Symbolically Modeling Concurrent MCAPI Executions *

Topher Fischer
Brigham Young University
javert42@cs.byu.edu

Eric Mercer
Brigham Young University
egm@cs.byu.edu

Neha Rungta
NASA Ames Research Center
neha.s.rungta@nasa.gov

Abstract

Improper use of Inter-Process Communication (IPC) within concurrent systems often creates data races which can lead to bugs that are challenging to discover. Techniques that use Satisfiability Modulo Theories (SMT) problems to symbolically model possible executions of concurrent software have recently been proposed for use in the formal verification of software. In this work we describe a new technique for modeling executions of concurrent software that use a message passing API called MCAPI. Our technique uses an execution trace to create an SMT problem that symbolically models all possible concurrent executions and follows the same sequence of conditional branch outcomes as the provided execution trace. We check if there exists a satisfying assignment to the SMT problem with respect to specific safety properties. If such an assignment exists, it provides the conditions that lead to the violation of the property. We show how our method models behaviors of MCAPI applications that are ignored in previously published techniques.

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

General Terms Languages, Verification

Keywords MCAPI, Symbolic Analysis, Multicore, SMT

1. Introduction

With the growing availability of multicore processors has come an increase in the amount and complexity of concurrent software. The non-deterministic interleaving of threads within a concurrent program can lead to bugs that are very difficult to discover. Misuse of Inter-Process Communication (IPC) within concurrent software creates data races that can then lead to errors within a system. Because the manifestations of the data races depend on the whims of the operating system's scheduler and delays in message transmission, error states that are the result of data races can be very difficult to produce, reproduce, and identify with ad-hoc testing methods.

Message passing is a form of IPC that involves defining message endpoints that provide different threads the ability to send data to specific endpoints. The Multicore Communications API (MCAPI) is a new message passing API that was released by an organization known as the Multicore Association [4]. MCAPI provides an interface designed for embedded systems that allows communication at different system levels. For example, MCAPI can be implemented in both hardware and software so that an application running in a Linux environment can communicate directly with a graphics processing unit (GPU) and a digital signal processor (DSP).

The semantics of message passing applications are difficult to model. Data non-determinism in message passing applications is expressed in instances where a single receive operation could receive a message from any number of send operations. Which message is received depends on process scheduling and the length of delays in message transmission. The two previously proposed methods for formal MCAPI application verification do not correctly model all the possible behaviors of MCAPI applications. This work specifically overcomes those limitations.

MCC (MCAPI Checker) is a model checker specifically written to verify MCAPI applications [5]. Although MCC represents important progress in the verification of MCAPI applications, it ignores certain behaviors of MCAPI applications because it is not able to consider non-deterministic delays in the communication network when sending messages from two different threads to a common endpoint. Ignoring these behaviors prevents MCC from performing a complete verification of MCAPI applications.

Wang et al. recently described an SMT-based model checking framework called Fusion [6]. Fusion employs a property-driven reduction technique for the verification of concurrent applications that use shared memory. In a series of benchmark tests it was shown that Fusion had significantly shorter runtimes than a comparable tool, Inspect, that uses dynamic partial-order reduction (DPOR) [3, 7]. The impressive results of Fusion were the inspiration for this work to generate SMT problems to model MCAPI applications.

A very closely related work attempts to model MCAPI application executions as SMT problems [2]. To the best of our knowledge this work ignores potential delays in the MCAPI communication network. By ignoring these possible delays this work does not properly model certain MCAPI application behaviors. As with MCC, by ignoring these possible behaviors the method defined in [2] cannot perform a complete verification of MCAPI applications.

We have defined a new method for modeling MCAPI application executions as SMT problems in a manner that symbolically models all possible concurrent executions and follow the same sequence of conditional branch outcomes as the provided execution trace. Our method captures all possible behaviors of the system, and it is much simpler than the method presented in [2]. We briefly describe our method in the following section.

2. Modeling MCAPI Executions

As an illustration we present an abstraction of a very simple MCAPI application in Figure 1. The $send$ calls are used to send messages to specified threads. A thread can obtain the message by calling $recv$. When multiple threads send messages to a single endpoint no order of messages is guaranteed when the receiving thread calls the $recv$ function. For example, since both $send$ calls

* This work is supported by NSF CCF-0903491 and SRC 2009-TJ-1994.

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.
ACM 978-1-4503-0119-0/11/02.

	Thread t_0	Thread t_1	Thread t_2
1:	recv(A)	recv(C)	send(Y):t_0
2:	recv(B)	send(X):t_0	send(Z):t_1

Figure 1: Simplified MCAPI program.

sending to thread t_0 can occur before thread t_0 is executed, either message could be received when thread t_0 calls $recv$ on line 1.

The process of creating an SMT problem to model an MCAPI application execution is started by generating an arbitrary execution trace through the program and then analyzing the trace to generate a set of possible matching send operations for each receive operation in the trace. A pair of send and receive operations *match* when the receive operation receives the message that was sent by the send operation. We call this pair of operations a "match pair".

1: $\mathcal{P}_{MatchPairs} = true$
2: **for each** $recv \in MatchPairs$ **do**
3: $\quad tmp := false$
4: $\quad$ **for each** $send \in \mathsf{getSends}(recv)$ **do**
5: $\quad\quad tmp := tmp \vee \mathsf{match}(recv, send)$
6: $\quad \mathcal{P}_{MatchPairs} := \mathcal{P}_{MatchPairs} \wedge (tmp)$

Figure 2: Match pair encoding algorithm.

Our trace analysis generates a set $MatchPairs$ that contains every receive operation in the trace, and it also provides a function getSends that maps each receive operation to the set of all the send operations that it could match with. This set and function are used in the simple algorithm shown in Figure 2 that encodes all possible match pairs in the execution trace being modeled. The inner loop of this algorithm constructs a disjunction, in the variable tmp, of all the send operations that a specific receive operation could pair with. The outer loop forms a conjunction of the disjunctions created for each of the receive operations in the trace. The result is a boolean representation of every possible match pair discovered in the trace analysis. The trace analysis also assigns each send operation a unique identifier for use in the SMT problem. Each receive operation is associated with an unbound identifier variable that an SMT solver will attempt to assign with the value of a single send operation's identifier. If such an assignment can occur, given the constraints of the problem, it signifies a match between the send and receive operations.

1: $\mathcal{P}_{Unique} = true$
2: **for each** $recv_i \in \{0 \ldots |Recvs|\}$ **do**
3: $\quad$ **for each** $recv_j \in \{i+1 \ldots |Recvs|\}$ **do**
4: $\quad\quad \mathcal{P}_{Unique} := \mathcal{P}_{Unique} \wedge \mathsf{isDiffSend}(recv_i, recv_j)$

Figure 3: Uniqueness assertion algorithm.

The match function on line 5 of Figure 2 is a simple function for use in the SMT model. Given a receive and a send operation the match function asserts that the call to send occurs before the call to receive, the message sent is the same message that is being received, and that the identifiers of the two operations are equal. MCAPI also provides a non-blocking receive operation along with a wait operation that blocks until the associated non-blocking call has completed. In the case of a non-blocking receive, the match function asserts that the call to send occurs before the call to the wait operation that is associated with the receive. Checking the identifiers assures that a specific receive operation is matched with a specific send operation, and that each receive operation is paired with a different send operation. To assert this we use the algorithm in Figure 3. The results from the two algorithms are conjuncted with assertions on program order ($\mathcal{P}_{Order}$), negated properties that define a correct system ($\neg\mathcal{P}_{Prop}$), and other events in the execution ($\mathcal{P}_{Events}$). The result is an SMT problem that models possible executions of an MCAPI application.

$t_2 : send(Y) \rightarrow t_0 : recv(A)$
$t_2 : send(Z) \rightarrow t_1 : recv(C)$
$t_1 : send(X) \rightarrow t_0 : recv(B)$

(a) Possible send/recv Pairings.

$t_2 : send(Z) \rightarrow t_1 : recv(C)$
$t_1 : send(X) \rightarrow t_0 : recv(A)$
$t_2 : send(Y) \rightarrow t_0 : recv(B)$

(b) Possible send/recv Pairings.

Figure 4: Possible Pairings of send/recv calls from Figure 1.

$$\mathcal{P} = \mathcal{P}_{Order} \wedge \mathcal{P}_{MatchPairs} \wedge \mathcal{P}_{Unique} \wedge \neg\mathcal{P}_{Prop} \wedge \mathcal{P}_{Events}$$

The SMT problem can be solved by a constraint solver such as Yices [1]. As the system properties are negated, if the SMT problem is found to be satisfiable then a property violation is possible in the application under test. A simple analysis of the set of satisfying assignments provides a description of the path to the error state.

As stated previously, the technique used in MCC does not consider non-deterministic delays that can occur when two messages are sent from different threads to a common endpoint. If applied to the scenario in Figure 1, MCC would only explore the behavior shown in Figure 4a. It would not consider that the message from send(Y) in thread t_2 could delay so long that the recv(A) operation would receive its message from send(X) in thread t_1. Besides presenting an unnecessarily complicated encoding method, the method presented in [2] also ignores the behavior represented in Figure 4b.

3. Results and Future Work

We have formally defined the semantics of a relevant subset of the MCAPI API in the Redex rewrite language. Based on these semantics we have created a tool that takes as input a trace, a set of match pairs, and a set of correctness properties. The output is an SMT problem that accurately models all possible executions of the trace. A precise set of match pairs can be generated through a depth-first abstract execution of the trace. Though precise, this method can be prohibitively expensive in computation time. As future work we plan to define a method for generating a reasonable over-approximation of the match-pair set, and to define a method for using the over-approximated set of match pairs to generate an SMT problem that models all possible behaviors of the trace.

References

[1] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, volume 4144 of *LNCS*, pages 81–94, 2006.

[2] M. Elwakil and Z. Yang. Debugging support tool for mcapi applications. In *PADTAD*, 2010.

[3] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121. ACM, 2005.

[4] http://www.multicore association.org/workgroup/mcapi.php.

[5] S. Sharma, G. Gopalakrishanan, E. Mercer, and J. Holt. Mcc - a runtime verification tool for mcapi user applications. In *FMCAD*, 2009.

[6] C. Wang, S. Chaudhuri, A. Gupta, and Y. Yang. Symbolic pruning of concurrent program executions. In *ESEC/FSE*, pages 23–32, New York, NY, USA, 2009. ACM.

[7] Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A runtime model checker for multithreaded c programs. Technical report, 2008.