Formal Specification of the MPI-2.0 Standard in TLA+

Guodong Li, Michael Delisi, Ganesh Gopalakrishnan, Robert M. Kirby
School of Computing, University of Utah
{ligd, delisi, ganesh, kirby}@cs.utah.edu

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods

General Terms Verification

Keywords MPI, Formal Specification, TLA+, Model Checking

1. Introduction

Parallel programs will increasingly be written using complex APIs such as MPI-2.0 [1], OpenMP [2], PThreads, etc. It is well documented that even experienced programmers misunderstand these APIs, especially when they are described in natural languages. The behavior of APIs observed through ad hoc experiments on actual platforms is not a conclusive or comprehensive description of the standard. Formal specifications, as currently written and distributed, are inaccessible to most practitioners.

In our previous work [4], we presented the formal specification of around 30% of the MPI-1.0 primitives (mainly for point-to-point communication) in TLA+ [6]. TLA+ enjoys wide usage within (at least) Microsoft and Intel by engineers (e.g., [5]). To make our specification accessible to practitioners, we built a C front-end in the Microsoft Visual Studio (VS) parallel debugger environment through which users can submit short (perhaps tricky) MPI programs with embedded assertions, called litmus tests. This input is turned into TLA+ code and run through the TLC model checker [6]. This permits practitioners to play with (and find holes in) the semantics. We also maintain cross-reference tags with the MPI reference document for traceability.

While we have demonstrated the merits of our previous work ([4]), this paper handles far more details including those pertaining to data transfers. In this work, we have covered much of MPI-2.0 (has over 300 API functions, as opposed to 128 for MPI-1.0). In addition, our new work provides a rich collection of tests that help validate our specifications. It also modularizes the specification, permitting reuse. The approximate sizes (without including comments and blank lines) of the major parts in the current TLA+ specification are shown in Table 1. We do not plan to model primitives whose behavior depends on the underlying operating system, The framework plus our TLA+ models can be downloaded from [3].

In [4], we describe an execution environment into which users can supply litmus tests and calculate their outcomes directly based on the formal semantics. The availability of a formal specification also has enabled us to create an efficient dynamic partial order reduction algorithms [8].

Main Module	**#primitives(#lines)**
Point to Point Communication	35(800)
Userdefined Datatypes	27(500)
Group and Communicator Management	34(650)
Intracommunicator Collective Communication	16(500)
Topology	18(250)
Environment Management in MPI 1.1	10(200)
Process Management	10(250)
One sided Communication	15(550)
Intercommunicator Collective Communication	14(350)
I/O	50(1100)
Interfaces and Environments in MPI 2.0	35(800)

Table 1. Size of the Specification

TLA+ is a formal specification language based on (untyped) ZF set theory [6]. TLC, a model checker for TLA+, explores all reachable states in the model, looking for a state where (a) an invariant is not satisfied, (b) there are no exits (deadlocks), (c) the type invariant is violated, or (d) a user-defined TLA+ assertion is violated. When TLC detects an error, a minimal-length trace that leads to the bad state is reported (which our framework turns into a VisualStudio debugger replay of the C source).

2. Specification

We define a formal semantics for MPI primitives where execution is modeled by state transitions. The specification starts by defining advanced data structures including array, map, and ordered set to model MPI objects. For instance, MPI groups and I/O files are represented as ordered sets. A system state consists of explicit and opaque objects that may be shared among all processes or be associated with individual processes. Virtual program counters of programs/processes are also introduced to identify the current execution point. In a system state (E, G, pc), E maps a process to its local environment, G maps a global variable to its value, and pc maps a process to its current program counter. A map $X * y$ is interpreted as separating conjunction (similar to separation logic [7]) such that y should be considered separately from the elements in X. This enables reasoning about mixtures of the changed portion y and unchanged portion X of the map.

Notation	**Meaning**
$X * (i \to v)$	a conjunction map containing a value v at address x
$c \; ? \; x \; : \; y$	the value of `if` c `then` x `else` y
`next`	a function returning the next label in the control flow
`op` l v	the value returned by applying primitive l to value v
$\mathring{p}$	the rendezvous for the communicator p belongs to
$\vec{p}$ and $\hat{p}$	an outgoing queue and an incoming queue of p
τ and Ψ	the status and the participants of a synchronization
$\Gamma; v$	appending value v into queue Γ

Table 2. Notations used in the semantics rules

Using the notations in Table 2, the following rules summarize the semantics of MPI primitives. A local primitive at process p only

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.
ACM 978-1-59593-960-9/08/0002.

updates p's local environment. A synchronizing collective primitive is implemented by a loose synchronization protocol: in the first "init" phase, p will proceed to its next "wait" phase provided that it hasn't participated in the current synchronization (say `S`) and S's status is either 'e' ("entering") or 'v' ("vacant"). Note that if all expected processes have participated then `S`'s status will advance to 'l' ("leaving"). In the "wait" phase, p is blocked if `S` is not leaving or p has left. The last leaving process will reset `S`'s status to be "vacant". A similar protocol is defined for collective primitives that do not enforce synchronization. .

$$\frac{(E * (p{\to}v),\ G,\ pc * (p{\to}l)) \wedge w \doteq \texttt{op}\ l\ v}{(E * (p{\to}w),\ G,\ pc * (p{\to}\texttt{next}\ l))}\ \text{local}$$

$$\frac{\begin{array}{c}(E,\ G * (\mathring{p} \to (\tau, \Psi)),\ pc * (p \to l_{init})) \ \wedge\ p \notin \Psi \ \wedge \\ \tau \notin \{\text{`}e\text{`}, \text{`}v\text{`}\} \ \wedge\ \tau' \doteq (\Psi \cup \{p\} = \mathring{p}.group)\ ?\ \text{`}l\text{`}\ :\ \tau\end{array}}{(E, G * (\mathring{p}{\to}(\tau', \psi \cup \{p\})), pc * (p{\to}l_{wait}))}\ \text{syn}_{init}$$

$$\frac{\begin{array}{c}(E,\ G * (\mathring{p} \to (\text{`}l\text{`}, \Psi)),\ pc * (p \to l_{wait})) \ \wedge\ p \in \Psi \ \wedge \\ \tau' \doteq (\psi \setminus \{p\} = \{\})\ ?\ \text{`}v\text{`}\ :\ \tau\end{array}}{(E, G * (\mathring{p}{\to}(\tau', \psi \setminus \{p\})), pc * (p{\to}\texttt{next}\ l_{wait}))}\ \text{syn}_{wait}$$

$$\frac{(E * (x \to v),\ G * (\vec{p} \to \Gamma),\ pc * (p \to l))}{(E * (x{\to}v), G * (\vec{p}{\to}\Gamma;(d,v)), pc * (p{\to}\texttt{next}\ l))}\ \text{asyn}_{send}$$

$$\frac{(E,\ G * (\vec{p} \to \Gamma),\ pc * (p \to l))}{(E * (x{\to}v), G * (\vec{p}{\to}\Gamma;(d,v)), pc * (p{\to}\texttt{next}\ l))}\ \text{asyn}_{recv}$$

$$\frac{(E,\ G * (\hat{p} \to (s, v); \Gamma),\ pc * (p \to l))}{(E * (x{\to}v), G * (\hat{p}{\to}\Gamma), pc * (p{\to}\texttt{next}\ l))}\ \text{asyn}_{wait}$$

$$\frac{\begin{array}{c}(E,\ G * (\vec{p} \to \Gamma_1^p; (q, v); \Gamma_2^p) * (\vec{q} \to \Gamma_1^q; (p, \_); \Gamma_2^q),\ pc) \\ \wedge\ \forall (d, w) \in \Gamma_1^p\ :\ d \neq q\ \wedge\ \forall (s, w) \in \Gamma_1^q\ :\ s \neq p\end{array}}{(E, G * (\hat{p}{\to}\Gamma_1^p;\Gamma_2^p) * (\vec{q}{\to}\Gamma_1^q;(p,v);\Gamma_2^q), pc)}\ \text{transfer}$$

An asynchronous protocol is used to implement point-to-point and one-sided communication. When sending a data v, process p posts a message of format $(destination, data)$ in an outgoing FIFO queue $\vec{p}$. There are multiple queues for each process, and the type of communication determines which queue should be used. In order to receive a message, p posts a request of format $(source, \_)$ in its incoming queue $\hat{p}$, with $\_$ indicating that the data is still missing. When this $\_$ is filled by an incoming message, p can fetch it and updates the local environment accordingly. It is the rule `transfer` that models the message passing mechanism: if a message in p's outgoing queue matches a request in q's incoming queue, then the data is transferred. By matching we require: (1) the source and destination of the message and the request should match; so do their communication contexts and tags. (2) messages from the same source to the same destination should be matched in a FIFO order. A non-blocking primitive is implemented as an asynchronous operation, while a blocking operation is implemented as an asynchronous operation followed immediately by a wait operation.

As an illustration, the TLA+ implementation of the syn_{wait} rule is shown below, where comments are shaded. Note that expression $s' = f\ s$ defines a transition from a state s to a new state s'; and f EXCEPT $!i = v$ defines a new map (i.e. function) $s * (i \to v)$. Notation @ in $[r$ EXCEPT $!.x = @]$ stands for the (old) value of field $r.x$. Clearly this code is more readable and comprehensible than its corresponding semantics rule.

```
syn_wait(comm, p) ==
 LET i == comm.cid IN
 ∧ rendezvous[i].state = "leaving"   ṗ → (‘l‘, Ψ)
 ∧ p ∈ rendezvous[i].participants    p ∈ Ψ
 ∧ rendezvous' =
   LET members == rendezvous[i].participants \ {p} IN
      [rendezvous EXCEPT ![i] =
        [@ EXCEPT
           !.participants = members,   ψ \ {p}
           !.state = IF members = {} THEN "vacant"
                     ELSE @   (ψ \ {p} = {}) ? ‘v‘ : τ
        ]
      ]
 ∧ changed({rendezvous_id})
```

We provide comprehensive unit tests and a rich set of short litmus tests of the specification. Generally it suffices to test Local, collective, and asynchronous primitives on one, two and three processes respectively.

Our modeling framework uses the Microsoft Phoenix compiler as a front end for C programs. From Phoenix intermediate representation (IR) we build a state-transition system by converting control flow graph into TLA+ relations and mapping MPI primitives to their names in TLA+. This transition system will capture completely the control skeleton of the input MPI program. Assignments are modeled by their effect on the memory. Jumps have standard transition rules modifying the values of the program counters.

3. Concluding Remarks

Often our formal specifications end up resembling programs written using detailed data structures, *i.e.* they are not as "declarative" as we like. We believe that this is in some sense inevitable when dealing with real world APIs. Even so, TLA+ based "programs" can be considered superior to executable models created in C: (i) the notation has a precise semantics, as opposed to C/PThreads, (ii) another specification in a programming language can provide complementary details, (iii) in our experience, there are still plenty of short but tricky MPI programs that can be executed fast in our framework. In future, we hope to write general theorems (inspired by our litmus tests), and establish them using the Isabelle theorem prover that has a tight TLA+ integration.

References

[1] The Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/.

[2] www.openmp.org.

[3] www.cs.utah.edu/formal_verification/ppopp08-mpispec/.

[4] Robert Palmer, Michael Delisi, Ganesh Gopalakrishnan and Robert M. Kirby. An Approach to Formalization and Analysis of Message Passing Libraries, Formal Methods for Industry Critical Systems (FMICS), FLoC, Berlin, 2007.

[5] Leslie Lamport, The Win32 Threads API Specification. research.microsoft.com/users/lamport/tla/threads/threads.html

[6] Leslie Lamport, research.microsoft.com/users/lamport/tla/tla.html

[7] John Reynolds. Separation logic: A logic for shared mutable data structures. LICS, 55–74, 2002.

[8] Salman Pervez, Robert Palmer, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp, Practical Model Checking Method for Verifying Correctness of MPI Programs, *EuroPVM/MPI*, 344–353, 2007.