Formal Operational Semantics Guided Model Checking and Active Testing for High Performance Computing

[Extended Abstract]

Charlie Jacobsen
University of Utah
50 S Central Campus Drive
Salt Lake City, Utah
charlie.jacobsen@utah.edu

Ganesh Gopalakrishnan (advisor)
University of Utah
50 S Central Campus Drive
Salt Lake City, Utah
ganesh@cs.utah.edu

ABSTRACT
Programs written for High Performance Computing (HPC) systems can be difficult to analyze due to the complicated semantics and nondeterminism of such systems. Analysis becomes especially difficult when multiple high performance libraries are used in the same program (e.g., MPI and CUDA). How does one verify that these programs are bug free? Implementing model checking algorithms for HPC systems also becomes more difficult as these algorithms may depend on the system’s runtime semantics. Powerful tools are available for expressing the behavior and state evolution of HPC systems using operational semantics and producing an executable model for running and testing programs. We present a technique for formalizing a small subset of MPI using an executable semantics framework and using this model in an implementation of ISP, a state-space exploration tool for MPI programs.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification—formal methods, model checking; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—operational semantics

General Terms
Verification

1. ALL (WILL BE BROKEN DOWN)
Programs that use high performance computing systems are difficult to debug and analyze. These systems can have hundreds of independent components operating concurrently and interacting in subtle ways, and programs can follow numerous execution paths. For example, when a program that uses MPI and CUDA is executed, the nodes in the cluster and the threads on each node’s GPU can all interact. Events may occur in a different order each time due to asynchronous data transfer and instruction execution, and the program may behave differently on various platforms.

Accurate, fine-grained testing for such programs is more complex. If building a model, the analyst has to incorporate the semantics of the high performance library in the model in order to accurately detect errors that are dependent on the system’s runtime behavior, such as deadlocks, data races, and buffer overflows. Some system-specific tools are available with the semantics of the system’s runtime built in. For example, MPI-SPIN is an extension of the SPIN model checking tool that includes features for modeling and verifying MPI programs [6, 3].

ISP is a state-space exploration tool for verifying MPI programs [7]. It efficiently explores all possible non-equivalent interleavings of transitions the MPI runtime can take when a program is executed. The scheduler in the implementation of ISP prioritizes transitions and executes those with a higher priority while adhering to the MPI standard. As a result, a large part of the scheduler contains the semantics of the MPI runtime.

The semantics of runtimes for high performance libraries can be clearly and accurately described using semantic engineering tools. Operational semantics describes how a program is executed using formal evaluation rules: how instructions are ordered and how that order can be manipulated with control statements, how the state of the system evolves and how the state can be updated, etc. Rules intentionally left missing or under-specified can make some aspects of the system nondeterministic. Most formal semantics are paper-only documents, but several tools are available for expressing the syntax and evaluation rules for a programming language. The tool can then produce a prototype module that can simulate the system’s behavior by executing the rules. K, Maude, and PLT-Redex are a few of the tools available [5, 1, 2].

Our goal is to build an isolated model of MPI using one of these semantics tools and utilize the executable model the tool produces in the ISP algorithm. Related work includes the 4M model and testing architecture for MCAPI programs.
We used the K framework for writing formal term rewriting rules for the MPI runtime model and the ISP algorithm. The K module for the MPI runtime model provides a ‘library’ of syntax and rules for parsing an MPI program, maintaining the state of the MPI runtime, calculating which transitions are possible from the current state, effecting a chosen transition, and reporting whether the model is deadlocked. The K modules for the ISP algorithm and front end contain the state-space exploration algorithm, and they use the MPI K module for tracking the state of the MPI runtime and for certain steps in the exploration algorithm.

Except for some parts of the front end, we used syntactic lists, maps, and other structures, using a LISP style, rather than K ‘configurations’. K configurations are for easily representing the system’s state and how the state evolves, perhaps nondeterministically. We set out to use configurations for the MPI runtime model; but we found it difficult to calculate all possible transitions the model could take and to eliminate unwanted nondeterminism.

The entire ISP-MPI package can read in a simple MPI program and analyze it using the state-space exploration algorithm. We found it was accurate but executed slowly on non-trivial example programs. This could be due to our inexperience with semantics engineering tools like K. Indeed, one rule we mistakenly wrote lead to exponential evaluation behavior. A simple changed fixed it.

We found semantics engineering tools to be accurate and effective means for modeling a complicated high performance system. We are continuing to use apply this technique for creating new model checking and active testing tools for HPC.

2. REFERENCES


