APPENDIX 1.3.2

V. Vanackère. The trust protocol analyser, automatic and efficient verification of cryptographic protocols. In *Verification Workshop - Verify02*, 2002.

# The TRUST protocol analyser
# Automatic and efficient verification of cryptographic protocols

Vincent Vanackère
Laboratoire d'Informatique Fondamentale de Marseille
Université de Provence,
39 rue Joliot-Curie, 13453, Marseille, FRANCE
*vanackere@cmi.univ-mrs.fr*

June 2002

**Abstract**

This paper presents TRUST, a verifier for cryptographic protocols. In our framework, a protocol is modeled as a finite number of processes interacting with an hostile environment; the security properties expected from the protocol are specified by inserting logical assertions on the environment knowledge in the processes.

Our analyser relies on an exact symbolic reduction method, combined with several techniques aiming to reduce the number of interleavings that have to be considered. We argue that our verifier is able to perform a full analysis on up to 3 parallel (interleaved) sessions of most protocols. Moreover, authentication and secrecy properties are specified in a very natural way, and whenever an error is found an attack against the protocol is given by our tool.

**Keywords:** cryptographic protocols, symbolic verification, state explosion problem

## 1 Introduction

The aim of this paper is to present TRUST, a verifier for cryptographic protocols relying on a symbolic reduction method introduced in [AL00] and further developed in [ALV01]. Although the symbolic reduction system allows us in theory to perform an exact analysis of an otherwise infinitely branching system, we face the same problem as in most model-checking tools: as the number of parallel threads goes up, the number of possible interleavings make the verification task harder - if not impossible - because of the state-space explosion problem. It should be noted that the verification problem we are discussing here was shown to be NP-complete [ALV01, RT01].

Our primary goals while developing this implementation were efficiency and ease of use. Most notably we use an *eager reduction* procedure in order to minimize the number of interleavings that have to be considered. Together with that, we have explored - and used - several symmetry and partial order reductions techniques. The end result is that our tool is able to handle up to 2 or even 3 parallel sessions of most protocols. We found by experience that inserting assertions within a protocol is a very natural way to specify security properties and is a good way to very quickly find a flaw. Our verifier handles nonces, symmetric and asymmetric keys; assertions consist of arbitrary boolean combinations of tests on equality, secrecy and authentication.

# 2 Theoretical background

Our formal model is presented in details in [ALV01], therefore we will only give a short presentation here.

We use the common Dolev-Yao model [DY83], where the network is under full control of an adversary that can analyse all messages exchanged and synthetize new ones. We work under the perfect encryption assumption, thus messages can be viewed as terms in a free algebra. We distinguish between basic names (agent's names, nonces, keys,...) and composed messages (pairs $< \_, \_ >$ and encrypted terms $E(\_, \_)$), with the restriction that only basic names may be used as encryption keys. The set of names is denoted by $\mathcal{N}$ and the full set of messages by $\mathcal{M}$.

## 2.1 Analysis and synthesis

The intruder capabilities are formally defined from two operators doing the analysis and synthesis on a set of messages.

We assume a (computable) relation $\mathcal{D} \subseteq \mathcal{N} \times \mathcal{N}$ with the following interpretation:

$$(C, C') \in \mathcal{D} \text{ iff messages encrypted with } C \text{ can be decrypted with } C'.$$

We define $Inv(C) = \{C' \mid (C, C') \in \mathcal{D}\}$. Further hypotheses, on the properties of $\mathcal{D}$ allow to model hashing, symmetric, and public keys. In particular: (i) for a *hashing* key $C$, $Inv(C) = \emptyset$, (ii) for a *symmetric* key $C$, $Inv(C) = \{C\}$, and (iii) for a *public* key $C$ there is another key $C'$ such that $Inv(C) = \{C'\}$ and $Inv(C') = \{C\}$.

Given a set of terms $T$ we can now define the $S$ (synthesis) and $A$ (analysis) operators as follows :

- $S(T)$ is the least set that contains $T$ and such that:

$$\begin{array}{ll} t_1, t_2 \in S(T) & \Rightarrow \langle t_1, t_2 \rangle \in S(T) \\ t_1 \in S(T), t_2 \in T \cap \mathcal{N} & \Rightarrow E(t_1, t_2) \in S(T) \ . \end{array}$$

- $A(T)$ is the least set that contains $T$ and such that:

$$\begin{array}{ll} \langle t_1, t_2 \rangle \in A(T) & \Rightarrow t_i \in A(T),\ i = 1, 2 \\ E(t_1, t_2) \in A(T), A(T) \cap Inv(t_2) \neq \emptyset & \Rightarrow t_1 \in A(T) \ . \end{array}$$

As an example, if $T = \{E(\langle A, B \rangle, K), K^{-1}\}$, then $A(T) = T \cup \{A, B, \langle A, B \rangle\}$ and *e.g.* $E(A, K^{-1}) \in S(A(T))$. Using these definitions, the set of messages that an adversary can derive from $T$ is $S(A(T))$; a trivial - but quite important - remark is that this set will be infinite as soon as $T$ is not empty.

## 2.2 Processes and configurations : semantics

In our framework, a protocol is modelled as a finite number of processes interacting with an environment. As our process syntax includes the parallel composition - commutative and associative - of two processes, we can define a configuration as a couple $(P, T)$ where $P$ is a process and $T$ a set of terms representing the current adversary knowledge, that is the initial knowledge augmented with all messages emitted by the participants of the protocol so far.

Figure 1 gives the semantic rules as a reduction system on configurations. Informally, a process can either:

(!) Write a message : the term is simply added to the environment knowledge.

(?) Read some message from the environment : this can be any message the adversary is able to build from its current knowledge.

$$
\begin{array}{lll}
(!) & (\mathsf{write}\, t.P \mid P', T) & \to (P \mid P', T \cup \{t\}) \text{ if } t \in \mathcal{M} \\
(?) & (\mathsf{read}\, x.P \mid P', T) & \to ([t/x]P \mid P', T) \text{ if } t \in S(A(T)) \\
(\mathsf{d}) & (x \leftarrow \mathsf{dec}(E(t, C), C').P \mid P', T) & \to ([t/x]P \mid P', T) \text{ if } C' \in Inv(C), t \in \mathcal{M} \\
(\mathsf{pl}) & (x \leftarrow \mathsf{proj}_i(\langle t, t' \rangle).P \mid P', T) & \to ([t/x]P \mid P', T) \text{ if } t, t' \in \mathcal{M} \\
(\mathsf{a}) & (\mathsf{assert}(\varphi).P \mid P', T) & \to \begin{cases} (P \mid P', T) & \text{if } \models_T \varphi \\ \mathsf{err} & \text{if } \not\models_T \varphi \end{cases} \\
(\mathsf{m}_1) & ([t = t]P_1, P_2 \mid P', T) & \to (P_1 \mid P', T) \text{ if } t \in \mathcal{M} \\
(\mathsf{m}_2) & ([t = t']P_1, P_2 \mid P', T) & \to (P_2 \mid P', T) \text{ if } t \neq t', t, t' \in \mathcal{M}
\end{array}
$$

Figure 1: Reduction on configurations

(d) Decrypt some (encrypted) term with a corresponding inverse key.

(pl) Perform some unpairing (the symmetric rule (pr) is not written).

($\mathsf{m}_i$) Test for equality/inequality of two messages.

(a) Check if some assertion $\varphi$ holds.

Missing from the figure is the terminated process, denoted by 0, as well as the syntax of the assertion language, that will be presented in the next section. err denotes a special configuration that can only be reached from a false assertion.

In our model, *a correct protocol is a protocol that cannot reach the* err *configuration* - or, put in other words, a protocol such that all assertions reachable from the initial configuration of the system hold.

## 2.3 Specifying security properties through assertions

The full assertion language we consider is the following:

$$\varphi ::= \quad true \mid false \parallel t = t' \mid t \neq t' \mid \mathsf{known}(t) \parallel \mathsf{secret}(t) \mid \varphi_1 \wedge \varphi_2 \parallel \varphi_1 \vee \varphi_2$$

This is equivalent to saying that we consider arbitrary boolean combinations of atomic formulas checking the equality of two messages $t = t'$ and the secrecy of a message $\mathsf{secret}(t)$ with respect to the current knowledge of the adversary. As shown in [ALV01], this language allows to easily express authentication properties such as aliveness and agreement ([Low97]).

We take as a short example the following 3 message version of the Needham-Schroeder Public Key protocol:

$$
\begin{array}{ll}
A \to B: & \{na, A\}_{Pub(B)} \\
B \to A: & \{na, nb\}_{Pub(A)} \\
A \to B: & \{nb\}_{Pub(B)}
\end{array}
$$

In our framework, the protocol can be modeled as follows (the open variables are to be instantiated by the process and peer identities before the actual symbolic reduction):

$$
\begin{array}{ll}
Init(myid, resp): & \text{fresh } na. \\
& \mathsf{write}\ E(\langle na, myid \rangle, Pub(resp)). \\
& \mathsf{read}\ e.\ \langle na', nb \rangle \leftarrow \mathsf{dec}(e, Priv(myid)).\ [na' = na]. \\
& \mathsf{write}\ E(nb, Pub(resp)). \\
& \mathsf{assert}(\mathsf{secret}(nb) \wedge \mathrm{auth}(resp, myid, na, nb)).\ 0 \\
\\
Resp(myid, init): & \mathsf{read}\ e.\ \langle na, a \rangle \leftarrow \mathsf{dec}(e, Priv(myid)).\ [a = init]. \\
& \text{fresh } nb. \\
& \mathsf{write}_{\mathsf{auth}(myid, init, na, nb)}\ E(\langle na, nb \rangle, Pub(init)). \\
& \mathsf{read}\ e'.\ [e' = E(nb, Pub(myid))]. \\
& 0
\end{array}
$$

The instruction "$\mathsf{write}_{\mathsf{auth}(msg)}t$" is some syntactic sugar to be replaced by "$\mathsf{write}\ \langle E(msg, K_{\mathsf{auth}}), t\rangle$", whereas the assertion "auth($msg$)" is a shortcut for "$\mathsf{known}(E(msg, K_{\mathsf{auth}}))$". These notations are actually supported by our tool and their usage reveals itself quite convenient in practice. In our example, the initiator specifies that at the end of its run of the protocol, the nonce $nb$ must be secret and expects an agreement with some responder on the nonces $na$ and $nb$.

### 2.4 Symbolic reduction

The main difficulty in the verification task is the fact that the input rule (?) is infinitely branching as soon as the environment is not empty. In [AL00, ALV01] it was shown that it is possible to solve this problem by using a symbolic reduction system that stores the constraints in a symbolic shape during the execution. As an example, the input rule $(\mathsf{read}\ x.P, T) \rightarrow ([t/x]P, T), t \in S(A(T))$ becomes $(\mathsf{read}\, x.P, T, E) \rightarrow (P, T, (E; x : T))$. The complete description of the symbolic reduction system can be found in [ALV01]. The main property we rely on is the fact that the symbolic reduction system is in lockstep with the ground one and provides a - sound and complete - decision procedure for processes specified using the full assertion language described in section 2.3.

## 3 Techniques for an efficient verification

Although the symbolic reduction system is satisfying from a theoretical point of view, an inherent limitation is that it does not handle iterated processes (as the general case for iterated processes is undecidable). Thus, in order to verify a protocol against replay attacks and/or parallel sessions attacks, it is quite important to handle cases where there is a finite - even if small - number of participants playing each role.

Of course, as the number of parallel threads goes up, the number of possible interleavings make the verification task harder - if not impossible - because of the state explosion problem. The main techniques that have been used/introduced in our tool are:

**Depth-first search:** this strategy brings here a lot of advantages, the main one being that no state needs to be explicitly saved (as all the necessary information indeed lies within the continuation of the program). As a consequence, the memory requirement of our tool is almost constant and quite low.

**Carefully chosen data structures:** substitutions are heavily used during the symbolic reduction process. By using a representation of terms as DAGs (directed acyclic graphs) where all variables are shared, substitutions on variables are done in $O(1)$ time. Other data structures (such as the one representing the environment knowledge) were chosen in order to allow for incremental computation whenever possible. These classical algorithmic optimizations do make a huge difference on the execution time: namely, the speed-up of our current tool w.r.t. our first prototype - mesured by the number of reductions per second - is greater than 500.

**Pruning of equivalent schedulings of parallel processes:** it is quite important not to explore all interleavings, but only those that have a significance. For this purpose, we have introduced an *eager reduction* technique that allows in some cases huge savings on the computation time. Aside from that, symmetry in the system is also exploited in order to further cut the state space.

We will now proceed in giving more details on our eager reduction procedure (section 3.1) and on the way we handle symmetry in the system (section 3.2). We then give a small note on other partial reduction techniques that may be applied.

## 3.1 Eager reduction

When verifying a system of parallel processes, only a small number of all possible interleavings need to be explored, because a lot of reduction steps are independent from each other[1]. While conceptually simple, the eager reduction procedure we introduced in our verifier has - to our knowlegde - never been described in the literature; this section is devoted to a high-level description of our method. Technical details and proofs can be found in appendix A.

In the following, we study the reduction of a configuration $(P_1 \mid \ldots \mid P_m, T)$, denoted by $(\Pi P_i, T)$. We will not allow the rewrite of $P \mid Q$ as $Q \mid P$, therefore we can define the relation $\rightarrow_x$ as a reduction on the $x$-th process of the parallel composition.

The eager reduction procedure relies on the fact that when considering a sequence of reductions $(\Pi P_i^{(1)}, T_1) \rightarrow \cdots \rightarrow (\Pi P_i^{(n)}, T_n)$ where $S(A(T_1)) = S(A(T_n))$ (i.e. the adversary knowledge does not increase during the reductions), then all reductions on the different processes are independent from each other. This leads to define a "big step" reduction *that amounts to reducing one process until it writes some term that was previously unknown to the environment*, thus we define the algorithm for an eager reduction as follows:

**Algorithm 3.1** *Step of eager reduction of* $(\Pi P_i, T)$*:*

1. *Choose* $j \in [1, n]$.
2. $c := (\Pi P_i, T)$
3. *Choose* $c'$ *such that* $c \rightarrow_j c'$.
4. *If* $c' \equiv (\Pi P_i', T')$ *and* $S(A(T')) = S(A(T))$ *then* $\{\ c := c'$ *; go to step* 3 $\}$ *else return c'*

A more formal definition, together with a proof of correctness and completeness, is given in appendix A.

### From ground eager reduction to symbolic eager reduction

We stress on the fact that although the eager reduction procedure has been described and proved here only on the ground reduction system, our verifier in fact relies on the symbolic counterpart of it. The *symbolic eager reduction procedure* matches closely the ground one, the only difference being that we (symbolically) reduce a process until it reaches error or writes a term *symbolically unknown* to the environment. Completeness of the symbolic eager reduction procedure follows from the completeness of the ground reduction (but is beyond the scope of this paper).

## 3.2 Exploiting symmetry

When studying several parallel sessions of protocols, it is useful to define protocol *roles*, which are parametric processes. All parameters will range over a finite set of principals names $\{\mathsf{Id}_0, \ldots, \mathsf{Id}_n\}$. In our verifier, the identifier $\mathsf{Id}_0$ is reserved to name a compromised participant, whereas all the names $\{\mathsf{Id}_1, \ldots, \mathsf{Id}_n\}$ are supposed to play a symmetric role in the protocol: then, we instanciate the parameters using basic injective renaming in order to generate all possible cases.

As a consequence of the completeness of eager reduction, we only need to consider "eager traces"; therefore, whenever some role is involved in a reduction to error, there is one process among those of that role that will do a step of eager ground reduction at first. Thus we can start the reduction by using only one process of each role, and *add another process of some role only after the last introduced process of the same role has performed a full step of eager reduction.* Although this may look simplistic, this allows a very important reduction in the number of states having to be explored, even when considering only 2 parallel sessions of a protocol.

[1] As a trivial example, consider two processes in parallel, one performing a decryption, and the other one an equality test: the order in which the two reduction steps are done does not affect at all the reachability of an error.

### 3.3 Going further...

We have also investigated some more advanced partial order techniques in order to further reduce the size of the state-space to be explored: it is namely possible, at the symbolic level, to detect that some eager reduction step was indeed independent from a previous one in the same trace. Then we can restrict the search to only explore traces that are in some (lexicographical) normal form (see [DM96]). Unfortunately, the proof of completeness for these methods become quite involved, and the gain observed in practice was not as important as expected: further investigation in this area is still needed.

## 4 Experimental results

TRUST was written in OCAML, and the syntax it accepts is very close to the one of the example from section 2.3 (see appendix B for a real example). This section provides some experimental results for our tool. Reassuring is the fact that our tool successfully found all known flaws on all protocol we have tried so far - even thoses the author was not yet aware of...

### Benchmarks

Figure 2 gives some figures for the full analysis of some typical protocols. For each protocol, we detail the number of roles involved and give the time to do a full search depending on the number of parallel (interleaved) sessions. In that benchmark, all roles parameters ranged over a set of 3 names $\{\mathsf{Id}_0, \mathsf{Id}_1, \mathsf{Id}_2\}$, $\mathsf{Id}_0$ being the name of a compromised principal whose private keys were initially known by the environment. All measures were done on a Pentium III at 733MHz, on which the tool performs more than 750.000 *basic* reductions per second. The total time spent is more or less proportional to the number of reductions done and, for instance, when verifying 3 interleaved sessions of the Needham-Schroeder protocol (with a key server), the verifier indeed performs around 88.000.000 reductions, checking more than 2.900.000 assertions.

| Protocol | # roles | # sessions | time |
|---|---|---|---|
| yahalom | 3 | 1 | $< 0.01s$ |
| yahalom | 3 | 2 | $12s$ |
| needham-schroeder | 2 | 3 | $0.50s$ |
| needham-schroeder | 2 | 4 | $22s$ |
| needham-schroeder (with a key server) | 3 | 2 | $0.11s$ |
| needham-schroeder (with a key server) | 3 | 3 | $115s$ |
| otway-rees | 3 | 1 | $< 0.01s$ |
| otway-rees | 3 | 2 | $1.90s$ |
| otway-rees | 3 | 3 | $1940s$ |
| kerberos v5 | 4 | 1 | $< 0.01s$ |
| kerberos v5 | 4 | 2 | $15s$ |
| kerberos v5 | 4 | 3 | $\approx 3d$ |

Figure 2: Times for the analysis of various protocols

Of course, we do not avoid the state explosion problem, but nevertheless the verification task stays practical up to at least 2 or 3 parallel sessions for all the protocols we have tried so far. Moreover, an interesting feature is that the memory usage of our analyser is almost constant and quite small (around 1MByte, for all protocols tested so far).

**Remark on the eager reduction:** it should be noted that, depending on the protocol, experimental results have shown that our eager reduction procedure - compared to the more classical input/output interleaving semantics - gives improvements ranging from a factor of 2 to more than 100...

### Finding attacks...

Here follows an example of an attack as reported by our tool. This particular one was on a (bad) variant of the Otway-Rees protocol introduced in [Pau97], whose full specification is given in appendix B:

```
0:Init(Id1,Id2) sends <N1,Id1,Id2,Crypt(<N1,Id1,Id2>,K(Id1))>

1:Resp(Id1,Id0) gets <na,Id0,Id1,e>
1:Resp(Id1,Id0) sends <na,Id0,Id1,e,N2,Crypt(<na,Id0,Id1>,K(Id1))>

2:Serv(Id0,Id1) gets <na,Id0,Id1,Crypt(<na1,Id0,Id1>,K(Id0)),N1,Crypt(<na,Id0,Id1>,K(Id1))>
2:Serv(Id0,Id1) sends <na,Crypt(<na,N3>,K(Id0)),Crypt(<N1,N3>,K(Id1))>

0:Init(Id1,Id2) gets <N1,Crypt(<N1,N3>,K(Id1))>
0:Init(Id1,Id2) sends Crypt(N4,N3)

0:Init(Id1,Id2) assert (Id2=Id0 or secret(N4))
```

A short explanation of the above example follows: $\mathsf{Id}_0$ is the identity of a compromised principal whose key $K(\mathsf{Id}_0)$ is initially known by the environment, and the initiator makes the (false) assertion that either it wanted to communicate with $\mathsf{Id}_0$ (in that particular trace leading to error, the initiator has identity $\mathsf{Id}_1$ and wants to communicate with $\mathsf{Id}_2$), or the data it sent at the last step must stay secret. This is actually not the case as clearly shown by the given attack.

It should be noted that by directly checking the secrecy of the key that the initiator gets at the end of its protocol run, we get the following - much shorter - error:

```
0:Init(Id1,Id2) sends <N1,Id1,Id2,Crypt(<N1,Id1,Id2>,K(Id1))>

0:Init(Id1,Id2) gets <N1,Crypt(<N1,Id1,Id2>,K(Id1))>
0:Init(Id1,Id2) assert (Id2=Id0 or secret(<Id1,Id2>))
```

This is a typical example of a type-flaw attack; although complex keys are not directly handled by our tool (namely, in our model, the pair <Id1,Id2> cannot be used as a valid encryption key), it is nevertheless possible to find some of those attacks with our tool.

# 5 Conclusion

We have presented the TRUST protocol analyser, a fully automatic verifier for cryptographic protocols. Our tool relies on a sound and complete symbolic reduction procedure: protocols are specified by the use of logical assertions on secrecy and authentication, and whenever an assertion is found to be invalid, an attack against the protocol is given. Our personal experience is that the description and specification of protocols using roles (parametric processes) and assertions is manageable even for non specialists, and is an easy way to find flaws in the protocols.

TRUST makes use of several techniques in order to alleviate the state space explosion problem. Most notably, it takes advantage of an eager reduction procedure, together with some basic symmetry reduction techniques. Experimental results show that - although the verification problem is actually NP-hard - our tool is able to handle efficiently 2 or even 3 interleaved sessions of most protocols from the literature.

As a sidenote, we believe that the idea behind our eager reduction procedure is simple and general enough to easily be adapted to other verification techniques such as those relying on tree automatas [Mon99, Gou00].

More information on our tool can be found at [Trust]. We are currently working on extending the symbolic decision method to particular cases when some processes - like a key server - can be iterated.

# References


[AL00] R. Amadio and D. Lugiez. On the reachability problem in cryptographic protocols. In *Proc. CONCUR00, Springer LNCS 1877*, 2000. Also RR-INRIA 3915.

[ALV01] R. Amadio, D. Lugiez and V. Vanackère. On the symbolic reduction of processes with cryptographic functions. RR-INRIA 4147, March 2001. To appear in *Theoretical Computer Science.*

[DM96] V. Diekert and Y. Métivier. Partial commutation and traces. In G. Rozenberg and A. Salomaa, editors, Handbook of Formal Languages, Vol. 3, Beyond Words, pages 457–534. Springer-Verlag, Berlin, 1997

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. on Information Theory*, 29(2):198–208, 1983.

[Gou00] J. Goubault. A method for automatic cryptographic protocol verification. In *Proc. FMPPTA, Springer-Verlag*, 2000.

[Hui99] A. Huima. Efficient infinite-state analysis of security protocols. In *Proc. Formal methods and security protocols, FLOC Workshop, Trento*, 1999.

[Low97] G. Lowe. A hierarchy of authentication specifications. In *Proc. 10th IEEE Computer Security Foundations Workshop*, 1997.

[Mon99] D. Monniaux. Abstracting cryptographic protocols with tree automata. In *Proc. Static Analysis Symposium, Springer LNCS*, 1999.

[Pau97] L. Paulson. Proving properties of security protocols by induction. In *Proc. IEEE Computer Security Foundations Workshop*, 1997.

[RT01] M. Rusinowitch and M. Turuani Protocol insecurity with finite number of sessions is NP-complete. RR INRIA 4134, March 2001.

[Trust] http://www.cmi.univ-mrs.fr/~vvanacke/trust/

# A Appendix

## Eager reduction : proof of correctness and completeness

For a ground configuration $k \equiv (\Pi P_i, T)$, we define $\mu(k) = S(A(T))$. $\mu(err) = \emptyset$. We note $k(j) = P_j$.

We will first state the main lemma on which all the eager reduction process is based :

**Lemma A.1** *If $k_1 \rightarrow_i k_2 \rightarrow_j k_3$ and $\mu(k_1) = \mu(k_2)$, $k_3 \neq err$ then $\exists k_2'(k_1 \rightarrow_j k_2' \rightarrow_i k_3)$ and $\mu(k_2') = \mu(k_3)$.*

PROOF. We assume $i \neq j$ (else the result is trivial) and do a basic case analysis on the rules used to reduce $P_i$ and $P_j$. All rules but (?), (a) and (!) do not depend at all from the environment nor modify it and thus the result holds whenever $\rightarrow_i \notin \{(\mathsf{a}), (?), (!)\}$ or $\rightarrow_j \notin \{(\mathsf{a}), (?), (!)\}$. On the 9 cases remaining, we can distinguish 4 relevant sub-cases by denoting $(\mathsf{r_i}) \in \{(\mathsf{a}), (?)\}$ :

1. $k_1 \xrightarrow{\mathsf{r}_1}_i k_2 \xrightarrow{\mathsf{r}_2}_j k_3$
2. $k_1 \xrightarrow{!}_i k_2 \xrightarrow{!}_j k_3$
3. $k_1 \xrightarrow{\mathsf{r}}_i k_2 \xrightarrow{!}_j k_3$
4. $k_1 \xrightarrow{!}_i k_2 \xrightarrow{\mathsf{r}}_j k_3$

Cases (1), (2) and (3) are straightforward. Note that case (3) when $\mathsf{r} = (?)$ is folklore and used very broadly in the literature. Case (4) is where the eager reduction procedure will take advantage: namely we can perform the input/assert rule first and then reach $k_3$ after an output from the process number $i$, due to the fact that $\mu(k_1) = \mu(k_2)$ and that the input/assert rule does not depend on the environment $T$ but only on the knowledge $S(A(T)) = \mu(k)$. □

Any sequence of reductions $k \rightarrow^* k'$ such that $\mu(k) = \mu(k')$ will preserve the environment knowledge : from the previous lemma, those reduction have the (nice) property that the order in which we reduce each process in the parallel composition does not matter. We will now annotate sequences of reductions to include the order in which the different processes modify the environment knowledge.

**Definition A.2** *We write:*

$$
\begin{array}{lll}
(1) & k \overset{\emptyset}{\rightarrow}^* k' & \text{iff } k \rightarrow^* k' \text{ and } \mu(k') = \mu(k) \\
(2) & k \overset{x}{\rightarrow}^* k' & \text{iff } \exists k_1 \mid k \overset{\emptyset}{\rightarrow}^* k_1 \rightarrow_x k' \text{ and } \mu(k') \neq \mu(k_1) \\
(3) & k \overset{p_1,\ldots,p_n}{\rightarrow}^* k' & \text{iff } k \overset{p_1}{\rightarrow}^* k_1 \overset{p_2}{\rightarrow}^* \cdots \overset{p_n}{\rightarrow}^* k'
\end{array}
$$

Informally, $\overset{\emptyset}{\rightarrow}^*$ denotes any sequence of reductions that preserves the environment knowledge $\overset{x}{\dashrightarrow}^*$ means that the environment knowledge was not modified until the last step, where the process numbered $x$ either performs an output of a previously unkown term, or reaches error.

$\overset{p_1,\ldots,p_n}{\rightarrow}^*$ is just syntactic sugar in order to shorten the notations.

**Remark A.3** *If $k \rightarrow^* k'$, then there exists a sequence $p_1, \ldots, p_n$ such that $k \overset{p_1,\ldots,p_n,\emptyset}{\rightarrow}^* k'$.*

**Definition A.4** ***(Eager reduction)*** *We define $\hookrightarrow_x$, a step of eager reduction on the process numbered $x$, as follows:*

$$k \hookrightarrow_x k' \text{ iff } k \overset{x}{\rightarrow}^*_x k'$$

We will write $k \hookrightarrow_{p_1,\ldots,p_n} k'$ whenever $k \hookrightarrow_{p_1} k_1 \hookrightarrow_{p_2} \cdots \hookrightarrow_{p_n} k'$.

Informally, eager reduction on the process $x$ means that we reduce only the process $x$ in the configuration until either a term unknown to the environment is written, or we reach error.

**Lemma A.5**

1. $k \overset{x}{\rightarrow}{}^* k'$ *and* $k' \neq err$ *implies* $\exists k_1\ k \hookrightarrow_x k_1 \overset{\emptyset}{\rightarrow}{}^* k'$
2. $k \overset{x}{\rightarrow}{}^* err$ *implies* $k \hookrightarrow_x err$

PROOF.

1. $k \overset{x}{\rightarrow}{}^* k'$ implies $\exists k'' \mid k \overset{\emptyset}{\rightarrow}{}^* k'' \rightarrow_x k'$ and $\mu(k') \neq \mu(k'')$. All reductions in $k \overset{\emptyset}{\rightarrow}{}^* k''$ preserve the environment knowledge, and thus by iterating lemma A.1 we can move all reductions on $x$ to the beginning of the sequence (details are left to the reader).

2. By the same reasoning : $k \overset{x}{\rightarrow}{}^* err$ implies $\exists k' \mid k \overset{\emptyset}{\rightarrow}{}^* k' \rightarrow_x$ err. Then we can use lemma A.1 to prove that $\exists k'' \mid k \overset{\emptyset}{\rightarrow}{}^*_x k'' \overset{\emptyset}{\rightarrow}{}^* k' \rightarrow_x err$ and such that there is no reduction on $x$ between $k''$ and $k'$. Then $k' \rightarrow_x err$ means that $k'(x)$ is a false assertion w.r.t. $\mu(k')$ (recall that an assertion in the environment $T$ only depends on $S(A(T))$), and as we have $k''(x) = k'(x)$ and $\mu(k'') = \mu(k)$, it implies $k'' \rightarrow_x err$. Thus $k \overset{\emptyset}{\rightarrow}{}^*_x k'' \rightarrow_x err$ and $k \overset{\emptyset}{\rightarrow}{}^*_x k'' \overset{\emptyset}{\rightarrow}{}^* k' \rightarrow_x err$

□

**Theorem A.6**

1. $k \overset{p_1,\ldots,p_n}{\rightarrow}{}^* k'$ *implies* $\exists k''\ k \hookrightarrow_{p_1,\ldots,p_n} k'' \overset{\emptyset}{\rightarrow}{}^* k'$
2. $k \overset{p_1,\ldots,p_n}{\rightarrow}{}^* err$ *implies* $\exists k''\ k \hookrightarrow_{p_1,\ldots,p_n} err$

PROOF.

1. Case ($n = 1$) was done in the previous lemma. Else $k \overset{p_1,\ldots,p_n}{\rightarrow}{}^* k'$ implies $k \overset{p_1,\ldots,p_{n-1}}{\rightarrow}{}^* k'' \overset{p_n}{\rightarrow}{}^* k'$ and by induction : $\exists k_{n-1}\ k \hookrightarrow_{p_1,\ldots,p_{n-1}} k_{n-1} \overset{\emptyset}{\rightarrow}{}^* k''$. Thus $k_{n-1} \overset{\emptyset}{\rightarrow}{}^* k'' \overset{p_n}{\rightarrow}{}^* k'$ and we can write more directly: $k_{n-1} \overset{p_n}{\rightarrow}{}^* k'$. By using the previous lemma, we have $\exists k_n\ k_{n-1} \hookrightarrow_{p_n} k_n \overset{\emptyset}{\rightarrow}{}^* k'$. QED.

2. By (1) : $\exists k_n, k''\ k \hookrightarrow_{p_1,\ldots,p_{n-1}} k_n \overset{\emptyset}{\rightarrow}{}^* k'' \overset{p_n}{\rightarrow}{}^* err$. Thus $k_n \overset{p_n}{\rightarrow}{}^* err$ and $k_n \hookrightarrow_{p_n} err$.

□

**Corollary A.7** *Correctness and completeness of the eager reduction method.*

PROOF. Completeness is stated in theorem A.6(2). Correctness comes trivially from $\hookrightarrow_x \subseteq \rightarrow^*$. □

# B An Otway-Rees variant

The protocol we wish to verify is the following:

$$\begin{array}{ll} A \to B: & N_a, A, B, \{N_a, A, B\}_{K_a} \\ B \to S: & N_a, A, B, \{N_a, A, B\}_{K_a}, N_b, \{N_b, A, B\}_{K_b} \\ S \to B: & N_a, \{N_a, K_{ab}\}_{K_a}, \{N_b, K_{ab}\}_{K_b} \\ B \to A: & N_a, \{N_a, K_{ab}\}_{K_a} \end{array}$$

...and the raw protocol description as fed to our tool is:

```
Principals:

Init(me,him):
   [me!=him] ; [me!=Id0]
   fresh na
   write <na,me,him,E(<na,me,him>,K(me))>
   read <m,e>
   [m=na] ; <na2,kab><-decrypt(e,K(me)) ; [na2=na]
   fresh confidential
   write E(confidential,kab)
   assert( (him=Id0) or secret(confidential) )
   nil


Resp(me,him):
   [me!=him] ; [me!=Id0]
   read <na,a,b,e>
   [b=me] ; [a=him]
   fresh nb
   write <na,a,b,e,nb,E(<na,a,b>,K(me))>
   read <na2,e1,e2> [na2=na] <nb2,kab><-decrypt(e2,K(me)) [nb2=nb]
   write <na,e1>
   nil


Serv(init,resp):
   read <na,a,b,e1,nb,e2>
   [a=init][b=resp][a!=b]
   k1<-K(init)
   k2<-K(resp)
   <na1,a1,b1><-decrypt(e1,k1) ; [<na1,a1,b1>=<na,a,b>]
   <na2,a2,b2><-decrypt(e2,k2) ; [<na2,a2,b2>=<na,a,b>]
   fresh kab
   write <na,E(<na,kab>,k1),E(<nb,kab>,k2)>
   nil


Environment:

Id0 ; K(Id0)
```