Toward Parameterized Verification of Synchronous Distributed Applications

Sagar Chaki  James Edmondson
Carnegie Mellon University, Pittsburgh, PA, USA
{chaki,jedmondson}@sei.cmu.edu

ABSTRACT
We present preliminary results on parameterized verification of distributed applications that assume a synchronous model of computation. Our theoretical results are negative—the problem is undecidable even if each node has a single bit of non-determinism and the property is a 1-index safety property. Further, even if each node is completely deterministic, and the property is again a 1-index safety, parameterized verification cannot be solved via the cutoff method. Empirically, we show how to encode such applications as Array-Based Systems and verify them using existing model checkers. We demonstrate this approach on protocols for distributed mutual exclusion and collision avoidance.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification

General Terms
Verification

Keywords
Parameterized verification, synchronous systems, array-based systems

1. INTRODUCTION
Distributed applications play crucial, often unseen, roles in our lives. There is also a growing need to incorporate them into safety-critical domains. For example, there are US mandates to incorporate communicating autonomous vehicles on roadways1. Indeed, researchers have already started developing intersection protocols [4] that rely on vehicle-to-vehicle communication. Bugs in such distributed protocols and applications have the potential to cause not only financial damage, but also injury and loss of human lives. Consequently, verifying the safety of such distributed protocols and applications before deployment in the real-world has tangible monetary and public safety benefits.

In this paper, we focus on the verification of synchronous distributed applications (SDAs). In such applications, each node executes in rounds, and messages sent (or variables written to) by a node in round $r$ are visible to other nodes in round $r + 1$. SDAs have been studied widely in the literature [13]. They are also easier to design and verify compared to asynchronous applications. For example, the use of PALS [1] — a “synchronizer” protocol in the hard real-time domain—has been shown to reduce verification time [14] of an avionics protocol from 35 hours to 30 seconds.

We assume shared memory based communication. Specifically, an SDA instance consists of $n \in \mathbb{N}$ nodes. Each node $N_i$ has access to its own unique id $i \in [1,n]$, and a local copy of an array $GV$ with $n$ elements. Each element of $GV$ is a bit-vector of known fixed width $W$. In each round $r$, $N_i$ computes a new value of $GV[i]$ based on the current value of all elements of $GV$. This value is propagated to the other copies of $GV$ prior to start of round $r + 1$ by the underlying communication infrastructure. Note that the element $GV[i]$ is modified only by node $N_i$. Initially, the first $Z$ (for some known $Z \in [0,W]$) bits of each array element are assigned non-deterministically, and the remaining bits are set to $\perp$.

As part of ongoing work [5], we have developed a domain-specific language, called DASL, for programming SDAs and safety requirements. We have also developed a verifying compiler for DASL programs that handles finite instantiations (i.e., when $n$ is fixed and known) using model checking. In this paper, we focus on the problem of parameterized verification of SDAs — i.e., proving their correctness for arbitrary number of nodes. We present the following results:

1. We show that parameterized verification of SDAs is undecidable, even when $Z = 1$, and the property is a 1-index safety property [8], i.e., of the form $\forall i : G(\phi(i))$. Thus, parameterized verification of synchronous systems is as hard as asynchronous ones even though non-parameterized verification is more tractable [14].

2. We show that even for $Z = 0$ (i.e., deterministic SDAs) and 1-index safety properties, parameterized verification cannot be solved via the cutoff method [8].

3. We present preliminary experimental results on verify-
ing SDAs by translating them into Array-Based Systems [10] (ABS). To this end, we present an encoding of the synchronous semantics of SDAs over the asynchronous semantics of ABS. We experimented with two ABS verification tools — mcmt [11] and cubicle [6]. Both tools were able to verify our simplest examples easily, but failed on more complex ones.

Related Work. Verification of parameterized systems has been widely studied. Due to limited space, we only touch upon some key points. One view of such systems is as a set of automata communicating over a network. Emerson and Namjoshi [8] present cutoff results in the case of ring networks. Delzanno et al. [7] present decidability and undecidability results for a range of network topologies with and without broadcast using the formalism of well-structured transition systems [9]. A good survey of work in this area, and new results, is provided by Aminof et al. [3].

Another view of parameterized systems, closer to ours, is that of an Array-Based System (ABS) [10]. An ABS consists of a set of arrays, and guarded transitions. In each step, one of the enabled transitions is applied to update the array. Such systems have received increased attention over the last few years, and new decision procedures and model checkers [11, 6] have been developed for verifying them. By encoding SDAs as ABSs, we are able to build on this work.

Verification of distributed algorithms is traditionally done upon some key points. One view of such systems is as a transition system [9]. Emerson upon some key points. One view of such systems is as a transition system [9].

The rest of this paper is organized as follows. Sec. 2 presents preliminary concepts. Sec. 3 presents our theoretical results. Sec. 4 presents experimental results, and concludes.

2. PRELIMINARIES

Formally, a SDA P is a 4-tuple \((GV, W, Z, \rho)\) where: (i) \(GV\) is the global array; (ii) each element of \(GV\) is a bitvector of width \(W\) \(\in\mathbb{N}\); (iii) \(Z\) \(\in\{0, W\}\) is the number of non-deterministic bits available to each node; and (iv) \(\rho\) is a procedure executed by each node in every round.

Syntax. Let \(IV\) be a set of id variables, and \(id\) be a distinguished variable such that \{\(GV\)\}, \(IV\) and \{\(id\)\} are mutually disjoint. The body of \(\rho\) is a statement. The "abstract" syntax of statements, lvalues, and expressions is given by the following BNF grammar (\(w\) is an integer in \([1, W]\)):

\[
\begin{align*}
\text{stmt} &::= \text{skip} \mid \text{lval} = \text{exp} \mid \text{ite}(\text{exp}, \text{stmt}, \text{stmt}) \\
& \mid \text{all}(\text{IV}, \text{stmt}) \mid \langle\text{stmt}^+\rangle \\
\text{lval} &::= \text{GV}[id][w] \\
\text{exp} &::= \top \mid \bot \mid \text{lval} \mid \text{GV}[IV][w] \mid \text{id} \mid \text{IV} \mid \phi(\text{exp}^+) 
\end{align*}
\]

Intuitively, \(\text{skip}\) is a nop, \(\text{lval} = \text{exp} \) is an assignment, \(\text{ite}\) is an "if-then-else", \(\text{all}(v, s, t)\) executes \(st\) iteratively by substituting \(v\) with the id of each node. \((s_{[1]}, \ldots, s_{[n]}\) executes \(s_t\) through \(s_k\) in sequence, and \(\phi\) is an operator. \(\text{all}\) enables iteration over all nodes of \(P\) without knowing the exact number of such nodes a-priori. \(IV\) and \(id\) are natural numbers. Statements are well-typed and variables are well-scope.

Semantics. The instance of SDA \(P\) with \(n \in \mathbb{N}\) nodes is denoted \(P(n)\). Let \(A(n)\) be the set of arrays with \(n\) elements, each a \(W\)-wide bitvector. The semantics of \(P(n)\) is the state transition system \((S, I, R)\) where: (i) \(S = A(n); (ii) I = \{a \in S \mid \forall i \in [1, n], v_i \in (Z, W), a[i][j] = \bot\}; and \(R \subseteq S \times S\) is the relation such that \((s, s') \in R\) iff for all \(i \in [1, n], s'[i]\) is the final value of \(GV[i]\) if \(p\) is executed from the state \(GV = s \land id = i\). Let \(R^*\) denote the transitive closure of \(R\).

Sequentialization. Note that \(P(n)\) is semantically equivalent to a sequential program \([P(n)]\) that maintains two copies of \(GV\) — \(GV_0\) and \(GV_1\) — and executes iteratively. Initially, \(GV_0 = I\). In each iteration, for each node \(Ni\), it executes \(\rho\) by reading from \(GV_0\) and writing to \(GV_1[i]\). After all nodes have been processed, it copies \(GV_1\) back to \(GV_0\) and proceeds with the next iteration. This observation is crucial for modeling and verifying SDAs as Array-Based Programs.

Specification. A specification \(\phi\) is a formula \(\forall i, \Psi(i)\) where \(\Psi(i)\) is an expression with the following grammar:

\[
\exp ::= \ast \mid \top \mid \bot \mid \text{lval} \mid \text{GV}[i][w] \mid \phi(\exp^+) 
\]

Let the semantics of \(P(n)\) be \((S, I, R)\). We say that \(P(n)\) satisfies \(\phi\), denoted \(P(n) \models \phi\) iff \(\forall a \in A(n) \forall i \in N \forall v_i \in I, (a, \Psi(j)) = \top\) where \((a, \Psi(j))\) is the evaluation of \(\Psi(i)\) after each \(GV[i][w]\) has been replaced with \(a[j][w]\).

Parameterized Model Checking. The input to the parameterized model checking problem is a SDA \(P\) and a specification \(\phi\). Its output, denoted \(\text{PARMODCK}(P, \phi)\), is \(\bot\) if \(\exists n \in \mathbb{N}. P(n) \not\models \phi\), and \(\top\) otherwise.

3. THEORETICAL RESULTS

We now show that the parameterized model checking problem is undecidable by reducing the Post Correspondence Problem [15] to it. Initially, we assume \(Z \in [1, W]\). Subsequently, we show that the undecidability holds even if \(Z = 1\).

Post’s Correspondence Problem. Let \(\Sigma\) be an alphabet with at least two letters. An instance \(I\) of PCP is given by two sequences \(U = (u_1, \ldots, u_m)\) and \(V = (v_1, \ldots, v_n)\) of strings \(u_i, v_i \in \Sigma^+\). The output, denoted \(\text{pcp}(I)\), is \(\bot\) if there exists a finite non-empty sequence (known as the solution) \(\langle i_1, \ldots, i_p \rangle\) with \(i_j \in [1, m]\) for \(j \in [1, p]\) such that:

\[
u_1 \bullet \ldots \bullet u_p = v_1 \bullet \ldots \bullet v_p
\]

where \(\bullet\) is string concatenation. PCP is undecidable [15].

For example, suppose \(U = (a, ab, bba)\) and \(V = (baa, ab, bb)\). Then \(\text{pcp}(I) \not\models \top\) since there exists a solution \((3, 2, 3, 1)\). This is because \(u_3 \bullet u_2 \bullet u_1 = bbaabbbbaa = v_3 \bullet v_2 \bullet v_1\). On the other hand, if \(U = (aa, aab, baaa)\) and \(V = (a, bb, abb)\), then \(\text{pcp}(I) = \top\) since there is no solution (each \(u_i\) has a bigger length than the corresponding \(v_i\)).

We show that for every instance \(I\) of PCP, there exists a SDA \(P\) and specification \(\phi\) such that \(\text{PARMODCK}(P, \phi) = \text{pcp}(I)\). For convenience, we assume that each element of \(GV\) is a record whose fields are finite datatypes. Specifically, there are five fields: (i) \(i\) and \(posu, posv, dve\), and the first is initialized non-deterministically; and (ii) \(st\) is initialized to 0. For simplicity, we write \(f[i]\) to mean \(GV[i][w]\) where \(f\) is a field.

At a high level, the SDA implements a protocol that works as follows. Variable \(st[i]\) indicates the overall progress of \(Ni\) in the protocol. Node \(N_i\) is a special node that helps in detecting success. Every other node \(N_i, i > 1\) represents two letters: (i) the \(posu[i][j]\)-th letter in \(u_{dve[i]}\), and (ii) the \(posv[i][j]\)-th letter in \(v_{dve[i]}\). The protocol succeeds iff the sequence of
if \((st[id] = 0)\)

- if \((id = 1)\) \(st[id] := 2;\)
- else if \((1 \leq id[u][id] \leq m \land 1 \leq pos[u][id] \leq |u|_{id[u][id]} \land u_{id[u][id]}[pos[u][id]] = v_{id[u][id]}[pos[u][id]])\) \(st[id] := 2;\)
- else \(st[id] := 1;\)
- else if \((id[u][id] = 2)\)
  - if \((id[1] = 1 \land pos[u][id] = 1)\) \(st[id] := 3;\)
  - else \(st[id] := 1;\)
- else if \((id[u][id] = 3)\)
  - if \((id[2] = 1 \land pos[u][id] = 1)\)
    - else \(st[id] := 4;\)
  - else \(st[id] := 1;\)

where:

\[
X_1 := pos[u][id] > 1 \implies (id[u][id] = id[u][id] \land pos[u][id] = pos[u][id] - 1)
\]

\[
X_2 := pos[v][id] > 1 \implies (id[v][id] = id[v][id] \land pos[v][id] = pos[v][id] - 1)
\]

\[
X_3 := pos[u][id] = 1 \implies pos[u][id] = |u|_{id[u][id] - 1}|
\]

\[
X_4 := pos[v][id] = 1 \implies pos[v][id] = |v|_{id[v][id] - 1}|
\]

Figure 1: Code Fragment for First Three Rounds.

Letters represented by \((N_2, \ldots, N_n)\) is a solution to \(\text{pcp}(I)\).

The procedure \(\rho\) works as follows (the code is in Figure 1):

1. First ensures that the letters represented are valid and identical. This is the case if \((st[id] = 0)\) in Figure 1. Note that \(st[id] = 1\) means no further progress.

2. Next checks that \(N_2\) represents the first letter of a word. This is the case if \((st[id] = 2)\) in Figure 1.

3. Next checks that the node before it represents either the previous letter in the same word or the last letter in another word, as appropriate. This is the case if \((st[id] = 3)\) in Figure 1.

4. The final stage of \(\rho\) denoted \(\text{eqcheck}\) checks that the nodes represent the same sequence of indices in both \(U\) and \(V\). Let the sequence of indices in \(U\) represented by the nodes be \((i_1, \ldots, i_{|u|})\), and the sequence of indices on \(V\) represented by the nodes by \((i_v, \ldots, i_{|v|})\). Starting with node \(N_1\), the program first computes \(i_{u1}\) and \(i_{v1}\) and compares them, then computes \(i_{u2}\) and \(i_{v2}\), and compares them, and so on. The protocol succeeds if \((i_{u1}, \ldots, i_{u_{|u|}}) = (i_{v1}, \ldots, i_{v_{|v|}})\). Success is indicated by \(N_1\) entering a special ok state.

We next define \(\text{eqcheck}\) in more detail.

\text{eqcheck: high-level idea.} Node \(N_1\) sends out two tokens \(tu\) and \(tv\) along the line \((N_1, \ldots, N_n)\). If node \(N_1\) receives \(tu\) there are two cases:

1. If \(N_1\) does not represent the last letter of its corresponding \(U\)-word (i.e., if \(pos[u][i] \neq |u|_{id[u][i]}\)), it passes \(tu\) to \(N_{i+1}\) and moves to the done state.

2. If \(N_1\) represents the last letter of its \(U\)-word (i.e., if \(pos[u][i] = |u|_{id[u][i]}\)), it waits for \(N_1\) to move to a special \(\text{green}\) state. In any subsequent round, if \(N_1\) detects that \(N_1\) is in the \(\text{green}\) state, it passes \(tu\) to \(N_{i+1}\) and moves to the done state.

else if \((st[id] = 4)\)

- if \((id = 1)\) \(tu[id] := 1; tv[id] := 1;\)
- else \(tu[id] := 0; tv[id] := 0; wtu[id] := 0; wtv[id] := 0; st[id] := \text{init};\)
- else if \((st[id] = \text{init}) \land (id = 1)\)
  - if \(\text{exists}(iu, wtu[iu] = 1) \land \text{exists}(iv, wtv[iv] = 1)\)
    - if \((i = iv)\) \(st[id] := \text{ok};\)
    - else if \((id[iu] = iu[iv])\) \(st[id] := \text{green};\)
  - else if \((st[id] = \text{init}) \land (id \neq 1)\)
    - if \((wto[\text{id}] \land \text{st}[1])\) \(wto[id] := 0; tu[id] := 1; st[id] := \text{done};\)
    - else if \((tu[id - 1] = 1 \land tu[id] = 0)\)
      - if \((pos[u][id] = |u|_{id[u][id]}\) \(wto[id] := 1;\)
      - else \(st[id] := \text{done};\)
    - else if \((tv[id - 1] = 1 \land tv[id] = 0)\)
      - if \((pos[v][id] = |v|_{id[v][id]}\) \(wto[id] := 1;\)
      - else \(tv[id] := 1; st[id] := \text{done};\)
  - else if \((st[id] = \text{green})\) \(st[id] := \text{init};\)

Figure 2: Code Fragment for EqCheck.

A symmetric behavior (involving \(v_{id[x]}[i]\) and \(pos[i]\)) occurs if \(N_1\) receives token \(tv\). Note that \(N_1\) may receive both \(tu\) and \(tv\) in the same round. Node \(N_1\) moves to the \(\text{green}\) state (for just one round) if it detects that both tokens have reached the ends of words from \(U\) and \(V\) with equal indices. Specifically, suppose \(tu\) is with node \(N_{iu}\) and \(tv\) is with node \(N_{iv}\). Then, \(N_1\) moves to the \(\text{green}\) state for just one round if the following condition holds:

\[
pos[u][iv] = |u|_{id[u][iv]} \land pos[v][iv] = |v|_{id[v][iv]} \\
\land GV[iu, idu = GV[iv, idv]
\]

Also, \(N_1\) moves to the \(\text{ok}\) state if, in addition to the above condition, \(iv = iv\) holds as well.

\text{eqcheck: implementation details.} To simulate token passing with shared variables, we introduce two extra Boolean fields \(tu\) and \(tv\) in each element of \(GV\). Then, node \(N_1\) has token \(tu\) (or \(tv\)) iff is the largest index such that \(tu[i - 1] = \top\) (or \(tv[i - 1] = \top\)). Thus, \(N_1\) passes \(tu\) (or \(tv\)) to \(N_{i+1}\) by setting \(tu[i]\) (or \(tv[i]\)) to \(\top\). We also have two more Boolean fields \(wtu\) and \(wtv\) to indicate that the node has token \(tu\) or \(tv\) and is waiting for \(N_1\) to move to the \(\text{green}\) state. In the first round of EqCheck, \(tu, tv, wtu\) and \(wtv\) are initialized appropriately. The code for EqCheck is shown in Figure 2. Note that \(\text{exists}\) can be implemented using \(\text{all}\) and additional variables.

Consider the specification \(\phi = \forall i, st[i] \neq \top\). It can be shown that \(\text{parmodck}(P, \phi) = \text{pcp}(I)\). Thus we have:

**Theorem 1.** \(\text{parmodck}(P, \phi)\) is undecidable.

We now show that \(\text{parmodck}(P, \phi)\) is undecidable even if \(Z = 1\). Specifically, for any SDA \(P = (GV, W, Z, \rho)\) where \(Z > 1\), there exists another SDA \(P = (GV, \hat{W}, 1, \beta)\), such that \(\forall n \in N, P(n)\) is simulated by \(\hat{P}(Z \times n)\). In the first round, every \(Z\)-th node of \(\hat{P}(Z \times n)\) copies the NDBs from its next \(Z-1\) neighbors. Now every \(Z\)-th node has \(Z\) NDBs.

Note that there are \(n\) such nodes. Subsequently, node \(N_1\) of \(P(Z \times n)\) simulates node \(N_1\) of \(P(n)\) iff \((i - 1) \times Z + 1\). This implies the next result.

**Theorem 2.** \(\text{parmodck}(P, \phi)\) is undecidable even when \(P = (GV, W, 1, \rho)\).
<table>
<thead>
<tr>
<th>Example</th>
<th>Time</th>
<th>N</th>
<th>SMT</th>
<th>Time</th>
<th>N</th>
<th>SMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUTEX-OK</td>
<td>0.46</td>
<td>21</td>
<td>1358</td>
<td>0.61</td>
<td>608</td>
<td>614</td>
</tr>
<tr>
<td>MUTEX-BUG1</td>
<td>1.2</td>
<td>70</td>
<td>2680</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUTEX-BUG2</td>
<td>0.52</td>
<td>48</td>
<td>1350</td>
<td>0.45</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>COLL-OK</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>248</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>COLL-BUG1</td>
<td>251</td>
<td>1045</td>
<td>195K</td>
<td>248</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>COLL-BUG2</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>2892</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 3: Experimental Results. Times are in second; N = no. of nodes; SMT = no. of calls to SMT solver; - = data not reported; * = timeout.

The cutoff approach [8] for parameterized verification is based on proving that for a certain class of specifications $\phi$ and parameterized system $P$, there exists a known $K \in \mathbb{N}$ such that $\forall n > 0 \cdot P(n) \models \phi \iff \forall n \leq K \cdot P(n) \models \phi$.

We now show that for SDAs, no such cutoff can exist even if each node is completely deterministic.

**Theorem 3.** For each $K \in \mathbb{N}$, there exists a specification $\phi$ and a SDA $P$ with $Z = 0$ such that $\forall n \leq K \cdot P(n) \models \phi \wedge P(K + 1) \not\models \phi$.

**Proof.** Consider the SDA $P$ where each element of $GV$ has one field $st$ initialized to 0, and following function $p$: if $(id > K) \Rightarrow st[id] := 2; \text{else} \Rightarrow st[id] := 1$. Let $\phi = \forall_1.st[i] \neq 2$. Thus, $\forall n \leq K \cdot P(n) \models \phi \wedge P(K + 1) \not\models \phi$. □

4. EXPERIMENTAL RESULTS

Recall that any SDA instance $P(n)$ is semantically equivalent to a sequential program $[P(n)]$ that operates over two copies of the array $GV$. We now show that $[P(n)]$ can be encoded as an Array-Based System (ABS). The main challenge is that in each iteration $[P(n)]$ processes every array element, while in an ABS, only one enabled transition is executed asynchronously in each step. Our solution is to: (i) implement a “barrier” using “universal guards” [2]; (ii) use the barrier to implement synchronicity via a protocol modeled after “two-phase commit”. Due to lack of space, we are unable to provide further details. However, all our examples are available at http://www.contrib.andrew.cmu.edu/~schaki/misc/spin14.tgz.

We experimented with two sets of examples – mutual exclusion and collision avoidance between mobile robots. Both protocols use ordering between ids to ensure that at most one node is in the critical section and no two nodes are in the same physical location (a coordinate on a two-dimensional grid). Mutual exclusion requires reserving and acquiring a single lock. Collision avoidance is more complicated and requires two locks – one for a node’s current location and the other for the location the node is moving to.

For each example, we created three versions – one correct and two buggy by omitting crucial checks in the protocol. We manually translated each version into the input language of two ABS model checkers—mcmt v2.5 and CUBICLE v0.5. We then applied the two tools – using their default settings – on their corresponding example files. All experiments were done on a 2.3GHz Machine running 64bit Linux with a time limit of 120 minutes and a memory limit of 4GB. Our experimental results are shown in Fig. 3. Both mcmt and CUBICLE perform symbolic backward reachability using an SMT solver and heuristics to prune out unfeasible executions, and detect fixed points. The results indicate that they are effective on all mutual exclusion examples and the buggy collision avoidance examples. However, the safe collision avoidance example is beyond the scope of both tools.

**Conclusion.** We presented preliminary results and caveats toward parameterized verification of SDAs. We are exploring several next steps: (i) formally defining and proving correctness of the translation from SDAs (written in dasl) to ABSs; (ii) developing verification algorithms that operate on SDAs directly instead of converting them to ABSs; (iii) implementing a robust parameterized model checker for SDAs; and (iv) performing a more comprehensive evaluation.

**Acknowledgments.** Copyright 2014 ACM. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0001199

5. REFERENCES


