J Supercomput
DOI 10.1007/s11227-014-1099-8

A BSP algorithm for on-the-fly checking CTL* formulas on security protocols

Frédéric Gava · Franck Pommereau · Michaël Guedj

Abstract This paper presents a distributed (Bulk-Synchronous Parallel or BSP) algorithm to compute on-the-fly whether a structured model of a security protocol satisfies a CTL* formula. Using the structured nature of the security protocols allows us to design a simple method to distribute the state space under consideration in a need-driven fashion. Based on this distribution of the states, the algorithm for logical checking of a LTL formula can be simplified and optimised allowing, with few tricky modifications, the design of an efficient algorithm for CTL* checking. Some prototype implementations have been developed, allowing to run benchmarks to investigate the parallel behaviour of our algorithms.

Keywords BSP · LTL · CTL* · Security protocols · State-space · Model-checking

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* such as confidentiality of data, authentication of participants, etc. This leads to search for a way to verify whether a protocol is secure or not [16].

Electronic supplementary material The online version of this article (doi:10.1007/s11227-014-1099-8) contains supplementary material, which is available to authorized users.

F. Gava (✉) · M. Guedj
LACL, University of Paris-East, Créteil, France
e-mail: gava@u-pec.fr

F. Pommereau
IBISC, University of Évry, Évry, France
e-mail: franck.pommereau@ibisc.univ-evry.fr

Published online: 12 February 2014

But designing and verifying secure protocols is a challenging problem. In spite of their apparent simplicity, they are notoriously error-prone. *Attacks* exploit *weaknesses* in the protocol that are due to the complex and unexpected *interleaving* of different protocol sessions generated by a malicious *intruder* which *resides in the network*. A famous example is the "man-the-middle" attack on the Needham–Schroeder public key protocol. The intruder is assumed to have a *complete control on the network* and to be powerful enough to perform potentially *dangerous actions* such as intercepting messages flowing over the network, or replacing them by new ones using the knowledge he has previously gained [20].

1.1 Model checking security protocols

Unfortunately, the question of whether a protocol achieves its security requirements or not is, in the general case, *undecidable* or *NP-complete* in the case of a bounded number of agents [6]. Even if security protocols should theoretically be checked under an unbounded number of concurrent protocol executions, violating their security requirements often exploits only a small number of sessions (that is, an execution of an instance of the protocol) and agents. For these reasons, it is in many cases of interest sufficient to consider a finite number of sessions (in which each agent performs a fixed number of steps) in order to *find flaws* (which is distinct from proving the protocol). *Formal methods* offer a promising approach for *automated* security analysis of protocols: the intuitive notions are translated into formal *specifications*, which are essential for a careful design and analysis. The development of formal techniques that can check various security properties is an important tool to meet this challenge [16]. *Enumerative (explicit) model checking* is well adapted to find flaws in this kind of asynchronous, non-deterministic systems [2,6]. In particular, when an execution of the protocol is discovered to violate a security property, it can be presented as a trace of the protocol execution, that is, an explicit attack scenario.

By focusing on the verification of a *bounded number of sessions*, model checking a protocol can be done by simply enumerating and *exploring all traces* of the execution of the protocol and looking for a violation of some of the requirements. Verification through model checking consists in defining a formal model of the system to be analysed and then using automated tools to check whether the expected properties (generally expressed in a temporal logic) are met or not on the *state space* of the model. To do so, all the different configurations of the execution of the agents evolving in the protocol need to be computed [6].

In this paper, we consider the problem of checking in a *distributed* way formulas expressed in the temporal logic CTL* over *labelled transition systems* (LTS) that model security protocols. Checking a logical formula over a protocol is not new [1,6] and has the advantage over dedicated tools for protocols (such as PROVERIF [9] or SCYTHER [18] to cite the most known) to be easily extensible to non standard behaviour of honest principals (e.g. contract-signing protocols in which participants are required to make progress toward an agreement) or to check some security goals that *cannot* be expressed as *reachability properties*, e.g. *fair exchange* [6].

1.2 Distributed model checking: problematic and contribution

But the greatest problem with explicit model checking in general (and for security protocols in particular) is the so-called *state explosion*: the fact that the number of states typically grows *exponentially* with the number of agents and sessions. This is especially true when complex data-structures are used in the model such as the knowledge of an intruder in a security protocol. Checking a CTL* formula over a security protocol may thus be *expensive* both in terms of *memory* and *execution time*.

Because explicit model checking can cause memory crashing on single or multiple processor systems, it has led to consider exploiting the larger memory space available in distributed systems [24], which also gives the opportunity to reduce the overall execution time. *Parallelising* the state-space construction on several machines is thus done to benefit from each machine's complete storage and computing resources. One of the main technical issues is to partition the state space, i.e. each subset of states is "owned" by a single machine.

To have efficient parallel algorithms for this state-space construction, it is common to have the following requirements. First, how states are partitioned across the processors must be computed quickly. Second, the *successor function* (of a state) must be defined so that successors states are likely mapped to the same processor as its predecessor; otherwise the computation will be overwhelmed by interprocessor communications (the so-called *cross transitions*) which obviously implies a drop of the computation locality and thus of the performances. Third, *balancing the workload* is obviously needed [33] in order to fully profit from available computational power and 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 state space are unknown and cannot be predicted in general. Moreover, during the state-space construction, all the explored states may need to be kept in memory to avoid multiple exploration of a same state. This can lead to fill the main memories and induce *swapping* which is known to significantly slow machines.

Furthermore, one may identify two basic approaches to model checking. The first one uses a global analysis to determine if a system satisfies or not a formula; the entire state space of the system is constructed and subjected latter to analysis. However, these algorithms may be used to perform *unnecessary* work because in many cases (especially when a system does not satisfy a specification), only a subset of the system states needs to be analysed to determine whether the system satisfies a formula or not. It is thus rarely necessary to compute the entire state space before finding a path that invalidates the logic formula (a flaw in a protocol). On the other hand, *on-the-fly* (or local) approaches to model checking attempt to take advantage of this observation by constructing the state space in a demand-driven fashion: on-the-fly algorithms are designed to build the state space and *check* the formula at the same time which is thus generally more efficient.

By exploiting the *well-structured* nature of security protocols, we propose a solution to simplify the writing of an efficient on-the-fly model checking distributed algorithm for finite scenarios. The structure of the protocols is exploited to *partition* the state space, to reduce cross transitions while increasing computation locality, to keep only a sub-part of the state space in the main memories (to avoid swapping on external/disk

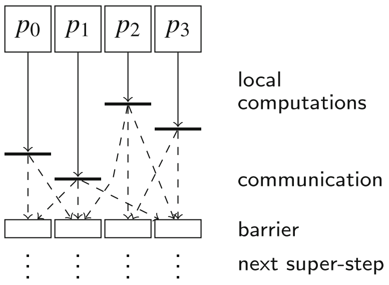


Fig. 1 A BSP super-step

memories) and to load balance the computations. At the same time, the BSP model of computation [8] (defined later in this paper) allows us to simplify the detection of the algorithm *termination* and to further load balance the computations.

Our work is based on the sequential algorithm of [7] which mainly combines the construction a *proof-structure* (a graph) together with a Tarjan's depth-first-search based Strongly Connected Components (SCC) algorithm for detecting on-the-fly a reachable accepting cycle in the underlying graph.

1.3 Outline

First, we briefly review in Sect. 2 the context of our work that is the BSP model, models of security protocols and their state-space representation as LTS, as well as the formal definition of two temporal logics LTL and CTL* together with their verification.

Section 3 is dedicated to the description of our new state-space algorithm constructed in a *step-wise* manner from a sequential one. Section 4 is dedicated to the design of a BSP algorithm for verification of a LTL formula on a security protocol and Sect. 5 is the generalisation of the above algorithm for CTL*. For all the algorithms, we briefly describe a prototype implementation and apply it to some typical protocol sessions, giving benchmarks to demonstrate the benefits of our approach.

Finally, related works are discussed in Sect. 6, while a conclusion and future works are presented in Sect. 7.

2 Context and general definitions

2.1 The BSP model of parallel execution

A BSP computer is seen as a set of *uniform* processor-memory pairs connected through a *communication network* allowing the inter-processor delivery of messages [8]. Clusters of PCs, multi-core, etc. can be considered as BSP computers.

A BSP program is logically executed as a sequence of *super-steps* (see Fig. 1), each of which is divided into three successive disjoint phases: (1) Each processor only uses its local data to perform sequential computations and to request data transfers to other nodes; (2) The network delivers the requested data; (3) 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.

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 classical 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.

This *structured* model of parallelism enforces a strict separation of communication and computation: during a super-step, no communication between the processors is allowed but only transfer requests, only at the synchronisation barrier information is actually exchanged. However, for better performances, a BSP library can send messages during the computation phase of a super-step, but this is hidden to programmers. On most 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, this execution policy has the main advantage that it removes non-determinism and guarantees the absence of deadlocks since barriers do not create circular data dependencies. This is also merely the most visible aspects of a parallel model that shifts the responsibility for timing and synchronisation issues from the applications to the communications library. This can be used at runtime to dynamically make decisions, for instance choose whether to communicate to re-balance data, or to continue an unbalanced computation. BSP libraries are generally implemented using MPI or low-level routines of the given specific architectures.

2.2 Security protocols and their state space

2.2.1 Brief overview of the security protocols

Security protocols[1] specify an exchange of *cryptographic messages* between *principals*, i.e. the agents (users, hosts, servers, etc.) participating in the protocol. Each instance of the protocol is called a session and an agent can participate to more than one session, sequentially or concurrently. A scenario is a particular choice of arrangement for different sessions involving a particular choice of agents. Messages are sent over open *networks*, such as the Internet, that are not secured. As a consequence, protocols should be designed to work fine even if messages may be eavesdropped or tampered with by an *intruder*—e.g. a dishonest or careless agent. Finally, each protocol is aimed to provide security guarantees such as *authentication* of principals or *secrecy* of some pieces of information (e.g. a *key*, a value that can crypt/decrypt a message or a *nonce*, a value that is new for each session) or *non-repudiation* and *fairness* for commercial protocol with a contract.

[1] More details on their modelling, semantics and attacks can be found in [6,16].

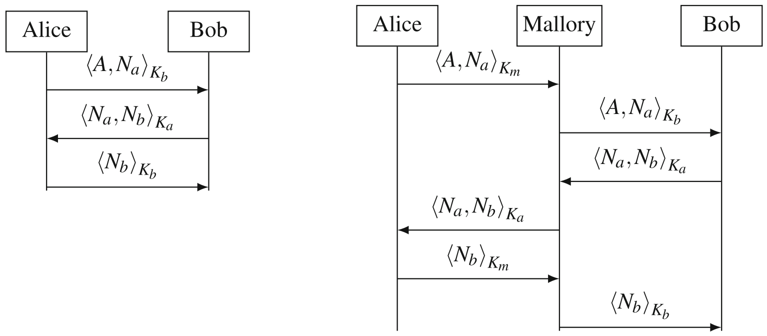

Alice
Bob
$\langle A, N_a \rangle_{K_b}$
$\langle N_a, N_b \rangle_{K_a}$
$\langle N_b \rangle_{K_b}$
Alice
Mallory
Bob
$\langle A, N_a \rangle_{K_m}$
$\langle A, N_a \rangle_{K_b}$
$\langle N_a, N_b \rangle_{K_a}$
$\langle N_a, N_b \rangle_{K_a}$
$\langle N_b \rangle_{K_m}$
$\langle N_b \rangle_{K_b}$

Fig. 2 The NS protocol (*left*) and the well-known "man-in-the-middle" attack (*right*)

Agents perform "ping-pong" data exchanges and some well-known strategies that an intruder might employ are: *man-in-the-middle*, the intruder imposing itself in the communications between the sender and receiver; *replay*, the intruder monitors a run of the protocol and at some later time replays one or more of the messages; etc.

We assume the use of keys sufficiently long of the best-known cryptographic algorithms to prevent a brute force attack in a feasible time. This is the well-known *perfect cryptography* assumption. The idea is that an encrypted message can be decrypted only using the appropriate decryption key, i.e. it is possible to retrieve M from the message encrypted $\{M\}_K$ only using K^{-1} as decryption key and it is hopeless to compute K^{-1} from K or to guess one of these keys.

Figure 2 (left) illustrates the standard Needham–Schroeder (NS) protocol which involves two agents Alice (A) and Bob (B) who want to mutually authenticate—N_a and N_b are nonces and K_a, K_b are the public keys of, respectively, Alice and Bob. The idea of the protocol is that each agent sends a challenge to the other agents. The challenge is under the form of a nonce (unguessable by nature) encrypted with the receiver's public key. The receiver is thus the only one able to decrypt the nonce and sends it back. In the right of Fig. 2, we show the well-known *flaw* of the NS protocol when initiated with a malicious third party Mallory (M); it involves two parallel sessions, with M participating in both of them; Mallory authenticates as Alice with Bob. This is called a *logical attack* because both sessions of the protocol are correct, but the overall security goals are not achieved.

In this paper, we thus consider that a Dolev/Yao attacker [20] 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 who 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 informations (i.e. attacker's knowledge). 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. The Dolev/Yao threat model is a worst-case model in the sense that the network, over which the participants communicate, is thought as being totally controlled by an omnipotent intruder. Therefore, there is no need to assume the existence of multiple attackers, because they together do not have more abilities than the single Dolev/Yao intruder.

Model checking attempts to find a reachable state or trace where a security property fails—e.g. secret term is learnt by the intruder or an incorrect authentication occurs. To ensure termination, these tools usually bound the maximum number of sessions. To model check a security protocol, one must construct its state space (all executions of the sessions) and check the property usually expressed in a temporal logic. We now formally define these steps.

2.2.2 State-space construction

The state-space construction problem is the problem of computing the explicit representation of a given model from the implicit one. In most cases, this space is constructed by exploring all the states reachable through a successor function from an initial state. The state space of a protocol thus includes all the executions of the sessions considering all the messages built by the intruder. The state-space (noted S) construction consists in constructing a LTS:

Definition 1 (*Labelled Transition System*, LTS) It is a tuple (S, T, L) where S is the set of states, $T \subseteq S \times S$ is the set of transitions, and L is an arbitrary labelling on $S \cup T$.

Given a model implicitly defined by its initial state s_0 and its successor function succ, the corresponding explicit LTS (s_0, succ) is 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. Now, assuming a set $\mathscr{A}$ of atomic propositions, we have:

Definition 2 (*Kripke structure*) A Kripke structure is a LTS (S, T, L) whose labelling is $L : S \to 2^{\mathscr{A}}$.

Mainly a Kripke structure is a LTS adjoining whose labelling function associates truth values to the states.

Definition 3 (*Path and related notions*) Let $M \overset{\text{df}}{=} (S, T, L)$ be a Kripke structure.

1. A path in M is a maximal sequence of states $\langle s_0, s_1, \ldots \rangle$ such that for all $i \geq 0$, $(s_i, s_{i+1}) \in T$.
2. If $x = \langle s_0, s_1, \ldots \rangle$ is a path in M, then $x(i) \overset{\text{df}}{=} s_i$ and $x^i \overset{\text{df}}{=} \langle s_i, s_{i+1}, \ldots \rangle$.
3. If $s \in S$ then $\Pi_M(s)$ is the set of paths x in M such that $x(0) = s$.

2.2.3 Properties of the state spaces of security protocols

In this paper, we model security protocols as LTS such that any state can be represented by a function from a set of *locations* to an arbitrary data domain. For instance, locations may correspond to local variables of agents, buffers, etc.

As a concrete formalism to model protocols, we have used an *algebra of coloured Petri nets* called ABCD [42] (not presented in this paper) allowing easy and structured modelling. This algebra is part of the SNAKES library [42] which is a general Petri net library that allows to model and execute PYTHON-coloured Petri nets: tokens are PYTHON objects and net inscriptions are PYTHON expressions. We refer to [26] for more details and examples of models of security protocols using ABCD.

However, our approach is largely independent of the chosen formalism and it is enough to assume that the following properties (P1) to (P4) hold.

(P1) locations can be partitioned into two sets $\mathscr{R}$ and $\mathscr{L}$, and LTS function succ can be partitioned into two functions $\mathsf{succ}_{\mathscr{R}}$ and $\mathsf{succ}_{\mathscr{L}}$ such that: for all state s and all $s' \in \mathsf{succ}(s)$, denoting by $s|_{\mathscr{R}}$ the state s restricted to the locations from $\mathscr{R}$, we have that $s'|_{\mathscr{R}} = s|_{\mathscr{R}} \implies s' \in \mathsf{succ}_{\mathscr{L}}(s)$, and $s'|_{\mathscr{R}} \neq s|_{\mathscr{R}} \implies s' \in \mathsf{succ}_{\mathscr{R}}(s)$. Intuitively, $\mathsf{succ}_{\mathscr{R}}$ corresponds to transitions upon which an agent (except the attacker) receives information and stores it, and $\mathscr{R}$ are the locations where these agents store the information they receive.

(P2) there is an initial state s_0 and there exists a function slice from states to natural numbers (a *measure*) such that if $s' \in \mathsf{succ}_{\mathscr{R}}(s)$, then there is no path from s' to any state s'' such that $\mathsf{slice}(s) \leq \mathsf{slice}(s'')$ and $\mathsf{slice}(s') = \mathsf{slice}(s)+1$. This is often called a *sweep-line* progression and corresponds to the fact that agents perform irreversible actions. In particular, a reception by an agent corresponds to an irreversible step in the sequence of messages forming the protocol.

(P3) there exists also a hash function $\mathsf{cpu}_{\mathscr{R}}$ from states to natural numbers such that for all state s if $s' \in \mathsf{succ}_{\mathscr{L}}(s)$, then $\mathsf{cpu}_{\mathscr{R}}(s) = \mathsf{cpu}_{\mathscr{R}}(s')$. This is to say that only information in $\mathscr{R}$ is taken into account to compute the hash of a state, and in particular, the knowledge of the intruder is not involved.

(P4) if $s_1, s_2 \in \mathsf{succ}_{\mathscr{R}}(s)$ and $\mathsf{cpu}_{\mathscr{R}}(s_1) \neq \mathsf{cpu}_{\mathscr{R}}(s_2)$, then there is no possible path from s_1 to s_2 and vice versa. This means that the receptions of two distinct messages lead to distinct executions of the protocol, which is the case for instance when agents permanently store the information they have received (this always holds in practice).

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 and, for virtually all models of classical protocols protocol, it is also easy to check that the above properties are satisfied. This is the case in particular for us using the ABCD formalism. However, protocols involving potentially unbounded loops (i.e. while loops) in the behaviour of agents cannot usually be modelled so that (P2) and (P4) hold. Fortunately, such protocols are actually rare, but considering them is one of our perspectives.

Note that our approach is compatible with the use of partial order reductions as in [23], where the main idea is that the knowledge of the intruder *always grows* and thus it is safe to *prioritise* the sending transitions with respect to receptions and local computations of agents. A simple modification of the successors functions is sufficient to achieve this.

2.3 Proof structure and temporal logical checking

Many security properties such as secrecy (confidentiality), authentication, integrity, anonymity can usually be expressed only using a state-space computation since these properties only force to a reachability analysis, i.e. finding a single state that breaks one of the above properties. However, more complex property may involve distinguishing several steps in an execution and thus require to resort to temporal logics.

2.3.1 Temporal logics

Temporal logics have mainly two kinds of operators: *logical* operators and *modal* operators. Logical operators are the usual operators such as $\wedge$, $\vee$, etc. Modal operators are used to reason about time such as "until", "next-time", etc. Quantifiers can also be used to reason about paths e.g. "a formula holds on all paths starting from the current state". In LTL, one can encode formulae about the *future of paths*, e.g. a condition will eventually be true, a condition will be true until another fact becomes true, etc. CTL is a branching-time logic, which means that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be an actual path that is realised.

We now give the formal definition of CTL*, that subsumes both LTL and CTL. Without loss of generality, we assume that relation T is total and thus all paths in M are infinite. This is only a convenience to define the algorithms, but may be easily removed. We fix a set $\mathscr{A}$ of atomic propositions, which will be ranged over by $a, a', \ldots$. We sometimes call *literals* formulas of the form a or $\neg a$; the set of all literals will be ranged over by $l, l_1, \ldots$ We use $p, p_1, q, \ldots,$ to range over the set of state formulas and $\phi, \phi_1, \gamma, \ldots,$ to range over the set of path formulas—both formally defined in the following. We also call *path quantifiers* **A** ("for all") and **E** ("exists"), and *path modalities* **X** ("next"), **U** ("until") and **R** ("release").

Definition 4 (*Syntax of* CTL*) The following grammar describes the syntax of CTL*:

$$\begin{aligned}\mathscr{S} &::= a \mid \neg a \mid \mathscr{S} \wedge \mathscr{S} \mid \mathscr{S} \vee \mathscr{S} \mid \mathbf{A}\mathscr{P} \mid \mathbf{E}\mathscr{P}\\ \mathscr{P} &::= \mathscr{S} \mid \mathscr{P} \wedge \mathscr{P} \mid \mathscr{P} \vee \mathscr{P} \mid \mathbf{X}\mathscr{P} \mid \mathscr{P}\mathbf{U}\mathscr{P} \mid \mathscr{P}\mathbf{R}\mathscr{P}\end{aligned}$$

We refer to the formulas generated from $\mathscr{S}$ as state formulas and those from $\mathscr{P}$ as path formulas. We define the CTL* formulas to be the set of state formulas.

Note that we use a particular construction on the formulas by putting the negation only adjoining to the atoms, which is a usual canonical form of CTL* formulas that is always possible to obtain. CTL consists of those CTL* formula in which every occurrence of a path modality is immediately preceded by a path quantifier and LTL are CTL* formula of the form $\mathbf{A}\phi$, where the only state sub-formula of ϕ are literals.

Definition 5 (*Semantic of* CTL*) Let $M = (S, R, L)$ be a Kripke structure with $s \in S$ and x a path in M. Then, the satisfaction relation $\models$ is defined inductively as follows:

- $s \models a$ if $a \in L(s)$ (recall $a \in \mathscr{A}$);
- $s \models \neg a$ if $s \not\models a$;
- $s \models p_1 \wedge p_2$ if $s \models p_1$ and $s \models p_2$;
- $s \models p_1 \vee p_2$ if $s \models p_1$ or $s \models p_2$;
- $s \models \mathbf{A}\phi$ if for every $x \in \Pi_M(s)$, $x \models \phi$;

$$\frac{s \vdash \mathbf{A}(\Phi,\phi)}{true}\,(R1)\ \text{if } s \models \phi \qquad \frac{s \vdash \mathbf{A}(\Phi,\phi)}{s \vdash \mathbf{A}(\Phi)}\,(R2)\ \text{if } s \not\models \phi \qquad \frac{s \vdash \mathbf{A}(\Phi,\phi_1 \vee \phi_2)}{s \vdash \mathbf{A}(\Phi,\phi_1,\phi_2)}\,(R3) \qquad \frac{s \vdash \mathbf{A}(\Phi,\phi_1 \wedge \phi_2)}{s \vdash \mathbf{A}(\Phi,\phi_1) \quad s \vdash \mathbf{A}(\Phi,\phi_2)}\,(R4)$$

$$\frac{s \vdash \mathbf{A}(\Phi,\phi_1\mathbf{U}\phi_2)}{s \vdash \mathbf{A}(\Phi,\phi_1,\phi_2) \quad s \vdash A(\Phi,\phi_2,\mathbf{X}(\phi_1\mathbf{U}\phi_2))}\,(R5) \qquad \frac{s \vdash \mathbf{A}(\Phi,\phi_1\mathbf{R}\phi_2)}{s \vdash \mathbf{A}(\Phi,\phi_2) \quad s \vdash \mathbf{A}(\Phi,\phi_1,\mathbf{X}(\phi_1\mathbf{R}\phi_2))}\,(R6)$$

$$\frac{s \vdash \mathbf{A}(\mathbf{X}\phi_1,\ldots,\mathbf{X}\phi_n)}{s_1 \vdash \mathbf{A}(\phi_1,\ldots,\phi_n) \quad s_m \vdash \mathbf{A}(\phi_1,\ldots,\phi_n)}\,(R7)\ \text{if } succ(s) = \{s_1,\ldots,s_m\}$$

Fig. 3 Proof rules for LTL checking [7]

- $s \models \mathbf{E}\phi$ if there exists $x \in \Pi_M(s)$ such that $x \models \phi$;
- $x \models p$ if $x(0) \models p$ (recall p is a state formula);
- $x \models p_1 \wedge p_2$ if $x \models p_1$ and $x \models p_2$;
- $x \models p_1 \vee p_2$ if $x \models p_1$ and $x \models p_2$;
- $x \models \mathbf{X}\phi$ if $x^1 \models \phi$;
- $x \models \phi_1\mathbf{U}\phi_2$ if there exists $i \geq 0$ such that $x^i \models \phi_2$ and for all $j < i$, $x^j \models \phi_1$;
- $x \models \phi_1\mathbf{R}\phi_2$ if for all $i \geq 0$, $x^i \models \phi_2$ or if there exists $i \geq 0$ such that $x^i \models \phi_1$ and for every $j \leq i$, $x^j \models \phi_2$.

The meaning of most of the constructs is straightforward. A state satisfies $\mathbf{A}\phi$ (resp. $\mathbf{E}\phi$) if every path (resp. some path) starting from the state satisfies ϕ, while a path satisfies a state formula if the initial state in the path does. $\mathbf{X}$ represents a "next-time" operator in the usual sense of "one transition forward", while $\phi_1\mathbf{U}\phi_2$ holds of a path if ϕ_1 remains true until ϕ_2 becomes true. The modal operator $\mathbf{R}$ may be thought of as a "release" operator: a path satisfies $\phi_1\mathbf{R}\phi_2$ if ϕ_2 remains true until both ϕ_1 and ϕ_2 (ϕ_1 releases the path from the obligations) or ϕ_2 is always true. For two examples of security properties:

1. Fairness is a CTL formula: $\mathbf{AG}(recv(c_1, d_2) \Rightarrow \mathbf{EF}\, recv(c_2, d_1))$ if we assume two agents c_1 and c_2 that possess items d_1 and d_2, respectively, and wish to exchange them; it asserts that if c_1 receives d_2, then c_2 has always a way to receive d_1.
2. The availability of an agent can be a LTL formula that requires that all the messages m received by this agent a will be processed eventually, which can be formalised as: $\mathbf{AG}(rcvd(a, m) \Rightarrow (\mathbf{F}\neg rcvd(a, m)))$

where the two syntaxic sugars are: (1) $\mathbf{G}(p)$ is for "globally" and is equal to $false\,\mathbf{R}\,p$; (2) $\mathbf{F}(p)$ is for "finally" and is equal to $true\ \mathbf{U}\ p$.

2.3.2 Checking an LTL formula

In [7], the authors give an efficient algorithm for model checking LTL then CTL* formula. The algorithm is based on a collection of top-down proof rules for inferring when a state in a Kripke structure satisfies a LTL formula. It is close to a Tableau method [25]. These rules are reproduced in Fig. 3, they work on assertions of the form $s \vdash \mathbf{A}\Phi$ where $s \in S$ and Φ is a set of path formula.

Semantically, $s \vdash \mathbf{A}\Phi$ holds if $s \models \mathbf{A}(\bigvee_{\phi \in \Phi} \phi)$. We write $\mathbf{A}(\Phi, \phi_1, \ldots, \phi_n)$ to represent $\mathbf{A}(\Phi \cup \{\phi_1, \ldots, \phi_n\})$ and we consider $\mathbf{A}(\emptyset) = \emptyset$. If σ is an assertion of the form $s \vdash \mathbf{A}\Phi$, then we use $\phi \in \sigma$ to denote that $\phi \in \Phi$. We may also drop $\mathbf{A}$ and write $s \vdash \Phi$ for an assertion if the context allows it.

Definition 6 (*Proof structure* [7]) Let Σ be a set of nodes, $\Sigma' \overset{\text{df}}{=} \Sigma \cup true$, $V \subseteq \Sigma'$, $E \subseteq V \times V$ and $\sigma \in V$. Then, $\langle V, E\rangle$ is a proof structure for σ if it is a maximal directed graph such that for every $\sigma' \in V$, σ' is reachable from σ, and the set $\{\sigma'' \mid (\sigma', \sigma'') \in E\}$ is the result of applying some rule to σ'.

Intuitively, a *proof structure* for σ is a *direct graph* that is intended to represent an (attempted) "proof" of σ. In what follows, we consider such a structure as a directed graph and use traditional graph notations for it. Note that in contrast with traditional definitions of proofs, proof structures may contain cycles. To define when a proof structure represents a valid proof of σ, we use the following notion:

Definition 7 (*Successful proof structure* [7]) Let $\langle V, E\rangle$ be a proof structure.

- $\sigma \in V$ is a leaf iff there is no σ' such that $(\sigma, \sigma') \in E$. σ is successful iff $\sigma \equiv true$.
- An infinite path $\pi = \langle \sigma_0, \sigma_1, \ldots\rangle$ in $\langle V, E\rangle$ is successful iff some assertion σ_i infinitely repeated in π satisfies the following: there exists $\phi_1\mathbf{R}\phi_2 \in \sigma_i$ such that for all $j \geq i$, $\phi_2 \notin \sigma_j$.
- $\langle V, E\rangle$ is partially successful iff every leaf is successful. $\langle V, E\rangle$ is successful iff it is partially successful and each of its infinite paths is successful.

Roughly speaking, an infinite path is successful if at some point a formula of the form $\phi_1\mathbf{R}\phi_2$ is repeatedly "regenerated" by application of rule R6, i.e. the right-hand sub-goal of this rule application appears each time on the path. Note that after $\phi_1\mathbf{R}\phi_2$ occurs on the path, ϕ_2 should not, because, intuitively, if ϕ_2 was true then the success of the path would not depend on $\phi_1\mathbf{R}\phi_2$, while if it was false then $\phi_1\mathbf{R}\phi_2$ would not hold. Note also that if no rule can be applied (i.e. $\Phi = \emptyset$) then the proof-structure is unsuccessful and thus the formula does not hold. We now have the following result:

Theorem 1 (Proof structure and LTL [7]) *Let M be a Kripke structure with $s \in S$ and $\mathbf{A}\phi$ an* LTL *formula, and let $\langle V, E\rangle$ be a proof structure for $s \vdash \mathbf{A}\{\phi\}$. Then $s \models \mathbf{A}\phi$ iff $\langle V, E\rangle$ is successful.*

One consequence of this theorem is that if σ has a successful proof structure, then all proof structures for σ are successful. Thus, it turns out that the success of a finite proof structure may be determined by looking at its *strongly connected components* (SCCs, we recall that an SCC of a directed graph is a maximal component in which every vertex can be reached from every other) or any accepting cycle. The efficient algorithm of [7] (described later) combines the construction of a proof structure with the process of checking whether the proof structure is successful using a Tarjan-like algorithm for SCC computation (and a recursive decomposition of a CTL* formula into several LTL formula) but a NDFS [28] one could be used equally.

Call a SCC $\mathscr{C}$ of $\langle V, E\rangle$ *nontrivial* if there exist (not necessary distinct) $v, v' \in \mathscr{C}$ such that there is a path containing a least one edge from v to v'. For any $V' \subseteq V$, we may define the *success* set of V' as follows:

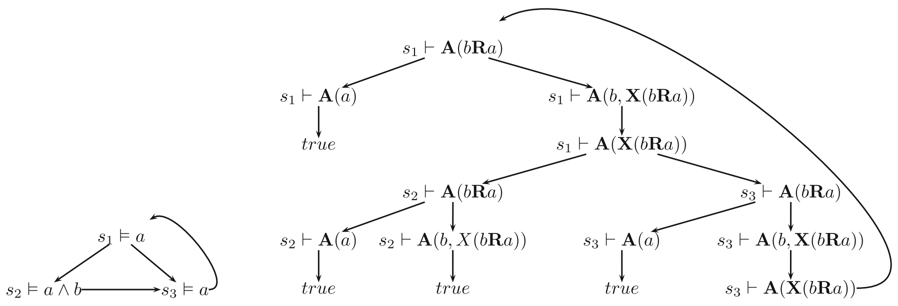

$s_1 \vDash a$
$s_2 \vDash a \wedge b$
$s_3 \vDash a$
$s_1 \vdash \mathbf{A}(b\mathbf{R}a)$
$s_1 \vdash \mathbf{A}(a)$
true
$s_1 \vdash \mathbf{A}(b, \mathbf{X}(b\mathbf{R}a))$
$s_1 \vdash \mathbf{A}(\mathbf{X}(b\mathbf{R}a))$
$s_2 \vdash \mathbf{A}(b\mathbf{R}a)$
$s_3 \vdash \mathbf{A}(b\mathbf{R}a)$
$s_2 \vdash \mathbf{A}(a)$
$s_2 \vdash \mathbf{A}(b, X(b\mathbf{R}a))$
$s_3 \vdash \mathbf{A}(a)$
$s_3 \vdash \mathbf{A}(b, \mathbf{X}(b\mathbf{R}a))$
$s_3 \vdash \mathbf{A}(\mathbf{X}(b\mathbf{R}a))$

Fig. 4 Proof-structure (*right*) of $s_1 \vdash \mathbf{A}(b\mathbf{R}a)$ for a simple Kripke structure (*left*)

$$Success(V') \overset{\text{df}}{=} \{\phi_1\mathbf{R}\phi_2 \mid \exists\sigma \in V' : \phi_1\mathbf{R}\phi_2 \in \sigma \text{ and } \forall\sigma' \in V' : \phi_2 \notin \sigma'\}.$$

We say that V' is successful if and only if $Success(V') \neq \emptyset$ we have the following:

Theorem 2 (SCC and LTL [7]) *A partially successful proof structure $\langle V, E\rangle$ is successful if and only if every nontrivial* SCC *of $\langle V, E\rangle$ is successful.*

For example, Fig. 4 gives the successful proof structure of the checking of $s_1 \vDash \mathbf{A}(b\mathbf{R}a)$ for a Kripke structure with three states such that atomic proposition a is always true and b is only true for state s_2.

3 BSP state-space construction of security protocols

Based on the properties defined in Sect. 2.2.3, we have designed, in an incremental manner, a BSP algorithms for efficiently computing the state space of security protocols. To explain our parallel algorithm, we start with a generic and sequential algorithm that corresponds to the usual construction of a state space and we also give a generic (in the sense of independent of **succ**) parallel algorithm for state-space computing which will be the basis for the parallel version. Successive *improvements* will result in a parallel 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 that this algorithm is *efficient* despite its simplicity.

3.1 Usual generic sequential algorithm

The algorithm given in Fig. 5 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` (taken using the **`pick`** routine) 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 S. Note that this algorithm could be made strictly depth-first using `todo` as a stack, and breadth-first using `todo` as a FIFO queue. This has not been considered here.

We now show how the sequential algorithm can be parallelised in BSP and how several successive improvements can be introduced.

```
1  def seq_construction() is
2    todo ← {$s_0$}
3    known ← ∅
4    while todo ≠ ∅ do
5      s ← todo.pick()
6      known ← known ∪ {s}
7      todo ← todo ∪ (succ(s) \ known)
8    done
```

Fig. 5 Sequential construction of the state space

```
1  def par_construction() is =
2    total ← 1
3    known ← ∅
4    if cpu($s_0$)=mypid
5      then todo ← {$s_0$}
6      else todo ← ∅
7    while total> 0 do
8      tosend ← local_successors(known,todo)
9      exchange(todo,total,known,tosend)
10   done
```

```
1  def exchange (todo,total,known,tosend) is
2    rcv,total ← BspExchange(tosend)
3    todo ← rcv \ known
```

```
1  def local_successors (known,todo) is
2    tosend ← [∅,··· ,∅]
3    while todo ≠ ∅ do
4      s ← todo.pick()
5      known ← known ∪ s
6      for s' in ((succ s) \ known) do
7        tgt=cpu(s')
8        if tgt=mypid
9          then todo ← todo ∪ s'
10         else tosend[tgt] ← tosend[tgt] ∪ s'
11     done
12   done
13   return tosend
```

Fig. 6 Generic and naive BSP algorithm for state-space construction

3.2 A naive and generic BSP algorithm for state-space computation

One of the main technical issues in the distributed-memory state-space construction is to partition the state space among the participating machines. 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 cpu that partitions the state space into subsets of states. Each such subset is then "owned" by a single machine. The partition function cpu returns for each state s a processor identifier, i.e. the processor numbered cpu(s) is the owner of s. Usually, this function is simply a hash (modulo the number of processors in the parallel computer) of the considered state.

We now show how the sequential algorithm can be parallelised in a BSP fashion and how several successive improvements can be introduced in the next subsections. The idea is that each process computes the successors for only the states it owns. This is rendered as algorithm called "Naive" in Fig. 6 ; notice that we assume that arguments are passed by references so that they may be modified by sub-programs.

This is a (SPMD) Single Program, Multiple Data algorithm and so, processor executes it. Sets known and todo (and all other variables) are strictly local to each processor and thus provide only a partial view on the ongoing computation. 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 local_successors is essentially the same as the sequential exploration, except that each processor computes only the successors for the states it actually owns and send other states to other processors. That is, function local_successors compute the successors of the states in todo and each computed state that is not owned by the local processor is recorded in the array of sets `tosend` together with its owner number. Array `tosend` is thus of size **`nprocs`**, the number of processors of the BSP machine: at processor `j`, `tosend`[i] represents the set of states that will be send by processor `j` to processor `i`. This partitioning of states is performed in lines 6–11. To finish, the function returns the states to be sent.

Then, function `exchange` is responsible for performing the actual communications between processors. It assigns to `todo` the set of received states that are not yet known locally together with the new value of `total`. The routine **`BspExchange`** performs a global (collective) synchronisation *barrier* which makes data available for the next super-step so that all the processors are now synchronised. The synchronous routine **`BspExchange`** sends each state s from the set `tosend[i]` to the processor i and returns the set of states received from the other processors, together with the total number of exchanged states—it is mainly the MPI's *alltoall* primitive. Notice that, by *postponing* communication, this function allows buffered sending and forbids sending several times the same state. More formally, at processor **`mypid`**:

$$\textbf{BspExchange}(\text{tosend}) = \begin{cases} \texttt{total} = \sum_{k=0}^{\textbf{nprocs}-1} \sum_{i=0}^{\textbf{nprocs}-1} |\texttt{tosend}[\![k]\!][i]| \\ \texttt{rcv} = \bigcup_{i=0}^{\textbf{nprocs}-1} \texttt{tosend}[\![i]\!][\textbf{mypid}] \end{cases}$$

where *tosend* $[\![i]\!]$ represents the array *tosend* at processor i.

To terminate the algorithm, we use the additional variable `total` in which we count the total number of sent states, i.e. `total` is an upper sum of the sizes of all the sets `todo`after the synchronisation. We have thus not used any complicated methods as the ones presented in [24]. It can be noted that the value of `total` may be greater than the total number of states in the `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 only one is kept upon reception. Moreover, if this state has been also computed by its owner, it will be ignored. This is 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.

We now consider how to incrementally optimise this BSP algorithm for the case of security protocols using their specific properties. An interesting point of this work is that the main loop of the BSP algorithm will be kept unchanged, i.e. only functions `local_successors` and `exchange` will be modified.

3.3 Dedicated BSP algorithm for state-space construction of security protocols

3.3.1 Increasing local computation time

Using the above naive parallel algorithm, function cpu distributes evenly the states over the processors. However, each super-step is likely to compute very few states because only too 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 communi-

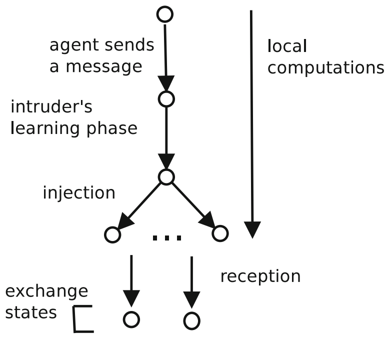


Fig. 7 Peculiarity of the state space

cation. 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 `local_successors`.

To achieve this goal, we consider a peculiarity of the models we are analysing that is depicted in Fig. 7. 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 reception by agents. Each such reception gives rise to a new state.

This whole process can be kept local to the processor and so without cross transition. To do so, we need to design our partition function $\mathsf{cpu}_{\mathscr{R}}$ such that it respects property (P1), i.e. for all states s_1 and s_2, if $s_1|_{\mathscr{R}} = s_2|_{\mathscr{R}}$ then $\mathsf{cpu}_{\mathscr{R}}(s_1) = \mathsf{cpu}_{\mathscr{R}}(s_2)$. This can be obtained using cpu but employing only the locations from $\mathscr{R}$, i.e. those locations where the honest agents store received information.

In this first improvement of the algorithm, when the function `local_successors` is called, then all new states from $\mathsf{succ}_{\mathscr{L}}$ are added in `todo` (states to be proceeded) and states from $\mathsf{succ}_{\mathscr{R}}$ are sent to be treated at the next super-step, enforcing an order of exploration of the state space that matches the progression of the protocol in *slices*. Another difference is that no state could be sent twice due to this order. The new function `local_successors` is given at the left of Fig. 8.

With respect to the previous algorithm, this one splits the local computations, avoiding calls to $\mathsf{cpu}_{\mathscr{R}}$ when they are not required. This may yield a performance improvement, both because $\mathsf{cpu}_{\mathscr{R}}$ is likely to be faster than cpu and because we only call it when necessary. But the main benefits in the use of $\mathsf{cpu}_{\mathscr{R}}$ instead of cpu is to generate less cross transitions since less states are need to be sent. Finally, notice that, on some states, $\mathsf{cpu}_{\mathscr{R}}$ 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 now show how this can be exploited.

3.3.2 Decreasing local storage

One can observe that the structure of the computation now *matches* the structure of the protocol execution: each super-step computes the executions of the protocol until

```
#An exploration to improve local computations
def local_successors (known,todo) is
  tosend ← [$\emptyset, \cdots, \emptyset$]
  while todo $\neq \emptyset$ do
    s ← todo.pick()
    known ← known $\cup$ s
    todo ← todo $\cup$ (succ$_{\mathscr{L}}$(s) $\setminus$ known)
    for s' in succ$_{\mathscr{R}}$(s) do
      tgt ← cpu$_{\mathscr{R}}$(s')
      tosend[tgt] ← tosend[tgt] $\cup$ s'
    done
  done
  return tosend
```

```
#Sweep−line implementation
def exchange (todo,total,known,tosend) is
  dump(known)
  todo,total ← BspExchange(tosend)
```

```
#Balancing strategy
def exchange (todo,total,known,tosend) is
  dump(known)
  todo,total ← BspExchange(balance(tosend))

def balance(tosend) is
  histoL ← $\{(i, \sharp\{(i,s) \in$ tosend $\})\}$
  compute histoG from BspMulticast(histoL)
  return BinPack(tosend,histoG)
```

Fig. 8 Dedicated BSP algorithms for state-space construction of security protocols

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. This corresponds to property (P2).

This kind of progression in a model execution is the basis of the *sweep-line* method [14] 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—a disk-based backup can also be made if it is necessary to restore the trace of a forbidden computation. Thus, in Fig. 8, 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.

Enforcing such an exploration order is usually made by defining on states a measure of progression slice as stated in property (P2). In our case, however, such a measure is not needed explicitly 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. This algorithm is as before except that we empty known at the end of each super-step, just before the next one. The corresponding new function exchange is given at the top-right of Fig. 8.

3.3.3 Balancing the computations

During our benchmark, we have found that using $\mathsf{cpu}_{\mathscr{R}}$ can introduce a bad balance of the computations due to a lack of information when hashing only on $\mathscr{R}$. Thus, the final optimisation step aims at rebalancing 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 (proportional actually) 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 states 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 $\mathsf{cpu}_{\mathscr{R}}$ without modulo. This function defines classes of states for which cpu_B returns the same value. Those classes are like "bag-of-tasks" [31] that can be

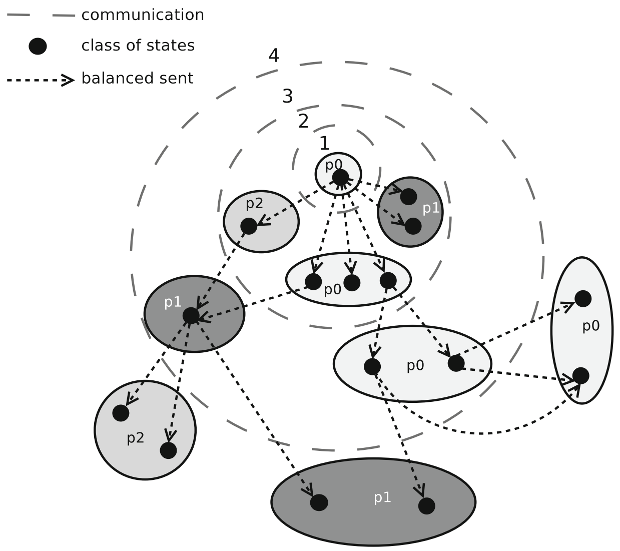


Fig. 9 Distribution of the sets of states

distributed over the processors independently, see Fig. 9. To do so, we compute a *histogram* of these classes on each processor, which summarises how $\mathsf{cpu}_{\mathscr{R}}$ would dispatch the states. This local histograms are then 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. We call this algorithm “Balance”.

These operations are detailed in the bottom-right part of Fig. 8, 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 **`BspMulticast`** is used to allow each processor to send its local histogram to every processor and receive in turn their histograms, allowing to build the global one. It thus 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 know which processor owns it: this could not be obtained efficiently when dynamic remapping is used. But that could not happen thanks to our sweep-line compatible exploration order. Our dynamic states *remapping* is thus correct because states classes match the locality of computation.

3.4 Experimental results

To evaluate our algorithms, we have implemented a prototype version in PYTHON, using SNAKES 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 BSP-PYTHON library [27] for the BSP routines (which are close to a MPI's "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 or computation of the states in SNAKES), this prototype nevertheless allows and accurate *comparison* of the various *algorithms*—execution times of PYTHON programs are very stable over several execution and not depend of code placement in the main memory or of unpredictable underlying optimizations of the compiler/processor. The benchmarks presented below have been performed using the cluster of the first author's laboratory that is 20 PCS connected through a 1 GB Ethernet network. Each PC is equipped with a 2 GHz Intel® Pentium® dual core CPU, with 2 GB of physical memory. This allowed to simulate easily a BSP computer with at most 40 processors equipped with 1 GB of memory each.

Our cases study involved the following five 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) Woo and Lam Pi (WLP) authentification protocol with public keys and trusted server; (5) Kao–Chow (KC) key distribution and authentication. All are documented at the Security Protocols Open Repository (SPORE) (http://www.lsv.ens-cachan.fr/Software/spore).

For each protocol, we have built a modular model allowing for defining easily various scenarios involving different numbers of each kind of agents. We note our scenarios NS_*x*-*y* indicating *x* instances of Alice and *y* instances of Bob with one unique sequential session; Y (*resp.* OR, KC, WLP)-*x*-*y*-*z*_*n* indicating *x* instances of the Server, *y* of Alice, *z* of Bob, involved in *n* sequential sessions.

We give the total time of computation and note **SWAP** when at least one processor has started to swap to disk due to a lack of main memory for storing its part of the state space. We also note **COMM** when a similar situation happens during communication: the system is unable to received data since not enough memory is available. We also give the number of states. For the Needham–Schroeder protocol, we have:

Scenario	Naive	Balance	Nb_states
NS _1-2	0m50.222s	0m42.095s	7807
NS _1-3	115m46.867s	61m49.369s	530713
NS _2-2	112m10.206s	60m30.954s	456135

For the Yahalom protocol:

Scenario	Naive	Balance	Nb_states
Y _1-3-1	12m44.915s	7m30.977s	399758
Y _1-3-1_2	30m56.180s	14m41.756s	628670
Y _1-3-1_3	481m41.811s	25m54.742s	931598
Y _2-2-1	2m34.602s	2m25.777s	99276
Y _3-2-1	**COMM**	62m56.410s	382695
Y _2-2-2	2m1.774s	1m47.305s	67937

For the Otway–Rees protocol:

Scenario	Naive	Balance	Nb_states
OR _1-1-2	38m32.556s	24m46.386s	12785
OR _1-1-2_2	196m31.329s	119m52.000s	17957
OR _1-1-2_3	411m49.876s	264m54.832s	22218
OR _1-2-1	21m43.700s	9m37.641s	1479

For the Woo and Lam Pi protocol:

Scenario	Naive	Balance	Nb_states
WLP _1-1-1	0m12.422s	0m9.220s	4063
WLP _1-1-1_2	1m15.913s	1m1.850s	84654
WLP _1-1-1_3	**COMM**	24m7.302s	785446
WLP _1-2-1	2m38.285s	1m48.463s	95287
WLP _1-2-1_2	**SWAP**	55m1.360s	946983

For the Kao–Chow protocol:

Scenario	Naive	Balance	Nb_states
KC _1-1-1	4m46.631s	1m15.332s	376
KC _1-1-2	80m57.530s	37m50.530s	1545
KC _1-1-3	716m42.037s	413m37.728s	4178
KC _1-1-1_2	225m13.406s	95m0.693s	1163
KC _1-2-1	268m36.640s	159m28.823s	4825

We can see that the overall performance of our dedicated “Balance” algorithm is always very good compared to the naive and general one. This holds for large state spaces as well as for smaller ones. Furthermore, the naive implementation can swap, which never happens for the “Balance” one.

By measuring the memory consumption of our “Balance” algorithm, we could confirm the benefits of our sweep-line implementation when large state spaces are computed. For instance, in a 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 a Y scenario with 1M states, the peak decreases from 97 to 60 % (states in Y use more memory that states in NS). Similarly, for the WLP _1-2-1_2, the peak decreases so that the computation does not swap. For Y _3-2-1, “Balance” used a little less memory but this is enough to *avoid crashing* the whole machine. 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.

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 (<100 states) for which the degree of intrinsic parallelism is very reduced but whose state space is in any way computed very quickly.

4 BSP on-the-fly LTL checking of security protocols

4.1 A sequential imperative algorithm for generic on-the-fly LTL checking

Bhat et al. [7] gives a recursive algorithm for LTL checking. It is mainly the *recursive Tarjan algorithm* for a SCC decomposition but working on proof structures and finding *on-the-fly* a *successful* SCC to validate or not the formula: it combines the construction of a proof structure with the process of checking whether it is successful; as soon as it is determined that the partially constructed structure cannot be extended successfully, the routine halts the construction of the structure and returns answer **`False`**.

To be close to our previous distributed algorithms, we have chosen to *derecursify* this algorithm using, as usual, an explicit stack to record the recursive calls. Instead of the recursive procedure, we use procedures `call_ltl`, `loop_ltl`, `up_ltl` and `ret_ltl` and an additional stack `todo` (which contains initially the initial state) to achieve a derecursification of the traditional recursive Tarjan’s algorithm. Note the definition of subroutines in the main procedure without their body which are given separately. This notation is used to define the scope of variables and to decompose the algorithm into several routines. Figure 10 gives this algorithm which operates as follows.

Roughly speaking, a break of the procedure “loop_ltl” resumes the nested exploration by popping the stack `todo` in which we have placed the next state to explore.

```
def modchkLTL_Seq() is
 $\sigma_0 = s_0 \vdash \phi$
 return SeqChkLTL($\sigma_0$)

def SeqChkLTL($\sigma$) is
  var dfn $\leftarrow 0$
  var stack $\leftarrow \varepsilon$
  var todo $\leftarrow [\sigma]$
  def init($\sigma$,valid) is (...)
  def loop_ltl($\sigma$) is (...)
  def up_ltl($\sigma$,$\sigma'$) is (...)
  def ret_ltl($\sigma$) is (...)
  def subgoals($\sigma$) is (...)
  while todo $\neq \varepsilon$
    $\sigma \leftarrow$ todo.pop()
    call_ltl($\sigma$)
  done
  return $\sigma$.flag
```

```
def subgoals($\sigma$) is
  case $\sigma$
    $s \vdash \mathbf{A}(\Phi, p)$ : $(R1 - R2)$
      if $(s \vDash p)$ then subg $\leftarrow$ {True}
      elif $\Phi = \emptyset$ then subg $\leftarrow \emptyset$
      else subg $\leftarrow \mathbf{A}(\Phi)$
    $s \vdash \mathbf{A}(\Phi, \phi_1 \vee \phi_2)$ : $(R3)$
      subg $\leftarrow \{s \vdash \mathbf{A}(\Phi, \phi_1, \phi_2)\}$
    $s \vdash \mathbf{A}(\Phi, \phi_1 \wedge \phi_2)$ : $(R4)$
      subg $\leftarrow \{s \vdash \mathbf{A}(\Phi, \phi_1), s \vdash \mathbf{A}(\Phi, \phi_2)\}$
    $s \vdash \mathbf{A}(\Phi, \phi_1 \mathbf{U} \phi_2)$ : $(R5)$
      subg $\leftarrow \{s \vdash \mathbf{A}(\Phi, \phi_1, \phi_2),$
      $s \vdash \mathbf{A}(\Phi, \phi_2, X(\phi_1 \mathbf{U} \phi_2))\}$
    $s \vdash \mathbf{A}(\Phi, \phi_1 \mathbf{R} \phi_2)$ : $(R6)$
      subg $\leftarrow \{s \vdash \mathbf{A}(\Phi, \phi_2),$
      $s \vdash \mathbf{A}(\Phi, \phi_1, \mathbf{X}(\phi_1 \mathbf{R} \phi_2))\}$
    $s \vdash \mathbf{A}(\mathbf{X}\phi_1, \ldots, \mathbf{X}\phi_n)$ : $(R7)$
      subg $\leftarrow \{s' \vdash \mathbf{A}(\phi_1, \ldots \phi_n) \mid s' \in$ succ$(s)\}$
  return subg
```

```
def init($\sigma$,valid) is
  dfn $\leftarrow$ dfn+1
  $\sigma$.dfsn $\leftarrow \sigma$.low $\leftarrow$ dfn
  $\sigma$.valid $\leftarrow \{\langle \phi_1 \mathbf{R} \phi_2$,sp$\rangle \mid \phi_2 \notin \sigma$
    $\wedge (\phi_1 \mathbf{R} \phi_2 \in \sigma \vee X(\phi_1 \mathbf{R} \phi_2) \in \sigma)$
    $\wedge$ sp=(sp$'$ if $\langle \phi_1 \mathbf{R} \phi_2$,sp$'\rangle \in$ valid else dfn)$\}$
```

```
def call_ltl($\sigma$) is
  if $\sigma$.parent $= \perp$
    valid $\leftarrow \emptyset$
  else
    valid $\leftarrow \sigma$.parent.valid
  init($\sigma$,valid)
```

```
  $\sigma.V \leftarrow$ True
  $\sigma$.instack $\leftarrow$ True
  stack.push($\sigma$)
  $\sigma$.children $\leftarrow$ subgoals($\sigma$)
  case $\sigma$.children
    {True} :
      $\sigma$.flag $\leftarrow$ True
      ret_ltl($\sigma$)
    $\emptyset$ :
      $\sigma$.flag $\leftarrow$ False
      ret_ltl($\sigma$)
    otherwise :
      loop_ltl($\sigma$)
```

```
def loop_ltl($\sigma$) is
  while $\sigma$.children $\neq \emptyset$ and $\sigma$.flag != False
    $\sigma' \leftarrow \sigma$.children.pick()
    if $\sigma'.V$
      if not $\sigma'$.flag
        $\sigma$.flag $\leftarrow$ False
      elif $\sigma'$.instack
        $\sigma$.low $\leftarrow$ min($\sigma$.low, $\sigma'$.low, $\sigma'$.dfsn)
        $\sigma$.valid $\leftarrow \{\langle \phi_1 \mathbf{R} \phi_2$,sp$\rangle \in \sigma$.valid
                                  $\mid$ sp$\leq \sigma'$.dfsn$\}$
        if $\sigma$.valid $= \emptyset$
          $\sigma$.flag $\leftarrow$ False
    else
      $\sigma'$.parent $\leftarrow \sigma$
      todo.push($\sigma'$)
      return
  done
  if $\sigma$.dfsn $= \sigma$.low
    var top $\leftarrow \perp$
    while top $\neq \sigma$
      top $\leftarrow$ stack.pop()
      top.instack $\leftarrow$ False
      if not $\sigma$.flag
        top.flag $\leftarrow$ False
    done
  ret_ltl($\sigma$)
```

```
def ret_ltl($\sigma$) is
  if $\sigma$.parent $\neq \perp$
    up_ltl($\sigma$.parent,$\sigma$)
```

```
def up_ltl($\sigma$,$\sigma'$) is
  $\sigma$.flag $\leftarrow \sigma'$.flag
  if $\sigma'$.low $\leq \sigma$.dfsn
    $\sigma$.low $\leftarrow$ min($\sigma$.low, $\sigma'$.low, $\sigma'$.dfsn)
    $\sigma$.valid $\leftarrow \sigma'$.valid
  loop_ltl($\sigma$)
```

Fig. 10 Sequential imperative algorithm for LTL model checking

The *backtracking* is done by the procedure `ret_ltl` which restores the control to its *parent call*, that in turn may possibly resume the exploration of its *children*.

Additional informations are stored in each assertion (a vertex) σ of the proof structure that enable the detection of unsuccessful SCC. We use an *implicit mapping* from the pairs $\langle state, \mathbf{A}\Phi \rangle$ (as keys) to fields of assertions that are assigned appropriately when assertions are first visited. These fields are the following:

– The algorithm of [7] maintains two sets of assertions: `V` (for visited), which records the assertions that have been encountered so far, and `F`, which contains assertions that have been determined to be **False** (by abuse of language, we say that the answer of the assertion is invalid). To implement this, σ has two boolean fields `.V`—initially **`False`** and `.flag`. The latter determines the validity of the assertion if σ`.V` is true. Initially `flag` is **`True`**, and it becomes **`False`** either if the set of subgoals of an assertion is empty or if one of these two conditions is satisfied:
  – one of the subgoals of the assertion is already visited and its `flag` is **`False`** (this case will actually occur when we check CTL* formulas);
  – an unsuccessful nontrivial strongly component is found by testing if the set `valid` is empty or not.
– The field `.parent` is a set of assertions σ' such that $(\sigma', \sigma) \in E$ that is there is a edge from σ' to σ in the proof structure (direct graph); it is mainly used for backtracking the results of nested computations; in the same manner, the field `.children` is also a set of assertions such that $(\sigma, \sigma') \in E$.
– As the algorithm consists of a depth-first exploration of the proof structure, σ has two specific fields used to detect SCCs: the depth-first search number of σ (`.dfsn`) and `.low` (the record of the dept-first search number of the "oldest" ancestor of σ that is reachable from σ), both expressing, respectively, the depth-first search number (by incrementation of the `dfn` variable) and the smallest depth-first search number of a state that is reachable from the considered state. The detection that a state belongs to a SCC is made by testing if a successor is in the global stack. A SCC is found at a certain point if at the end of a some course of the proof structure, the field `.low` coincides with the field `.dfsn`.
– We associate the field `.valid` which is the set of pairs of the form $\langle \phi_1 \mathbf{R} \phi_2$, `sp`$\rangle$. Intuitively, the formula component of such a pair may be used as evidence of the success of the SCC that σ might be in, while `sp` records the "starting point" of the formula, i.e. the depth-first number of the assertion in which this occurrence of the formula first appeared.
– We also need a test of membership of assertions in the global stack. To have a constant time test avoiding to actually explore the stack, we add another field `.stack` that is a Boolean answering whether the assertion is in the stack or not.

For model checking LTL formulas, we begin by the procedure `modchkltl _Seq` which initiates the variables `dfn` and `stack` and start the depth-first exploration by putting the initial assertion in `todo` (lines 6–13). The main loop over `todo` is to construct a successful proof structure (lines 14–17).

Procedure `call_ltl` proceeds as follows. The successors of the current assertion are computed by subroutine `subgoals` (line 10): it applies the rules of Fig. 3 and when no subgoal is found an error occurs (this is an unsuccessful proof structure).

If the children (subgoals) of σ are all **`True`** (valid), then it backtracks to the parent call (using procedure `ret_ltl`). Else there is no child and thus it is an unsuccessful proof structure, the assertion is not valid and it again backtracks to the parent call. Otherwise, we need to iterate over the children using a call to `loop_ltl`.

Procedure `loop_ltl` proceeds as follows: If subgoal σ' has already been examined (i.e. field `V` is true in line 4) and found to be **`False`** (line 5), then the proof structure cannot be successful, and we terminate the processing to return **`False`**: we pop all the assertions from the stack and if they are in the same SCC, they are marked to be **`False`** (lines 19–23). If σ' has not been found **`False`**, and if σ' is in the stack (meaning that its SCC is still being constructed), the σ and σ' will be in the same SCC: we reflect this by updating σ`. low` accordingly. We also update σ`.valid` by removing formulas whose starting points occur *after* σ'; as we show below, these formulas cannot be used as evidence for the success of the SCC containing σ and σ' (lines 8–14). Once the subgoal processing is completed, `loop_ltl` checks to see whether a new SCC component has been detected; if no, it removes it from the stack (lines 18–23) and finally backtracks to the parent call (line 25).

Procedure `ret_ltl` is just a call to `up_ltl` if the assertion has no "parent". Procedure `up_ltl` update the field`.low` and `.dfsn` as the traditional Tarjan algorithm and restarts the exploration of the other children by a call to `loop_ltl`.

Notice that using "proof-structures" is not common, LTL checking is traditionally perform by the test of emptiness of a Büchi automaton which is the product of the LTS and of the formula translated to an automaton. In general, a NDFS algorithm checks the presence of an accepting cycle. Our approach "simplifies" the use of our two successors functions and allows us to check CTL* formula without using any (alternative hesitant) automaton which are slow to compute.

4.2 BSP on-the-fly checking a LTL formula over security protocols

As explained in the previous sections, we use two LTS successor functions for constructing the Kripke structure: $\mathsf{succ}_{\mathscr{R}}$ ensures a measure of progression "slice" that intuitively decomposes the Kripke structure into a sequence of slices $S_0, \ldots, S_n$ where transitions from states of S_i to states of S_{i+1} come only from $\mathsf{succ}_{\mathscr{R}}$ and there is no possible path from states of S_j to states S_i for all $i < j$. In this way, we have used a distribution of the Kripke structure across the processors using the cpu_B function, we thus naturally extend this function to assertions σ using only the state field. Then, with this distribution, the only possible accepting cycles or SCCs *are local to each processor*. Thus, because proof structures follow the Kripke structure (rule R7), accepting cycles or SCCs are also only locals. Call this sequential algorithm **`SeqChkLTL`** (the only difference with the previous one is the subprocedure "subgoal" due to the two successors functions) which takes an assertion $\sigma \equiv s \vdash \mathbf{A}\Phi$. It also modifies the set of assertions to be sent (for the next super-step). Now, we can design our BSP algorithm which is mainly an iteration over the *independent slices*, one slice per super-step and, on each processor, working on independent sub-parts of the slice by calling **`SeqChkLTL`**. This SPMD (this executed by each processor executes) algorithm is given in Fig. 11.

```
def modchkLTL_Par() is
 return ParChkLTL($\sigma_0$)

def ParChkLTL(($s \vdash \Phi$) as $\sigma_0$) is
  super_step,dfn,tosend,todo$\leftarrow$0,0,$\emptyset$,$\emptyset$
  flag,total$\leftarrow \perp$,1
  def SeqChkLTL($\sigma$) is (as previously)
  def subgoals($\sigma$) is (...)
  def exchange() is (...)
  if cpu($\sigma_0$)=mypid
    todo$\leftarrow$todo$\cup\{\sigma_0\}$
  while flag=$\perp \wedge$ total>0
    tosend$\leftarrow\emptyset$
    while todo $\neq \emptyset \wedge$ flag=$\perp$
      $\sigma \leftarrow$todo.pick()
      if not $\sigma$.V
        flag$\leftarrow$SeqChkLTL($\sigma$)
    done
    if flag$\neq\perp$
      tosend$\leftarrow\emptyset$
      exchange()
  done
  case flag
    | $\perp \Longrightarrow$ return "OK"
    | $\sigma \Longrightarrow$ return Build_trace($\sigma$)

def exchange() is
  dump (V,E) at super_step
  super_step$\leftarrow$super_step+1
  tosend$\leftarrow$tosend$\cup\{(i,$flag$) \mid 0 \leq i < \mathbf{p}\}$
  rcv,total$\leftarrow$BspExchange(balance(tosend))
  flag,todo$\leftarrow$filter_flag(rcv)

def subgoals($\sigma$) is
  case $\sigma$
  | $s \vdash \mathbf{A}(\Phi, p) \Longrightarrow$ subg$\leftarrow$if $s \vDash p$ then {True}
       else $\{s \vdash A(\Phi)\}$ ($R1,R2$)
  | ($R3$), ($R4$), ($R5$) , ($R6$) $\Longrightarrow$ (as previously)
  | $s \vdash \mathbf{A}(\mathbf{X}\phi_1,...,\mathbf{X}\phi_n) \Longrightarrow$
      subg$\leftarrow\{s' \vdash \mathbf{A}(\phi_1,...\phi_n) \mid s' \in succ_L(s)\}$
      send$\leftarrow\{s' \vdash \mathbf{A}(\phi_1,...\phi_n) \mid s' \in succ_R(s)\}$
      E$\leftarrow$E$\cup\{\sigma \mapsto_R \sigma' \mid \sigma' \in$ send $\}$
      if subg=$\emptyset \wedge$ send$\neq\emptyset$
        subg$\leftarrow${True}
      tosend$\leftarrow$send$\cup$tosend ($R7$)
  V$\leftarrow$V$\cup$subg
  E$\leftarrow$E$\cup\{\sigma \mapsto_L \sigma' \mid \sigma' \in$ subg $\}$
  return subg
```

Fig. 11 A BSP algorithm for LTL checking of security protocols

The main procedure `ParChkltl` first initialises so that one processor owns the initial assertion and saves it in its `todo` list. The variable `total` stores the number of states to be processed at the beginning of each super-step; V and E store the proof structure (in fact, we manipulate an implicit mapping of assertions through the fields but it is sometime more readable to refer to V and E directly); `super_step` stores the current super-step number; `flag` is used to check whether the formula has been proved false (`flag` sets to the violating states) or not (`flag`=$\perp$).

The main loop processes each σ in `todo` using the sequential **SeqChk**LTL, which is possible because the corresponding parts of the proof structure are independent (property P4). **SeqChk**LTL uses `subgoals` to traverse the proof structure. For rules (R1) to (R6), the result remains local because SCC can only be locals. However, for rule (R7), we compute separately the next states for $\text{succ}_{\mathscr{L}}$ and $\text{succ}_{\mathscr{R}}$: the former results in local states to be processed in the current step, while the latter results in states to be processed in the next step. If no local state is found but there exists remote states, we set `subg`$\leftarrow${**True**} which indicates that the local exploration succeeded (P2) and allows to proceed to the next super-step in the main loop. When all the local states have been processed, states are exchanged, which leads to the next slice, i.e. the next super-step. To terminate the algorithm as soon as one processor discovers a counterexample, each locally computed flag is sent to all the processors and the received values are then aggregated using function `filter_flag`hat selects the non-$\perp$ flag with the lowest `fn` value computed on the processor with the lowest number, which allows to ensure that every processor chooses the same flag and then computes the same trace. If no such flag is selectable, `filter_flag` returns $\perp$. To balance the computation, we

```
def Build_trace($\sigma$) is
  def Local_trace ...
  def Exchange_trace ...
  end$\leftarrow$ False
  repeat
    $\pi \leftarrow \varepsilon$
    my_round$\leftarrow$ (cpu($\sigma$)=mypid)
    end$\leftarrow$ ($\sigma$=$\sigma_0$)
    send$\leftarrow \emptyset$
    if my_round
      dump $(V,E)$ at super_step
      super_step$\leftarrow$ super_step$-$1
      undump $(V,E)$ at super_step
      $\sigma,\pi \leftarrow$ Local_trace($\sigma,\pi$)
      $\pi \leftarrow$ Reduce_trace($\pi$)
      $F \leftarrow F \cup$ set_of_trace($\pi$)
      print $\pi$
    $\sigma \leftarrow$ Exchange_trace(my_round,$\sigma$)
  until $\neg$end

def Exchange_trace(my_round,tosend,$\pi$) is
  if my_round
    tosend$\leftarrow$ tosend $\cup \{(i,\sigma) \mid 0 \leq i < \mathbf{p}\}$
  $\{\sigma\}$,_$\leftarrow$ BspExchange(tosend)
  return $\sigma$

def Local_trace($\sigma,\pi$) is
  if $\sigma = \sigma_0$
    return $(\sigma,\pi)$
  tmp$\leftarrow$ prec($\sigma$) $\setminus$ set_of_trace($\pi$)
  if tmp=$\emptyset$
    $\sigma' \leftarrow$ min_dfsn(prec($\sigma$))
  else
    $\sigma' \leftarrow$ min_dfsn(tmp)
  $\pi \leftarrow \pi.\sigma'$
  if $\sigma' \mapsto_R \sigma$
    return($\sigma',\pi$)
  return $(\sigma',\pi)$
```

Fig. 12 BSP algorithm for building the trace after an error

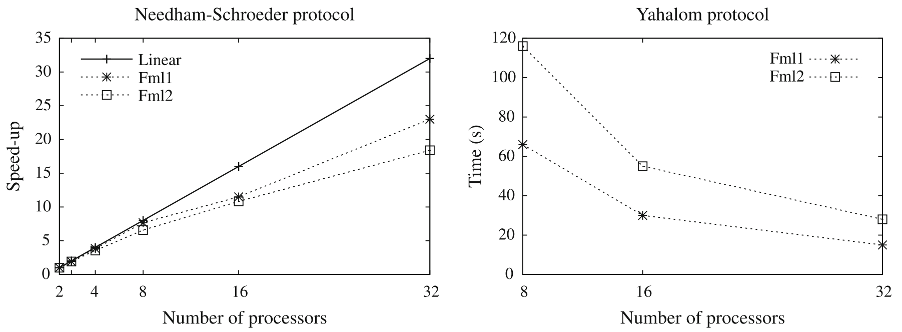
use the number of states as well as the size of the formula—on which the number of subgoals directly depends.

Notice also that at each super-step, each processor dumps V and E to its local disk, recording the super-step number, to be able to reconstruct a trace. When a state σ that invalidates the formula is found, a trace from the initial state to σ is constructed. The data to do so are distributed among processors into local files, one per super-step. We thus use exactly as many steps to rebuild the trace as we have used to reach σ. Figure 12 gives this algorithm. A trace π whose "oldest" state is σ is reconstructed following the proof structure backward. The processor that owns σ invokes `Local_trace` to find a path from a state σ', that was in `todo` at the beginning of the super-state, to σ. Then, it sends σ' to its owner to let the reconstruction continue. To simplify things, we print parts of the reconstructed trace as they are locally computed. Among the predecessors of a state, we always choose those that are not yet in the trace π (`set_of_trace(`π`)` returns the set of states in π) and selects one with the minimal `dfsn` value (using function `min_dfsn`), which allows to select shorter traces.

4.3 Experiments

As before, we have implemented a prototype of this algorithm using PYTHON and SNAKES again. While suboptimal comparing to a traditional model checker, this prototype nevertheless allows an accurate *comparison* for speedup.

To evaluate our algorithm, we have used two common formulas for verifying security protocols of the form ϕ **U** `deadlock`, where `deadlock` is an atomic proposition that holds iff a state has no successor and ϕ is a formula that checks for an attack on the considered protocol: Fml1 is the classical "secrecy" (ϕ is *auth* **R** *learn* where *auth* is an atomic proposition of authentification of the agents and *learn* the fact that the intruder has broken the secrecy) and Fml2 is the "availability" formula (presented



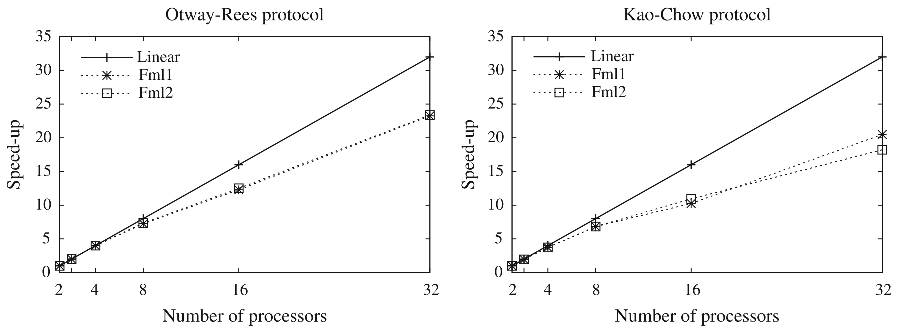


Fig. 13 Benchmark results of our BSP algorithm for LTL checking apply to four protocols where Fml1 is "secrecy" and Fml2 is "availability"

above) [1]. The chosen formula globally hold so that the whole proof structure is computed. Indeed, on several instances with counterexamples, we have observed that the sequential algorithm can be faster than the parallel version when a violating state can be found quickly: our parallel algorithm uses a global breadth-first search, while the sequential exploration is depth-first, which usually succeeds earlier. But when all the exploration has to be performed, which is widely acknowledged as the hardest case, our algorithm is always much faster. Moreover, we sometimes could not compute the state space sequentially while the distributed version succeeded.

Figure 13 gives the speedup for each of the two formulas and two sessions of each protocol. For the Yahalom protocol, the computation fails due to a lack of main memory if less that four nodes are used: so we could not give the speedup but only execution times. We observed a relative speedup with respect to the number of processors.

5 BSP on-the-fly CTL* checking of security protocols

As for LTL, we first present a sequential algorithm and then specialised parallel algorithms for security protocols. The first parallel algorithm called "naive" is a first attempt to extend the parallel algorithm for LTL checking to CTL* formulas, whereas the second one optimises the communications and reduces the number of super-steps.

```
def SeqRecChkCTL∗($\sigma$) is
  def SeqChkLTL($\sigma$) is (as previously)
  if not $\sigma.V$
    $\sigma.V \leftarrow$ True
    case $\sigma$
      $s \vdash p$ where $p \in \{a, \neg a\}$, $a \in \mathscr{A}$
        $\sigma$.flag $\leftarrow s \models p$
      $s \vdash p_1 \wedge p_2$
        $\sigma$.flag $\leftarrow$ SeqRecChkCTL∗($s \vdash p_1$)
                  $\wedge$ SeqRecChkCTL∗($s \vdash p_2$)
      $s \vdash p_1 \vee p_2$
        $\sigma$.flag $\leftarrow$ SeqRecChkCTL∗($s \vdash p_1$)
                  $\vee$ SeqRecChkCTL∗($s \vdash p_2$)
      $s \vdash \mathbf{A}\phi$
        $\sigma$.flag $\leftarrow$ SeqChkLTL($\sigma$)
      $s \vdash \mathbf{E}\phi$
        $\sigma$.flag $\leftarrow$ not SeqChkLTL($s \vdash$ neg($\mathbf{E}\phi$))
  return $\sigma$.flag
```

Fig. 14 Sequential recursive algorithm for CTL* model checking

5.1 A sequential algorithm for CTL* checking

The algorithm of [7] (named `SeqRecChkctl*` and presented in Fig. 14) processes a formulae P either by recursively call **`SeqChkLTL`** appropriately when it encounters assertions of the form $s \vdash \mathbf{A}\Phi$ or $s \vdash \mathbf{E}\Phi$, or by decomposing the formulae P into sub-formulas. Note the use of an equivalence of an exits-formula with the negation of the corresponding forall-formula to check the latter.

Note also a slight but important modification to procedure **`subgoals`**: when it encounters an assertion of the form $s \vdash \mathbf{A}(p, \Phi)$ (notably where p is $\mathbf{A}\phi$ or $\mathbf{E}\phi$), it recursively invokes `SeqRecChkctl*` $(s \vdash p)$ to determine if $s \models p$ and then decides if rule R1 or rule R2 (of Fig. 3) needs to be applied. In other words, by extending the atomic test in `subgoals` (and using `SeqRecChkctl*` for these sub-formula), we have a *double recursivity* of `SeqRecChkctl*` and **`SeqChk`**LTL.

Also note, that each call to **`SeqChk`**LTL creates a new empty stack and a new dfn (depth-first number) since a new LTL checking is run: by abuse of language, we will name them "LTL *sessions*" (or just *sessions*). These sessions can share assertions which thus share their validity (is in F or not). Take for example formula $\mathbf{A}(p\mathbf{U}(\mathbf{E}(r\mathbf{U}p)))$. There will be two sessions, one for the global formula and one for the sub-formula $\mathbf{E}(r\mathbf{U}p)))$. It is clear that the atomic proposition p need thus to be tested twice on the states of the Kripke structure. But the two sessions need only to share atomic propositions.

More subtly, LTL sessions do *not* share their depth-first numbers (`low` and `dfsn` fields) and their `valid` fields—except for literal. This is due to the rules of Fig. 3 that force that any recursive call of `SeqRecChkctl*` within a session (for checking a sub-formula that is not LTL and thus generating another session) is only performed on a sub-part of the original assertion that is strictly smaller. That guarantee there is no intersection of the proof structures of the parent sessions and disjoint SCCs.

The double recursion would be a problem to have an efficient *coarse-grain* parallel algorithm: it is not easy to stop and backup recursive calls. As for LTL, we have thus made the choice to derecursify the above algorithm to have an iterative algorithm. This allows to have only one main loop that has the advantage to easily stop the computation, whereas results of other processors are expected in parallel algorithms. For sake of conciseness, we do not give this purely technical algorithm and we refer to [26] where all the algorithms are fully described.

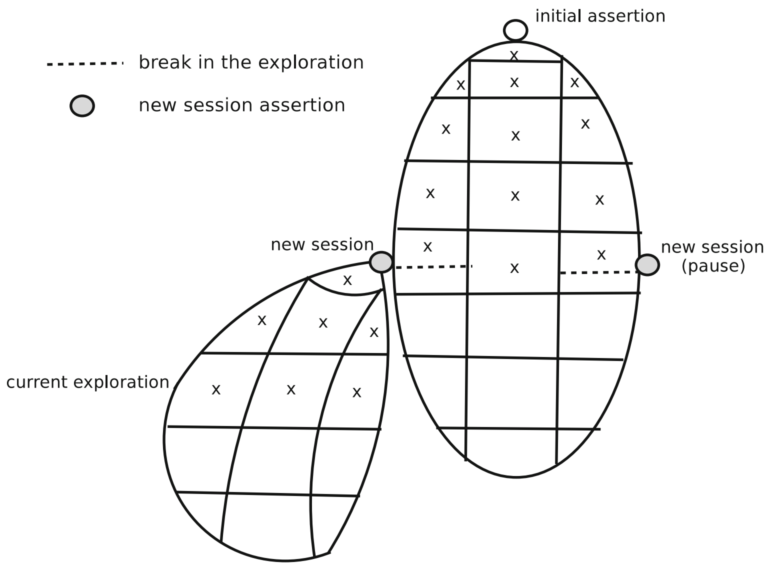


Fig. 15 Naive generation of the LTL sessions

5.2 A naive BSP algorithm for CTL* checking

We give here a first and naive attempt to parallelise the imperative algorithm for CTL* model checking. It is naive because it could imply a number of super-steps equal to the number of states in the underlying Kripke structure.

5.2.1 General idea

The algorithm works as follows. A first step is to recursively decomposing the formulae for finding the first assertion of the form $s \vDash \mathbf{A}(\phi)$. Then, a main loop is used to proceed the received assertions: for each of them, an exploration is used to decompose the formulae and run `SeqChkctl*` adequately to check for an unsuccessful SCC in the proof structure. Recall that we name this computation a `ltl session` and considering it as a distinguished object. During the generation of the proof structure, when a sub-formulae beginning by **A** or **E** is found (case of rules $R1$ and $R2$), the *ongoing session is paused* (see Fig. 15), pushed onto a stack of waiting sessions and is kept their until the result of a new session to check the validity of $s \vDash p$ is available.

There are three main problems: (1) different processors can throw sessions, (2) a session can induce several super-steps (slices) if it is a path formula. This is due to the double recursion of the CTL* checking, (3) the different sessions are not fully disjoints: states of the Kripke structures as well as assertions can be shared, that happens when the same sub-parts of the Kripke structure are generated and when sets of formula in the assertions are not disjoints. This results in an *embarrassingly* parallel computation on this set of sessions. A naive solution is to globally select one of these generated sessions (the other still remains in a distributed global stack) and to entirely compute

```
1  #$\sigma_0 = s_0 \vdash \phi$
2  var slice $\leftarrow 0$
3  var $V \leftarrow \emptyset$
4  var $F \leftarrow \emptyset$
5
6  #First recursive decomposition for
7  #finding an assertions of the form $s \vdash \mathbf{A}\phi$
8  def ParNaiveChkCTL*($\sigma_0$) is
9    if not $\sigma.V$
10     $\sigma.V \leftarrow$ True
11     case $\sigma$
12   $s \vdash p$ where $p \in \{a, \neg a\}$, $a \in \mathscr{A}$ :
13      $\sigma$.flag $\leftarrow s \models p$
14   $s \vdash p_1 \wedge p_2$ :
15      $\sigma$.flag $\leftarrow$ ParNaiveChkCTL*($s \vdash p_1$)
16                 $\wedge$ ParNaiveChkCTL*($s \vdash p_2$)
17   $s \vdash p_1 \vee p_2$:
18       $\sigma$.flag $\leftarrow$ ParNaiveChkCTL*($s \vdash p_1$)
19                 $\vee$ ParNaiveChkCTL*($s \vdash p_2$)
20   $s \vdash \mathbf{A}\phi$ :
21     $\sigma$.flag $\leftarrow$ main_ParNaive*($\sigma$)
22   $s \vdash \mathbf{E}\phi$ :
23     $\sigma$.flag $\leftarrow$ not main_ParNaive($s \vdash$ neg($\mathbf{E}\phi$))
24   return $\sigma$.flag
25
26 def main_ParNaive($\sigma$) is
27   out_stack $\leftarrow \varepsilon$
28   answer_ltl, flag_list, mck
29   repeat
30     if $\sigma \neq \perp$
31       mck $\leftarrow$ new LTL_SESSION($\sigma$)
32       flag_list,$\sigma$ $\leftarrow$ mck.explore()
33       out_stack.push(mck)
34     else
35       if flag_list $\neq \emptyset$
36         answer_ltl $\leftarrow$ False
37         mck $\leftarrow$ out_stack.top()
38         mck.updateF(flag_list)
39       else
40         answer_ltl $\leftarrow$ True
41       out_stack.pop()
42       if out_stack $\neq \varepsilon$
43         mck $\leftarrow$ out_stack.top()
44         flag_list,$\sigma$ $\leftarrow$(
45                mck.recovery(answer_ltl))
46   until out_stack $= \varepsilon$
47   return answer_ltl
```

Fig. 16 Main procedures for the naive algorithm for parallel CTL* model checking

this session until another session is thrown or an answer is found (validity of a $s \models p$). A part of this algorithm called `ParNaiveChkctl*` is given in Fig. 16, and the full algorithm is available in [26].

5.2.2 Main loop

The initial recursive decomposition is performed in lines 9–23. Then, calls to the main loop `main_ParNaive*` are performed in lines 21 and 23. The following variables are used during the computation of the main loop:

- `out_stack` (initially empty) manages the exploration "depth" of the sessions by storing the LTL sessions;
- `answer_ltl` saves the answer/validity (**`True`** or **`False`**) when a session is finished;
- `flag_list` contains the assertions infringing the computation and is used for the backtracking;
- `mck` represents the session currently used (exploration, recovery, backtracking).

The main loop proceeds, until the stack of sessions is empty, by creating a session for the assertion if it is new (line 30); then by performing the exploration (`.explore`) and by pushing this session at the top of the stack. The flags are assertions that are not valid for the session (which does not induce that the overall formula is invalid if the session is run from within a **E** quantifier). In this case (line 35), the answer is potentially false and we must backtrack using method `.updateF` on the last pushed object. Otherwise, the answer of the session is true (line 38) and method `.updateF` works in the same manner as the procedure `Build_trace` of LTL (Fig. 12) for computing all the assertions that are not valid from the given `flag_list`—except that the full trace is not computed, but instead we gather all the assertions that are not valid.

5.2.3 Methods of the LTL *sessions*

The methods of the LTL sessions are given in Fig. 17. The method `.explore` of a "LTL session" generates in a parallel way the proof structure whose initial assertion is σ and stop when (line 17) either:

- a sub-formulae $\phi \in \sigma$ of an assertion $\sigma \equiv s \vdash \{\Phi, *\phi\}$ where $* \in \mathbf{A}$ or $\mathbf{E}$ is found (line 23), then the return value is $([], s \vdash *\phi)$. This first case corresponds to a halting of the current session;
- or, if the assertion is checked to be true, then the return value will be $([], \bot)$, else some assertions $\sigma_1, ..., \sigma_k$ invalidate the ongoing computation, i.e. the initial assertion is not valid; The return value will be thus $([\sigma_1, ..., \sigma_k], \bot)$.

In this method, `next_slice` (line 23) and `previous_slice` (line 26) are used to dump and undump the current proof structure when changing of slice according to the progression of our state-space construction.

We also recall that during the call to `subgoals` (computations of the SCCS i.e. `ParChkltl` procedure) when a sub-formulae beginning by **A** or **E** is found, the computation needs to be paused to begin a new session. To achieve this, we make a straightforward modification of `subgoal` by returning an empty set of successors and a flag that indicates if this is due to an invalid formulae or to the need for pausing the current computation. In the latter case, the ongoing "session" is paused and is set waiting for the answer of the new session based on the appropriate assertion.

Finally, method `.recovery` resumes a paused computation by passing as an argument the flag value corresponding to the validity of the assertion previously returned—and awaiting to test. This flag is an answer corresponding of the validity required on the assertion returned by `.explore`. Thus, as for method `.explore`, if the assertion is not checked then method `.recovery` returns the assertions that invalidate the ongoing computation. More precisely, the backtracking was already performed during the last computed slice, in accordance to the state-space algorithm. It remains to continue the backtracking from the assertions on the previous slices until the initial slice, i.e. the slice of the initial assertion of the ongoing session. This recovery of the backtracking is performed by the method `.updateF` which, as its name indicates, updates the set `F` of the false assertions. The method also uses the variable σ_{halted} which represents the last assertion computed before the computation of the session was halted. All these methods and a full example are fully available in [26].

5.2.4 Drawbacks of this naive algorithm

This naive approach suffers of three main drawbacks. First, each time a session is thrown, it can traverse all the state space in several super-steps. The number of super-steps would be *proportional* to the number of states in the Kripke structure. This can happen when the session has been thrown by a formula which contains modal

```
class LTL_Session is
  #member variables
  var stack, tosend, dfn
  var $\sigma$, $\sigma_{halted}$, todo

  #constructor
  LTL_Session($\sigma$) is
    self.$\sigma \leftarrow \sigma$
    $\sigma_{halted}$, stack, tosend $\leftarrow \perp$, $\varepsilon$, $\emptyset$
    dfn, todo $\leftarrow$ 0, $\emptyset$

  method explore() is
    total, flag, $\leftarrow$ 1, $\perp$
    flag_list, $\sigma_{totest}$ $\leftarrow \emptyset$, $\perp$
    if cpu($\sigma$)=my_pid
      todo$\leftarrow$todo $\cup$ $\{\sigma\}$
    while not flag_list and total>0
          and $\sigma_{totest} \neq \perp$
      tosend, $\sigma_{totest}$ $\leftarrow \emptyset$, todo.pick()
      flag$\leftarrow$SeqChkCTL*($\sigma_{totest}$)
      if flag $\neq \perp$
        next_slice()
        flag_list,total$\leftarrow$ (
           BspExchange(balance(tosend)))
    done
    previous_slice()
    if $\phi \neq \emptyset$ and ($\sigma_{totest} \equiv s \vdash \mathbf{A}\phi$
                or $\sigma_{totest} \equiv s \vdash \mathbf{E}\phi$)
      $\sigma_{halted} \leftarrow \sigma_{totest}$
      if $\sigma_{totest} \equiv s \vdash \mathbf{E}\phi$
        $\sigma_{totest} \leftarrow s \vdash$ neg($\mathbf{E}\phi$)
    return flag_list,$\sigma_{totest}$

  method recovery(answer_ltl) is
    if $\sigma_{halted} = p \vdash E\phi$ and answer_ltl = True
      $F \leftarrow F \cup \{\sigma_{halted}\}$
      $V \leftarrow V \cup \{\sigma_{halted}\}$
    var $\sigma_{totest}$ $\leftarrow \perp$
    flag $\leftarrow \perp$
    flag_list $\leftarrow \emptyset$
    if cpu($\sigma_{halted}$) = my_pid
      todo$\leftarrow$todo $\cup \sigma_{halted}$
    while todo and not flag and $\sigma_{totest} \neq \perp$
      (the rest as for .explore but with
       a test of membership of$\sigma_{totest} \in V$)

  method updateF(flag_list) is (...)
```

Fig. 17 LTL session for the naive CTL* BSP algorithm

operators, e.g. a formula of the form $\mathbf{AA}p$. This is due to the fact that the algorithm works as follows for this formula: for each state, test if $\mathbf{A}p$ is valid; thus, run each time a LTL session which would imply several super-steps to test $\mathbf{A}p$ (if literal p is valid on all the states of the Kripke structure). This can thus generate too many barriers and induce very poor performance.

Second, the sweep-line strategy used in the previous section cannot be applied: each slice does not correspond to a super-step and thus during backtracking of the answers, the data dumped on disks must be loaded back into the main memory (work of `next` and `previous_slice`). This can be very costly. The alternative is to keep everything in the main memory but with a serious risk of swapping.

Third, the balance of the assertions over the processors is done dynamically at each slice of each session: this ensures that two assertions for the same Kripke state are held by the same processor, which avoids duplication of computation. But if two sessions are run in sequence, the first one will balance some assertions and the second session, if some states are shared, must balance the assertions using this first partial scheme of balance which is complicated and largely suboptimal. The alternative to re-balance all the assertions would be too costly.

5.3 A "purely breadth" BSP algorithm for CTL* checking

To avoid these problems, we will take into account the "nature" of proof structures that include an explicit decomposition of the logical formulae which can help to choose where a parallel computation is needed or not. The main idea of the algorithm is to consider rules R1 and R2 of Fig. 3 and compute $s \vDash \phi$ together with $s \vdash \mathbf{A}(\Phi)$. This way, we will able to choose which rule ($R1$ or $R2$) must be applied.

More precisely, in the case of rules $R1$ and $R2$ of the decomposition of a LTL formulae, ϕ is an atomic proposition, which can thus be sequentially computed. But in the case of CTL*, ϕ can be any formulae. In the naive algorithm, we thus run another LTL session and pause the current computation until a result is provided. The approach proposed for the new algorithm is to compute both $s \vDash \phi$ and $s \vdash \mathbf{A}(\Phi)$, which provides the information to decide whether $R1$ or $R2$ must be applied. As previously, the computation of $s \vDash \phi$ can be performed by a kind of LTL session, while the computation of $s \vdash \mathbf{A}(\Phi)$ can be performed following the execution of the SCC computation. In a sense, we anticipate the result of $s \vDash \phi$ by computing the validity of the assertion $s \vdash \mathbf{A}(\Phi)$.

We see three main advantages. First, as we can compute both $s \vDash \phi$ and $s \vdash \mathbf{A}(\Phi)$ in parallel, we can *aggregate* the super-steps of the both computations and thus reduce the overall number of super-steps to the maximal number of slices of the model (slice progression). Second, we also aggregate the computations and the communications without losing their balance: we have in the same place all the assertions of each slice, which allows a better balancing than the use of the partial balances in the naive algorithm. Third, the computation of the validity of $s \vdash \mathbf{A}(\Phi)$ can be used later in different LTL sessions. On the other hand, the pre-computation of $s \vdash \mathbf{A}(\Phi)$ may generate unnecessary work. If we assume a sufficient number of processors, this is not a problem concerning scalability, and the exploration is performed in a breadth fashion that brings a higher degree of parallelism.

5.3.1 Main loop

Figure 18 gives the main loop of the algorithm. The computation is performed until the answer of the initial assertion σ_0 is found, which is recorded in variable `finish`. The computation works as follows and can be divided into three phases. First (lines 11–18), the current exploration of received assertions (processed one by one in lines 13–17) is performed. Secondly (lines 22–24), the propagation of the backtracks of the answers (not equal to $\bot$) found especially on other machines is performed. Note that in the first stage some backtracks of answers can also be performed but they are local and done during the ongoing exploration. Between these two phases, an exchange between the machines is performed (line 20). Finally (line 25), dump from the main memory all the assertions that are no more used for the computation due to slice progression (sweep-line method described latter).

We have thus the overall `stack` (initially empty) due to our derecursification of the Tarjan algorithm and to the recursive decomposition of the CTL* formulae. `dfn` is the `deep first number` that can be intuitively shared by all SCC decompositions. For the management of the sending assertions, we use two distinct sets of messages. The first one (`snd_todo`) is to store the assertions which are used to continue the exploration of the distributed proof structure; The second one (`snd_back`) is for backtracking answers (for the case of rules R1/R2 expecting the answer about a $s \vDash \phi$). This way, at the beginning of a super-step, we first read answers regarding paused sessions (stored in a stack) which could then continue their SCC computations. Then,

```
def modchkCTL*() is
 $\sigma_0 = s_0 \vdash \phi$
 return ParBreadthChkCTL*($\sigma_0$)

def ParBreadthChkCTL*($\sigma_0$) is
  dfn,stack,snd_todo,snd_back $\leftarrow 0,\varepsilon,\emptyset,\emptyset$
  rcv,back,finish,todo $\leftarrow \emptyset,\emptyset$,False,$\emptyset$
  def all sub-procedures(...)
  if cpu($\sigma_0$)=mypid
    rcv $\leftarrow$ rcv $\cup \{\sigma_0\}$
  while not finish
    for $\sigma$ in rcv while not finish
       if not $\sigma.V$
         todo $\leftarrow [\sigma]$
         while todo$\neq \varepsilon$ and not finish
           $\sigma \leftarrow$ todo.pop()
           call_ctlstar($\sigma$)
         done
    done
    finish,back,rcv $\leftarrow$ BspExchange(
      finish, snd_back,balance(snd_todo))
    while back$\neq \varepsilon$ and not finish
      $\sigma$, child $\leftarrow$ back.pop()
      up_trace($\sigma$, child)
    sweep()
  done
  return $\sigma_0$.flag

def subgoals($\sigma$) is
  case $\sigma$
    $s \vdash \mathbf{A}(\Phi, p)$, $p \in \mathscr{A}$ or $p = A\phi$ or $p = E\phi$
      subg$\leftarrow$ $\{s \vdash p \vee A(\Phi)\}$
   | $(R3)$, $(R4)$, $(R5)$ , $(R6)$ $\Longrightarrow$ (as usual)
   | $s \vdash A(X\phi_1,...,X\phi_n)$ :
      subg$\leftarrow \{s' \vdash A(\phi_1,...\phi_n) \mid s' \in \mathbf{succ}_L(s)\}$
      tosend$\leftarrow \{s' \vdash A(\phi_1,...\phi_n) \mid s' \in \mathbf{succ}_R(s)\}$
      $\sigma'$.pred $\leftarrow \sigma'$.pred$\cup\{\sigma\}$ $\forall \sigma' \in$ tosend
      if subg=$\emptyset$ $\wedge$ tosend $\neq \emptyset$
         subg$\leftarrow \{\bot\}$
      snd_todo$\leftarrow$ snd_todo$\cup$tosend $(R7)$
      if subg $\neq$ {True}
        for all $\sigma' \in$ subg
          $\sigma'$.pred $\leftarrow \sigma'$.pred$\cup\{\sigma\}$
  return subg
```

Fig. 18 Main procedure of the breadth CTL* model checking algorithm

the algorithm explores the sub-parts of the proof structures for the received assertions. All these works are done until the initial assertion (of the first session) gets its answer. In the case of a flaw, we rebuild the trace as for LTL checking. This requires a minor change in the global exchange function which also sends answers and globally compute if one processor has finally reached an answer for σ_0.

We thus also modify the function `subgoals` (see Fig. 18) to take into account the management of the sends, like in our algorithm for LTL checking. Also, we add arcs between the assertions, via the field `.pred` for each assertion to know its parents, which implicitly gives the graph of the proof structure. We will use them to backtrack the answers. The function `call_ctlstar` is modified consequently to manage field `.pred`.

The difficulty in this algorithm is to correctly manage the answers. Indeed, we do not know the answer to an assertion when we compute the validity of $s \vDash \phi$ or when it has been send to another processor. Thus, we need to modify backtracking when an answer is unknown by considering a third possibility of answer: $\bot$, the case when we cannot conclude. This way, the LTL session is paused until an actual Boolean answer is computed, mainly in the next slice (and thus in the next super-step). This is illustrated in Fig. 19 where we have classes of assertions among which some need to start another LTL session (in grey). But in the same classes, we also need to continue the SCC decomposition of the proof structure to keep our slice progression. The development of a new LTL session means that we initiate the generation of a new proof structure (LTL session) for the checking of an assertion (rules R1/R2). We thus start the new session together with the other sessions whose exploration is in progress, which increases the parallelism.

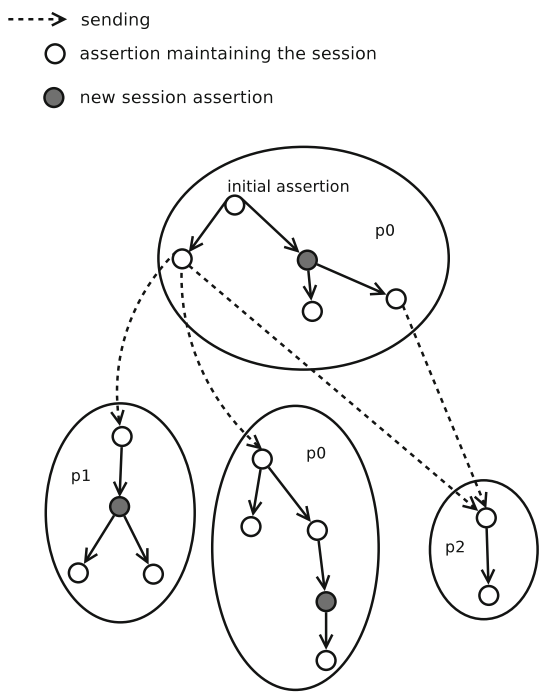


Fig. 19 Breath LTL session generation

5.3.2 Iterative CTL* *decomposition*

The backtracking between the LTL sessions and CTL* is performed using two new fields: `.parentctl*` and `.parentltl`. Each sent assertion has its fields `.parentltl` and `.parentctl*` set to $\perp$ since these assertions are not called by their parents, in the sense that their parents do not put them onto the stack `todo` of assertions awaiting for exploration. Note also that procedure `call_ctlstar` can call procedure `call_ltl` (line 19) of Fig. 20 which corresponds to starting another LTL session.

We modify the functions `call_ctlstar` and `up_ctlstar` accordingly by adding an additional field for each disjunctive and conjunctive assertion: `.wait`—see Fig. 20. Initially `.wait` is a set containing the two children of the assertion, like the field `.children`. If the children of a conjunctive or disjunctive assertion return an answer equal to $\perp$, i.e. each one has an unknown answer, then the child assertion will be removed from the field `.children` but retained in the field `.wait` so we know that this assertion has not its answer yet. This trick provides us the answer (possibly $\perp$) for the parent assertion.

Take for example the assertion $\sigma \vdash \phi_1 \vee \phi_2$ which has for children $\sigma_1 \vdash \phi_1$ and $\sigma_2 \vdash \phi_1$. Initially, σ`.children` = σ`.wait` = $\{\sigma_1, \sigma_2\}$. Assume that σ first calls σ_1, then σ_1 is removed from field σ`.children` but is kept in σ`.wait`. Field σ`.wait` will contain the children assertions for which the answer is not yet known.

```
1  def call_ctlstar($\sigma$) is
2    if $\sigma.V$
3      return $\sigma$.flag
4    else
5      $\sigma.V \leftarrow$ True
6      case $\sigma$
7        | $s \vdash p$ where $p \in \{a, \neg a\}$, $a \in \mathbf{A}$ :
8          $\sigma$.flag $\leftarrow s \models p$
9          ret_ctlstar($\sigma$)
10       | $s \vdash \phi_1 \wedge \phi_2$ :
11         $\sigma_1 \leftarrow s \vdash \phi_1$
12         $\sigma_2 \leftarrow s \vdash \phi_2$
13         $\sigma_1$.pred $\leftarrow \sigma_1$.pred$\cup\{\sigma\}$
14         $\sigma_2$.pred $\leftarrow \sigma_2$.pred$\cup\{\sigma\}$
15         $\sigma$.wait $\leftarrow \sigma$.children $\leftarrow \{\sigma_1, \sigma_2\}$
16         loop_ctlstar($\sigma$)
17       | $s \vdash \phi_1 \vee \phi_2$ : (as for $\wedge$ case)
18       | $s \vdash \mathbf{A}(\phi)$ :
19         call_ltl($\sigma$)
20       | $s \vdash \mathbf{E}(\phi)$ :
21         $\sigma_1 \leftarrow s \vdash$neg($\mathbf{E}\phi$)
22         $\sigma_1$.pred $\leftarrow \sigma_1$.pred$\cup\{\sigma\}$
23         $\sigma$.children $\leftarrow \{\sigma_1\}$
24         loop_ctlstar($\sigma$)

1  def loop_ctlstar($\sigma$) is
2    if $\sigma$.children $\neq \emptyset$
3      child $\leftarrow \sigma$.children.pop()
4      child.parentCTL$*$ $\leftarrow \sigma$
5      todo.push(child)
6    else
7      ret_ctlstar($\sigma$)

1  def ret_ctlstar($\sigma$) is
2    if $\sigma$.parentCTL$*$ $\neq \perp$
3      up_ctlstar($\sigma$.parentCTL$*$, $\sigma$)
4    elif $\sigma$.parentLTL $\neq \perp$
5      ret_ltl($\sigma$)
6    else
7      ret_trace($\sigma$)

1  def up_ctlstar($\sigma$,child) is
2    case $\sigma$
3      | $s \vdash \phi_1 \wedge \phi_2$ :
4        if child.flag = True
5          $\sigma$.wait.pop(child)
6          if $\sigma$.wait = $\emptyset$
7            $\sigma$.flag = True
8            ret_ctlstar($\sigma$)
9          else
10           loop_ctlstar($\sigma$)
11       elif child.flag = False
12         $\sigma$.wait = $\emptyset$
13         $\sigma$.flag $\leftarrow$ False
14         ret_ctlstar($\sigma$)
15       else
16         if $\sigma$.children = $\emptyset$
17           $\sigma$.flag = $\perp$
18           ret_ctlstar($\sigma$)
19         else
20           loop_ctlstar($\sigma$)
21     | $s \vdash \phi_1 \vee \phi_2$ : (as for $\wedge$ case)
22     | $s \vdash \mathbf{A}\phi$ :
23       $\sigma$.flag $\leftarrow$ flag
24       ret_ctlstar($\sigma$)
25     | $s \vdash \mathbf{E}\phi$ :
26       $\sigma$.flag = not child.flag
27       ret_ctlstar($\sigma$)
```

Fig. 20 CTL* decomposition part for the breath BSP CTL* model checking algorithm

After some computation, the answer for σ_1 is returned, say $\perp$. Therefore, we cannot conclude about σ. Assume now that σ now calls σ_2. σ_1 is thus removed from field σ`.children`. After some computation, the answer for σ_2 is returned, say **`True`**. σ_2 is thus removed from field `.wait`, because its answer is now known. But the field `.wait` of σ, containing σ_1 ensures that we can do not conclude, we first have to wait for the answer about σ_1. The procedures work as follows:

- `call_ctlstar` decomposes the assertions, builds the graph of calls, adds the children into field `.wait` and finally calls `loop_ctlstar` (lines 16 and 24 to continue to compute over those assertions) or `ret_ctlstar` (we have an answer about the assertion) if it is an atomic proposition (line 8);
- `loop_ctlstar` processes the children by putting them (lines 3–5) in the set of assertions to process (`todo`), or finishes the computation by a call (line 7) to `ret_ctlstar` if all the children have been processed;

- `ret_ctlstar` returns an answer to the appropriate parent if there is one, otherwise,the answer is backtracked using `ret_trace` (line 7) to all the assertions that expect it (even on other machines by putting the answer in `snd_back`);
- `up_ctlstar` computes the answer of an assertion with respect to the answer for its children, possibly concluding even if there are still answers awaited in field `.wait`. For instance, for logical operator $\wedge$, if field `.flag` of one of the children is **`False`** (line 11) then the assertion is invalid regardless of other answers that could come later, and so, we can backtrack this new answer by a call to `ret_ctlstar` (line 14). However, if the answer for a child is **`True`**, we have to wait for other answers, until `.wait` is empty which means that all the children answers have been **`True`**.

Note that for procedure `call_ctlstar` and `up_ctlstar`, the answer for quantifier **E** is the opposite as for **A** (we use function `neg`). The iterative procedures `all_ltl` `loop_ltl` , `ret_ltl` and `up_ltl` for SCC decomposition (LTL sessions) and `up_trace`, `ret_trace` (for backtracking) are also modified accordingly to take into account the new fields and are fully described and available in [26].

5.3.3 Sweep-line technique

The previous sweep-line strategy cannot work directly here because some assertions do not have their answers (equal to $\perp$) during a slice. So, we cannot dump them when changing slice. To adapt to this new situation and be able to dump assertions that are no longer needed (i.e. those for which we have the answers and that belong to a previous slice), we use a variable `CACHE` that contains all the assertions. At each end of the treatment of a session, we iterate on `CACHE` to dump all the unnecessary assertions, thus freeing memory for the next sessions. This methods avoids a complex traversal of the proof structures and can be compared to a garbage collection.

5.4 Experiments

To evaluate our two algorithms in PYTHON/SNAKES, we have tested two formulas: the first one is the LTL formula [1] for secrecy, whereas the second one is the CTL formula for fairness [29] (as presented above). As previously, the formulas globally hold so that the whole proof structure is computed.

In Fig. 21, we give the speedup of the two latter algorithms ("Naive CTL*" and "Pure Breadth") for three different protocols and for the two formula (as previously, results for Yahalom cannot be computed with low number of processors so we have not speedup to show). As we could expect, the naive algorithm scales less for both formula. Note that for Kao–Chow, both algorithms do not scale well. This is mainly due to a lack of possible attacks which implies less classes of states: executions are almost not branching and so the protocol provides very few intrinsic parallelism.

Figure 22 shows the execution times for our two formulas for each protocol, using 32 processors. In the figure, the total execution time is split into three parts: the computation time (black) that essentially corresponds to the computation of successful SCC of the proof structures on each processor; the global and thus collective communication time (grey) that corresponds to assertions exchange; the waiting time, i.e. latencies

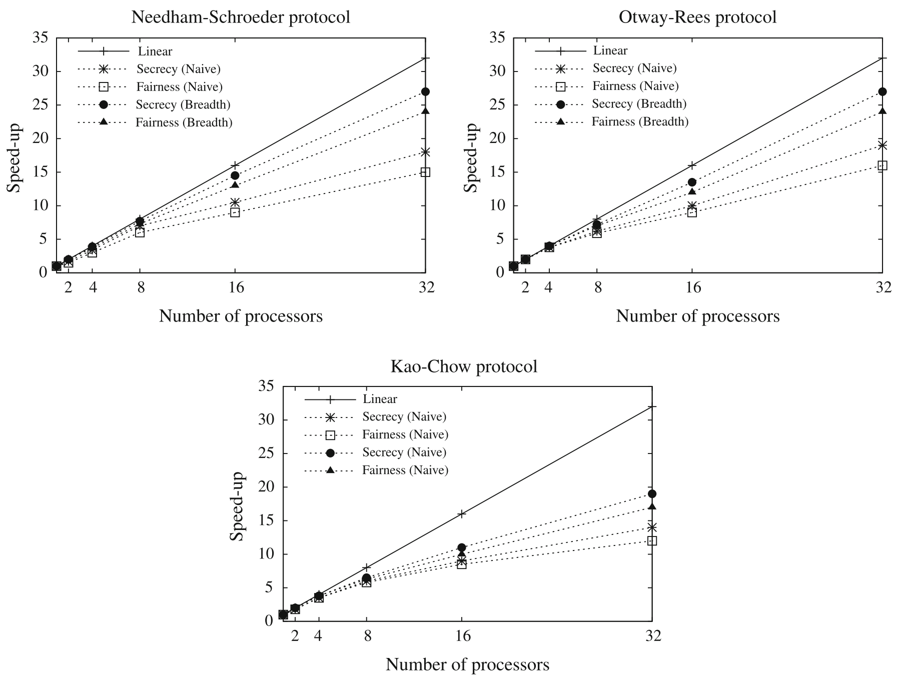


Fig. 21 Speedup results for three of the protocols

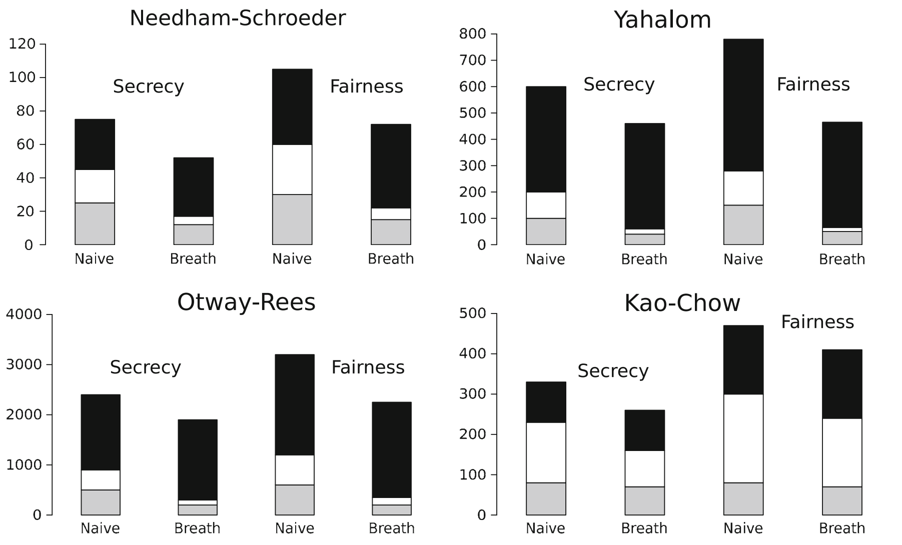


Fig. 22 Timing of the two algorithms ("Naive CTL*" and "Pure Breadth") with respect to formulas Secrecy and Fairness, for the four studied protocols. Times are given in seconds and decomposed as computation time in *black*, communication time in *grey* and waiting time in *white*

(white) that occur when processors finish their computation early and are forced to wait for the others before to enter the communication phase of each super-step. We can see on these graphs that the overall performance of our "Breath" algorithm is always good compared to the naive one. As expected, the "Breath" algorithm reduce both latencies due to less super-steps and a better balance of communications—since they are more en masse. Fairness needs more computation since it is a more complicated formulae: the bigger the formulas and the model, the better is "Breath" algorithm performs.

6 Related work

6.1 Tools and methods for security protocols

Gavin Lowe has discovered the now well-known attack on the Needham–Schroeder public key protocol using the model checker FDR [36]. In spite of this, over the last two decades, a wide variety of security protocol analysis tools have been developed that are able to detect attacks on protocols or, in some cases, establish their correctness. We distinguish three classes: tools that attempt *verification* (proving a protocol correct), those that attempt *falsification* (finding flaws, i.e. counterexamples), and hybrids that attempt to provide both proofs and counterexamples. In the first category, we find the use of theorem provers [40] and dedicated tools such as PROVERIF [9] or SCYTHER [18], etc., falsification is the domain of model checking [1,6]—such as the lazy intruder of AVISPA [2].

Paper [19] presents different cases study of verifying security protocols with various standard tools. To summarise, there is currently no tool that provides all the expected requirements.

6.1.1 Theorem proving for security protocols

To the best of our knowledge, the first work using theorem proving for verifying security protocols is [40]. And now different researches have been conducted in this way. Using a theorem prover, one formalises the system (the agents running the protocol along with the attacker) as a set of possible communication traces. Afterwards, one can state and prove theorems expressing that the system has certain desirable properties, The proofs are usually carried out under strong restrictions, e.g. that all variables are strictly typed and that all keys are atomic.

The main drawback of this approach is that verification is quite time consuming and requires considerable *expertise* [3]. Moreover, theorem provers provide poor support for error detection when the protocols are flawed, even with the work on integrating automatic methods in theorem provers for security protocols as in [12].

6.1.2 Dedicated tools

The first class of tools that focus on verification typically relies on encoding protocols as *Horn clauses* and applying resolution—without termination guarantee. The most known tool is certainly PROVERIF [9]. The system can handle an unbounded number of sessions of the protocol but performs some *approximations*—e.g. on random numbers.

As a consequence, when the system claims that the protocol preserves the secrecy of a value, this is correct. This tool is thus needed when no flaw has been found in the protocol (with model checking) and one wants to have a test for an unbounded number of sessions.

Most of dedicated tools limit possible kinds of attacks or limit in their modelling language how agents can be manipulated in ad hoc protocols. The three main drawbacks of these tools are thus (1) the restricted language used for modelling the protocols; (2) the lack of building *traces* in case of a flaw (this is not the case using a model checking method); (3) the limitation of their verification to simple properties (e.g. fairness is generally not taken into account [32]) and of their models essentially limited to "ping-pong" protocols.

6.1.3 Model checking of security protocols

On the contrary, our approach is based on model checking [6] that is not tied to any particular application domain. Using CTL*, we can also express many complex properties that some dedicated tool cannot, but that also restrict our approach to finite scenario. There are many paper about model checking of security protocols and the reader can find a gentle survey in [6]. For example, in [38], the authors have used the MURPHI modelling language and different distributed model checkers for MURPHI now exist. Even if those programs would clearly outperform our prototype tool (due to the use of PYTHON), the algorithm [43] uses a naive random hash function.

For finite scenarios checking (and enumerative state-space construction), the most well-known tool is certainly AVISPA [2] that uses dedicated modelling language and algorithms. In contrast, our approach is based on a general modelling framework (algebras of Petri nets) with explicit state-space construction, that is not tight to any particular application domain. Using PYTHON in our implementation allows us to manipulate any kind of data structures that could be used by agents in protocols. This is a well-desired feature for complex protocols like P2P security protocols in [13]. We believe that our observations and the subsequent *optimisations* are general enough to be adapted to the model checkers 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 four simple conditions (P1 to P4) that can be easily fulfilled within most concrete modelling formalisms.

6.2 Distributed and parallel model checking

6.2.1 Distributed and parallel state-space construction

The main idea of most known approaches to the distributed memory state-space generation is similar to the "naive" algorithm of [24] which usually introduces too many cross transitions. More references can be found in [22].

Examples from the literature are the various techniques used 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* from 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. Close to our hashing technique, [41] presents a hashing function that ensures that most of the successors are local: the partition function is computed by a *round-robin* on the successor states. This improves the locality of the computations but that can duplicate states. Moreover, it only works well when network communication is substantially slower than computation, which is not the case on modern parallel architectures. We can also find a balancing strategy in [15], where a balance is performed each time the system detects too many states on a node. That is not needed and would imply too much communication in our case.

In [34], a distributed state-space algorithm derived from the SPIN model checker is implemented using a *master/slave* paradigm. 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, similarly to $\mathsf{cpu}_{\mathscr{R}}$. The algorithm performs well on homogeneous networks of machines, but 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 choose correctly the fraction of states that has to be considered for hashing, while we have relied on reception locations from $\mathscr{R}$. SPIN has also been used for verifying security protocols [37].

In [39], a user-defined *abstract interpretation* is used to reduce the size of the state space, and so it allows to distribute the abstract graph; the concrete graph 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.

In [10], authors used complex *distributed file systems* or shared *databases* to optimise the sending of the states, especially when complex data structures are used internally in the states—as ours. That can improve our implementation but not the idea of the method. In [21], the authors used heuristics for the sweep-line method with the drawback that these heuristics can fail. In our case of security protocols, no such heuristic is necessary since the structured model gives the progression.

6.2.2 Distributed and parallel temporal logic verification

If model checking of LTL formula has been the more studied, works for CTL can be found in [11] and in [35] for the μ-calculus–which is more expressive than CTL*.

Close to our idea of *localising cycles*, we can cite [4] which both used partition functions that enable cycles to be local only—as for us. The limits of the method are the cost of their functions as well as the number of SCCS which is not sufficient to scale. [5] presents distributed algorithms for SCC computation. In our work, all SCCS are purely local, which is easier to handle and more efficient.

A kind of tree (*hesitant*) Büchi automata is used in [30], where parallel SCC computations are performed. The automaton is hesitant is the sense that as for rules R1 and R2, it cannot conclude and thus initiates the two possible computations. That generated what they call "games" (close to our "sessions") and the algorithm has to manage how to store partial results of games. *Shared memory* computations and heuristics are used here to simplify this management. The algorithm has also expensive management of invalid SCCS, which seems not feasible for a distributed architecture. These algorithms have also been tested to check security protocols in [29].

7 Conclusion and future work

Designing security protocols is *complex* and *error prone*: various *attacks* are reported in the literature to protocols thought to be "correct" for many years. There are now many tools that check the *security* of cryptographic protocols and *model checking* is one solution. It is mainly used to find flaws in *finite scenario* (bounded number of agents), but not to prove the correctness of a protocol. To check if scenario contains *flaw* or not, we thus propose to resort to *explicit distributed model checking*, using an algebra of coloured Petri nets to model the protocol, together with security properties that could be expressed as reachability properties, LTL, or CTL* formulas. Reachability properties lead us to construct the state space of the model (i.e. the set of its reachable states). LTL and CTL* involve the construction of the state graph (i.e. the reachable states together with the transitions from states to others) which is combined with the formula under analysis into a so-called proof structure. In both cases, *on-the-fly* analysis allows to stop states explorations as soon as a conclusion is drawn.

Using a *distributed algorithm* is a common solution to benefit from more memories and computations units. But, the critical problem of state-space construction is to determine whether a newly generated state has been explored before. In a serial implementation, this question is answered by organising known states in a specific data structure, and looking for the new states in that structure. As this is a centralised activity, a 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. After a state has been generated, it is sent to its assigned location, where a local search determines whether the state already exists. Applying this method to security protocols fails in two points. First, the number of *cross transitions* (i.e. transitions between two states assigned to distinct processors) is too high and leads to a too heavy network use. Second, memorising all of them in the *main memory* is *impossible* without crashing the whole parallel machine and is not clear when it is possible to put some states in *disk* and if *heuristics* (e.g. a caching strategy) would work well for complex protocols.

Our parallel algorithm for the state-space computation (basis of model checking) of the *finite scenario* of protocols, use the *well-structured* nature of the protocols to choose which part of the state space is really needed for the partition function and to empty the data structures in each *super-step* of the parallel computation. The state space is thus distributed in such a way that there is no *roll back* in the super-step, which allows to divise the state space into *slices* and ensures that there is no cross transitions during local computations. Our algorithms also entail automated classification of states into *classes*, and the *dynamic mapping* of classes to processors. We find that both our methods execute significantly *faster* than the traditional one and achieve a better network use, memory balance and computation time.

With these properties in mind, we have designed two algorithms for verifying temporal logical formula over finite scenario of protocols, one for LTL checking and another one for CTL* checking. Both are parallelisation of an existing algorithm based on building proof structures and computing strongly connected components (SCCS) using a Tarjan-like method. The structure of state-space exploration is thus preserved but enriched with the construction of the *proof structure* and its *on-the-fly* analysis. This allows parallel machines to apply automated reasoning techniques, to perform a formal analysis of security protocols. In the case of LTL, we have seen that no cross transition occurs within a SCC, which is crucial to conclude about formula truth value. In the case of CTL* however, local conclusions may need to be delayed until a further recursive exploration is completed, which might occur on another processor. Rather than continuing such an exploration on the same processor, which would limit parallelism, we designed a way to organise the computations so that *inconclusive* nodes in the proof structure can be kept available until a conclusion is drawn from a recursive exploration, allowing to *dump* them immediately from the main memory. This more complex bookkeeping appears necessary due to the recursive nature of CTL* checking that can be regarded as nested LTL analysis.

The fundamental message is that for parallel model checking, exploiting certain characteristics of the system and structuring the computation is essential. We have demonstrated techniques that proved the feasibility of this approach and demonstrated its potential. Key elements to our success were: (1) an automated classification that reduces cross transitions and memory use and growth locality of the computations; (2) using global barriers (which is a low-overhead method) to compute a global remapping and thus balancing workload and achieved a good scalability for the state-space generation of security protocols; (3) careful extension of this state-space algorithm to handle the case of LTL first, then CTL*.

Future works will be dedicated to build an efficient implementation from our prototypes. Using it, we would like to run benchmarks to compare our approach with existing tools such as AVISPA, which is currently meaningless due to our PYTHON implementations. Using BSP-PYTHON is good for a short development cycle of the prototypes but that generates inefficient parallel programs. We would like also to test our algorithms on parallel computers with more processors to confirm the scalability observed on 40 processors. More practically: we would like to have a tool able to translate HSPSL models [1] (a standard language for describing security protocols) to ABCD ones since HSPSL is mainly used by the community.

Finally, we would like to generalise our present results by extending its application domain to more *complex protocols* with branching and looping structures, as well as complex data types manipulations as in protocols for secure storage distributed through peer-to-peer communication [13], secured routing protocols [17], etc.

References

1. Armando A, Carbone R, Compagna L (2009) Ltl model checking for security protocols. Appl Non Class Log 19(4):403–429
2. Armando A, et al (2005) The AVISPA tool for the automated validation of Internet security protocols and applications. In: Etessami K, Rajamani SK (eds) Proceedings of Computer Aided Verification (CAV), LNCS. Springer, vol 3576, pp 281–285
3. Backes M, Unruh D (2008) Theory and application of cryptology and information security (ASIACRYPT), LNCS. In: Pieprzyk J (ed) Limits of constructive security proofs. Springer, New York, pp 290–307

4. Barnat J, Brim L, Cëerná I (2002) Property driven distribution of nested dfs. In: Leuschel M, Ultes-Nitsche U (eds) Workshop on verification and computational logic (VCL), vol DSSE-TR-2002-5, pp 1–10. Department of Electronics and Computer Science, University of Southampton (DSSE), UK, Technical Report
5. Barnat J, Chaloupka J, Pol JVD (2011) Distributed algorithms for SCC decomposition. J Log Comput 21(1):23–44
6. Basin D, Cremers C, Meadows C (2011) Model checking security protocols, chap 24. Springer, New York
7. Bhat G, Cleaveland R, Grumberg O (1995) Efficient on-the-fly model checking for ctl*. In: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS). IEEE Computer Society, pp 388–398
8. Bisseling RH (2004) Parallel scientific computation. A structured approach using BSP and MPI. Oxford University Press, Oxford
9. Blanchet B (2001) An efficient cryptographic protocol verifier based on Prolog rules. In: IEEE CSFW'01. IEEE Computer Society
10. Blom S, Lisser B, van de Pol J, Weber M (2011) A database approach to distributed state-space generation. J Log Comput 21(1):45–62
11. Boukala MC, Petrucci L (2012) Distributed model-checking and counterexample search for ctl logic. IJCCBS 3(1/2):44–59
12. Brucker AD, Mödersheim S (2009) Integrating automated and interactive protocol verification. In: Formal Aspects in Security and Trust (FAST), LNCS, vol 5983. Springer, New York, pp 248–262
13. Chaou S, Utard G, Pommereau F (2011) Evaluating a peer-to-peer storage system in presence of malicious peers. In: Smari WW, McIntire JP (eds) High performance computing and simulation (HPCS). IEEE, pp 419–426
14. Christensen S, Kristensen LM, Mailund T (2001) A sweep-line method for state space exploration. In: Margaria T, Yi W (eds) Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol 2031. Springer, New York, pp 450–464
15. Ciardo G, Gluckman J, Nicol DM (1998) Distributed state space generation of discrete-state stochastic models. INFORMS J Computg 10(1):82–93
16. Comon-Lundh H, Cortier V (2011) How to prove security of communication protocols? a discussion on the soundness of formal models w.r.t. computational ones. In: STACS, pp 29–44
17. Cortier V, Degrieck J, Delaune S (2012) Principles of security and trust (POST), LNCS. In: Degano P, Guttman JD (eds) Analysing routing protocols: four nodes topologies are sufficient. Springer, New York, pp 30–50
18. Cremers CJF (2006) Scyther-semantics and verification of security protocols. Ph.D. thesis, Technische Universiteit Eindhoven
19. Cremers JF, Lafourcade P, Nadeau P (2009) Comparing state spaces in automatic security protocol analysis. In: Formal to Practical Security, LNCS, vol 5458. Springer, New York, pp 70–94
20. Dolev D, Yao AC (1983) On the security of public key protocols. IEEE Trans Inf Theory 29(2):198–208
21. Evangelista S, Kristensen LM (2012) Application and theory of petri nets, LNCS. In: Haddad S, Pomello L (eds) Hybrid on-the-fly ltl model checking with the sweep-line method. Springer, New York, pp 248–267
22. Ezekiel J, Lüttgen G (2008) Measuring and evaluating parallel state-space exploration algorithms. Electron Notes Theor Comput Sci 198(1):47–61
23. Fokkink W, Dashti MT, Wijs A (2010) Conference on Application of Concurrency to System Design (ACSD). In: Gomes L, Khomenko V, Fernandes JM (eds) Partial order reduction for branching security protocols. IEEE Computer Society, Portugal, pp 191–200
24. Garavel H, Mateescu R, Smarandache IM (2001) Proceedings of SPIN, LNCS. In: Dwyer MB (ed) Parallel state space construction for model-checking. Springer, New York, pp 217–234
25. Goranko V, Kyrilov A, Shkatov D (2010) Tableau tool for testing satisfiability in ltl: implementation and experimental analysis. Electron Notes Theor Comput Sci 262:113–125
26. Guedj M (2012) Bsp algorithms for ltl & ctl* model checking of security protocols. Ph.D. thesis, University of Paris-Est
27. Hinsen K (2007) Parallel scripting with Python. Comput Sci Eng 9(6):82–89
28. Holzmann G, Peled D, Yannakakis M (1996) The spin verification system. On nested depth first search (extended abstract). American Mathematical Society, USA, pp 23–32

29. Inggs C, Barringer H, Nenadic A, Zhang N (2004) Model checking a security protocol. In: Southern African Telecommunications Network and Applications Conference (SATNAC)
30. Inggs CP, Barringer H (2006) Ctl* model checking on a shared-memory architecture. Form Methods Syst Des 29(2):135–155
31. Losup A, Sonmez O, Anoep S, Epema D (2008) The performance of bags-of-tasks in large-scale distributed systems. In: Symposium on High performance distributed computing (HPDC). ACM, USA, pp 97–108
32. Kremer S, Markowitch O, Zhou J (2002) An intensive survey of fair non-repudiation protocols. Comput Commun 25(17):1606–1621
33. Kumar R, Mercer EG (2005) Load balancing parallel explicit state model checking. In: ENTCS, vol 128. Elsevier, Amsterdam, pp 19–34
34. Lerda F, Sista R (1999) Proceedings of SPIN, no. 1680 in LNCS. In: Dams D, Gerth R, Leue S, Massink M (eds) Distributed-memory model checking with SPIN. Springer, New York, pp 22–39
35. Leucker M, Somla R, Weber M (2003) Parallel model checking for ltl, ctl*, l. Electron Notes Theor Comput Sci 1–1
36. Margaria T, Steffen B (eds) (1996) Tools and algorithms for construction and analysis of systems (TACAS), LNCS. Breaking and fixing the needham-schroeder public-key protocol using fdr. Springer, New York, pp 147–166
37. Maggi P, Sisto R (2002) Model Checking of Software (SPIN), LNCS. In: Bosnacki D, Leue S (eds) Using spin to verify security properties of cryptographic protocols. Springer, New York, pp 187–204
38. Mitchell JC, Mitchell M, Stern U (1997) Automated analysis of cryptographic protocols using murphi. In: IEEE Symposium on Security and Privacy. IEEE Computer Society, pp 141–151
39. Orzan S, van de Pol J, Espada M (2005) A state space distributed policy based on abstract interpretation. In: ENTCS, vol 128. Elsevier, Amsterdam, pp 35–45
40. Paulson LC (1998) The inductive approach to verifying cryptographic protocols. J Comput Secur 6(1–2):85–128
41. Petcu D (2003) Parallel explicit state reachability analysis and state space construction. In: Proceedings of ISPDC. IEEE Computer Society, pp 207–214
42. Pommereau F (2010) Algebras of coloured petri nets. Lambert Academic Publisher, Germany (ISBN 978-3-8433-6113-2)
43. Stern U, Dill DL (2001) Parallelizing the murj verifier. Form Methods Syst Des 18(2):117–129