Static Verification of Message Passing Programs

Wytse Oortwijn, Stefan Blom, and 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

Keywords: static verification, message passing interface, process algebra, parameterised model checking, distributed systems

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 message exchanges may occur in any order. Developing correct message passing programs is therefore very challenging, which makes proving their correctness crucial [4].

A popular API for implementing message passing programs is the Message Passing Interface (MPI) library. We focus on the modular verification of MPI programs. Related work mainly focuses on communication and considers abstract models rather than concrete implementations [10,5]. We consider concrete Java code and combine static verification with well-known techniques for reasoning about concurrent and distributed programs, based on process algebras [7].

Overview. Global system correctness depends on the correctness of the individual processes and their interactions. We use separation logic to reason about local correctness. In addition, we algebraically model the communication protocol in the μ-calculus. We refer to these algebraic terms as *futures* as they predict how components interact during program execution [11]. We establish a provable link between futures and program code so that reasoning about futures corresponds to reasoning about the concrete program. Analysis of futures can then be reduced to model checking to allow reasoning about functional and global system correctness. Since model checkers inherently target finite-state systems, we use abstraction and cut-offs to reason about programs with infinite behaviour.

Our approach. MPI programs assume a fixed number of processes $p_1, \ldots, p_N$ distributed over one or more devices connected via a network. Each process p_j executes the same program in which j is called the *rank* and N the *size*. Processes maintain a private memory and accesses to memories of other processes are solely handled via message exchanges. Verifying MPI programs involves verifying functional and communicational correctness for *every* rank and size.

We use permission-based separation logic [8] to reason about local behaviour of MPI processes, i.e. without considering message passing. Permissions are included to avoid data-races since processes may spawn threads during execution.

To reason about global correctness we algebraically model the program's communication protocol with futures [11]. We extend the μ-calculus with special actions like: **send**, **recv**, and **bcast** that model the semantics of the concrete MPI operations. The communication protocol can then be modelled via these abstract actions, as is done in verification with Session Types [6].

We will extend the VerCors toolset [2,1] for verifying local behaviour and finding a correspondence between futures and program code. We use the mCRL2 toolset [3] to reason about futures and thereby making a reduction to parameterised model checking [9]. 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. Other interesting properties to check are: deadlock freedom, resource leakage, and global termination.

```
requires $0 \leq n \leq N \wedge \text{Future}(\mathsf{S}(n,t).\mathsf{F})$
ensures $\text{Future}(\mathsf{F})$
process Server(int $n$, int $t$):
  if $n < N$
    int $x \leftarrow$ mpi_recv($*$)
    Server($n+1$, $t+x$)
  else mpi_bcast($t$)
```

```
requires $\text{Future}(\mathsf{C}(v).\mathsf{F})$
ensures $\text{Future}(\mathsf{F})$
process Client(int $v$):
  mpi_send($v$, $Server$)
  int $sum \leftarrow$ mpi_recv($Server$)
```

Fig. 1: A simple annotated client/server program with simplified MPI syntax.

Example program. We now illustrate our approach on a small example with one server and several clients. All clients send an integer to the server. The server sums up and broadcasts all received values. Figure 1 shows annotated pseudocode.

Initially, `Server` starts with $n = 0$ and $t = 0$ and we *predict* a behaviour as specified by the future $\mathsf{S}(0,0)$, where $\mathsf{S}(n,t) \equiv \mathbf{recv}(x).\mathsf{S}(n+1,t+x)$ and $\mathsf{S}(N,t) \equiv \mathbf{bcast}(t)$. Similarly, we predict that `Client`(v) behaves as specified by the future $\mathsf{C}(v)$, where $\mathsf{C}(v) \equiv \mathbf{send}(v).\mathbf{recv}(t)$. The specifications also show a future F, which is the future of the process that invoked `Server` or `Client`.

When invoking `Server` its precondition must hold, i.e. $0 \leq n \leq N$ with future $\mathsf{S}(n,t).\mathsf{F}$. We extend the VerCors toolset to verify that `Server` satisfies its contract and correctly executes from the given future. For example, when `Server` receives its first value v it recursively calls `Server` with future $\mathsf{S}(1,v).\mathsf{F}$. Finally, after receiving all $N-1$ values, `Server` should broadcast t as it has future $\mathsf{S}(N,t).\mathsf{F}$ and terminates after performing **bcast**. The `Client` process is verified in a similar way. Observe that the abstract **send**, **recv**, and **bcast** actions correspond to the concrete MPI functions.

After showing that `Server`$(0,0)$ conforms to $\mathsf{S}(0,0)$ and `Client`(v_i) conforms to $\mathsf{C}(v_i)$ for each client i, we use mCRL2 to reason about the global system with one server and $N-1$ clients. In particular, for any value N we verify that $\mathsf{S}(0,0)$ eventually performs $\mathbf{bcast}(t)$ and that every $\mathsf{C}(v_i)$ eventually performs $\mathbf{recv}(t)$, with t the sum of all values v_i. In that case, the server receives values from all clients and all processes communicate correctly. Note that we also worked out more involved examples, including a leader election protocol and mergesort.

References

1. A. Amighi, S. Blom, S. Darabi, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski. Verification of Concurrent Systems with VerCors. In *Formal Methods for Executable Software Models*, pages 172–216. Springer, 2014.
2. S. Blom and M. Huisman. The VerCors Tool for verification of concurrent programs. In *FM 2014: Formal Methods*, pages 127–131. Springer, 2014.
3. J.F. Groote and M.R. Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014.
4. T. Hoare and J. Misra. Verified Software: Theories, Tools, Experiments Vision of a Grand Challenge Project. In *Verified Software: Theories, Tools, Experiments*, pages 1–18. Springer, 2008.
5. T. Hoare and P.W. O'Hearn. Separation logic semantics for communicating processes. In *1st FICS conference, ENTCS*, volume 212, pages 3–25. Elsevier, 2008.
6. K. Honda, E. Marques, F. Martins, N. Ng, V.T. Vasconcelos, and N. Yoshida. Verification of MPI Programs using Session Types. 2012.
7. R. Milner. A Calculus of Communicating Systems. 1980.
8. P.W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1):271–307, 2007.
9. J. Pang, J. van de Pol, and M. Espada. Abstraction of Parallel Uniform Processes with Data. In *Software Engineering and Formal Methods*, pages 14–23. IEEE, 2004.
10. A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. Kirby, and R. Thakur. Formal Verification of Practical MPI Programs. In *PPOPP*, pages 261–270. ACM, 2009.
11. M. Zaharieva-Stojanovski. Closer to Reliable Software: Verifying Functional Behaviour of Concurrent Programs, PhD Thesis. 2015.