Protocol Verification in Millipede

Jan Bækgaard Pedersen & Alan Wagner
Department of Computer Science, University of British Columbia, Vancouver, Canada

Abstract. In this paper we present MOPED, a module in the Millipede debugging system. Millipede is a multi-level debugging system for parallel message passing programs. MOPED allows the user to specify a protocol to which the communication of the program should adhere, and automatically have all messages sent in the system checked against the protocol. The specification language is small, and easy to use, yet powerful enough to specify a wide range of protocols. Program variables can be passed easily to the verification module, allowing the construction of more dynamic protocol specifications. Protocols can be specified incrementally, starting out very general working towards a more complex specification. Finally, the verification module can be run either online, that is, while the application is executing, or offline, using log files generated when the application was executed.

1 Introduction

Debugging parallel and distributed programs is a difficult task. Studies have shown that users typically spend the same amount of time debugging their parallel code as they do initially writing it [13]. Unfortunately, tools to aid the programmer in the debugging process have not been widely adopted [6, 11, 12]. It has been estimated that between 35% to 90% of programmers still rely on print statements embedded in their code as the main debugging tool [6, 13]. This approach becomes less applicable when dealing with a number of asynchronously executing processes passing messages between each other.

Millipede is a multi-level debugging environment, a collection of tools, that attempt to provide the programmer with an easy to use tool for debugging parallel message passing programs written in PVM or MPI.

Millipede contains several modules (sub tools), each tailored to a specific debugging task; ranging from a tool to extract a process from a parallel system to allow sequential debugging to deadlock detection and correction and, as we describe in this paper, a module for checking and verifying communication protocols.

The multi-level debugging strategy can be summarized in the following points:
Instead of providing one monolithic tool to debug and correct all types of errors, we believe that providing tools specifically tailored toward the type of error in question is more likely to be successful.

It is necessary to provide a flexible and easy to use collection of specific tools.

At the lowest level in Millipede is the Sequential Debugging Module that is used to extract a sequential process from a parallel system, and then sequentially debug the process using any sequential debugger such as gdb or purify. This module is described in detail in [14], and briefly explained in section 3 along with the other modules in Millipede.

The module described in this paper is the protocol verification module, also known as MOPED (Millipede Online Protocol Error Detection). The main purpose of this module is to allow the programmer to easily write a specification of the protocol that he has implemented and have Millipede check all messages sent in the system against the protocol specification and report any messages that do not adhere to the protocol. The protocol can be specified at different levels of granularity, from very general to very refined as the debugging process evolves.

The rest of the paper is organised as follows: Section 2 discusses some of the related work in this field. In section 3 we briefly give an introduction to Millipede and the other existing modules. Sections 4 through 6 covers the MOPED protocol language, syntax and semantics, and sections 7 and 8 give a couple of examples of how to use this module. In section 9 and 10 we describe the two different modes that the protocol verification module can be run in. Section 12 briefly discusses the implementation of MOPED, and 13 offers some ideas for improvements and future features.

2 Related Work

A common denominator for all 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 debugging parallel programs. However, most existing tools require the protocol/model to be specified/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 of the more well known approaches for protocol specification include CSP [8], CTL/μ-calculus [2] and coloured Petri nets [10]. Specifications written in CSP can be verified and checked using the FDR model checking tool [16]. FDR (Failures-Divergence Refinement) allows 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 that use this abstraction in model checking include SMV, Mur#[3] and VIS [1]. 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 [9] system also falls into this category of tools, although it is based on LTL (Linear Temporal Logic) and not CTL.

The approach to model checking with coloured petri nets is slightly different; the programmer 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. The amount of work it takes to transcribe a petri net model to a C program is large, and the risk for introducing errors is increased as the translation from a graphical representation and a functional specification must be performed manually.

3 Millipede

In this section we will briefly explain the overall structure of the Millipede debugging system. The reason for including this is twofold:

1. To give the reader a better understanding of the idea behind multi-level debugging, and briefly familiarise him with the other modules of Millipede.

2. To easier understand how the lower levels of the Millipede system ties into the protocol verification module.

Millipede consists of a number of modules, each capable of performing debugging at a different point in the development process (see Fig. 1). At the lowest level in Millipede is the Sequential Debugging Module that allows the programmer to individually debug each of the processes in the system in a sequential debugging environment. Once the errors in the sequential parts of the parallel code have been found, there remains the possibility of communication related errors. Identifying and correcting communication errors is the main objective of the other modules in Millipede. The next level, a Message Debugging Module, allows the user to control the contents of messages sent and received in one or more processes while the entire program is executing. The higher levels are tailored towards deadlock detection and correction as well as communication protocol verification.

3.1 The sequential debugging module

The Sequential Debugging Module works as follows. It first requires the user to compile and execute the program in parallel. During execution Millipede intercepts messages sent to all
of the processes in the system and writes the contents of these messages to a file, one file for each process. In addition, the return values of all message passing function calls are also written to the file. The log files collected for each process stores the messages in the order they were released to the process. These log files can be used for sequential debugging of one process of the parallel system, but also for the offline protocol checking. A more thorough explanation of this module with examples can be found in [14].

3.2 The deadlock correction module

The deadlock correction module of Millipede can be used to locate and correct deadlocks. An analysis is performed on the messages left in the system and on the most recent sent messages, and suggestions to the cause of the deadlock are given by Millipede. In addition, the system attempts to resolve the deadlock by suggesting a way to change the source code, such that the deadlock disappears. The module is not intended to provide a general deadlock correction mechanism, but rather assumes that the type of errors introduced are mainly typographical errors in the message passing calls, and that the number of errors is small. A more detailed explanation of this module can be found in [15].
4 MOPED Protocol Specification

In order to use the Millipede Online Protocol Error Detection Module the programmer must specify the protocol using the MOPED language. The following sections are concerned with the grammar and semantics of the protocol language, and the writing of protocols. Before getting started on defining protocols we first introduce a few concepts and definitions.

4.1 Preliminaries

A group of processes is a number of processes all spawned from the same \texttt{pvm\_spawn} call. There can be several groups of the same program depending on the number of spawn calls. An instance is one process of 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 \texttt{pvm\_send} or a \texttt{pvm\_recv} or an identifier. If an identifier is used Millipede will search the appropriate source file for comments of the form \texttt{/* PROTOCOL(line-label) */} where line-label is the identifier used in the specification of the protocol.

4.2 Protocol Contents

To use the protocol verification module of Millipede the programmer first writes a protocol specification file that contains a description of the protocol that she wishes to check her program against. This specification file is then passed to the Millipede system.

A protocol file consists of a number of lines that specify which sends can send to which receives. One of the powerful features of the MOPED module is the ability to start out by specifying a very general version of the protocol, check it, and as errors are detected and corrected further specialize the protocol step by step. An example is shown in section 7.

A protocol consists of a number of lines of the form:

\[
pcname_1[e_1] \{e_2\} \{e_3\} \rightarrow pcname_2[e_4] \{e_5\} \{e_6\}
\]

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

\[ \forall \ id : RelExpression \]

The first part states that a process created from program \texttt{pcname_1} with instance number \texttt{e_2} (see section 13 on how to change a process’ instance number) in group \texttt{e_1} can send from a send call in line \texttt{e_3} to a receive call in a process created from a program \texttt{pcname_2} with instance number \texttt{e_5} in group \texttt{e_4} with a receive call in line \texttt{e_6}. Values for \texttt{e_1}, \texttt{e_2} and \texttt{e_3} can be
either omitted, a number or an identifier. If \( e_3 \) is the identifier \( xyz \) and \( pgname_1.c \) contains a `pvm_send` followed by a `/* PROTOCOL (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 then they are bound to the group and the instance number of the process sending the message which is being verified.

If any or all of \( e_1, e_2, e_3 \) are omitted no check is done for the missing expression. This is equivalent to a wild card match. Values for \( e_4, e_5, e_6 \) can be either expressions or they can be omitted. Again, if omitted a wild card match is performed. If an expression is entered this expression is evaluated and matched to the actual values of the group, instance and line number of the process that received the message. A quantifier introduces a new variable to be used in the expressions \( e_1, \ldots, e_6 \). These can be qualified by both lower and upper bounds or bound by other expressions. Examples will be given in sections 7 and 8. A message (sent from a sender to a receiver) is a tuple:

\[
\mathcal{M} = (P_s, P_r, (G_s, I_s, L_s), (G_r, I_r, L_r), N_s, N_r)
\]

where \( P_s \) and \( P_r \) are the program names of the sender and receiver process, \( G_s, I_s, L_s \) denotes the group, instance and line of the send and \( G_r, I_r, L_r \) denotes those of the receive. \( N_s \) and \( N_r \) are the total number of processes in group \( G_s \) and \( G_r \).

## 5 The MOPED Grammar

For completeness the grammar of the MOPED language has been included in Fig. 2. This grammar is a subset of the grammar for expressions in the C programming language.

## 6 MOPED Semantics

In Fig. 3 the semantics for computing expressions and relational expressions is shown. \( \sigma \) denoted a symbol table that links values and variables together. The first rule of the semantics \( (\mathcal{E}["\"]\sigma = \bot) \) simply states that the semantic value of evaluating the empty expression is the special symbol \( \bot \) (read bottom). This is used since the grammar allows the expressions \( e_1, \ldots, e_6 \) to be left empty, and for these cases (referred to as wild card matches) we may compare a numeric value to the \( \bot \) symbol, which must evaluate to true. Thus the use of the `\( \triangleq \)` operator defined in Fig. 5.

The \( \Sigma \) function in Fig. 4 is used to add variable/value pairs to the symbol table.
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 ::= ∀ Identifier ’:’ RelExpression
Index ::= ε | Number | Identifier
ClassExpression ::= ε | Expression
Expression ::= Expression ’*’ Expression | Expression ’/’ Expression
| Expression ’+’ Expression | Expression ’–’ Expression
| Expression ’%’ Expression | Expression ’ˆ’ 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 2: The MOPED grammar.

Figure 6 is interpreted in the following way:

- \((P_s = \beta \land P_r = \delta)\): Determines if the line matches the message with respect to the names of the sender and receiver.

- \(\bigwedge_{Q_3=E_{vr}} (v, r) \in \sigma \land \mathcal{R}[r] \sigma\): This expression states that all quantified values are correct, that is, within the boundaries of their definitions.

- \(E[e_4] \sigma \models G_r\): States that the senders group is allowed to communicate with the receivers group.

- \(E[e_5] \sigma \models I_r\): The instance of the sender is allowed to send to the receiver with instance number \(I_r\).
$$\mathcal{E}[\text{""""}]\sigma = \bot \quad \mathcal{R}[\text{true}]\sigma = \text{true}$$
$$\mathcal{E}[\text{Number}]\sigma = \text{Number} \quad \mathcal{R}[\text{false}]\sigma = \text{false}$$
$$\mathcal{E}[(\text{Identifier})]\sigma = \sigma(\text{Identifier}) \quad \mathcal{R}[e_1 < e_2]\sigma = \mathcal{E}[e_1]\sigma < \mathcal{E}[e_2]\sigma$$
$$\mathcal{E}[e_1 \times e_2]\sigma = \mathcal{E}[e_1]\sigma \times \mathcal{E}[e_2]\sigma \quad \mathcal{R}[e_1 > e_2]\sigma = \mathcal{E}[e_1]\sigma > \mathcal{E}[e_2]\sigma$$
$$\mathcal{E}[e_1 / e_2]\sigma = \mathcal{E}[e_1]\sigma / \mathcal{E}[e_2]\sigma \quad \mathcal{R}[e_1 \leq e_2]\sigma = \mathcal{E}[e_1]\sigma \leq \mathcal{E}[e_2]\sigma$$
$$\mathcal{E}[e_1 + e_2]\sigma = \mathcal{E}[e_1]\sigma + \mathcal{E}[e_2]\sigma \quad \mathcal{R}[e_1 \geq e_2]\sigma = \mathcal{E}[e_1]\sigma \geq \mathcal{E}[e_2]\sigma$$
$$\mathcal{E}[e_1 - e_2]\sigma = \mathcal{E}[e_1]\sigma - \mathcal{E}[e_2]\sigma \quad \mathcal{R}[e_1 = e_2]\sigma = \mathcal{E}[e_1]\sigma = \mathcal{E}[e_2]\sigma$$
$$\mathcal{E}[e_1 \% e_2]\sigma = \mathcal{E}[e_1]\sigma \mod \mathcal{E}[e_2]\sigma \quad \mathcal{R}[e_1 \neq e_2]\sigma = \mathcal{E}[e_1]\sigma \neq \mathcal{E}[e_2]\sigma$$
$$\mathcal{E}[\exp(e_1)\sigma, e_2]\sigma = \exp(\mathcal{E}[e_1]\sigma, \mathcal{E}[e_2]\sigma) \quad \mathcal{R}[r_1 \& \& r_2]\sigma = \mathcal{R}[r_1]\sigma \& \mathcal{R}[r_2]\sigma$$
$$\mathcal{E}[\sqrt{e}]\sigma = \sqrt{\mathcal{E}[e]\sigma} \quad \mathcal{R}[r_1 || r_2]\sigma = \mathcal{R}[r_1]\sigma \lor \mathcal{R}[r_2]\sigma$$
$$\mathcal{E}[\text{sqrt}(e)]\sigma = \sqrt{\mathcal{E}[e]\sigma} \quad \mathcal{R}[(r)]\sigma = \mathcal{R}[r]\sigma$$

Figure 3: Semantics for evaluating expressions and relational expressions.

$$\Sigma[e](v)\sigma = \begin{cases} 
\sigma \quad & \text{if } e = \text{""""} \\
\sigma \cup \{e = v\} \quad & \text{if } e \neq \text{""""}
\end{cases}$$

Figure 4: Adding elements to the symbol table.

$$\doteq: \{\bot\} \cup \mathbb{N} \times \{\bot\} \cup \mathbb{N} \to \{\text{true}, \text{false}\}$$

$$\begin{align*}
\bot \doteq n & \quad \text{:=} \quad \text{true} \\
n \doteq \bot & \quad \text{:=} \quad \text{true} \\
\bot \doteq \bot & \quad \text{:=} \quad \text{true} \\
n \doteq n & \quad \text{:=} \quad \text{true} \\
n \doteq m \ (n \neq m) & \quad \text{:=} \quad \text{false} \\
n \doteq m & \quad \text{:=} \quad \begin{cases} 
\text{true} \quad & \text{if } n = m \\
\text{false} \quad & \text{if } n \neq m
\end{cases}
\end{align*}$$

Figure 5: The $\doteq$ operator.

- $\mathcal{E}[e_0]\sigma \doteq L_r$: The receive in line number $L_r$ may receive from the specified sender line number.

A message $\mathcal{M}$ is said to be verified or checked with respect to a protocol specification line
\[
\mathcal{B}[\beta[e_1][e_2][e_3] \rightarrow \delta[e_4][e_5][\varepsilon] :: Q] \mathcal{M} = \\
(P_s = \beta \land P_r = \delta) \land \left( \bigwedge_{Q \exists q = \forall \forall} ((v, \cdot) \in \sigma \land \mathcal{R}[r]\sigma) \right) \land \\
(\mathcal{E}[e_4] \sigma = G_r) \land (\mathcal{E}[e_5] \sigma = I_r) \land (\mathcal{E}[\varepsilon] \sigma = L_r)
\]

where \( \sigma = \Sigma \mathcal{H}[e_3](L_s) (\Sigma \mathcal{H}[e_2](I_s) (\Sigma \mathcal{H}[e_1](G_s) \emptyset) \)

and \( Q = \forall v_i : \text{RelExp}; \ldots \forall v_j : \text{RelExp}; \)

Figure 6: Semantics for a MOPED line.

L, if

\[
\mathcal{B}[\varepsilon] \mathcal{M} = \text{true}
\]

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

- If \( e_i \) is a number (\( c_i \)) \( e_i \) is replaced by \( \alpha_i \) and the quantifier \( \forall \alpha_i : \alpha_i = c_i \) is added to \( Q \).
- If \( e_i \) is an identifier replace all its occurrences by \( \alpha_i \). This step is not necessary, but it clarifies the following explanation.
- If \( e_i \) is left blank, replace the blank with \( \alpha_i \) 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. The table in Fig. 7 shows the steps taken in order to check a message against a protocol line. If any of the checks past step 2 fail, the protocol is violated by the message, and if all checks evaluate to true, the message is said to be verified by the protocol line.
<table>
<thead>
<tr>
<th>Step</th>
<th>Check</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$(P_s = \beta \land P_r = \delta)$</td>
<td>If false move on to the next protocol line. If true go to step 2.</td>
</tr>
<tr>
<td>2</td>
<td>$\bigwedge_{Q \exists q = a_i: r_{i-1,2,a}} \mathcal{R}[r_i] \sigma$</td>
<td>This checks if the sender part of the message matches the sender part of the protocol line. If false move on to the next protocol line. If true go to step 3.</td>
</tr>
<tr>
<td></td>
<td>$*$</td>
<td>Check the rest (all) of the quantifiers. If false report a quantifier error. If true go to step 4.</td>
</tr>
<tr>
<td>4</td>
<td>$\mathcal{E}[e_i] \sigma = G_r$</td>
<td>Check if the receiver group may receive this message. If false report a group error. If true go to step 5</td>
</tr>
<tr>
<td>5</td>
<td>$\mathcal{E}[e_5] \sigma = I_r$</td>
<td>Check if the receiver instance may receive this message. If false report an instance error. If true go to step 6</td>
</tr>
<tr>
<td>6</td>
<td>$\mathcal{E}[e_6] \sigma = L_r$</td>
<td>Check if the receiver line may receive this message. If false report a line error. If true protocol line verified/checked message according to semantics of figure 6</td>
</tr>
</tbody>
</table>

**Figure 7:** How to check a message against a protocol line.

## 7 Examples

In this section we present a few smaller examples and one larger example of how to write a protocol.

### 7.1 Simple examples

As stated in the previous section a protocol can start out being very general. In Fig. 8 the smallest possible protocol is shown.

This protocol consists of only one line:
which states that any \( \beta \) process can send to any other \( \beta \) process regardless of instance or line number. The picture in Fig. 8 shows that \( \beta \) processes send to other \( \beta \) processes, that any \( \beta \) process can send to any other, and that any send can send to any receive.

We can specialize this very simple protocol to represent a system where \( \beta \) process number \( i \) can send to another \( \beta \) process with instance number \( i + 1 \), and where process number \( n - 1 \) sends to instance number 0. In summation:

\[
\beta[0]() \rightarrow \beta[1]() ;
\beta[1]() \rightarrow \beta[2]() ;
\vdots
\beta[n-1]() \rightarrow \beta[0]() ;
\]

Or in short notation:

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

In Fig. 9 this protocol is shown graphically with the correct MOPED line.

Consider a more complicated example that also include the use of line numbers. The pseudocode for the pipe-and-roll matrix multiplication algorithm\[7\] is shown in Fig. 10. Processes communicate sub blocks 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 Fig. 11).
As we can see from the 2 functions Pipe_A and Roll_B, a process executing a pipe call can 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 $\sqrt{N} \times \sqrt{N}$). Let us assume, for simplicity that $N = 16$ in the following, that is, we are working with a $4 \times 4$ grid of processes.

A process with instance $j$ doing a pipe_A operation can send to process $(j + 1)\%4$, and a process $j$ doing a roll_B operation can send to process $(j + 12)\%16$. This can be expressed by the following two MOPED lines:

$$\text{Matrix}[\{j\}(\text{SendPipe}) \rightarrow \text{Matrix}[\{(j + 1)\%4\}(\text{ReceivePipe}) :: \forall j : j < 16;$$

$$\text{Matrix}[\{j\}(\text{SendRoll}) \rightarrow \text{Matrix}[\{(j + 12)\%16\}(\text{ReceiveRoll}) :: \forall j : j < 16;$$

The graphical representation can be seen in Fig. 11. This only includes the communication between the worker processes (called Matrix).

To add protocol lines to verify communication between the master (Master) and the workers add the following two lines:

$$\text{Master}[\{0\}(\text{SendParams}) \rightarrow \text{Matrix}[\{}(\text{ReceiveParams});$$

$$\text{Matrix}[\{}(\text{SendResult}) \rightarrow \text{Master}[\{0\}(\text{ReceiveResult});$$

Also note that the group numbers have been left out to simplify the description of the protocol.
Master:
Let \( N \) be the number of processors
Map concurrent computer on to array of \( \sqrt{N} \times \sqrt{N} \) processors
Distribute sub blocks of \( A \) and \( B \) to processors
Await sub block results in matrix \( C \)

Matrix (slave):
Initialize sub block matrix \( C \) to 0
Receive sub blocks \( A \) and \( B \)
For \( i=0 \) to \( \sqrt{N}-1 \) do
\( T = \text{Pipe}_A \)
\( C = C + T \times B \)
Roll_B
Send sub block \( C \) 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 10: Pseudocode for the pipe-and-roll matrix multiplication algorithm.

7.2 Limitations

By looking closer at the communication pattern in the program pseudocode it becomes clear that the pipe communication does not have wrap-around, that is, the last processor in the pipe does not 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 state in the application. In section 13 we describe a way to resolve this problem and expand the set of protocols that can be specified.

8 Example

Let us consider a parallel master/slave program to solve a hyperbolic differential equation. There is one master process and $n$ slave processes. Figure 12 shows the algorithm for the master and the slave programs.

The most general protocol, $\mathcal{P}_1$, (covering all sends) we can specify for the master/slave system is illustrated in Fig. 13. This $\mathcal{P}_1$ protocol reads as follows: “Any send in any instance in any group of a Master program can send to any receiver in any instance in any group of a Slave program (1) and vice versa (2), and any send in any instance of any group of the Slave program can send to any other Slave (3)”.

$\mathcal{P}_1$ is not very useful, as it does not say 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 Master program with instance 0). This changes the left part of the first line and the right part of the second line in Fig. 13 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 $Slave[0]\{\}$. Let $\mathcal{P}_1'$ denote this version of the protocol as shown in Fig. 14.

Now, knowing that the communication between the slaves is according to Fig. 15, that is, slave number $i$ can send to slave number $i + 1$ if $i < n - 1$ (assuming the system has $n$ slave processes) and also that slave number $i$ can send to slave number $i - 1$ if $i > 0$, we can incorporate this into the protocol and get the second version, as seen in Fig. 16.
Master:

Send parameters to slaves 0, ..., n - 1 \(/^* \text{PROTOCOL(MS)} */^\)

Repeat n times:
  Receive result from slave \(/^* \text{PROTOCOL(MR)} */^\)

Slave:

Receive parameters from master \(/^* \text{PROTOCOL(SR)} */^\)

Loop n times:
  if \(i\) > 0
    Send to slave \(id - 1\) \(/^* \text{PROTOCOL(S1)} */^\)
  if \(i\) < \(n - 1\)
    Send to slave \(id + 1\) \(/^* \text{PROTOCOL(S2)} */^\)
  calculate
    if \(i\) > 0
      Receive from slave \(id - 1\) \(/^* \text{PROTOCOL(R1)} */^\)
    if \(i\) < \(n - 1\)
      Receive from slave \(id + 1\) \(/^* \text{PROTOCOL(R2)} */^\)
  Send result to Master \(/^* \text{PROTOCOL(SS)} */^\)

Figure 12: Pseudocode for master/slave algorithm for a differential equation solver.

Note that line 3 has been split into line 3a (representing \(i\) send to \(i + 1\) situation) and line 3b (representing \(i\) send to \(i - 1\) situation). Also note the use of the two quantifier expressions following the lines. Looking closer at the lines 3a and 3b and comparing with the pseudocode in Fig. 12 we see that the \(P2\) protocol 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 was delivered to the wrong receive there would be a violation of the communication protocol, so we need to add this information to the protocol specification. Thus, line 3a represents the message passed between \(S1\) send and \(R1\) receive and line 3b represents the message passed between \(S2\) send and \(R2\) receive. Adding this to the protocol specification...
we obtain the third version as shown in Fig. 17. For completeness we added line information about the parameter and result messages sent to and from the master.

Figure 18 shows an extended version of $P_3$, where we have added information about the
1:  Master[0]{0}( )  →  Slave[0]{0}( );
2:  Slave[0]{0}( )  →  Master[0]{0}( );
3a:  Slave[0]{i}( )  →  Slave[0]{i + 1}( )  : ∀ i : i < n – 1;
3b:  Slave[0]{i}( )  →  Slave[0]{i – 1}( )  : ∀ i : 0 < i;

Figure 16: P2 – Version 2 of the protocol.

1:  Master[0]{0}(MS)  →  Slave[0]{0}(SR);
2:  Slave[0]{0}(SS)  →  Master[0]{0}(MR);
3a:  Slave[0]{i}(S1)  →  Slave[0]{i + 1}(R1)  : ∀ i : i < n – 1;
3b:  Slave[0]{i}(S2)  →  Slave[0]{i – 1}(R2)  : ∀ i : 0 < i;

Figure 17: P3 – Version 3 of the protocol.

instance of the slaves in lines 1 and 2. Furthermore, we have 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 it does allow the system to predict which sends/receives are legal, whereas P3 can only be checked, not predicted (see section 11 for more information on protocol prediction).

1:  Master[0]{0}(MS)  →  Slave[0]{i}(SR)  : ∀ i : (0 ≤ i) & (i < n);
2:  Slave[0]{i}(SS)  →  Master[0]{0}(MR)  : ∀ i : (0 ≤ i) & (i < n);
3a:  Slave[0]{i}(S1)  →  Slave[0]{i + 1}(R1)  : ∀ i : (0 ≤ i) & (i < n – 1);
3b:  Slave[0]{i}(S2)  →  Slave[0]{i – 1}(R2)  : ∀ i : (0 < i) & (i < n);

Figure 18: P3’ – Extended version 3 of the protocol.

9 Online Protocol Checking

The protocol verification module of Millipede can be used in two different modes: online or offline. The online mode checks the protocol specification as the communication takes place; each message sent in the system is captured by Millipede and checked against the specification. If an error occurs, that is, there is a violation of the protocol, 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 8. The first version of the protocol can be very general, and then gradually refined until errors are discovered. Once the error has been corrected, the protocol can be further refined, if the program still does not function
correctly. If a program deadlocks the deadlock detection and correction module can also be deployed to more easily locate and fix the error. This module is briefly described in Section 3.2.

9.1 Strictness

A protocol can be verified using different levels of strictness. When using the refinement technique, that is, starting out with a simple protocol, some messages might not match any lines, thus violating the protocol. The programmer might not perceive this as a violation as the protocol is not totally specified; if this is the case a lower level of strictness can be adopted. Figure 19 shows the 3 different levels of strictness that MOPED currently supports.

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 or more protocol lines may match with respect to program name and sender quantifiers.</td>
</tr>
<tr>
<td>2</td>
<td>At least one protocol line must match with respect to program name and sender quantifiers.</td>
</tr>
<tr>
<td>3</td>
<td>Exactly one protocol line must match with respect to program name and sender quantifiers.</td>
</tr>
</tbody>
</table>

Figure 19: 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 could match more than one protocol line), and level 3 if a unique full specification has been given.

10 Offline protocol verification

As described in the previous section Millipede can verify the protocol online, that is, while the program is running all messages are checked against the protocol. However, if Millipede is generating log files while the program is run, the protocol verification can be performed offline. All the information needed to check the protocol can be extracted from the set of log files generated and stored by Millipede. To use offline protocol verification simply switch Millipede to offline protocol verification using the supplied control center program, run and specify the set of log files and the protocol file to be used.
11 Protocol Prediction

As mentioned earlier, if all MOPED 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 Fig. 18 the prediction table is shown in Fig. 20 (for n=4):

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master$[0]{0}(MS)$</td>
<td>Slave$[0]{0}(SR)$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Slave$[0]{1}(SR)$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Slave$[0]{2}(SR)$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Slave$[0]{3}(SR)$</td>
<td>1</td>
</tr>
<tr>
<td>Slave$[0]{0}(SS)$</td>
<td>Master$[0]{0}(MR)$</td>
<td>2</td>
</tr>
<tr>
<td>Slave$[0]{1}(S1)$</td>
<td>Slave$[0]{1}(R1)$</td>
<td>3a</td>
</tr>
<tr>
<td>Slave$[0]{2}(S2)$</td>
<td>Slave$[0]{0}(R2)$</td>
<td>3b</td>
</tr>
<tr>
<td>Slave$[0]{2}(SS)$</td>
<td>Master$[0]{0}(MR)$</td>
<td>2</td>
</tr>
<tr>
<td>Slave$[0]{2}(S1)$</td>
<td>Slave$[0]{3}(R1)$</td>
<td>3a</td>
</tr>
<tr>
<td>Slave$[0]{2}(S1)$</td>
<td>Slave$[0]{1}(R2)$</td>
<td>3b</td>
</tr>
<tr>
<td>Slave$[0]{3}(SS)$</td>
<td>Master$[0]{0}(MR)$</td>
<td>2</td>
</tr>
<tr>
<td>Slave$[0]{3}(S1)$</td>
<td>Slave$[0]{2}(R2)$</td>
<td>3b</td>
</tr>
</tbody>
</table>

Figure 20: Prediction table.

A prediction table can help to determine if the protocol specified matches what the programmer had in mind. Naturally there is always a risk that the an error is present in the protocol specification; this is similar to the risk mentioned in section 2 of introducing errors into the implementation of a protocol that has been verified using a model checker. However, we believe that the number of errors introduced here would be considerably smaller than in the implementation stage.

12 Implementation

As with all other Millipede modules the MOPED module is a separate process that runs the protocol checking algorithm. It receives messages from the main module containing information about send/receives that have taken place and it verifies them using the specification.
When the module is run offline (i.e. the application is not currently running) the messages are extracted from the log files. (In Fig. 21 the thick solid line represents the online checking, and the dashed line represents the offline checking)

The MOPED module reads the protocol specification file and generates a parse tree that can be traversed at run time and checked against the group, instance and line number values for the sender and the receiver. The protocol lines are checked against a message one at a time, and depending on the strictness level, errors are reported to the user.

![Diagram of Millipede overview](image-url)
13 Future Work

A number of interesting extensions should be added to the protocol verification module in order to strengthen the quality of the checks performed. We will briefly describe some of these in the following.

Since we are working with message passing systems like PVM [5] and MPI [4] where all sends are annotated with a message tag, it is possible to add information about message tags to a MOPED protocol line. This means expanding $\beta[\{\}()] \rightarrow \delta[\{\}()]$; to $\beta[\{\}()] < > \rightarrow \delta[\{\}()]$; where $< >$ represents an expression that determines the message tag.

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

$$e \in [v_1, v_2, \ldots, v_n]$$

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

This could 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 the notion of typed channels assure that the correct type of data is always received on a channel.

However, in C or C++ the notion of channels does not exist, and when using message passing a static analysis and type checking is not possible.

A possible solution could be specifying message content using a specification language like XDR or XML for defining data types, or using the MPI_Datatype, which specifies an internal message data type, for all calls.

13.1 Defining values

Introducing defines in the protocol file and the ability to get information about the size of a group makes writing the specification more dynamic and much easier to read. Consider the following protocol code:

```plaintext
define a "Wave_master";
define b "Wave_slave";
define n $groupsize(0,b);
```
Often a protocol specification depends on the size of a group, in the above the size of the 0th instance of Wave_slaves has been bound to the variable \( n \).

A more general way of solving such a problem is to allow a comment line like

\[
\text{/* PROTOCOL(varname) */}
\]

to be placed on process creation lines, \( \text{varname} \) will contain the group number, and allow statement like

\[
\text{define n $groupsize(varname)}
\]

which returns the group size of the group spawned by the line that had the

\[
\text{/* PROTOCOL(varname) */ comment.}
\]

13.2 State Dependent Communication

As the last part of this section we briefly return to the problem stated in section 7.2. The problem was defining a protocol that depend on values stored in the program at run time. The example at hand is more clearly illustrated in Fig. 22 (the Roll part of the protocol has been left out for clarity). Depending on the program variable \( k \), a number of processes do not send anything; this set of non sending processes vary according to the row number a process is located in as well as the number \( k \).

A simple formula that determines which instance numbers should not send, given a value \( k \) and a row number is:

\[
(row + k + n - 1) \% n + row \times n
\]

where \( n \) is the group size. We could write the following protocol line

\[
\text{Matrix}[]_j(SendPipe) \rightarrow \text{Matrix}[]_j((j + 1) \% 4)(ReceivePipe) :: \forall j : j <= 0 \& \& j < n \& \& j != (row + k + n - 1) \% n + row \times n;
\]

This would require the values of the program variables \( k \) and \( row \) (\( row \) could be computed as \( n/4 \)). These can be obtained by adding lines to the program in the following way:

\[
\text{protocol_sym(row);} \\
\text{protocol_sym(k)};
\]
Figure 22: The 4 different stages of the pipe operation.

\begin{verbatim}
pvm_send(...); /* PROTOCOL(SendPipe) */
\end{verbatim}

The program values $row$ and $k$ will then be packed and sent to the protocol checking module (and added to the log files), and inserted into the symbol table $\sigma$ at check time.

A process instance number is automatically assigned to be the rank of the process within the group that it was spawned in. If this is not the instance number the user wishes to use, she can change it by a call to \texttt{protocol_instance()} with the new instance number.
References


