Computer Science Technical Reports | Computer Science

1-23-2011

Golok: Push-button Verification of Parameterized Systems

Youssef Wasfy Hanna
Iowa State University, youssef.hanna@gmail.com

David Samuelson
Iowa State University

Samik Basu
Iowa State University, sbasu@iastate.edu

Hridesh Rajan
Iowa State University, hridesh@iastate.edu

Recommended Citation

Hanna, Youssef Wasfy; Samuelson, David; Basu, Samik; and Rajan, Hridesh, "Golok: Push-button Verification of Parameterized Systems" (2011). *Computer Science Technical Reports*. 331.
http://lib.dr.iastate.edu/cs_techreports/331

Golok: Push-button Verification of Parameterized Systems

Abstract

Parameterized systems verification is a long-standing problem, where the challenge is to verify that a property holds for all (infinite) instances of the parameterized system. Existing techniques aim to reduce this problem to checking the properties on smaller systems with a bound on the parameter referred to as the "cut-off" such that if the property holds for system instances of size cut-off that implies that it holds for larger system instances. In most existing techniques, human guidance is required to deduce the invariants for the system's behavior, which are then used to compute cut-off. In contrast, we present an fully automatic sound method (but necessarily incomplete) for generating the cut-off that works for synchronous parameterized systems with heterogeneous processes communicating via single-cast and/or broadcast. Our technique is independent of the system topology and the property to be verified. Given the specification and the topology of the system, our technique generates the system-specific cut-off. We have realized our technique in a tool, Golok, which shows that it can be automated. We present the results of running Golok on 15 parameterized systems where we obtain smaller cut-offs than those presented in the existing literature for 14 cases.

Keywords

paramterized model checking, behavioral automaton, composition

Disciplines

Theory and Algorithms

Golok: Push-button Verification of Parameterized Systems

Youssef Hanna, David Samuelson, Samik Basu and Hridesh Rajan

TR #11-02
Initial Submission: January 23, 2011.

Keywords: parameterized model checking, behavioral automaton, composition

CR Categories:
D.2.4 [*Software/Program Verification*] Formal Methods
D.2.4 [*Software/Program Verification*] Model Checking
F.3.1 [*Specifying and Verifying and Reasoning about Programs*] Mechanical verification, Specification technique

Submitted on March 29, 2011.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Golok: Push-button Verification of Parameterized Systems

YOUSSEF HANNA, DAVID SAMUELSON, SAMIK BASU, and HRIDESH RAJAN
Iowa State University

Parameterized systems verification is a long-standing problem, where the challenge is to verify that a property holds for all (infinite) instances of the parameterized system. Existing techniques aim to reduce this problem to checking the properties on smaller systems with a bound on the parameter referred to as the *cut-off* such that if the property holds for system instances of size cut-off that implies that it holds for larger system instances. In most existing techniques, human guidance is required to deduce the invariants for the system's behavior, which are then used to compute cut-off. In contrast, we present an fully automatic sound method (but necessarily incomplete) for generating the cut-off that works for synchronous parameterized systems with heterogeneous processes communicating via single-cast and/or broadcast. Our technique is independent of the system topology and the property to be verified. Given the specification and the topology of the system, our technique generates the system-specific cut-off. We have realized our technique in a tool, Golok, which shows that it can be automated. We present the results of running Golok on 15 parameterized systems where we obtain smaller cut-offs than those presented in the existing literature for 14 cases.

Categories and Subject Descriptors: D.2.4 [**Software/Program Verification**]: Formal Methods; D.2.4 [**Software/Program Verification**]: Model Checking; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical verification

General Terms: Verification

Additional Key Words and Phrases: model checking, parameterized systems

1. INTRODUCTION

A parameterized system is a class of software system that consists of variable number of homogeneous processes, where the parameter denotes the number of homogeneous processes in the system [Manna and Pnueli 1995; 1990]. Since the parameter can vary, a

The work described in this article is the revised and extended version of the two papers presented at the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009) in Amsterdam, The Netherlands and at the 12th International Conference on Formal Engineering Methods (ICFEM 2010) in Shanghai, China. This work has been supported in part by the US National Science Foundation under grants CNS-06-27354, CNS-07-09217, and CCF-08-46059.
Authors' address: Y. Hanna, ywhanna@iastate.edu, Computer Science, Iowa State University, Ames, IA 50011. D. Samuelson, sralmai@iastate.edu, Computer Science, Iowa State University, Ames, IA 50011. S. Basu, sbasu@iastate.edu, Computer Science, Iowa State University, Ames, IA 50011. H. Rajan, hridesh@iastate.edu, Computer Science, Iowa State University, Ames, IA 50011.

parameterized system describes an infinite family of systems where instances of the family can be obtained by fixing the parameter [Emerson and Namjoshi 1998; Roscoe and Lazic 1998]. Verification of correctness of such systems amounts to verifying the correctness of every member of the infinite family described by the system. For example, for distributed mutual exclusion protocols [Wolper and Lovinfosse 1990], the objective is to verify that the critical section is accessed in a mutually exclusive fashion regardless of the number of processes participating in the protocol.

Given a parameterized system $sys(n)$ containing n processes and a property φ, verification of whether $sys(n)$ satisfies φ (denoted by $\forall n : sys(n) \models \varphi$) is undecidable in general [Apt and Kozen 1986]. A number of sound but incomplete verification techniques have been proposed and developed in the recent past, e.g. those that rely on abstraction [German and Sistla 1992; Pnueli et al. 2002; Clarke et al. 2006; 2008] and/or smart representation [Bouajjani et al. 2000; Abdulla et al. 2002; Abdulla et al. 2007; Fisman et al. 2008; Bouajjani et al. 2006; Bouajjani et al. 2007; Baldan et al. 2005; Baldan et al. 2008; Saksena et al. 2008; Llorens and Oliver 2004] of the system behavior and the property. In essence, these techniques depend on computing the *invariant* or the common global behavior of $sys(n)$ for all n and identifying the smallest $k < n$ such that $sys(k)$ exhibits that behavior. It can be shown that $sys(k) \models \varphi \Leftrightarrow \forall n \geq k : sys(n) \models \varphi$, i.e., verification of an infinite family of systems is reduced to verification of a single instance (the k-th instance) of the family; where k is referred to as the *cut-off*.

1.1 The Problems and their Importance

There are two main problems with existing techniques for generating cut-offs for parameterized systems. We elaborate them below.

(1) **Lack of automation makes adoption difficult.** The main drawback is that no extant technique is automatic. This is because there is no general method for computing system invariants. Rather, for every new system under verification, researchers manually study it to identify the invariants. This invariant is then used to generate the cut-off. We elaborate the different classes of systems for which different techniques are required.

- **System class.** For every class of systems, a different technique with unique principles is required to be able to generate cut-offs. For instance, Emerson and Namjoshi provided a model for verifying parameterized client-server systems that is decidable for properties expressed in an indexed propositional temporal logic [Emerson and Namjoshi 1996]. Since there are many parameterized systems that do not follow the client-server pattern, new and different techniques had to be implemented, where every technique is specific to exactly one class of systems. For instance, a new technique was required for resource allocation parameterized systems designed for the purpose of resolving conflicts in concurrent systems [Emerson and Kahlon 2002]. Again, this work couldn't be used on cache coherence protocols, for which a different technique was implemented [Emerson and Kahlon 2003].
- **System topology.** Similar to the problem of dependency on system class, systems with different topologies often require different techniques. For instance, the work focusing on parameterized systems that follow a token passing model where the system is in the form of a ring topology and a single token is passed among processes in a clockwise fashion [Emerson and Namjoshi 1995] could not work on parameter-

ized systems with full mesh topologies (i.e. all processes are connected together), therefore a new technique was required [Emerson and Kahlon 2003].

The lack of automation has a real impact on the applicability and eventual uptake of these techniques in software verification. An average user may not have the time or the necessary expertise to learn the theories behind cut-off generation for each new system that they want to verify.

(2) **Generic techniques may lead to large cut-offs, thus increased verification cost.** A class of solutions for the parameterized systems is able to generate cut-offs that are generic enough to work on any system. A problem with such techniques is that they often produce larger cut-off values. For example, Emerson and Namjoshi [Emerson and Namjoshi 1995] show that for any system with ring topology following the token passing model, the cut-off for properties involving two neighbor processes is 4, whereas we prove that tighter bounds can be obtained if the behavior of the participating processes in the parameterized system is considered [Hanna et al. 2009]. Having a tighter cut-off may translate to reduction in verification cost, since the system to be verified will be smaller in size. While there has been work that focused on generating protocol specific cut-off (i.e. [Emerson and Kahlon 2003]), they work only for certain protocols and it seems hard to generalize them.

1.2 Contributions to the State-of-the-art

We have designed, implemented, and evaluated fully automatic sound method but incomplete (due to the undecidability of the problem) for computing the cut-off value for synchronous parameterized systems where multiple type of homogeneous processes may exist. Such systems are typically called multi-parameterized systems. We will denote a multi-parameterized system by $sys(\bar{n}_t)$. Here, $\bar{n}_t := n_1, n_2, \ldots, n_t$ is the collection of parameters for t types of processes, where n_p denotes the number of processes of type p. Singly-parameterized system is a system with just one type of parameterized process.

Given a multi-parameterized system ($sys(\bar{n}_t)$) the objective of all parameterized system verification techniques is to check whether the system satisfies a given property for all possible valuations of each n_p in $\bar{n}_t$ where $p \in [1, t]$. Our technique achieves this objective by computing the cut-off values for each of the parameters in the system.

Our technique consists of two steps. The first step is to compute the set of *maximal* behavior of the system that can be induced by a process of every type *in an arbitrary environment*. The second step is to identify an instance of the parameterized system, $sys(\bar{k}_t)$ (where $\bar{k}_t := k_1, k_2, \ldots, k_t$), that can exhibit the maximal behavior for every process type. We prove that the parameter values corresponding to this instance is the cut-off; more precisely, for any $\mathtt{LTL}\backslash\mathtt{X}$ (Linear Temporal Logic without "next" operator) property φ which involves either one process or two adjacent processes communicating directly with each other, the instance of the parameterized system with the cut-off valuation for the parameters satisfies the property if and only if any other larger (in terms of parameter values) instance satisfies the same property (i.e., $\forall p \in [1, t] : n_p \geq k_p, sys(\bar{k}_t) \models \varphi \Leftrightarrow sys(\bar{n}_t) \models \varphi$).

Since the first step in our technique does not make assumptions about system structure, communication pattern, and topology, our technique is, unlike most existing work, independent of these. In addition, our technique generates cut-off values for systems where processes can communicate via both single-cast and broadcast. Finally, our technique does not depend on actual properties to be verified for the parameterized system; the results of our technique are applicable to any properties involving any one process and properties involving any two adjacent processes communicating directly with each other that are not necessarily of the same process type.

Contributions. In summary, contributions of this work are:

(1) A fully automatic but incomplete method for generating cut-offs for synchronous parameterized system that is independent of the topology between the processes in the system and the property to be verified. Our technique works for both singly and multi-parameterized systems and for system with communication via single-cast and/or broadcast. We prove the soundness of our technique, i.e., if our method terminates then it does with a cut-off $\bar{k}_t = k_1, k_2, \ldots, k_t$ for a system with t types of processes.

(2) Our technique is system specific and as such the computed cut-off is also system specific. This allows us to obtain different bounds to different types of parameterized systems even when the underlying communication topology of the systems under consideration are identical. [Emerson and Namjoshi 1995] proved that for parameterized systems with ring topology where processes communicate through a token, the cut-off $\bar{k}_1$ is 4 for properties of involving neighbor processes. We show that tighter bounds can be obtained if the behavior of the participating processes in the parameterized system is considered. For example, using our technique, the cut-off for the parameterized token ring protocol is 2 for the same class of properties.

(3) We have implemented our technique in an automated tool that we call *Golok*[1]. Golok is available for free for download. Golok serves as a proof of feasibility of our work.

(4) We have studied a large number of systems from different classes of parameterized systems. For instance, we generated cut-offs for the Distributed Mutual Exclusion [Wolper and Lovinfosse 1990] and Dining Philosophers [Dijkstra 1978] protocols in the class of ring topology systems, Spin Lock [Anderson 1990] in the class of systems with star topology, Right-Left Dining Philosophers [Emerson and Kahlon 2002] in both the classes of multi-parameterized systems and resource allocation systems and finally the cache coherence protocols [Handy 1993] in the class of parameterized systems with broadcast-based communication. For 14 out of 15 of these cases the generated cut-off values was smaller than the ones generated by existing work. For the remaining case, it was the same.

1.3 Contributions over Our Previous Work

In our previous work [Hanna et al. 2009], we proposed an initial version of this technique that followed the same steps as discussed earlier in this section to generate cut-off values for parameterized systems. We showed that the generated system-specific cut-off values are smaller than existing work; however, it worked only on singly-parameterized systems. In our following work [Hanna et al. 2010], we proved that our technique works for multi-parameterized systems and presented our tool Golok that implements our technique.

This work significantly enhances our own previous work in the following ways:

(1) In our previous works [Hanna et al. 2009; Hanna et al. 2010], we modeled the behavior of a process in such a way that a process is always blocked after sending an event to the environment. In other words, it cannot send any other event until it receives a reply from the environment to the event it sent. Similarly, when a process

[1] A cutting tool typically used in Indonesia and the Philippines.

replies to its neighbor, it cannot send an event on its own. This way of modeling the process behavior prevented concurrency between processes as two neighbor processes cannot perform actions with their other neighbors independently from each other. In this work, we have reformulated our technique such that communication between processes is purely synchronous. This helped us extend the applicability of our technique.

(2) Our previous work allowed only one-to-one communication between processes in a system. We have since discovered that our technique also applies to systems where processes can communicate via both single-cast and broadcast. Following this discovery we have enhanced our representation, formalisms and implementation to handle broadcast-based systems. We have also analyzed several broadcast-based systems.

These technical underpinnings extended the applicability of our technique as follows.

(1) Adding these two new features allowed us to generate cut-off for new classes of systems such as 7 snoop-based cache coherence protocols [Handy 1993] and German's cache coherence protocols [German 2000]. As a result, we now present a case study of 15 parameterized systems, which is also new to this article.

(2) Our enhanced way of modeling the processes resulted in smaller cut-off values for 14 out of 15 case studies that we verified, which in turn may reduce the verification cost of these systems.

Organization. The rest of the paper is organized as follows. Section 2 describes our technique for specifying a protocol and building a parameterized system using a variant of the Dining Philosophers Protocol as an illustrative example. Section 3 describes how the maximal behavior of a process in the context of any environment is generated as well as our procedure for generating the cut-off. Proof of soundness of our technique is presented in Section 4. Section 5 presents our tool Golok and its different components. Section 6 describes our case studies. We also analyze performance of our tool Golok. These results are presented in Section 7. Section 8 discusses related work and Section 9 concludes.

2. PARAMETERIZED SYSTEMS

A parameterized system $sys(\bar{n}_t)$ can be described by the collective behavior of all the processes of different types interacting with each other, where $\bar{n}_t = n_1, n_2, \ldots, n_t$ is the system parameters for the t types of processes in the system. The key idea behind our approach is to provide a mechanism for specifying the behavior of a process in the parameterized system as a collection of *atomic steps*, which we call *behavioral automaton*.

An important property of our specification technique is that it enables automatic composition of these behavioral automata to obtain the maximal behavior a process of each process type $p \in [1, t]$ can induce in an arbitrary environment.

One direct benefit of this property is that it helps us reduce the problem of finding the cut-off value $\bar{k}_t$ for the system parameters to an equivalence detection problem between the maximal behavior that a process of every type $p \in [1, t]$ can induce in an arbitrary environment and the parameterized system of size $\bar{k}_t$. As we show in Sections 3.3 and 5, by providing a sound but incomplete method and a tool that implements our method, respectively, this problem can be easily automated.

Illustrative Example. The terminology used in this paper and the salient aspects of the proposed technique are explained using a variant of the Dining Philosophers protocol [Dijkstra 1978], a model illustrating a classic multi-process synchronization problem. The

```
# This diner picks up left fork first
process left-diner {
 [l-idle]              ->[l-waitl, l-askl]
 [l-waitl, l-lfree] ->[l-hasl]
 [l-hasl]              ->[l-waitr, l-askr]
 [l-waitr, l-rfree] ->[l-eat]
 [l-eat]               ->[l-hasr, l-rel+]
}
```

(a)

```
# This diner picks up right fork first
process right-diner {
 [r-idle]              ->[r-waitr, r-askr]
 [r-waitr, r-rfree] ->[r-hasr]
 [r-hasr]              ->[r-waitl, r-askl]
 [r-waitl, r-lfree] ->[r-eat]
 [r-eat]               ->[r-hasl, r-rel+]
}
```

(b)

```
process fork {
 [free, l-askl] -> [l-lasked]
 [free, l-askr] -> [l-rasked]
 [free, r-askr] -> [r-rasked]
 [free, r-askl] -> [r-lasked]
 [busy, r-rel] ->[free]
```

```
 [l-lasked] -> [busy, l-lfree]
 [l-rasked] -> [busy, l-rfree]
 [r-rasked] -> [busy, r-rfree]
 [r-lasked] -> [busy, r-lfree]
 [busy, l-rel] ->[free] }
```

(c)

Fig. 1. Behavioral Automata for (a) “Left” Philosophers (b) “Right” Philosophers (c) Forks

standard definition models processes as philosophers sitting in a circle (a ring topology) with a fork between each two philosophers. The main objective of a philosopher process is to acquire the fork to her left and right and start eating. We use a variant of this protocol referred to as the Right-Left Dining Philosophers (RLDP) algorithm [Emerson and Kahlon 2002], where two types of philosophers exist: “Left” philosophers grab the left fork first and “Right” philosophers grab the right fork first. In this protocol, adjacent philosophers are of different types; therefore, the number of “Left” and “Right” philosophers is equal.

2.1 Representing Process Behavior in terms of Behavioral Automata

A homogeneous process in our work is specified in terms of a *behavioral automaton*. A behavioral automaton describes an atomic side-effect free action of a process. There are two types of behavioral automaton: a sender automaton and a receiver automaton. These model a send and a receive action respectively. They are defined as follows:

Definition 2.1 Sender Automaton. *A sender automaton $Snd = (q_I, q_F, \Delta_I, \Delta, \Delta_F, E)$, where q_I is the initial state, q_F is the final state, $\Delta_I = \emptyset$ is the initial transition relation, $\Delta = \{(q_I, q_F)\}$ is the internal transition relation, $\Delta_F = \{\{q_F\} \times E\}$ is the output transition relation and $\emptyset \subset E \subset \{e, e^+\}$ is the set of events, where e is an event and e^+ is a broadcast of event e. We write $q_I \longrightarrow q_F$ if $(q_I, q_F) \in \Delta$ and $q_F \xrightarrow{e} \bullet$ if $(q_F, e) \in \Delta_F$.*

Definition 2.2 Receiver Automaton. *A receiver automaton $Rcv = (q_I, q_F, \Delta_I, \Delta, \Delta_F, E)$, where q_I is the initial state, q_F is the final state, $\Delta_I = \{(q_I, e)\}$ is the initial transition relation, $\Delta = \{(q_I, q_F)\}$ is the internal transition relation, $\Delta_F = \emptyset$ is the output transition relation and $E = \{e\}$ is the set of events. We write $\bullet \xrightarrow{e} q_I$ if $(e, q_I) \in \Delta_I$ and $q_I \longrightarrow q_F$ if $(q_I, q_F) \in \Delta$.*

Definition 2.3 Behavioral Automaton. *A behavioral automaton A is either a sender automaton or a receiver automaton.*

A behavioral automaton describes the state in which a process can be, and what action it can take when it is in this state[2].

Figures 1(a), (b) and (c) display the behavioral automata for philosophers of types "Left" and "Right" and the "Fork" processes of the RLDP protocol, respectively. The statement of the form `[q]->[q',e]` denotes a sender automaton with $q \to q'$ and $q' \xrightarrow{e} \bullet$, and the statement of the form `[q,e]->[q']` denotes a receiver automaton with $\bullet \xrightarrow{e} q$ and $q \to q'$. The reason of modeling transitions in our automata this way as opposed to the commonly used form $q_I \xrightarrow{e} q_F$ is to be able to differentiate between send and receive actions.

The automaton in Line 3 in Figure 1(a) is an example of a sender automaton. It presents the behavior of a philosopher of type "Left" who, while not eating (i.e. state `l-idle`), without any external stimuli, changes its state to `l-waitl` (i.e. waiting for the left fork) and sends the request for the left fork (event `l-askl`). This request is received by a "Fork" process, and the behavior of such action is modeled in the receiver automaton in Line 2 in Figure 1(c). The automaton represents the action of receiving the request for the fork while not being used by any other philosopher (state `free`), then the state of the fork is changed to `l-lasked`. In the automaton defined in Line 8 in Figure 1(c), the fork sends an acknowledgement `l-lfree`, and changes its state to `busy`. The action of the philosopher receiving the acknowledgement is modeled by the receiver automaton in Line 4 in Figure 1(a), where the philosopher changes her state to reflect that she has the left fork (state `l-hasl`). A "Right" philosopher can perform similar actions (Figure 1(b)), only that she asks for the right fork first.

While philosophers pick the forks one at a time, we modeled the protocol so that every philosopher releases both forks at the same time. This is done via broadcast of `rel` event. Sender automaton in Line 6 in Figure 1 is an example of such broadcast. We modeled the protocol in such a way to show that, in our technique, a system can have communication via both single-cast and broadcast (as opposed to [Emerson and Kahlon 2002] where only broadcast can be used to acquire the forks).

Every process type is specified a collection of behavioral automata that defines all the atomic actions a process of this type can take. The set of these collections determines all the actions that can take place in the system. More precisely:

DEFINITION 2.4 PROCESS AND SYSTEM SPECIFICATION. *A process specification for process type p is $\textsf{Prot}_p = (\mathcal{A}_p, q_{I_p})$ where $\mathcal{A}_p = \{A_1, A_2, \ldots, A_m\}$ is a set of behavioral automata and q_{I_p} is the initial process state such that q_{I_p} is the initial state of an automaton A_i and $A_i \in \mathcal{A}_p$.*

Let $\textsf{Prot}_p = (\mathcal{A}_p, q_{I_p})$ and Q_p be the set of all states in the all the behavioral automata in $\mathcal{A}_p$. A system specification for a system with t types of processes is $\textsf{Prot} = \bigcup_{1 \le p \le t} \textsf{Prot}_p$ such that, $\forall i, j \in [1, t], i \neq j : Q_i \cap Q_j = \emptyset$. At least one process specification $\textsf{Prot}_p$ in $\textsf{Prot}$ must have q_{I_p} as the initial state of a sender automaton Snd_i such that $Snd_i \in \mathcal{A}_p$.

In our example, there are three process specifications; the first is for the "Left" philosophers $\textsf{Prot}_l = (\mathcal{A}_l, \texttt{l-idle})$ whose set of behavioral automata $\mathcal{A}_l$ are displayed in Fig-

[2] Behavioral automata are closely related to the Input/Output automata [Lynch and Tuttle 1989], where one automaton describes the different actions a process can do and the pre/post conditions for every action. The main difference between behavioral automaton and I/O automaton is that the former enforces that one automaton represents only one atomic send/receive action, whereas the latter allows more than one action to be merged. This constraint on behavioral automaton allows more fine-grained interleaving between processes compared to the I/O automaton. This property is crucial for computing the maximal behavior as we will demonstrate in Section 3.2.

ure 1(a), the second for the "Right" philosophers $\mathsf{Prot}_r = (\mathcal{A}_r, \texttt{r-idle})$ with behavioral automata displayed in Figure 1(b), and the third is for the forks $\mathsf{Prot}_f = (\mathcal{A}_f, \texttt{free})$ with the behavioral automata displayed in Figure 1(c), where the types "Left", "Right" philosophers and "Fork" are represented by the letters l, r and f, respectively.

To allow a "Left" philosopher to initiate her behavior by picking up her left forks, the process specification of the "Left" philosophers has as the initial process state the state `l-idle`, which is the initial state of the sender automaton displayed in Line 3 in Figure 1(a). A "Left" philosopher in this state can thus act according to the behavior described by this automaton and acquire the left fork. Similarly, the process specification of the "Right" philosophers has as the initial process state the state `r-idle`, which is the initial state of sender automaton in Line 3 of Figure 1(b) responsible for acquiring the right fork. On the other hand, the initial automaton for the "Fork" process specification has as its initial process state the state `free` which is the initial state of the receiver automata in Lines 2-5 in Figure 1(c). The initial process state of the fork allows it to be ready for any request from a "Left" or "Right" philosopher.

2.2 Behavior of a Parameterized System

Any system behavior is constrained by the topology that describes which processes in the system can directly communicate with each other.

Definition 2.5 Communication Topology. *Given a system specification $\mathsf{Prot} = \bigcup_{1\leq p\leq t} \mathsf{Prot}_p$, where t is the number of different types of processes and $\mathsf{Prot}_p = (\mathcal{A}_p, q_{I_p})$, a topology is a set of tuples, $\mathsf{Topo} \subseteq E \times (\mathcal{I} \times \mathcal{T}) \times (2^{\mathcal{I}\times\mathcal{T}})$, where $E = \bigcup_{1\leq p\leq t}\bigcup_{1\leq r\leq l_p}\{E_r : E_r \text{ is set of events in } A_{r_p} \in \mathcal{A}_p\}$, $\mathcal{I} \in \mathbb{N}$ is the domain of number of processes of any type, and $\mathcal{T}$ is the domain of types. A tuple $(e, i_{p1}, R) \in \mathsf{Topo}$ where $R = \{(\mathcal{I}_l, p_l) \mid l \in [1,t]\}$ implies that output e from i^{th} process of type p_1 can be consumed by at most one of the processes in the set R. A tuple $(e^+, i_{p1}, R) \in \mathsf{Topo}$ implies that e is consumed by* all *processes in R as e in this case is sent via broadcast.*

For our example, there are two constraints on communication patterns: first that adjacent philosophers are of different types (therefore there is an equal number of "Left" and "Right" philosophers), and second that it is a ring topology. For instance, the topology for the system instance containing 1 "Left", 1 "Right" philosophers and 2 forks is

$$\mathsf{Topo} = \{(\texttt{l-askl}, 1_l, \{2_f\}),\ (\texttt{r-askr}, 1_r, \{2_f\}),\ (\texttt{l-rel}^+, 1_l, \{1_f, 2_f\}), \ldots\}$$

In this topology (Figure 2(a)), "Left" philosopher 1_l has the fork 2_f on her left side, therefore the philosopher's request for her left fork `l-askl` goes to this fork. This fork is also shared with "Right" philosopher 1_r (on the right side of 1_r, therefore it receives the request for right fork `r-askr` sent from 1_r). The event for releasing forks `l-rel` sent by the "Left" philosopher (not shown in the figure) 1_l is *simultaneously* received by forks 1_f and 2_f as the release event is sent via broadcast. We now precisely define a parameterized system.

Definition 2.6 Parameterized System. *Given a specification Prot with t different types of processes, a parameterized system containing n_p number of processes of type p ($p \in [1,t]$) is defined as $sys(\bar{n}_t) = (S, S_I, T, \mathsf{Topo})$, where $\bar{n}_t := n_1, n_2, \ldots, n_t$, S is the set of system states, $S_I \subseteq S$ is the set of initial system states and $T \subseteq S \times E \times S$ is the transition relation. A system state in S contains $\sum_{p=1}^{t} n_p$ process states q_{i_p}, where q_{i_p} is the state of the i-th process of type p.*

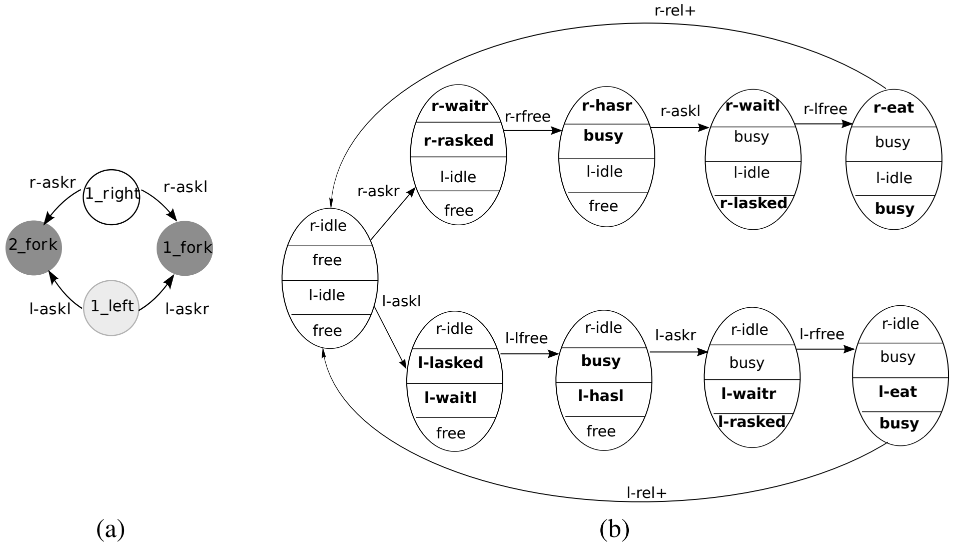


Fig. 2. (a) Topology of the system instance $sys(1_r, 1_l, 2_f)$ of the RLDP Protocol with one philosopher of each type and two forks (b) The system instance $sys(1_r, 1_l, 2_f)$

We use $s \xrightarrow{e} s'$ *to denote* $(s, e, s') \in T$. *Let* $p_1, p_2 \in [1, t], i \in [1, n_{p1}]$, $j \in [1, n_{p2}]$, *and* $R = \{l_p \mid l \in [1, n_p] \ \wedge \ p \in [1, t]\}$ *a set of processes, we define the transition relation* T *as follows*

—*A single-cast transition* $s \xrightarrow{e} s' \in T$, *where* $s = \langle q_{i_{p1}}, q_{j_{p2}}, Q\rangle$ *and* $s' = \langle q'_{i_{p1}}, q'_{j_{p2}}, Q\rangle$ *if*

$$\begin{array}{l}\exists Snd_{p1} = (q_{i_{p1}}, q'_{i_{p1}}, \Delta_{I_{p1}}, \Delta_{p1}, \Delta_{F_{p1}}, \{e\}) \in \mathcal{A}_{p1} \in \textsf{\textit{Prot}}_{p1} \wedge \\ \exists Rcv_{p2} = (q_{j_{p2}}, q'_{j_{p2}}, \Delta_{I_{p2}}, \Delta_{p2}, \Delta_{F_{p2}}, \{e\}) \in \mathcal{A}_{p2} \in \textsf{\textit{Prot}}_{p2} \wedge \\ (e, i_{p1}, R) \in \textsf{\textit{Topo}} \ \wedge \ j_{p2} \in R\end{array}$$

In the above, Q *represents the states of processes other than* i_{p1} *and* j_{p2} *(these states are unaffected by the transition).*

—*A broadcast transition* $s \xrightarrow{e^+} s' \in T$, *where* $s = \langle q_{i_{p1}}, Q_R, Q\rangle$ *and* $s' = \langle q'_{i_{p1}}, Q'_R, Q\rangle$ *if*

$$\begin{array}{l}\exists Snd_{p1} = (q_{i_{p1}}, q'_{i_{p1}}, \Delta_{I_{p1}}, \Delta_{p1}, \Delta_{F_{p1}}, \{e^+\}) \in \mathcal{A}_{p1} \in \textsf{\textit{Prot}}_{p1} \wedge \\ \forall l_p \in R, \exists Rcv_p = (q_{l_p}, q'_{l_p}, \Delta_{I_p}, \Delta_p, \Delta_{F_p}, \{e\}) \in \mathcal{A}_p \in \textsf{\textit{Prot}}_p \wedge \\ (e^+, i_{p1}, R) \in \textsf{\textit{Topo}}\end{array}$$

In the above, $Q_R = \bigcup_{l_p \in R}\{q_{l_p}\}$ *and* $Q'_R = \bigcup_{l_p \in R}\{q'_{l_p}\}$ *represent the states of processes in set* R *at system states* s *and* s', *respectively.* Q *represents the states of processes other than* i_{p1} *and processes in* R *(the states in* Q *are unaffected by the transition).*

We define the functions $sender(s, e, s') = i_p$ *and* $receiver(s, e, s') = \{j_{p'} \mid p' \in [1, t] \ \wedge \ j \in [1, n_{p'}]\}$ *as the functions that produce the indexes of the processes and their types that sent and received event* e, *respectively, at the system state* s *that caused the transition* $s \xrightarrow{e} s'$ *to occur.*

Figure 2(b) shows the system instance $sys(1_r, 1_l, 2_f)$ which is the system of two philosophers with one philosopher of type "Right", one of type "Left", and two forks. Each system state contains four process states, one for every philosopher and one for each of the two forks. Two possible transitions can happen from the initial system state: the transition on the top left belongs to the "Right" philosopher 1_r requesting her right fork 2_f by sending event `r-askr`. The state of the "Right" philosopher changes from `r-idle` to `r-waitr` as described by the sender behavioral automaton on Line 3 in Figure 1(b). The state of the fork 2_f on the right side of the philosopher changes from `free` to `r-rasked` as described by the receiver behavioral automaton on Line 4 in Figure 1(c). This transition denoted as $(s, \texttt{r-askr}, s')$, the sender in this transition is $sender(s, \texttt{r-askr}, s') = 1_r$, and the receiver in this transition $receiver(s, \texttt{r-askr}, s') = \{1_f\}$.

Similarly, the bottom left transition models the request of the "Left" philosopher 1_l for her left fork 2_f. The top sequence of transitions in the figure models the behavior of the "Right" philosopher 1_r getting her right fork, her left fork, then release forks, where the transition at the top of the figure labeled as $\texttt{r-rel}^+$ is modeling sending the release event via broadcast, where the two forks are released simultaneously. The lower sequence of transitions models the behavior of the "Left" philosopher 1_l who gets the left fork, then the right one, then releases both forks.

3. CUT-OFF COMPUTATION FOR PARAMETERIZED SYSTEMS

In this section, we describe our technique for computing the cut-off value for a parameterized system. Given the specification for all process types in the system as behavioral automata and the topology as input, our technique consists of two steps. First, it computes the maximal behavior a process of each type can induce when it initiates the protocol *in any environment*. Second, it finds a parameterized system instance whose behavior exhibits all the maximal behaviors that can be induced by processes of different types (if such an instance exists). We prove that the size of this system instance is the cut-off for the parameterized system. We start by describing what is the maximal behavior of a system instance induced by a process (Section 3.1), then we proceed with the computation of all possible maximal behavior induced by a process (Section 3.2) and finally we describe our technique for generating the cut-off (Section 3.3).

3.1 Maximal Behavior Induced by a Process in a Specific System Instance

Intuitively, the maximal behavior induced by a process of type p in the context of a specific system instance is all possible sequences of transitions that can be caused when a process of that type initiates the protocol by sending an event e. We will use π (with appropriate subscripts) to denote sequence of events.

To compute the maximal behavior induced by process i_p, we first check if the process can perform an action on its own (i.e. start sending an event without being triggered by any external stimuli). As mentioned in Section 2.2 (Definition 2.4), a process of type p can initiate the protocol if the initial state of the process defined in the process specification $\mathsf{Prot}_p = (\mathcal{A}_p, q_{I_p})$ is the initial state of sender automaton $Snd_p \in \mathcal{A}_p$.

If process i_p is able to send an event e_0 on its own, a sequence of events $\pi_p \in \textsc{Max}_p(sys(\bar{k}_t))$ will have $\pi_p[0] = e_0$. For all processes involved in such transition (i.e. the sender and receiver(s) of e_0), if any of these processes can send an event e_1, then $\pi_p[1] = e_1$. $\pi_p[2] = e_2$ if the sender or the receiver of e_1 were able to send e_2, and so on. The set $\textsc{Max}_p(sys(\bar{k}_t))$ will have all the sequences of events that are possible when a process i_p in $sys(\bar{k}_t)$ initiates the protocol. We now formally define the maximal behavior induced by a process of type p in $sys(\bar{k}_t)$.

DEFINITION 3.1 MAXIMAL BEHAVIOR INDUCED BY A PROCESS OF TYPE p IN $sys(\bar{k}_t)$. *Given a system specification $\textsf{Prot} = \bigcup_{1 \leq p \leq t} \textsf{Prot}_p$ where t is the number of different types of processes and $\textsf{Prot}_p = (\mathcal{A}_p, q_{I_p})$ and given a parameterized system instance $sys(\bar{k}_t)$, the maximal behavior induced by a process of type $p \in [1,t]$ in $sys(\bar{k}_t)$, denoted by $\textsc{Max}_p(sys(\bar{k}_t))$, is*

$$\textsc{Max}_p(sys(\bar{k}_t)) = \{\pi_p \mid \forall x \geq 0 : \pi_p[x] = \pi[h^\pi_p(x)] \wedge \eta_0 \in S_I \wedge \forall y \geq 0 : \eta_y \overset{\pi[y]}{\rightarrow} \eta_{y+1}\}$$

In the above, $h^\pi_p(x) = y$, such that

(1) $\exists i \in [1, k_p]$, $\eta_y = \langle q_{i_p}, Q \rangle$, $\eta_{y+1} = \langle q'_{i_p}, Q' \rangle : q_{i_p} = q_{I_p} \in \textsf{Prot}_p$, $\{q_{i_p} \rightarrow q'_{i_p}\} = \Delta, \{q'_{i_p} \overset{\pi[y]}{\rightarrow} \bullet\} = \Delta_F$ *where* $\Delta, \Delta_F \in Snd_p \in \mathcal{A}_p$ *and,* $\forall z \in [h^\pi_p(0), h^\pi_p(x-1)] : sender(\eta_{h^\pi_p(z)}, \pi[h^\pi_p(z)], \eta_{h^\pi_p(z)+1}) \neq i_p$

In the above, Q and Q' represent the states of processes other than q_{i_p} and q'_{i_p}, respectively.

(2) $sender(\eta_y, \pi[y], \eta_{y+1}) \in \{sender(\eta_{h^\pi_p(x-1)}, \pi[h^\pi_p(x-1)], \eta_{h^\pi_p(x-1)+1}), receiver(\eta_{h^\pi_p(x-1)}, \pi[h^\pi_p(x-1)], \eta_{h^\pi_p(x-1)+1})\}$ *and,* $\forall z \in [h^\pi_p(x-1)+1, y-1]$, $sender(\eta_y, \pi[y], \eta_{y+1}) \notin receiver(\eta_z, \pi[z], \eta_{z+1})$

In our example, since both types of philosophers can initiate the protocol, there exists a set of maximal behavior for each type. In the system instance $sys(1_r, 1_l, 2_f)$ displayed in Figure 2(b), the set of maximal behavior for each type of philosopher contain only one path; for the "Right" philosopher, the only path in her set of maximal behavior in this system instance is the upper path. This path represents the maximal behavior the philosopher 1_r can induce in this system instance. Similarly, the set of maximal behavior for the "Left" philosopher 1_l in this system instance contains only one path, the lower path in Figure 2(b).

3.2 Maximal Behavior Induced by a Process in Any Environment

In this section, we describe how to compute the maximal behavior a process of type p can induce *in any environment*. Our method takes this maximal behavior and generates the cut-off value for the parameterized system by finding one system instance that can exhibit the maximal behavior for every process type. We prove that the size of this instance is the cut-off value for the system.

The maximal behavior induced by a process of type p is computed by *chaining* the output event from all sender automata $Snd_i \in \mathcal{A}_p$ whose initial states are the same as the initial process state q_{I_p} in the process specification $\textsf{Prot}_p$ with all other receiver automata in the system specification with the same input event. For every chaining between a sender and a receiver automata, we check the final states of both the automata involved in the chaining; if these states are also input states for some other sender automata, chaining is done between the corresponding sender automata and all receiver automata that has an input event that is the same as the output event of these sender automata. The chaining ends when there are no more automata to chain. As there exists a finite number of automata in any system specification, the process of automata chaining is guaranteed to be finite. We refer to the result of such chaining as $1\mathrm{E}_p$ and we show that $1\mathrm{E}_p$ includes all possible behavior induced by the process of type p.

DEFINITION 3.2 $1\mathrm{E}_p$. *Given a system specification $\mathsf{Prot} = \bigcup_{1\leq p\leq t} \mathsf{Prot}_p$ with t different types of processes, $\mathsf{Prot}_p = (\mathcal{A}_p, q_{I_p})$, let Q be the set of all states in all the behavioral automata in Prot. $1\mathrm{E}_p$ of the process type p is defined as a tuple $(Q_{1\mathrm{E}_p}, q_{I1\mathrm{E}_p}, \Delta_{1\mathrm{E}_p})$, where $Q_{1\mathrm{E}_p} \subseteq Q$, $q_{I1\mathrm{E}_p} = q_{I_p}$, and $\Delta_{1\mathrm{E}_p} \subseteq Q_{1\mathrm{E}_p} \times E \times Q_{1\mathrm{E}_p}$ is the transition relation. We define the transition relation as follows:*

—*A single-cast transition $q_{1\mathrm{E}_p} \overset{e}{\rightarrow} q'_{1\mathrm{E}_p} \in \Delta_{1\mathrm{E}_p}$ if $\exists Snd_i = (q, q', \Delta_{I_i}, \Delta_i, \Delta_{F_i}, \{e\}) \in \mathcal{A}_p : q \in q_{1\mathrm{E}_p} \ \wedge$*

$$q'_{1\mathrm{E}_p} = \left\{ \begin{array}{l} q' \Leftarrow \exists Snd = (q', q_F, \Delta_I, \Delta, \Delta_F, E) \in \mathcal{A}_p \\ \bigcup \\ \{q_y\} \Leftarrow \exists p' \in [1,t], \exists Rcv_m, Snd_n \in \mathcal{A}_{p'} \in \mathsf{Prot}_{p'} : \\ \qquad\qquad\quad Rcv_m = (q_{I_m}, q_y, \Delta_{I_m}, \Delta_m, \Delta_{F_m}, \{e\}) \wedge \\ \qquad\qquad\quad Snd_n = (q_y, q_{F_n}, \Delta_{I_n}, \Delta_n, \Delta_{F_n}, E) \\ \bigcup \\ Q_r = \{q_r \mid q_r \in q_{1\mathrm{E}_p} \ \wedge \ q_r \neq q\} \end{array} \right\}$$

—*A broadcast transition $q_{1\mathrm{E}_p} \overset{e^+}{\rightarrow} q'_{1\mathrm{E}_p} \in \Delta_{1\mathrm{E}_p}$ if $\exists Snd_i = (q, q', \Delta_{I_i}, \Delta_i, \Delta_{F_i}, \{e^+\}) \in \mathcal{A}_p : q \in q_{1\mathrm{E}_p} \ \wedge$*

$$q'_{1\mathrm{E}_p} = \left\{ \begin{array}{l} q' \Leftarrow \exists Snd = (q', q_F, \Delta_I, \Delta, \Delta_F, E) \in \mathcal{A}_p \\ \bigcup \\ Q_y = \left\{ \begin{array}{l} q_y \mid \forall p' \in [1,t], \mathcal{A}_{p'} \in \mathsf{Prot}_{p'}, \\ \qquad \forall Rcv_m = (q_{I_m}, q_y, \Delta_{I_m}, \Delta_m, \Delta_{F_m}, \{e\}) \in \mathcal{A}_{p'}, \\ \qquad\qquad \exists Snd_n = (q_y, q_{F_n}, \Delta_{I_n}, \Delta_n, \Delta_{F_n}, E) \in \mathcal{A}_{p'} \end{array} \right\} \\ \bigcup \\ Q_r = \{q_r \mid q_r \in q_{1\mathrm{E}_p} \ \wedge \ q_r \neq q \ \wedge \\ \qquad\quad \forall p' \in [1,t], \nexists Rcv = (q_r, q_F, \Delta_I, \Delta, \Delta_F, \{e\}) \in \mathcal{A}_{p'} \in \mathsf{Prot}_{p'}\} \end{array} \right\}$$

Figure 3 presents $1\mathrm{E}_l$, the maximal behavior a "Left" philosopher can induce in any environment. The first state on the left belongs to the initial state $q_I \in \mathsf{Prot}_l$ as described in Section 2.1. This state belongs to the sender automaton on Line 3 in Figure 1(a), therefore the process can acquire the left fork by sending event `l-askl`. This event is the same as the input event for the automaton of the forks process specification on Line 2 in Figure 1(c), therefore the sender automaton is chained with this receiver automaton. The resulting state reflect which processes are allowed to send after the transition `l-askl`. According to the sender automaton of "Left" philosopher, the philosopher changes her state to `l-waitl`. Since there is no automaton in the process specification of "Left" philosophers that has a sender automaton with an initial state as `l-waitl`, this means that the philosopher is blocked as this state; she can only receive events at this state, but not send. On the other hand, when the fork receive the event `l-askl`, it changes its state to `l-lasked` according to the automaton in Line 2 in Figure 1(c). As there exists an automaton in the fork specification that is a sender automaton and has this states as its initial state (Line 8 in Figure 1(c)), this means that after this transition, the fork is able to send an event. Therefore, the resulting state in $1\mathrm{E}_l$ in Figure 3 after the first chaining of automata is `l-lasked`. This process is repeated till every automaton in the protocol specification is chained together.

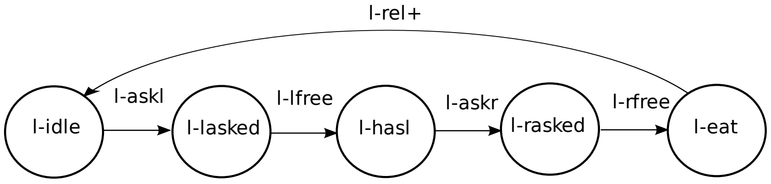


Fig. 3. $1\mathrm{E}_l$: The maximal behavior a "Left" philosopher can induce in *any environment*

For type "Right" philosophers, $1\mathrm{E}_r$ is constructed in the same manner. As processes of type "Fork" cannot initiate any behavior, maximal behavior induced by it ($1\mathrm{E}_f$) is empty.

In this example, a process is blocked after it sends an event (i.e. it reaches a state that no sender automaton has as its initial state). However, if the process that sends the event goes to a state allowing it to send events, then the state resulting from chaining in $1\mathrm{E}_p$ will have this process state. If the receiver of this event also goes to a state that allows to send, then state resulting from the chaining in $1\mathrm{E}_p$ will have both the states of the sender (that can send again) and the receiver (that can send after receiving event e). We show an example of such behavior in Section 6.5.

We now prove that every sequence in $\textsc{Max}_p(sys(\bar{k}_t))$ is present as a path in $1\mathrm{E}_p$. We define the set of sequences of input/output events in $1\mathrm{E}_p = (Q_{1\mathrm{E}_p}, q_{I1\mathrm{E}_p}, \Delta_{1\mathrm{E}_p})$ as

$$\textsc{Path}(1\mathrm{E}_p) = \{\pi \mid q^0_{I1\mathrm{E}_p} = q_{I1\mathrm{E}_p} \wedge \forall i \geq 0 : \ q^i_{1\mathrm{E}_p} \overset{\pi[i]}{\to} q^{i+1}_{1\mathrm{E}_p} \in \Delta_{1\mathrm{E}_p}, \text{where } q^i_{1\mathrm{E}_p} \in Q_{1\mathrm{E}_p}\}$$

For ease of explanation and brevity of the proof, we introduce the following functions.

$$F_1(\pi, \Pi) = \ \pi' \mid \pi' \in \Pi \wedge \pi' \sqsubset \pi \wedge \ \nexists \pi'' \in \Pi : (\pi' \sqsubset \pi'' \sqsubset \pi) \vee (\pi'' = \pi) \qquad (1)$$

where $\sqsubset$ denotes the strict substring relationship, i.e., $\pi' \sqsubset \pi$ implies π' is a substring of π and $\pi' \neq \pi$. The above function computes a set of substrings π' of π such that there are no other substrings of π in Π that are longer than the elements in the resulting set. We define the following function over sequences of events.

$$F_2(\pi, \pi') = e \text{ such that } \pi' \sqsubset \pi \ \wedge \ \pi[|\pi'|] = e \qquad (2)$$

The above function identifies the event on which the sequence π diverges from π'.

THEOREM 1. *Given a protocol specification* Prot *with t different types of processes,* $\forall \bar{k}_t, \forall p \in [1, t] : \textsc{Max}_p(sys(\bar{k}_t)) \subseteq \textsc{Path}(1\mathrm{E}_p)$.

PROOF. The proof is by contradiction. Assume that $\exists \bar{k}_t, \exists p \in [1, t] : \textsc{Max}_p(sys(\bar{k}_t)) \nsubseteq \textsc{Path}(1\mathrm{E}_p)$. In other words, there exists a path π such that $\pi \in \textsc{Max}_p(sys(\bar{k}_t))$ and $\pi \notin \textsc{Path}(1\mathrm{E}_p)$. Using Equations 1 and 2, $\forall \pi' \in F_1(\pi, \textsc{Path}(1\mathrm{E}_p)), \exists e : F_2(\pi, \pi') = e$.

There are two possible cases.

Case 1. e is the output event from the sender automata $Snd_i \in \mathcal{A}_p$ whose initial state is the same as the initial process state q_{I_p} in the process specification Prot_p. According to Definition 3.1, the first transition in any path π in $\textsc{Max}_p(sys(\bar{k}_t))$ is caused by a send action of an automaton $Snd \in \mathcal{A}_p$ whose initial state is the same as the initial process state $q_{I_p} \in \mathsf{Prot}_p$. Similarly, the first transition in $1\mathrm{E}_p$ is caused by a process of type p

sending e from its initial state (Definition 3.2). Therefore, this case is not possible, i.e., our assumption that $\pi \notin \text{PATH}(1\text{E}_p)$ must be false.

Case 2. e is the output event caused by the send action of an automaton other than Snd_i mentioned in the first case. If $\pi[l] = e$ for some $l > 0$, let $\pi[l-1] = e_0$. From Equation 2, $F_2(\pi, \pi') = e$ and therefore, $\pi'[l-1] = e_0$ and $\pi'[l] \neq e$. In order for this to be possible, there must be a sender behavioral automaton that produces event e_0, a receiver behavioral automaton that takes event e_0 as input and changes the state of the receiver process to q' (so that $\pi'[l-1] = e_0$), and one of the following must hold so that chaining in 1E_p is not possible (i.e. so that $\pi'[l] \neq e$):

(1) there is no sender behavioral automaton in Prot with state q' as its initial state that produces event e as output event.

(2) there is a sender behavioral automaton in Prot that has a state q' as its initial state that produces event e, but there is no receiver behavioral automaton in Prot that takes event e as input event.

If any one of the above cases holds, then it is not possible to have any sequence π in $\text{MAX}_p(sys(\bar{k}_t))$ that has $\pi[l-1] = e_0$ and $\pi[l] = e$. This contradicts our assumption that $\pi \in \text{MAX}_p(sys(\bar{k}_t))$. □

3.3 Finding the Cut-Off Value

The cut-off of parameter values for a parameterized system is such that the instance of the parameterized system at the cut-off (cut-off instance) satisfies a property if and only if all instances of the parameterized system larger than the cut-off instance satisfies the same property. We will consider two types of properties in the logic of $\text{LTL}\setminus\text{X}$ [Emerson 1990]:

—TYPE I PROPERTY: Property involving the actions of one process. For example, if a "Left" philosopher tries to pick the left fork, she can eventually eat.

—TYPE II PROPERTY: Property involving the actions of two processes that directly communicate via events. For example, if a "Left" philosopher tries to pick the left fork and the left fork is free then the left fork is no longer free.

We will use the standard notation $[\![\varphi]\!]$ to denote the semantics of an $\text{LTL}\setminus\text{X}$ property φ; it represents the set of sequence of states that satisfy φ. A system $sys(\bar{k}_t)$ satisfies φ, denoted by $sys(\bar{k}_t) \models \varphi$, if and only if all paths starting from all start states of the system result in a set of sequence of states such that this set is a subset of $[\![\varphi]\!]$. For details of semantics of LTL, we refer the reader to [Emerson 1990].

DEFINITION 3.3 CUT-OFF. *Given a system specification* Prot *for t different types of processes and a topology* Topo*, for any $\text{LTL}\setminus\text{X}$ properties of Type I and Type II, denoted by φ, $\bar{k}_t := k_1, k_2, \ldots, k_t$ is said to be cut-off if and only if the following holds: $sys(\bar{k}_t) \models \varphi \Leftrightarrow \forall \bar{n}_t \geq \bar{k}_t : sys(\bar{n}_t) \models \varphi$ where $\bar{n}_t \geq \bar{k}_t \Leftarrow \forall p \in [1, t] : n_p \geq k_p$.*

Procedure CutOff presents our automatic method for obtaining the cut-off. To generate the cut-off value for RLDP protocol, the initial system instance that procedure CutOff will use is $sys(1_l, 1_r, 2_f)$ displayed Figure 2(b) as there can be no smaller instance for this protocol. As described in Section 3, both "Left" and "Right" philosophers can start acting without external stimuli, therefore 1E_l (Figure 3) and 1E_p are constructed. No 1E_f is constructed for processes of type "Forks" as they rely on external stimuli to start acting.

```
Procedure CutOff (Prot, $t$, Topo)
  Construct initial $sys(\bar{k}_t)$ using Prot and Topo
  for all $p \in [1, t]$ Compute $1\mathrm{E}_p$ from Prot do
    while $\exists \pi \in \textsc{Path}(1\mathrm{E}_p) : \pi \notin \textsc{Max}_p(sys(\bar{k}_t))$ do
      Increase $\bar{k}_t$ in a breadth-first manner
    end while
  end for
  return $\bar{k}_t$;
```

If we compare the system instance $sys(1_l, 1_r, 2_f)$ and $1\mathrm{E}_l$ in Figure 2(b) and Figure 3, respectively, we can see that for every path in $1\mathrm{E}_l$ (in this case there is only one path) the "Left" philosopher in the system instance can induce the same path of events (in Figure 2(b), the lower sequence of events starting from the start state). Therefore, $\forall \pi \in \textsc{Path}(1\mathrm{E}_l), \pi \in \textsc{Max}_l(sys(1_l, 1_r, 2_f))$. Similarly, $\forall \pi \in \textsc{Path}(1\mathrm{E}_r), \pi \in \textsc{Max}_r(sys(1_l, 1_r, 2_f))$. Therefore, the return of the procedure is $(1_l, 1_r, 2_f)$; the size of the system instance $sys(1_l, 1_r, 2_f)$ is the cut-off value for the RLDP protocol.

To ensure termination, a variation of the Procedure CutOff can be used that takes an additional argument a bound on the parameters. In that case, the while loop in the procedure will increase $\bar{k}_t$ only up to the supplied bound on the parameters.

4. PROOF OF SOUNDNESS

We now introduce definitions and propositions that will be used to prove the soundness of Procedure CutOff.

Definition 4.1 Projection on processes. *Given a multi-parameterized system $sys(\bar{n}_t) = (S, S_I, T,$ Topo$)$ with t different types of processes and a set $R \subseteq \{i_p \mid i \in [1, n_p] \ \wedge\ p \in [1, t]\}$, the projected behavior w.r.t. R is denoted by $sys(\bar{n}_t){\downarrow}R = (S, S_I, T{\downarrow}R,$ Topo$)$. For any system state $s \in S$, let $s = \langle Q_R, Q\rangle$, Q_R and Q represents states of processes in R and not in R, respectively, at state s. The transition relation $T{\downarrow}R$ is defined as follows:*

—*Transition $s \xrightarrow{e} s' \in T{\downarrow}R$, where $s = \langle Q_R, Q\rangle$ and $s' = \langle Q'_R, Q'\rangle$ if*

$$\langle Q_R, Q\rangle \xrightarrow{e} \langle Q'_R, Q'\rangle \in T \ \wedge\ \exists i_p \in R : q_{i_p} \in Q_R \ \wedge\ q'_{i_p} \in Q'_R \ \wedge\ q_{i_p} \neq q'_{i_p}$$

—*Transition $s \xrightarrow{\tau} s' \in T{\downarrow}R$, where $s = \langle Q_R, Q\rangle$ and $s' = \langle Q'_R, Q'\rangle$ if*

$$\langle Q_R, Q\rangle \xrightarrow{e} \langle Q'_R, Q'\rangle \in T \ \wedge\ \forall i_p \in R : q_{i_p} \in Q_R \ \wedge\ q'_{i_p} \in Q'_R \ \wedge\ q_{i_p} = q'_{i_p}$$

We will use $\pi{\downarrow}R$ to denote projection of a sequence of events on R.

Proposition 1. *For any multi-parameterized system $sys(\bar{k}_t)$ with t different types of processes, the following holds for all properties φ (in the logic of $\mathtt{LTL}\setminus\mathtt{X}$) defined over states of processes whose indices belong to $R = \{i_p \mid i \in [1, k_p] \wedge p \in [1, t]\}$: $sys(\bar{k}_t) \models \varphi \ \Leftrightarrow\ sys(\bar{k}_t){\downarrow}R \models \varphi$.*

In the following, we define the set of sequences of input/output events in the system-instance $sys(\bar{k}_t) = (S, S_I, T,$ Topo$)$ as $\textsc{Path}(sys(\bar{k}_t), S) = \{\pi \mid \eta_0 = s \in S \ \wedge\ \forall i \geq 0 : \eta_i \xrightarrow{\pi[i]} \eta_{i+1} \in T\}$.

PROPOSITION 2. *Let Φ be the set of all properties (in the logic of $\mathtt{LTL}\backslash X$) defined over states of processes whose indices belong to $R = \{i_p \mid i \in [1, k_p] \wedge p \in [1, t]\}$. The following holds for any two instances of multi-parameterized systems, $sys(\bar{k}_t)$ and $sys(\bar{k}'_t)$.*

$$\forall \varphi \in \Phi : \big(sys(\bar{k}_t) \models \varphi \;\Leftrightarrow\; sys(\bar{k}'_t) \models \varphi\big) \Rightarrow \\ \textsc{Path}(sys(\bar{k}_t){\downarrow}R, S_I^{\bar{k}_t}) = \textsc{Path}(sys(\bar{k}'_t){\downarrow}R, S_I^{\bar{k}'_t})$$

In the above, $S_I^{\bar{k}_t}$ and $S_I^{\bar{k}'_t}$ are the initial state-sets of $sys(\bar{k}_t)$ and $sys(\bar{k}'_t)$, respectively.

PROOF. From Proposition 1, we conclude

$$\forall \varphi \in \Phi : \big(sys(\bar{k}_t) \models \varphi \;\Leftrightarrow\; sys(\bar{k}'_t) \models \varphi\big) \Rightarrow \big(sys(\bar{k}_t){\downarrow}R \models \varphi \;\Leftrightarrow\; sys(\bar{k}'_t){\downarrow}R \models \varphi\big)$$

If π denotes a path in a system over sequence of input/output actions, we denote the corresponding sequence of states in the path by $\mathtt{seq}(\pi)$. Therefore,

$$\forall \varphi \in \Phi : \big(sys(\bar{k}_t){\downarrow}R \models \varphi \;\Leftrightarrow\; sys(\bar{k}'_t){\downarrow}R \models \varphi\big) \;\Rightarrow \\ \forall \pi \in \textsc{Path}(sys(\bar{k}_t){\downarrow}R, S_I^{\bar{k}_t}) : \exists \pi' \in \textsc{Path}(sys(\bar{k}'_t){\downarrow}R, S_I^{\bar{k}'_t}) : \mathtt{seq}(\pi) = \mathtt{seq}(\pi') \\ \bigwedge \\ \forall \pi' \in \textsc{Path}(sys(\bar{k}'_t){\downarrow}R, S_I^{\bar{k}'_t}) : \exists \pi \in \textsc{Path}(sys(\bar{k}_t){\downarrow}R, S_I^{\bar{k}_t}) : \mathtt{seq}(\pi') = \mathtt{seq}(\pi) \\ \Rightarrow\; \textsc{Path}(sys(\bar{k}_t){\downarrow}R, S_I^{\bar{k}_t}) = \textsc{Path}(sys(\bar{k}'_t){\downarrow}R, S_I^{\bar{k}'_t})$$

□

PROPOSITION 3. *For any parameterized system with t types of processes,*

$$\forall \bar{n}_t \geq \bar{k}_t : \textsc{Path}(sys(\bar{k}_t), S_I^{\bar{k}_t}) \subseteq \textsc{Path}(sys(\bar{n}_t), S_I^{\bar{n}_t}) \\ \forall p \in [1, t], \pi \in \textsc{Path}(1\mathrm{E}_p) : \pi \in \textsc{Max}_p(sys(\bar{k}_t)) \Rightarrow\; \pi \in \textsc{Max}_p(sys(\bar{n}_t))$$

where $S_I^{\bar{k}_t}$ and $S_I^{\bar{n}_t}$ are the sets of initial states of $sys(\bar{k}_t)$ and $sys(\bar{n}_t)$, respectively.

THEOREM 2. *Given a parameterized system with t different types of processes each defined using a set of behavioral automata **Prot**, the following holds for all Type I and II properties φ in the logic of $\mathtt{LTL}\backslash X$*

$$\forall p \in [1, t], \pi \in \textsc{Path}(1\mathrm{E}_p) : \pi \in \textsc{Max}_p(sys(\bar{k}_t)) \Rightarrow\; \big(sys(\bar{k}_t) \models \varphi \Leftrightarrow sys(\bar{n}_t) \models \varphi\big)$$

where $\bar{n}_t = n_1, n_2, \ldots, n_t$ and $\bar{k}_t = k_1, k_2, \ldots, k_t$.

PROOF. We first prove the theorem for Type I properties. Recall from Section 3.3 that Type I property specification is concerned with actions of one process. Therefore, using Propositions 1, 2 and 3, it is required to prove that $\forall p \in [1, t], \forall i \leq n_p, \exists j \leq k_p$,

$$\textsc{Path}(sys(\bar{n}_t){\downarrow}\{i_p\}, S_I^{\bar{n}_t}) = \textsc{Path}(sys(\bar{k}_t){\downarrow}\{j_p\}, S_I^{\bar{k}_t})$$

such that and $S_I^{\bar{n}_t}$ and $S_I^{\bar{k}_t}$ are initial state-sets of $sys(\bar{n}_t)$ and $sys(\bar{k}_t)$, respectively.

Assume that there exists a sequence π of events in $\textsc{Path}(sys(\bar{n}_t), S_I^{\bar{n}_t})$ such that $\pi{\downarrow}\{i_p\}$ is not present in $\textsc{Path}(sys(\bar{k}_t) \downarrow \{j_p\}, S_I^{\bar{k}_t})$ for any $j_p \leq k_p$. I.e., $\exists i_p \leq n_p, \forall j_p \leq k_p$: $\textsc{Path}(sys(\bar{n}_t) \downarrow \{i_p\}, S_I^{\bar{n}_t}) \neq \textsc{Path}(sys(\bar{k}_t) \downarrow \{j_p\}, S_I^{\bar{k}_t})$. This assumption implies that

(using Equations 1 and 2)

$$F_1(\pi{\downarrow}\{i_p\},\, \bigcup_{j_p} \textsc{Path}(sys(\bar{k}_t){\downarrow}\{j_p\}, S_I^{\bar{k}_t})) = \chi_1 \neq \emptyset \\ \Rightarrow \forall \pi'_1 \in \chi_1 : \exists e_0 : F_2(\pi{\downarrow}\{i_p\}, \pi'_1) = e_0 \tag{3}$$

This, in turn, implies two possibilities as explained below:

Case 1. e_0 is the output event from the sender automata $Snd_l \in \mathcal{A}_p$ whose initial state is the same as the initial process state q_{I_p} in the process specification Prot_{p_1}(i.e. event sent without any external stimuli). As such a move is absent in all paths in $sys(\bar{k}_t)$, and as this is the first event a process of type p can send, we can conclude that $\exists \pi' \in \textsc{Path}(1\textsc{E}_p) : \pi' \notin \textsc{Max}_p(sys(\bar{k}_t))$.

Case 2. e_0 is an event that process i_p sent as a result of an external stimuli. Let i'-th process of type p' in $sys(\bar{n}_t)$ be the process that received e_0, while in $sys(\bar{k}_t)$, there is no process of type p' that can receive such event. Now let's assume that e_1 is the event that i_p received from some process i_1 of type p_1 in $sys(\bar{n}_t)$ (and similarly that j_p received from some process j_1 of type p_1 in $sys(\bar{k}_t)$) that caused process i_p to send event e_0 in $sys(\bar{n}_t)$ while process j_p was not able to send e_0 in $sys(\bar{k}_t)$. If e_1 is the first event the process i_1 of type p_1 can send without any external stimuli, then this means that the sequence of event $\pi = e_1, e_0 \in \textsc{Path}(1\textsc{E}_{p1})$ while $\pi \notin \textsc{Max}_{p_1}(sys(\bar{k}_t))$ (Case 1). If e_1 is not the first event process i_1 of type p_1 can send without any external stimuli, then we will repeat the above argument for the event that i_1 of type p_1 received in $sys(\bar{n}_t)$ (and similarly that process j_1 of type p_1 in $sys(\bar{k}_t$ received) that lead both processes to send event e_1. The above argument is repeated until only Case 1 is applicable and it follows that $\exists p \in [1, t], \pi' \in \textsc{Path}(1\textsc{E}_p) : \pi' \notin \textsc{Max}_p(sys(\bar{k}_t))$. This concludes the proof for Type I properties.

TYPE II PROPERTY. The proof for type II properties follows similar arguments as provided above.

□

THEOREM 3 SOUNDNESS. *If Procedure* **CutOff** *terminates, the return $\bar{k}_t$ is the cut-off as per the Definition 3.3.*

PROOF. Follows from Theorem 2 □

5. GOLOK - TOOL FOR AUTOMATIC CUT-OFF GENERATION

We have implemented our technique in a tool, *Golok* [Samuelson et al. 2010]. It is written in Scheme [Abelson et al. 1998] in ∼3.5K lines of code. An overview of Golok's architecture is presented in Figure 4. Golok serves as a proof that our technique can be fully automated. As we discuss later, we have applied Golok to several parameterized systems.

As input, Golok takes the system and the topology specification. The maximal behavior ($1\textsc{E}_p$) is generated for every process type p using this input file, then the generated model is given to the System Instance Generator that find the smallest system instance that simulates all the maximal behaviors (if such a system instance exists). If successful, Golok outputs the cut-off number along with a graphical representation of the simulating path of the system model and the maximal behavior for every process in the form of dot graphs [Gansner and North 1999] for visual inspection.

We now describe the input language to Golok using the RLDP example and describe several optimizations that we have implemented in our tool.

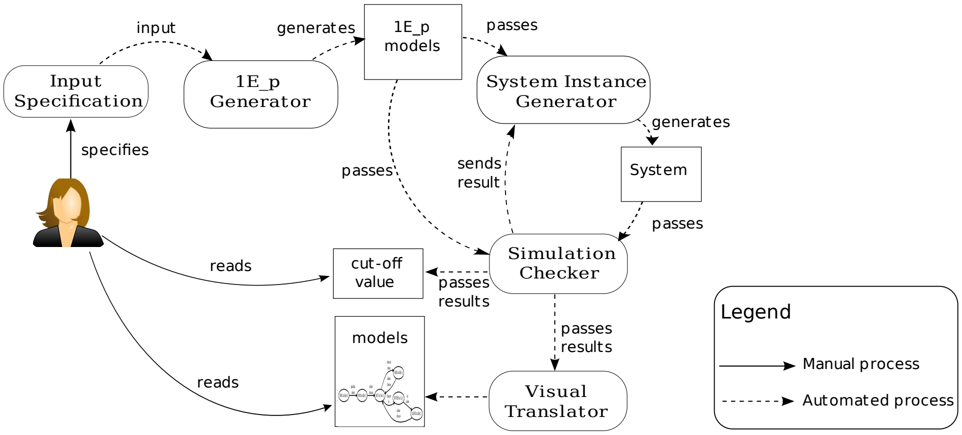


Fig. 4. Overview of Golok and its components

5.1 Front End: Input Language of Golok

The input file containing the specification for the RLDP protocol is displayed in Figure 5. It has two main parts: the process specification and the topology specification.

Process Specification. The process specifications (Lines 1-6) contain the behavioral automata for every process type (Lines 1-3) as described in Section 2.1 (Figures 1(a), (b) and (c)). In addition, it contains the initial state specification (Line 5) that is responsible for showing the initial states of every process type as described in Section 2.1.

For some protocols, we might need one process of some type to be in a different state. For example, for the DME protocol only one process starts in a different state where it has token (further described in Section 6.4). The special state specification (Line 6) specifies if any process needs to start from a different state other than its initial state.

```
1  process left-diner { ... }
2  process right-diner { ... }
3  process fork { ... }

5  initialstates {l-idle, r-idle, free}
6  specialstates {  }

8  topology {

10  connectivity {
11   left-diner  0 -- fork        0
12   fork        0 -- right-diner 0
13   right-diner 0 -- fork        1
14   ...
15  }

17  additionrule add-two {
18   create: left-diner a
19   create: right-diner b
20   create: fork c
21   create: fork d
22   require: right-diner e -- fork 0
23   remove: var e -- fork 0
24   add: var e -- var c
25   add: var c -- var a
26   add: var a -- var d
27   add: var d -- var b
28   add: var b -- fork 0
29  }

31  msgs {
32   (left-diner, l-askl, lpeer)
33   (fork, l-lfree, rpeer)
34   ...
35  }
36 }
```

Fig. 5. Input file for the RLDP protocol

Topology Specification. As discussed previously in Section 2 the topology specification serves to restrict communication patterns between processes. It is defined using the keyword **`topology`** (lines 8 - 36) and is composed of three parts. The first part of the topology specification (lines 7-12) specifies the topology of the initial system instance. In RLDP, the initial system instance has one philosopher of each type (processes are zero-indexed).

The second part of the topology specification (lines 17-29) are the addition rules that are used to generate larger system instances from the initial (starting) system instance. These rules ensure that newly generated system instances follow the communication topology of the protocol. For RLDP, the addition rule `add-two` (lines 17-29) states that any new system instances will create two new philosophers of different types (lines 14-15), two forks (lines 20-21) and that they are linked to other processes to preserve the system's constraint that neighbor philosophers are of different types (lines 22-28).

The final part of the topology specification (lines 31-36) is responsible for specifying the direction of the flow of events between processes, where every tuple (d, e, s) describes the event to be received e, the type of the recipient process d and the index of the sender process s. There are four choices for s: `rpeer` (a message sent by the right neighbor of d), `lpeer` (a message sent by the left neighbor of d), and `peer` (a message sent by some neighbor of d). The choices `lpeer`, `rpeer` are only used for describing ring topologies and the choice `peer` is used for describing other topologies.

Note that, unlike many of the existing work on parameterized systems, our technique is not directly dependent on the property to be verified; therefore, Golok does not require properties as input.

5.2 System Instance Generator/Checker

The System Instance Generator/Checker (SIGC) is the main module of Golok. The goal of SIGC is to construct a system instance $sys(\bar{k}_t)$ (see Procedure CutOff) and check whether for all $p \in [1, t], \pi \in \textsc{Path}(1\mathrm{E}_p), \pi \in \textsc{Max}_p(sys(\bar{k}_t))$, i.e. all paths in the maximal behavior induced by a process of type p in any environment exist in the system instance. The main challenge in implementing SIGC is to reduce the computational cost involved in checking for path inclusion by considering all possible paths from all states. We describe several optimization techniques we have implemented in Golok to help reduce this cost.

Simulation-based cut-off computation. While Procedure CutOff is based on language inclusion ($\forall \pi \in \textsc{Path}(1\mathrm{E}_p), \pi \in \textsc{Max}_p(sys(\bar{k}_t))$), Golok cut-off computation is based on simulation relation instead.

Simulation relation [Milner 1982] identifies pairs of states in a transition system such that one element of the pair simulates all possible behavior (in terms of sequence and branching of transitions) of the other. It is a stronger relation than language inclusion. Furthermore, computing simulation relation is linear to the state-space of the transition system as opposed to computing language inclusion which is exponential to the state-space [Vardi 2001]. As a result, it is computationally efficient to use simulation rather than language inclusion. Given a protocol specification Prot and a multi-parameterized system $sys(\bar{k}_t)$ with t different types of processes, a state r in $sys(\bar{k}_t)$ is said to simulate a state s in $1\mathrm{E}_p$, denoted as $s \prec r$, if the following holds:

$$\forall s', e : s \xrightarrow{e} s' \in 1\mathrm{E}_p \Rightarrow \exists r' : r \rightsquigarrow^{e} r' \in sys(\bar{k}_t) \wedge s' \prec r'$$

where $r \rightsquigarrow^{e} r'$ denotes a sequence of transitions containing zero or more transitions over actions not induced by a process of type p and a single action e induced by a process of type p. We say that $1\mathrm{E}_p$ is simulated by $sys(\bar{k}_t)$ if and only if there exists a state r in $sys(\bar{k}_t)$ such that for all start states s in $1\mathrm{E}_p$, $s \prec r$. As simulation relation is stronger than language inclusion, $s \prec r$ where s and r are states in $1\mathrm{E}_p$ and $sys(\bar{k}_t)$, respectively, implies that all paths in $\textsc{Path}(1\mathrm{E}_p)$ belong to the set of paths in $\textsc{Max}_p(sys(\bar{k}_t))$. Consequently, the results of Theorem 3 still holds.

In the example of RLDP, there exists no difference between the language inclusion and the simulation relation. To show the difference, assume that $1\mathrm{E}_l$ in Figure 3 had two different paths of events π_1 and π_2 that start from the start state as opposed to one. To satisfy the language inclusion condition $\forall \pi \in \textsc{Path}(1\mathrm{E}_p), \pi \in \textsc{Max}_p(sys(\bar{k}_t))$, there must exist one state s_1 in $sys(\bar{k}_t)$ from which path π_1 in $1\mathrm{E}_p$ starts, and another state s_2 (which could be the same as s_1) from which path π_2 starts. On the other hand, for a state s in $sys(\bar{k}_t)$ to simulate $1\mathrm{E}_l$ ($1\mathrm{E}_l \prec sys(\bar{k}_t)$), both paths π_1 and π_2 *must* start from the same state s; therefore the simulation relation is stronger than language inclusion.

Reducing the number of Simulation Checks. As the size of a system instance could be prohibitively large, performing a simulation check on every state to verify if it simulates $1\mathrm{E}_p$ can still be expensive. To reduce the number of simulation checks, we construct the system instances on-the-fly (i.e. states are generated when needed). Furthermore, for every system state s in $sys(\bar{k}_t)$, the following constant-time check is done before performing a simulation check between the system and $1\mathrm{E}_p$. If the system state s does not have any process of type p that is able to initiate the protocol, this system state s is never expanded. The reason is that, since the first transition in any $1\mathrm{E}_p$ must come from a process of type p initiating the protocol, then it is not possible that a system state s where no process of type p can initiate the protocol is the state that simulates $1\mathrm{E}_p$.

6. CASE STUDIES

We have applied Golok to several nontrivial systems to validate our approach: the variant of the dining philosopher protocol that we discussed as the running example, the standard Dining Philosophers protocol, the Bounded Buffer protocol [Silberschatz et al. 2004], the Spin lock [Anderson 1990], a locking protocol for mutual exclusive access to an object, the Distributed Mutual Exclusion protocol (DME) [Wolper and Lovinfosse 1990], all the snoop cache-coherence protocols presented in [Handy 1993], and the German's cache-coherence protocols [German 2000].

We compared our results with that obtained using existing work. This comparative analysis is presented in Table I. In 14 out of 15 cases, Golok has identified a smaller cut-off value compared to the ones known in the existing work (shown in bold font in Table I). For the remaining case Golok identified the same cut-off value. This can be attributed primarily to the fact that existing techniques for cut-off identification are independent of the system behavior (only topology dependent, e.g., [Emerson and Namjoshi 1995]) or rely on abstractions that are sufficient but not necessary (e.g., [Basu and Ramakrishnan 2006]).

In contrast to existing techniques, our technique uniformly handles systems of different classes (e.g. resource allocation, cache coherence protocols), systems with more than one types of unbounded processes, with different topologies, and systems where processes are communicating via single-cast and/or broadcast. We now discuss these case studies.

Table I. Experimental results of our tool *Golok* compared to existing techniques

		Protocol Spec			EXISTING WORK		GOLOK
Protocol	**Topology**	**# of types**	**# of Params**	**Broadcast Comm.**	**References**	**Known Cut-off**	**Computed Cut-off ($\bar{k}_t$)**
Dining Philosophers	Ring	1	1		[Emerson and Kahlon 2002]	4	**2**
					[Hanna et al. 2009]	3	
RLDP	Ring	3	(r, l, f) 3	$\surd^{\dagger}$	[Emerson and Kahlon 2002]	$2_r, 2_l, 4_f$	$\mathbf{1}_r, \mathbf{1}_l, \mathbf{2}_f$
					[Hanna et al. 2010]	$3_r, 3_l, 6_f$	
Bounded-Buffer	Star	3	(p, co) 2		[Hanna et al. 2009]	$2_p, 1_{co}$	$\mathbf{1}_p, \mathbf{1}_{co}$‡
Spin Lock	Star	2	(t) 1		[Basu and Ramakrishnan 2006]	3	**2**
Spin Lock (multi)	Multi-star*	2	(t, o) 2		[Hanna et al. 2010]	$2_t, 2_o$	$\mathbf{2}_t, \mathbf{1}_o$
DME	Ring	1	(fc) 1		[Emerson and Namjoshi 1995]	4	$\mathbf{2}_{fc}$
DME (multi)	Ring	2	(fr, c) 2		[Hanna et al. 2010]	$1_{fr}, 1_c$	$\mathbf{1}_{fr}, \mathbf{1}_c$
MSI	Full Mesh	1	1	$\surd$	[Emerson and Kahlon 2003]	7	**2**
MESI							
MOESI							
Berkeley							
Synapse N+1							
Firefly	Full Mesh	1	1	$\surd$	X$^{\diamond}$	X$^{\diamond}$	**2**
Dragon							
German's Cache	Star	2	(ca)1	$\surd^{\pm}$	[Emerson and Kahlon 2003]	7_{ca}	$\mathbf{2}_{ca}$

†: Communication is via both single-cast and broadcast.
‡: Golok produced same cut-off value for different sizes of the buffer, displayed performance results are for the system with buffer of size 1.
*Multi-star: All processes of different types are connected;
◇: To the best of our knowledge, no known results exist.
± : The original protocol is not broadcast-based, however it can be reduced to the ESI protocol which is broadcast-based as shown in [Emerson and Kahlon 2003].
r: right philosopher, l: left philosopher, f: fork; p: producer, co: consumer; t: thread, o: object; fc: dme node, fr: forward dme node, c: critical dme node; ca: cache.

6.1 Dining Philosophers Protocols

For the RLDP protocol that was used as the illustrative example throughout this article, example properties that can be verified are: a "Left" philosopher will eventually eat (Type I property) or two neighbor philosophers cannot eat at the same time (Type II property). As described in Section 3.3, properties are defined in terms of events. The former property can be defined as `GF(l-rfree)`, which reflects the behavior of the "Left" philosopher acquiring the right fork (thus already has the left fork), therefore the philosopher will be able to eat. For the latter property, it can be defined as `G(l-rfree -> not (r-lfree) U relr)`, which translates to the following: if the "Left" philosopher is eating, then the "Right" philosopher cannot eat until the "Left" philosopher releases her right fork. `G`, `F` and `U` are temporal operators in `LTL\X` denoting "globally", "eventually" and "until", respectively. We also generated the cut-off for the classic dining philosophers protocols [Dijkstra 1978], where only one type of philosophers exists, the "Left" philosophers.

6.2 Bounded-Buffer protocol

The Bounded-Buffer protocol [Silberschatz et al. 2004] is a typical producer-consumer system where a buffer can be accessed by both consumers (that consume the contents of the buffer) and producers (that provide the contents of the buffer). Figure 6 shows the behavioral automata for this protocol, where the size of the buffer is 1.

The main objective of this protocol is to ensure that at any point of time, only one type of processes (consumer or producer) can have access to the buffer, i.e., a producer cannot write to the buffer if it is full, while a consumer cannot read from the buffer if it is empty. This property is an example of type I property. It can be verified by defining in

```
process producer {
  [produce]       -> [produce, add-item]
}
process consumer {
  [consume]       -> [consume, take-item]
}
process buffer {
  [emptybuffer, add-item] -> [fullbuffer]
  [fullbuffer, take-item] -> [emptybuffer]
}
```

Fig. 6. Behavioral automata for the Bounded-Buffer Protocol

terms of the object as follows: `G(add-item -> (not(add-item) U take-item))`, and `G(take-item -> (not(take-item) U add-item))`.

6.3 Spin Lock

Spin locks [Anderson 1990] offer a mechanism to realize mutually exclusive access of objects by threads, when several threads try to access one object. It is considered as a singly-parameterized system with only the number of threads being parameterized.

```
process thread {
  [start]      -> [has-object, own]
  [has-object] -> [start, rel]
}
process object {
  [free, own]  -> [taken]
  [taken, rel] -> [free]
}
```

Fig. 7. Behavioral automata for Spin Lock protocol

The behavioral automata for this system are displayed in Figure 7. The object can have two states: free (when it is not accessed by any thread) and taken (when it accessed by some thread). A free object becomes taken upon a request from a thread (event `own`) so that this thread gets access to the object. A busy object, on the other hand, does not accept any requests from threads, but can receive a `rel` (release) event from the lock releasing thread and, as a consequence, the object returns back to free state. In addition to singly-parameterized spin lock, we have also applied Golok to a multi-parameterized variant of spin locks, where both the number of threads and objects are parameterized.

Properties of interest for Spin lock includes mutually exclusive access to objects by threads and liveness of threads. Both properties fall in the category of type I and type II properties. The former is defined on objects `G(own -> (not(own) U rel))`; while the latter is defined on threads `GF(own)`.

6.4 Distributed Mutual Exclusion Protocol

The objective of the Distributed Mutual Exclusion (DME) protocol [Wolper and Lovinfosse 1990] is to ensure that for a distributed system of n processes in a network with ring topology, only one process in the system is in the critical section at a given point of time. A token is passed between the different processes in the ring. A process can enter the critical section only if it holds the token. Once the process is out of the critical section, it passes the token to its neighbor.

The input file of the DME for Golok is displayed in Figure 8. The special state specification (Line 12) is ensuring that one thread will trigger the protocol by sending the token when it is in the `start` state (sender automaton in Line 3), while other threads will be initially in the `wait` state waiting for the token (receiver automaton in Line 2). The process type `shared-resource` models the shared resource accessed by the thread that posses

```
1  process thread {
2   [wait, token] -> [choose]
3   [start]       -> [wait, token]
4   [choose]      -> [wait, token]
5   [choose]      -> [cs, in]
6   [cs, done]    -> [start]
7  }
8  process shared-resource {
9   [idle, in] -> [busy]
10  [busy]     -> [idle, done]
11 }
12 initialstates { wait, idle }
13 specialstates { start }
14 topology {
15  connectivity {
16   thread 0 -- thread 1
17   thread 0 -- shared-resource 0
18   thread 1 -- shared-resource 0
19  }
20  additionrule { ... }
21  msgs {
22   (thread, token, lpeer)
23   (thread, in, peer)
24   (critical-section, done, peer)
25  }
26 }
```

Fig. 8. Input Specification for the Distributed Mutual Exclusion protocol

the token. There is only one process of this process type. The shared resource is initially in state `idle` as state in Line 12.

The topology section (Lines 14-26) shows the flow of the messages. All the threads are connected to the shared resource (Lines 17-18). Threads send the token to their left neighbor (Line 22), and send the event `in` to the shared-resource (Line 23). As there exists no receiver automata in the process specification of `thread` that takes `in` as their input event, the event `in` is always received by the shared-resource. Similarly, threads receive event `done` when sent by the shared-resource (Line 24).

In our variant of the DME protocol (not displayed in the figure), we consider two types of processes: the processes of the first type always skip entering the critical section and only forward the token to their neighbor, hereafter called "Forward" type of processes. The processes of the second type always enter the critical section upon receiving the token then forward it, henceforth called "Critical" type of processes.

While ensuring mutual exclusion is straightforward in the above protocol and its variant, a challenging problem is to verify liveness, i.e., a process (on the "Critical" process in the variant) can always eventually enter the critical section if it intends to do so. This property is a property of type I and can be written as `GF(in)`.

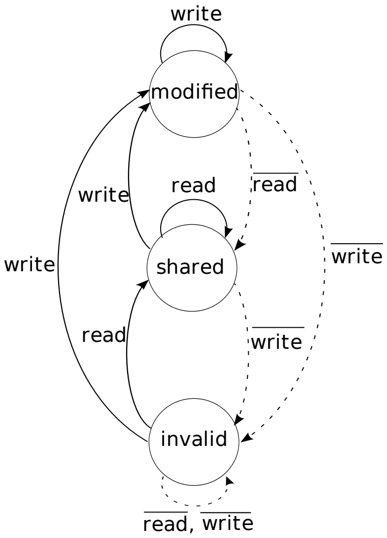
6.5 Cache Coherence Protocols

The cache coherence protocols [Handy 1993] are used in multi-processor systems with shared memory, where each processor possesses its own private cache and maintains its own copy of same memory block in its private cache. The main concern is to ensure that at any point of time, multiple cached copies of same memory block are consistent in their data content.

There are two types of cache coherence protocols: snoop-based [Handy 1993] and directory based [German 2000]. In snoop-based protocols, all caches communicate with each other through a bus and it is by snooping on that bus that caches know the status of the memory blocks of interest, while in directory-based protocols the caches do not communicate with each other but communicate directly with the directory that has the memory blocks.

We have directly applied Golok to several snoop-based [Handy 1993] protocols and used the reduction provided by [Emerson and Kahlon 2003] to the directory-based protocol proposed by [German 2000].

6.5.1 *Snoop-based Protocols.* There are two types of snoopy cache coherence protocols: invalidation based and update based. In protocols of the first type, whenever a shared memory block is written to by a processor, the block being written to is invalidated in all other caches [Handy 1993]. On the other hand, in update based protocols, on a write operations, the value of the shared memory block written to is updated in the caches of all other processors holding that block without invalidation. We used Golok to generate cut-offs for the invalidation based cache coherence protocols (MSI, MESI, MOESI, Berkeley and Synapse N+1) and update based protocols (Dragon and Firefly) [Handy 1993]. Table I shows the results where the cut-off generated by Golok is much smaller than the one generated by existing techniques. We now present the Modified-Shared-Invalid (MSI) protocol as an example to show how we modeled snoopy cache coherence protocols.


write
modified
write
read
$\overline{\text{read}}$
write
shared
$\overline{\text{write}}$
read
$\overline{\text{write}}$
invalid
$\overline{\text{read}}$, $\overline{\text{write}}$

(a)

```
process processor {
 [shared]                 ->[modified, write+]
 [shared, write]          ->[invalid]
 [shared]                 ->[shared, read+]
 [shared, read]           ->[shared]
 [modified]               ->[done, is-modified+]
 [modified, read]         ->[shared]
 [modified, write]        ->[invalid]
 [done, read]             ->[shared]
 [done, write]            ->[invalid]
 [invalid]                ->[shared, read+]
 [invalid]                ->[modified, write+]
 [invalid, write]         ->[invalid]
 [invalid, is-modified]->[invalid]
}

initialstates { shared }
specialstates { }
```

(b)

Fig. 9. (a) The MSI cache coherence protocol. (b) Process specification for MSI protocol

MSI protocol. Figure 9(a) shows the different states of a processor in MSI protocol. A dashed arrow models a process receiving an event. The protocol defines three distinct states for each processor: modified, shared and invalid. Modified processor state implies that the processor is the exclusive owner of the memory block (it is the only one that can modify on the contents of this block). Shared state implies that processor has current copy of memory block in its cache while invalid processor state implies that the processor's cached copy of memory block is outdated. Each processor can either perform autonomous read (in all states except invalid) or write (only in write state) actions. The goal is to ensure that only one process is in the modified state at a certain point of time.

Since we define properties in terms of events and not in terms of process state, we cannot define such goal with the events presented in Figure 9(a) as there is no event that distinguishes the state `modified`. We needed to add some event `is-modified` to distinguish the state `modified`, which can only be produced once when a process is in this state. To define the property that no two processes can be in the state `modified` at the same time, we would say `G(is-modified -> not(is-modified) U write)`; that is, if a process is

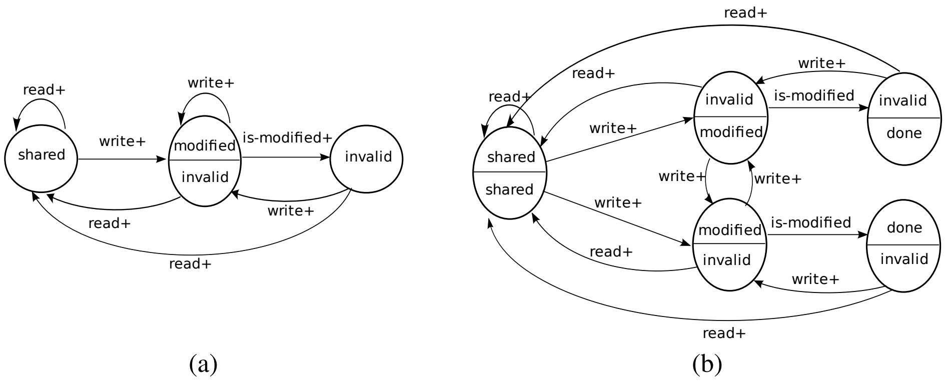


Fig. 10. (a)$1\mathrm{E}_p$: the maximal behavior a processor in MSI protocol can induce in any environment (b) System Instance for MSI protocol with two processes

in state `modified` then there cannot be another process in the same state until a `write` event is sent, where, as displayed in Figure 9(a), the event `write` invalidates the process that was already in state `modified`.

The process specification for this protocol is displayed in Figure 9(b). Line 6 shows the event `is-modified` that the process in state `modified` can send that distinguished the process in that state. Once in state `done`, the process is known to be the exclusive owner of the block, and gets out of this state if another process wants to read this memory block (Line 7) or write on it (Line 8).

Building the maximal behavior ($1\mathrm{E}_p$). As mentioned in Section 3.2, any state in $1\mathrm{E}_p$ reflects which processes are allowed to send events at a certain point of time. Figure 10(a) shows $1\mathrm{E}_p$ of a process in MSI cache coherence protocol. After a process in state `shared` broadcasts event `write`, the sender state changes to `modified` (according to the behavior defined in the sender automaton in Line 2 in Figure 9(b)). State `modified` is an initial state for the sender automaton in Line 6, therefore the process that broadcast event `write` can still send another event. The receiver automata in Lines 3, 10 and 13 show that the receiver(s) of event `write` go to state `invalid`, which is the initial state for sender automaton in Line 11. Therefore, the action of broadcasting and receiving event `write` allows both the sender and the receiver of the event to send events (no process is blocked after this transition). In $1\mathrm{E}_p$, the resulting state from such transition has the states of the sender and receiver(s) processes involved in the transition.

On the other hand, the transition `is-modified+` in Figure 10(a) modeling the chaining between sender and receiver automata in Lines 6 and 14, respectively, will result in only one state since only the receiver of `is-modified` can send after this action (the final state of the receiver automaton in Line 14 is state `invalid`, which is the initial state of sender automaton in Line 11).

Figure 10(b) shows the system instance for MSI protocol with 2 processes. The instance has all the paths in $1\mathrm{E}_p$, therefore 2 is the cut-off for this protocol.

6.5.2 *Directory-based protocols.* Unlike snoopy protocols, the caches in directory-based protocols cannot communicate with each other. In directory-based protocols, there is a "home" directory that possesses the memory blocks, and there is a one-to-one commu-

```
process processor {
 [invalid]        ->[req-shared, ask-shared]
 [invalid]        ->[req-excl, ask-excl]
 [invalid, read]  ->[invalid]
 [invalid, write]->[invalid]
 [shared]         ->[req-shared, ask-shared]
 [shared]         ->[req-excl, ask-excl]
 [shared, read]   ->[shared]
 [shared, write]  ->[invalid]
 [req-shared]     ->[shared, read+]
 [req-shared, read]  ->[invalid]
 [req-shared, write] ->[invalid]
 [req-excl]          -> [excl, write+]
 [req-excl, read]    -> [invalid]
 [req-excl, write]   -> [invalid]
 [excl, read]        -> [invalid]
 [excl, write]       -> [invalid]
}
process internal-trans {
 [dummy, ask-shared] -> [dummy]
 [dummy, ask-excl]   -> [dummy]
}
initialstates{ invalid, dummy }
```

Fig. 11. Behavioral automata for the ESI protocol. [Emerson and Kahlon 2003] show that the verification of the German's cache coherence protocol reduces to the ESI protocol.

nication between the directory and each of the caches.

An example of directory-based cache coherence protocols is the one suggested by Steven German [German 2000]. In this protocol, every cache has three possible states: invalid, shared and exclusive. As clients do not communicate with each other, the home shares with every client three channels through which the home can communicate with each client: one for the client to request the memory block in the shared or the exclusive state, the second for the home to send an invalidation or grant access to the requesting client, and the last one is used by client to acknowledge the invalidation requests by the home directory.

The problem with verifying the directory-based cache coherence protocol is that they contain so many interleaving steps which make them behaviorally more complex and harder to reason about than snoopy protocol [Emerson and Kahlon 2003].

To reduce the complexity of verifying directory-based protocols, Emerson and Kahlon show that the verification of these protocols can be reduced to the snoopy ones [Emerson and Kahlon 2003]. They also show the reduction from the German's cache coherence protocol to the ESI snoopy protocol, a modification of the MSI protocol presented earlier; we therefore generate the cut-off for the ESI protocol.

ESI Protocol. In the ESI protocol, a cache can be in one of three states: `exclusive`, `shared` or `invalid`. It has almost the same behavior as the behavior of MSI protocol displayed in Figure 9(a), with `modified` state named as `exclusive`. The only difference in ESI is that, when a cache is in state `exclusive` and receives event `read`, the cache changes its state to `invalid`, as opposed to MSI protocol where a cache in `modified` goes to `shared` state upon receiving `read` event.

An $\mathtt{LTL}\backslash\mathtt{X}$ example property φ to verify in the German's protocol is that, if a cache request a shared access to the memory block, the cache will eventually be granted the access from the home directory. Since ESI doesn't have such scenario (i.e. only read events are broadcast) [Emerson and Kahlon 2003] added two states to the ESI protocol, namely `request_shared` and `request_exclusive`. Before the cache broadcasts the event `read` to be in the `shared` state, it makes an internal transitions to change its current state to `request_shared`, then it can change its state to `shared` by broadcasting the event `read`. Similarly, to go to the exclusive state, the cache needs to make an internal transition to change its current state to `request_exclusive`, then it can change its state to `exclusive` by broadcasting the event `write`. The property φ can now be represented as $\mathtt{G(request\_shared} \Rightarrow \mathtt{F(shared))}$ [Emerson and Kahlon 2003].

Figure 11 shows the behavioral automata for the ESI protocol. Caches are initially in invalid state (Line 23). We model an internal transition by creating the process type `internal-trans` (Lines 19-22). The sole goal of a process of this type is to be a recipient of the events `ask-shared` and `ask-excl` (Lines 20, 21 respectively). These event are the ones that a cache sends when it wants to go to the `shared` state (automata in Lines 2, 6) and the `exclusive` state (automata in Lines 3, 7), respectively. Once a cache is in the state `req-shared`, it can broadcast the event `read` and change its state to `shared` (automaton in Line 10). Similarly, in state `req-excl`, the cache can go to state `excl` by broadcasting event `write` (automaton in Line 13). The property φ can be represented as `G(ask_shared` $\Rightarrow$ `F(read))`

Our technique produced a cut-off of 2 for the ESI protocol, as opposed to the cut-off 7 presented in [Emerson and Kahlon 2003].

7. PERFORMANCE OF GOLOK

In the previous section, we showed that, for 14 out of 15 cases, the values of the cut-offs generated by Golok are smaller than the ones presented by existing techniques. In this section, we show the performance results of Golok in terms of the sizes of the system instances bounded by the cut-off.

Table II shows the performance results of running Golok on all the presented case studies. All experiments were run on a single core Pentium 4, 2.53 Ghz with 2 GB of RAM. The system instances generated by Golok are generally small in size, which may reduce the verification cost of these systems.

Table II. Performance Results of Golok

Protocol	**Protocol Spec**			GOLOK PERFORMANCE		
	# of params	**# of BA**	**Broadcast Comm.**	**Cut-off($\bar{k}_t$)**	**States in $sys(\bar{k}_t)$**	**Time(sec)**
Dining Philosophers	1	12		2	15	3.5
RLDP	(r, l, f) 3	20	$\surd^{\dagger}$	$1_r, 1_l, 2_f$	9	3.6
Bounded-Buffer	(p, co) 2	4		$1_p, 1_{co}$	3	3.5
Spin Lock	1	4		2	3	3.5
Spin Lock (multi)	(t, o) 2	4		$2_t, 1_o$	6	3.3
DME	1	7		2_{fc}	6	3.4
DME (multi)	(fr, c) 2	9		$1_{fr}, 1_c$	5	3.3
MSI	1	13	$\surd$	2	5	3.4
MESI	1	22	$\surd$	2	10	3.5
MOESI	1	30	$\surd$	2	13	3.6
Berkeley	1	25	$\surd$	2	12	3.5
Synapse N+1	1	14	$\surd$	2	7	3.4
Firefly	1	25	$\surd$	2	10	3.5
Dragon	1	23	$\surd$	2	12	3.6
German's Cache	1	18	$\surd^{\pm}$	2	18	3.7

†: Communication is via both single-cast and broadcast.
± : The original protocol is not broadcast-based, however it can be reduced to the ESI protocol which is broadcast-based as shown in [Emerson and Kahlon 2003].
r: right philosopher, l: left philosopher, f: fork; p: producer, co: consumer; t: thread, o: object; fc: dme node, fr: forward dme node, c: critical dme node.

The way processes communicate in the system is the main criterion that affects the size of system instances. Systems where communication is broadcast-based tend to have instances with smaller size than systems where communication is done via single-cast.

As displayed in the table, even though the RLDP protocol has more parameters and more behavioral automata in its protocol specification than the classical Dining Philoso-

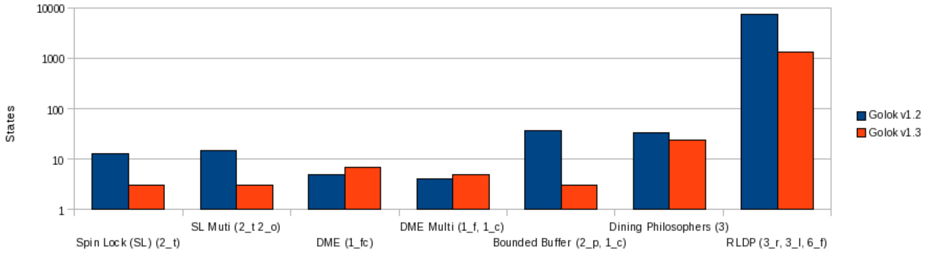


Fig. 12. Comparison between the old and new versions of Golok in terms of the size of the state space of the system instance bounded by the cut-off values generated from the old version of Golok. Synchronous-based communication in the new version of Golok results in smaller system instances vs. the asynchronous-based version of Golok.

phers protocol, the number of states in $sys(\bar{k}_t)$ is smaller in RLDP than in the Dining Philosophers. The reason behind such small number of states in the RLDP instance is that an eating philosopher in this protocol is modeled in a way that she releases both forks at the same time via broadcasting the release command. On the other hand, in the classical Dining Philosophers protocol, we modeled the philosopher to release one fork at a time via single-cast. Systems with single-cast communication therefore have more transitions than systems with broadcast-based communication and thus they have more states.

Synchronous communication also plays a role in reducing the size of the system instance state space. Figure 12 shows that the Golok version presented in this paper provides smaller system instances compared to our previous version. The reason behind such decrease in the size of state space is that the communication in the new version of Golok is purely synchronous, which reduces the number of possible transitions since an event is not sent (or broadcast) unless the receiver(s) is (are) ready to receive the event. The small state space of the system instance bound by the cut-off may result in reduced verification cost when verifying these parameterized systems.

8. RELATED WORK

In this section, we discuss different techniques that have tackled the problem of verifying parameterized systems and their limitation of being specific to certain classes of systems.

8.1 Abstraction techniques.

Abstraction techniques include counter abstraction [German and Sistla 1992; Pnueli et al. 2002] and environment abstraction [Clarke et al. 2006; 2008]. The idea behind counter abstraction is to abstract process identities, where every abstract state contains an abstract counter denoting the number of processes in the state.

Environment abstraction is the closest to our work. It follows a similar approach as counter abstraction; however, the counting is done for the number of processes satisfying a given predicate. An abstract environment is generated for a *reference* process to show all possible relations this process can have with its environment; which is closely related to our technique where we compute the maximal behavior that a process can induce in an arbitrary environment.

The advantage of these abstraction-based techniques is that they can verify a class of parameterized systems that we do not currently cover: systems whose individual processes are not necessarily finite-state such as the Lamport's Bakery algorithm [Lamport 1974] and Szymanski's algorithm [Szymanski 1988]; however, typically these techniques either require human guidance for obtaining the appropriate abstraction or are applicable to a certain restricted class of systems and/or properties (e.g., universal path properties).

8.2 Smart representation-based Techniques

Another class of techniques that rely on smart representation mechanism are techniques based on regular language [Bouajjani et al. 2000; Abdulla et al. 2002; Abdulla et al. 2007; Fisman et al. 2008] or graph-based [Saksena et al. 2008; Baldan et al. 2005; Baldan et al. 2008; Llorens and Oliver 2004] representations of the state-space of the parameterized system. These approaches are typically applicable for the verification of safety/reachability properties of parameterized systems. Recently, petri-net based representation has been proposed [Bouajjani et al. 2007] where tokens in the petri-net are used to denote the parameter of the system and a new logic, colored markings logic (CML), is developed to reason about such petri-nets. The work provides a generic framework for representing parameterized systems and identifies a fragment of CML for which the satisfiability problem is decidable.

Similar to the environment abstraction, some these techniques allow verification of systems with infinite-state processes. On the other hand, most of these techniques do not provide *algorithmic* techniques and focus only on one or two classes of systems while we provide a algorithmic (but incomplete) technique that works for several classes. We also present an implementation of our technique to prove its applicability.

8.3 Induction-based Techniques

Techniques based on induction use network invariants to reduce the problem of parameterized system verification to a finite state model checking problem [Wolper and Lovinfosse 1990; Arons et al. 2001; Clarke et al. 1995]. The idea behind these techniques is to find a network invariant I where the invariant is preserved by all computation steps of the system. Therefore, if I satisfies the desired property specification φ then the parameterized system also satisfies φ. The invariant generation process in these techniques usually requires manual guidance, while in our technique the cut-off generation is fully automatic.

Pnueli *et al.* [Pnueli et al. 2001] present a technique where invariants are computed automatically once the appropriate abstraction relation is provided. The advantadge of this technique is that it covers the bounded-data parameterized systems that we currently do not model; however, the technique works for systems with only one type of parameterized processes and it is not clear how it may be generated to multi-parameterized systems.

8.4 Cut-off Computation Techniques

Closest to our technique is the class of solutions that focus on computing a cut-off of the system parameter [Emerson and Namjoshi 1995; Emerson and Kahlon 2003; Emerson et al. 2006; Emerson and Namjoshi 1996; Emerson and Kahlon 2002; Ball et al. 2001; Yang and Li 2010; Ip and Dill 1996; Bingham and Hu 2005; Siirtola 2010]. This class of solutions aims to reduce the parameterized system verification problem to an equivalent finite-state one, is based on finding an appropriate cut-off k of the parameter of the system. The objective is to establish that a property is satisfied by the system with k processes if and only if it is satisfied by any number ($\geq k$) processes. Emerson and Namjoshi [Emerson and Namjoshi 1995] provide such cut-off values for different types of properties of parameterized systems with ring topology.

In contrast to the above techniques, our approach is fully automatic, does not depend on a specific representation mechanism of the system and/or property and is independent of the communication topology of the processes in the system. We present a sound but incomplete method which takes as input the description of the parameterized system in terms of standard input/output automata and establishes the cut-off of the parameter value. While Emerson and Namjoshi [Emerson and Namjoshi 1995] establish for the first time the cut-off bound for any parameterized system with ring topology given a specific type of property; we show that by considering the parameterized system being verified in the computation of the cut-off, a tighter bound of the cut-off can be obtained. For instance, using Emerson and Namjoshi's approach [Emerson and Namjoshi 1995], to verify the property that in a parameterized system with ring topology two processes cannot enter the critical section at the same, the cut-off size is 4; while the cut-off value identified using our approach for a specific parameterized system with ring topology (token ring protocol) is only 2. In short, while Emerson and Namjoshi [Emerson and Namjoshi 1995] focus on obtaining a generic cut-off for parameterized systems with a specific topology, the central theme of our technique is to develop a generic approach that can be applied to parameterized systems independent of the communication topology.

9. CONCLUSION AND FUTURE WORK

Verification of correctness properties for parameterized systems is an important problem [German and Sistla 1992; Pnueli et al. 2002; Clarke et al. 2006; 2008; Emerson and Namjoshi 1995; Emerson and Kahlon 2003; Emerson et al. 2006]. Considering that this problem is undecidable in general [Apt and Kozen 1986], techniques and heuristics for solving it for a subset of scenarios is an equally important problem. To that end, computing the cut-off of the system parameter is shown to be an effective technique for solving the parameterized verification problem [Emerson and Namjoshi 1995; Emerson and Kahlon 2003; Emerson et al. 2006].

In contrast to the existing techniques, our approach, based on behavioral-automata composition, can be applied to any parameterized systems independent of the communication topology. It provides a fully automatic sound but incomplete method for generating system cut-off for a parameterized system, where there exists more than one process type whose processes are unbounded and where processes can communicate via both single-cast and broadcast. Furthermore, effectively utilizing system descriptions allows us to obtain a system-specific cut-off, which in 14 out of 15 cases were found to be lower than previously discovered bounds. A system cut-off, to a large extent, dictates the state space that needs to be explored by a formal verification technique. The systematic approach of finding this cut-off that our approach provides is thus an important and foundational advance towards improved scalability of formal verification techniques. We also implemented our technique in a tool, Golok, to show the applicability of our technique.

Future work includes extending the theoretical and practical treatment of behavioral-automata composition to explore more expressive representation of protocols that can capture parameterized systems with infinite-domain data. It would also be sensible to develop techniques to extract behavioral automata from common programming languages such that Golok can be applied to large real-world parameterized software systems.

REFERENCES

ABDULLA, P. A., DELZANNO, G., HENDA, N. B., AND REZINE, A. 2007. Regular model checking without transducers (on efficient verification of parameterized systems). In *TACAS'07: Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*. Springer-Verlag, London, UK, 721–736.

ABDULLA, P. A., JONSSON, B., NILSSON, M., AND D'ORSO, J. 2002. Regular model checking made simple and efficient. In *CONCUR '02: Proceedings of the 13th International Conference on Concurrency Theory*. Springer-Verlag, London, UK, 116–130.

ABELSON, H., DYBVIG, R. K., HAYNES, C. T., ROZAS, G. J., ADAMS IV, N. I., FRIEDMAN, D. P., KOHLBECKER, E., STEELE, JR., G. L., BARTLEY, D. H., HALSTEAD, R., OXLEY, D., SUSSMAN, G. J., BROOKS, G., HANSON, C., PITMAN, K. M., AND WAND, M. 1998. Revised report on the algorithmic language scheme. *Higher Order Symbol. Comput. 11,* 1, 7–105.

ANDERSON, T. E. 1990. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst. 1,* 1, 6–16.

APT, K. R. AND KOZEN, D. C. 1986. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett. 22,* 6, 307–309.

ARONS, T., PNUELI, A., RUAH, S., XU, J., AND ZUCK, L. D. 2001. Parameterized verification with automatically computed inductive assertions. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*. Springer-Verlag, London, UK, 221–234.

BALDAN, P., CORRADINI, A., AND KÖNIG, B. 2008. A framework for the verification of infinite-state graph transformation systems. *Inf. Comput. 206,* 7, 869–907.

BALDAN, P., KÖNIG, B., AND RENSINK, A. 2005. Summary 2: Graph grammar verification through abstraction. In *Graph Transformations and Process Algebras for Modeling Distributed and Mobile Systems*, B. König, U. Montanari, and P. Gardner, Eds. Number 04241 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany.

BALL, T., CHAKI, S., AND RAJAMANI, S. K. 2001. Parameterized verification of multithreaded software libraries. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, London, UK, 158–173.

BASU, S. AND RAMAKRISHNAN, C. R. 2006. Compositional analysis for verification of parameterized systems. *Theor. Comput. Sci. 354,* 2, 211–229.

BINGHAM, J. AND HU, A. J. 2005. Empirically efficient verification for a class of infinite-state systems. In *TACAS '05: Proceedings of the 11th international conference on Tools and algorithms for the construction and analysis of systems*. Springer, 77–92.

BOUAJJANI, A., JONSSON, B., NILSSON, M., AND TOUILI, T. 2000. Regular model checking. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*. Springer-Verlag, London, UK, 403–418.

BOUAJJANI, A., JURSKI, Y., AND SIGHIREANU, M. 2006. Reasoning about Dynamic Networks of Infinite-State Processes with Global Synchronization. CCSd/HAL : e-articles server (based on gBUS) [http://hal.ccsd.cnrs.fr/oai/oai.php] (France).

BOUAJJANI, A., JURSKI, Y., AND SIGHIREANU, M. 2007. A generic framework for reasoning about dynamic networks of infinite-state processes. In *TACAS*. Springer-Verlag, London, UK, 690–705.

CLARKE, E. M., GRUMBERG, O., AND JHA, S. 1995. Verifying parameterized networks using abstraction and regular languages. In *CONCUR*. Springer, London, UK, 395–407.

CLARKE, E. M., TALUPUR, M., AND VEITH, H. 2006. Environment abstraction for parameterized verification. In *VMCAI*. Springer, London, UK, 126–141.

CLARKE, E. M., TALUPUR, M., AND VEITH, H. 2008. Proving ptolemy right: The environment abstraction framework for model checking concurrent systems. In *TACAS*. Springer, London, UK, 33–47.

DIJKSTRA, E. 1978. Two starvation free solutions to a general exclusion problem. EWD 625, Plataanstraat 5, 5671 AL Neunen, The Netherlands.

EMERSON, E. A. 1990. Temporal and modal logic. *Handbook of theoretical computer science (vol. B): formal models and semantics*, 995–1072.

EMERSON, E. A. AND KAHLON, V. 2002. Model checking large-scale and parameterized resource allocation systems. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, London, UK, 251–265.

EMERSON, E. A. AND KAHLON, V. 2003. Exact and efficient verification of parameterized cache coherence protocols. In *CHARME*. Springer, London, UK, 247–262.

EMERSON, E. A. AND NAMJOSHI, K. S. 1995. Reasoning about rings. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 85–94.

EMERSON, E. A. AND NAMJOSHI, K. S. 1996. Automatic verification of parameterized synchronous systems (extended abstract). In *CAV*. Springer, London, UK, 87–98.

EMERSON, E. A. AND NAMJOSHI, K. S. 1998. On model checking for non-deterministic infinite-state systems. In *LICS '98: Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, Washington, DC, USA, 70.

EMERSON, E. A., TREFLER, R. J., AND WAHL, T. 2006. Reducing model checking of the few to the one. In *ICFEM*. Springer, London, UK, 94–113.

FISMAN, D., KUPFERMAN, O., AND LUSTIG, Y. 315-331, 2008. On verifying fault tolerance of distributed protocols. In *TACAS*. Springer, London, UK.

GANSNER, E. R. AND NORTH, S. C. 1999. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience 30*, 1203–1233.

GERMAN, S. M. 2000. *Private Communication*.

GERMAN, S. M. AND SISTLA, A. P. 1992. Reasoning about systems with many processes. *J. ACM 39,* 3, 675–735.

HANDY, J. 1993. *The cache memory book*. Academic Press.

HANNA, Y., BASU, S., AND RAJAN, H. 2009. Behavioral automata composition for automatic topology independent verification of parameterized systems. In *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, New York, NY, USA, 325–334.

HANNA, Y., SAMUELSON, D., BASU, S., AND RAJAN, H. 2010. Automating cut-off for multi-parameterized systems. In *ICFEM 2010: Proceedings of the 12th Internation Conference on Formal Engineering Methods*. Springer, London, UK, 338–354.

IP, C. N. AND DILL, D. L. 1996. Verifying systems with replicated components in murphi. In *CAV*. Springer, London, UK, 147–158.

LAMPORT, L. 1974. A new solution of dijkstra's concurrent programming problem. *Commun. ACM 17,* 8, 453–455.

LLORENS, M. AND OLIVER, J. 2004. Introducing structural dynamic changes in petri nets: Marked-controlled reconfigurable nets. In *ATVA*. Springer, London, UK, 310–323.

LYNCH, N. A. AND TUTTLE, M. R. 1989. An introduction to input/output automata. *CWI Quarterly 2*, 219–246.

MANNA, Z. AND PNUELI, A. 1990. An exercise in the verification of multi-process programs. 289–301.

MANNA, Z. AND PNUELI, A. 1995. Verification of parameterized programs. In *in Specification and Validation Methods*. University Press, 167–230.

MILNER, R. 1982. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

PNUELI, A., RUAH, S., AND ZUCK, L. D. 2001. Automatic deductive verification with invisible invariants. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, London, UK, 82–97.

PNUELI, A., XU, J., AND ZUCK, L. D. 2002. Liveness with (0, 1, infty)-counter abstraction. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*. Springer-Verlag, London, UK, 107–122.

ROSCOE, A. W. AND LAZIC, R. 1998. Verifying Determinism of Concurrent Systems Which Use Unbounded Arrays. In *Proceedings of INFINITY'98*. extended version as Oxford University Computing Laboratory TR-2-98.

SAKSENA, M., WIBLING, O., AND JONSSON, B. 2008. Graph grammar modeling and verification of ad hoc routing protocols. In *TACAS'08/ETAPS'08: Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*. Springer-Verlag, London, UK, 18–32.

SAMUELSON, D., HANNA, Y., BASU, S., AND RAJAN, H. 2010. http://www.cs.iastate.edu/~slede/golok.

SIIRTOLA, A. 2010. Automated multiparameterized verification by cut-offs. In *ICFEM 2010: Proceedings of the 12th Internation Conference on Formal Engineering Methods*. Springer, London, UK.

SILBERSCHATZ, A., GALVIN, P. B., AND GAGNE, G. 2004. *Operating System Concepts*. Wiley.

SZYMANSKI, B. K. 1988. A simple solution to lamport's concurrent programming problem with linear wait. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*. ACM, New York, NY, USA, 621–626.

VARDI, M. Y. 2001. Branching vs. linear time: Final showdown. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS 2001. Springer-Verlag, London, UK, 1–22.

WOLPER, P. AND LOVINFOSSE, V. 1990. Verifying properties of large sets of processes with network invariants. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*. Springer-Verlag New York, Inc., New York, NY, USA, 68–80.

YANG, Q. AND LI, M. 2010. A cut-off approach for bounded verification of parameterized systems. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM, New York, NY, USA, 345–354.