Communicating Process Architectures 2015
K. Chalmers, J.B. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2015

© 2015 The authors and Open Channel Publishing Ltd. All rights reserved.

Towards Lightweight Formal Development of MPI Applications

Nelson S. ROSA [a,1], Alan WAGNER [b] and Humaira KAMAL [b]
[a] *Universidade Federal de Pernambuco*
Centro de Informática
Recife, PE, Brazil
[b] *University of British Columbia*
Department of Computer Science
Vancouver, BC, Canada

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 typically work with complete applications and fixing any undesired behaviour at this late stage of application development is difficult and time consuming. To address this problem, we present a lightweight formal approach that helps developers build safety into the MPI applications from the early stages of the program development. Our approach consists of a methodology that includes verification during the program development process. We provide tools that hide the more difficult formal aspects from developers making it possible to verify properties such as freedom from deadlock as well as automatically generating partial skeletons of the code. We evaluate our approach with respect to its ability and efficiency in detecting deadlocks.

Keywords. formal methods, parallel applications, MPI

Introduction

A significant number of parallel programs are implemented using MPI (Message Passing Interface) [1]. MPI was especially designed to help in the development of efficient and portable parallel applications and, as mentioned in [2], has been widely adopted in all scientific areas that rely on parallel computing. Despite its popularity, the development of safe applications using MPI is a complex task because of the many activities happening in parallel and the need to coordinate these activities through the passing of messages. Although MPI provides a standardised higher-level access to the underlying communication hardware, MPI imposes few restrictions to enforce correct messaging behaviour with regards to the use of the library. There are potential problems caused by incorrect arguments resulting in unmatched messages or incorrectly received messages that can lead, and usually not immediately, to incorrect behaviour. As a result, the behaviour of even relatively simple parallel MPI applications can be challenging to understand and fully explore.

[1]Corresponding Author: *Nelson S. Rosa, Av. Jornalista Anibal Fernandes, s/n - Cidade Universitária 50.740-560 - Recife, PE, Brazil*. Tel.: +55 81 2126 8430; Fax: +55 81 2126 8438; E-mail: `nsr@cin.ufpe.br`.

Due to these challenges there have been many attempts to verify MPI applications. Previous approaches can be classified into three main groups: purely formal approaches [3,4,5,6], runtime based approaches [7,8,9] and hybrid approaches [10,11]. However, all these approaches typically work when the application is already fully implemented. In these cases, fixing any undesired behaviour is difficult because of the possibility of many compounding errors in a large parallel code base.

In this paper, we propose a lightweight formal development methodology that incorporates verification into the early stages of the MPI code development process. Our approach consists of (1) a development methodology that guides and helps the development of MPI applications, and adopts software architecture principles, (2) a formal model used to specify these applications, and (3) the LFD-MPI tool (Lightweight Formal Development in MPI) that supports all steps of the proposed process. Fundamental to this solution is the simplification of the MPI application development by identifying application patterns known to have potential to deadlock, and shielding the application developer from the formal techniques used to verify correctness properties.

The unique contributions of this paper are (i) a formal lightweight development process and tools for rapid prototyping of MPI applications and (ii) the formal models for specifying MPI applications in CSP (Communicating Sequential Processes)[12] and (iii) the definition of an architecture description language (ADL) for describing MPI applications' software architectures. The use of software architecture principles through the proposed ADL also creates a new way of structuring parallel applications by explicitly organising their structure using well defined and widely adopted concepts of components and connectors [13].

This paper is organised as follows. Section 1 gives a short overview of MPI. Our lightweight development approach is discussed in Section 2. Section 3 makes an initial evaluation of our approach by applying it to MPI application patterns. Section 4 presents existing approaches to verifying parallel applications. Finally, Section 5 presents conclusions and future work.

1. Message Passing Interface

The Message Passing Interface (MPI) [1] has been highly successful for high performance computing for over the last two decades. It is considered essential for writing scalable scientific code and libraries that can run on a wide range of platforms [14,15]. There are a number of factors that have led to its success; these include support for software libraries, portability, and dependable performance on a wide range of network fabrics. An ecosystem developed by the MPI Forum comprising universities, vendors, national laboratories and software developers ensures active maintenance and support of the MPI standard.

The MPI standard provides a rich set of routines for message passing. It includes routines for point to point communications, collective operations, creation of communicators (for safe communication among groups of processes), to name a few. All communication in MPI is explicitly specified in the application and there must be a corresponding receive operation for every message sent. The matching criteria, within the MPI middleware, for send and receive operations is based on the following three parameters:

- The process *rank* is a unique identifying integer, assigned by MPI in the range `0` to `n-1`, where `n` is the total number of MPI processes launched at the initialisation of the application. To send a message to another process, the sender process needs to know its target's rank.

- The message *tag* is a user defined integer that is used to identify a message. All messages sent by one process to another with the same tag must be received in the order in which they were sent.
- The *context identifier* is a unique value associated with a communicator. A communicator defines the scope of the communication and is associated with a group of processes. Communication within one communicator cannot interfere with communication inside another communicator. Each of the processes in a communicator is assigned a rank in the range `0` to `size-1`, where `size` is the number of the processes associated with the communicator[1].

The send operation must specify a unique receiver through target's rank, tag and communicator. A receive operation may specify a wildcard `MPI_ANY_SOURCE` for sender rank, and/or a wildcard `MPI_ANY_TAG` for message tag indicating any sender and/or tag value are acceptable. The receiver, however, cannot specify a wildcard value for the communicator.

In addition to the above three parameters for matching, which are related to message progression inside the MPI middleware, there are conditions for message data type and size that must be met as well. The message data type in the send operation has to match the type specified by the receive operation. For example, if the send operation uses `MPI_INT`, then the receive operation must specify that as well, otherwise the program would be erroneous[2]. The message size (number of elements of the specified type) in the receive operation must be at least as large as that in the corresponding send operation[3].

At program initialisation, a pre-defined communicator called `MPI_COMM_WORLD` is created which contains all the `n` processes. MPI applications typically start with `MPI_Init` that initialises the MPI execution environment and end with a call to `MPI_Finalize`, which terminates it.

2. Lightweight Formal Development

Our approach consists of (i) an MPI application development process, (ii) formal models used in the development process to specify MPI applications, and (iii) tools to support all phases of the proposed process. Using the tools, developers first provide an initial description of the application structure and then enrich it with behavioural annotations. Next, this description is used both to generate a formal specification and produce a semi-implementation. The formal specification is used to verify properties like deadlock freedom.

The basic principles of our development process are as follows.

- *Focus on development time*: MPI applications should be verified at development time as it can minimise the impact of fixing possible undesired behaviours, like deadlock, before the application is tested;
- *Adoption of Software Architecture principles*: MPI applications are described using software architecture abstractions [13] such that the parallel application is structured

[1] We are limiting the discussion in this paper to MPI intra-communicators.

[2] There are exceptions related to this rule for `MPI_PACKED`. The data type mismatch error depends on individual cases. For example, `MPI_INT` in the send operation and `MPI_CHAR` in the corresponding receive operation may result in a `MPI_ERR_TRUNCATE` (message truncated on receive) error.

[3] If the message size in the receive operation is smaller than the message size in the corresponding send operation then it will report `MPI_ERR_TRUNCATE` error. It is not considered erroneous if the receive operation specifies a message size that is larger than the send message size.

as a set of components (computation elements) wired through connectors (communication elements);
- *Focus on communication primitives*: We consider a subset of MPI communication primitives in our approach that can potentially lead to a deadlock. As mentioned in [16], mismatches between senders and receivers rank high among the most common MPI errors. We focus on blocking and non-blocking MPI send and receive operations to demonstrate our approach.
- *Lightweight*: While the formalisation plays a key role in the process, application developers should be shielded from the complexity of directly manipulating formal specifications.

By adopting these principles, the next sub-sections present details of our development process, the formalisation strategy and the tools.

2.1. Development Process

The process for building MPI applications has five key activities, shown in Figure 1: *Structural Design*, *Behavioural Design*, *Formal Modelling*, *Verification* and *Code Generation*.

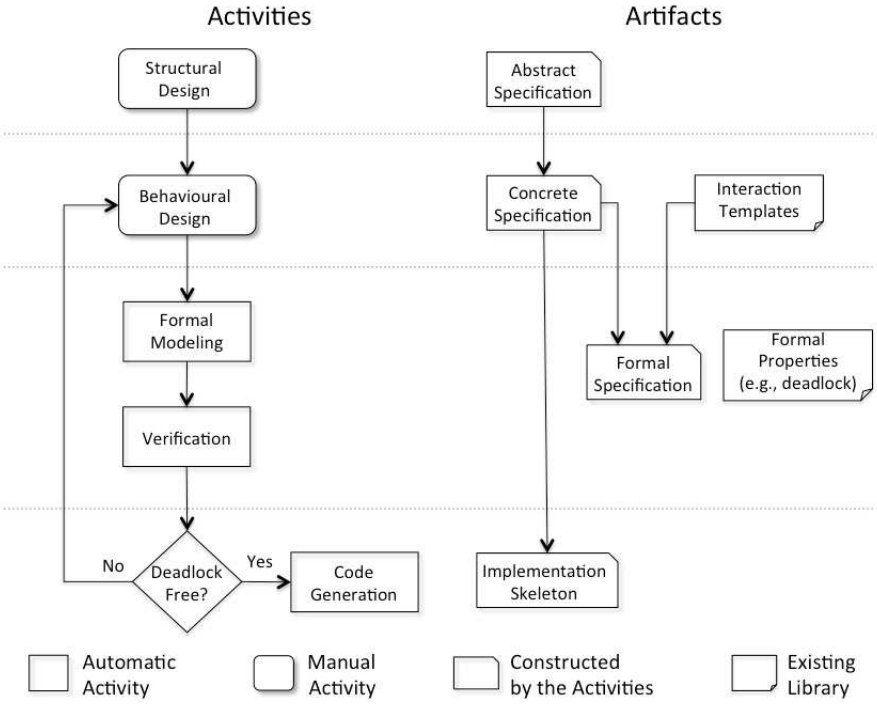

Figure 1. LFD-MPI development process.

The process starts with the activity *Structural Design* in which application developers define the components that make up the parallel application and how they are connected.

Next, in *Behavioural Design*, developers enrich the structural specification with communication primitives used by the components to interact with each other, including specifying patterns of communication such as Request-Reply, Send-Receive or publish/subscribe.

In the third activity, *Formal Modelling*, a formal model of the application is generated and then verified (*Verification*) against desired properties, e.g., deadlock freedom. If properties are not satisfied, it is necessary to promote changes in the behavioural design. Finally, the activity *Code Generation* generates an artifact that serves as a partial implementation.

As usual, each activity of the development process has input and output artifacts generated manually or automatically. The first artifact, namely *Abstract Specification*, describes the structure of the MPI application and is described by using the proposed ADL (Architecture Description Language) by the application developer. This initial specification includes the set of components that compose the application and serves as the basis for the *Concrete Specification*.

The *Concrete Specification* is obtained by enriching the *Abstract Specification* with interaction patterns responsible for expressing the communication between the application's components. Still informal, this specification defines the temporal ordering of actions executed by the components and the interaction patterns associated to the connectors, which are passive elements.

At this point, it is important to observe that the actions carried out by the components are specific to each application (usually not reusable), whilst the actions associated with the connectors are usually selected from a library of *Interaction Templates*. In practice, connectors are more than simple channels, in that they will enforce pre-defined patterns of communication across them. Finally, we also defined a catalogue of *Formal Properties* used in the verification.

From the *Concrete Specification*, two additional artifacts are generated: the *Formal Specification* and the *Implementation Skeleton*. The specification is checked against desired behavioural properties (*Formal Properties*), whilst the skeleton is a partial implementation of the MPI application.

2.2. Structural Design

As mentioned earlier, this activity consists of describing the components that make up the application and how they are wired through connectors. In practice, application developers must explicitly decompose the application into a set of elements that perform the computation (components) and elements that define how the interactions between the components are carried out (connectors). Finally, these elements are composed together into a configuration that specifies the complete structure of the application.

The output of this activity is an *Abstract Specification* expressed in ArcMPI, our architecture description language. The key characteristics of ArcMPIare as follows: (a) components (computation elements), connectors (communication elements) and configuration (composition of components and connectors) are first-class elements of the language, (b) components can be composed to make up new components (composite components), (c) there is always a connector between any two components, i.e., interacting components are never connected directly, (d) components and connectors have interaction points (ports) through which they communicate with the external world, and (e) the description of components and connectors includes their behaviour in addition to their structure.

The role of the connectors is to decouple the communication from the business logic by encapsulating the communication protocol between the components. Connectors do not initiate actions by themselves but carry out actions in response to external invocations. In the current version, the connectors act as simple channels to enable components to communicate without sight of each other's ports.

Ports in ArcMPI are currently untyped and bi-directional since most communication is 2-way. Typed ports are a desirable addition for the future. An initial set of basic connectors are provided together with the language, while software architects have the flexibility to define new ones according to their needs.

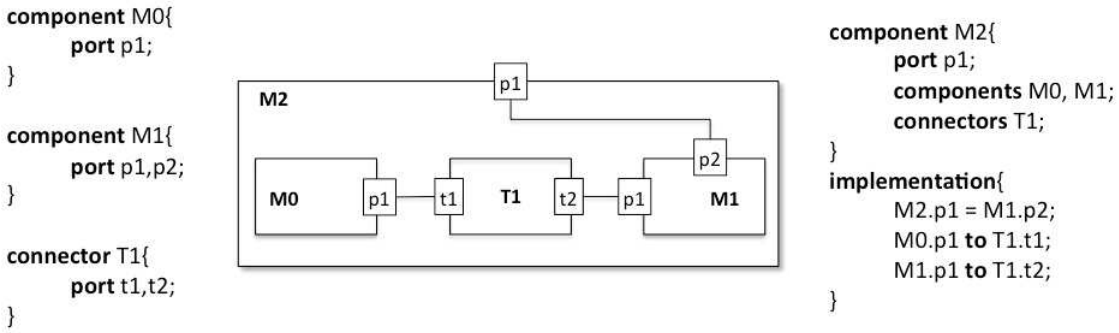

Figure 2. Abstract specification.

Figure 2 illustrates a simple *Abstract Specification* written in ArcMPI. This specification consists of three components (*M0*, *M1* and *M2*) and a connector (*T1*). *M2* is a composite component made up by the composition of *M0* and *M1* and whose interaction is realised through *T1*. As a component, *M2* can also be composed with other components. It is worth observing that *M2* externalises port *p2* of *M1* in such way that components that interact with *M2* actually communicate with *M1*.

2.3. Behavioural Design

In this activity, components and connectors introduced during the Structural Design are enriched with behavioural descriptions that define how they interact. A key part of this interaction is the pattern of message-flows between them.

Due to their impact on the way the components themselves are built [17], the behaviour of the connectors is specified first.

The behaviour of connectors in ArcMPI is defined through patterns that explicitly constrain the way components interact, e.g., *pipe-filter*, *send-receive (MPI)*, *request-reply*, *multicast*, *publish/subscribe*, and so on. For example, in the request-reply pattern a component (e.g., a client) invokes an operation (request) to a remote component (e.g., a server), which processes the request and responds (reply) to the client with the result of the operation. In the send-receive interaction, a send operation must have a matching receive operation and the send blocks until the message is stored on the receiver side. ArcMPI provides a set of pre-defined interaction patterns based on MPI primitives, whilst it also allows application developers to define their own interaction patterns.

```
1. uses "mpi.template";
2. Connector T1:SendRecvSync {
3.   port t1,t2;
4.   implementation {
5.     t1 = SendRecvSync.t1;
6.     t2 = SendRecvSync.t2;
7.   }
8. }
```

Figure 3. Connector specification.

In order to illustrate the behavioural specification of a connector in ArcMPI, the description of the connector *T1* (see Figure 2) is shown in Figure 3. As mentioned before, the interaction patterns used by the connector must be defined prior to the component behaviour.

In this case, connector *T1* is specified according to the MPI template (line 1) and reuses an existing interaction pattern named *SendRecvSync* (line 2). In lines 5 and 6, ports *t1* and *t2* of connector *T1* are associated with ports *t1* and *t2* of pattern *SendRecvSync* (see Figure 6 and Appendix A1).

The message flow through *T1* is defined by *SendRecvSync* in terms of MPI primitives (for example, an instance of message flow can be described by *MPI_Send* through port *t1* and *MPI_Recv* through port *t2* – see Figure 6). The pattern *SendRecvSync* uses *MPI_Send* and *MPI_Recv* calls in the standard blocking communication mode, i.e., the *MPI_Send* call does not return until the message has been safely stored at the receiver.[4]

It is worth observing that a different behaviour (e.g., request-reply) could be specified instead by *T1*. In this case, however, the behaviour of components *M0* and *M1* must be modified to conform with the new interaction pattern. After the specification of the connector, the components' behaviour is described by the user. A library pattern template is not used for these specifications. As shown in Figure 4, the user defines the component behaviour directly in terms of MPI primitives (line 5).

```
1. uses "mpi.template";
2. component M0{
3.   port p1;
4.   implementation {
5.     p1.MPI_Send(msg);
6.     p1.MPI_Recv(msg);
7.     ...
8.   }
9. }
```

```
uses "mpi.template";
component M1 {
  port p1,p2;
  implementation {
    p1.MPI_Recv(msg);
    p1.MPI_Send(msg);
    ...
  }
}
```

Figure 4. Components' behaviour in ArcMPI.

A simple action infix operator (";") is used to define the temporal ordering of actions executed by the component. In practice, this operator defines that "a_1;a_2" means that a_2 only starts when a_1 finishes. The pattern in lines 5-6 repeats indefinitely as the ArcMPI does not at present have a stop/exit operator to indicate the termination of the components.

It is worth noting that from the point of view of the application developer, the interaction between the components is anonymous in the sense that *M0* does not know (and does not need to know) who is the recipient of the message (*msg*), whilst *M1* similarly has no idea about the sender.

2.4. Formal Modelling

While the *Abstract*, *Concrete* and *Implementation Skeletons* are informal descriptions of the parallel application, the *Formal Specification* artifact, as the name suggests, is described in a formal description language. Hence, a mapping is necessary to generate a formal specification from the *Concrete Specification* described in ArcMPI . This mapping is performed automatically without the programmer's intervention.

CSP [12] and LOTOS [18] have been adopted as target formalism in the modelling of MPI applications written in ArcMPI. Both formalisms have good tool support, namely FDR3 [19] and CADP Toolbox [20], that enable us to integrate the formal verification into the

[4]According to the MPI Standard, correct MPI applications do not rely on system buffering in the standard communication mode. For pattern *SendRecvSync*, such buffering has not been assumed.

development process in a satisfactory. As CSP is a formalism more widely adopted nowadays, only CSP specifications will be shown next. The LOTOS mapping can be found in [21].

The modelling consists of representing ArcMPI elements into CSP constructs and can be divided into structural and behavioural modelling. The structural modelling defines that components and connectors are modelled as CSP processes, ports are defined as CSP channels and the configuration is a parallel composition of components and connectors.

Considering the example introduced in Section 2.2, the component *M2* (see Figure 2) is modelled into CSP as follows:

$$\mathbf{T1} = \text{SendRecvSync}(\{0,1\})$$

$$\mathbf{M2} = (\mathbf{M0}(0)[[p1 \leftarrow t1]] \;|||\; \mathbf{M1}(1)[[p1 \leftarrow t2]]) \quad [|\{|t1,t2|\}|] \quad \mathbf{T1}$$

In this model, $T1$ is defined by instantiating the interaction pattern *SendRecvSync* that connects components $M0$ and $M1$, whose ids are *0* and *1*, respectively. $M0$ and $M1$ are composed in parallel without synchronisation ($|||$). The resulting composition is placed in parallel with the connector *T1* and synchronises on events in $\{t1,t2\}$.

After modelling the structure, it is time to model the behaviour. The behaviour modelling defines how the clause **implementation** of the concrete specification of components and connectors is modelled into CSP. As mentioned before, this clause in ArcMPI consists basically of a temporal ordering of actions executed by the component in its ports. CSP for generic behaviour is shown in Figure 5.

```
        ArcMPI                          CSP
1.  …
2.  component Mx{
3.    port pi;
4.    implementation {
5.      pi.ai;                  ⇨   Mx = pi.ai -> pi.aj… -> Mx
6.      pi.aj;
7.      …
8.    }
9.  }
```

Figure 5. Generic behaviour model of ArcMPI components.

Connectors are highly reusable and common behaviours (e.g., request-reply and publish/subscribe) are already specified in the interaction templates (one of the *Artifacts* shown in Figure 1). For *T1*, the behaviour of the connector is simply instantiated from one of these existing templates: *SendRecvSync*.

Figure 6 shows part of the definition of *SendRecvSync*, where *s* is the set of components (ids) connected by this connector. *SendRecvSync* is initially ready to interact with one of two components through ports *t1* and *t2*, and each component can initially invoke one of two MPI primitives *MPI_Send* or *MPI_Recv* (lines 2-9).

If the component connected to port *t1* invokes MPI primitive *MPI_Send* (line 2), *SendRecvSync* behaves like *SendRecvSyncT1'* (line 3). At this point, the connector waits for the other component (connected to port *t2*) to invoke the MPI primitive *MPI_Recv* (line 12). When this event occurs, *SendRecvSyncT1'* runs *SendRecvSync''''* (line 15). Finally, the *SendRecvSync''''* returns to the interacting components that both invocations (*MPI_Send* and *MPI_Receive*) were successfully terminated (lines 16-17) and recurses back to *SendRecvSync(s)*. [Code for the other three possible interactions is in Appendix A1.]

```
1.  SendRecvSync(s) =
2.      t1.MPI_Send?msg?ct?dt?src:s?dst:s?tag?comm
3.          -> SendRecvSyncT1'(s,msg,ct,dt,dst,src,tag,comm) []
4.      t1.MPI_Recv?msg?ct?dt?src:s?dst?tag?comm?sta
5.          -> SendRecvSyncT1''(s,msg,ct,dt,dst,src,tag,comm,sta) []
6.      t2.MPI_Send?msg?ct?dt?src:s?dst?tag?comm
7.          -> SendRecvSyncT2'(s,msg,ct,dt,dst,src,tag,comm) []
8.      t2.MPI_Recv?msg?ct?dt?src:s?dst?tag?comm?sta
9.          -> SendRecvSyncT2''(s,msg,ct,dt,dst,src,tag,comm,sta)
10.
11. SendRecvSyncT1'(s,msg,ct,dt,src,dst,tag,comm) =
12.     t2.MPI_Recv?msg?ct?dt!src?ip2:diff(s,src)!tag!comm?sta
13.         -> SendRecvSync''''(s)
14.
15. SendRecvSync''''(s) =
16.     (t1!MPI_SUCCESS -> SKIP)|||(t2!MPI_SUCCESS -> SKIP);
17.         -> SendRecvSync(s)
18.
19. -- The complete specification of processes SendRecvSyncT1'',
20. -- SendRecvSyncT2' and SendRecvSyncT2'' can be found
21. -- in Appendix A1.
22. ...
```

Figure 6. Specification of *SendRecvSync* in CSP.

2.5. Code Generation

In this section we present a small application to demonstrate the code generation process. The application is a sieve of Eratosthenes implemented by composing MPI processes to form a pipeline. A pipeline is a commonly used pattern in process-oriented environments for creating process networks within programs and is also used in dataflow applications [22].

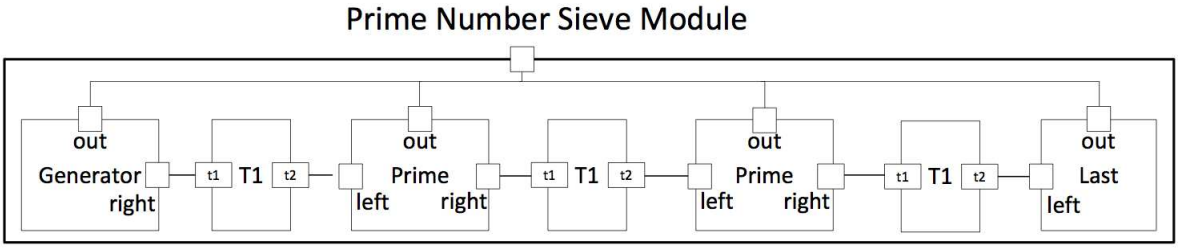

Figure 7. Structural Elements of the Module.

Figure 7 depicts the structural elements that comprise the pipeline application. As shown in Figure 7, there are three types of user defined processes whose intended behaviour is as follows:

- The `generator` process generates odd numbers that are passed down the pipeline. The `generator` keeps the prime number 2 for itself.
- The `prime` process keeps the first prime number it sees and filters the remaining numbers by either discarding them or passing them to the next process in the chain.
- The `last` process terminates the prime number generation at the end of the sieve.

```
#include "firstP.h"
PROCESS generator(MPI_Comm comm, PROCESSVARS *pargs, PROCESSPORTS *ports)
{
      int n=0, ack=1;
      MPI_Send(&n,1,MPI_INT,ports->right.dst,0,ports->right.out);
      MPI_Recv(&ack,1,MPI_INT,ports->right.src,0,ports->right.in,&status);
      return 0;
}

#include "middleP.h"
PROCESS prime(MPI_Comm comm, PROCESSVARS *pargs, PROCESSPORTS *ports)
{
      int n=0, ack=1;
      MPI_Recv(&n,1,MPI_INT,ports->left.src,0,ports->left.in,&status);
      MPI_Send(&ack,1,MPI_INT,ports->left.dst,0,ports->left.out);
      MPI_Send(&n,1,MPI_INT,ports->right.dst,0,ports->right.out);
      MPI_Recv(&ack,1,MPI_INT,ports->right.src,0,ports->right.in,&status);
      return 0;
}

PROCESS last(MPI_Comm comm, PROCESSVARS *pargs, PROCESSPORTS *ports)
{
      int n=0, ack=1;
      MPI_Recv(&n,1,MPI_INT,ports->left.src,0,ports->left.in,&status);
      MPI_Send(&ack,1,MPI_INT,ports->left.dst,0,ports->left.out);
}
```

Figure 8. Partial implementation of the prime number sieve processes.

The number of prime numbers generated is equal to the length of the pipeline (i.e., the total number of MPI processes in this application). For illustration here, we have four MPI processes in this example: one `generator` process, just two `prime` processes and one `last` process.

Figure 7 specifies the structure of the application and information about the communication protocol between the modules in the pipeline. The *T1* connectors are as described in Sections 2.2 and 2.3. An output port was added to each of the processes inside the module. The "out" ports in Figure 7 are connected to a single shared port on the prime sieve module. Each of the processes write their prime number to the out port before sending anything to the next process in the chain. This "many-to-one" sharing of a port is implementable within our system but we have not formally verified its behaviour. The code generation tool uses the structure information to create the following program artifacts:

- stub processes (`.c` files) in the target directory that can be compiled into an application corresponding to the modules in the structure diagram;
- part of the structure (not shown) is the process type which specifies the main program template to be used with the associated processes;
- Makefile entries that take extracted information from the structure diagram to fill in the main program template and then compile and link this with its associated process code;
- an MPI configuration file is also generated from information from the structure diagram;
- finally, the protocol types between modules are used to generate prototypical code that follows the specified communication protocol. Figure 8 shows some of the code generated.

At this point we have a working partial implementation of the application (Figure 8) that executes and exercises each of the connections. The correctness of the partial implementation was verified in the verification step of the methodology in Figure 1. Achieving a working implementation for a given communication pattern is always a time-consuming step in the development of MPI applications and, with these tools, we can continue to refine the structure or begin to add the application logic to the processes. The important point is that we are starting from a working framework that avoids the costly MPI semantic errors that often arise to get to this point in the development. For the completed processes in the resulting application, an explanation of the termination logic and the output port, see the text and code outlined in Appendix A2.

2.6. Tool Support

As mentioned previously, the development process presented in Section 2.1 is supported by a tool named LFD-MPI (Lightweight Formal Development in MPI). As shown in Figure 9, LFD-MPI is composed by a *Graphical Editor*, the A2F (Architecture to Formal) and A2C (Architecture to Code) mappers, *Formal Adapters*, and a repository of *Interaction Templates*. The graphical editor helps application developers in activities *Structural Design* and *Behavioural Design*, whilst the mappers generates the *Formal Specification* and *Implementation Skeleton*.

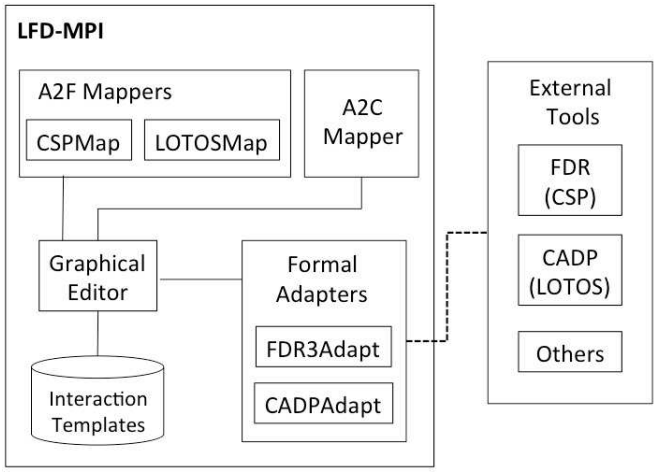

Figure 9. Architecture of LFD-MPI.

The *Formal Adapters* interact with existing tools to verify the formal specification (activity *Verification*). In the current implementation, two formal adapters (*FDR3Adapt* and *CADPAdapt*) interact with FDR3 [19] and CADP Toolbox [20], respectively. The interaction with the CADP Toolbox is carried out by executing some of its components (e.g., compiler CAESAR or on-the-fly model-checker EVALUATOR) and processing the generated files yielded by them. In the case of FDR3Adapt, the interaction occurs by directly invoking the FDR API that exposes part of FDRs internals to external tools.

The *Abstract Specification* and *Concrete Specification* are modelled in ArcMPI and stored as XML-based files; the *Formal Specification* is a CSP or LOTOS file, and the *Implementation Skeleton* is a semi-implementation of the MPI application written in language C. It is worth observing that our support is decoupled from a particular formal description technique, which means that implementation of adapters and mappers is needed for each supported formalism.

3. Evaluation

To evaluate our approach, we initially developed four MPI application patterns where existence of deadlock is known a priori: Recv-Recv, Send-Send, Sched-Dependent, and AnySource.

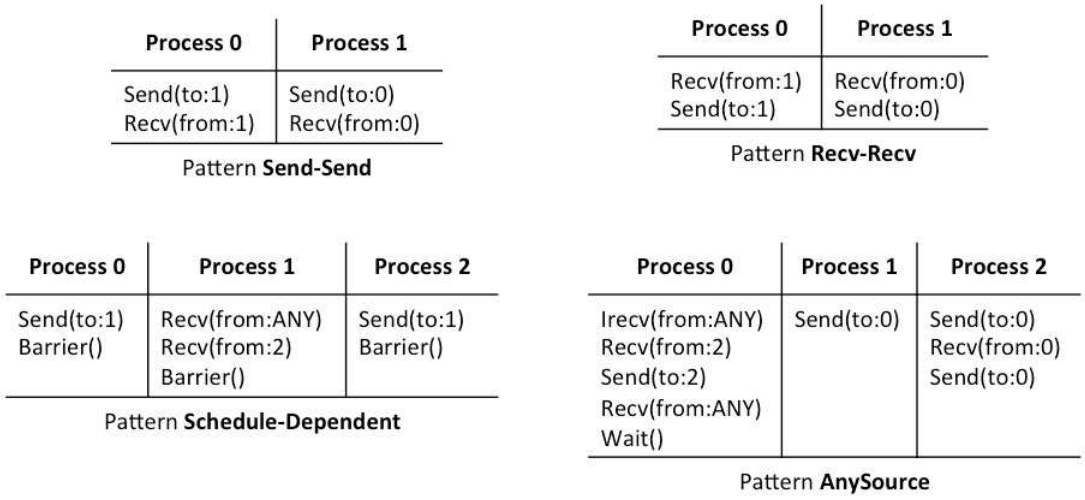

Figure 10. Evaluated Application Patterns.

Figure 10 shows these patterns using a simple notation: "Send(to:x)" and "Recv(from:x)" mean *MPI_Send* and *MPI_Recv* respectively. "x" specifies the destination rank in case of the send operation and the source rank in the receive operation while the remaining arguments are omitted. *MPI_Recv* performs a blocking MPI receive call for a message. This call does not return until a matching message has been stored into the receive buffer.

"Irecv(from:x)" represents *MPI_Irecv*. This call begins a non-blocking MPI receive operation. A request handle is associated with a non-blocking receive call which can later be used to query the status of the communication or wait for its completion. One such routine is *MPI_Wait*, which uses the request handle returned by *MPI_Irecv* to wait for its completion. It is unsafe to access any part of the receive buffer in the program after a non-blocking receive operation has started and before the receive has completed. "Recv(from:ANY)" or "Irecv(from:ANY)" specifies that a message from any sender can potentially match this receive operation provided the rest of the message matching criteria is met (see Section 1).

Deadlock is immediate (and obvious) from the Send-Send and Recv-Recv patterns. For Schedule-Dependent, if the Send from Process 0 is the first message received by Process 1, there will be no deadlock. However, if Process 2 gets its message to Process 1 first, Process 1 will be left waiting for another from Process 2 in its second receive and will never reach its barrier, leaving the three processes deadlocked. For the AnySource pattern, the request handle associated with Irecv(from:ANY) in Process 0 may be matched by the first Send from Process 2.[5] The subsequent Recv(from:2) in Process 0 will then be blocked forever resulting in all three processes unable to proceed further.

These patterns, originally introduced in [23], were developed using our process and then checked for deadlock in the *Formal Specification* generated by LFD-MPI.

[5]In the AnySource pattern, Process 0 can post two outstanding receive requests to the middleware: one through Irecv(from:ANY) and the other through Recv(from:2). This introduces an extra reliance on the implementation of the matching engine in the middleware.

Table 1 shows results considering whether various toolsets (see Section 4) are able to detect deadlock. “Run” means it can be detected, but only at run-time. This is an extension of the comparison originally presented in [23] that includes the *AnySource* application and LFD-MPI. As *AnySource* was not present in the original table, its evaluation by some tools is set to “-” (no evaluation performed). Similarly to ISP, LFD-MPI can generate the sequence of MPI invocations that leads to a deadlock when the CADP adapter (for LOTOS) is used[6], e.g., “first one”, “shortest one” or “all”.

Table 1. Deadlock detection by existing tools (original table shown in [23])

Approach	Recv-Recv	Send-Send	Sched-Dep	AnySource
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

We also measured the response time to detect deadlock in the patterns in Figure 10. The test setup comprised a single Mac OS machine (Version 10.9.5), with 2.9 GHz Intel Core i7 processor and 8 GB of memory. Figure 11 shows the deadlock detection time for each of the patterns using LFD-MPI and the external (LOTOS) tools. The response time is the time from the invocation of the external tool till the receipt of the response from it. In particular, in the case of CADP Toolbox, it was configured to stop and respond to the invocation as soon as the first deadlock is found. For each application, the external tool was invoked 100 times and the results shown in Figure 11 are the mean response time when the CADP is used.

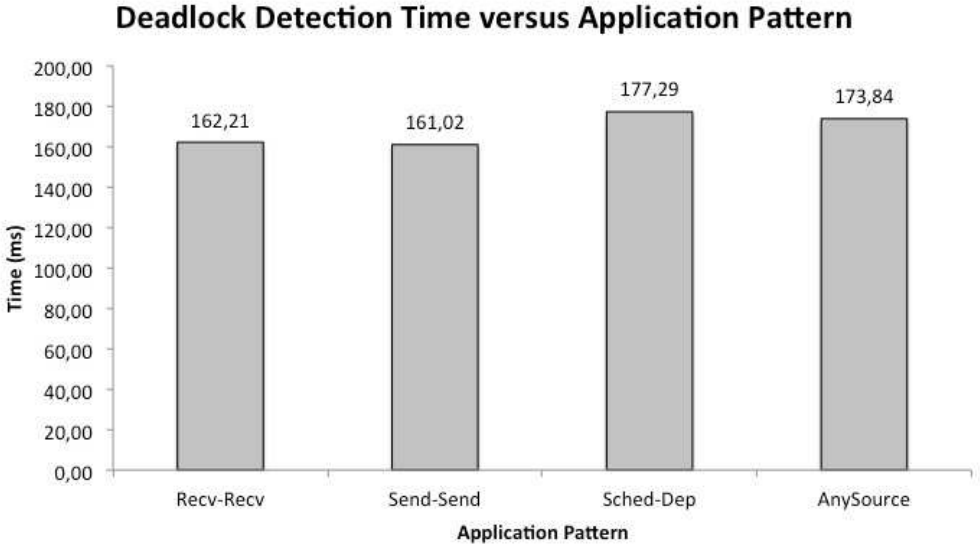

Figure 11. Evaluation results (LFD-MPI and LOTOS).

These preliminary results show that the deadlock detection time has low variation (standard deviation equal to just under 15 ms for a mean of approximately 165 ms). Furthermore,

[6]The current implementation of LFD-MPI still under development and its integration with FDR3 is not complete.

considering that our 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. Moreover, the MPI developer does not require knowledge of the formalism (CSP or LOTOS) used.

4. Related Work

There are a number of approaches for verification of MPI applications. These approaches can be organised into three main groups: purely formal approaches, runtime based approaches and hybrid approaches.

In the formal group, formalisms such as CSP [3], Session types [4] and Abstract State Machines [5] 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. Another well-known approach for verifying MPI applications is MPI-SPIN [6,24]. 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 infrastructure can be executed. This solution was implemented as an extension to SPIN [25] and incorporated into the SPIN specification language Promela.

The runtime group essentially works by intercepting MPI calls at runtime and detecting several kinds of errors of MPI applications. MUST (Marmot Umpire Scalable Tool) [23,7] 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. Essentially, MUST represents the wait-for condition as a graph and uses graph analysis to decide whether a deadlock exists at a particular step of the application execution. This detection strategy follows that used by the earlier Marmot [8] and Umpire [9] toolsets.

The hybrid group uses combinations of the previous strategies. ISP (In-situ Partial Order) [11,15] 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) [26]. Similarly to ISP, DAMPI also intercepts MPI calls, but it executes a distributed scheduling algorithm.

TASS (Tookit for Accurate Scientific Software) [27] also belongs to the hybrid group. It takes a MPI program as input and generates an abstract model through 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 between these approaches and LFD-MPI is that LFD-MPI serves mainly as a tool that helps the *development* of the applications instead of *testing complete* applications. This significantly reduces the code analysed by the verification tool. Additionally, LFD-MPI also generates MPI code. The similarity with purely formal and hybrid approaches is the use of formal methods to help the verification process. However, application developers do not have to work directly with the formalisms, thanks to the adapters.

A recent work [28] presents a framework for generation of type-safe and deadlock-free MPI applications. Their solution is able to verify correctness properties of MPI applications and automatically generates MPI code. Their process is centered around the use of Pabble [29], a language based on the theory of multiparty session types (MPST) [30]. The MPI application is generated from Pabble specification in such way that guarantees, by construction, type-safety, communication-safety and deadlock-freedom.

Similar to our work, the Pabble-based approach also focuses on reducing the development time of MPI applications. The most important differences are the adopted formalisms (session types vs. process algebra), the way of dealing with deadlock (correct by construction vs. verification) and the final product of both approaches (skeletons vs. final code).

5. Conclusion and Future Work

In this paper we presented a lightweight formal approach to build safety into the MPI applications during the application development process. The development is supported by a modelling tool that shields the application developer from the formal techniques used to verify correctness properties. LFD-MPI is designed to serve as a tool that helps the *development* of the applications instead of *testing complete* applications.

Our solution is not complete, as only a subset of MPI point-to-point communication primitives have been considered. However, it is a novel combination of three key aspects: the adoption of a formal lightweight approach for rapid prototyping of MPI applications, the use of a formal model (in CSP and LOTOS) to specify MPI applications and the identification of a strategy for verifying MPI applications that works with partial implementation codes. Other contributions include the automatic generation of skeletons of MPI applications and the presentation of deadlock sequences if deadlock is detected.

A key next step in this research is to extend ArcMPI to allow the definition of alternative event sequences, to explicitly indicate the termination of a component, and to allow the definition of uni-directional and typed ports. These extensions will be useful for understanding and eliminating some connection errors (e.g., input port to input port). Alternatively, we are also planning to integrate the current solution with other verification engines like MPI-Spin [31]. Future extensions include additional interaction patterns such as request-reply, publish/subscribe, and MPI collective communication operations. The motivation of this work came from experience in programming in MPI using FG-MPI [32,33] and the design of exascale-like applications with thousands and potentially millions of processes [34,35]. This work is an important step in externalising the communication inside MPI programs to allow for both verification and performance portability through mapping the application processes onto large compute clusters. There remains challenges in addressing scalability in both verification, representation and mapping.

Acknowledgements

We gratefully acknowledge the contribution of the anonymous reviewers whose invaluable comments significantly improved the paper.

References

[1] MPI Forum. Message-passing interface standard. `http://www.mpi-forum.org/`.

[2] Sarvani Vakkalanka. *Efficient Dynamic Verification Algorithms for MPI Applications*. PhD thesis, Salt Lake City, UT, USA, 2010.

[3] Umberto Souza da Costa, Ivan Soares de Medeiros Junior, and Marcel Vinicius Medeiros Oliveira. *Specification and Verification of a MPI Implementation for a MP-SoC*, volume 6255 of *Lecture Notes in Computer Science*, pages 168–183. 2010.

[4] Kohei Honda, Eduardo Marques, Francisco Martins, Nicholas Ng, Vasco Vasconcelos, and Nobuko Yoshida. Verification of MPI Programs Using Session Types. In JesperLarsson Traff, Siegfried Benkner, and Jack Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 7490 of *Lecture Notes in Computer Science*, pages 291–293. Springer Berlin Heidelberg, 2012.

[5] Igor Grudenic and Nikola Bogunovic. Modeling and Verification of MPI Based Distributed Software. In Bernd Mohr, JesperLarsson Traff, Joachim Worringen, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science*, pages 123–132. Springer Berlin Heidelberg, 2006.

[6] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Combining Symbolic Execution with Model Checking to Verify Parallel Numerical Programs. *ACM Trans. Softw. Eng. Methodol.*, 17(2):10:1–10:34, May 2008.

[7] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Muller. MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. *Scientific Programming*, 21(3):109–121, 2013.

[8] Bettina Krammer, Tobias Hilbrich, Valentin Himmler, Blasius Czink, Kiril Dichev, and MatthiasS. Mller. MPI Correctness Checking with Marmot. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 61–78. Springer Berlin Heidelberg, 2008.

[9] J.S. Vetter and B.R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 51–51, Nov 2000.

[10] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: A Tool for Model Checking MPI Programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 285–286, New York, NY, USA, 2008. ACM.

[11] Anh Vo, Sarvani Vakkalanka, and Ganesh Gopalakrishnan. ISP Tool Update: Scalable MPI Verification. In Matthias S. Mller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 175–184. Springer Berlin Heidelberg, 2010.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[13] David Garlan and Mary Shaw. An Introduction to Software Architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[14] Victor R. Basili, Jeffrey C. Carver, Daniela Cruzes, Lorin M. Hochstein, Jeffrey K. Hollingsworth, Forrest Shull, and Marvin V. Zelkowitz. Understanding the High-Performance-Computing Community: A Software Engineer's Perspective. *IEEE Software*, 25(4):29–36, 2008.

[15] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen Siegel, Rajeev Thakur, William Gropp, Ewing Lusk, Bronis R. De Supinski, Martin Schulz, and Greg Bronevetsky. Formal Analysis of MPI-based Parallel Programs. *Communications of the ACM*, 54(12):82–91, 2011.

[16] Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, Scalable Debugging of MPI Programs with Intel Message Checker. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, SE-HPCS '05, pages 78–82, New York, NY, USA, 2005. ACM.

[17] Valerie Issarny, Mauro Caporuscio, and Nikolaos Georgantas. A Perspective on the Future of Middleware-based Software Engineering. In *Proc. Future of Software Engineering FOSE '07*, pages 244–258, 23–25 May 2007.

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

[19] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3: a Parallel Refinement Checker for CSP. *International Journal on Software Tools for Technology Transfer*, pages 1–19, 2015.

[20] Hubert Garavel, Frdric Lang, Radu Mateescu, and Wendelin Serwe. 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.

[21] Nelson Rosa, Humaira Kamal, and Alan Wagner. A LOTOS-based Lightweight Approach to Formally Verify MPI Applications. Technical report, Universidade Federal de Pernambuco, Centro de Informática, 2015. `http://gfads.cin.ufpe.br/bib/Rosa2015.pdf`.

[22] Adam Sampson. *Process-Oriented Patterns for Concurrent Software Engineering*. PhD thesis, Computing, University of Kent, CT2 7NF, September 2008.

[23] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Muller. MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 30:1–30:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[24] Stephen F. Siegel and Ganesh Gopalakrishnan. *Formal Analysis of Message Passing*, volume 6538 of *Lecture Notes in Computer Science*, pages 2–18. 2011.

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

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

[27] Stephen F. Siegel and Timothy K. Zirkel. Automatic Formal Verification of MPI-Based Parallel Programs. *ACM Sigplan Notices*, 46(8):309–310, 2011.

[28] Nicholas Ng, Jose G.F. Coutinho, and Nobuko Yoshida. Protocols by Default: Safe MPI Code Generation based on Session Types. In *CC 2015*, volume 9031 of *LNCS*, pages 212–232. Springer, 2015.

[29] N. Ng and N. Yoshida. Pabble: Parameterised Scribble for Parallel Programming. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 707–714, Feb 2014.

[30] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *SIGPLAN Not.*, 43(1):273–284, January 2008.

[31] Stephen F. Siegel and George S. Avrunin. Verification of Halting Properties for MPI Programs Using Nonblocking Operations. In Franck Cappello, Thomas Herault, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 326–334. Springer Berlin Heidelberg, 2007.

[32] Humaira Kamal and Alan Wagner. An Integrated Fine-Grain Runtime System for MPI. *Computing*, 96(4):293–309, 2014.

[33] Humaira Kamal and Alan Wagner. Added Concurrency to Improve MPI Performance on Multicore. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 229 –238, Sept. 2012.

[34] Sarwar Alam, Humaira Kamal, and Alan Wagner. Service Oriented Programming in MPI. In *Communicating Process Architectures 2013*, pages 93–112. Open Channel Publishing Ltd., England, August 2013.

[35] Sarwar Alam, Humaira Kamal, and Alan Wagner. A Service-oriented Scalable Dictionary in MPI. In *Communicating Process Architectures 2014*. Open Channel Publishing Ltd., England, August 2014.

Appendices

Appendix A1: Modelling of SendRecvSync in CSP

Here is the complete specification of *SendRecvSync* (outlined in Figure 6, Section 2.4).

```
SendRecvSync(s) =
  t1.MPI_Send?msg?ct?dt?src:s?dst:s?tag?comm
    -> SendRecvSyncT1'(s,msg,ct,dt,dst,src,tag,comm)
  []
  t1.MPI_Recv?msg?ct?dt?src:s?dst?tag?comm?sta
    -> SendRecvSyncT1''(s,msg,ct,dt,dst,src,tag,comm,sta)
  []
  t2.MPI_Send?msg?ct?dt?src:s?dst?tag?comm
    -> SendRecvSyncT2'(s,msg,ct,dt,dst,src,tag,comm)
  []
  t2.MPI_Recv?msg?ct?dt?src:s?dst?tag?comm?sta
    -> SendRecvSyncT2''(s,msg,ct,dt,dst,src,tag,comm,sta)

SendRecvSyncT1'(s,msg,ct,dt,src,dst,tag,comm) =
  t2.MPI_Recv?msg?ct?dt!src?ip2:diff(s,{src})!tag!comm!0
    -> SendRecvSync''''(s)

SendRecvSyncT2'(s,msg,ct,dt,src,dst,tag,comm) =
  t1.MPI_Recv?msg?ct?dt!src?ip2:diff(s,{src})!tag!comm!0
    -> SendRecvSync''''(s)

SendRecvSyncT1''(s,msg,ct,dt,src,dst,tag,comm,sta) =
  t2.MPI_Send!msg!ct!dt!src?ip2:diff(s,{src})!tag!comm
    -> SendRecvSync''''(s)

SendRecvSyncT2''(s,msg,ct,dt,src,dst,tag,comm,sta) =
  t1.MPI_Send!msg!ct!dt!src?ip2:diff(s,{src})!tag!comm
    -> SendRecvSync''''(s)

SendRecvSync''''(s) =
   (t1!MPI_SUCCESS -> SKIP) ||| (t2!MPI_SUCCESS -> SKIP);
            SendRecvSync(s)
```

Appendix A2: Completed Prime Sieve Application

Figure 12 repeats the prime sieve module (Figure 7) together with a print module, `Out`, where the many-to-one port of the prime sieve module connects to the print module by a *T2* connector. The *T2* connector as implemented provides a many-to-one unordered connection.

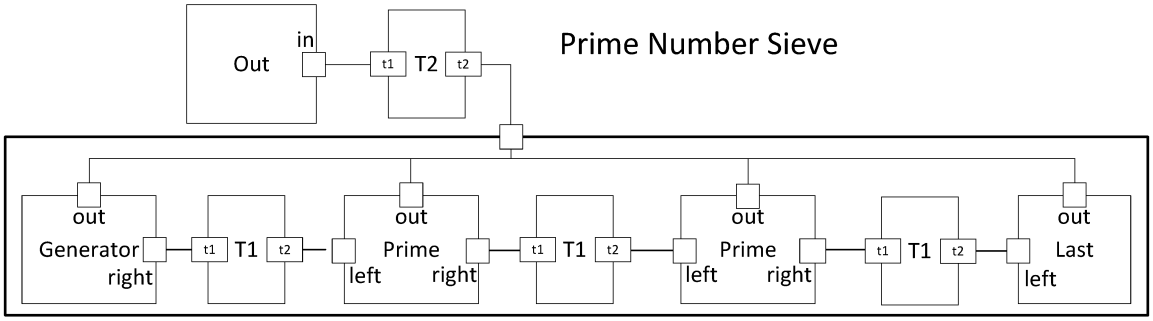

Figure 12. Prime sieve module, as in Figure 7, connected to a print module.

Every number sent on a *right* port in Figure 7 is acknowledged by a *Boolean* signal sent by the process on its right: this conforms to the *SendRecvSync* pattern. The first number received by a `Prime` process is prime: it acknowledges this with `FALSE`, remembers the prime and outputs the value to its *out* port. Subsequent numbers received are also acknowledged with `FALSE` unless, having forwarded that number (because it was not divisible by its prime), it receives a `TRUE` acknowledgement. The `TRUE` acknowledgement signals termination: it passes this signal to the process on its left, as acknowledgement of the number received, and then terminates. The `Last` process simply acknowledges the first number (a prime) it receives with `TRUE`, sends the number to its *out* port and terminates. Using FDR, this termination can be verified not to result in deadlock.

The *out* port is a many-to-one port that uses `MPI_ANY_SOURCE` to receive a prime number from any of the sieve processes. Because of the process codings (Figure 13), one prime only will arrive from each process in strict left-to-right order of the processes as drawn in Figure 12 (and they will be in ascending order with no primes missed).

```
#include "outP.h"
/* output process to receive and print all of the primes */
PROCESS out(MPI_Comm groupcomm, PROCESSVARS *pargs, PROCESSPORTS *ports) {
  int size, myprime=0;
  MPI_Status status;
  MPI_Comm_size(ports->in.out,&size);   // ANY-PORT
  while (--size) {
    MPI_Recv(&myprime,1,MPI_INT,MPI_ANY_SOURCE,0,ports->in.out,&status);
    printf("%d\n",myprime);
  }
  pargs->shutdown = TRUE;
  return 0;
}

#include "firstP.h"
/* generates a sequence of odd numbers */
PROCESS generator(MPI_Comm groupcomm, PROCESSVARS *pargs, PROCESSPORTS *ports) {
  MPI_Status status;
  int myprime = 2, number = 3, stop = FALSE;
  MPI_Send(&myprime,1,MPI_INT,ports->out.src,0,ports->out.in);
  while ( !stop ) {
    MPI_Send(&number,1,MPI_INT,ports->right.dst,0,ports->right.out);
    MPI_Recv(&stop,1,MPI_INT,ports->right.src,0,ports->right.in,&status);
    number = number+2;
  }
  pargs->shutdown = TRUE;
  return 0;
}

#include "middleP.h"
PROCESS prime(MPI_Comm groupcomm, PROCESSVARS *pargs, PROCESSPORTS *ports) {
  MPI_Status status;
  int number, myprime, stop = FALSE;
  MPI_Recv(&myprime,1,MPI_INT,ports->left.src,0,ports->left.in,&status);
  MPI_Send(&stop,1,MPI_INT,ports->left.dst,0,ports->left.out);
  MPI_Send(&myprime,1,MPI_INT,ports->out.src,0,ports->out.in);
  while ( !stop ) {
    MPI_Recv(&number,1,MPI_INT,ports->left.src,0,ports->left.in,&status);
    if ( number % myprime != 0) {
      MPI_Send(&number,1,MPI_INT,ports->right.dst,0,ports->right.out);
      MPI_Recv(&stop,1,MPI_INT,ports->right.src,0,ports->right.in,&status);
    }
    MPI_Send(&stop,1,MPI_INT,ports->left.dst,0,ports->left.out);
  }
  pargs->shutdown = TRUE;
  return 0;
}

#include "lastP.h"
PROCESS last(MPI_Comm groupcomm, PROCESSVARS *pargs, PROCESSPORTS *ports) {
  MPI_Status status;
  int myprime, stop = TRUE;
  MPI_Recv(&myprime,1,MPI_INT,ports->left.src,0,ports->left.in,&status);
  MPI_Send(&stop,1,MPI_INT,ports->left.dst,0,ports->left.out);
  MPI_Send(&myprime,1,MPI_INT,ports->out.src,0,ports->out.in);
  pargs->shutdown = TRUE;
  return 0;
}
```

Figure 13. C code of processes implementing the prime number sieve.