# A LOTOS-based Lightweight Approach to Formally Verify MPI Applications

Nelson Rosa
Universidade Federal de Pernambuco
Centro de Informática
Recife, PE, Brazil
E-mail: nsr@cin.ufpe.br

Humaira Kamal
University of British Columbia
Department of Computer Science
Vancouver, BC, Canada
E-mail: humaira@cs.ubc.ca

Alan Wagner
University of British Columbia
Department of Computer Science
Vancouver, BC, Canada
E-mail: wagner@cs.ubc.ca

***Abstract*—A significant number of parallel applications are implemented using MPI (Message Passing Interface) and several existing approaches focus on their verification. However, these approaches usually work with complete applications and to fix any undesired behaviour is very time consuming as the application is already completely implemented. To address this problem, we present a lightweight formal approach that helps developers to build safety MPI applications since the early stages of their development. The proposed approach consists of a development environment that hides formal aspects from developers, allows the verification of properties like deadlock freedom and generates partial skeletons of the implementation. The proposed approach is evaluated considering its ability and performance in detecting deadlocks.**

***Index Terms*—Formal Methods, Parallel Applications, MPI.**

## INTRODUCTION

Nowadays, a very significant group of parallel programs are implemented using MPI (Message Passing Interface) [4]. MPI was specially designed to help in the development of efficient and portable parallel applications and, as mentioned in [17], has been virtually adopted in all scientific areas demanding parallel processing.

Despite of its popularity, the development of safe applications using MPI is a complex task as they have potentially a large number of errors related to the use of invalid arguments, wrong use of MPI resources, and messaging deadlock [3].

A goal of MPI was to provide a standardized access to communication hardware and MPI does not impose many restrictions to enforce the correct behaviour. As a result even relatively simple applications can be challenging to understand and explore the behaviour of parallel MPI programs.

There is a considerable number of approaches to verify MPI applications. These approaches can be organized into three main groups: purely formal approaches [2], [7], [11], [15], runtime based approaches [9], [13], [19] and hybrid approaches [18], [21]. However, all these approaches typically work when the application is already fully implemented. In these cases, to fix any undesired behaviour is a more time-consuming task.

In this paper, we proposed a lightweight formal strategy that helps MPI application developers since the early stages of the development. The proposed approach consists of a set of steps that guides and helps the development of MPI applications, a formal model used to specify these applications, and the LFD-MPI tooling (Lightweight Formal Development in MPI) that supports all steps of the proposed process. Basic in this solution is the simplification of the MPI application development by concentrating on the use of a subset of MPI primitives that potentially leads to deadlock, and the fact that application developers are kept away from manipulating formal specifications, whilst take benefits of their use.

The unique contributions of this paper are (i) a formal lightweight development process and a tooling for rapid prototyping of MPI applications, (ii) the formal models for specifying MPI applications in LOTOS, and (iii) the strategy on working with partial MPI applications in order to avoid the problem of state-space explosion. Minor contributions are related to the definition of a domain-specific language (DSL) for partially describing MPI applications and the possibility of detecting deadlock sequences in addition to the deadlock detection itself.

This paper is organized as follows. Section I introduces basic concepts of MPI and the formal description technique adopted. Next, Section II presents the proposed approach. Section III makes an evaluation of the proposed approach by applying it to develop traditional parallel applications. Section IV presents existing approaches on verifying parallel applications. Finally, Section V presents conclusions and some future works.

## I. BACKGROUND

This section introduces two basics concepts necessary to understand the proposed approach, namely MPI and LOTOS.

### *A. MPI*

MPI (Message Passing Interface) [4] is a standard API for parallel programming. In MPI, all parallelism is explicit in the sense that application developers are responsible for correctly identifying parallelism and implementing parallel algorithms using the communication primitives defined in the API.

MPI has been widely adopted for several reasons. Firstly, it is the only existing message passing library considered a standard. Secondly, it is highly portable and source codes are migrated to different platforms with little or very small modifications. Thirdly, as an API, vendor implementations usually explore native hardware features to optimize performance. Finally, the whole API has over 440 primitives, which support almost any functionality required by parallel application developers.

The main primitives specified by MPI can be divided into three groups: environment management routines used to get and set information about the execution environment, such as *MPI_Init* and *MPI_Finalize*; point-to-point communication routines used to exchange messages between two processes and subdivided into blocking (e.g., *MPI_Send* and *MPI_Recv*) and non-blocking (e.g., *MPI_Irecv*, *MPI_Wait*) primitives; and collective communication routines used to collective communication inside of a group, e.g., *MPI_Barrier*. Additional routine groups include process group routines, communicators routine, derived data types routines, virtual topology routines and other miscellaneous routines.

By using these routines, a typical MPI application has the following general and intuitive structure:

```
#include "mpi.h"
...
// declarations and prototypes
...
// Serial code
...
MPI_Init(...); // begin or parallel code
        ...
        // Message passing calls and implementation
        of the application functionality
        ...
MPI_Finalize(); // end of parallel code
...
// Serial code
```

## B. LOTOS

A LOTOS (Language Of Temporal Ordering Specification) [1], [12] specification describes a system through a hierarchy of active components or processes. A process is an entity able to realise non observable internal actions and to interact with other processes through externally observable actions.

The unit of atomic interaction among processes is called an event. Events correspond to a synchronous communication that may occur among processes able to interact with one another. Events are atomic, in the sense that they happen instantaneously and are not time consuming. The point where an event interaction occurs is known as a port. Such event may or may not actually involve the exchange of values. A non-observable action is referred to as an internal action or internal event. A process has a finite set of ports that can be shared.

An essential component of a specification or process definition is its behaviour expression. A behaviour expression is built by applying an operator (e.g., parallel operator "||") to other behaviour expressions. A behaviour expression may also include instantiations of other processes, whose definitions are provided in the "where" clause following the expression [1]. Next, we present the LOTOS specification of a simple client-server system:

```
1: specification ClientServer[request,reply] : noexit
2:     behaviour
3:         Client[request,reply] || Server[request,reply]
4: where
5:     process Client[request,reply] : noexit :=
6:         request; reply; Client[request, reply]
7:     endproc
8:     process Server[request,reply] :noexit:=
9:         hide processRequest in
10:            request;
11:            processRequest;
12:            reply;
13:            Server [request, reply]
14:    endproc
15: endspec
```

The top-level specification (3) is a parallel composition (operator "||") of the processes *Client* and *Server*, i.e., every action externally observable executed by the process *Client* must be synchronised to the process *Server*. The process *Client* (5) performs two actions, namely *request* and *reply* (6), and then reinstantiates. The action-prefix operator (';') defines the temporal ordering of the actions request and reply (the action request occurs before the action reply) in the Client. Informally, the *Server* (8) receives a request (10), processes it (11) and then sends a reply (12) to the process *Client*.

## II. Lightweight Formal Development

The proposed approach consists of (i) an MPI application development process, (ii) a formal model to specify MPI applications in the development process, and (iii) tools to support all phases of the proposed process. Using the tools developers first provide an initial description of the MPI application structure, then enrich the description with communication annotations, whereupon it is then translated into a formal specification, which verifies formal properties and then generates a partial code skeleton of the application which can then to expanded on.

The proposed approach has been developed following some basic principles:

- *Focus on development time*: MPI applications are verified at development time as it can minimize the impact of fixing possible undesired behaviours, like deadlock, before the application is executed;
- *Formalism agnose*: The proposed solution is decoupled from a particular formal description technique, whilst it is based on the use of process algebras;
- *Focus on communication*: While complete MPI applications include several imperative commands, what is considered in the proposed approach is the set of MPI primitives that potentially leads to deadlock; and
- *Lighweight*: While the formalisation is the basis for the proposed approach, application developers are shielded from the complexity of directly manipulating formal specifications.

By adopting these principles, next sections present details of the proposed development process, the formalisation strategy and the implemented tools.

### A. Development Process

The proposed process for building MPI applications consists of four main phases as illustrated in Figure 1: *Structural*

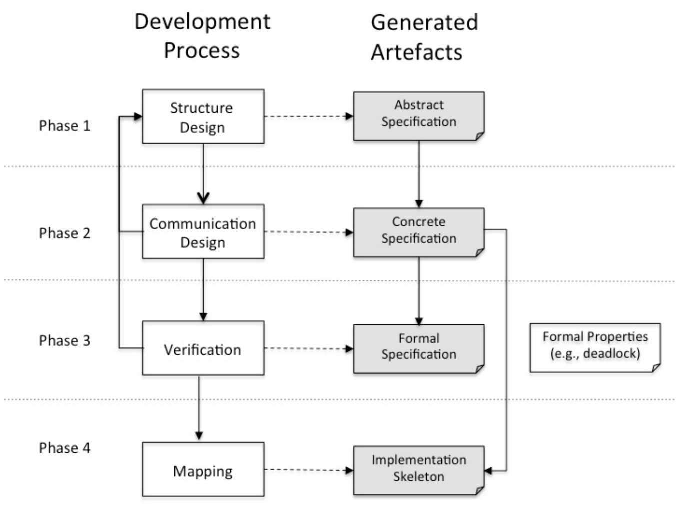


Fig. 1. General overview of the development process

*Design*, *Communication Design*, *Verification* and *Mapping*. In phase *Structural Design*, application developers define the components that make up the parallel application. Next, in phase *Communication Design*, application developers must define the communication primitives used by the components to interact with each other. In the third phase, namely *Verification*, the application is verified against some desired properties, e.g., deadlock freedom. In the case the properties are not satisfied, it is necessary to promote changes in structural and/or communication specifications. Finally, phase *Mapping* generates an artifact that serves as an initial step for the actual implementation.

As usual, each phase of the development process has input and output artifacts that are generated manually or automatically. The first artifact, namely *Abstract Specification*, describes the structure of the MPI application and is defined in a simple DSL (Domain Specification Language). This initial specification includes the set of processes that compose the application and serves as basis for the *Concrete Specification*. The *Concrete Specification* is obtained by enriching the initial specification with MPI primitives responsible for expressing the communication between the application's components. Still informal, this specification includes several processes (one for each application's component) and an ordered set of MPI primitives invoked by each process. The *Concrete Specification* is mapped into a *Formal Specification* described in LOTOS. The formal specification is checked against desired behavioural properties (*Formal Properties*). Finally, the *Concrete Specification* serves as input to the generation of the *Implementation Skeleton*, which is a semi-implementation of the MPI application.

## B. Formal Specification

While the *Abstract*, *Concrete* and *Implementation Skeletons* are informal representations of the MPI application, the *Formal Specification*, as the name suggests, is described in a formal description technique. LOTOS (see Section I-B) has been adopted to formally specify the behaviour and structure of MPI applications. LOTOS has a good tooling support that enables us to simulate and verify the resulting formal specification, it has a powerful expressiveness and has been widely adopted to specify distributed and parallel systems. In adition to the adoption of LOTOS, only a subset of MPI primitives has been considered: MPI_Send, MPI_Recv, MP_IRecv, MPI_Wait, MPI_Barrier, MPI_Init and MPI_Finalize. It is worth observing that the complete MPI API consists of more than 400 operations. However, the primitives being considerd are the ones more commonly adopted in the majority of MPI applications.

The *Formal Specification* is defined as follows:

*1) Definition 1 (Formal Specification).:* The formal specification of an application is $S_F = B$, where $B$ is a LOTOS behaviour expression defined as $B = (P_0|||...|||P_n)||M$, where

- $P_i$ is a process obtained by mapping a concrete process into a LOTOS process.
- $M$ is the LOTOS process that specifies the MPI middleware behaviour.

$P_i$ has a behaviour obtained by mapping each individual action of the concrete process into its respective LOTOS action:

| **Concrete Process** | | **LOTOS Process** |
|---|---|---|
| Rank (id) | | **Process** Rank [g] (id): **noexit** := |
| $m_1$ (id,$p$) | | g!id !$m_1$(id,$p$); |
| $m_2$ (id,$p$) | ⇒ | g!id !$m_2$(id,$p$); |
| ... | | ... |
| $m_n$ (id,$p$) | | g!id !$m_n$(id,$p$) |
| | | **endproc** |

The LOTOS action is defined as $g!id!m_i(id_i, p_1)$, where $g$ is a LOTOS gate, $id$ is the identification of the process that executes the action, $m_i$ and $p_1$ are the MPI primitive being invoked and its parameter, respectively. By adopting Definition 1, a simple MPI application is formally defined in LOTOS as follows:

```
1:  Specification SendRecv [g] : noexit
2:    (* types and variables definitions *)
3:  behaviour
4:    (Rank0 [g] (0) ||| Rank1 [g] (1))
5:       ||
6:    MPI_Middleware [g] (inv, nRank)
7:  where
8:    process Rank0 [g] (id:Nat) : noexit :=
9:       g !id !MPI_Send (id, 1);
10:      g !id !MPI_Recv (id, 1);
11:      Rank0 [g] (id)
12:   endproc
13:   process Rank1 [g] (id:Nat) : noexit :=
14:      g !id !MPI_Recv (id, 0);
15:      g !id !MPI_Send (id, 0);
16:      Rank1 [g] (id)
17:   endproc
18:   (* middleware specification *)
19: endspec
```

This application consists of two processes, namely *Rank0*(8) and *Rank1*(13), where *Rank0* and *Rank1* send and receive messages synchronously. In this example, *Rank0* first sends a message to *Rank1* (9) and then becomes ready to receive a message from *Rank1* (10), whilst *Rank1* first receives a message from *Rank0* (14) and then sends a message to *Rank0* (15).

The *MPI_Middleware* (instantiated in Line 6) is responsible for defining the order the MPI primitives invoked in each process can be actually executed. In practical terms, it coordinates the interactions between the MPI processes by receiving an invocation and deciding what happens next in the behaviour of the application. This ordering is defined based on the semantics of the MPI primitives. The MPI middleware behaviour is defined in LOTOS as follows:

```
1: process MPI_Middleware [g] (lInv:INVOCATION, nRank:Nat) : noexit :=
2:    g ?id:Nat ?inv:INVOCATION [Enabled (inv, lInv, nRank)];
3:      MPI_Middleware [g] (inv, nRank)
4: endproc
```

This middleware specification has a key element, function $Enabled$, that decides whether an MPI primitive of a particular process can be executed or not. This function is used in predicate $[Enabled(inv, lInv, nRank)]$ (Line 2) that enables the execution of the primitive ($inv$) when it is evaluated to *true*. In practice, this function serves as a *hook* to enforce the MPI semantics, whilst it in some way overrides the semantics of the parallel LOTOS operators (||).

In order to better illustrate the role of function *Enabled*, the following traces have been generated when this function is in action (*MPI Semantics*) or not in action (*LOTOS Semantics*) in the previous example:

| Line | MPI semantics | LOTOS Semantics |
|---|---|---|
| 1 | G !0 !MPI_Send(0,1) | G !0 !MPI_Send(0,1) |
| 2 | G !1 !MPI_Recv(1,0) | G !0 !MPI_Recv(0,1) |
| 3 | G !1 !MPI_Send(1,0) | G !1 !MPI_Send(1,0) |
| 4 | G !0 !MPI_Recv(0,1) | G !1 !MPI_Recv(1,0) |

This LOTOS trace (second column) is not valid according to the MPI semantics. The action in the second line ($G!0!MPI\_Recv(0, 1)$) should not be possible in this example until the execution of fourth line ($G!1!MPI\_Recv(1, 0)$).

Function *Enabled* defines which primitives are allowed to be executed next by the process based on: the primitives already invoked but that still waiting to be executed (*Lists*); the MPI primitive just invoked by the process (*invMPI*); the candidate MPI primitives that are enabled according to the LOTOS semantics (*invFDT*); and the MPI semantics ($invMPI\$Semantics(Lists,\ invMPI,\ invFDT[i])$).

```
1: Enabled (Lists,invMPI,invFDT)
2:     nProc ← number of MPI processes
3:     for i ← 1 to nProc
4:         rsp [i] ← invMPI$Semantics(Lists,invMPI,invFDT[i])
5:     end for
6: return rsp
```

This function is invoked every time a MPI process interacts with the *MPI_Middleware*. In its turn, the middleware maintains internally a set of execution lists (*Lists*), one for each application process.

The semantics of the MPI primitives aforementioned are enforced in the LOTOS specification using the functions described in the following. All functions have a similar structure in the sense that they first update the execution list and then decide if the primitive enabled by the LOTOS semantics is also enabled by the MPI semantics.

At this point, it is worth observing that the semantics of each MPI primitive have been defined only considering a subset of the actual primitive's parameters. For example, primitive *MPI_Send* has six parameters while we only consider two. By abstracting some of these parameters was essential to initially deal with the complexity of these primitives.

***InitSemantics and FinalizeSemantics*** Due to lack of space and their simplicity, the semantics of *MPI_Init* and *MPI_Finalized* are not fully described here. In practice, $MPI\_Init$ initializes all execution lists and allows the execution of the first primitives of each process, and *MPI_Finalize* terminates the MPI execution environment and all invocations performed by the application are disabled.

***SendSemantics*** This function enforces the semantics of MPI_Send (src, dst). When this primitive is invoked, process *src* performs a synchronous send to process *dst* and becomes blocked until a synchronization happens. The invoked MPI primitive (*invMPI*) is initially stored in the execution list of process *src* (2). Next, it is necessary to check whether a synchronization is possible with process *dst* (4). This synchronization is only possible, however, if process *dst* has some primitive already stored in its execution list (3). The decision of enabling or not the LOTOS primitive (*invFDT*) is based on the size of the execution list of process *scr* (6-10). If the list is empty (6), i.e., process *src* has not pending primitives to be completed, the candidate MPI primitive initially enabled by LOTOS semantics (semantics of parallel operator) is also enabled by the MPI semantics (7). Otherwise, if the process has a pending MPI primitive to be executed, it needs to wait for its completion and the MPI primitive enabled by the LOTOS semantics is not allowed to be executed according to MPI semantics (9).

```
1:  SendSemantics (Lists,invMPI,invFDT)
2:      addList(Lists[invMPI→src],invMPI)
3:      if not listEmpty(Lists[invMPI→dst])
4:          sync ← CheckSynchronization(Lists[invMPI→src],Lists[invMPI→dst])
5:      end if
6:      if listEmpty(Lists[invFDT→src])
7:          rsp ← true
8:      else
9:          rsp ← false
10:     end if
11: return rsp
```

Before presenting the remaining semantics functions, it is necessary to describe how function *CheckSynchronization* works. This function has two input parameters (execution lists) used to check whether the synchronization is possible or not. Depending on this check, both execution lists $lst_1$ and $lst_2$ are updated.

```
1:  CheckSynchronization (lst$_1$,lst$_2$)
2:      head$_1$ ← listHead(lst$_1$)
3:      head$_2$ ← listHead(lst$_2$)
4:      sync ← false
5:      ...
6:      if head$_1$ →op = MPI_RECV
7:          if head$_2$ →op = MPI_SEND
8:                  and (head$_2$ →src = head1→dst or head1→dst = ANY)
9:                  and head$_2$ →dst = head1→src
10:             sync ← true
11:             listRemove(lst$_1$)
12:             listRemove(lst$_2$)
13:         end if
14:     end if
```

```
15:     ...
16:  return sync
```

In order to check the possibility of synchronization, the head of both execution lists (2-3) are the elements to be actually considered as they represent the primitives that were first invoked and whose execution is not completed. For simplicity and lack of space, only the part of the function associated to primitive $MPI\_RECV$ is shown[1]. In this case, the synchronization happens when the heads have two different combinations of primitives (8-9): $MPI\_RECV(dst, src)$ and $MPI\_SEND(src, dst)$, or $MPI\_RECV(src, ANY)$ and $MPI\_SEND(src, dst)$. In this case, both primitives are removed from the execution lists indicating their synchronization and completion (11-12).

***RecvSemantics*** This function is associated to the semantics of MPI_Recv(src, dst). In the invocation of this primitive, process *src* performs a synchronous receive and becomes blocked until a synchronization occurs. Similarly to *MPI_Send*, the primitive is stored in the execution list of process *src* (3) and the syncrhonization is checked. However, due to the possibility of using the wildcard *ANY* in this primitive (4), the syncronization checking follows two different ways. On one hand, in the case wildcard *ANY* is used (4-12), all execution lists have to be checked for the synchronization. When the first synchronization occurs, the checking process stops. On the other hand, the synchronization process is similar to *SendSemantics*(13-16). In both cases, the decision on enabling or not the LOTOS primitive is also based on the size of execution lists (18-22).

```
1:   RecvSemantics (Lists,invMPI,invFDT)
2:     nProc = number of MPI processes
3:     addList(Lists[invMPI→src],invMPI)
4:     if invMPI→dst = ANY
5:       idx ← 0
6:       sync ← false
7:       while idx < nProc and not sync
8:         if not listEmpty(Lists[idx])
9:           sync ← CheckSynchronization(Lists[invMPI→src],Lists[invMPI→idx])
10:        end if
11:        idx ← idx + 1
12:      end while
13:    else
14:      if not listEmpty(Lists[invMPI→dst])
15:        sync ← CheckSynchronyzation(Lists[invMPI→src],Lists[invMPI→dst])
16:      end if
17:    end if
18:    if listEmpty(Lists[invFDT→src])
19:      rsp ← true
20:    else
21:      rsp ← false
22:    end if
23:  return rsp
```

***IrecvSemantics*** This function enforces the semantics of MPI_IRecv (src, dst) and it begins a nonblocking receive from *dst*. This function is very similar to *RecvSemantics*, but it has a key difference in the decision of enabling or not the LOTOS primitive (*invFDT*). As *MPI_Irecv* is not blocking, *invFDT* is enabled even when the execution list of process *src* is not empty (4). In practice, process *src* does not become blocked after executing *MPI_Irecv*.

```
1:   IrecvSemantics (Lists,invMPI,invFDT)
2:     (* Similar to Lines 1-16 of Function RecvSemantics*)
3:     tempHead ← listHead(Lists[invFDT→src])
4:     if listEmpty(Lists[invFDT→src]) or tempHead→op = MPI_IRECV
5:       rsp ← true
6:     else
7:       rsp ← false
8:     end if
9:   return rsp
```

***BarrierSemantics*** This function is associated to the semantics of primitive MPI_Barrier that blocks until all processes have reached this routine. This function essentially checks if all processes have invoked this routine (5-10) and, when it occurs, the execution lists of all processes are updated (11-13). The processes that already invoked this primitive still blocked waiting for the others also invoke *MPI_Barrier*.

```
1:   BarrierSemantics (Lists,invMPI,invFDT)
2:     nProc ← number of MPI processes
3:     addList(Lists[invMPI→src],invMPI)
4:     allBarrier ← true
5:     for idx ← 0 to nProc - 1
6:       tempHead ← listHead(Lists[idx])
7:       if (tempHead→op not MPI_BARRIER)
8:         allBarrier ← false
9:       end if
10:    end for
11:
12:    ifallBarrier
13:      for idx ← 0 to nProc - 1
14:        listRemove(Lists[idx])
15:      end for
16:    end if
17:
18:    if listEmpty(Lists[invFDT→src])
19:      rsp ← true
20:    else
21:      rsp ← false
22:    end if
23:  return rsp
```

***WaitSemantics*** This function enforces the semantics of primitive MPI_Wait. It waits for an MPI send or receive to complete. *MPI_Wait* blocks the process if it has a MPI_Send, MPI_Recv or MPI_Irecv pending to be executed (2-4). In this case, the process waits for the completion of these operations.

```
1:  WaitSemantics (Lists,invMPI,invFDT)
2:    if listOperationExist(Lists[invMPI→src],MPI_SEND)
3:      or listOperationExist(Lists[invMPI→src],MPI_RECV)
4:      or listOperationExist(Lists[invMPI→src],MPI_IRECV)
5:        listAdd(Lists[invMPI→src],invMPI)
6:    end if
7:    if listEmpty(Lists[invFDT→src])
8:      rsp ← true
9:    else
10:     rsp ← false
11:   end if
12: return rsp
```

## C. Tool Support

As previously mentioned, the entire program development process presented in Section II-A is supported by tool we developed, which we have called LFD-MPI (Lightweight Formal Development in MPI). LFD-MPI includes a *Graphical Editor*, two notation mappers (*C2F* and *F2I*), and a *Formal*

[1]The implementation of all functions in Language C can be accessed at http://gfads.cin.ufpe.br/mpi

*Adapter*. The graphical editor helps application developers in phases *Structural Design* and *Communication Design*, whilst the mappers generates the *Formal Specification* and *Implementation Skeleton*.

The *Formal Adapter* interacts with a formal existing tooling, to verify the formal specification (phase *Verification*). In the current implementation, the formal adapter interacts with the CADP Tooling [5]. The *Abstract Specification* and *Concrete Specification* are modelled in a simple XML-based DSL (Domain Specification Language), the *Formal Specification* is a LOTOS document, and the *Implementation Skeleton* is a semi-implementation of the MPI application written in language C. It is worth observing that the proposed solution is decoupled from a particular formal description technique, which means that adapters and mappers need to be implemented to each supported formalism. As aforementioned, current implementation have mappers and adapters with support to LOTOS.

Figure 2 shows a screenshot of the Graphical Editor. The edition are is divided into two main parts that shows the input and output artifacts. These artifacts were introduced in the development process shown in Figure 1: the input artifacts are those manually inserted by the application developers and are related to the abstract specification (upper left) and concrete specification (upper right). Meanwhile, the output artifacts (bottom) are ones automatically generated by the LFD-MPI and whose contents are the Formal Specification and the Implementation Skeleton.

## III. Evaluation

In order to evaluate the proposed approach, we initially developed four MPI applications whose existence of dealock is known a priori: Recv-Recv, Send-Send, Sched-Dep, and Any. These applications[2], as proposed in [8], were developed using the proposed tooling and then checked in order to evaluate the ability of our solution to detect the presence of deadlock.

| Approach | Recv-Recv | Send-Send | Sched-Dep | Any |
|---|---|---|---|---|
| Marmot and MPI-Check | Yes | No | Run | - |
| ISP | Yes | Yes | Yes | Yes |
| DAMPI | Yes | No | Yes | - |
| MUST | Yes | Yes | Run | - |
| LFD-MPI | Yes | Yes | Yes | Yes |

TABLE I
Deadlock detection by existing tools (original table shown in [8])

Table III shows the results considering whether the approach is able or not to detect deadlock. It is worth observing that this is an extension of the comparison originally presented in [8] that includes the *Any* application and LFD-MPI. As *Any* was not present in the original table, its evaluation in some tools is set to "-" (no evaluation performed). Similarly to ISP, it is also possible to generate the sequence of MPI invocations that leads to a deadlock, e.g., "first one", "shortest one" or "all".

Another aspect that has been evaluated is the performance of the proposed solution in terms of response time to detect

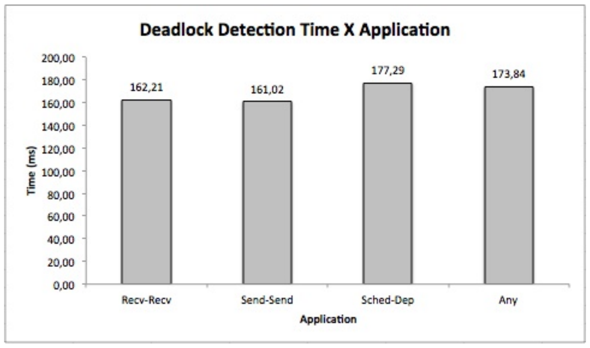


Fig. 3. Evaluation results

the deadlock of the aforementioned applications. The whole solution (LFD-MPI and CADP Tooling) run on a single Mac OS machine (Version 10.9.5), with 2.9 GHz Intel Core i7 processor and 8 GB of memory. Figure 3 shows the deadlock detection time for each application aforementioned. The response time is the time spent since the invocation of the CADP tool until the receipt of the response from it. The CADP was configured to stop and respond to invocation as soon as the first deadlock has been found. For each application, the CADP tooling was invoked 100 times and the results shown in the figure are the mean response time.

This result shows that the deadlock detection time has low variation despite the fact that, for example, the application *Any* is much more complex than *Send-Recv*. Furthermore, considering that the proposed solution is used at development time, these results make viable the execution of deadlock checks several times while the application is being developed, e.g., to each change in the application code, the deadlock can be verified with a low time cost.

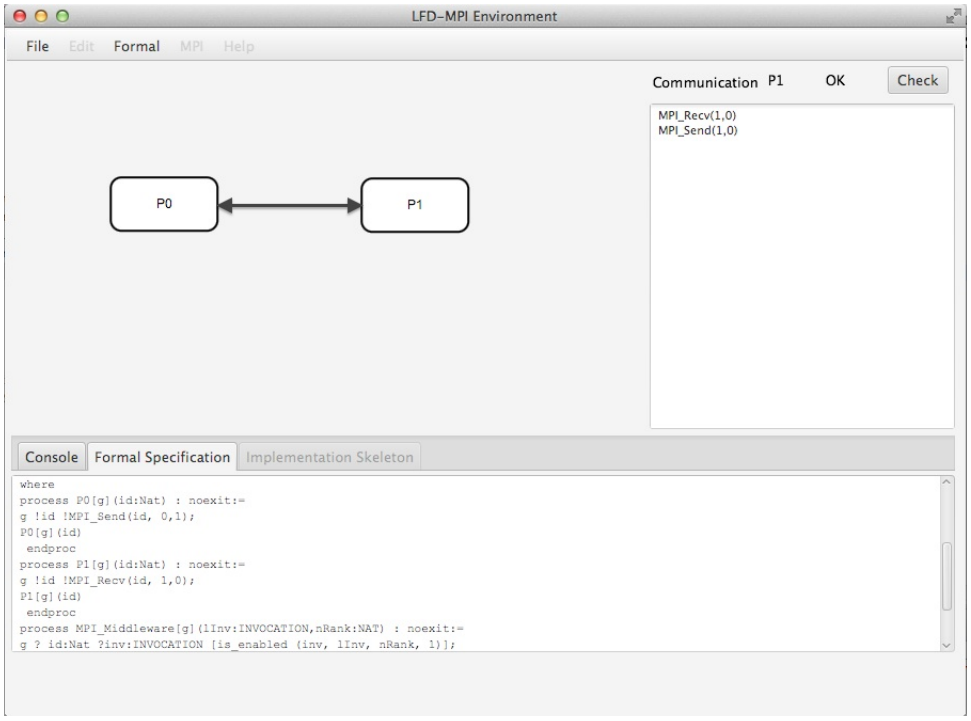
## IV. Related Work

There is a considerable number of approaches to verify MPI applications. These approaches can be organized into three main groups: purely formal approaches, runtime based approaches and hybrid approaches.

In the first group, formalisms such as CSP [2], Session types [11] and Abstract State Machines [7] are used to create formal models of MPI applications. These models serve as basis for the verification of properties like deadlock freedom, type safety and communication safety.

Yet in this group, another well-know approach for verifying MPI applications is the MPI-SPIN [14], [15]. MPI-SPIN uses model checking techniques to verify nonblocking MPI applications. An MPI application is modelled as a set of guarded transition systems (one for each process that composes the application) and a global array of communication records that models buffered messages and outstanding requests. The semantics of the execution are defined in such way that given a global state, either a process's enabled transition or an action of the infrastrcuture can be executed. This solution was implemented as an extension to SPIN [10] and incorporated into the SPIN specification language Promela.

[2]Full LOTOS specification of these applications can be found at http://gfads.cin.ufpe.br/mpi



```
MPI_Recv(1,0)
MPI_Send(1,0)
```

```
where
process P0[g](id:Nat) : noexit:=
g !id !MPI_Send(id, 0,1);
P0[g](id)
 endproc
process P1[g](id:Nat) : noexit:=
g !id !MPI_Recv(id, 1,0);
P1[g](id)
 endproc
process MPI_Middleware[g](lInv:INVOCATION,nRank:NAT) : noexit:=
g ? id:Nat ?inv:INVOCATION [is_enabled (inv, lInv, nRank, 1)];
```

Fig. 2. Screenshot of LFD-MPI

The second group essentially works by intercepting MPI calls at runtime and detecting several kinds of errors of MPI applications. MUST (Marmot Umpire Scalable Tool) [8], [9] intercepts MPI calls of all processes that make up the application and then runs a graph-based deadlock detector. The detector determines if the MPI call can complete or must wait for another communication call. Basically, MUST represents the wait-for condition as a graph and uses graph analisys to decide whether a deadlock exists at a particular step of the application execution. Marmot [13] and Umpire [19] adopt a verification strategy similar to MUST.

Finally, the third group combines in some way the previous strategies. ISP (In-situ Partial Order) [6], [21] is an tool that uses runtime model checking methods to verify MPI applications. By using ISP, application developers directly check the existence of deadlocks without any contact with the formalisms behind it. In practice, ISP provides a replacement function for every MPI primitive supported by the solution. When a replacement function (e.g., $MPI\_Send$) is invoked, a scheduler is consulted and it decides whether the function can be actually executed or not. Another tool developed in the context of ISP efforts is DAMPI (Dynamic Analyzer MPI) [20]. Similarly to ISP, DAMPI also intercepts MPI calls, but it executes a distributed scheduling algorithm.

Unlike ISP, TASS (Tookit for Accurate Scientific Software) [16] starts from a MPI program and generates an abstract model used to symbolic execution and state space enumeration. By exploring the state space, TASS is able to check properties such as absence of deadlock, buffer overflow and memory leaks.

The key difference of these approaches to what is being proposed is the fact that LFD-MPI serves mainly as a tool that helps the development of the applications instead of treating with complete applications. This fact reduces significantly the code to be analised by the verification tool. Additionally, by enforcing its role in the application development, LFD-MPI is the only that generates semi-implementations. The similarity with purely formal and hybrid approaches is the use of formal methods to help the verification process. However, application developers have not contact with the formalisms.

## V. Conclusion and Future Work

This paper presented a solution that allows the development of safe parallel applications using MPI. The proposed solution consists of a development process that uses formal methods in a lightweight and is supported by a modeling tool that avoids the contact of application developers with the formal technique being used in the backend. The proposed process also focuses on only treating with the parts of the application that have more potential to lead to deadlock, e.g., ones that use point-to-point communication and collective routines. In practice, instead of verifying the whole application at runtime, which demands the treatment of the message passing invocations and the whole application functionality, our focus is only on specific parts of the application.

While the solution is not complete as only a subset of MPI primitives is used and the semantics of these primitives have been only partially defined, it advances on three key aspects: by adopting a formal lightweight approach for rapid prototyping MPI applications, introducing a formal model in

LOTOS to specify MPI applications and defining an strategy for verifying MPI applications that works with partial implementation codes. Minor contributions refer to the automatic generation of skeletons of MPI applications and the detection of deadlock sequences in addition to deadlock detection itself.

A key next step in this research is to extend the set of MPI primitives supported by the current solution. Simultaneously, it is also necessary to enrich the semantics of the MPI primitives by considering their full set of parameters. Finally, it is also necessary to deal with scalability issues in order to allow the development of MPI applications with thousand or even millions of processes.

## References

[1] Bolognesi, T., Brinksma, E.: Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems 14, 25–59 (1987)

[2] da Costa, U.S., de Medeiros Junior, I.S., Medeiros Oliveira, M.V.: Specification and Verification of a MPI Implementation for a MP-SoC, Lecture Notes in Computer Science, vol. 6255, pp. 168–183 (2010)

[3] DeSouza, J., Kuhn, B., de Supinski, B.R., Samofalov, V., Zheltov, S., Bratanov, S.: Automated, scalable debugging of mpi programs with intel&reg; message checker. In: Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications. pp. 78–82. SE-HPCS '05, ACM, New York, NY, USA (2005), `http://doi.acm.org/10.1145/1145319.1145342`

[4] Forum, M.P.I.: MPI: A Message-Passing Interface Standard Version 3.0 (September 2012)

[5] Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. International Journal on Software Tools for Technology Transfer 15(2), 89–107 (2013), `http://dx.doi.org/10.1007/s10009-012-0244-z`

[6] Gopalakrishnan, G., Kirby, R.M., Siegel, S., Thakur, R., Gropp, W., Lusk, E., De Supinski, B.R., Schulz, M., Bronevetsky, G.: Formal Analysis of MPI-based Parallel Programs. Communications of the ACM 54(12), 82–91 (2011)

[7] Grudenic, I., Bogunovic, N.: Modeling and Verification of MPI Based Distributed Software. In: Mohr, B., Träff, J., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science, vol. 4192, pp. 123–132. Springer Berlin Heidelberg (2006)

[8] Hilbrich, T., Protze, J., Schulz, M., de Supinski, B.R., Müller, M.S.: MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 30:1–30:11. SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA (2012)

[9] Hilbrich, T., Protze, J., Schulz, M., Supinski, B.R.d., Muller, M.S.: MPI runtime error detection with MUST: Advances in deadlock detection. Scientific Programming 21(3), 109–121 (2013)

[10] Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2004)

[11] Honda, K., Marques, E.R., Martins, F., Ng, N., Vasconcelos, V.T., Yoshida, N.: Verification of MPI Programs Using Session Types. In: Träff, J., Benkner, S., Dongarra, J. (eds.) Recent Advances in the Message Passing Interface, Lecture Notes in Computer Science, vol. 7490, pp. 291–293. Springer Berlin Heidelberg (2012)

[12] ISO: Enhancements to LOTOS. ISO/IEC JTC1/SC21/WG7 (Jan 2001)

[13] Krammer, B., Hilbrich, T., Himmler, V., Czink, B., Dichev, K., Müller, M.: MPI Correctness Checking with Marmot. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.) Tools for High Performance Computing, pp. 61–78. Springer Berlin Heidelberg (2008)

[14] Siegel, S.F., Gopalakrishnan, G.: Formal Analysis of Message Passing, Lecture Notes in Computer Science, vol. 6538, pp. 2–18 (2011)

[15] Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Combining Symbolic Execution with Model Checking to Verify Parallel Numerical Programs. ACM Trans. Softw. Eng. Methodol. 17(2), 10:1–10:34 (May 2008)

[16] Siegel, S.F., Zirkel, T.K.: Automatic Formal Verification of MPI-Based Parallel Programs. ACM Sigplan Notices 46(8), 309–310 (2011)

[17] Vakkalanka, S.: Efficient Dynamic Verification Algorithms for MPI Applications. Ph.D. thesis, Salt Lake City, UT, USA (2010), aAI3413092

[18] Vakkalanka, S., Sharma, S., Gopalakrishnan, G., Kirby, R.M., Acm: Isp: A tool for model checking mpi programs. Ppopp'08: Proceedings of the 2008 Acm Sigplan Symposium on Principles and Practice of Parallel Programming pp. 285–286 (2008)

[19] Vetter, J., de Supinski, B.: Dynamic Software Testing of MPI Applications with Umpire. In: Supercomputing, ACM/IEEE 2000 Conference. pp. 51–51 (Nov 2000)

[20] Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., de Supinski, B., Schulz, M., Bronevetsky, G.: A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In: High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for. pp. 1–10 (Nov 2010)

[21] Vo, A., Vakkalanka, S., Gopalakrishnan, G.: ISP Tool Update: Scalable MPI Verification. In: Müller, M.S., Resch, M.M., Schulz, A., Nagel, W.E. (eds.) Tools for High Performance Computing 2009, pp. 175–184. Springer Berlin Heidelberg (2010)