Future-based Static Analysis of Message Passing Programs

Wytse Oortwijn Stefan Blom Marieke Huisman
Formal Methods and Tools, Dept. of EEMCS, University of Twente
P.O.-box 217, 7500 AE Enschede, The Netherlands

{w.h.m.oortwijn, s.c.c.blom, m.huisman}@utwente.nl

Message passing is widely used in industry to develop programs consisting of several distributed communicating components. Developing functionally correct message passing software is very challenging due to the concurrent nature of message exchanges. Nonetheless, many safety-critical applications rely on the message passing paradigm, including air traffic control systems and emergency services, which makes proving their correctness crucial. We focus on the modular verification of MPI programs by statically verifying concrete Java code. We use separation logic to reason about local correctness and define abstractions of the communication protocol in the process algebra used by mCRL2. We call these abstractions *futures* as they predict how components will interact during program execution. We establish a provable link between futures and program code and analyse the abstract futures via model checking to prove global correctness. Finally, we verify a leader election protocol to demonstrate our approach.

1 Introduction

Many industrial applications, including safety-critical ones, consist of several disjoint components that use message passing to communicate according to some protocol. These components are typically highly concurrent, since messages may be sent and received in any order. Developing functionally correct message passing programs is therefore very challenging, which makes proving their correctness crucial [9].

The Message Passing Interface (MPI) is a popular API for implementing message passing programs. We focus on the modular verification of MPI programs, in particular programs written with Java and the MPJ library [2]. Existing research mainly focusses on proving communication correctness over an abstract system model [10, 15]. Instead, we target concrete Java source code and combine well-known techniques for static verification with process algebras to reason about communication correctness [13]. Communication correctness refers to the correctness of handling messages, i.e. freedom of deadlocks and livelocks, resource leakage, proper matching of sends and receives, etcetera.

Global system correctness depends on the correctness of all individual processes and their interactions. We use permission-based separation logic [14] to statically reason about local correctness. In addition, we model the communication protocol in the mCRL2 process algebra. We refer to these algebraic terms as *futures* as they predict how components will interact during program execution [16][1]. We extend the VerCors toolset [3, 5] to establish a provable link between futures and program code so that reasoning about futures corresponds to reasoning about the concrete program. Analysing futures can then be reduced to a parameterised model checking problem to reason about the functional and communication correctness of the system. Since model checkers inherently target finite-state systems, we use abstraction and cut-offs to reason about programs with infinite behaviour whenever possible.

[1] Not to be confused with the concepts of "futures and promises" used in MultiLisp, described by Halstead [7] and Liskov and Shrira [12].

D. Orchard and N. Yoshida (Eds.): Programming Language Approaches to Concurrency- and Communication-Centric Software (PLACES 2016).
EPTCS 211, 2016, pp. 65–72, doi:10.4204/EPTCS.211.7

In this paper we analyse the behaviour of several standard MPI operations and model their semantics as process algebra terms. We define several actions, like **send**, **recv**, and **bcast**, that can be used in futures to specify process communication, like that done with Session Types [11]. After that, we show how to analyse futures, in combination with the process algebra terms that model the semantics of the MPI runtime. The analysis is done via parameterised model checking, for which we plan to use the mCRL2 toolset [6]. The key element of our approach is that, by verifying safety and liveness properties on the futures, we actually verify equivalent properties on the concrete system of *any* size. This allows us to check whether process communication always leads to a valid system state in every system configuration. We also plan to check other interesting safety and liveness properties, including: deadlock freedom, resource leakage, and global termination.

This paper is organised as follows. Section 2 introduces the message passing paradigm and discusses the semantics of several standard MPI operations. Section 3 contributes an algebraic abstract model of the network environment. Section 4 shows how futures and the abstract network environment are used to reason about concrete program code. Section 5 demonstrates our approach by verifying a leader election protocol. Finally, Section 6 summarises our conclusions and presents future work.

2 The message passing paradigm

MPI programs consist of a group of interconnected processes $(p_0, \ldots, p_{N-1})$ distributed over one or more devices connected via some network. Each process p_j executes the same program in Single Program Multiple Data (SPMD) fashion, in which j is called the *rank* and N the *size*. MPI mainly targets distributed memory systems; processes maintain a private memory and accesses to non-local memories are handled exclusively via message exchanges. Verifying MPI programs involves verifying functional and communication correctness for *every* rank and size.

The MPI standard describes a set of subroutines to allow processes to exchange messages. We briefly discuss several standard point-to-point and collective MPI operations and their semantics.

Standard operations. A process p_i can send a message with data element D to process p_j by calling p_i.`MPI_Send`(j,D). Similarly, p_i may receive a message from p_j with content D by invoking $D := p_i$.`MPI_Recv`(j). We remove the prefix "p_i." if the sending process can be inferred from the context. Every `MPI_Send` must be matched by a `MPI_Recv`. We allow wildcard receives, i.e. calling p_i.`MPI_Recv`$(\star)$ to receive from any process. In this paper we omit tags and communicators.

The MPI standard describes four sending modes for point-to-point communication, namely: blocking mode, buffered mode, synchronous mode, and ready mode. The p_i.`MPI_Send`(j,D) operation is a blocking mode send, as it *may* block p_i until matched by a call p_j.`MPI_Recv`(i). The buffered variant, p_i.`MPI_BSend`, does not block p_i until matched by a `MPI_Recv` if the request can be buffered. The synchronous variant, p_i.`MPI_SSend`, *always* blocks p_i until matched by a `MPI_Recv`. Finally, the ready mode variant, `MPI_RSend`, only succeeds if a matching `MPI_Recv` has already been posted. From now, we only consider the blocking mode `MPI_Send` variant to illustrate our approach. In future work we also provide support for the other three communication modes.

Non-blocking variants. The immediate send, `MPI_ISend`, is an immediate variant on the standard blocking send. Calling $r := p_i$.`MPI_ISend`(j,D) is *not* allowed to block p_i for a matching `MPI_Recv` by p_j. Instead, `MPI_ISend` returns a request handle r which can be used to query or influence the status of the request. The operation p_i.`MPI_Wait`(r) blocks p_i until the operation corresponding to the handle

r has been completed. Two consecutive non-blocking calls p_i.MPI_ISend$(j,\star)$ and p_i.MPI_ISend$(k,\star)$ made by process p_i are *forced* to match in order if $j = k$, but may be handled in *any* order if $j \neq k$.

Apart from the immediate blocking send, also the other three sending modes have immediate variants, which are: MPI_IBsend, MPI_ISsend, and MPI_IRsend. In this paper we only focus on MPI_ISend. In future work we provide support for the other three immediate operations.

Collective operations. The collective p_i.MPI_Barrier() operation blocks execution of p_i until matched by a call p_j.MPI_Barrier() for every $j \neq i$ and can therefore be used to synchronise all processes. Non-blocking operations invoked before calling MPI_Barrier may complete while lingering in the barrier, as they are handled by the network environment. We plan to provide support for other collective operations, like: MPI_Reduce, MPI_Scatter, and MPI_Gather, but this is future work. The collective p_i.MPI_Bcast(D) operation is used to broadcast a data element D to every participating process. For now we assume that p_i.MPI_Bcast(D) simply calls p_i.MPI_Send(j,D) for every $j \neq i$.

3 Modelling the communication network

We use futures to predict the communication protocol of MPI programs. In this section we specify abstract actions, e.g. **send**, **recv**, and **bcast**, that correspond to concrete MPI operations. We also model the network environment and thereby the semantic behaviour of the MPI operations. Futures are constructed via the following syntax, which is a subset of the multi-action process algebra used in mCRL2, proposed by Groote and Mousavi [6]:

$$P ::= \mathbf{a} \mid P + P \mid P \cdot P \mid P \parallel P \mid c \to P \mid c \to P \diamond P \mid \sum_{d \in T} P(d) \mid \mathsf{X}(u_1, \ldots, u_n)$$

Actions have the form $\mathbf{a} : T_1 \times \cdots \times T_n$, where all T_i are types, e.g. sets, maps, integers, sequences, and so on. We include τ as a special silent action. Sequential composition is denoted by $P \cdot P'$ (P is executed before P') and choice is denoted by $P + P'$ (either P or P' is executed). Summations $\sum_{d \in T} P(d)$ of choices are also supported, where P is executed for some model d of type T. The expression $P \parallel P'$ means putting P and P' in parallel, thereby allowing their actions to be interleaved. Conditions are written either as $c \to P$ (execute P only if c is true) or $c \to P \diamond P'$ (execute P if c is true, otherwise execute P'). Finally, $\mathsf{X}(u_1, \ldots, u_n)$ calls a process X with a list of n arguments of appropriate types.

Two actions $\mathbf{a}, \mathbf{b}$ may communicate by specifying a *communication pair*, written $\mathbf{a}|\mathbf{b}$. Two communicating actions are forced to happen simultaneously and thereby impose restrictions on the evaluation of actions. As an effect, communication pairs introduce synchronisation points between processes.

Modelling the network environment. Recall that MPI programs are executed on a group of N processes $(p_0, \ldots, p_{N-1})$. Let $\mathscr{R} = \{0, \ldots, N-1\}$ be the set of ranks. The network can be modelled as a collection of queues that store pending messages, i.e. messages sent but not yet received. Therefore, we model the network environment as a recursive process, named Network, that maintains a set $T = \{Q_{i,j} \mid i, j \in \mathscr{R}\}$ of queues $Q_{i,j}$ for each pair of ranks $i, j \in \mathscr{R}$ to maintain the state of the network. Network is defined as follows, where $\mathscr{M}$ denotes the set of all possible messages:

$$\textbf{process } \mathsf{Network}(T) \equiv \sum_{i,j \in \mathscr{R}} \sum_{m \in \mathscr{M}} \Big(\mathbf{nrecv}(i,j,m) \cdot \mathsf{Network}(T.enqueue(i,j,m)) + T.peek(i,j,m) \to \mathbf{nsend}(j,i,m) \cdot \mathsf{Network}(T.dequeue(i,j)) \Big)$$

Observe that Network uses two actions, namely: (1) **nsend** for sending a message from the network to a process; and (2) **nrecv** for receiving a message from a process. Processes may use the network by communicating with these two actions. More specifically, we define two actions, **send** and **recv**, that correspond to the functions `MPI_Send` and `MPI_Recv`, respectively. Standard blocking-mode sends and receives are performed via the communication pairs **send**|**nrecv**, **recv**|**nsend**, and **send**|**recv**. For the immediate blocking send, `MPI_ISend`, we also define a corresponding action **isend** and use the communication pair **isend**|**nrecv**. Moreover, the $T.enqueue(i,j,m)$ and $T.dequeue(i,j)$ functions are just auxiliary functions used to enqueue/dequeue elements from $Q_{i,j}$, which can easily be specified in mCRL2. In the remaining of this paragraph, we give more detail on the abstract actions and communication pairs.

When a future performs the $\mathbf{send}(i,j,m)$ action, which corresponds to a call p_i.`MPI_Send`(j,m) in the program code, the network may receive the message via communication with $\mathbf{nrecv}(i,j,m)$, i.e. by having the communication pair **send**|**nrecv**. After communication, the message is stored in the network by applying $T.enqueue(i,j,m)$, which enqueues m onto $Q_{i,j}$. Similarly, the $\mathbf{recv}(i,j,m)$ action, which corresponds to the invocation $m := p_i$.`MPI_Recv`(j), communicates with **nsend**, i.e. **recv**|**nsend**. The network environment can send a message m to process p_i via the $\mathbf{nsend}(i,j,m)$ action if p_i chooses to communicate with the network via a matching $\mathbf{recv}(i,j,m)$ action. The network can only send the top element of $Q_{i,j}$ as the MPI standard enforces a FIFO order, hence the check $T.peek(i,j,m)$, which returns true only if m is the top element of $Q_{i,j}$. When p_i chooses to receive the message m, the queue $Q_{i,j}$ is updated by dequeuing the message m, which is done by applying $T.dequeue(i,j,m)$. Since p_i.`MPI_Send` *may* block p_i until matched by a `MPI_Recv` but is not forced to do so, we also allow **send** to communicate directly with **recv**, i.e. having the communication pair **send**|**recv**, thereby bypassing the network process.

For the immediate blocking send, `MPI_ISend`, we define a corresponding action **isend**, so that calling $r := p_i$.`MPI_ISend`(j,D) corresponds to $\mathbf{isend}(i,j,m,r)$. The **isend** action can not directly communicate with **recv** and therefore *only* communicates via the network: **isend**|**nrecv**. Note that two consecutive actions $\mathbf{isend}(i,j,m,r)$ and $\mathbf{isend}(i,k,m',r')$ performed by rank i match in order if $j = k$, since m and m' are consecutively added to the same queue $Q_{i,j} = Q_{i,k}$ and thus handled in that specific order. The two actions may perform in any order if $j \neq k$, as they are added to different queues $Q_{i,j} \neq Q_{i,k}$.

To support other send modes, the Network process is still too simple. For example, to support the buffered sends `MPI_Bsend` and `MPI_IBsend`, buffer space needs to be taken into account, which may potentially break the FIFO ordering. Ready-mode sends require extra checks to ensure that the network already contains a matching receive. Synchronous-mode sends only allow direct synchronisations, e.g. having the communication pair **ssend**|**recv**, where **ssend** corresponds to `MPI_Ssend`.

Short example of network interaction. To illustrate the use of multi-actions to communicate with the network environment, we give a short producer/consumer example. Suppose that the network is used by two processes: (1) a producer that only sends messages; and (2) a consumer that only receives messages sent by the producer. The producer and consumer are defined as follows:

$$\textbf{process } \mathsf{Producer}(v:\mathbb{Z}) \equiv \mathbf{send}(0,1,v) \cdot \mathsf{Producer}(v+1)$$

$$\textbf{process } \mathsf{Consumer}(t:\mathbb{Z}) \equiv \sum_{n\in\mathbb{Z}} \mathbf{recv}(1,0,n) \cdot \mathsf{Consumer}(t+n)$$

Consider the initial configuration $\mathsf{Producer}(0) \parallel \mathsf{Consumer}(0) \parallel \mathsf{Network}(T_\emptyset)$, where $T_\emptyset$ denotes the set of empty queues, i.e. $|Q_{i,j}| = 0$ for every $Q_{i,j} \in T_\emptyset$. From the initial configuration, the only allowed transition is the multi-action **send**|**nrecv**, which results into the configuration $\mathsf{Producer}(1) \parallel$

$\mathsf{Consumer}(0) \parallel \mathsf{Network}(T')$, where T' is equal to $T_\emptyset$, but with 0 enqueued onto $Q_{0,1}$. From this configuration, either **send**|**nrecv** (the producer sends another value to the network) or **recv**|**nsend** (the consumer receives a value from the network) may happen, and this process repeats forever.

Modelling broadcasts and barriers. For broadcasting we define the action $\mathbf{bcast}(i,m)$ that corresponds to the function call $p_i.\texttt{MPI\_Bcast}(m)$. Broadcasts are handled by a separate process, called Bcast, dedicated to transform a call $p_i.\texttt{MPI\_Bcast}(m)$ into a series of calls $p_i.\texttt{MPI\_Send}(j,m)$ for every $j \neq i$. The Bcast process is defined as follows:

$$\textbf{process } \mathsf{Bcast}() \equiv \sum_{i \in \mathscr{R}} \sum_{m \in \mathscr{M}} \Big(\mathbf{breq}(i,m) \cdot \mathsf{Handle}(i,m,\mathscr{R}\backslash\{i\}) \Big)$$

$$\textbf{process } \mathsf{Handle}(i,m,R) \equiv (R \neq \emptyset) \rightarrow \sum_{j \in R} \Big(\mathbf{nsend}(i,j,m) \cdot \mathsf{Handle}(i,m,R\backslash\{j\}) \Big) \diamond \mathsf{Bcast}()$$

When a future starts broadcasting by performing $\mathbf{bcast}(i,m)$, the Bcast process receives the broadcast request by communicating with **breq**, i.e. via the communication pair **bcast**|**breq**. The Handle process actually handles the broadcast request by generating a $\mathbf{nsend}(i,j,m)$ action for every $j \neq i$. Any process p_j may then receive the message m by communicating via a standard **recv** action. Note that $\mathsf{Handle}(i,m,R)$ only calls Bcast if all ranks $j \in \mathscr{R}\backslash\{i\}$ have synchronised on $\mathbf{nsend}(i,j,m)$.

For handling barriers we define the action $\mathbf{barrier}(i)$ that corresponds to the call $p_i.\texttt{MPI\_Barrier}()$. Similar to broadcasts, also barriers are handled by a dedicated process, named Barrier, which simply synchronises with the **barrier** action. In particular, performing an action $\mathbf{barrier}(i)$ prevents further actions to be executed until Barrier has synchronised with $\mathbf{barrier}(j)$ for every $j \neq i$. We omit the definition of Barrier, as it is essentially the same as Bcast.

4 Linking futures to program code

We use permission-based separation logic [14] to reason about local correctness of MPI programs. We include the permissions, since we allow MPI processes to create threads. In particular, we assume that MPI programs S have preconditions P and postconditions Q, so that partial correctness can be proven via Hoare triples $\{P\}S\{Q\}$. The conditions P and Q may then use permissions to guarantee data-race freedom in case of multi-threading.

Hoare triple reasoning. To find a correspondence between program code and futures, we use Hoare triple reasoning. More specifically, we extend Hoare triples to verify that an MPI program satisfies its algebraically predicted behaviour. For example, we may specify a future: "$\mathbf{send}(i,j,m) \cdot \mathbf{recv}(j,i,n)$" that predicts the behaviour of the program fragment: "$p_i.\texttt{MPI\_Send}(j,m);\ n := p_i.\texttt{MPI\_Recv}(j)$". We prove a link between the future and the program fragment via the Hoare triples: $\{\mathbf{send}(i,j,m) \cdot \mathbf{recv}(j,i,n) \cdot \mathsf{F}\}p_i.\texttt{MPI\_Send}(j,m)\{\mathbf{recv}(j,i,n) \cdot \mathsf{F}\}$ and $\{\mathbf{recv}(j,i,n) \cdot \mathsf{F}\}n = \texttt{MPI\_Recv}(j)\{\mathsf{F}\}$, where F describes the future of the remaining program. To generalise, for the **send**, **recv**, **bcast**, and **barrier** actions we use the following four Hoare triple axioms:

$$[\text{send}]: \frac{}{\{\mathbf{send}(i,j,m) \cdot \mathsf{F}\}p_i.\texttt{MPI\_Send}(j,m)\{\mathsf{F}\}} \qquad [\text{recv}]: \frac{}{\{\mathbf{recv}(i,j,m) \cdot \mathsf{F}\}m := p_i.\texttt{MPI\_Recv}(j)\{\mathsf{F}\}}$$

$$[\text{bcast}]: \frac{}{\{\mathbf{bcast}(i,m) \cdot \mathsf{F}\}p_i.\texttt{MPI\_Bcast}(m)\{\mathsf{F}\}} \qquad [\text{barrier}]: \frac{}{\{\mathbf{barrier}(i) \cdot \mathsf{F}\}p_i.\texttt{MPI\_Barrier}()\{\mathsf{F}\}}$$

All other actions that correspond to MPI functions, e.g. immediate sends like **isend**, alternative send modes like **ssend**, **bsend**, etcetera, are proven to correspond to the MPI program in the same way. We extend the VerCors toolset to verify these Hoare triples by specifying appropriate triples to handle sequential composition of futures: $\mathsf{F} \cdot \mathsf{G}$, choices between futures: $\mathsf{F} + \mathsf{G}$, and so on.

Splitting futures. Since we allow multi-threading, we use permission-based separation logic to prove local correctness of MPI programs. The process algebraic equivalent to multi-threading is the parallel composition: $\mathsf{F} \parallel \mathsf{G}$. We assign permissions to futures, written "Future(π, F)", meaning that a permission fragment π is assigned to the future F. Then we allow splitting and merging of futures: "Future$(\pi_1 + \pi_2, \mathsf{F} \parallel \mathsf{G}) * -* \text{Future}(\pi_1, \mathsf{F}) * \text{Future}(\pi_2, \mathsf{G})$" in separation logic style. The splitted futures can then be distributed among parallel threads, where the permissions can be used for allocating resource invariants.

We refer to the initial future as the *global* future, since it has full permission. The global future can be split into *local* futures with fractional permissions. Proving correctness of local futures with respect to a program fragment is done via standard Hoare logic reasoning [16]. For example, the algorithm in Figure 1, which is discussed in detail in Section 5, shows how program code is annotated with futures. In Figure 1, all futures have full permission since the algorithm does not fork additional threads.

Communication correctness. Let P be an MPI program and F its predicted global future. If we can prove that P correctly executes according to F (i.e. proving $\{\mathsf{F}\}P\{\varepsilon\}$ for the empty process ε), then we need to analyse $\mathsf{F} \parallel \cdots \parallel \mathsf{F}$ to reason about functional and communication correctness of the network of processes all running P. Let $\mathsf{F}^n = \mathsf{F} \parallel \cdots \parallel \mathsf{F}$ be the parallel composition of n futures F. In particular, we need to verify correctness of F^N for *every* size N, in combination with the network environment. Therefore, we will use mCRL2 [1] to analyse the following configuration, where $T_\emptyset$ denotes the set of empty queues:

$$\mathsf{F}^N \parallel \Big(\mathsf{Network}(T_\emptyset) \parallel \mathsf{Bcast}() \parallel \mathsf{Barrier}()\Big)$$

5 Example: A leader election protocol

We illustrate our approach on a small example with N processes $(p_0, \ldots, p_{N-1})$ performing a leader election protocol. The processes communicate in a ring topology, so that process p_i only sends messages to p_{i+1} and only receives messages from p_{i-1}, counting modulo N. Each process p_i holds a *unique* integer v_i so that $v_i \neq v_j$ for $i \neq j$. The leader is the process p_j with the highest value v_j, that is, $v_j = max\{v_0, \ldots, v_{N-1}\}$. The protocol operates in a number of rounds. In each round, each process circulates and remembers its highest encountered value. Ultimately, after N rounds, all processes know the highest participating value v_j and the leader announces itself by broadcasting its rank.

Figure 1 shows annotated pseudocode. The program distinguishes between two kinds of messages: $\texttt{elect}\langle n \rangle$ for communicating an integer $n \in \mathbb{Z}$, and $\texttt{lead}\langle j \rangle$ for communicating a rank $j \in \mathscr{R}$. Below the predicted futures Elect and Choose are given for `election` and `chooseLeader`, respectively.

$$\begin{aligned}
&\textbf{process } \mathsf{Elect}(i,v,h,n) \equiv (n < N) \to \sum_{h' \in \mathscr{M}} \Big(\textbf{send}(i, i+1 \bmod N, \texttt{elect}\langle h \rangle) \\
&\qquad \cdot \textbf{recv}(i-1 \bmod N, i, \texttt{elect}\langle h' \rangle) \cdot \mathsf{Elect}(i, v, max(h,h'), n+1)\Big) \diamond \mathsf{Choose}(i,h,v) \\
&\textbf{process } \mathsf{Choose}(i,h,v) \equiv \Big((h = v) \to \textbf{bcast}(i, \texttt{lead}\langle i \rangle) \diamond \sum_{j \in \mathscr{R}} \textbf{recv}(j, i, \texttt{lead}\langle j \rangle)\Big) \cdot \textbf{barrier}(i)
\end{aligned}$$

```
1  requires 0 ≤ n ≤ N ∧ h ≥ v
2  requires Future(Elect(i,h,v,n) · F)
3  ensures Future(F)
4  def election(int h, int v, int n):
5      int i ← MPI_Rank()
6      int N ← MPI_Size()
7      MPI_Send(i+1 mod N, elect⟨h⟩)
8      elect⟨h'⟩ ← MPI_Recv(i−1 mod N)
9      if h < h' then h ← h'
10     if n < N then election(h, v, n+1)
11     else chooseLeader(h, v)
```

```
1  requires h ≥ v
2  requires Future(Choose(i,h,v) · F)
3  ensures Future(F)
4  def chooseLeader(int h, int v):
5      int i ← MPI_Rank()
6      if h = v then
7          MPI_Bcast(lead⟨i⟩)
8      else
9          lead⟨j⟩ ← MPI_Recv(⋆)
10     MPI_Barrier()
```

Figure 1: Annotated example program of a leader election protocol with simplified MPI syntax.

Assume that each process p_i is started by invoking $\texttt{election}(v_i, v_i, 0)$ and therefore starts with the initial future $\mathsf{Elect}(i, v_i, v_i, 0)$. We *predict* with Elect that p_i sends the value h to process p_{i+1} in an `elect` message, receives a value h' from process p_{i-1} as an `elect` message, and repeats this process as long as $n < N$ while considering $h \leftarrow max(h, h')$. After N rounds we predict the leader p_j to broadcast its rank by sending a $\texttt{lead}\langle j\rangle$ message and all other processes p_i to receive the $\texttt{lead}\langle j\rangle$ message.

By observing the code, a leader is elected by circulating all values v_i through the ring topology in N rounds, hence the check at line 10. In each round, every process p_i receives a message $\texttt{elect}\langle h'\rangle$ from p_{i-1} (line 8), where h' is the maximum value encountered by p_{i-1}. In turn, p_i send its highest encountered value h to p_{i+1} (line 7). Finally, h is updated (line 9) to remain the highest encountered value before starting the next round (line 10). After N rounds, each process p_i invokes the function $\texttt{chooseLeader}(h, v)$. If $h = v$, then p_i is the leader due to the uniqueness of the values v_i. The leader p_j broadcasts its rank j to all other processes via a $\texttt{lead}\langle j\rangle$ message (line 7). All other processes receive the rank of the leader (line 9). Finally, all processes synchronise by entering a barrier.

The `election` function is proven via the following Hoare triple: $\{0 \le n \le N \land v \le h \land \mathsf{Elect}(i, v, h, n) \cdot \mathsf{F}\}p_i.\texttt{election}(h, v, n)\{\mathsf{F}\}$, with p_i the executing process. Similarly, the `chooseLeader` function is proven via the triple: $\{h \ge v \land \mathsf{Choose}(i, h, v) \cdot \mathsf{F}\}p_i.\texttt{chooseLeader}(h, v)\{\mathsf{F}\}$. After that, we use mCRL2 to reason about the global state and thereby reason about program executions in a global setting. For example, we may verify that the invocation $\texttt{chooseLeader}(h, v_i)$ receives a parameter $h \in \mathbb{Z}$ such that $h = max\{v_0, \ldots, v_{N-1}\}$ and $h = v_j$ for exactly one $j \in \mathscr{R}$. In particular, we verify that the leader p_j eventually broadcasts the message $\texttt{lead}\langle j\rangle$ containing its rank, and that all other processes eventually receive that message. In that case, all processes receive the rank of the leader and communicate correctly according to the predicated future.

6 Conclusion and future work

The work described in this paper is still in its early stages. We have already manually worked out a couple of verification examples, including the leader election protocol described in Section 5. We are currently working on providing tool support by extending the VerCors toolset. In particular, we define new Hoare triples to handle futures in combination with the abstract MPI actions described in Section 3. Moreover, we are extending the network environment to support more MPI functions, like: `MPI_Scatter`, `MPI_Gather`, and `MPI_Reduce`, which are focused on distributing data in a specific way. After extending

VerCors we plan to make a connection with mCRL2 for future-based analysis of the global system consisting of N instances of the program. Finally, we plan to show correctness of some industrial message passing programs in a case study.

Acknowledgements. Oortwijn is funded by the NWO TOP project VerDi (projectnr. 612.001.403). Blom and Huisman are funded by the ERC 258405 VerCors project.

References

[1] *mCRL2: Analysing System Behaviour*. Available at `http://www.mcrl2.org`.

[2] *MPJ Express*. Available at `http://mpj-express.org`.

[3] A. Amighi, S. Blom, S. Darabi, M. Huisman, W. Mostowski & M. Zaharieva-Stojanovski (2014): *Verification of Concurrent Systems with VerCors*. In: *SFM: Executable Software Models*, LNCS, vol. 8483, Springer, Heidelberg, pp. 172–216, doi:10.1007/978-3-319-07317-0_5.

[4] K.R. Apt & D.C. Kozen (1986): *Limits for Automatic Verification of Finite-State Concurrent Systems*. *Information Processing Letters* 22(6), pp. 307–309, doi:10.1016/0020-0190(86)90071-2.

[5] S. Blom & M. Huisman (2014): *The VerCors Tool for verification of concurrent programs*. In: *FM 2014: Formal Methods*, LNCS, vol. 8442, Springer, Heidelberg, pp. 127–131, doi:10.1007/978-3-319-06410-9_9.

[6] J.F. Groote & M.R. Mousavi (2014): *Modeling and Analysis of Communicating Systems*. MIT Press.

[7] R.H. Halstead (1985): *Multilisp: A Language for Concurrent Symbolic Computation*. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7(4), pp. 501–538, doi:10.1145/4472.4478.

[8] Y. Hanna, S. Basu & Rajan H. (2009): *Behavioral Automata Composition for Automatic Topology Independent Verification of Parameterized Systems*, pp. 325–334. doi:10.1145/1595696.1595758.

[9] T. Hoare & J. Misra (2008): *Verified Software: Theories, Tools, Experiments Vision of a Grand Challenge Project*. In: *Verified Software: Theories, Tools, Experiments (VSTTE)*, LNCS, vol. 4171, Springer, Heidelberg, pp. 1–18, doi:10.1007/978-3-540-69149-5_1.

[10] T. Hoare & P.W. O'Hearn (2008): *Separation Logic Semantics for Communicating Processes*. In: *First International Conference on Foundations of Informatics, Computing and Software (FICS)*, ENTCS, Sci. 212, Elsevier, pp. 3–25, doi:10.1016/j.entcs.2008.04.050.

[11] K. Honda, E. Marques, F. Martins, N. Ng, V.T. Vasconcelos & N. Yoshida (2012): *Verification of MPI Programs using Session Types*. In: *EuroMPI'12*, LNCS, vol. 7940, Springer, pp. 291–293, doi:10.1007/978-3-642-33518-1_37.

[12] B. Liskov & L. Shrira (1988): *Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems*. *PLDI*, doi:10.1145/960116.54016.

[13] R. Milner (1980): *A Calculus of Communicating Systems*. LNCS, vol. 92, Springer-Verlag, Berlin, Germany, doi:10.1007/3-540-10235-3.

[14] P.W. O'Hearn (2007): *Resources, concurrency, and local reasoning*. *Theoretical Computer Science* 375(1), pp. 271–307, doi:10.1016/j.tcs.2006.12.035.

[15] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. Kirby & R. Thakur (2009): *Formal Verification of Practical MPI Programs*. In: *PPOPP*, vol. 44, ACM, New York, pp. 261–270, doi:10.1145/1594835.1504214.

[16] M. Zaharieva-Stojanovski (2015): *Closer to Reliable Software: Verifying Functional Behaviour of Concurrent Programs*. CTIT Ph.D. Thesis Series No. 15-375, University of Twente, doi:10.3990/1.9789036539241.