i,c}$, and $v_{j,d}$ be as in Lemma E.4.1 and let $v_{i,t}$ be the receive vertex corresponding to the send vertex $v_{j,d}$. Rule $\text{recv} \rightarrow \text{yel}$ will never be applied to vertex $v_{i,t}$ before vertex $v_{i,c}$ is coloured green.*

The preceding corollary implies that a token, which is needed to colour the receive vertex $v_{i,t}$ yellow, need not be available until the vertex on which $v_{i,t}$ is terminally communication dependent is coloured green. Hence, it is sufficient to ensure token availability just before colouring the respective send vertex green; this is also necessary.

Theorem E.4.3 *Given G , let $v_{i,c}$ be a send vertex and $v_{i,t}$ be a receive vertex that is terminally communication dependent on vertex $v_{i,c}$. A token for the application of rule $\text{recv} \rightarrow \text{yel}$ on arc $(v_{j,d}, v_{i,t})$ must be available as soon as vertex $v_{i,c}$ is coloured green.*

Proof: Let $v_{j,d}$ be the send vertex corresponding to the receive vertex $v_{i,t}$ and let $Q = \{v_{i,q} \mid c < q < t\}$ be the set of vertices that are predecessors of $v_{i,t}$, but on which $v_{i,t}$ is not communication dependent.

Since $v_{i,t}$ is not communication dependent on the vertices in Q , we can construct a colouring sequence on G that fixes the vertices in Q to be red, and colours vertex $v_{j,d}$ yellow, making the application of rule $\text{recv} \rightarrow \text{yel}$ possible in the next step. Since no progress is made in the i th process component after colouring vertex $v_{i,c}$ green, the state of the associated token pool does not change until the application of rule $\text{recv} \rightarrow \text{yel}$ to vertex $v_{i,t}$. Hence, when vertex $v_{i,c}$ is coloured green, the token pool must have a token destined for arc $(v_{j,d}, v_{i,t})$. ■

Thus, if a receive vertex r is terminally communication dependent on a send vertex s , then it is necessary and sufficient that a token, which is used to apply rule $recv \rightarrow yel$ to receive vertex r , must be available as soon as the send vertex s is coloured green; the start vertex may be thought of as a special send vertex. Since the interval corresponding to r begins when s is coloured green, and ends when r is coloured green, a token must be available for the $recv \rightarrow yel$ rule, which can occur during this interval. Computing the maximum overlap of intervals yields the required number of tokens.

Example Use of the NBAP_r Algorithm

To demonstrate the NBAP_r algorithm we have implemented it, and analyzed the pipe-and-roll parallel matrix multiplication algorithm [FJL⁺88]. The program has one control process and a number of worker processes arranged in a 2 dimensional mesh. We ran the NBAP_r algorithm on meshes of size 2×2 , 3×3 and 4×4 . The communication graph for the smallest example, comprising four workers ordered in a 2×2 mesh, is depicted in Figure E.9. The corresponding optimal buffer assignment is listed in the second column of Table 1.

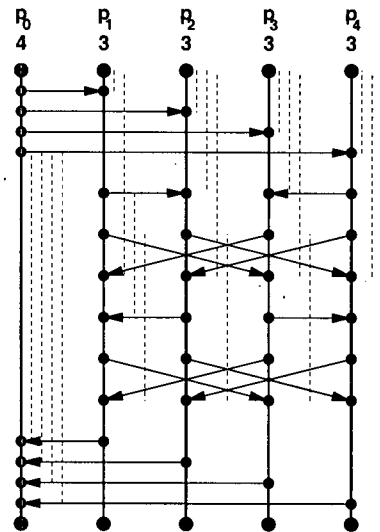


Figure E.9: The communication system for a 2×2 worker process mesh.

In this example, process 0 is the control process and processes 1 through 4 are the workers. The control process needs 4 buffers and the workers each need 3 to execute without blocking. The results obtained when executing the NBAP_r algorithm on a 3×3 worker system is 9 buffers for the control process and between 4 and 5 buffers for the worker processes. For the 4×4 system the numbers are 16 for the control process and between 5 and 7 buffers for the workers.

Proc.	Max overlap	Overlap for intervals I_j								
		I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
0	4	0	0	0	0	4	3	2	1	0
1	3	2	1	2	3	2	1	1	0	0
2	3	3	2	1	2	1	1	1	0	0
3	3	3	2	1	2	1	1	1	0	0
4	3	2	1	2	3	2	1	1	0	0

Table 1. The result of running the NBAP_r algorithm on the 2×2 worker example.

Approximating BAP_r with NBAP_r

The NBAP_r algorithm is useful for determining a token assignment that prevents deadlock, that is, approximating BAP_r. Since a nonblocking colouring sequence does not deadlock, a token assignment determined by the NBAP_r algorithm ensures that the graph is deadlock free. However, the token assignment may be far from optimal. A simple example of this phenomena is a two process component graph comprised of n arcs emanating from the first component and incident on the second. Such a graph requires zero tokens to avoid deadlock, but requires n tokens to be block free. Thus, the aforementioned token assignment may entail many more tokens than required.

E.5 Buffer Allocation in Systems with Send Side Buffers

In this section we consider the second of the four buffer placement strategies: send side buffers. Buffers are now allocated on the sending process side if the receive is not ready to accept the message. Correspondingly, the token pool used when applying rule $recv \rightarrow yel$ to the receive vertex of arc (s, r) belongs to the process component containing the send vertex s . We call this the **send side allocation scheme**.

The Buffer Allocation Problem (BAP_s) remains intractable. The problem is conjectured to be NP-complete (see the following paragraph). The NP-hardness follows from the observation that each t-ring in the construction in Theorem E.3.1 has to have a token assigned to a process component pair in order to prevent deadlock. It does not matter if the token is allocated from the token pool of the sending or the receiving process component. Hence, the reduction used in Theorem E.3.1 can be applied with no modification.

We conjecture that the corresponding Buffer Sufficiency Problem (BSP_s) is in P. This is because the relative order in which tokens from a particular token pool are utilized is invariant with respect to the colouring sequences. Hence, we believe that the determining sufficiency is similar to the nonblocking buffer allocation problem and hence is in P. If this is the case, BAP_s is NP-complete.

The Nonblocking Buffer Allocation Problem (NBAP_s) remains in P. The problem can be solved

by first reversing all arcs in the communication graph, swapping the start and end vertices, and then running the algorithm described in Figure E.8.

E.6 Buffer Allocation in Systems with Send and Receive Side Buffers

So far we have considered systems exclusively with send side or receive side buffers. In this section we investigate systems with buffers on both the send and the receive sides; many communication systems use per-host buffer pools for both receiving and sending messages. The choice of where to buffer the message—on the sender or on the receiver—increases the difficulty of determining the system's properties.

We assume a lazy mechanism for utilizing buffers: first use a buffer from the sender's pool. If none is available, use a buffer from the receiver's pool. If neither is available, attempt to free a send side buffer by transferring its contents to a buffer belonging to the corresponding receiver. Intuitively, the system attempts to maximize buffer use, without attempting to predict the future.

The corresponding colouring game allows tokens to be allocated from the pools belonging to both the sending component and the receiving component. Correspondingly, a lazy token utilization scheme is used: let (s_i, r_j) be a communication arc from process component P_i to process component P_j . The following rules apply during the application of rule $recv \rightarrow yel$ to vertex r_j :

1. If a token belonging to component P_i is available, use it.
2. Otherwise, if a token belonging to component P_j is available, use it.
3. Otherwise, if a token belonging to component P_i is currently placed on arc (t_i, r_k) , $t_i \in P_i$, $r_k \in P_k$, and a token belonging to component P_k is available. Then the token on arc (t_i, r_k) may be replaced with the one belonging to P_k , freeing a token to be used in the current application of rule $recv \rightarrow yel$.

We call this the **mixed allocation scheme**.

Not unexpectedly, the Buffer Allocation Problem (BAP_{SR}) remains intractable within the mixed allocation scheme. This is because the receive side allocation scheme, which provides no choice of token pools, can be simulated within the mixed allocation scheme. Concretely consider the receive side allocation scheme analyzed in Section E.3: to simulate the receive side allocation scheme on communication graph G , within the mixed allocation scheme, each arc in G is replaced by the component illustrated in Figure E.10. Since vertex q cannot be coloured green until vertex r is coloured yellow, and component P' has no tokens, applying rule $recv \rightarrow yel$ to r requires that P_j has an available token, regardless of whether P_i has an available token.

Similarly, the Buffer Sufficiency Problem (BSP_{SR}) within the mixed allocation scheme is also

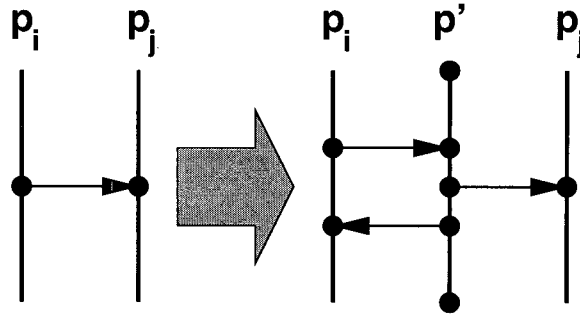


Figure E.10: Nullifying send side token pools.

coNP-complete. The hardness follows from Theorem E.3.2 and the preceding argument. Since a colouring sequence also serves as a deadlock certificate in this case, the coNP-completeness result follows.

The interesting property of the mixed allocation scheme is that the Nonblocking Buffer Allocation Problem (NBAP_{SR}) is intractable; the choice of token pools increases the search space of solutions exponentially! The reduction is from 3SAT.

Theorem E.6.1 *The Nonblocking Buffer Allocation Problem (NBAP_{SR}) is NP-hard.*

Proof: Let F be an instance of 3SAT, comprising n variables, labeled x_i , $i = 1 \dots n$, and c clauses. We construct a communication graph G such that there exists a token assignment of $n + 2$ tokens that prevents any colouring sequence from blocking on G if and only if the corresponding assignment satisfies F .

The graph G comprises $2n + 3$ process components: the first $2n$ are labeled P_{x_i} and $P_{\bar{x}_i}$, $i = 1 \dots n$, and the remaining three process components are labeled P , Q_0 and Q_1 , respectively. The graph is divided into $c + 1$ epochs: epoch 0 corresponds to the variable assignment, and epochs 1 through c correspond to clause evaluation.

In epoch 0 each process component P_{x_i} contains a single send vertex s_i that is adjacent to the receive vertex r_i located in epoch 0 of process component $P_{\bar{x}_i}$. Process component Q_0 (and Q_1) contains four vertices: two receive vertices $q_{0,1}$ and $q_{0,2}$ (respectively $q_{1,1}$ and $q_{1,2}$), followed by two send vertices $t_{0,1}$ and $t_{0,2}$ (respectively $t_{1,1}$ and $t_{1,2}$). Finally, process component P contains eight vertices: two send vertices, $s_{0,1}$ and $s_{0,2}$, that are adjacent to vertices $q_{0,1}$ and $q_{0,2}$; two receive vertices, $r_{0,1}$ and $r_{0,2}$, that are adjacent to $t_{0,1}$ and $t_{0,2}$; two more send vertices, $s_{1,1}$ and $s_{1,2}$, that are adjacent to $q_{1,1}$ and $q_{1,2}$; and two more receive vertices, $r_{1,1}$ and $r_{1,2}$, that are adjacent to $t_{1,1}$ and $t_{1,2}$. See Figure E.11. Epoch 0 has two important properties.

Property E.6.2 *Any token assignment must assign at least one token to either component P_{x_i} or $P_{\bar{x}_i}$ to prevent the colouring sequence from blocking after colouring vertex s_i yellow.*

Property E.6.3 A token assignment on G having only $n + 2$ tokens must assign two tokens to process component P to prevent a colouring sequence from blocking after yellow colouring one of the send vertices $s_{0,1}$, $s_{0,2}$, $s_{1,1}$ or $s_{1,2}$.

Proof: Since n tokens must be allocated to the process components P_{x_i} or $P_{\bar{x}_i}$, $i = 1, \dots, n$, this leaves only two tokens to be allocated. Since the colouring rule sequence $\text{send} \rightarrow \text{yel}$, $\text{recv} \rightarrow \text{yel}$, $\text{send} \rightarrow \text{grn}$, $\text{send} \rightarrow \text{yel}$, $\text{recv} \rightarrow \text{yel}$ can colour send vertices $s_{0,1}$ and $s_{0,2}$, or send vertices $s_{1,1}$ and $s_{1,2}$, component pairs (P, Q_0) and (P, Q_1) must each have two tokens between them. This can only happen by assigning the tokens to P . ■

A corollary of these properties is that once a legal token assignment is made, no colouring sequence will block in epoch 0. The choice of allocating the token on P_{x_i} versus $P_{\bar{x}_i}$ corresponds to fixing the variable assignment.

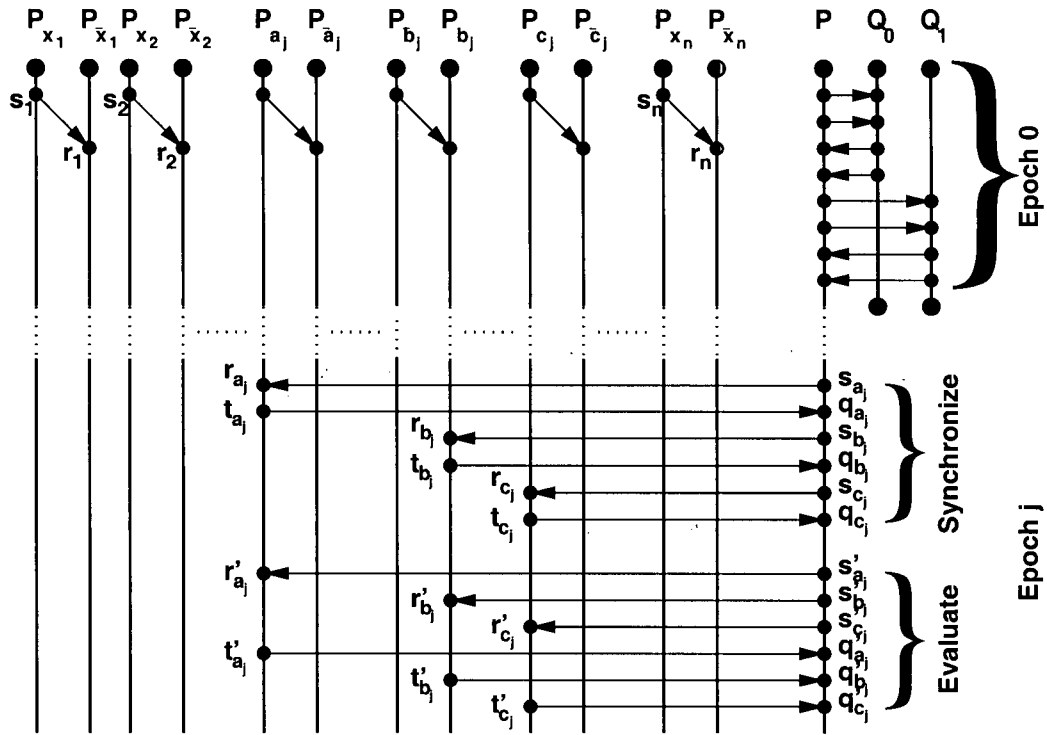


Figure E.11: Reduction from 3SAT to NBAP_{sr}.

For $j = 1 \dots c$, epoch j corresponds to the j th clause. Each epoch comprises two parts of six arcs each: the synchronization part and the evaluation part. Four process components are involved in an epoch: the three components, P_{a_j} , P_{b_j} , and P_{c_j} , whose labels are the literals in the j th clause, where $a_j, b_j, c_j \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$, and component P , which is involved in every epoch. Epoch j of component P_{a_j} comprises four vertices: receive vertex $r_{a_j,j}$, send vertex

$t_{a_j,j}$, receive vertex $r'_{a_j,j}$, and send vertex $t'_{a_j,j}$. Process components P_{b_j} and P_{c_j} are analogously formed.

In epoch j component P has 12 vertices, the first six are these: send vertex $s_{a_j,j}$, receive vertex $q_{a_j,j}$, send vertex $s_{b_j,j}$, receive vertex $q_{b_j,j}$, send vertex $s_{c_j,j}$, and receive vertex $q_{c_j,j}$. These are followed by three send vertices: $s'_{a_j,j}$, $s'_{b_j,j}$, and $s'_{c_j,j}$, and three receive vertices: $q'_{a_j,j}$, $q'_{b_j,j}$, and $q'_{c_j,j}$.

Each vertex $s_{l,j}$ is adjacent to vertex $r_{l,j}$, each vertex $t_{l,j}$ is adjacent to vertex $q_{l,j}$, each vertex $s'_{l,j}$ is adjacent to vertex $r'_{l,j}$, and each vertex $t'_{l,j}$ is adjacent to vertex $q'_{l,j}$; see Figure E.11. For conciseness we drop the last index, j , if it is obvious from the context. Epoch j has three important properties:

Property E.6.4 *If vertex q'_{c_j} (in epoch j) is coloured green and vertex $s_{a_{j+1}}$ (in epoch $j+1$) is still red, then no tokens that belong to component P are assigned to arcs. The same applies to vertex pairs (q_{a_j}, s_{b_j}) , (q_{b_j}, s_{c_j}) , and (q_{c_j}, s'_{a_j}) , also in epoch j .*

Proof: All ancestors of q'_{c_j} must be coloured green and all descendants of $s_{a_{j+1}}$ must be coloured red. This includes all vertices in G , except some vertices s_i and r_i in epoch 0, which are not adjacent to vertices in component P . Hence, the tokens belonging to P are not assigned to any arc. The same argument applies to the other vertex pairs. ■

Property E.6.5 *A colouring sequence on G can block only when yellow colouring receive vertices r'_{a_j} , r'_{b_j} , r'_{c_j} , q'_{a_j} , q'_{b_j} , or q'_{c_j} .*

Proof: As a corollary of properties E.6.2 and E.6.3, no colouring sequence can block in epoch 0. Thus, we need only check that no colouring sequence can block in the first part of epoch j , $j = 1 \dots c$.

By property E.6.4, if s_{a_j} is red and its predecessor is green, then no tokens of P are in use. Hence, to colour s_{a_j} green, a token is available to colour r_{a_j} yellow. Since vertex r_{a_j} is a predecessor of t_{a_j} , vertex r_{a_j} must be coloured green before t_{a_j} may be coloured yellow. Thus the token is freed before t_{a_j} is coloured green, and may be used to colour vertex q_{a_j} yellow after t_{a_j} is coloured yellow. A similar argument applies to the vertices r_{b_j} , q_{b_j} , r_{c_j} , and q_{c_j} . ■

Property E.6.6 *A colouring sequence can block in epoch j if and only if none of the three process components, P_{a_j} , P_{b_j} , and P_{c_j} , have a token assigned.*

Proof: For the 'if' direction consider a colouring sequence that colours vertex q_{c_j} green, but has not yet coloured vertex s'_{a_j} yellow. By definition, blocking does not occur, if rule $recv \rightarrow_{yel}$ may always be applied to colour a receive vertex yellow. To colour the send vertices s'_{a_j} , s'_{b_j} , and s'_{c_j} yellow and then green, the receive vertices r'_{a_j} , r'_{b_j} , and r'_{c_j} , must be coloured yellow via rule $recv \rightarrow_{yel}$. Since the receive vertices r'_{a_j} , r'_{b_j} , and r'_{c_j} are not ancestors of the send vertices s'_{a_j} , s'_{b_j} , and s'_{c_j} , none of the receive vertices need be coloured green before the send vertices are

coloured yellow. However, component P has only two tokens, and none of components P_{a_j} , P_{b_j} , P_{c_j} have any. Hence, rule $recv \rightarrow yel$ can only be invoked twice, instead of the requisite three times. Thus, a colouring sequence can block in epoch j .

For the ‘only if’ direction we claim that if a literal component P_{a_j} , P_{b_j} , or P_{c_j} has a token, rule $recv \rightarrow yel$ can be invoked on any of the six receive vertices r'_{a_j} , r'_{b_j} , r'_{c_j} , q'_{a_j} , q'_{b_j} , and q'_{c_j} . Since r'_{a_j} is a predecessor of t'_{a_j} , r'_{a_j} must be coloured green before t'_{a_j} , and hence before q'_{a_j} is coloured yellow. Thus, the same token that was allocated upon the application of rule $recv \rightarrow yel$ to vertex r'_{a_j} , may also be allocated upon the application of rule $recv \rightarrow yel$ to vertex q'_{a_j} ; the same argument is applicable to vertices q'_{b_j} and q'_{c_j} . Applying rule $recv \rightarrow yel$ to vertices r'_{a_j} and r'_{b_j} , uses the two tokens from component P . To colour vertex r'_{c_j} yellow there are three possible scenarios:

1. the colouring sequence has already freed one of the tokens, allowing it to be reused,
2. component P_{c_j} has a token, in which case it is used, or
3. component P_{a_j} (or P_{b_j}) has a token, in which case it replaces the token used to yellow colour vertex r'_{a_j} (or r'_{b_j}) and the freed token is used to colour vertex r'_{c_j} .

Since at least one component P_{a_j} , P_{b_j} , or P_{c_j} have a token, the claim is proven. ■

By property E.6.6 a colour sequence will block in epoch j if and only if none of the process components P_{a_j} , P_{b_j} , or P_{c_j} has a token, which corresponds to the j th clause having no literals that are true. Thus, a token assignment of size $2n + 2$ prevents any colouring sequence on G from blocking if and only if the corresponding assignment satisfies F . ■

E.7 Buffer Allocation in Channel Based Systems

In channel based systems processes communicate via pairwise connections that are created at startup. Each connection, called a channel, is specified by its endpoints and is used by one process to send messages to the other. Each channel functions independently of other channels in the system, and resources such as buffers are allocated on a per channel basis, rather than per process. Finally, channels behave like queues, that is, messages are removed from the channel in the same order that they are inserted.

Channels may either be unidirectional, comprising source and destination endpoints, or bidirectional, comprising two symmetric endpoints. In the former case, only the source process may insert messages into the channel and only the destination process may remove messages from the channels. A bidirectional channel is equivalent to two unidirectional channels, allowing both processes to insert and remove messages from the channel. Here we only consider unidirectional channels.

Except for buffer allocation, channel based communication does not differ from the previously described send/receive mechanism. In fact, an unbuffered channel communication is just a synchronous send/receive communication. Thus, we can derive similar results for channel based systems.

In the corresponding colouring game, tokens are allocated to channels (component pairs) instead of to components. This change does not change the properties used in our proofs. In fact, Lemma E.2.1 may be used unchanged. We call this the **per channel allocation scheme**.

E.7.1 The Buffer Allocation Problem

The corresponding Buffer Allocation Problem (BAP_{sr}) is this: given a communication graph G and an integer k , determine whether there exist a token assignment of size k , such that no colouring sequence deadlocks on G . Even though token utilization, during the colouring of a communication graph, is only dictated by the communication arcs within a process component pair, determining the number of tokens needed remains NP-hard. The proof is similar in spirit to Theorem E.3.1.

Theorem E.7.1 *The Buffer Allocation Problem (BAP_{sr}) is NP-hard.*

Proof: We prove this by reducing 3SAT to BAP_{sr} . For any 3SAT instance F we construct a corresponding communication graph G —polynomial in size of F —such that for a token assignment of size n , any colouring sequence will complete on G if and only if the corresponding variable assignment satisfies F .

Let F be an instance of 3SAT on n variables and comprising c clauses. The construction is nearly identical to that in Theorem E.3.1, except for the components representing the clauses of F . The graph G has $2n$ process components that are labeled by the literals of F , P_{x_i} and $P_{\bar{x}_i}$, $i = 1 \dots n$. Each component comprises $c + 1$ epochs, where each epoch contains zero or two vertices.

As in Theorem E.3.1, epoch 0 fixes a variable assignment. In epoch 0 each component has two vertices: a send vertex, labeled s_{x_i} (or $s_{\bar{x}_i}$), and a receive vertex r_{x_i} , (respectively $r_{\bar{x}_i}$), $i = 1 \dots n$. Vertex s_{x_i} is adjacent to vertex $r_{\bar{x}_i}$, and vertex $s_{\bar{x}_i}$ is adjacent to vertex r_{x_i} ; this is a 2-ring, identical to epoch 0 in Theorem E.3.1. Epoch 0 has the the following property:

Property E.7.2 *Any colouring sequence on G will deadlock in epoch 0 unless each process component pair has a token assigned to the token pool of either $(P_{x_i}, P_{\bar{x}_i})$, or $(P_{\bar{x}_i}, P_{x_i})$, $i = 1 \dots n$. Thus, the token assignment must be of at least size n . (Follows from Lemma E.2.1.)*

Property E.7.2 yields the following correspondence between assignments on F and token assignments of size n .

Property E.7.3 *The corresponding token assignment of a variable assignment on F assigns a token to the channel $(P_{x_i}, P_{\bar{x}_i})$ if x_i is true, or to $(P_{\bar{x}_i}, P_{x_i})$ if x_i is false.*

The j th epoch represents the j th clause of F , denoted (a_j, b_j, c_j) , where $a_j, b_j, c_j \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$. The process components $P_{a_j}, P_{\bar{a}_j}, P_{b_j}, P_{\bar{b}_j}, P_{c_j}$, and $P_{\bar{c}_j}$ form a 6-ring, while the remaining components have no vertices in the j th epoch. Process component P_{a_j} has two vertices in the j th component: a send vertex, $s_{a_j,j}$, and a receive vertex $r_{a_j,j}$; similarly, the other five components have a send and receive vertex that are correspondingly named. The arcs linking the 6 components are these: $(s_{a_j,j}, r_{\bar{a}_j,j})$, $(s_{\bar{a}_j,j}, r_{b_j,j})$, $(s_{b_j,j}, r_{\bar{b}_j,j})$, $(s_{\bar{b}_j,j}, r_{c_j,j})$, $(s_{c_j,j}, r_{\bar{c}_j,j})$, and $(s_{\bar{c}_j,j}, r_{a_j,j})$. These form a 6-ring, as illustrated in Figure E.12. The key property of the j th epoch is this:

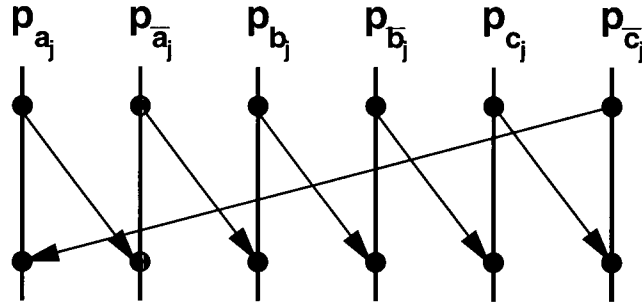


Figure E.12: The clause representation in epoch j .

Property E.7.4 No colouring sequence on G will deadlock in the j th epoch if and only if at least one of the channels has a token: $(P_{a_j}, P_{\bar{a}_j})$, $(P_{\bar{a}_j}, P_{b_j})$, $(P_{b_j}, P_{\bar{b}_j})$, $(P_{\bar{b}_j}, P_{c_j})$, $(P_{c_j}, P_{\bar{c}_j})$, $(P_{\bar{c}_j}, P_{a_j})$. (Follows from Lemma E.2.1.)

A refined version of property E.7.4 is more useful:

Property E.7.5 For any token assignment of size n such that no colouring sequence deadlocks on G in epoch 0, no colouring sequence on G will deadlock in the j th epoch if and only if at least one of the channels $(P_{a_j}, P_{\bar{a}_j})$, $(P_{b_j}, P_{\bar{b}_j})$, and $(P_{c_j}, P_{\bar{c}_j})$, has a token.

Proof: By property E.7.2, all token assignments that do not cause deadlock in epoch 0 only assign tokens to channels of the form $(P_{x_i}, P_{\bar{x}_i})$ or $(P_{\bar{x}_i}, P_{x_i})$. Hence, only channels $(P_{a_j}, P_{\bar{a}_j})$, $(P_{b_j}, P_{\bar{b}_j})$, and $(P_{c_j}, P_{\bar{c}_j})$ can have a token. By property E.7.4, no colouring sequence on G will deadlock in epoch j if one of these channels has a token. ■

We claim that given a token assignment of size n , any colouring sequence will complete on G if and only if the corresponding variable assignment satisfies F .

If an assignment x satisfies F , then every clause has at least one literal that evaluates to true. By Property E.7.3, in each of the j epochs at least one of the channels listed in Property E.7.5 will be allocated a token. Hence, by Property E.7.5 no colouring sequence will deadlock on G .

If an assignment x does not satisfy F then there is at least one clause in which all literals are false. Let (a_j, b_j, c_j) be the unsatisfied clause. By property E.7.3, the corresponding token

assignment will not assign a token to $(P_{a_j}, P_{\bar{a}_j})$, $(P_{b_j}, P_{\bar{b}_j})$, or $(P_{c_j}, P_{\bar{c}_j})$, hence, by Property E.7.5, all colouring sequences will deadlock.

Thus, NBAP_{SR} is NP-hard. ■

Since tokens are assigned on a per channel basis, token usage depends only on the two process components that comprise the channel. Consequently, the sufficiency of a token assignment can be verified in linear time. Thus, the easier problem BSP_{SR} is in P, implying that BAP_{SR} is NP-complete. We describe the verification algorithm and prove its correctness.

E.7.2 The Buffer Sufficiency Problem

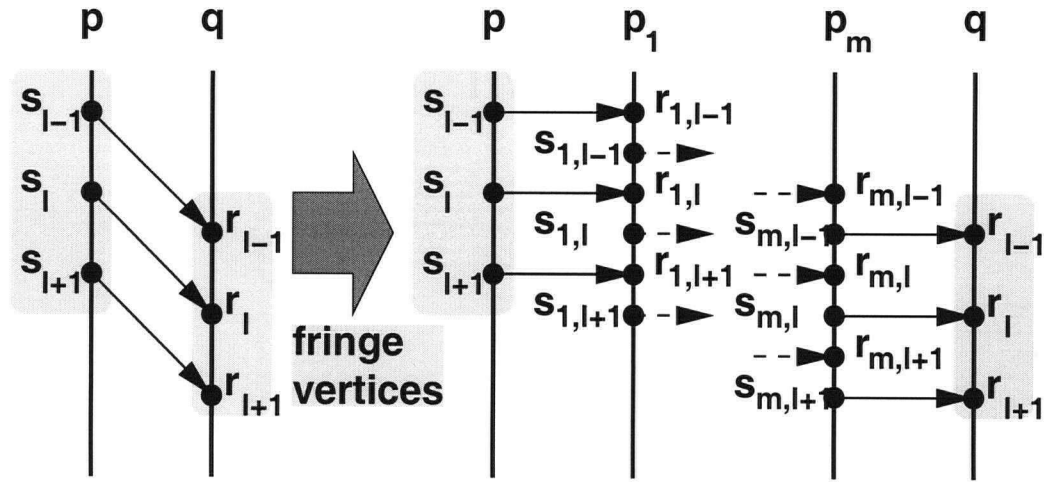
To verify the sufficiency of a token assignment, perform a colouring on G : at each step of the colouring a vertex of G is coloured according to the rules in section E.1. Using a queue to keep track of colourable vertices means that determining which vertex to colour next takes $O(1)$ time. Since each vertex changes colour at most twice—the maximum length of any colouring sequence is $2|V|$ colourings—colouring a graph takes $O(|V|)$ time. The token assignment is sufficient if and only if the colouring sequence completes. The algorithm's correctness follows immediately from the following theorem: any colouring sequence on G completes if and only if some colouring sequence on G completes. Thus, a token assignment is sufficient if and only if some colouring sequence on G completes.

Theorem E.7.6 *Let G be a communication graph and B a token assignment on G . Any colouring sequence on G completes if and only if a colouring sequence on G completes.*

Proof: For any communication graph G , we construct a new graph G' where every token is simulated by a process component, the size of the corresponding token assignment is zero, and every colouring sequence on G corresponds to a colouring sequence on G' , such that a colouring sequence on G completes if and only if the corresponding colouring sequence on G' completes. Since the token assignment on G' is zero, by Lemma E.2.2 a colouring sequence on G' completes if and only if every colouring sequence on G' completes. Hence, every colouring sequence on G completes if and only if a colouring sequence on G completes.

To simulate an m token channel (a channel that has been assigned m tokens) m process components are chained together. For each channel (P, Q) with m tokens, m process components P_1, P_2, \dots, P_m are interspersed between P and Q . The channel (P, Q) is replaced with these channels: $(P, P_1), (P_1, P_2), \dots, (P_{m-1}, P_m), (P_m, Q)$. Each arc from P to Q is replaced by a chain of arcs from $P \rightarrow P_1 \rightarrow P_2 \rightarrow \dots \rightarrow P_{m-1} \rightarrow P_m \rightarrow Q$. The replacement is illustrated in Figure E.13.

We claim that a colouring sequence, Σ , on G will deadlock if and only if the corresponding colouring sequence, Σ' , on G' deadlocks. First, we construct the correspondence and argue its

Figure E.13: Simulating m tokens by m components.

correctness. Second, we argue that sequence Σ deadlocks on G if and only if the corresponding sequence Σ' deadlocks on G' . Finally, we apply Lemma E.2.2 to prove our result.

Since the transformation is iterative—each m token channel is independent of the other channels—it is sufficient to derive the correspondence between the colouring sequence on G and the graph G' where a single m token channel has been replaced. Let (P, Q) denote the channel in G that is replaced in G' .

Let $(s_l, r_l) \in G$, $l = 1, 2, \dots$, denote the arcs from process component P to Q . The corresponding paths in G' are

$$(s_l, \underbrace{r_{1,l}, s_{1,l}}_{P_1}, \underbrace{r_{2,l}, s_{2,l}}_{P_2}, \dots, \underbrace{r_{m,l}, s_{m,l}}_{P_m}, r_l),$$

where each arc $(r_{k,l}, s_{k,l})$ is within process component P_k and each arc $(s_{k,l}, r_{k+1,l})$ is between process components P_k and P_{k+1} ; the vertices s_l and r_l , $l = 1, 2, \dots$ are called the **fringe vertices**.

A colouring sequence Σ can be represented as a sequence of differences (or moves), δ_i , between every two consecutive colourings χ_i and χ_{i+1} . The sequence $\Delta_\Sigma = \delta_1 \delta_2 \dots$ is a sequence of colouring game moves $\delta_i = \langle v, \text{colour} \rangle$ such that applying δ_i to colouring χ_i yields χ_{i+1} , the next colouring in Σ ; Δ_Σ can be derived from Σ and Σ can be derived from Δ_Σ and G . The sequence Δ_Σ comprises two types of moves: those that colour fringe vertices, called **fringe moves**, and those that do not, called **normal moves**.

Given a colouring sequence Σ on G , we transform it into the corresponding colouring sequence Σ' on G' . The transformation replaces some fringe moves in sequence Δ_Σ with sequences of moves, resulting in the corresponding move sequence $\Delta_{\Sigma'}$. This sequence comprises normal moves and **added moves**; added moves are a mixture of fringe moves and moves on the vertices within the added components P_i . There are four types of fringe moves in Δ_Σ : colour a send

vertex s_l yellow ($\langle s_l, \text{yel} \rangle$), colour a send vertex s_l green ($\langle s_l, \text{grn} \rangle$), colour a receive vertex r_l yellow ($\langle r_l, \text{yel} \rangle$), and colour a receive vertex r_l green ($\langle r_l, \text{grn} \rangle$). The transformation is performed in the order that the moves occur in sequence Δ_Σ .

- If $\delta_i = \langle s_l, \text{yel} \rangle$, then no action is taken.
- If $\delta_i = \langle s_l, \text{grn} \rangle$, we replace it with the sequence

$$\langle r_{1,l}, \text{yel} \rangle, \langle s_l, \text{grn} \rangle, \langle r_{1,l}, \text{grn} \rangle, \langle s_{1,l}, \text{yel} \rangle,$$

suffixed by the sequences

$$\langle r_{j,l}, \text{yel} \rangle, \langle s_{j-1,l}, \text{grn} \rangle, \langle r_{j,l}, \text{grn} \rangle, \langle s_{j,l}, \text{yel} \rangle, \quad j = 2 \dots k-1$$

where k is the smallest integer such that the move $\langle s_{k,l-1}, \text{grn} \rangle$ has not yet been inserted into the move sequence Δ_Σ , that is, vertex $s_{k,l-1}$ has not yet been coloured green.

- If $\delta_i = \langle r_l, \text{yel} \rangle$, we remove it from the sequence; it is restored when we replace the move $\langle r_l, \text{grn} \rangle$.
- If $\delta_i = \langle r_l, \text{grn} \rangle$ we replace this move with the sequence

$$\langle r_l, \text{yel} \rangle, \langle s_{m,l}, \text{grn} \rangle, \langle r_l, \text{grn} \rangle,$$

suffixed with the sequences

$$\langle r_{g_j, h_j}, \text{yel} \rangle, \langle s_{g_j-1, h_j}, \text{grn} \rangle, \langle r_{g_j, h_j}, \text{grn} \rangle, \langle s_{g_j, h_j+1}, \text{yel} \rangle, \quad j = 0 \dots k-1,$$

where $g_j = m - j$, $h_j = l + 1 + j$, and k is the smallest integer such that the move $\langle s_{m-k, l+1+k}, \text{yel} \rangle$ has not yet been inserted into the sequence, that is, vertex $s_{m-k, l+1+k}$ has not yet been coloured yellow. Since the head of this sequence colours $s_{m,l}$ green, $r_{m,l+1}$ could be coloured yellow, if $s_{m-1, l+1}$ is yellow, then $s_{m-1, l+1}$ could be coloured green followed by $r_{m,l+1}$ and finally $s_{m, l+1}$ could be coloured yellow; this colouring cascades down the added process components.

It is important to note that each of the replacement sequences is maximal, that is, no additional valid colouring moves on the chain process components P_i , $i = 1 \dots m$, may be suffixed to them. The new sequence looks like this:

$$\Delta_{\Sigma'} = \underbrace{\delta_1 \dots \delta_{h_1}}_{\text{normal moves}} \underbrace{\delta'_1 \dots \delta'_{g_1}}_{\text{added moves}} \underbrace{\delta_{h_1+1} \dots \delta_{h_2}}_{\text{normal moves}} \underbrace{\delta'_{g_1+1} \dots \delta'_{g_2}}_{\text{added moves}} \dots$$

Since G is a contraction of G' , all normal vertices are coloured by $\Delta_{\Sigma'}$ in the same order as in Δ_Σ . Recall that normal vertices are not adjacent to the process component chain, and hence, are not affected by the transformation. While normal vertices within process components P and

Q may depend on the order that the fringe vertices are coloured, the dependence is via process arcs, not communication arcs. Consequently, the normal vertices only depend on the order that the fringe vertices are coloured green. Fortunately, this order is preserved. By inspection, the replacement sequences of moves are valid. Thus, the transformed sequence $\Delta_{\Sigma'}$ is valid. Additionally, all green colouring moves on fringe vertices are preserved by the transformation; a vertex is coloured green by Δ_{Σ} if and only if the corresponding vertex is coloured green by $\Delta_{\Sigma'}$. The following property is key:

Property E.7.7 Δ_{Σ} deadlocks on G if and only if $\Delta_{\Sigma'}$ deadlocks on G' .

Proof: By contradiction, suppose that Δ_{Σ} deadlocks on G while $\Delta_{\Sigma'}$ can be extended, that is, another vertex colouring move may be suffixed to $\Delta_{\Sigma'}$. Let v be the vertex that can be coloured by the extension. Vertex v may either be a normal vertex, a fringe vertex, or a vertex belonging to a process chain. The latter is impossible because every replacement sequence of moves is maximal.

If v is a normal vertex, then its predecessors are either a normal vertex or a fringe vertex that has been coloured green. Since the transformation preserves the colourings of normal vertices and the order in which vertices are coloured green, if $\Delta_{\Sigma'}$ can be extended by colouring v , then so can Δ_{Σ} , which is a contradiction.

If v is a fringe vertex, there are four cases: either v is a send vertex s_l being coloured yellow or green, or v is a receive vertex r_l being coloured yellow or green. The transformation does not affect moves that colour send vertices yellow and such a colouring only depends on its process component predecessor being green. Hence, if the colouring can be suffixed to $\Delta_{\Sigma'}$, it can also be suffixed to Δ_{Σ} ; resulting in a contradiction. If the extension colours the send vertex green, this means that the original sequence Δ_{Σ} can be extended by either adding the colourings $\langle s_l, \text{grn} \rangle$ or $\langle r_l, \text{yel} \rangle \langle s_l, \text{grn} \rangle$, depending on whether r_l has been coloured yellow or not in the original sequence Δ_{Σ} ; thus, it is a contradiction.

Similarly, if v is a fringe receive vertex being coloured green, this is not possible, because the transformation colours fringe receive vertices yellow, then green, by a single replacement sequence. Finally, if v is a fringe receive vertex r_l that can be coloured yellow, the original sequence Δ_{Σ} can be extended by the move $\langle r_l, \text{grn} \rangle$, because in the original sequence the corresponding send vertex s_l has already been coloured green. Thus, we have another contradiction.

In the other direction, if the original sequence can be extended, then transforming the extension of the sequence Δ_{Σ} yields an extension to the presumably deadlocked sequence $\Delta_{\Sigma'}$. Thus, Δ_{Σ} deadlocks on G if and only if $\Delta_{\Sigma'}$ deadlocks on G' . ■

A corollary of Property E.7.7 is that the colouring sequence Σ deadlocks if and only if the colouring sequence Σ' deadlocks.

By Lemma E.2.2 a colouring sequence on G' completes if and only if all colouring sequences on G' complete. Hence, a colouring sequence on G completes if and only if all colouring sequences on G complete. ■

Corollary E.7.8 *A colouring sequence on G completes if and only if the token assignment is sufficient.*

E.7.3 The Nonblocking Buffer Allocation Problem

For the Nonblocking Buffer Allocation Problem, the algorithm derived in section E.3.3 suffices with a small modification. Since the token pools are per channel, rather than per process component, the computation must be performed on a per pool basis. Hence, there is an additional factor of n in the runtime. Since each process may be using up to n channels, the runtime of the algorithm becomes $O(|V|n^2 + |V|n \log(|V|n))$; the cost increases because the number of allocations to be made becomes quadratic in n .

E.8 Summary

As message passing becomes increasingly popular, the problem of determining k -safety plays an increasingly important role. The relevance of this problem grows as more and more functionality of message passing systems is offloaded to the network interface card, where limited buffer space is a serious issue. Even if message passing is kept in main memory, buffer space can still be limited due to the sometimes very large data sets used in many parallel and distributed programs. Unfortunately, determining k -safety is intractable.

We have shown that in the receive buffer model, determining the number of buffers needed to assure safe execution of a program is NP-hard, and that even verifying whether a number of assigned buffers is sufficient is coNP-complete. On the positive side, if we require that no send blocks, we provide a polynomial time algorithm for computing the minimum number of buffers. By allocating this number of buffers, safe execution is guaranteed. In addition, we have implemented the NBAP_r algorithm, and it is now part of the Millipede debugging system.

For systems with only send buffers, the Buffer Allocation Problem remains NP-complete. In addition, we conjecture that the Buffer Sufficiency Problem can be solved in polynomial time because the order of the sends in each process is fixed. The Nonblocking Buffer Allocation problem for systems with only send buffers can be solved in polynomial time.

For systems with both send and receive buffers, the Buffer Allocation Problem as well as the Buffer Sufficiency Problem remain intractable. More interestingly, the Nonblocking Buffer Allocation problem has become intractable.

For systems with unidirectional channel buffers, both the Buffer Sufficiency Problem as well

as the Nonblocking Buffer Allocation Problem have polynomial time algorithms. However, the Buffer Allocation Problem still remains an NP-complete problem. The results (conjectures) are summarized in Table E.1.

Problem	Buffer Placement			
	Receive	Send	Send & Receive	Channel
BAP	NP-hard	NP-hard	NP-hard	NP-complete
BSP	coNP-complete	(P)	coNP-complete	P
NBAP	P	P	NP-hard	P

Table E.1: Results for the three problems under the four different buffer placement schemes.