Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 198 (2008) 93–99
www.elsevier.com/locate/entcs

Distributed Partial Order Reduction for Security Protocols

M. Torabi Dashti, A. Wijs and B. Lisser[1]

CWI, Amsterdam

Abstract

We describe a distributed partial order reduction algorithm for security protocols. Some experimental results using an implementation of the algorithm in the distributed μCRL toolset are also reported.

Keywords: Partial Order reduction, Distributed state space generation, Security protocols

1 Introduction

We have recently developed a partial order reduction (POR) technique for model checking security protocols, fully described in [8]. The algorithm is based on breadth-first exploration and preserves LTL_{-X} properties. In this paper we report on extending our POR algorithm to a distributed setting.

In the following, we first briefly describe the POR algorithm of [8] for security protocols. Next, we adapt the algorithm to a distributed setting, resulting in, as we call it, DPOR, which we present in pseudo-code. Some experimental results and discussions on effectiveness and scalability of DPOR are reported in section 5. Our implementation of DPOR is available at `http://www.cwi.nl/~mcrl/`. Finally, section 6 concludes the paper with our future and related work.

2 Preliminaries

Labelled transition systems (LTS) An LTS is a tuple (Σ, s_0, Act, Tr), where Σ is a set of states, $s_0 \in \Sigma$ is the initial state, Act is a set of actions and $Tr \subseteq \Sigma \times Act \times \Sigma$ is the transition relation. We write $s \xrightarrow{a} s'$ when $(s, a, s') \in Tr$. The set

[1] Email: {dashti,a.j.wijs,bertl}@cwi.nl

doi:10.1016/j.entcs.2007.10.022

of enabled actions in state s is $en(s) = \{a \in Act \mid \exists s' \in \Sigma.\ s \xrightarrow{a} s'\}$. For $A \subseteq Act$, let $nxt(s, A) = \{s' \in \Sigma \mid \exists a \in A.\ s \xrightarrow{a} s'\}$.

Security protocols We model a security protocol as a finite number of processes and an asynchronous communication subsystem. The set of honest processes is P, and one single intruder process models all malicious participants and, moreover, controls the communication channels. We consider the Dolev-Yao intruder model [7]. Each process $p \in P$ is modelled as a finite, acyclic and uniquely named LTS. As security protocols normally have only finite runs, our POR algorithm aims at cycle-free state spaces. To interact with the communication subsystem, a process $p \in P$ uses actions $send_p(m)$ and $recv_p(m)$, in which message m is produced and consumed, respectively. Apart from these two actions, all other actions of processes are assumed internal, i.e. not interacting with the communication subsystem. An internal action is called *invisible* if it does not appear in the properties being verified. Else, it is called *visible*. We assume that all internal actions of process p contain p as a subscript, e.g. $secret_p(m)$ can be an internal action of p performed when p concludes that m is a secret. The set of actions of a process can thus be partitioned into four disjoint sets: $Act_p = V_p \cup I_p \cup S_p \cup R_p$, with V_p and I_p denoting the set of visible and the set of invisible internal actions of p, respectively, $S_p = \{send_p\}$ and $R_p = \{recv_p\}$. Since all these actions are sub-scripted, we have $\forall p, q \in P.\ p \neq q \implies Act_p \cap Act_q = \emptyset$. We write $V = \cup_{p \in P} V_p$, and similarly for I, S and R. As $Act = \cup_{p \in P} Act_p$, we have $Act = V \cup I \cup S \cup R$.

3 POR for security protocols

Algorithm 1 shows our POR algorithm for security protocols. The idea is that *send* and invisible internal actions can be prioritised over other actions. Moreover, if one action of process p is explored at state s, then all other actions of $Act_p \cap en(s)$ have to be explored at s as well. See [8] for a detailed presentation of *why* this algorithm works.

We define the projection function $\pi : Act \to P$ as $\pi(a) = p,\ \textit{if}\ a = a_p(m),\ a_p \in Act_p$. We define the relation $\sim \subseteq Act \times Act$ as $\forall a, a' \in Act.\ a \sim a' \iff \pi(a) = \pi(a')$. The relation $\sim$ is an equivalence, and thus partitions Act into equivalence classes such that each class contains only actions performed by one particular process. The set of all equivalence classes in $A \subseteq Act$ given the equivalence relation $\sim$ is denoted $A/\sim$. For $A \subseteq Act$, we let $\mathbb{V}(A) = A \cap V$ and $\mathbb{R} = A \cap R$.

Note that the purpose of using the *Expanded* set is to avoid state revisits, and it is not needed to guarantee the termination of the algorithm as the LTSs are cycle free. Therefore, this set can gradually be removed from memory and be stored on disks, in case memory limits are reached, without endangering the termination of the algorithm. For similar approaches to memory management see [3,9].

Algorithm 1 POR for security protocols

Current := $\{s_0\}$
Expanded := $\emptyset$ //used for duplicate detection, in case confluent traces exist
while *Current* $\setminus$ *Expanded* $\neq \emptyset$ **do**
  Next := $\emptyset$
  for all $s \in$ *Current* $\setminus$ *Expanded* **do**
    Construct $en(s)/\sim$ and name its elements as $c_1, \ldots, c_\ell$.
    $T := \{c_i \mid i \in \{1, \ldots, \ell\}, \mathbb{V}(c_i) \cup \mathbb{R}(c_i) = \emptyset\}$
    if $T \neq \emptyset$ **then**
      Pick the smallest $c \in T$
      Next := *Next* $\cup$ $nxt(s, c)$
    else if $T = \emptyset$ **then**
      Next := *Next* $\cup$ $nxt(s, en(s))$
    end if
  end for
  Expanded := *Expanded* $\cup$ *Current*
  Current := *Next*
end while

4 DPOR: Distributed POR for security protocols

As is evident from the pseudo-code of algorithm 1, when expanding state s, the decision to prune (some of its outgoing transitions) depends solely on $en(s)$ and can therefore be made locally. This implies that in a distributed implementation, no extra communication is needed for the POR pruning part.

Distributed state space generation algorithms usually comprise the code ran by each client and the code for a manager process. In algorithm 2, we show the part of our DPOR algorithm which is ran by the clients. The manager keeps track of the progress and decides to terminate the generation once no new states are found. We here omit the code of the manager process as it has no interactions with the POR pruning part. See, e.g., [12] for a generic specification of the manager process.

In algorithm 2, we have included standard constructs for distributed state space generation: each client has a unique identity `ID`, and is provided with a hash function h that assigns to each state a unique "owner" client which is responsible for its expansion. The procedure *SendToClientsNextLevel*(S), with S being a set of states, sends the states of S to their corresponding owners (determined by h). Conversely, the procedure S := *ReceiveFromClientsNextLevel*() receives from all clients the states that are to be processed by the current client and returns them in the set S. The procedures *RecvFromMgr*() and *SendToMgrNewStatesFound*(`ID`, $\mid S \mid > 0$) are used to communicate with the manager. The procedure *RecvFromMgr*() asks the manager if the client should continue the search and *SendToMgrNewStatesFound* reports to the manager whether any new states have been found in the current round.

Algorithm 2 DPOR for security protocols

```
REQUIRES: ID , h : Σ → {IDs}
  Expanded := ∅; Current := ∅
  if h(s_0) = ID then
    Current := {s_0}
  end if
  repeat
    Next := ∅
    for all s ∈ Current \ Expanded do
      Construct en(s)/∼ and name its elements as c_1, ..., c_ℓ.
      T := {c_i | i ∈ {1, ..., ℓ}, 𝕍(c_i) ∪ ℝ(c_i) = ∅}
      if T ≠ ∅ then
        Pick the smallest c ∈ T
        Next := Next ∪ nxt(s, c)
      else if T = ∅ then
        Next := Next ∪ nxt(s, en(s))
      end if
    end for
    SendToMgrNewStatesFound(ID, | Next |> 0)
    command := RecvFromMgr()
    if command ≠ finish then
      SendToClientsNextLevel(Next)
      Expanded := Expanded ∪ Current
      Current := ReceiveFromClientsNextLevel()
    end if
  until command = finish
```

5 Experimental results

This section reports some experimental results using an implementation of DPOR, explained in section 4, in the distributed μCRL toolset (e.g. [1,2]).

To demonstrate the effectiveness of the proposed POR algorithm we have modelled a Digital Rights Management (DRM) protocol, described in detail in [11]. Below, we shortly describe this protocol and our experimental results.

The protocol of [11] comprises a finite set of trusted content rendering devices C and a finite set of trusted entities T. The goal of the protocol is to provide a secure environment for fair exchange of digital items among the members of C, in presence of malicious players. In case of a malicious act, the suffered party resorts to one of the trusted entities. The set D contains the items available in the protocol. Each item is bundled with a right declaring the terms of use of that particular item. The set of rights is denoted R. To keep the state space finite, each $c \in \mathsf{C}$ only has access to a finite set of fresh nonces N_c to start new sessions. Below we consider $|\mathsf{C}| = 2$ and $|\mathsf{D}| = |\mathsf{R}| = 1$.

Table 1
Effectiveness of DPOR. (Time is in min:sec format.)

Instance		Exhaustive		DPOR		Reduction	
$\|\mathsf{N}_c\|$	$\|\mathsf{T}\|$	# States	Time	# States	Time	in # States	in Time
1	2	89,155	02:27.74	45,871	01:56.49	48.5%	21.2%
2	2	277,459	06:23.82	145,559	05:21.97	47.5%	16.1%
1	3	2,674,940	52:47.48	1,082,122	34:09.28	59.5%	35.3%
2	3	11,896,384	269:08.64	4,794,745	169:40.64	59.7%	36.9%

5.1 Experiment setting

In all our experiments, we use machines with a single 64 bit Athlon 2.2 GHz CPU and 1 GB RAM, running Fedora Core 6 Linux, connected with Gigabit Ethernet (1Gbps). In the following, measured "time" refers to elapsed time (wall clock time), that is the time taken from the beginning till the end of an experiment. Therefore, this is not only computation time, but also reflects periods of waiting, etc. The experiments were performed using the μCRL toolset version 2.17.13.

5.2 Effectiveness of DPOR

In this section, we discuss the effectiveness of our DPOR algorithm by comparing the number of states and generation time when using the DPOR algorithm versus using exhaustive distributed breadth-first state space generation (both implemented in the μCRL toolset). In table 1, we consider four instances of the DRM protocol described above. The number of fresh nonces available to each rendering device is denoted by $|\mathsf{N}_c|$, and $|\mathsf{T}|$ denotes the number of concurrent trusted party processes. The time column shows the amount of time (in min:sec) required by 16 machines to complete the generation task. It is worth mentioning that next-state generation is in general relatively slow in security protocols, as it involves matching the messages that protocol participants can receive with the messages that the attacker process can construct.

In table 1, we observe that approximately a 50% reduction in the number of states is achieved. Although DPOR loads the generation algorithm with some book-keeping and extra computations, the gained reduction definitely compensates for these costs, as is evident from the time columns.

5.3 Scalability of DPOR

In table 2, we compare the generation time required by DPOR using different numbers of machines. As was explained earlier, since the DPOR algorithm does not require extra communications to synchronise on the POR pruning part, we expect it to scale up well. Figure 1 shows the results of this table on log-scale graphs. These measurements indeed confirm that DPOR exhibits reasonable scalability.

A phenomenon that can be observed in the case of $(|\mathsf{N}_c|, |\mathsf{T}|) = (2, 3)$, is that the speed-up factor [2] for 8 and 16 machines experiments are absurdly high. Although

[2] We adopt the following definition of speed-up factor: $\frac{t_1}{t_n}$, where t_1 is the time required by one machine

Table 2
Scalability of DPOR.

Instance \ # Machines		1	4	8	16
$\|N_c\|$	$\|T\|$	Time	Time	Time	Time
1	2	11:56.44	04:45.27	03:16.72	01:56.49
2	2	43:14.15	15:54.48	10:15.75	05:21.97
1	3	374:33.38	123:44.43	71:45.94	34:09.28
2	3	2676:34.70	1286:15:36	356:18.81	168:05.45

reasoning based on speed-up factor has many defects, e.g. see [6], nonetheless, we find it illuminating to discuss why this behaviour occurs.

Through the experiments on the (2,3) case, we witnessed that when using 1 and 4 machines, the available RAM of the machines is not enough, and therefore the operating system starts swapping, i.e. using high-latency disk memory besides RAM. Therefore, these experiments take much more time than expected. For instance, the reduced state space of the (2,3) case is roughly 5 times larger than the reduced state space of the case (1,3), see table 1, whereas table 2 shows that when using 1 and 4 machines, the (2,3) case is about nine times slower than the case (1,3). This is because of the time penalty that is imposed by swapping.

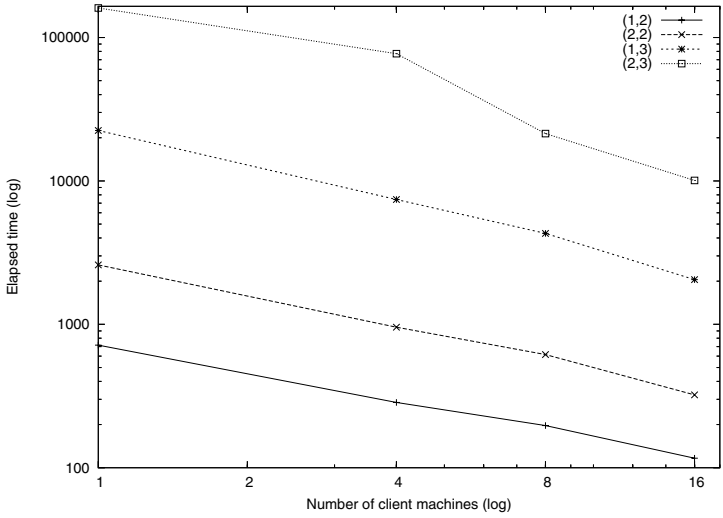

Fig. 1. Elapsed time in DPOR

6 Future and related work

Future work We consider experimenting with various classes of security protocols to assess the effectiveness of DPOR in different settings.

Related work When it comes to distributed POR, a major issue is to find an efficient way to satisfy the *cycle condition*, see, e.g., [5]. Notable papers which

to perform the generation using the parallel algorithm, and t_n is the time required by n parallel machines to perform the same job.

tackle this problem in the distributed setting are [4,10]. Our work is in some sense orthogonal to these papers, as the cycle condition is irrelevant in cycle-free state spaces, which is our framework. In contrast, we propose an algorithm to efficiently find a suitable subset of $en(s)$ to be explored at each state s, with solely local inspections.

Acknowledgement

We are grateful to Jaco van de Pol, Michael Weber and anonymous reviewers for their valuable comments on our results, and to Jens Calamé for proof reading the paper.

References

[1] S. Blom, J. Calamé, B. Lisser, S. Orzan, J. Pang, J. van de Pol, M. Torabi Dashti, and A. Wijs. Distributed analysis with μCRL: A compendium of case studies. In *TACAS '07*, volume 4424 of *LNCS*, pages 683–689. Springer, 2007.

[2] S. Blom, W. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J. van de Pol. μCRL: A toolset for analysing algebraic specifications. In *CAV '01*, volume 2102 of *LNCS*, pages 250–254. Springer, 2001.

[3] G. Behrmann, K. Larsen, and R. Pelánek. To store or not to store. In *CAV '03*, volume 2725 of *LNCS*, pages 433–445. Springer, 2003.

[4] L. Brim, I. Černá, P. Moravec, and J. Šimša. Distributed partial order reduction of state spaces. In *PDMC '04*, volume 128 of *ENTCS*, pages 63–74. Springer, 2005.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[6] L. Crowl. How to measure, present, and compare parallel performance. *IEEE Parallel and Distributed Technology*, 2(1):9–25, 1994.

[7] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. on Information Theory*, IT-29(2):198–208, 1983.

[8] W. Fokkink, M. Torabi Dashti, and A. Wijs. Partial order reduction for branching security protocols. In *Proc. WITS '07*, pages 178–193, 2007. http://homepages.cwi.nl/~dashti/p/por.pdf.

[9] M. Hammer and M. Weber. "To Store or Not To Store" reloaded: Reclaiming memory on demand. In *Formal Methods: Applications and Technology*, volume 4346 of *LNCS*, pages 51–66. Springer, 2006.

[10] R. Palmer and G. Gopalakrishnan. A distributed partial order reduction algorithm. In *FORTE '02*, volume 2529 of *LNCS*, page 370. Springer, 2002.

[11] M. Torabi Dashti, S. Krishnan Nair, and H. Jonker. Nuovo DRM paradiso: Towards a verified fair DRM scheme. In *FSEN '07*, volume 4767 of *LNCS*, pages 33–48, 2007.

[12] A. Wijs. *What to Do Next? Analysing and optimising system behaviour in time*. PhD thesis, Vrije Universiteit, 2007.