A BSP Algorithm for the State Space Construction of Security Protocols

Frédéric Gava, Michael Guedj, Franck Pommereau

A BSP algorithm for the state space construction of security protocols

Frédéric Gava, Michaël Guedj
LACL, University of Paris-East Créteil, France
Email: frederic.gava@univ-paris-est.fr, michael.guedj@univ-paris-est.fr

Franck Pommereau
IBISC, University of Évry, France
Email: franck.pommereau@ibisc.univ-evry.fr

Abstract

This paper presents a Bulk-Synchronous Parallel (BSP) algorithm to compute the discrete state space of structured models of security protocols. The BSP model of parallelism avoids concurrency related problems (mainly deadlocks and non-determinism) and allows us to design an efficient algorithm that is at the same time simple to express. A prototype implementation has been developed, allowing to run benchmarks showing the benefits of our algorithm.

1. Introduction

In a world strongly dependent on distributed data communication, the design of secure infrastructures is a crucial task. At the core of computer security-sensitive applications are security protocols, *i.e.*, sequences of message exchanges aiming at distributing data in a cryptographic way to the intended users and providing security guarantees. This leads to search for a way to verify whether a system is secure or not. Enumerative model-checking is well-adapted to this kind of asynchronous, non-deterministic systems containing complex data types. In this paper, we consider the problem of constructing the state space of *labelled transition systems* (LTS) that model security protocols.

Let us recall that the state space construction problem is the problem of computing the explicit representation of a given model from the implicit one. This space is constructed by exploring all the states reachable through a successor function from an initial state. Generally, during this operation, all the explored states must be kept in memory in order to avoid multiple exploration of a same state. Once the state space is constructed, or during its construction, it can be used as input for various verification procedures, such as reachability analysis or model-checking of temporal logic properties.

State space construction may be very consuming both in terms of memory and execution time: this is the so-called state explosion problem. The construction of large discrete state spaces is so a computationally intensive activity with extreme memory demands, highly irregular behavior, and poor locality of references. This is especially true when complex data-structures are used in the model as the knowledge of an intruder in security protocols. Because this construction can cause memory thrashing on single or multiple processor systems, it has led to consider exploiting the larger memory space available in distributed systems [1], [2]. Parallelize the state space construction on several machines is thus done in order to benefit from all the storage and computing resources of each machine. This allows to reduce both the amount of memory needed on each machine and the overall execution time.

Distributed state space construction. One of the main technical issues in the distributed memory state space construction is to partition the state space among the participating computers. Most of approaches to the distributed memory state space construction use a partitioning mechanism that works at the level of states which means that each single state is assigned to a machine. This assignment is made using a function that partitions the state space into subsets of states. Each such a subset is then “owned” by a single machine.

To have efficient parallel algorithms for state space construction, we see two requirements. First, the partition function must be computed quickly and defined such that a successor state is likely to be mapped to the same processor as its predecessor; otherwise the computation will be overwhelmed by inter-processor communications (the so called *cross transitions*) which obviously implies a drop of the locality of the computation and thus of the performances. Second, balancing of the workload is obviously needed [3] because it is necessary to fully profit from available computational power to achieve the expected speedup. In the case of state space construction, the problem is hampered by the fact that future size and structure of the undiscovered portion of the space space is unknown and cannot be predicted in general.

While it has been showed that a pure static hashing for the partition function can effectively balance the workload and achieve reasonable execution time as well [4], this method suffers from some obvious drawbacks [5], [6]. First, it causes too much cross transitions. Second, if ever in the course of the construction just one processor is so burdened with states that it exhausts its available main memory, the whole computation fails or slows too much due swapping.

Verifying Security protocols. Designing security protocols is complex and often error prone: various attacks are reported in the literature to protocols thought to be “correct” for many years. These attacks exploit weaknesses in the protocol that are due to the complex and unexpected interleavings of different protocol sessions as well as to the possible interference of malicious participants, *i.e.*, the attacker.

Furthermore, attacks are not as simple that they appear [7]: the attacker is powerful enough to perform a number of potentially dangerous actions as intercepting messages or replacing them by new ones using the knowledge it has previously gained; or it is able to perform encryption and decryption using the keys within its knowledge [8]. Consequently the number of potential attacks generally grows exponentially with the number of exchanged messages.

Formal methods offer a promising approach for automated security analysis of protocols: the intuitive notions are translated into formal specifications, which is essential for a careful design and analysis, and protocol executions can be simulated, making it easier to verify various security properties. Formally verifying security protocols is a well established domain that is still actively developed. Different approaches exist as [9], [10], [11] and tools are dedicated to this purpose as [12], [13].

Contribution. In this paper, we exploit the well-structured nature of security protocols and match it to a model of parallel computation called BSP [14], [15]. This allows us to simplify the writing of an efficient algorithm for computing the state space of finite protocol sessions. The structure of the protocols is exploited to partition the state space and reduce cross transitions while increasing computation locality. At the same time, the BSP model allows to simplify the detection of the algorithm termination and to load balance the computations.

Outline. First, we briefly review in Section 2 the context of our work that is the BSP model, models of security protocols and their state space representation as labelled transitions systems (LTS). Section 3 is dedicated to the description of our new algorithm. Then, in Section 4, we briefly describe a prototype implementation and apply it to some typical protocol sessions, giving benchmarks to demonstrate the benefits of our approach. Related works are discussed in Section 5 while a conclusion and future works are presented in Section 6.

2. Context and general definitions

2.1. The BSP model

In the BSP model, a computer is a set of uniform processor-memory pairs connected through a communication network allowing the inter-processor delivery of messages [15], [14]. Supercomputers, clusters of PCs, multi-core and GPUs, *etc.*, can be considered as BSP computers.

A BSP program is executed as a sequence of *super-steps* (see Fig. 1), each one divided into three successive disjoint phases: first, each processor only uses its local data to perform sequential computations and to request data transfers to other nodes; then, the network delivers the requested data; finally, a global synchronisation barrier occurs, making the transferred data available for the next super-step. The execution time (cost) of a super-step is the sum of the maximum of the local processing, the data delivery and the barrier times. The cost of a program is the total sum of the cost of its super-steps.

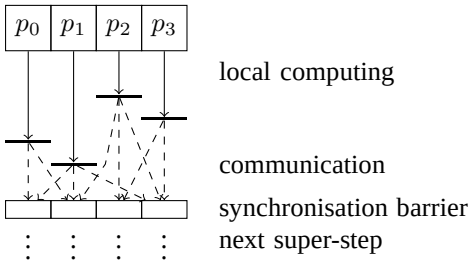

Fig. 1. A BSP super-step

On most of cheaper distributed architectures, barriers often become more expensive when the number of processors increases. However, dedicated architectures make them much faster and they have also a number of attractions. In particular, they dramatically reduce the risks of deadlocks or livelocks, since barriers do not create circular data dependencies.

The BSP model considers communication actions *en masse*. This is less flexible than asynchronous messages, but easier to debug since there are many simultaneous communication actions in a parallel program, and their interactions are usually complex. Bulk sending also provides better performances since it is faster to send a block of data rather than individual data because of less network latency.

2.2. State spaces of protocol models

A *labelled transition system* (LTS) is an implicit representation of the state space of a modelled system. It is defined as a tuple (S, T, ℓ) where S is the set of states, $T \subseteq S^2$ is the set of transitions, and ℓ is an arbitrary labelling on $S \cup T$. Given a model defined by its initial state s_0 and its successor function succ, the corresponding explicit LTS is $\mathsf{LTS}(s_0, \mathsf{succ})$, defined as the smallest LTS (S, T, ℓ) such that s_0 in S, and if $s \in S$ then for all $s' \in \mathsf{succ}(s)$ we also have $s' \in S$ and $(s, s') \in T$. The labelling may be arbitrarily chosen, for instance to define properties on states and transitions with respect to which model checking is performed.

In the paper, we consider models of security protocols involving a set of *agents* and we assume that any state can be represented by a function from a set $\mathcal{L}$ of *locations* to an arbitrary data domain $\mathcal{D}$. For instance, locations may correspond to local variables of agents, shared communication buffers, *etc.*

As a concrete formalism to model protocols, we have used an *algebra of coloured Petri nets* [16] allowing for easy and structured modelling. However, our approach is largely independent of the chosen formalism and it is enough to assume that the following properties hold:

(P1) Any state of the system can be described as a function $\mathcal{L} \to \mathcal{D}$.

(P2) There exists a subset $\mathcal{L}_R \subseteq \mathcal{L}$ of *reception locations* corresponding to the information learnt (and stored) by agents from their communication with others.

(P3) Function succ can be partitioned into two successor functions succ_R and succ_L that correspond respectively

to the successors that change states or not on the locations from $\mathcal{L}_R$.

More precisely: for all state s and all $s' \in \mathsf{succ}(s)$, if $s'|_{\mathcal{L}_R} = s|_{\mathcal{L}_R}$ then $s' \in \mathsf{succ}_L(s)$, else $s' \in \mathsf{succ}_R(s)$; where $s|_{\mathcal{L}_R}$ denotes the state s whose domain is restricted to the locations in $\mathcal{L}_R$. Intuitively, succ_R corresponds to transitions upon which an agent receives information and stores it. On concrete models, it is generally easy to distinguish syntactically the transitions that correspond to a message reception in the protocol with information storage. Thus, is it easy to partition succ as above. This is the case in particular for the algebra of Petri nets that we have used.

In the following, the presented algorithms compute only S. This is made without loss of generality and it is a trivial extension to compute also T and ℓ, assuming for this purpose that $\mathsf{succ}(s)$ returns tuples $(t, \ell(t), s', \ell(s'))$. This is usually preferred in order to be able to perform model-checking of temporal logic properties.

2.2.1. Dolev-Yao attacker. We consider models of protocols where a Dolev-Yao attacker [8] resides on the network. An execution of such a model is thus a series of message exchanges as follows. (1) An agent sends a message on the network. (2) This message is captured by the attacker that tries to learn from it by recursively decomposing the message or decrypting it when the key to do so is known. Then, the attacker forges all possible messages from newly as well as previously learnt information. Finally, these messages (including the original one) are made available on the network. (3) The agents waiting for a message reception accept some of the messages forged by the attacker, according to the protocol rules.

2.2.2. Sequential state space construction. In order to explain our parallel algorithm, we start with Algorithm 1 that corresponds to the usual sequential construction of a state space. The sequential algorithm involves a set $todo$ of states that is used to hold all the states whose successors have not been constructed yet; initially, it contains only the initial state s_0. Then, each state s from $todo$ is processed in turn and added to a set $known$ while its successors are added to $todo$ unless they are known already. At the end of the computation, $known$ holds all the states reachable from s_0, that is, the state space.

Algorithm 1 Sequential construction

```
1:  todo ← {s_0}
2:  known ← ∅
3:  while todo ≠ ∅ do
4:     pick s from todo
5:     known ← known ∪ {s}
6:     for s' ∈ succ(s) do
7:        if s' ∉ known ∪ todo then
8:           todo ← todo ∪ {s'}
9:        end if
10:    end for
11: end while
```

3. A BSP algorithm for state space construction

We now show how the sequential algorithm can be parallelised in BSP and how several successive improvement can be introduced. This results in an algorithm that remains quite simple in its expression but that actually relies on a precise use of a consistent set of observations and algorithmic modifications. We will show in the next section that this algorithm is efficient despite its simplicty.

3.1. A naive BSP version

Algorithm 1 can be naively parallelised by using a partition function cpu that returns for each state a processor identifier, *i.e.*, the processor numbered $\mathsf{cpu}(s)$ is the owner of s. Usually, this function is simply a hash of the considered state modulo the number of processors in the parallel computer. The idea is that each process computes the successors for only the states it owns. This is rendered as Algorithm 2; notice that we assume that arguments are passed by references so that they may be modified by sub-programs.

Algorithm 2 Naive BSP construction

```
1:  todo ← ∅
2:  total ← 1
3:  known ← ∅
4:  if cpu(s_0) = mypid then
5:     todo ← todo ∪ {s_0}
6:  end if
7:  while total > 0 do
8:     tosend ← Successor(known, todo)
9:     todo, total ← Exchange(known, tosend)
10: end while
```

$Successor(known, todo)$:

```
1:  tosend ← ∅
2:  while todo ≠ ∅ do
3:     pick s from todo
4:     known ← known ∪ {s}
5:     for s' ∈ succ(s) do
6:        if s' ∉ known ∪ todo then
7:           if cpu(s') = mypid then
8:              todo ← todo ∪ {s'}
9:           else
10:             tosend ← tosend ∪ {(cpu(s'), s')}
11:          end if
12:       end if
13:    end for
14: end while
15: return tosend
```

$Exchange(known, tosend)$:

```
1:  received, total ← BSP_EXCHANGE(tosend)
2:  return (received \ known), total
```

This is a SPMD (Single Program, Multiple Data) algorithm so that each processor executes it. Sets $known$ and $todo$ are

still used but become local to each processor and thus provide only a partial view on the ongoing computation. So, in order to terminate the algorithm, we use an additional variable $total$ in which we count the total number of states waiting to be proceeded throughout all the processors, *i.e.*, $total$ is the sum of the sizes of all the sets $todo$. Initially, only state s_0 is known and only its owner puts it in its $todo$ set. This is performed in lines 4–6, where mypid evaluates locally to each processor to its own identifier.

Function $Successor$ is then called to compute the successors of the states in $todo$. It is essentially the same as the sequential exploration, except that each processor computes only the successors for the states it actually owns. Each computed state that is not owned by the local processor is recorded in a set $tosend$ together with its owner number. This partitioning of states is performed in lines 7–11.

Then, function $Exchange$ is responsible for performing the actual communication between processors. The primitive **BSP_EXCHANGE** send each state s for a pair (i, s) in $tosend$ to the processor i and returns the set of states received from the other processors, together with the total number of exchanged states. The routine **BSP_EXCHANGE** performs a global (collective) synchronisation barrier which makes data available for the next super-step so that all the processors are now synchronised. Then, function $Exchange$ returns the set of received states that are not yet known locally together with the new value of $total$. Notice that, by postponing communication, this algorithm allows buffered sending and forbids sending several times the same state.

It can be noted that the value of $total$ may be greater than the intended count of states in $todo$ sets. Indeed, it may happen that two processors compute a same state owned by a third processor, in which case two states are exchanged but then only one is kept upon reception. Moreover, if this states has been also computed by its owner, it will be ignored. This not a problem in practise because in the next super-step, this duplicated count will disappear. In the worst case, the termination requires one more super-step during which all the processors will process an empty $todo$, resulting in an empty exchange and thus $total = 0$ on every processor, yielding the termination.

3.2. Increasing local computation time

Using Algorithm 2, function cpu distributes evenly the states over the processors. However, each super-step is likely to compute very few states because only few computed successors are locally owned. This results in a bad balance of the time spent in computation with respect to the time spent in communication. If more states can be computed locally, this balance improves but also the total communication time decreases because more states are computed during each call to function $Successor$.

To achieve this result, we consider a peculiarity of the models we are analysing. The learning phase of the attacker is computationally expensive, in particular when a message can be actually decomposed, which leads to recompose a lot of new messages. Among the many forged messages, only a (usually) small proportion are accepted for a reception by agents. Each such reception gives rise to a new state.

This whole process can be kept local to one processor. To do so, we need to design a new partition function cpu_R such that, for all states s_1 and s_2, if $s_1|_{\mathcal{L}_R} = s_2|_{\mathcal{L}_R}$ then $\mathsf{cpu}_R(s_1) = \mathsf{cpu}_R(s_2)$. For instance, this can be obtained by computing a hash (modulo the number of processors) using only the locations from $\mathcal{L}_R$.

On this basis, function $Successor$ can be changed as shown in Algorithm 3.

Algorithm 3 An exploration to improve local computation

$Successor(known, todo)$:

```
 1: tosend ← ∅
 2: while todo ≠ ∅ do
 3:    pick s from todo
 4:    known ← known ∪ {s}
 5:    for s' ∈ succ_L(s) \ known do
 6:       todo ← todo ∪ {s'}
 7:    end for
 8:    for s' ∈ succ_R(s) \ known do
 9:       tosend ← tosend ∪ {(cpu_R(s'), s')}
10:    end for
11: end while
12: return tosend
```

The rest is as in Algorithm 2.

With respect to Algorithm 2, this one splits the for loop, avoiding calls to cpu_R when they are not required. This may yield a performance improvement, both because cpu_R is likely to be faster that cpu and because we only call it when necessary. But the main benefits in the use of cpu_R instead of cpu is to generate less cross transitions since less states are need to be send. Notice that in the second loop, no state from $todo$ may be obtained through succ_R because of the progression. So we can use $\setminus known$ to replace test $s' \notin known \cup todo$ from the previous algorithm. Finally, notice that, on some states, cpu_R may return the number of the local processor, in which case the computation of the successors for such states will occur in the next super-step. We show now on how this can be exploited.

3.3. Decreasing local storage

One can observe that the structure of the computation is now matching closely the structure of the protocol execution: each super-step computes the executions of the protocol until a message is received. As a consequence, from the states exchanged at the end of a super-step, it is not possible to reach states computed in any previous super-step. Indeed, the protocol progression matches the super-steps succession.

This kind of progression in a model execution is the basis of the *sweep-line* method [17] that aims at reducing the memory footstep of a state space computation by exploring states in an

order compatible with progression. It thus becomes possible to regularly dump from the main memory all the states that cannot be reached anymore. Enforcing such an exploration order is usually made by defining on states a measure of progression. In our case, such a measure is not needed because of the match between the protocol progression and the super-steps succession. So we can apply the sweep-line method by making a simple modification of the exploration algorithm, as shown in Algorithm 4.

Algorithm 4 Sweep-line implementation

$Exchange(tosend, known)$:

1: **dump**$(known)$
2: **return BSP_EXCHANGE**$(tosend)$

The rest is as in Algorithm 3.

The statement **dump**$(known)$ resets $known$ to an empty set, possibly saving its content to disk if this is desirable. The rest of function $Exchange$ is simplified accordingly.

3.4. Balancing the computation

The final optimisation step aims at balancing the workload. To do so, we exploit the following observation: for all the protocols we have studied so far, the number of computed states during a super-step is usually closely related to the number of states received at the beginning of the super-step. So, before to exchange the states themselves, we can first exchange information about how many state each processor has to send and how they will be spread onto the other processors. Using this information, we can anticipate and compensate balancing problems.

To compute the balancing information, we use a new partition function cpu_B that is equivalent to cpu_R but works for an infinite number of processors, *i.e.*, we have $\mathsf{cpu}_R(s) = \mathsf{cpu}_B(s) \bmod P$, where P is the number of processors. In practise, cpu_B computes a hash using only information from the locations in $\mathcal{L}_R$, and without using a modulo. This function defines classes of states for which cpu_B returns the same value. We compute a histogram of these classes on each processor, which summarises how cpu_R would dispatch the states. This information is then globally exchanged, yielding a global histogram that is exploited to compute on each processor a better dispatching of the states it has to send. This is made by placing the classes according to a simple heuristic for the bin packing problem: the largest class is placed onto the less charged processor, which is repeated until all the classes have been placed. It is worth noting that this placement is computed with respect to the global histogram, but then, each processor dispatches only the states it actually holds, using this global placement. Moreover, if several processors compute a same state, these identical states will be in the same class and so every processor that holds such states will send them to the same target. So there is no possibility of duplicated computation because of dynamic states remapping.

Algorithm 5 Balancing strategy

$Exchange(tosend, known)$:

1: **dump**$(known)$
2: **return BSP_EXCHANGE**$(Balance(tosend))$

$Balance(tosend)$:

1: $histoL \leftarrow \{(i, \sharp\{(i,s) \in tosend\})\}$
2: compute $histoG$ from **BSP_MULTICAST**$(histoL)$
3: **return** $BinPack(tosend, histoG)$

The rest is as in Algorithm 4, using cpu_B instead of cpu_R.

These operations are detailed in Algorithm 5 where variables $histoL$ and $histoG$ store respectively the local and global histograms, and function $BinPack$ implements the dispatching method described above. In function $Balance$, $\sharp X$ denotes the cardinality of set X. Function **BSP_MULTICAST** is used so that each processor sends its local histogram to every processor and receives in turn their histograms, allowing to build the global one. Like any BSP communication primitive it involves a synchronisation barrier.

It may be remarked that the global histogram is not fully accurate since several processors may have a same state to be sent. Nor the computed dispatching is optimal since we do not want to solve a NP-hard bin packing problem. But, as shown in our benchmarks below, the result is yet fully satisfactory.

Finally, it is worth noting that if a state found in a previous super-step may be computed again, it would be necessary to known which processor owns it: this could not be obtained efficiently when dynamic remapping is used. But that could not happen thanks to the exploration order enforced in Section 3.2 and discussed in Section 3.3. Our dynamic states remapping is thus correct because states classes match the locality of computation.

4. Experimental results

In order to evaluate our algorithm, we have implemented a prototype version in Python, using SNAKES [18] for the Petri net part (which also allowed for a quick modelling of the protocols, including the inference rules of the Dolev-Yao attacker) and a Python BSP library [19] for the BSP routines (which is very close to an MPI "alltoall"). We actually used the MPI version (with MPICH) of the BSP-Python library. While largely suboptimal (Python programs are interpreted and there is no optimisation about the representation of the states in SNAKES), this prototype nevertheless allows and accurate *comparison* of the various algorithms.

With respect to the presented algorithms, our implementations differ only on technical details (*e.g.*, value $total$ returned by **BSP_EXCHANGE** is actually computed by exchanging also the number of values sent by each processor) and minor improvements (*e.g.*, we used in-place updating of sets and avoided multiple computations of $\mathsf{cpu}(s)$ using an intermediate variable).

The benchmarks presented below have been performed using a cluster with 20 PCs connected through a 1Gb Ethernet network. Each PC is equipped with a 2GHz Intel® Pentium® dual core CPU, with 2Gb of physical memory. This allowed to simulate a BSP computer with 40 processors equipped with 1Gb of memory each.

These experiments are designed to reveal how various aspects of the new method contribute to the overall performance. Our cases study involved the following four protocols:

1) Needham-Schroeder (NS) public key protocol for mutual authentication.
2) Yahalom (Y) key distribution and mutual authentication using a trusted third party.
3) Otway-Rees (OR) key sharing using a trusted third party.
4) Kao-Chow (KC) key distribution and authentication.

These protocols and their security issues are documented at the Security Protocols Open Repository (SPORE) [20].

For each protocol, we have built a modular model allowing for defining easily various scenarios involving different numbers of each kind of agents (but only one attacker, which is always enough).

4.1. Global performances

Figure 2 shows the execution times for two scenarios for each protocol; the depicted results are fair witnesses of what we could observe from the large number of scenarios we have actually run. In the figure, we have distinguished: the computation time that essentially corresponds to the computation of successor states on each processor; the communication time that corresponds to states exchange; the waiting times that occur when processors are forced to wait the others before to enter the communication phase of each super-step.

We can see on these graphs that the overall performance of our last algorithm (right-most bars) is always very good compared to the naive algorithm (left-most bars). In particular, the communication and waiting times are always greatly reduced. This holds for large state spaces as well as for smaller ones.

An important waiting time corresponds to an unbalanced computation: if some processors spend more time computing successors, the others will have to wait for them to finish this computation before every processor enters the communication phase. In several occurrences, we can observe that, by increasing the local computation, we have worsen the balance, which increased the waiting time. This corresponds to graphs where the middle part in the second column is taller than the same part in the left column. However, we can observe that our last optimisation to improve the balance is always very efficient and results in negligible waiting time in every case. The variations of observed computation times are similarly caused by a bad balance because we depicted the maximum accumulated time among the processors.

Finally, by comparing the left and right columns of results, we can observe that the overall speedup is generally better when larger state spaces are computed. This is mainly due to the fact that the waiting time accumulation becomes more important on longer runs.

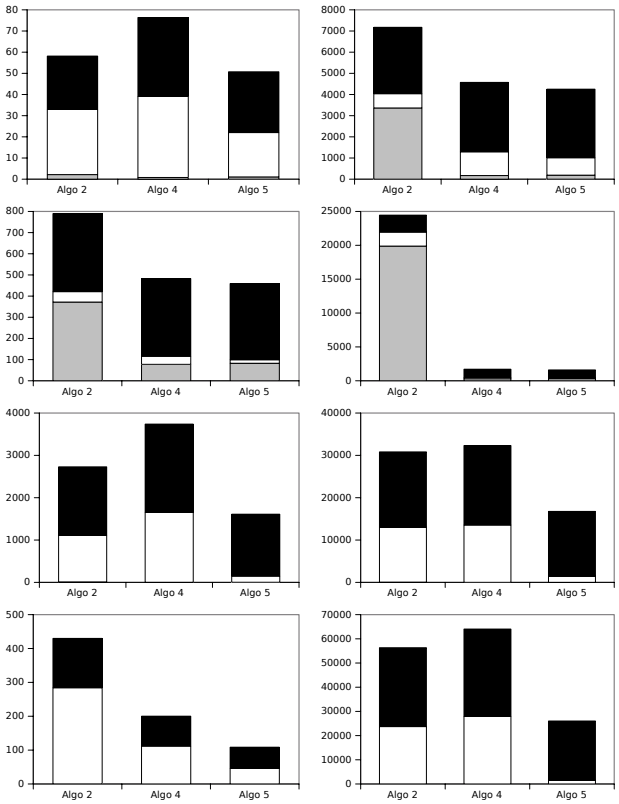

Fig. 2. Computation times (in seconds) of Algorithms 2, 4 and 5 for the four studied protocols. Top row: two instances of NS yielding respectively about 8K (left) and 5M states (right). Second row: two instances of Y with about 400K (left) and 1M states (right). Third row: two instances of OR with about 12K (left) and 22K states (right). Bottom row: two instances of KC with about 400 (left) and 2K states (right). Each bar show the maximums among processors of the accumulated computation times (black), waiting times (white) and communication time (gray).

4.2. Memory consumption

By measuring the memory consumption of our various algorithms, we could confirm the benefits of our sweep-line implementation when large state spaces are computed. For instance, in the NS scenario with 5M states, we observed an improvement of the peak memory usage from 97% to 40% (maximum among all the processors). Similarly, for the Y scenario with 1M states, the peak decreases from 97% to 60% (states in Y use more memory that states in NS). We also observed, on very large state spaces, that the naive implementation exhausts all the available memory and some processors start to use the swap, which causes a huge performance drop. This never happened using our

sweep-line implementation. However, notice that, in all the presented scenarios, no swapping has occurred, which would have dramatically biased the results.

4.3. Scalability

As a last observation about our algorithm, we would like to emphasise that we observed a linear speedup with respect to the number of processors. In general, most parallel algorithms suffer from an amortised speedup when the number of processors increases. This is almost always caused by the increasing amount of communication that becomes dominant over the computation. Because our algorithm is specifically dedicated to reduce the number of cross transitions, and thus the amount of communication, this problem is largely alleviated and we could observe amortised speedup only for very small models (less than 100 states) for which the degree of intrinsic parallelism is very reduced but whose state space is in any way computed very quickly.

5. Related works

Distributed state space construction has been studied in various contexts. All these approaches share a common idea: each machine in the network explores a subset of the state space. This procedure continues until the entire state space is generated and so no messages are sent anymore [4]. To detect this situation a termination detection procedure is usually employed. However, they differ on a number of design principles and implementation choices such as: the way of partitioning the state space using either static hash functions or dynamic ones that allow dynamic load balancing, *etc.* In this section, we focus on some of these technics and discuss their problems and advantages. More references can be found in [5].

In [21], a distributed state space exploration algorithm derived from the Spin model-checker is implemented using a master/slave model of computation. Several Spin-specific partition functions are experimented, the most advantageous one being a function that takes into account only a fraction of the state vector. The algorithm performs well on homogeneous networks of machines, but it does not outperform the standard implementation except for problems that do not fit into the main memory of a single machine. Moreover, no clue is provided about how to correctly choose the fraction of states to consider for hashing.

In [6] various technics from the literature are extended in order to avoid sending a state away from the current processor if its 2nd-generation successors are local. This is complemented with a mechanism that prevents re-sending already sent states. The idea is to compute the missing states when they become necessary for model-checking, which can be faster than sending it. That clearly improves communications but our method achieves similar goals, in a much simpler way, without ignoring any state.

There also exist approaches, such as [22], in which parallelization is applied to "partial verification", *i.e.* state enumeration in which some states can be omitted with a low probability. In our project, we only address exact, exhaustive verification issues. For completeness, we can also mention an alternative approach [23] in which symbolic reachability analysis is distributed over a network of workstations: this approach does not handle states individually, but sets of states encoded using BDDs.

For the partition function, different technics have been used. In [4] authors used of a prime number of virtual processors and map them to real processor. This improves load balancing but has no real impact on cross transitions. In [24], the partition function is computed by a round-robin on the successor states. This improves the locality of the computations but can duplicate states. Moreover, it works well only when network communication is substantially slower than computation, which is not the case on modern architectures for explicit model-checking. In [25], an user defined abstract interpretation is used to reduce the size of the state space and then it allows to distribute the abstract graph; the concrete graphs is then computed in parallel for each part of the distributed abstract graph. In contrast, our distribution method is fully automated and does not require input from the user.

There are many tools dedicated to the modelling and verification of security protocols as [26], [9], [10], the most well known is certainly AVISPA [12]. In contrast, our approach is based on a modelling framework (algebras of Petri nets) with explicit state space construction, that is not tight to any particular application domain. Our approach however, relies on the particular structure of security protocols. We believe that our observations and the subsequent optimisations are general enough to be adapted to the tools dedicated to protocol verification: we worked in a very general setting of LTS, defined by an initial state and a successor function. Our only requirements are three simple conditions (P1 ro P3) which can be easily fulfilled within most concrete modelling formalisms.

6. Conclusion and future works

The critical problem of state space construction is determining whether a newly generated state has been explored before. In a serial implementation this question is answered by organizing known states in a specific data-structure, and looking for the new states in that structure. As this is a centralized activity, any parallel or distributed solution must find an alternative approach. The common method is to assign states to processors using a static partition function which is generally a hashing of the states [4]. After a state has been generated, it is sent to its assigned location, where a local search determines whether the state already exists. This lead to two main difficulties. First the number of cross transitions is too high, leading to a too heavy network use. Second, memorising all the states in the main memory is impossible without crashing the whole computation and is not clear when it is possible to dump some states in disk and if heuristics like those in [21], [2] would work well for complex protocols.

Our first solution is to use the well-structured nature of security protocols to choose which part of the state is really needed for the partition function and to empty the data-structure in each super-step of the parallel computation. Our second solution entails automated classification of states and dynamic mapping of classes to processors. We find that both our methods execute significantly faster and achieve better network use than a classical method. Furthermore, we find that our method to balance states does indeed achieve better network use, memory balance and runs faster.

The fundamental message is that for parallel discrete state space construction, it is essential to exploit characteristics of the models and to structure the computation accordingly. We have demonstrated techniques that prove the feasibility of this approach and demonstrate its potential. Key elements to our success were (1) an automated states classification that reduces cross transitions and memory footstep, while improving the locality of computation (2) using global barriers (which is a low-overhead method) to compute a global remapping of states and thus improve balancing workload, achieving a good scalability.

Future works will be dedicated to build a real and efficient implementation from our prototype. It will feature in particular a temporal logic model-checker, allowing to verify more than reachability properties. Using this implementation, we would like to run benchmarks in order to compare our approach with existing tools. We would like also to test our algorithm on parallel computer with more processors in order to confirm the scalability that we could observe on 40 processors.

Moreover, we are working on the formal proof of our algorithm. Proving a verification algorithm is highly desirable in order to certify the truth of the diagnostics delivered by such an algorithm. Such a proof is possible because, thanks to the BSP model, our algorithm remains simple in its structure. Finally, we would like to generalise our present results by extending the application domain. In the security domain, we will consider more complex protocols with branching and looping structures, as well as complex data types manipulations. In particular, we will consider protocols for secure storage distributed through peer-to-peer communication [27]. Another generalisation will be to consider symbolic state space representations, in particular those based on decision diagrams.

References

[1] D. Nicol and G. Ciardo, "Automated parallelization of discrete state-space generation," *Journal of Parallel and Distributed Computing*, vol. 4, no. 2, pp. 153–167, 1997.

[2] S. Evangelista and L. M. Kristensen, "Dynamic State Space Partitioning for External Memory Model Checking," in *Proceedings of Formal Methods In Computer Sciences (FMICS)*, ser. LNCS, vol. 5825. Springer, 2009, pp. 70–85.

[3] R. Kumar and E. G. Mercer, "Load balancing parallel explicit state model checking," in *ENTCS*, vol. 128. Elsevier, 2005, pp. 19–34.

[4] H. Garavel, R. Mateescu, and I. Smarandache, "Parallel state space construction for model-checking," in *Workshop on Model Checking of Software SPIN*, May 2001.

[5] J. Barnat, "Distributed memory LTL model checking," Ph.D. dissertation, Faculty of Informatics Masaryk University Brno, 2004.

[6] C. Pajault, "Model checking paralléle et réparti de réseaux de Petri colorés de haut-niveau," Ph.D. dissertation, Conservatoire National des Arts et Métiers, 2008.

[7] D. Basin, "How to evaluate the security of real-life cryptographic protocols? the cases of ISO/IEC 29128 and CRYPTREC," in *Workshop on Real-life Cryptographic Protocols and Standardization*, 2010.

[8] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983.

[9] A. Armando and L. Compagna, "SAT-based model-checking for security protocols analysis," *Int. J. Inf. Sec.*, vol. 7, no. 1, pp. 3–32, 2008.

[10] A. Armando, R. Carbone, and L. Compagna, "LTL model checking for security protocols," in *Proceedings of CSF*. IEEE Computer Society, 2007, pp. 385–396.

[11] H. Gao, "Analysis of security protocols by annotations," Ph.D. dissertation, Technical University of Denmark, 2008.

[12] A. Armando and al., "The AVISPA tool for the automated validation of Internet security protocols and applications," in *Proceedings of Computer Aided Verification (CAV)*, ser. LNCS, K. Etessami and S. K. Rajamani, Eds., vol. 3576. Springer, 2005, pp. 281–285.

[13] C. J. F. Cremers, "Scyther - semantics and verification of security protocols," Ph.D. dissertation, Technische Universiteit Eindhoven, 2006.

[14] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl, "Questions and Answers about BSP," *Scientific Programming*, vol. 6, no. 3, pp. 249–274, 1997.

[15] R. H. Bisseling, *Parallel Scientific Computation. A structured approach using BSP and MPI*. Oxford University Press, 2004.

[16] F. Pommereau, "Algebras of coloured Petri nets," Habilitation thesis, University Paris-East Créteil, 2009.

[17] S. Christensen, L. M. Kristensen, and T. Mailund, "A sweep-line method for state space exploration," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, T. Margaria and W. Yi, Eds., vol. 2031. Springer, 2001, pp. 450–464.

[18] F. Pommereau, "Quickly prototyping Petri nets tools with SNAKES," ser. ACM Digital Library. ACM, 2008, pp. 1–10.

[19] K. Hinsen, "Parallel scripting with Python," *Computing in Science & Engineering*, vol. 9, no. 6, 2007.

[20] E. C. LSV, "SPORE: Security protocols open repository," http://www.lsv.ens-cachan.fr/Software/spore.

[21] F. Lerda and R. Sista, "Distributed-memory model checking with SPIN," in *Proceedings of SPIN*, ser. LNCS, D. Dams, R. Gerth, S. Leue, and M. Massink, Eds., no. 1680. Springer-Verlag, 1999, pp. 22–39.

[22] W. J. Knottenbelt, M. A. Mestern, P. G. Harrison, and P. Kritzinger, "Probability, parallelism and the state space exploration problem," in *Proceedings of Computer Performance Evaluation-Modeling, Techniques and Tools (TOOLS)*, ser. LNCS, R. Puigjaner, N. N. Savino, and B. Serra, Eds., no. 1469. Springer-Verlag, 1998, pp. 165–179.

[23] T. Heyman, D. Geist, O. Grumberg, and A. Schuster, "Achieving scalability in parallel reachability analysis of very large circuits," in *Proceedings of Computer Aided Verification (CAV)*, ser. LNCS, no. 1855, 2000, pp. 20–35.

[24] D. Petcu, "Parallel explicit state reachability analysis and state space construction," in *Proceedings of ISPDC*. IEEE Computer Society, 2003, pp. 207–214.

[25] S. Orzan, J. van de Pol, and M. Espada, "A state space distributed polivy based on abstract interpretation," in *ENTCS*, vol. 128. Elsevier, 2005, pp. 35–45.

[26] T. Genet, Y.-M. Tang-Talpin, and V. V. T. Tong, "Verification of copy-protection cryptographic protocol using approximations of term rewriting systems," in *Workshop on Issues in the Theory of Security (WITS)*, 2003.

[27] S. Sanjabi and F. Pommereau, "Modelling, verification, and formal analysis of security properties in a P2P system," in *Workshop on Collaboration and Security (COLSEC'10)*, ser. IEEE Digital Library. IEEE, 2010, pp. 543–548.