Mechanised verification of distributed state-space algorithms for security protocols

Frédéric Gava
University of Paris-East
frederic.gava@univ-paris-est.fr

Arthur Hidalgo
University of Paris-East

Jean Fortin
University of Paris-East
jean.fortin@univ-paris-est.fr

***Abstract*—Explicit model-checking (MC) is a classical solution to find flaws in a security protocol. But it is well-known that for non trivial protocols, MC may enumerate state-spaces of astronomical sizes — the famous state-space explosion problem. Distributed model checking is a solution but complex and subject to bugs: a MC can validate a model but miss an invalid state. In this paper, we focus on using a verification condition generator that takes annotated distributed algorithms and ensures their termination and correctness. We study five algorithms (one sequential and four distributed where three of them are dedicated and optimised for security protocol) of state-space construction as a first step towards mechanised verification of distributed model-checkers.**

***Keywords*-BSP; Mechanized proof; Security protocol**

I. Introduction

Security protocols are small and standard components of systems that communicate over untrusted networks. Their relatively small size, combined with their critical role, makes them a suitable target for formal analysis [1]. Model-checking (MC) is common solution to find flaws [2]. Ideally, we would also like to have a proof of the protocol's correctness or of the attack found: generate a "certificate" [3] that can be checked later. But the generation of large discrete state spaces of some non traditional protocols [4] (especially when complex data-structures are used by the agents such as lists of trusted servers *etc.*) is such a computationally intensive activity that the use of distributed machines is desirable and is a great challenge of research.

But MCs, especially distributed ones [5], like any complex softwares are subject to bugs. And generating distributed certificates to be later machine-checked using a theorem prover such as Coq is currently not reasonable since provers are critical softwares that can not be altered without much attention. For this purpose, we proposed to prove the correctness of the distributed MC itself and not its results as "certifying MC" [3] generally does.

But unlike [6] where authors only focus on safety properties such as no overflows, no deadlocks *etc.*, we use the condition generator (VCG) Why [7] and extend the deductive verification to the correctness of the final result: has the full state-space been well computed (in parallel) without adding unknown states? We consider the mechanised verification of different annotated algorithms: a sequential one as an introduction of the methodology; the traditional distributed state space algorithm and three specialised distributed algorithms for computing the state space of security protocols [8]. All distributed algorithms used a model of parallel computation called BSP [9]. The annotated source codes are available at http://lacl.fr/gava/cert-mc.tar.gz.

II. Context and Definitions

A. Deductive verification of algorithms using Why [7]

Why is a framework for algorithms verification. Basically, it is composed of two parts: a (polymorphic first-order) logical language called Why with an infrastructure to translate it to existing theorem provers and SMT solvers; and an intermediate verification programming language called WhyML with a VCG. The examples of the standard library propose finite sets of data and several operations with their axiomatisation — which can be proved using Coq. In the logical formula, x@ is the notation for the value of x in the prestate, *i.e.* at the precondition point and x@label for the value of x at a certain point (marked by a label) of the algorithm. Mutable data types can be introduced, by means of polymorphic references: a reference r to a value of type σ has type **ref** σ, is created with the function **ref**, is accessed with !r, and assigned with r $\leftarrow$e. Algorithms are annotated using pre- and post-conditions, loop invariants, and variants to ensure termination. VCG is computed using a weakest precondition calculus and then passed to the back-end of Why to be sent to provers. Notice that in Why, sets are purely applicative and thus only a reference on a set can be modified and assigned to another set.

B. Definition of the finite state-space

Let us recall that the finite state-space construction problem is the problem of computing the explicit graph representation (also known as *Kripke structure*) of a given model from the implicit one. This graph is constructed by exploring all the states reachable through a successor function succ from an initial state s_0. Generally, during this operation, all the explored states must be kept in memory in order to avoid multiple explorations of a same state.

In the following, the algorithms only compute the state-space, noted **StSpace**. This is made without loss of generality and it is a trivial extension to compute the full Kripke structure — usually preferred for checking temporal logic formula. To represent **StSpace**, we used the following:

```
logic s0: state     logic succ: state → state set     logic StSpace: state set
axiom contain_state_space: ∀ss:state set. StSpace ⊆ ss ↔
    (s0 ∈ ss and (∀ s:state. s ∈ ss → s ∈ StSpace → succ(s) ⊆ ss))
```

i.e. defines which sets can contain the state-space. Now ss is the state-space (ss=**StSpace**) if and only if, the two following properties hold: (A) ss ⊆ **StSpace** and (B) **StSpace** ⊆ ss; that is equality of sets using extensionality.

C. *Verification of a sequential algorithm*

Fig. 1 gives a common sequential algorithm in WhyML (logical assertions are in curly brackets) using an appropriate syntax for set operations. A set called known contains all the states that have been processed and would finally contain **StSpace**. The set todo is used to hold all the states whose successors have not been constructed yet; each state s from todo is processed in turn (lines 4 and 12) and added to known (line 13) while its successors are added to todo unless they are known already — line 14. Note that the algorithm can be made strictly depth-first by choosing the most-recently discovered state (*i.e.* todo as a heap), and breadth-first by choosing the least-recently one. This has not been studied here.

We need to prove three properties regarding this code: it does not fail, it indeed computes the state-space and it terminates. The first property is immediate since the only operation that could fail is **pick** (where the precondition is "not take any element from an empty set") and this is assured by the **while**'s boolean condition. Only four invariants (lines $6-9$) are needed: (1) known and todo are subsets of **StSpace**; at the end, todo will be empty which ensures (A); (2) these sets are disjoint which ensures that only new states are added to known; (3) and (4) known is as **StSpace** and when todo will be empty, then it ensures (B). Also, the termination of this algorithm is ensured by the following variant: |**StSpace** \ known| and this variant holds at every step since the algorithm only adds a new state s since (known ∩ todo)=∅.

All the obligations produced by the VCG of WhyML are automatically discharged by a combination of SMT provers: CVC3, Z3, Simplify, Alt-Ergo, Yices and Vampire. For each prover, we give a timeout of 10 seconds. In the following table, we give the number of generated obligations and how many are discharged by the provers:

Algo/SMT	**Total**	Alt-Ergo	Simplify	Z3	CVC3	Yices	Vampire
Seq	11	2	10	11	7	3	3

One could notice that SMT solvers Simplify and Z3 give the best results. In practice, we mostly used them: Simplify is the faster and Z3 sometime verified some obligations that had not be discharged by Simplify. We also have worked with the provers as black-boxes and we have thus no explanation for this fact. It also took a day for the first author to annotate this first algorithm.

III. DISTRIBUTED STATE-SPACE CONSTRUCTION

A BSP computer is a set of uniform processor-memory pairs connected through a communication network [9]. A BSP program is executed as a sequence of *super-steps*, each

```
1  let seq_algo () =
2   let known = ref ∅ in
3   let todo = ref {s0} in
4    while todo ≠ ∅ do
5    {
6     invariant (1) (known ∪ todo) ⊆ StSpace
7          and (2) (known ∩ todo)=∅
8          and (3) s0 ∈(known ∪ todo)
9          and (4) (∀ e:state. e ∈known → succ(e) ⊆ (known ∪ todo))
10    variant |StSpace \ known|
11   }
12     let s = pick todo in
13         known←!known ⊕ s;
14         todo←!todo ∪ (succ(s) \ !known)
15   done;
16   !known
17   {result=StSpace} (* result is the value of known*)
```

Figure 1. Sequential annotated algorithm

one divided into three successive disjoint phases: (1) each processor only uses its local data to perform sequential computation 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.

A. *Deductive verification of BSP algorithms*

Our tool BSPWhy extends WhyML with BSP primitives (message passing and synchronisation) and definitions of collective operations. A special constant $nprocs$ (equal to $\mathbf{p}$ the number of processors) and a special variable my_pid (with range $0, \ldots, \mathbf{p}-1$) were also added to WhyML expressions. A special syntax for BSP annotations is also provided which is simple to use and is sufficient to express conditions in most practical programs: we add the construct $t{<}i{>}$ which denotes the value of a term t at processor id i, and $<x>$ denotes a $\mathbf{p}$-values x (represented by $fparray$, purely applicative arrays of constant size $\mathbf{p}$) that is a value on each processor by opposition to the simple notation x which means the value of x on the current processor.

We used the WhyML language as a back-end of our own BSPWhyML language. This transformation is based on the fact that, for each super-step, if we execute sequentially the code for each processor and then perform the simulation of the communications by copying the data, we have the same results as in really truly doing it in parallel. Also, when transforming a `if` or `while` structure, there is a risk that a global synchronous instruction (a collective operation) might be executed on a processor and not on the others. We generate an assertion to forbid this case, ensuring that the condition associated with the instruction will always be true on every processor at the same time and thus forbidding deadlocks. The details and some examples are available in [10].

B. *A generic BSP algorithm for state-space generation*

This sequential algorithm can be easily parallelised in a SPMD (Single Program, Multiple Data) fashion by using a partition function cpu that returns for each state a processor id, *i.e.*, the processor numbered cpu(s) is the owner of s. The idea is that each processor computes the successors

```
1  let naive_state_space () =
2    let known = ref ∅      in let todo = ref ∅ in
3    let pastsend = ref ∅   in let total = ref 1 in
4      if cpu(s0) = bsp_pid then
5        todo ←s0 ⊕ !todo;
6      while total>0 do
7        { invariant
8      (1) ⋃(<known>) ∪ ⋃(<todo>) ⊆ StSpace
9  and (2) (⋃(<known>) ∩ ⋃(<todo>))=∅
10 and (3) GoodPar(<known>) and GoodPart(<todo>)
11 and (4) (∀ i,j:int. isproc(i) → isproc(j) → total<i> = total<j>)
12 and (5) total<0> ≥ |⋃(<todo>)|
13 and (6) s0 ∈(⋃(<known>) ∪ ⋃(<todo>))
14 and (7) (∀ e:state. e ∈⋃(<known>) → succ(e) ⊆ (⋃(<known>) ∪
   ⋃(<todo>)))
15 and (8) (∀ e:state. ∀i:int. isproc(i) → e ∈known<i> → succ(e) ⊆
   (known<i> ∪ pastsend<i>))
16 and (9) ⋃(<pastsend>) ⊆ StSpace
17 and (10) (∀ i:int. isproc(i) → ∀e:state. e ∈pastsend<i> → cpu(e)≠ i)
18 and (11) ⋃(<pastsend>) ⊆ (⋃(<known>) ∪ ⋃(<todo>))
19         variant pair(total<0>,| S \ ⋃(known) |) for lexico_order
20       }
21         let tosend=(local_successors known todo pastsend) in
22           exchange todo total !known !tosend
23       done;
24       !known
25       {⋃(<result>)=StSpace and GoodPart(<result>)}
```

Figure 2. Parallel annotated algorithm

for only the states it owns. This is rendered as the BSP algorithm of Fig. 2. Sets known and todo are still used but become local to each processor and thus provide only a partial view on the ongoing computation. For lack of space, we only present the code of the main parallel loop: other functions are available in the source code.

Function local_successors computes the successors of the states in todo where each computed state that is not owned by the local processor is recorded in a set tosend together with its owner number. The set pastsend contains all the states that have been sent during the past super-steps — the past exchanges. This prevents returning a state already sent by the processors. Function (synchronous) exchange is responsible for performing the actual communications: it returns the set of received states that are not yet known locally together with the new value of $total$.

In order to terminate the algorithm, we use the additional variable total in which we count the total number of sent states. We have thus not used any complicated methods as the ones presented in [11], [12]. It can be noted that the value of total may be greater than the intended count of states in todo sets. Indeed, it may happen that two processors compute a same state owned by a third processor, in which case two states are exchanged but then only one is kept upon reception. 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.

C. Verification of this generic distributed algorithm

We use the following predicates:

- isproc(i) defines what is a valid processor id, that is 0≤ i<**nprocs**;
- ⋃(p_set) is the union of the sets of the $\mathbf{p}$-value p_set that is $\bigcup_{pid=0}^{\mathbf{p}} \text{p\_set}(pid)$;
- GoodPart(<p_set>) is used to indicate that each processor only contains the states it owns that is ∀i:int. isproc(i) → ∀s:state. s ∈p_set<i> → cpu(s)=i.

As above, we need to prove that the code does not fail, indeed computes the entire state-space and terminates. The first property is immediate since only **pick** is used as above. Absence of deadlock (the main loop contains exchange which implies a global synchronisation of all the processors) can easily be maintained using invariant (4) (line 11): total has the same value on all the processors during the entire execution of the algorithm. Let us now focus on the two other properties.

The invariants (lines 8−18) of the main parallel loop work as follow: (1) as in the sequential algorithm, we need to maintain that known (even distributed) is a subset of **StSpace** which finally ensures (A) when todo is empty; (2) as usual, the states to treat are not already known; (3) our sets are well distributed (there is no duplicate state that is, each state is only kept in a unique processor); (4) total is a global variable, we thus ensure that it is the same value on each processor; (5) ensures that no state remain in todo (to be treated) when leaving the loop since total is upper to the size of todo, total is an over-approximation of the number of sent states; (6) and (7) usually ensure property (B); (8) states in known have their successors locally present or already sent; (9) past sent states are in the state-space; (10) pastsend only contains states that are not owned by the processor and (11) all these states, that were sent, are finally received and stored by a processor.

In the post-condition (line 25), we can also ensure that the result is well distributed: the state-space is complete and each processor only contains the states it owns depending of the function "cpu".

For the local computations, the termination is ensured as in the sequential algorithm. The main loop is more subtle: total is an over-approximation and thus could be greater to 0 whereas todo empty. This happens when all the received states are already in known. The termination has thus two cases: (1) in general the set known globally (that is in the point of view of all processors) grows and we have thus the cardinal of **StSpace** minus known which is strictly decreasing; (2) if there is no state in any todo of a processor (case of the last super-step), no new states would be computed and thus total would be equal to 0 in the last stage of the main loop. We thus used a lexicographic order (this relation is well-founded ensuring termination) on the total size of the $\mathbf{p}$ values known following with total (which is the same value on each processor) when no new states are computed and thus when no state would be sent during the next super-step. Finally, one processor can have received no states during a super-step. We thus need an invariant in the local_successors for maintaining the fact that the set known potentially grows with at least the states of todo. We also maintain that if todo is empty then no state would be sent (in local_successors) and received, making total equal to 0 after the exchange function.

With some obvious axioms on the predicates, all the produced obligations are automatically discharged by a combination of the SMT solvers. In the following table, for each part of this parallel algorithm, we give the number of obligations and how many are discharged by the provers:

part/SMT	**Total**	Alt-Ergo	Simplify	Z3	CVC3	Yices	Vampire
main	106	49	90	92	0	0	81
successor	94	45	90	88	75	0	58
exchange	90	42	80	78	74	0	75

Now the combination of all provers is needed since none of them (or at least a couple of them) is able to prove all the obligations. This is certainly due to their different heuristics. We also note that Simplify and Z3 continue to remain the most efficient. It took one month for the authors to annotate this parallel algorithm.

IV. Dedicated algorithms for protocols

A. *BSP computing the state-space of security protocols [8]*

We model security protocols as a labelled transition system (LTS) where *agents* send messages over a network which contains a Dolev-Yao attacker [13]. The intruder can overhear, intercept, and synthesise any message and is only limited by the constraints of the cryptographic methods used. It is enough to assume that the following properties hold: (P1) LTS function succ can be partitioned into two successor functions $\mathsf{succ}_{\mathcal{R}}$ and $\mathsf{succ}_{\mathcal{L}}$ that correspond respectively to transitions upon which an agent (except the intruder) receives information (and stores it), and to all the other transitions; (P2) there is an initial state s_0 and there exists a function slice from states to natural numbers (a measure) such that if $s' \in \mathsf{succ}_{\mathcal{R}}(s)$ then there is no path from s' to any state s'' such that $\mathsf{slice}(s) = \mathsf{slice}(s'')$ and $\mathsf{slice}(s') = \mathsf{slice}(s) + 1$ (it is often called a sweep-line progression); (P3) there exists a function cpu from states to natural numbers (a hashing) such that for all state s if $s' \in \mathsf{succ}_{\mathcal{L}}(s)$ then $\mathsf{cpu}(s) = \mathsf{cpu}(s')$; mainly, the knowledge of the intruder is not taken into account to compute the hash of a state; (P4) if $s_1, s_2 \in \mathsf{succ}_{\mathcal{R}}(s)$ and $\mathsf{cpu}(s_1) \neq \mathsf{cpu}(s_2)$ then there is no possible path from s_1 to s_2 and *vice versa*. Based on the following properties, we have designed in [8], in an incremental manner, three different BSP algorithms for effectively computing the state space of security protocols. Only the functions local_successors and exchange have been modified in the distributed algorithms.

In the first algorithm, called "Incr", when the function local_successors is called, then all new states from $\mathsf{succ}_{\mathcal{L}}$ are added in todo (states to be proceeded) and states from $\mathsf{succ}_{\mathcal{R}}$ 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. Another difference is the forgotten variable "pastsend" since no state could be sent twice due to this order. Fig. 3 schemes this idea. In the second algorithm, called "Sweep", and using the previous

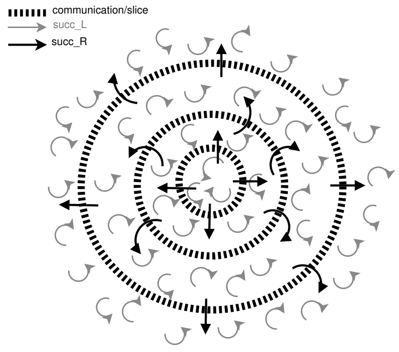

Figure 3. Scheme of the "Incr" distributed algorithm

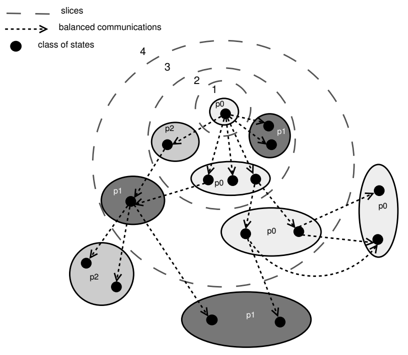

Figure 4. Scheme of the "Balance" distributed algorithm

hypothesis, at the beginning of each super-step, we also dump from the main memory all the known states because they cannot be reached anymore due to the sweep-line progression. In the third algorithm, called "Balance", states to be sent are also first balanced across the processors. Classes of states (consistent with partition function cpu) are grouped on processors so there is no possibility of duplicated computation. Fig. 4 schemes this idea. These algorithms (especially the third one) give better performances than a naive distributed one for security protocols [8]. Note that partial-order reductions [14] can also be trivially introduced.

B. *Verification of these dedicated parallel algorithms*

For all these algorithms, the termination is proved correct as above. For lack of space, we present only the differences in the main loop of the algorithms and not in local_successor and exchange — see the source code.

1) Algorithm "Incr": The invariants are the same of Fig 2 but with these changes. First, we need to forget all the behaviour about pastsend in the invariants of Fig 2 that is invariants (10), (15) and (18) since we no longer use this variable. Second, we introduce these two new invariants:

```
(12) and (∀ e:state. e ∈⋃(<known>) → slice(e)<ghost_slice)
(13) and (∀ e:state. e ∈⋃(<todo>) → slice(e)=ghost_slice)
```

We need here to introduce the ghost variable[1] ghost_slice which is incremented at each super-step and thus corresponds to the measure of progression of the protocol.

[1]Additional codes not participating in the computation but accessing the program data and allowing the verification of the original code.

Invariant (12) is needed to prove that the set known contains only states of the past slices and invariant (13) proves that in todo there are only states of the current slice.

2) Algorithm "Sweep": This algorithm works as "Incr" except that known is empty at the beginning of each super-step. We thus need to maintain this fact by using another invariant (14) $\bigcup$(known)=$\emptyset$. Note that known only grows in function local_successor.

Also, we can thus no longer use known as the variable which contains the full state-space. We thus introduce another ghost variable called ghost_known which will grow at each super-step by recovering all the states of known. In this way, in all the previous invariants, we must replace known per ghost_known for having the correctness of this algorithm.

3) Algorithm "Balance": In this algorithm, we no longer used the partition function cpu since states are distributed per class and classes are distributed across the processors using a balance. We thus need a predicate **class**(e,e') that logically define that two states belong to the same class. We also need to redefine the predicate GoodPart(<p_set>) as follow: ∀i,j:int. isproc(i) → isproc(j) → i≠ j → ∀s,s':state. s ∈p_set<i> → s' ∈p_set<j> → ¬**class**(e,e') which denotes that two states that belong to two different processors are not in the same class.

We also need to assert that after the computation of the balance (currently axiomatised since a heuristic of a NP-problem [8]), sent states respect the predicate GoodPart. We also introduce this new invariant:

```
(14) and (∀ i:int. isproc(i) → ∀e,e':state. e ∈ghost_known<i> →
class(e,e') → e' ∈ghost_known<i>)
```

which denotes that known states respect the fact that all states in a class belong to the same slice at the same processor. local_successor would verify this fact.

4) "Proof obligation results": In the following table, for each part of each parallel algorithm, we give the number of obligations and how many are discharged by the provers:

Algorithm	Part	**Total**	Alt-Ergo	Simplify	Z3	CVC3	Yices	Vampire
Incr								
	main	109	50	93	85	0	0	85
	successor	105	55	102	101	77	0	73
	exchange	32	15	28	22	19	0	27
Sweep								
	main	129	62	114	109	0	0	92
	successor	107	58	103	102	81	0	78
	exchange	31	14	29	23	21	0	28
Balance								
	main	135	71	123	119	0	0	102
	successor	113	62	111	108	87	0	81
	exchange	38	16	31	29	22	0	29

As above, only the combination of all provers is able to prove all the obligations. And few of them (not necessarily the harder) need that provers run minutes. Simplify and Z3 still remain the most efficient. An interesting point is that the second author, as a master student (when writing this article), was able to perform the job (annotate these parallel algorithms) in three months. Based on this fact, it seems conceivable that a more seasoned team in formal methods can tackle more substantial algorithms (of model-checking) in a real programming language.

V. Related works

There are many tools dedicated to the verification of security protocols: see [1] for an overview. The main idea of most known approaches to the distributed memory state space generation is similar to the naive algorithm [5]. Some developments using theorem provers are related to model checking. In [15] and [16], authors present development of BDDs and tree automata using Coq. The verification of a μ-calculus computation has also been done in Coq in [17]. A sequential state-space algorithm (with a partial order reduction) has been checked in B in [18]. Our methodology is also based on perfect cryptography. The author of [19] annotated cryptographic algorithms to mechanize the proof of their correctness.

To our knowledge, there are three existing approaches for automatically generating machine-checked protocol security proofs. The first approach is in [20] where a protocol and its properties are modeled as a set of Horn-clauses and where the certificate is machine-checked in Coq. The second [21] used the theorem prover Isabelle and computed a fixpoint of an abstraction of the transition relation of the protocol of interest — this fixpoint over-approximates the set of reachable states of the protocol. The latter [22] also used Isabelle but two strong protocol-independent invariants have been derived from an operational semantics of the protocols. We see three main drawbacks to these approaches. First, they limits (reasonably) protocols and properties that can be checked. Second, each time the proof of the tested property of the protocol need to be machine-checked; in our approach, the results of the MC are correct by construction. Third, there is currently no possibility of distributed computations for larger protocols.

VI. Conclusion and future work

Designing security protocols is complex and often error prone: various attacks are reported in the literature to protocols thought to be "correct" for many years. There are now many tools that check the security of cryptographic protocols and model-checking is one of the solution [1]. But model checkers use sophisticated algorithms that can miss a state which can be an unknown attack of the security protocol. Mechanized correctness is thus vital.

In this work, we focus on correctness of a well-known sequential algorithm for finite state-space construction (which is the basis for explicit model-checking) and on distributed ones where three are dedicated to security protocols. We annotated the algorithms for finite sets operations (available in Coq) and used the VCG Why (certifying in Coq [23]) to obtain goals that were entirely checked by SMT solvers. These goals ensure the termination of the algorithms as well as their correctness for any successor function — assumed correct and generating a finite state-space. We thus gained more confidence in the code. We also hope to have convinced that this approach is humanly feasible and applicable to true (parallel or not) model-checking algorithms.

In future works, we plan to check model-checking algorithms (in the sense of determining if a logical LTL/CTL* formula holds a model) as Tarjan like algorithms. This is challenging in general but using an appropriate VCG, we believe that a team can "quickly" do it. Compressions aspects (symmetry, partial order, *etc.*) must also be studied since they can generate wrong algorithms. The work of [18] which uses the B method could be a good basis. Furthermore, the transformation of BSPWhyML into WhyML is potentially not correct. The third authors is working on this. The successor function (computation of the transitions of the state-space) is currently an abstract function. A machine-checked proof of an implementation is needed. Finally, we are currently proving algorithms and not the effective code. Regarding the code structure, this is not really an issue and translating the resulting proof into a verification tool for true programs should be straightforward, mostly if high level data-structures are used: the Why framework allows a plugin of Frama-C (http://frama-c.com/) to generate WhyML codes from C ones — a tool for Java codes is also present.

References

[1] H. Comon-Lundh and V. Cortier, "How to prove security of communication protocols? a discussion on the soundness of formal models w.r.t. computational ones," in *STACS*, 2011, pp. 29–44.

[2] A. Armando, R. Carbone, and L. Compagna, "Ltl model checking for security protocols," *Applied Non-Classical Logics*, 2009.

[3] K. S. Namjoshi, "Certifying model checkers," in *Computer Aided Verification (CAV)*, ser. LNCS, vol. 2102. Springer, 2001, pp. 2–13.

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

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

[6] J. Sun, Y. Liu, and B. Cheng, "Model checking a model checker: A code contract combined approach," in *Formal Engineering Methods (ICFEM)*, ser. LNCS, vol. 6447. Springer, 2010, pp. 518–533.

[7] J.-C. Filliâtre, "Verifying two lines of C with Why3: an exercise in program verification," in *Verified Software: Theories, Tools and Experiments (VSTTE)*, 2012.

[8] F. Gava, M. Guedj, and F. Pommereau, "A bsp algorithm for the state space construction of security protocols," in *PDMC*. IEEE Computer Society, 2010, pp. 37–44.

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

[10] J. Fortin and F. Gava, "BSP-Why: an intermediate language for deductive verification of BSP programs," in *High-level Parallel Programming and Applications (HLPP)*. ACM Press, 2010.

[11] J. Barnat, "Distributed memory LTL model checking," Ph.D. dissertation, University of Brno, 2004.

[12] H. Garavel, R. Mateescu, and I. M. Smarandache, "Parallel state space construction for model-checking," in *Proceedings of SPIN*, ser. LNCS, M. B. Dwyer, Ed., vol. 2057. Springer, 2001, pp. 217–234.

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

[14] M. Torabi Dashti, A. Wijs, and B. Lisser, "Distributed partial order reduction for security protocols," *ENTCS*, vol. 198, pp. 93–99, 2008.

[15] K. N. Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar, "Reflecting BDDs in Coq," in *Asian Computing Science Conference (ASIAN)*, ser. LNCS, vol. 1961. Springer, 2000, pp. 162–181.

[16] X. Rival and J. Goubault-Larrecq, "Experiments with finite tree automata in coq," in *Theorem Proving in Higher Order Logics (TPHOL)*, ser. LNCS, vol. 2152. Springer, 2001, pp. 362–377.

[17] C. Sprenger, "A verified model checker for the modal μ-calculus in coq," in *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, ser. LNCS, vol. 1384. Springer, 1998, pp. 167–183.

[18] E. Turner, M. Butler, and M. Leuschel, "A refinement-based correctness proof of symmetry reduced model checking," in *Abstract State Machines, Alloy, B and Z*, ser. LNCS. Springer, 2010, pp. 231–244.

[19] J. den Hartog, "Towards mechanized correctness proofs for cryptographic algorithms: Axiomatization of a probabilistic hoare style logic," *Sci. Comput. Program.*, vol. 74, no. 1–2, pp. 52–63, 2008.

[20] J. Goubault-Larrecq, "Finite models for formal security proofs," *Journal of Computer Security*, vol. 18, no. 6, pp. 1247–1299, 2010.

[21] A. D. Brucker and S. Mödersheim, "Integrating automated and interactive protocol verification," in *Formal Aspects in Security and Trust (FAST)*, ser. LNCS, vol. 5983. Springer, 2009, pp. 248–262.

[22] S. Meier, C. J. F. Cremers, and D. A. Basin, "Strong invariants for the efficient construction of machine-checked protocol security proofs," in *Computer Security Foundations (CSF)*. IEEE Computer Society, 2010, pp. 231–245.

[23] P. Herms, "Certification of a chain for deductive program verification," in *2nd Coq Workshop, satellite of ITP'10*, Y. Bertot, Ed., 2010.