Partial Order Trace Analyzer (POTA) for Distributed Programs

Alper Sen¹  Vijay K. Garg²,³

Department of Electrical and Computer Engineering
The University of Texas at Austin, Austin, TX, 78712, USA

Abstract
Checking the correctness of software is a growing challenge. In this paper, we present a prototype implementation of Partial Order Trace Analyzer (POTA), a tool for checking execution traces of both message passing and shared memory programs using temporal logic. So far runtime verification tools have used the total order model of an execution trace, whereas POTA uses a partial order model. The partial order model enables us to capture possibly exponential number of interleavings and, in turn, this allows us to find bugs that are not found using a total order model. However, verification in partial order model suffers from the state explosion problem – the number of possible global states in a program increases exponentially with the number of processes.

POTA employs an effective abstraction technique called computation slicing. A slice of a computation (execution trace) with respect to a predicate is the computation with the least number of global states that contains all global states of the original computation for which the predicate evaluates to true. The advantage of this technique is that, it mitigates the state explosion problem by reasoning only on the part of the global state space that is of interest. In POTA, we implement computing slicing algorithms for temporal logic predicates from a subset of CTL. The overall complexity of evaluating a predicate in this logic upon using computation slicing becomes polynomial in the number of processes compared to exponential without slicing.

We illustrate the effectiveness of our techniques in POTA on test cases such as the General Inter-ORB Protocol (GIOP) [18]. POTA also contains a module that translates execution traces to Promela [16] (input language SPIN). This module enables us to compare our results on execution traces with SPIN. In some cases, we were able to verify traces with 250 processes compared to only 10 processes using SPIN.

©2003 Published by Elsevier Science B. V.
1 Introduction

A fundamental problem in distributed systems is that of predicate detection detecting whether a finite execution trace of a distributed program satisfies a given predicate. There are applications of predicate detection in many domains such as testing, debugging, and monitoring of distributed programs. For example, when debugging a distributed mutual exclusion algorithm, it is useful to monitor the system to detect concurrent accesses to the shared resources.

A finite trace can be modeled in two ways. The first model imposes a partial order between events, for example Lamport’s happened-before relation \(^20\). The second model imposes a total order (interleaving) of events. We use the former approach in this paper, which is a more faithful representation of concurrency \(^20\).

Consider an execution of a distributed program. The partial order model of the resulting execution trace is shown in Figure 1(a). In the trace, there are two processes \(P_1\) and \(P_2\) with integer variables \(x\) and \(y\), respectively. The events are represented by solid circles. Process \(P_2\) sends a message to process \(P_1\) by executing event \(e_1\) and process \(P_1\) receives that message by executing event \(e_1\). Each event is labeled with the value of the respective variable immediately after the event is executed. For example, the value of \(x\) immediately after executing \(e_1\) is 2. The first event on each process initializes the state of the process. The set of all global states reachable from the initial state \(\{e_0, f_0\}\) is displayed in Figure 1(b). In the figure, we represent a global state as a tuple where each element is the last event that occurred on a process. Observe that \(\{e_1, f_0\}\) is not a reachable global state because it depicts a situation where a message has been received from \(P_2\) by \(P_1\), that is \(e_1\), but \(P_2\) has not yet sent the message. By using a partial order representation, we are able to capture all possible interleavings of events, namely ten in total, rather than a single interleaving. One such interleaving sequence is \(\{e_0, f_0\}, \{e_0, f_1\}, \{e_1, f_1\}, \{e_2, f_1\}, \{e_3, f_1\}, \{e_3, f_2\}, \{e_3, f_3\}\) as shown in Figure 1(b) with thick lines. Therefore we can obtain better coverage in terms of testing and debugging by capturing all interleavings. This coverage may translate into finding bugs that are not found using a single interleaving.

The main problem in predicate detection in the partial order model is the state explosion problem the set of possible global states of a distributed program with \(n\) individual processes can be of size exponential in \(n\). A variety of strategies for ameliorating the state explosion problem, including symbolic representation of states and partial order reduction have been explored \(^{23,21,22,26,31,33}\).

---

1 Email: sen@ece.utexas.edu Homepage: http://www.ece.utexas.edu/~sen/
2 supported in part by the NSF Grants ECS-9907213, CCR-9988225, Texas Education Board Grant ARP-320, an Engineering Foundation Fellowship, and an IBM grant
3 Email: garg@ece.utexas.edu Homepage: http://www.ece.utexas.edu/~garg/
Fig. 1. (a) A computation (b) its set of all reachable global states (c) its slice with respect to \((2 \leq x \leq 4) \land (y \neq 2)\) (d) its slice with respect to \(\mathbf{EF}((2 \leq x \leq 4) \land (y \neq 2))\)

In this paper, we present a prototype implementation of Partial Order Trace Analyzer (POTA) tool for checking execution traces of distributed programs. POTA consists of an instrumentation module, a translator module that translates execution traces into Promela \cite{10} (SPIN input language) and an analyzer module. The use of an effective abstraction technique called computation slicing for temporal logic verification is the most significant aspect of POTA and constitutes the analyzer module.

Computation slicing was introduced in \cite{11,24} as an abstraction technique for analyzing distributed computations (finite execution traces). A computation slice, defined with respect to a global predicate, is the computation with the least number of global states that contains all global states of the original computation for which the predicate evaluates to true. Slicing can be used to throw away the extraneous global states of the original computation in an efficient manner, and focus on only those that are currently relevant for our purpose.

Using the results in \cite{11,24} and \cite{28}, we can efficiently use computation slicing for the subset of CTL \cite{4} with the following three properties. First, temporal operators are \(\mathbf{EF}, \mathbf{EG}, \text{ and } \mathbf{AG}\) and boolean operators are conjunction and disjunction. Second, atomic propositions are regular predicates, which we will define later. Third, negation operator has been pushed onto atomic propositions. We call this logic Regular CTL plus (RCTL+), where the plus denotes that the disjunction and negation operators are included in the logic. We also consider a disjunction and negation free subset of RCTL+ and denote this by Regular CTL (RCTL). In RCTL+, we use the class of predicates, called regular predicates, that was introduced in \cite{11}. The slice with
respect to a regular predicate contains precisely those global states for which
the predicate evaluates to true. Regular predicates widely occur in practice
during verification. Some examples of regular predicates are conjunction of
local predicates such as “all processes are in red state”, certain channel
predicates such as “at most k messages are in transit from process $P_i$ to
$P_j$”, and some relational predicates.

To illustrate predicate detection using computation slicing, consider the
computation in Figure 1(a). Let $p = (2 \leq x \leq 4) \land (y \neq 2)$, and suppose we
want to detect $\text{EF}(p)$. Without computation slicing, we are forced to examine
all global states of the computation, thirteen in total, to decide whether the
computation satisfies the predicate. Alternatively, we can compute the slice
of the computation with respect to regular predicate $\text{EF}(p)$ and use this slice
for predicate detection. For this purpose, first we compute the slice with
respect to the atomic proposition $p$ as follows. Immediately after executing
$f_2$, the value of $y$ becomes 2 which does not satisfy $y \neq 2$. To reach a global
state satisfying $y \neq 2$, $f_3$ has to be executed. In other words, any global
state in which only $f_2$ has been executed but not $f_3$ is of no interest to us
and can be ignored. The slice is shown in Figure 1(c). It is modeled by a
partial order on a set of meta-events; each meta-event consists of one or more
“primitive” events. A global state of the slice either contains all the events in
a meta-event or none of them. Moreover, a meta-event “belongs” to a global
state only if all its incoming neighbours are also contained in the state. The
slice contains only four states $C, D, V$ and $W$ and has much fewer states than
the computation itself — exponentially smaller in many cases — resulting in
substantial savings. Using the slice in Figure 1(c), we can obtain the last
state of the computation, which is denoted by $W$. We also
know from the definition of $\text{EF}(p)$ that every global state of the computation
that occurs before $W$ satisfies $\text{EF}(p)$, e.g. states enclosed in the dashed ellipse
in Figure 1(b). Therefore, applying this observation we can compute the slice
with respect to $\text{EF}(p)$ as shown in Figure 1(d). Finally, we check whether the
initial state of the computation is the same as the initial state of the slice. If
the answer is yes then the predicate is satisfied, otherwise not.

POTA implements predicate detection algorithms for RCTL and RCTL+
which use computation slicing. We show in 23, that the complexity of pred-
icate detection for a predicate $p$ in RCTL is $O(|p| \cdot n^2 |E|)$, where $|p|$ is
the number of boolean and temporal operators in $p$ and $E$ is the total number
of events. To the best of our knowledge, there did not exist tools that imple-
ment efficient algorithms (polynomial in the number of processes) to detect
predicates that contain nested temporal logic predicates. An example of a
nested predicate is $\text{AG(\text{EF}(\text{reset}))}$, which states that reset is possible from
every state. Furthermore, we validate with experiments that even for RCTL+
predicates our computation slicing based technique is very effective.

We performed experiments using POTA on several protocols. We also used
the POTA translator module to enable comparison with SPIN on execution
traces. In fairness, SPIN is designed for checking correctness of programs and not traces. However, to the best of our knowledge it is the best distributed program verification tool we can use for our partial order models. Some of the protocols we used for experiments are the General Inter-ORB Protocol (GIOP) [13] and the primary secondary protocol [32]. GIOP is a central feature of the Common Object Request Broker Architecture (CORBA) that aids in achieving the desired interoperability between ORBs. The CORBA specification defines a standard protocol to allow communication of object invocations between ORBs. Kamel and Leue [13] could not fully verify a model of GIOP with 10 processes. Instead, they verified a simplified version of the protocol without server migration functionality. In one case, we generated execution traces of unsimplified GIOP protocol for a configuration with 250 processes. However, even with an execution trace input, SPIN failed to complete verification with more than 10 processes. We also injected faults into the protocol and analyzed the resulting execution traces. With SPIN, we used bit-state hashing approximation option to handle larger number of processes, but in this case SPIN failed to find the faults before running out of memory. However, POTA was able to find the faults easily. In all cases, our algorithms are significantly faster and space efficient than SPIN. We have measured over three orders of magnitude gain over SPIN in some experiments.

Computation slicing can indeed be used to facilitate predicate detection even for a larger class of predicates than RCTL+ as illustrated by the following example. Consider a predicate $p$ that is a conjunction of two clauses $p_1$ and $p_2$. Now, assume that $p_1$ is such that it belongs to RCTL+ but $p_2$ has no structural property that can be exploited for efficient detection, such as, $(x_1 * x_2 + x_3 > x_4)$, where $x_i$ is an integer variable on process $i$. To detect $p$, without computation slicing, we are forced to use global-state-space-construction-based approaches, which do not take advantage of the fact that $p_1$ can be detected efficiently. With computation slicing, however, we can first compute the slice for $p_1$. If only a small fraction of global states satisfy $p_1$, then instead of detecting $p$ in the computation, it is much more efficient to detect $p$ in the slice. Therefore by spending only polynomial amount of time in computing the slice we can throw away exponential number of global states, thereby obtaining an exponential speedup overall.

2 Related Work

Predicate detection is a hard problem. Detecting even a 2-CNF predicate under EF modality has been shown to be NP-complete in general [25].

Predicate detection is a widely-studied problem. There are three major approaches to solving predicate detection: global-snapshot-based approach [2], global-state-space-construction-based approach (including model checking) [45], and predicate-restriction-based approach [10]. The first approach can detect only stable predicates (which remain true once they become true),
and the second approach suffers from the state explosion problem. We follow the predicate-restriction-based approach that exploits the structure of the predicate and directly uses the computation to detect if the predicate is satisfied in a global state. Some examples of the predicates for which the predicate detection can be solved efficiently are: conjunctive [10, 17], disjunctive [10], stable [2], observer-independent [3, 10], linear [10, 29], and non-temporal regular [11, 24] predicates. These predicate classes have been so far detected under some or all of the temporal operators EF, EG, AG, AF and under the until operator of CTL [29], but not under any nesting of these operators. For example, a predicate EF(p ∨ EG(q)), where p and q are conjunctive predicates, cannot be efficiently detected using only the efficient algorithms for conjunctive predicates. In POTA, we can detect such nested temporal logic predicates efficiently.

The idea of using temporal logic for analyzing execution traces (also referred to as runtime verification) has recently been attracting a lot of attention. We first presented a temporal logic framework for partially ordered execution traces in [29] and gave efficient algorithms for predicates of the form EG(p) and AG(p) when p is an linear predicate. The efficiency of those algorithms depended on the fact that p was a state predicate and therefore we could efficiently evaluate the satisfiability of p at a global state. However, in this paper we present implementation of efficient algorithms even when p is a temporal predicate.

Some other examples of using temporal logic for checking execution traces are the commercial Temporal Rover tool (TR) [7], the MaC tool [19], the JPaX tool [13], and the JMPaX tool [30]. TR allows the user to specify the temporal formula in programs. These temporal formula are translated into Java code before compilation. The MaC and JPaX tools consider a totally ordered view of an execution trace and therefore can potentially miss bugs that can be deduced from the trace.

JMPaX tool is closer to POTA because of the partial order trace model. The differences in both approaches can be summarized as follows. JMPaX uses a subset of temporal logic with safety where atomic propositions can be arbitrary. Whereas POTA uses a subset of temporal logic with both safety and liveness where atomic propositions are restricted. The complexity of the predicate detection algorithm in POTA is polynomial-time in the number of processes whereas the complexity can be exponential-time in the number of processes (as large as the width of the lattice of global states) in JMPaX.

3 Overview of POTA Architecture

The overall structure of POTA architecture is shown in Figure 2. The tool consists of 3 main modules; analyzer, translator, and instrumentor.

The analyzer module contains our computation slicing and predicate detection algorithms. Given an execution trace and a predicate (specification)
in RCTL+, the computation slice may contain more states than the ones that satisfy the predicate. Therefore, the analyzer module uses the following strategy to decide whether the predicate is satisfied or not. Case 1, if the slice and the input trace have different initial states then the predicate is not satisfied. In this case a counterexample is generated. Case 2, if the predicate is from RCTL and the slice and the input trace have the same initial states then the predicate is satisfied. Case 3, when the predicate does not belong to RCTL (that is, it contains disjunction or negation operators) and the slice and the input trace have the same initial states then we have to take an extra step. This is because the initial state of the slice may not satisfy the predicate. Therefore, we employ the translation module and translate the slice into Promela [10] (input language of SPIN). Then we use SPIN to check the trace assuming that there are equivalent specifications in LTL.

The translator module takes a partial order representation of a trace and generates output in specific languages. This module serves two purposes; to enable comparison of our slicing technique with other techniques such as partial order reduction and to enable verification of predicates that do not belong to RCTL but for which we can take advantage of computation slicing. The latter purpose is served when the predicate belongs to RCTL+ as explained in Case 3 in the above paragraph or when the predicate is a conjunction of predicates where one of the conjuncts belong to RCTL+ as explained in the introduction. Since we are working with distributed programs which exhibit a lot of parallelism and independency, partial order reduction techniques can take advantage of these properties of distributed programs. The SPIN model checker contains implementation of partial order reduction techniques. Currently, translation from traces to Promela is supported. The translation mechanism is similar to the technique explained in [21] for translations from message sequence charts (MSC) to Promela.

The instrumentation module inserts code at the appropriate places in the program to be monitored. The instrumented program is such that it outputs
the values of variables relevant to the predicate in question and keeps a vector clock that is updated for each internal, send and receive event according to the Fidge/Mattern algorithm [923]. We use the vector clock to obtain a partial order representation of traces.

Upon running the instrumented program a separate log file for each process is generated. Each log file consists of a sequence of local states that a process goes through. Each local state contains the values of variables relevant to the predicate being verified and a vector clock. Log files for every process are then combined to obtain a partial order representation of the execution trace.

Instead of using a log file, if every process sends its trace to a dedicated process which combines them during runtime, we can obtain an on-line verification environment.

Currently, programs are manually instrumented. We conducted experiments with Java and Promela programs. For SPIN programs, we insert code into Promela programs and also made changes to the SPIN source code so that we can obtain a partial order model when we run SPIN in simulation mode with the option for generating a message sequence chart output. SPIN’s MSC output is by default a total ordered execution. However, we observed from this MSC output that there are unnecessary dependencies therefore events do not need to be totally ordered such as request messages from two different processes sent to two different servers do not need to be totally ordered. We are in the process of choosing an appropriate instrumentation technique for Java programs. The choice is between Java JDI as in [1] or byte code instrumentation as in JPaX.

4 Model

A distributed program consists of \( n \) processes denoted by \( P_1, P_2, \ldots, P_n \). Traditionally, a distributed computation is modeled as a partial order on a set of events, called happened-before relation [20]. The happened-before relation between any two “primitive” events \( e \) and \( f \) can be formally stated as the smallest relation such that \( e \) happened-before \( f \) if and only if \( e \) occurs before \( f \) in the same process, or \( e \) is a send of a message and \( f \) is a receive of that message, or there exists an event \( g \) such that \( e \) happened-before \( g \) and \( g \) happened-before \( f \). In this paper we relax the restriction that the order on events must be a partial order. More precisely, we use directed graphs to model distributed computations as well as slices. Directed graphs allow us to handle both of them in a uniform and convenient manner. Furthermore, we can extend the happened-before relation to read and write events of shared variables as in [30].

Given a directed graph \( G \), let \( V(G) \) and \( E(G) \) denote the set of vertices and edges, respectively. We define a consistent cut (global state) on directed graphs as a subset of vertices such that if the subset contains a vertex then it contains all its incoming neighbours. Formally, \( C \) is a consistent cut of \( G \),
if \( \forall e, f \in V(G) : (e, f) \in E(G) \land (f \in C) \Rightarrow (e \in C) \). We say that a strongly connected component is non-trivial if it has more than one vertex. We denote the set of consistent cuts of a directed graph \( G \) by \( \mathcal{C}(G) \). Observe that the empty set \( \emptyset \) and the set of vertices \( V(G) \) trivially belong to \( \mathcal{C}(G) \). We call them trivial consistent cuts. We use \( \mathcal{P}(G) \) to denote the set of pairs of vertices \( (u, v) \) such that there is a path from \( u \) to \( v \) in \( G \). We assume that each vertex has a path to itself.

We model a distributed computation (or simply a computation), denoted by \( \langle E, \rightarrow \rangle \), as a directed graph with vertices as the set of events \( E \) and edges as \( \rightarrow \). We use event and vertex interchangeably. To limit our attention to only those consistent cuts that can actually occur during an execution, we assume that \( \mathcal{P}(\langle E, \rightarrow \rangle) \) contains at least the Lampert’s happened-before relation \( \leq \). A distributed computation in our model can contain cycles. This is because whereas a computation in the happened-before model captures the observable order of execution of events, a computation in our model captures the set of possible consistent cuts. Intuitively, each strongly connected component of a computation can be viewed as a meta-event; a meta-event consists of one or more primitive events.

We assume the presence of a fictitious global initial and a global final event, denoted by \( \perp \) and \( \top \), respectively. The global initial event occurs before any other event on the processes and initializes the state of the processes. The global final event occurs after all other events on the processes. Any non-trivial consistent cut will contain the global initial event and not the global final event. Therefore, every consistent cut of a computation in traditional model (happened-before model) is a non-trivial consistent cut of the computation in our model and vice versa. Note that the empty consistent cut, \( \emptyset \), in the traditional model corresponds to \( \{ \perp \} \) in our model and the final consistent cut, \( E \), in the traditional model corresponds to \( E \setminus \{ \top \} \) in our model and we denote this by \( \mathcal{E} \). We use uppercase letters \( C, D, H, V, \) and \( W \) to represent consistent cuts.

Figure 3 shows a computation and its lattice of (non-trivial) consistent cuts. A consistent cut in the figure is represented by its frontier. For example, the consistent cut \( C = \{ e_3, e_2, e_1, f_2, f_1, \perp \} \) is represented by \( \{ e_3, f_2 \} \).

![Fig. 3. (a) A computation \( \langle E, \rightarrow \rangle \) (b) and its lattice corresponding to \( \mathcal{C}(G) \)](image)

Given a consistent cut, a predicate is evaluated with respect to the values
of variables resulting after executing all events in the cut. If a predicate \( p \) evaluates to true for a consistent cut \( C \), we say that \( C \) satisfies \( p \). We leave the predicate undefined for the trivial consistent cuts.

5 Background on Slicing

The notion of computation slice is based on the Birkhoff’s Representation Theorem for Finite Distributive Lattices \([6]\). The readers who are not familiar with earlier papers on computation slicing are urged to read the extended version of the paper from \([27]\).

5.1 Computation Slice

Roughly speaking, a computation slice (or simply a slice) is a concise representation of all those consistent cuts of the computation that satisfy the predicate. More precisely,

**Definition 5.1** [slice \([24]\)] A slice of a computation with respect to a predicate is a directed graph with the least number of consistent cuts that contains all consistent cuts of the given computation for which the predicate evaluates to true.

We denote the slice of a computation \( \langle E, \rightarrow \rangle \) with respect to a predicate \( p \) by \( \text{slice}(\langle E, \rightarrow \rangle, p) \). Note that \( \langle E, \rightarrow \rangle = \text{slice}(\langle E, \rightarrow \rangle, \text{true}) \). It was proven in \([24]\) that the slice exists and is uniquely defined for all predicates. Every slice derived from the computation \( \langle E, \rightarrow \rangle \) has the trivial consistent cuts (\( \emptyset \) and \( E \)) among its set of consistent cuts. A slice is empty if it has no non-trivial consistent cuts \([24]\). In the rest of the paper, unless otherwise stated, a consistent cut refers to a non-trivial consistent cut. In general, a slice will contain consistent cuts that do not satisfy the predicate (besides trivial consistent cuts). In case a slice does not contain any such cut, it is called lean. We next give the class of predicates for which the slice is lean.

6 Regular Predicates

Given a computation, the set of consistent cuts satisfying a regular predicate forms a sublattice of the set of consistent cuts of the computation \([14]\). Equivalently,

**Definition 6.1** [regular predicate \([24]\)] A predicate is regular if given two consistent cuts that satisfy the predicate, the consistent cuts obtained by their set union and set intersection also satisfy the predicate. Formally, given a regular predicate \( p \),

\((C \text{ satisfies } p) \land (D \text{ satisfies } p) \Rightarrow (C \cap D \text{ satisfies } p) \land (C \cup D \text{ satisfies } p)\)

We say that a regular predicate is non-temporal if it does not contain temporal operators such as \( \mathbf{EF}, \mathbf{AG}, \) and \( \mathbf{EG} \), otherwise it is a temporal
regular predicate. In polynomial-time algorithms are given for computing slices for non-temporal regular predicates. In [23], we showed that \( \text{EF}(p) \), \( \text{AG}(p) \), and \( \text{EG}(p) \) are temporal regular predicates when \( p \) is regular and gave polynomial-time algorithms to compute these slices, which we will briefly explain in the next section.

Some examples of non-temporal regular predicates are monotonic channel predicates such as “there are at least \( k \) messages in transit from \( P_i \) to \( P_j \)”, conjunction of local predicates such as “\( P_i \) and \( P_j \) are in critical section”, and relational predicates such as \( x_1 - x_2 \leq 5 \), where \( x_i \) is a monotonically non-decreasing integer variable on process \( i \). From the definition of a regular predicate we deduce that a regular predicate has a least satisfying cut and a greatest satisfying cut. Furthermore, the class of regular predicates is closed under conjunction.

Also in [21] polynomial-time algorithms are given to compute slices with respect to boolean combination of regular predicates. Given the slices with respect to two regular predicates, the complexity of computing the slice for the conjunction and disjunction of these regular predicates is \( O(n^2 |E|) \). The complexity of computing the slice for the negation of a regular predicate is \( O(n^2 |E|^2) \). Note that regular predicates are not closed under disjunction and negation operators therefore slices obtained with respect to predicates that contain these operators may not be lean.

6.1 RCTL+ Syntax and Semantics

We define successor of a cut by a relation \( \triangleright \subseteq \mathcal{C}(G) \times \mathcal{C}(G) \) such that \( C \triangleright D \) if and only if \( D = C \cup e \), where \( e \) is the set of vertices in some strongly connected component in \( \langle E, \rightarrow \rangle \) and \( e \cap C = \emptyset \). We denote the reflexive closure of this relation by \( \trianglerighteq \). A consistent cut sequence \( C_0, C_1, \ldots, C_k \) of \( (\mathcal{C}(G), \subseteq) \) satisfies that for each \( 0 \leq i < k \), \( C_i \triangleright C_{i+1} \). We say that a cut \( D \) is reachable from a cut \( C \) if \( C \subseteq D \).

Propositional temporal logics use a finite set of atomic propositions \( AP \), each one of which represents some property of the global state. A labeling function \( \lambda: \mathcal{C}(G) \to 2^{AP} \) assigns to each global state the set of predicates from \( AP \) that hold in it. In this paper we assume that atomic propositions are non-temporal regular predicates and their negations.

The formal syntax of RCTL+ is given below.
- Every predicate \( ap \in AP \) is an RCTL+ formula.
- If \( p \) and \( q \) are RCTL+ formulas, then so are \( p \lor q \), \( p \land q \), \( \text{EF}(p) \), \( \text{EG}(p) \), and \( \text{AG}(p) \).

Given a finite distributive lattice \( L = (\mathcal{C}(G), \subseteq) \), the formulas of RCTL+ are interpreted over the consistent cuts in \( \mathcal{C}(G) \). Let \( p \) be an RCTL+ formula and \( C \) be a consistent cut in \( \mathcal{C}(G) \). Then, the satisfaction relation, \( L, C \models p \) means that predicate \( p \) holds at consistent cut \( C \) in lattice \( L = (\mathcal{C}(G), \subseteq) \) and is defined inductively below. We denote \( C \models p \) as a short form for \( L, C \models p \),
when $L$ is clear from the context.

- $C \models ap$ iff $ap \in \lambda(C)$ for an atomic proposition $ap$.
- $C \models p \land q$ iff $C \models p$ and $C \models q$.
- $C \models p \lor q$ iff either $C \models p$ or $C \models q$.
- $C \models \text{EG}(p)$ iff for some consistent cut sequence $C_0, \ldots, C_k$ such that (i) $C_0 = C$, (ii) $C_k = \mathcal{E}$, (iii) $C_i \models C_{i+1}$ for $0 \leq i < k$, we have (iv) $C_i \models p$ for all $0 \leq i \leq k$.
- $C \models \text{AG}(p)$ iff for all consistent cut sequences $C_0, \ldots, C_k$ such that (i) $C_0 = C$, (ii) $C_k = \mathcal{E}$, (iii) $C_i \models C_{i+1}$ for $0 \leq i < k$, we have (iv) $C_i \models p$ for all $0 \leq i \leq k$.
- $C \models \text{EF}(p)$ iff for some consistent cut sequence $C_0, \ldots, C_k$ such that (i) $C_0 = C$, (ii) $C_k = \mathcal{E}$, (iii) $C_i \models C_{i+1}$ for $0 \leq i < k$, we have (iv) $C_i \models p$ for some $0 \leq i \leq k$.

We define $L \models p$ if and only if $L, \{\bot\} = p$. The formula $C \models \text{AG}(p)$ (resp. $C \models \text{EG}(p)$) intuitively means that for all consistent cut sequences (resp. for some consistent cut sequence) $C, \ldots, \mathcal{E}$, $p$ holds at every cut of the sequence. The formula $C \models \text{EF}(p)$ intuitively means that for some consistent cut sequence $C, \ldots, \mathcal{E}$, there exists a consistent cut that satisfies $p$.

We define RCTL as the subset of RCTL+ where disjunction and negation operators are not allowed.

The predicate detection problem is to decide whether the initial consistent cut of a distributed computation satisfies a predicate.

7 Algorithms for Computing Slices for Temporal Predicates

Our distributed program analysis tool POTA uses computation slicing for predicate detection. Mittal and Garg \cite{MittalGarg} also used computation slicing for efficient detection of predicates of the form $\text{EF}(p)$, $\text{EG}(p)$, $\text{AG}(p)$ for non-temporal regular $p$. However, their predicate detection algorithm is based on computing slices for non-temporal regular predicates. Therefore, it cannot be used for detecting nested temporal predicates such as $\text{AG}(p)$ when $p$ is a temporal predicate like $p = \text{EF}(q)$. In this section, we explain our slicing algorithms from \cite{Lan} for temporal regular predicates to enable efficient predicate detection for RCTL+ which also includes nested temporal predicates.

The slice of a computation with respect to a temporal predicate is the smallest computation that contains all consistent cuts of the given computation for which the predicate holds. We proved in \cite{Lan} that temporal predicates $\text{EF}(p)$, $\text{EG}(p)$, and $\text{AG}(p)$ are regular for regular $p$. Therefore, the slices for these temporal predicates are lean.

The input to each algorithm in this section is a computation $\langle E, \rightarrow \rangle$ and its slice with respect to a regular predicate $p$, that is, slice$(\langle E, \rightarrow \rangle, p)$. The output of each algorithm is an application of a temporal operator on the slice. For
example, in order to generate a slice with respect to $\text{AG}(\text{EF}(p))$, where $p$ is a
non-temporal regular predicate, we can use the slicing algorithms explained in
this section as follows: First, we compute the slice for $p$ using the algorithms in
$\text{FF}$ for non-temporal regular predicates. Then, we give this slice and
the computation to the $\text{EF}$ slicing algorithm to obtain slice$(\langle E, \rightarrow \rangle, \text{EF}(p))$.
Finally, the output of $\text{EF}$ slicing algorithm and the computation is given as
an input to $\text{AG}$ slicing algorithm to obtain slice$(\langle E, \rightarrow \rangle, \text{AG}(\text{EF}(p)))$.

Since the consistent cuts of the slice of a computation is a subset of con-
istent cuts of the computation, the slice can be obtained by adding edges to
the computation. In other words, the slice contains additional edges that do
not exist in the computation. For example, consider Figure 8(a) that displays
the slice of the computation in Figure 8 with respect to $\neg((x = 5) \land (y = 2))$.
The only consistent cut in the computation that does not satisfy the predicate
is $\{e_3, e_2, e_1, f_1, \bot\}$. By adding the edge $(f_2, e_3)$, we disallow this consistent
cut from the slice. Below, for computing slices for $\text{EF}(p)$, we will show which
dges we add to the computation. Similarly, since the consistent cuts of the
slice for $\text{AG}(p)$ is a subset of consistent cuts of the slice for $p$, the slice for
$\text{AG}(p)$ can be obtained by adding edges to the slice for $p$.

Now we explain Algorithm A1 in Figure 8 for generating the slice of a
computation with respect to $\text{EF}(p)$. From the definition of $\text{EF}(p)$, all con-
sistent cuts of the computation that can reach the greatest consistent cut that
satisfy $p$, say $W$, will also satisfy $\text{EF}(p)$ and furthermore these are the only
cuts that satisfy $\text{EF}(p)$. We can find the cut $W$ using slice$(\langle E, \rightarrow \rangle, p)$ when it
is nonempty. We construct the slice for $\text{EF}(p)$ from the computation so that
$W$ is the final cut of the slice. To ensure that all cuts which cannot reach $W$
do not belong to the slice, we add edges from $\top$ to the successors of events
in the frontier of $W$. Adding an edge from $\top$ to an event makes any cut
that contains the event trivial. Figure 11 shows the application of Algorithm
A1. Given the slice of the computation in Figure 8(a) for some predicate $p$ as
shown in Figure 11(a), first we compute the final cut of the slice for $p$, that is,
$\{e_2, f_3\}$. Then, on the computation, we add an edge from $\top$ to the successor
of $e_2$, that is $e_3$. The successor of $f_2$ does not exist so we do not add any other
dges. The resulting slice for $\text{EF}(p)$ is displayed in Figure 11(c).

Now we describe the $\text{AG}(p)$ slicing algorithm in Figure 8. We explained
above that to obtain the slice for $\text{AG}(p)$ we will add edges to the slice for
$p$ and eliminate consistent cuts that do not belong to slice for $\text{AG}(p)$. Now
we show which edges we should add. We claim that consistent cuts of the
slice$(\langle E, \rightarrow \rangle, p)$ that do not include vertex $e$ of each additional edge $(e, f)$
do not satisfy $\text{AG}(p)$. For simplicity, let the slice$(\langle E, \rightarrow \rangle, p)$ have a single
additional edge $(e, f)$. For example, consistent cuts $\{\bot\}, \{f_1, \bot\}, \{e_1, f_1, \bot\},$
and $\{e_2, e_1, f_1, \bot\}$ of the slice in Figure 8(a) do not include vertex $f_2$ of the
additional edge $(f_2, e_3)$. It is easy to see that these four consistent cuts do not
satisfy $\text{AG}(p)$ and therefore we should add edges to eliminate them. We now
give a proof sketch of the correctness of the algorithm for the simplified case
Fig. 4. (a) A slice of $\langle E, \rightarrow \rangle$ in Fig. 3 (b) the corresponding sublattice (c) The application of the temporal operator $\mathbf{EF}$ on the slice in (a) (d) the corresponding sublattice with a single additional edge. The proof for full case can be found in [28].

**Theorem 7.1** Given a computation $\langle E, \rightarrow \rangle$ and $\text{slice}(\langle E, \rightarrow \rangle, p)$, a consistent cut $D$ in $\langle E, \rightarrow \rangle$ satisfies $\mathbf{AG}(p)$ iff it includes vertex $e$ of the additional edge $(e, f)$ in $\text{slice}(\langle E, \rightarrow \rangle, p)$.

**Proof Sketch:**

If a consistent cut $D$ does not include vertex $e$ then there exists a consistent cut $H$ that can be reached from $D$ in the computation such that $H$ does not include $e$ but includes $f$. In this case, it is clear that $H$ does not satisfy $p$ since $(e, f)$ is an edge in the $\text{slice}(\langle E, \rightarrow \rangle, p)$ and every consistent cut of $\text{slice}(\langle E, \rightarrow \rangle, p)$ that includes $f$ must include $e$. Therefore from the definition of $\mathbf{AG}(p)$, $D$ does not satisfy $\mathbf{AG}(p)$.

Now we prove the other direction. If a consistent cut $D$ does not satisfy $\mathbf{AG}(p)$ then there exists a consistent cut $H$ reachable from $D$ such that $H$ does not satisfy $p$. We know that only the consistent cuts that include $f$ but not $e$ do not satisfy $p$. Since $H$ is reachable from $D$ and $H$ does not include $e$, we have that $D$ also does not include $e$. \hfill \Box

In Algorithm A2, for any additional edge $(e, f)$, we add an edge from vertex $e$ to vertex $\bot$. This ensures that consistent cuts of the computation that do not include vertex $e$ of any additional edge $(e, f)$ are disallowed from the slice, whereas the rest still belong to $\text{slice}(\langle E, \rightarrow \rangle, \mathbf{AG}(p))$. For example, consistent cut $\{e_1, f_1, \bot\}$ of the slice in Figure 4(a) does not include vertex $f_2$ of the additional edge $(f_2, e_3)$ in Figure 4(a), therefore we add an edge $(f_2, \bot)$ and obtain the slice in Figure 4(c). The cut $\{e_1, f_1, \bot\}$ cannot be a consistent cut of this new slice since it has to include vertex $f_2$. 

14
Algorithm A1

Input: A computation \( \langle E, \rightarrow \rangle \) and slice(\( \langle E, \rightarrow \rangle, p \))
Output: slice(\( \langle E, \rightarrow \rangle, \text{EF}(p) \))
Step 1. Let \( G \) be \( \langle E, \rightarrow \rangle \) and let \( W \) be the final cut of
slice(\( \langle E, \rightarrow \rangle, p \))
Step 2. If \( W \) exists then
Step 3. \( \forall e \in \text{frontier}(W) \): add an edge from
the vertex \( \top \) to \( \text{succ}(e) \) in \( G \)
Step 4. return \( G \)
Step 5. else return empty slice

Algorithm A2

Input: A computation \( \langle E, \rightarrow \rangle \) and slice(\( \langle E, \rightarrow \rangle, p \))
Output: slice(\( \langle E, \rightarrow \rangle, \text{AG}(p) \))
Step 1. Let \( G \) be \( \langle E, \rightarrow \rangle, p \))
Step 2. For each pair of vertices \( (e, f) \) in \( G \) such that,
\( (i) \) \( (e \rightarrow f) \) in \( \langle E, \rightarrow \rangle \), and
\( (ii) \) \( (e \rightarrow f) \) in \( G \)
add an edge from vertex \( e \) to the vertex \( \bot \)
Step 3. return \( G \)

Fig. 5. Algorithms for generating a slice with respect to \( \text{EF}(p) \) and \( \text{AG}(p) \)

The algorithm for \( \text{EG}(p) \) slicing is similar to the \( \text{AG}(p) \) slicing algorithm
and is explained in \[28\]. The complexity of the temporal slicing algorithms is
\( O(n|E|) \) \[28\].

Complexity of RCTL Predicate Detection: Given a predicate in
RCTL we can compute the slice for the predicate recursively from inside-out
by applying the appropriate temporal or boolean operator on the slices. It
is then easy to determine whether the predicate is satisfied by just checking
whether the initial state of the computation and the slice are the same. The
complexity of predicate detection is dominated by the complexity of com-
puting the slice with respect to a non-temporal regular predicate, which has
\( O(n^2|E|) \) complexity \[14,24\]. Therefore, the overall complexity of predicate
detection for RCTL is \( O(|p| \cdot n^2|E|) \), where \( |p| \) is the number of boolean and
temporal operators in \( p \). The predicate detection in RCTL+ has worst case
exponential-time complexity. However, the slice is in general much smaller
Fig. 6. (a) The slice of $\langle E, \rightarrow \rangle$ in Fig. 8 with respect to $\neg((x = 5) \land (y = 0))$ (b) the corresponding sublattice (c) The slice of $\langle E, \rightarrow \rangle$ in Fig. 8 with respect to $AG \neg((x = 5) \land (y = 0))$ (d) the corresponding sublattice than the computation which we validate with experiments in the next section.

8 Experimental Results

In order to evaluate the effectiveness of POTA, we performed experiments with scalable protocols, comparing our computation slicing based approach with partial order reduction based approach of SPIN [10]. All experiments were performed on a 1.4 Ghz Pentium 4 machine running Linux. We restricted the memory usage to 512MB, but did not set a time limit. The two performance metrics we measured are running time and memory usage. In the case of slicing both metrics also include the overhead of computing the slice. We use the symbol * to denote that the verification was not completed due to running out of memory.

We consider the following distributed programs distributed dining philosophers, primary-secondary, and GIOP protocols. Further experimental results can be obtained from POTA website [27].

**Distributed Dining Philosophers (ddph):** We use the Java protocol from [14] for this exercise and check the following properties. The complement of the safety property, that is, $\bigvee_{i,j \in 0..(n-1)} (EF(eat_i \land eat_j))$ where $i$ and $j$ denote philosophers next to each other. The complement of the liveness property, that is, $\bigvee_{i \in 0..(n-1)} (EF(hungryi \land EG(\neg eat_i)))$, for each philosopher $i$. Observe that the negation of a local predicate $\neg eat_i$ is also a local predicate and furthermore it is a regular predicate. Finally, we check the property $AG(EF(eat_i))$ which denotes that eating is possible from every state. Table
displays our results for the liveness property.

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>100</th>
<th>250</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>POTA</strong></td>
<td>T</td>
<td>0.14</td>
<td>0.17</td>
<td>0.19</td>
<td>0.23</td>
<td>0.22</td>
<td>0.49</td>
<td>3.54</td>
<td>10.37</td>
<td>18.1</td>
<td>137.2</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>0.18</td>
<td>0.29</td>
<td>0.36</td>
<td>0.42</td>
<td>0.5</td>
<td>0.92</td>
<td>1.57</td>
<td>4.5</td>
<td>6.8</td>
<td>33.4</td>
</tr>
<tr>
<td><strong>SPIN</strong></td>
<td>T</td>
<td>0.1</td>
<td>1.16</td>
<td>15.6</td>
<td>144.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>1.67</td>
<td>4.13</td>
<td>35.4</td>
<td>223.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Primary Secondary:** The primary secondary program concerns an algorithm designed to ensure that the system always contains a pair of processes acting together as primary and secondary. The property requires that there is a pair of processes $P_i$ and $P_j$ such that (1) $P_i$ is acting as a primary and correctly thinks that $P_j$ is its secondary, and (2) $P_j$ is acting as a secondary and correctly thinks that $P_i$ is its primary. Both the primary and secondary may choose new processes as their successor at any time. The complement of the safety property is $\text{EF} \land (\neg\text{Primary}_i \lor \neg\text{Secondary}_j \lor (\text{Secondary}_i \neq P_j) \lor (\text{Primary}_j \neq P_i))$ when $i, j \in \{0, \ldots, n-1\}, i \neq j$. Note that this predicate contains disjunction operators and the slice may not be lean. However, Table 2 shows that even in this case slicing can reduce the state space substantially.

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>POTA</strong></td>
<td>T</td>
<td>0.01</td>
<td>0.03</td>
<td>0.08</td>
<td>0.15</td>
<td>0.28</td>
<td>0.42</td>
<td>0.65</td>
<td>0.9</td>
<td>4.07</td>
<td>20.53</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>0.41</td>
<td>0.75</td>
<td>1</td>
<td>2.03</td>
<td>2.75</td>
<td>3.77</td>
<td>7.78</td>
<td>8.82</td>
<td>28.89</td>
<td>199.49</td>
</tr>
<tr>
<td><strong>SPIN</strong></td>
<td>T</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.12</td>
<td>0.38</td>
<td>2.51</td>
<td>7.92</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>1.57</td>
<td>1.57</td>
<td>1.67</td>
<td>2.29</td>
<td>5.05</td>
<td>21.95</td>
<td>81.54</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**GIOP:** In this section, we present experimental results for the General Inter-ORB Protocol (GIOP) which was verified in using SPIN.

The Common Object Request Broker Architecture (CORBA) describes the architecture of a middleware platform that supports the implementation of applications in distributed and heterogeneous environments. The CORBA standard is issued by OMG.

The ORB is the key component of the CORBA programming model. An ORB is responsible for transferring operations from Clients to Servers. This requires the ORB to locate a Server implementation (and possibly activate it), transmit the operation and its parameters, and finally return the results back to the Client.
The General Inter-ORB Protocol (GIOP) is the abstract protocol which is used for communications between CORBA ORBs. It specifies the transfer syntax and a standard set of message formats for ORB interoperation over any connection-oriented transport Protocol. GIOP is designed to be simple and easy to implement, while still allowing for reasonable scalability and performance. In order to allow server objects to move between different ORBs and have messages forwarded to them wherever they are, GIOP supports server migration.

Figure 7 displays the high level view of the Promela model of the GIOP protocol as depicted in [18]. The protocol consists of User, Client, Transport, Agent and Server processes. Here, we conduct experiments for 4 of the 8 LTL predicates used in [18] (properties (iv) and (v) are considered as one). Below, formulas express the complement of the property expressed in English.

(i) After sending a \textit{U}Request message a User should eventually receive the corresponding \textit{U}Reply message.
\[
\text{EF}(\text{URequestSent}_i \land \text{EG}(\neg \text{UReplyReceived}_i)), \text{ for all users } i.
\]

(ii) After sending an \textit{S}Request the GIOP-Agent should eventually receive a corresponding \textit{S}Reply.
\[
\text{EF}(\text{SRequestSent}_i \land \text{EG}(\neg \text{SReplyReceived}_i)), \text{ for all agents } i.
\]

(iii) Requests sent by a client are responded to eventually by a reply unless they have been cancelled.
\[
\text{EF}\left(\text{CRequestSent}_i \land \text{EG}(\neg \text{CReplyReceived}_i \lor \neg \text{CCancelSent}_i)\right), \text{ for all clients } i.
\]

(iv) If the user received no exception, its request was performed exactly once.
\[
\text{AG}\left(\neg \text{NoException}_i \lor (\bigvee_k \bigwedge_j \text{Server}_j\text{Processed}_i = m)\right), \text{ where } m = 1 \text{ if } k = j \text{ and } m = 0 \text{ otherwise, for all users } i \text{ and for all servers } j,k.
\]

(v) If the user received exception, its request was performed at most once.
\[
\text{AG}\left(\neg \text{SystemException}_i \lor (\bigvee_k \bigwedge_j \text{Server}_j\text{Processed}_i = m) \lor (\bigwedge_i \text{Server}_i\text{Processed}_i = 0)\right), \text{ where } m = 1 \text{ if } k = j \text{ and } m = 0 \text{ otherwise, for all users } i \text{ and for all servers } j,k,l.
\]

The full verification of GIOP by Kamel and Leue [18] even for the configu-
ration in Figure 7 with 10 processes was not completed due to state explosion. They could verify a simplified version of the protocol without server migration with 10 processes. To enable verification for larger number of processes, they used an approximation technique in SPIN called bit-state hashing where two bits of memory are used to store a reachable state. SPIN displays a state coverage number (hash-factor) at the end of a verification with bit-state hashing. With bit-state hashing, they could verify the unsimplified version of the protocol with 20 processes with 1.5 hash-factor, which means that the coverage was less than one percent since best coverage is obtained when the hash-factor is greater than 100.

We generated execution traces for a variety of GIOP architectures where we duplicated the User and Server blocks. In one case, we generated execution traces from unsimplified version of GIOP protocol where the total number of processes was increased to 250 and we completed full verification of these traces. In Table 3, we present experimental results for the GIOP models with server migration.

<table>
<thead>
<tr>
<th>Property (i)</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
<th>120</th>
<th>160</th>
<th>200</th>
<th>250</th>
</tr>
</thead>
<tbody>
<tr>
<td>POTA</td>
<td>T</td>
<td>0.2</td>
<td>0.24</td>
<td>4.6</td>
<td>50.8</td>
<td>183.8</td>
<td>1001.1</td>
<td>1291</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>1.7</td>
<td>1.3</td>
<td>1.9</td>
<td>16.3</td>
<td>33.7</td>
<td>63.2</td>
<td>76.7</td>
</tr>
<tr>
<td>SPIN</td>
<td>T</td>
<td>362.6</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>320</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property (v)</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>60</th>
<th>80</th>
<th>120</th>
</tr>
</thead>
<tbody>
<tr>
<td>POTA</td>
<td>T</td>
<td>0.1</td>
<td>5.3</td>
<td>5.6</td>
<td>60.7</td>
<td>218.8</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>0.4</td>
<td>4.4</td>
<td>21.6</td>
<td>150.4</td>
<td>301.4</td>
</tr>
<tr>
<td>SPIN</td>
<td>T</td>
<td>319.2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>305.4</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8.1 Discussion

In all execution trace verifications, SPIN could verify upto only 6 processes in ddpf, 9 in primary secondary and 10 in GIOP protocols, even when DCOL LAPSE and DMA compilation options were used. Observe that since we use a larger memory than the one used in [18], the verification of the unsimplified GIOP with 10 processes is now possible in SPIN. We obtain three orders of magnitude speed up and state space reduction compared to partial order reduction with SPIN as shown in GIOP experiments. Using our slicing based technique we could verify upto 250 processes in some cases. We also injected
faults into the traces and compared results of faulty protocols. Using SPIN, even with bit-state hashing enabled verification, the faults could not be found because the state spaces were too large and the coverage was low. Using POTA, the faults were easily found.

For problem sizes that preclude exhaustive program verification or exhaustive runtime verification, POTA proves to be an effective tool. Our technique is orthogonal to other reduction techniques, that is, one can always use POTA to reduce the state space as long as we can exploit the specification for computation slicing.

Acknowledgements: We would like to acknowledge Neeraj Mittal for his contribution in the implementation of POTA. We also thank Gerard J. Holzmann for discussion on SPIN.

References


