Verification of two versions of the Challenge Handshake Authentication Protocol (CHAP)

Guy Leduc

Research Unit in Networking (RUN)
Université de Liège
Institut Montefiore, B28
B-4000 Liège, Belgium
leduc@montefiore.ulg.ac.be
http://www-run.ulg.ac.be/

Abstract

The Challenge Handshake Authentication Protocol, CHAP, is an authentication protocol intended for use primarily by hosts and routers that connect to a network server via switched circuits or dial-up lines, but might be applied to dedicated links as well. In this paper, we specify two versions of the protocol, using the formal language LOTOS, and apply the EUCALYPTUS model-based verification tools to prove that the first version has a flaw, whereas the second one is robust to passive and active attacks. The paper is written in a tutorial fashion with a strong emphasis on the methodology used. The relative simplicity of the CHAP protocol allows one to include complete LOTOS specifications and definitions of properties, so that the experiment can be reproduced easily.

Keywords: Security protocols; Verification; Model-checking; LOTOS; CHAP

1. Introduction

The Challenge Handshake Authentication Protocol, CHAP, is a protocol used to authenticate both ends of a communication link. Such authentication protocol is intended for use primarily by hosts and routers that connect to a PPP (Point-to-Point Protocol) network server via switched circuits or dial-up lines, but might be applied to dedicated links as well. Its description can be found in [RFC1994]. It uses a challenge response mechanism based on a nonce (a sort of random number used only once) and a shared secret. This secret is only known to the entities that want to authenticate each other. Mutual authentication can be achieved by running the protocol in both directions. In this case the shared secret can be the same for both directions, but the RFC highly recommends to use two different secrets.

In this paper, we specify these two versions formally in LOTOS, and use the EUCALYPTUS model-based verification tools [Gar96] to prove that the first version has a flaw, whereas the second one is robust to passive and active attacks. Although we did not know the flaw before starting this experiment in late 1997, it was well-known to some people in the field, including someone who challenged us to find the flaw with our method and tools. Interestingly, it took only a couple of hours to specify the protocol and find the flaw.

Our verification methodology is based on the following principle. We derive the service provided by the CHAP protocol when no intruder is present. This service is expressed by suitable orderings of service primitives (such as Authentication Request, Indication and Confirmation) that we have placed at appropriate places in the behaviour of the protocol. We prove that this service fulfils the authentication properties. Then we insert an intruder process, which behaves as an insecure full-duplex communication link. This means that messages can possibly be eavesdropped, interrupted, intercepted, modified or inserted in the channels by

some simple but powerful intruder process. This intruder process can thus mimic the classical passive and active attacks against the CHAP protocol, but it is written in such a way that it cannot access the secret shared by the two protocol entities directly, and cannot break the one-way hash function used by the protocol. By that we mean that the intruder is unable to forge a message that would have a known hash. These hypotheses are sensible, otherwise it is clear that CHAP, and in fact all security protocols, would be vulnerable. To verify that a protocol is robust, we derive the actual service provided when the intruder is present, and verify it against the authentication properties.

The structure of the paper is as follows. In section 2 we give an informal description of the CHAP protocol, followed in section 3 by its formalization in LOTOS. In section 4, we verify that the protocol satisfies the authentication property when no intruder is present. Finally, in section 5, we verify that an intruder can break the single-secret version of CHAP, but cannot break the two-secret version in the scenarios checked.

2. Description of CHAP

CHAP provides mutual authentication between two parties. The basic principle of a one-way authentication is as follows. Let us call the two entities the verifier and the prover. The verifier begins by sending a challenge (i.e. a nonce) to the prover. The prover concatenates the received nonce with a shared secret and sends back a hash of this concatenated message.

Mutual authentication can be achieved by running the protocol twice: one run in each direction. In this case the shared secret can be the same for both directions, but the RFC highly recommends to use two different secrets.

Figure 1 explains the protocol with a mutual authentication and one shared secret S_{ab}. Both authentications can be interlaced. Thus the only requirements are that the first message must occur before the second one, and the third one must occur before fourth one. We will describe the other version later.

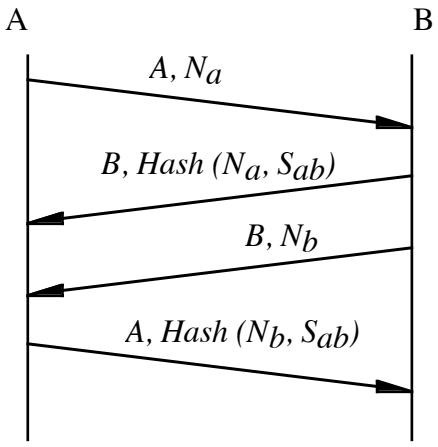

Figure 1: CHAP - A double authentication with one shared secret

3. Formal specification

The specification has been written in LOTOS, which is a standardized formal description language suitable for the description of distributed systems. It is made up of two components:

- A process algebra, mostly inspired by CCS [Mil89] and CSP [Hoa85], with a structured operational semantics. It describes the behaviour of processes and their interactions. LOTOS has a rich set of operators (multiway synchronization and abstraction like in CSP, disabling, …), and an explicit internal action like in CCS. LOTOS is briefly introduced in the appendix.
- An algebraic datatype language, ACT ONE [EM85]. A type is defined by its signature (sorts + operations on the sorts) and by equations to give a meaning to the operations.

The LOTOS specification is composed of four processes: two CHAP entities *A* and *B*, and two simplex communication media (figure 2). Both entities share a common secret S_{ab}. We also parametrize each CHAP entity by a nonce. N_a (resp. N_b) is the nonce that *A* (resp. *B*) will use to authenticate *B* (resp. *A*). The actual values of the nonces do not matter. The only requirement is that they be distinct in order to model that these random values, used once, are thus very unlikely to be the same in practice.

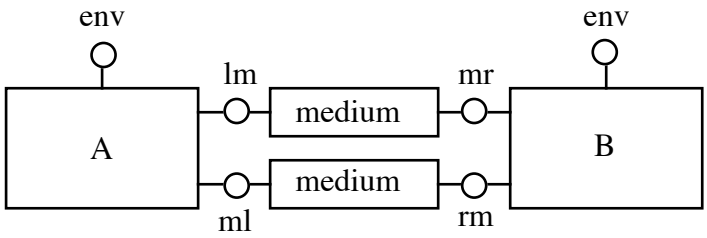

Figure 2: The architecture of the system

The structure of the LOTOS specification will thus be as follows:

```
Specification CHAP_protocol [env,lm,ml,rm,mr] :noexit

Behaviour
(CHAP_entity [env,lm,ml] (A,B,Na,Sab)
 |||
 CHAP_entity [env,rm,mr] (B,A,Nb,Sab)
)
 |[lm,ml,rm,mr]|
(medium [lm,mr]
 |||
 medium [rm,ml]
)
where
... (* Here follows the descriptions of the four processes *)
endspec
```

Before giving the description of the processes, we have to describe the data types, and discuss their abstraction levels. Entities will be identified by elements of the sort `id`. At this stage we have only two entities *A* and *B*. We also need data carriers for nonces and secret keys shared by parties. As discussed above, two distinct nonces `Na` and `Nb` are enough. Moreover, for this first version of the protocol, a single shared key `Sab` is used.

Abstract data types turn out to very useful to model in an elegant and abstract way the various cryptographic functions. Some examples of public key operations are given in [LG99]. In CHAP, the only operation we have to model is the one-way hash function, which takes as arguments a nonce and a secret key and returns a hash code. We do not care about the exact algorithm which is used and do not want to model it. What matters first is that the hash codes should be different when either the nonces or the secret keys differ. This is an ideal model of a

hashing algorithm. Secondly, it should be impossible to get the secret key from the hash code and the nonce. To this end, it suffices to define an operation `hash`, with a nonce and a secret key as arguments and which returns a hash code. The two properties are easily fulfilled by not providing any operation accessing the arguments. The absence of equations ensures automatically that all terms constructed with distinct arguments are themselves distinct.

```
Type ids is
sorts id
opns A,B: -> id
endtype

Type nonces is
sorts nonce
opns Na,Nb: -> nonce
endtype

Type keys is
sorts key
opns Sab: -> key
endtype

Type hashing is nonces, keys
sorts hash_code
opns hash: nonce,key -> hash_code
endtype
```

Besides the basic data types, we introduce service primitives which are events occurring at the interface between the CHAP protocol and the higher layer (i.e. the `env` gate). We consider three such primitives:

- Authentication request (AuthReq): Initiated by a higher layer to authenticate the other party.
- Authentication indication (AuthInd): Initiated by a CHAP entity to indicate that another party has requested an authentication.
- Authentication confirm (AuthConf): Initiated by a CHAP entity to confirm that the other party has been authenticated.

```
Type primitives is
sorts primitive
opns AuthReq, AuthInd, AuthConf: -> primitive
endtype
```

The behaviour of a CHAP entity is divided into two independent parts corresponding to the two roles of the entity: an initiator role and a responding role. When a CHAP entity receives an authentication request, it starts acting in the initiator role. It will first send its identity and a nonce to the other entity, then wait for the correct hash code before confirming the success of the authentication. In parallel, the CHAP entity can behave in the responding role when it receives a message from another party requesting an authentication. In that case, it will indicate the beginning of the authentication phase and then send back a hash of the received nonce and the secret key.

The two roles are modelled as independent processes running in parallel as both behaviours should be allowed to execute concurrently. These processes are parametrized by the identities of the involved parties, by the used nonce (for the initiator role) and by the shared secret.

```
Process CHAP_entity [env,send,get] (my_id,your_id:id, my_nonce:nonce,
                                    our_secret:key) :noexit :=

Initiator [env,send,get] (my_id,your_id,my_nonce,our_secret)
|||
Responder [env,send,get] (my_id,your_id,our_secret)

where

Process Initiator [env,send,get] (my_id,your_id:id,
                                  my_nonce:nonce,our_secret:key) :noexit
:=
env!AuthReq!my_id!your_id;   (* Request to authenticate other party *)
send!my_id!my_nonce;         (* Send id and nonce to other party *)
get!your_id?h:hash_code;     (* Wait for hash code from other party *)
env!AuthConf!my_id!your_id [h = hash(my_nonce,our_secret)];
                             (* Confirm success of authentication
                                if hash code is correct *)
stop
endproc

Process Responder [env,send,get] (my_id,your_id:id,our_secret:key) :noexit
:=
get!your_id?n:nonce;           (* Request from other party *)
env!AuthInd!your_id!my_id;     (* Indicate start of authentication phase *)
send!my_id!hash(n,our_secret); (* Send back a hash to other party *)
stop
endproc

endproc (* CHAP_entity *)
```

The last process to be described is the medium. It is modelled as a simple one-place buffer.

```
Process Medium [input,output] :noexit :=
input?x:id?n:nonce; output!x!n; Medium [input,output]
[]
input?x:id?h:hash_code; output!x!h; Medium [input,output]
endproc
```

4. Protocol verification without intrusion

We have used the Eucalyptus toolbox [Gar96] (figure 3) which is composed of:

- APERO [Pec96]: a front-end tool that supports more user-friendly notations than ACT ONE to define complex data types, and compiles them into ACT ONE types;
- XELUDO [STS94]: a simulator of the LOTOS specifications;
- CÆSAR [FGK+96]: a compiler of LOTOS specifications into Labelled Transition System (LTS);
- ALDEBARAN [FGK+96]: a verifier which minimizes a LTS while preserving an equivalence, and compares two LTS according to various preorders and equivalences;
- EXHIBITOR [Gar 98]: a tool that checks whether a trace matching a given criterion can be found in a LTS.

The first step consists of using the CÆSAR tool to generate a LTS from the LOTOS specification. Let us call this LTS CHAP.

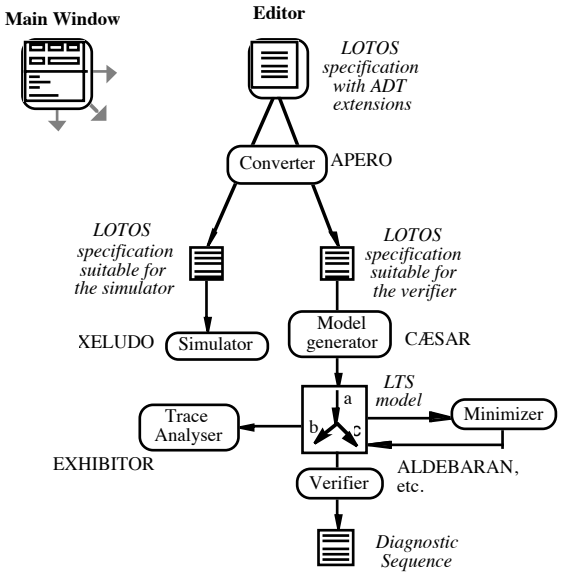

Figure 3: The Eucalyptus toolbox

The second step consists of using the ALDEBARAN tool to minimize the resulting graph. This first minimization is always done modulo the strong bisimulation equivalence [Par81], which preserves all the properties of the graph. We call the reduced LTS `CHAP.bisim`. This version of the LTS will be very useful to find detailed diagnostic sequences afterwards. One can check the absence of livelocks (also called divergences, i.e. loops of internal actions), and the presence of one deadlock state, which is simply the normal completion state.

The next step is to derive the service provided by this protocol, when no security failure is observed. To this end, we simply have to hide all the actions except the service primitives, and further reduce the graph modulo a suitable equivalence. The point here is to choose this equivalence. Formal definitions of some of them are given in appendix 2.

- The strong bisimulation equivalence [Par81] will preserve all interesting properties. After reduction, we get `CHAP_service.bisim`.
- If there are no divergences, the branching bisimulation equivalence [vGW89] will preserve all properties expressible in the ACTL* logic, while reducing the LTS further. ACTL* is ACTL [DNV90] without the *next* operator. These properties include all relevant safety and liveness properties. After reduction, we get `CHAP_service.branching`. If there are divergences, they will disappear in the reduction. Therefore, if divergences are considered harmless (or fair), this has no impact, but if divergences may be considered harmful (or unfair), this reduction is not adequate.
- If we are only interested in safety properties, disregarding liveness ones, we can reduce the LTS even further modulo the safety equivalence. This reduction preserves all the properties expressible in BSL (Branching Time Safety Logic) [BFG+91]. After reduction we get `CHAP_service.safety`. As security properties are almost always safety properties, this reduction will be very useful in practice. The only liveness security property is the non-denial of service.

The application of these reductions to the CHAP protocol without any security thread leads to the following LTS sizes:

	Nb. of states	Nb. of transitions
CHAP	62	104
CHAP.bisim	62	104
CHAP_service.bisim	62	104
CHAP_service.branching	16	24
CHAP_service.safety	16	24

The example is so simple (e.g. deterministic), that no reduction is achieved by the strong bisimulation, not even when interactions with the media are hidden. The LTS of CHAP_service.safety is actually isomorphic to the LTS generated by the following LOTOS behaviour:

```
env!AuthReq!A!B; env!AuthInd!A!B; env!AuthConf!A!B; stop
|||
env!AuthReq!B!A; env!AuthInd!B!A; env!AuthConf!B!A; stop
```

It is easy to check that the service primitives appear in correct sequences, namely that requests precede indications, themselves preceding confirmations, for each direction. However, as this will be useful in the second part of this paper, we are going to express the authentication properties formally.

Authentication means that the other party is the one it claims to be. Using the service primitives, we can formally express this property as follows:

> An env!AuthConf!X!Y should not occur before an env!AuthInd!X!Y.

Indeed, if this were not fulfilled, this would mean that user X could get a confirmation of its authentication request to user Y, while user Y would not have been notified of anything. This would clearly indicate a security breach in the authentication protocol, as someone else would have successfully impersonated Y.

This property is easy to check with the exhibitor tool. For one direction, it suffices to code the target error sequence as follows:

```
<while> ~[env!AuthInd!A!B] <until> [env!AuthConf!A!B]
```

meaning a sequence that eventually executes env!AuthConf!A!B without executing any env!AuthInd!A!B before.

In this case, no such sequence is of course found and the property is fulfilled.

5. Protocol verification with intrusion

In this section, we verify the robustness of this protocol when passive and active attacks are taken into account. To this end, we replace the full-duplex communication medium by an intruder process (figure 4), which can of course behave as a reliable medium, but can also perform the following classical attacks [Sta99]:

- Interception: This is a (passive) attack on confidentiality. The intruder gains access to a protocol message, stores it in its data base for possible reuse of (parts of) it. This models the classical wiretapping.
- Modification: This is an (active) attack on integrity. The intruder not only gains access to but tampers with a captured message.

- Fabrication: This is an (active) attack on authenticity. The intruder inserts couterfeit objects into the system. For example, it can insert spurious messages in the communication channel.

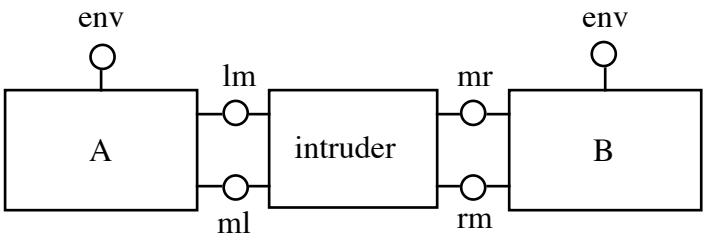

Figure 4: The system in the presence of an intruder

These attacks potentially allow the intruder to perform masquerade, replay and modification of messages.

- A masquerade takes place when the intruder pretends to be a different entity. This may be achieved e.g. by capturing authentication sequences and replaying them.
- Replay involves the passive capture of a data unit and its subsequent retransmission.
- Modification of a message means that some portion of a legitimate message is altered, or that messages are reordered to produce an unauthorized effect.

In this paper focusing on authentication, we will not consider pure service interruption attacks, such as denial of service, although this would be very easy to do. The denial of service basically prevents or inhibits the normal use of a facility, such as a communication channel or a server. Examples are the suppression of messages in transit, or the overloading of a network or system with messages so as to degrade performance.

The behaviour of the system with an intruder in place can be described as follows:

```
Specification CHAP_intruder [env,lm,ml,rm,mr] :noexit

Behaviour

(CHAP_entity [env,lm,ml] (A,B,Na,Sab)
 |||
 CHAP_entity [env,rm,mr] (B,A,Nb,Sab)
)
|[lm,ml,rm,mr]|
Intruder [lm,ml,rm,mr] (cons(A,cons(B,cons(INT,nil_id))),
                        cons(Ni,nil_nonce),
                        nil_hash,
                        cons(Si,nil_key))
...
endspec
```

The intruder process has four parameters (see below) which represent its current knowledge. They are lists of user ids, nonces, hash codes and secret keys he knows initially or intercepts during the protocol run. The list data type used to store e.g. ids could be defined as follows:

```
Type id_lists is ids, Boolean
sorts id_list
opns nil_id:-> id_list
     cons: id,id_list -> id_list
     add: id,id_list -> id_list   (* add without duplicates *)
     _isin_: id,id_list -> Bool
eqns forall x,y:id, il:id_list
     ofsort id_list
     x isin il => add(x,il) = il;
     not (x isin il) => add(x,il) = cons(x,il);
     ofsort Bool
     x isin nil_id = false;
     x isin cons(y,il) = (x eq y) or (x isin il);
endtype
```

From the above instantiation of the intruder process, we can notice that the initial knowledge of the intruder is composed of:

- His own identity *INT*, and the identities of the involved parties *A* and *B*. They are not secret and are sent in clear anyway, so they could have been learnt from a previous run.
- A "nonce" the intruder can use to initiate a run. He could possibly use it several times.
- A key S_i the intruder will use to fabricate fake hash codes. This key should be different from S_{ab}, which is supposed to be known by *A* and *B* only.

The LOTOS specification of the intruder process can now be given. It models a process that can intercept any message (and increase his knowledge accordingly), and send at any time any (fake) message which can be built from its knowledge.

```
Process Intruder [lm,ml,rm,mr] (il:id_list, nl: nonce_list, hl:hash_list,
                                sl: secret_list) :noexit :=

(* first four lines are interceptions of all possible messages *)

lm?x:id?n:nonce; Intruder [lm,ml,rm,mr] (add(x,il),add(n,nl),hl,sl)
[]
rm?x:id?n:nonce; Intruder [lm,ml,rm,mr] (add(x,il),add(n,nl),hl,sl)
[]
lm?x:id?h:hash_code; Intruder [lm,ml,rm,mr] (add(x,il),nl,add(h,hl),sl)
[]
rm?x:id?h:hash_code; Intruder [lm,ml,rm,mr] (add(x,il),nl,add(h,hl),sl)

(* next lines are transmissions of all possible messages the intruder can
   fabricate *)
[]
(choice x:id,n:nonce [] [(x isin il) and (n isin nl)] ->
      (ml!x!n; Intruder [lm,ml,rm,mr] (il,nl,hl,sl)
       []
       mr!x!n; Intruder [lm,ml,rm,mr] (il,nl,hl,sl))
)
[]
(choice x:id,h:hash_code [] [(x isin il) and (h isin hl)] ->
      (ml!x!h; Intruder [lm,ml,rm,mr] (il,nl,hl,sl)
       []
       mr!x!h; Intruder [lm,ml,rm,mr] (il,nl,hl,sl))
)
```

```
(* a hash code can be fabricated by just replaying an observed one
   (as in the line above) or by contructing a new one from known nonces
   and keys (as in the line below) *)

[]
(choice x:id,n:nonce,k:key [] [(x isin il) and (n isin nl) and (k isin sl)]
   -> (ml!x!hash(n,k); Intruder [lm,ml,rm,mr] (il,nl,hl,sl)
       []
       mr!x!hash(n,k); Intruder [lm,ml,rm,mr] (il,nl,hl,sl))
)
endproc
```

We are now ready to apply again the verification steps explained in section 4. We get the following numbers:

	Nb. of states	Nb. of transitions	
`CHAP_intruder`	9 178	27 790	
`CHAP_intruder.bisim`	916	4 234	
`CHAP_intruder_service.bisim`	419	1 434	No livelock
`CHAP_intruder_service.branching`	126	418	
`CHAP_intruder_service.safety`	37	76	Still nondeterministic

Even with 37 states and 76 transitions, the last graph is difficult to read. So we check whether the authentication property is satisfied using EXHIBITOR.

The verdict is that there is a sequence violating the authentication property. It is given by:

```
<initial state>
env!AuthReq!A!B
env!AuthInd!B!A
env!AuthConf!A!B
<goal state>
```

We can indeed notice that a confirmation is received by *A*, while *B* has not received any indication. The indication present in the sequence occurs at *A*'s side, not at *B*'s.

However, it is not easy to understand the security failure from this sequence of service primitives. To get a detailed scenario of the attack, we have to search for an invalid sequence on the `CHAP_intruder.bisim` specification. This gives the following sequence:

```
<initial state>
env!AuthReq!A!B
lm!A!Na
ml!B!Na
env!AuthInd!B!A
lm!A!hash(Na,Sab)
ml!B!hash(Na,Sab)
env!AuthConf!A!B
<goal state>
```

The attack is now Crystal clear, and shown on figure 5:
1. The intruder intercepts the authentication request that *A* sends to *B*,
2. It impersonates *B* and sends another authentication request to *A* using the same nonce N_a
3. *A* sends back a hash of N_a with the shared secret S_{ab}
4. The intruder intercepts this hash code and uses it as a response to the initial *A*'s request

5. *A* believes he has talked to *B*, while he has only talked to the intruder

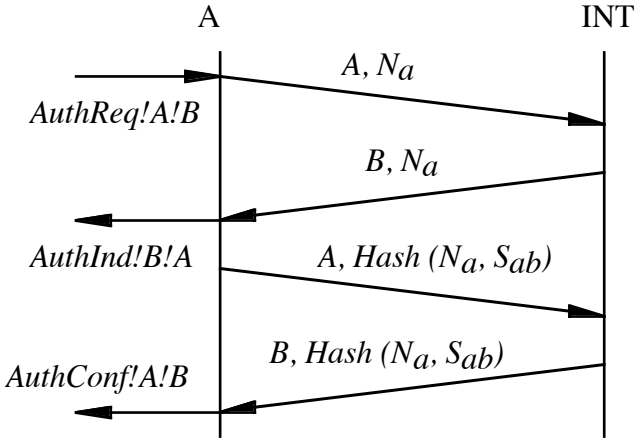

Figure 5: The scenario of the attack

The CHAP protocol thus fails to perform its authentication service. The essence of this attack is the fact that the intruder uses *A* himself as a way to compute the hash code he has to send back to *A* to complete the protocol. A proposed better variant of CHAP consists of having two shared secrets, one per direction (figure 6). It would at least bar the above attack, as *A* would expect `ml!B!hash(Na,Sba)` instead of `ml!B!hash(Na,Sab)` in step 5, which would not allow the intruder to send a valid authentication response.

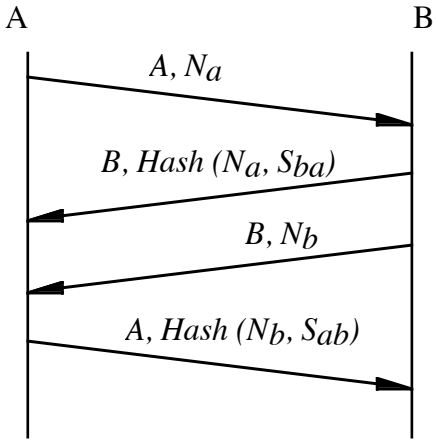

Figure 6: CHAP with two shared secrets

We will check whether this variant is robust. For that purpose we need to adapt our specification as shown below. The only difference is the presence of two secret keys S_{ab} and S_{ba} shared by *A* and *B*. One is used in the initiator role, and the other one is used in the responder role.

```
Specification CHAP2_intruder [env,lm,ml,rm,mr] :noexit

Behaviour

(CHAP_entity [env,lm,ml] (A,B,Na,Sab,Sba)
 |||
 CHAP_entity [env,rm,mr] (B,A,Nb,Sba,Sab)
)
|[lm,ml,rm,mr]|
Intruder [lm,ml,rm,mr] (cons(A,cons(B,cons(INT,nil_id))),
                        cons(Ni,nil_nonce),
                        nil_hash,
                        cons(Si,nil_key))
where

Process CHAP_entity [env,send,get] (my_id,your_id:id,
                                    my_nonce:nonce,
                                    our_secret_out, our_secret_in:key)
                                :noexit :=
Initiator [env,send,get] (my_id,your_id,my_nonce,our_secret_in)
|||
Responder [env,send,get] (my_id,your_id,our_secret_out)
endproc
...
endspec
```

We are now ready to apply again the verification steps explained in section 4. We get the following numbers:

	Nb. of states	Nb. of transitions	
`CHAP2_intruder`	9 206	27 720	
`CHAP2_intruder.bisim`	885	4 156	
`CHAP2_intruder_service.bisim`	288	900	No livelock
`CHAP2_intruder_service.branching`	64	176	
`CHAP2_intruder_service.safety`	25	50	Deterministic

In this case, no sequence violating the authentication property is found by the exhibitor tool. However, that does not mean that the intruder has no effect on the system. If we look carefully at the minimized graph projected on the service primitives, we see more states and transitions than without the presence of the intruder. This means that the users can perceive some of the intruder's actions, as tentative attacks, but without any breach in the authentication. An example of such sequence is the occurrence of an `AuthInd!A!B` without a prior `AuthReq!A!B`. This is because the intruder can generate a fake authentication request to *B*, composed of the well-known identity of *A* and a nonce. This will inevitably trigger an `AuthInd!A!B`, but without consequence for the security of the system.

Even though CHAP proved to be robust in the presence of our intruder, an interesting question is the confidence we can have in this result. There are potential weak points in our approach. The first one is the model of the intruder and its initial knowledge. In our opinion, the very simple structure of the intruder process gives high confidence. It is constructed in such a systematic way that it could even be automated. The initial knowledge is perhaps more debatable. It is clear that the power of the intruder can be increased by adding new pieces of information in his initial knowledge. Clearly, we have initialized it with a minimum and

reasonable information. Adding the nonces *A* and *B* are going to use in the protocol run would only make sense if our intruder could guess the nonces in advance. We did not consider this realistic. Anyway, as the nonces are sent in clear, the intruder can learn them quite easily and reuse them at will.

Another potential weakness of our approach is the studied scenario, in which *A* and *B* merely try and authenticate themselves. It is not sure that this scenario is sufficiently general. Suppose an entity called *INT* is known to *A* as a respectable party and both of them share two secret keys S_{ai} and S_{ia} for authentication. This can be similar between *INT* and *B*. Now, suppose *INT* starts behaving as an intruder. Using the shared secrets, would it be possible for *INT* to impersonate *A* when talking to *B*, for example? Such a question cannot be answered directly, and shows that the context in which the verification is carried out is very important. In [Low96] and [LBK+96], attacks have been found respectively in the Needham-Shroeder and Equicrypt protocols in such contexts.

To extend our proof, we will consider the following richer scenario:
- *A* (resp. *B*) may initiate a CHAP protocol either with *B* (resp. *A*) or *INT*
- *A* (resp. *B*) shares distinct secret keys with *B* (resp. *A*) and *INT*
- *A* (resp. *B*) can respond to an authentication request from *B* (resp. *A*) or *INT*

To strengthen the power of the intruder, we consider that the secrets he shares with *A* and *B* for both directions are all equal. This potentially allows the intruder to reuse messages in more contexts.

The structure of the LOTOS specification becomes the following:

```
Specification CHAP2_intruder_ext [env,lm,ml,rm,mr] :noexit

Behaviour

(CHAP_entity [env,lm,ml] (A,B,INT,Na,Sab,Sba,Si)
 |||
 CHAP_entity [env,rm,mr] (B,A,INT,Nb,Sba,Sab,Si)
)
|[lm,ml,rm,mr]|
Intruder [lm,ml,rm,mr] (cons(A,cons(B,cons(INT,nil_id))),
                        cons(Ni,nil_nonce),
                        nil_hash,
                        cons(Si,nil_key))
where
Process CHAP_entity [env,send,get] (my_id,your_id,your_id_2:id,
                                    my_nonce:nonce,
                                    our_secret_out, our_secret_in,
                                    our_secret_2:key) :noexit :=
(Initiator [env,send,get] (my_id,your_id,my_nonce,our_secret_in)
 []
 Initiator [env,send,get] (my_id,your_id_2,my_nonce,our_secret_2)
)
|||
Responder [env,send,get] (my_id,your_id,our_secret_out)
|||
Responder [env,send,get] (my_id,your_id_2,our_secret_2)
endproc
...
endspec
```

Unfortunately, the size of the generated model turns out to be too big. In such a case, we could have used the compositional approach of the toolset to generate the global LTS from those of the components (after minimization). This approach sometimes works, but for some systems, the LTS of the components are even larger than the LTS of the whole system. Therefore, we favoured another approach. Fortunately, the structure of the specification is such that we can tackle the problem in parts. Indeed, the CHAP entity is a process of the form `(X [] Y) ||| Z`. Then the two CHAP entities are again combined by `|||`. And finally, this resulting process is in synchronized parallelism with the intruder.

Given the well-known property that `(X|Z)[](Y|Z)` has the same traces as `(X[]Y)|Z`, where `|` means any parallel composition operator, we can split the set of traces of the specification into 4 subsets:

- when *A* can only initiate a run with *B*, and *B* can only initiate a run with *A*
- when *A* can only initiate a run with *B*, and *B* can only initiate a run with *INT*
- when *A* can only initiate a run with *INT*, and *B* can only initiate a run with *A*
- when *A* can only initiate a run with *INT*, and *B* can only initiate a run with *INT*

The union of these cases will give all the possible sequences. Therefore it suffices to deal with the 4 cases independently and check with the exhibitor tool that none of them contains any sequence matching our given pattern.

We could think that the first case has been verified already with the `CHAP2_intruder` specification. This is not true. In this specification, the CHAP entities did not accept requests from *INT*, which we would like to allow now. The second and third cases are symmetrical, so that only one needs to be generated. fourth case can be left out, as this scenario can never generate any `AuthConf!A!B`, nor any `AuthConf!B!A`, which makes it impossible to violate the authentication property.

To do so, it suffices to prune the two instances of the `CHAP_entity` process. For example, to check the second case, we consider:

```
Specification CHAP2_int_AB_BI [env,lm,ml,rm,mr] :noexit
...
Behaviour
((Initiator [env,lm,ml] (A,B,Na,Sba)
  |||
  Responder [env,lm,ml] (A,B,Sab)
  |||
  Responder [env,lm,ml] (A,INT,Si)
 )
 |||
 (Initiator [env,rm,mr] (B,INT,Nb,Si)
  |||
  Responder [env,rm,mr] (B,A,Sba)
  |||
  Responder [env,rm,mr] (B,INT,Si)
 )
)
|[lm,ml,rm,mr]|
Intruder [lm,ml,rm,mr] (cons(A,cons(B,cons(INT,nil_id))),
                        cons(Ni,nil_nonce),
                        nil_hash,
                        cons(Si,nil_key))
endspec
```

If we apply again the verification steps explained in section 4 to the two specifications representing cases 1 and 2 above (called `CHAP2_int_AB_BA` and `CHAP2_int_AB_BI` respectively), we get the following numbers:

	Nb. of states	Nb. of transitions	
`CHAP2_int_AB_BA`	673 990	3 238 734	
`CHAP2_int_AB_BA.bisim`	41 808	298 608	
`CHAP2_int_AB_BA_service.bisim`	1 152	4 500	No livelock
`CHAP2_int_AB_BA_service.branching`	64	176	
`CHAP2_int_AB_BA_service.safety`	25	50	Deterministic

`CHAP2_int_AB_BI`	789 744	3 816 751	
`CHAP2_int_AB_BI.bisim`	54 624	389 040	
`CHAP2_int_AB_BI_service.bisim`	1 164	4 657	No livelock
`CHAP2_int_AB_BI_service.branching`	16	30	
`CHAP2_int_AB_BI_service.safety`	10	15	Deterministic

The verdict of the EXHIBITOR tool is that there is no sequence, in none of these two specifications, that falsifies our authentication property. We can thus consider that the second version of the CHAP protocol is robust to the attacks and in the configurations we have considered.

Does it mean that this version of CHAP is absolutely secure?

We cannot declare that for the following reasons:

- We verified the robustness in a single protocol run. We did not prove a more general result stating that our conclusions are still valid if we consider an intruder which can participate in multiple protocol runs and possibly acquire additional knowledge that he could reuse to break the protocol. Moreover, scenarios with infinitely many protocol runs cannot be finite-state (and thus cannot be verified by our method), because this would require to assign infinitely many differents nonces to the entities. Executing several successive runs with the same nonce (to keep the system finite-state but with infinite traces), would not model the system correctly.
- Our model of cryptographic functions, such as hashing, is idealized. If there is a weakness in the hashing function used by the protocol, an intruder might be capable of breaking it without having the secret key.
- We have not modelled the establishment of the shared secrets between the parties, and therefore we did not consider attacks on this protocol. If this protocol has a hole, the intruder could acquire some knowledge of the shared secret, which we did not take into account in our verification.

6. Conclusion

This paper illustrates a model-based formal verification process for security protocols by using the specification language LOTOS, and the CHAP protocol as an example.

We have shown how intrusion can be taken into account by adding an intruder process replacing the communication channels. Our model of this intruder is very simple and powerful. He can mimic very easily real-world non cryptographic and non repetitive attacks on the behaviour of the protocol. The idea of explicitly introducing an intruder was first proposed in [DEK82, DY83] in another setting. This idea was then used in the Interrogator system

[MCF87], where the participants are modelled as communicating state-machines and the network is assumed to be under the control of an intruder, which can intercept messages, destroy or modify them, or pass them through unmodified. The NRL Protocol Analyser [KMM94, Mea94] is similar to the Interrogator, but the goal is here to prove the unreachability of some undesirable states. It can deal with infinite-state systems but the search is less automated than in the Interrogator. The difference between our approach and these methods is that we do not have to define some pathological target states to be searched for by the tool. We just give safety properties expressed on sequences of service primitives.

We have explained the validation process and the formalization of security properties as safety properties. These properties are similar to the correspondence properties, used in [WL93], which require that certain events can take place only if others have taken place previously. Our approach is similar to [Low96, LR97] where authentication protocols were specified in CSP [Hoa85] and checked by the FDR tool by verifying the trace inclusion relation between the system and the property. This tool and the one we have used are not classical model-checkers but rather equivalence or preorder checkers. Model-checkers (e.g. [MCJ97, MSS97]) have also been used in similar ways.

The model-based methods are extremely powerful at finding subtle flaws in protocols, but are less adequate to prove correctness when no bug is found. This is because they are applied on simplified, though realistic, models of the systems. On the other hand, theorem provers [Kem89, CG90, Bol96, Sch98] can provide such proofs and can also deal more easily with infinite-state systems. However, the proofs are usually less automated, and when no proof has been derived for a given property, it is not easy to know whether the property is wrong or whether the tool simply find it. In particular, an attack that falsifies the property is not provided automatically.

In our approach, the verification is quite automatic and allows one to make efficient corrections and improvements. However, as with any model-checking methods, we have had to simplify the model to keep it finite-state. There exist ways to extend the method to infinite-state systems. In [Low96], an additional induction proof has been provided to extend the correctness guarantee to an arbitrary number of involved entities.

Another approach which circumvents the problem of adding an explicit intruder process is proposed in [AG97] where the Spi-calculus is used to describe security protocols. The idea is to verify that the protocol specification placed in any Spi-calculus context is equivalent to the expected ideal behaviour (i.e. without intruder). Threads expressible in the Spi-calculus are thus implicitly considered among the possible contexts. However, this approach is not so easy to use in practice because the equivalence is sometimes too strong. For example, some intruder's actions may be such that the equivalence is not fulfilled, while the security of the system is not in danger, because the non equivalence simply results from the falsification of an irrelevant property. In the CHAP protocol, we have indeed seen that indruder’s actions are often noticeable by the users of the protocol (e.g. by the occurrence of some extra indication primitives) while the authentication property was still verified.

References

[AG97] M. Abadi and A.D. Gordon. *A Calculus for Cryptographic Protocols: The Spi Calculus*. Proceedings of the 4th ACM Conference on Computer and Communication Security, 1997.

[BFG+91] A. Bouajjani, J.-C. Fernandez, S. Graf, C. Rodriguez and J. Sifakis. *Safety for Branching Time Semantics*. In: 18th ICALP, Berlin, July 1991. Springer-Verlag.

[BoB87] T. Bolognesi and E. Brinksma. *Introduction to the ISO Specification Language LOTOS*. Computer Networks and ISDN Systems 14 (1) 25-59 (1987).

[Bol96] D. Bolignano. *Formal verification of cryptographic protocols*. In: Proc. of the 3rd ACM Conference on Computer and Communication Security, 1996.

[CG90] P. Chen and V. Gligor. *On the Formal Specification and Verification of a Multiparty Session Protocol*. In: Proc. of the IEEE Symposium on Research in Security and Privacy, 1990.

[DEK82] D. Dolev, S. Even, and R. Karp. *On the Security of Ping-Pong Protocols*. Information and Control, pp. 57-68, 1982.

[DNV90] R. De Nicola, F.W. Vaandrager. *Actions versus State Based Logics for Transition Systems*. Proc. Ecole de Printemps on Semantics of Concurrency, LNCS 469, Springer Verlag, Berlin, 1990, 407-419.

[DoY83] Dolev, and A. Yao. *On the Security of Public Key Protocols*. IEEE Transactions on Information Theory, 29(2):198-208, March 1983.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics*. In: W. Brauer, B. Rozenberg, A. Salomaa, eds., EATCS , Monographs on Theoretical Computer Science, Springer Verlag, 1985.

[FGK+96] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier and M. Sighireanu. *CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox*. In: R. Alur and T. Henzinger, eds, Proc. of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA), Aug. 1996.

[Gar96] H. Garavel. *An overview of the Eucalyptus Toolbox*. In: Proc. of the COST247 workshop (Maribor, Slovenia), June 1996.

[Gar98] H. Garavel. *OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation and Testing*. Proc. of TACAS'98, LNCS 1384, Springer-Verlag, Berlin, 1998, 68-84.

[GL97a] F. Germeau, G. Leduc. *Model-based Design and Verification of Security Protocols using LOTOS*. Proc. of the DIMACS Workshop on Design and Formal Verification of Security Protocols, Rutgers University, NJ, USA, Sept. 97, 22 p.

[GL97b] F. Germeau, G. Leduc. *A computer-aided design of a secure registration protocol*. Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE/PSTV' 97, Chapman & Hall, London (1997), 145-160.

[Hoa 85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[ISO8807] ISO/IEC. Information Processing Systems – Open Systems Interconnection – *LOTOS, a Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. IS 8807, February 1989.

[Kem89] R. Kemmerer. Using *Formal Methods to Analyse Encryption Protocols*. IEEE Journal on Selected Areas in Communications, 7(4):448-457, 1989.

[KMM94] R. Kemmerer, C. Meadows, and J. Millen. *Three Systems for Cryptographic Protocol Analysis*. Journal of Cryptology, 7(2):14-18, 1989.

[LBK+96] G. Leduc, O. Bonaventure, E. Koerner, L. Léonard, C. Pecheur, D. Zanetti. *Specification and verification of a TTP protocol for the conditional access to services*. In: Proc. of 12th J. Cartier Workshop on "Formal Methods and their Applications: Telecommunications, VLSI and Real-Time Computerized Control System", Montreal, Canada, 2-4 Oct. 96.

[LBL+99] G. Leduc, O. Bonaventure, L. Léonard, E. Koerner, C. Pecheur. *Model-Based Verification of a Security Protocol for Conditional Access to Services*. Formal Methods in System Design, Vol. 14, No. 2, March 1999, 171-191.

[LG99] G. Leduc and F. Germeau. *Verification of Security Protocols using LOTOS - Method and Application*. To appear in Computer Communications, special issue on "Formal Description Techniques in Practice".

[Low96] G. Lowe. *Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR*. In: T. Margaria and B. Steffen (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, LNCS 1055, Springer-Verlag, 1996.

[LR97] G. Lowe and B. Roscoe. Using CSP to Detect Errors in the TMN Protocol, IEEE Transactions on Software Engineering, vol. 23 (10), Oct. 1997, 659-669.
[MCF87] J. Millen, S. Clark, and S. Freedman. *The Interrogator: Protocol Security Analysis*. IEEE Transactions on Software Engineering, SE-13(2), 1987.
[MCJ97] W. Marrero, E. Clarke, and S. Jha. *A Model Checker for Authentication Protocols*. Proc. of the DIMACS Workshop on Design and Formal Verification of Security Protocols, Rutgers University, Sept. 1997.
[Mea94] C. Meadows. *The NRL Protocol Analyser: An Overview*. Journal of Logic Programming 1994:19, 20:1-679.
[Mil 89] R. Milner. *Communication and Concurrency*. Prentice-Hall Intern., London, 1989.
[MSS97] J. Mitchell, V. Shmatikov, U. Stern. *Finite-State Analysis of SSL 3.0 and Related Protocols*. Proc. of the DIMACS Workshop on Design and Formal Verification of Security Protocols, Rutgers University, Sept. 1997.
[NFG+91] R. de Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action based framework for verifying logical and behavioural properties of concurrent systems. University of La Sapienza, Roma.
[Par81] D. Park. *Concurrency and Automata on Infinite Sequences*.In P. Deussen ed., Theoretical Computer Science, LNCS 104, Springer Verlag, March 1981, 167-183.
[Pec96] C. Pecheur. *Improving the Specification of Data Types in LOTOS*. Doctoral Dissertation, University of Liège, July 1996.
[RFC1994] W. Simpson. *PPP Challenge Handshake Authentication Protocol (CHAP)*. RFC 1994, August 1996.
[Sch98] S. Schneider. *Verifying Authentication Protocols in CSP*. IEEE Transactions on Software Engineering vol. 24 (9), Sept. 1998, 751-758.
[Sta99] W. Stallings. *Cryptography and Network Security - Second Edition*. Prentice-Hall, 1999.
[STS94] B. Stepien, J. Tourrilhes, and J. Sincennes. *ELUDO: The University of Ottawa Toolkit*. Technical Report, Univesity of Ottawa, 1994.
[vGW89] R. van Glabeek and W. Weijland. *Branching-Time and Abstraction in Bisimulation Semantics*. Proc. of the 11th World Computer Congress, San Francisco, 1989.
[WL93] Woo and S. Lam. *A Semantic Model for Authentication Protocols*. In : Proc. of IEEE Symposium on Research in Security and Privacy, 1993.

Appendix 1: Overview of the LOTOS operators

- **stop** is an inactive (deadlocked) process.
- $go_1...o_n[SP]; P$ (action-prefixing) is a process that first performs an (observable) action on gate g and then behaves like P. The tuple $o_1...o_n$ determines the data exchanged during the synchronisation: either data sent, by $!tx$, or data (of sort s) received, by $?x:s$. The variables declared in $o_1...o_n$ to receive data can appear in the selection predicate (i.e. the boolean expression) SP. Data can be received only if they verify SP.
- **i**; P is a process that first performs an internal action and then behaves like P.
- **exit**$(e_1,...e_n)$ is a process that successfully terminates. It performs an action on gate δ and then turns into **stop**. The tuple $e_1,...e_n$ determines the data transmitted to the subsequent process (see the enabling operator).
- $P_1 \Box P_2$ (choice) is a process that can behave either like P_1 or like P_2. The choice is resolved by the first process which performs an action. Notice that internal actions also resolve the choice.
- $P_1 |[\Gamma]| P_2$ is the parallel composition of P_1 and P_2 with synchronisation on the gates in Γ.
- **hide** Γ **in** P hides actions of P occurring at gates present in the set Γ, i.e. renames them **i**.
- $P_1 \gg$ **accept** $x_1:s_1,...x_n:s_n$ **in** P_2 (enabling) is the sequential composition of P_1 and P_2, i.e. P_2 can start when P_1 has terminated successfully. A process terminating successfully can

transmit data to its successor: the tuple $e_1,...e_n$ associated with **exit** determines the data values transmitted and **accept** $x_1{:}s_1,...x_n{:}s_n$ **in** specifies the data P_2 expects to receive.

- P_1 [> P_2 (disabling) allows P_2 to disable P_1 provided P_1 has not terminated successfully.
- [SP]→ P (guard) behaves like P if the guard SP is true and like **stop** otherwise.
- **let** $x_1{=}tx_1,...x_n{=}tx_n$ **in** P (instantiation) instantiates the free variables $x_1...x_n$ in P.
- **choice** $x_1{:}s_1,...x_n{:}s_n$ □ P (choice over values). Assuming P depends on the variables $x_1...x_n$, (of sorts $s_1...s_n$), **choice** $x_1{:}s_1,...x_n{:}s_n$ □ P offers a choice between the processes $P(tx_1...tx_n)$ for all the combinations of values $(tx_1...tx_n)$ of sorts $(s_1...s_n)$. For example, **choice** $x{:}Nat$ □ $P(x)$ means $P(0)$ □ $P(1)$ □ $P(2)$ □ ...

Appendix 2: Definitions of equivalences

Consider a LTS = $\langle S, A, T, s_0\rangle$ where S is the set of states, A the alphabet of actions (with i denoting the internal action), T the set of transitions and s_0 the initial state.

A relation $R \subseteq S \times S$ is a (strong) **bisimulation** iff $\forall \langle P,Q\rangle \in R$,$\forall\ a \in A$:

(i) if $P \xrightarrow{a} P'$, then $\exists\ Q'$ such that $Q \xrightarrow{a} Q'$ and $\langle P',Q'\rangle \in R$,

(ii) if $Q \xrightarrow{a} Q'$, then $\exists\ P'$ such that $P \xrightarrow{a} P'$ and $\langle P',Q'\rangle \in R$.

$\text{Sys}_1 = \langle S_1, A, T_1, s_{0_1}\rangle$ and $\text{Sys}_2 = \langle S_2, A, T_2, s_{0_2}\rangle$ are **bisimilar**, denoted $\text{Sys}_1 \sim \text{Sys}_2$, iff there exists a strong bisimulation relation $R \subseteq S_1 \times S_2$, such that $\langle s_{0_1},s_{0_2}\rangle \in R$

A relation $R \subseteq S \times S$ is a **branching bisimulation** iff $\forall \langle P,Q\rangle \in R$,$\forall\ a \in A$:

(i) (if $P \xrightarrow{a} P'$, then $\exists\ Q', Q''$ such that $Q \xrightarrow{i^*} Q' \xrightarrow{a} Q''$ and $\langle P,Q'\rangle \in R$ and $\langle P',Q''\rangle \in R$)
or (if $P \xrightarrow{i} P'$, then $\langle P',Q\rangle \in R$),

(ii) (if $Q \xrightarrow{a} Q'$, then $\exists\ P', P''$ such that $P \xrightarrow{i^*} P' \xrightarrow{a} P''$ and $\langle P',Q\rangle \in R$ and $\langle P'',Q'\rangle \in R$)
or (if $Q \xrightarrow{i} Q'$, then $\langle P,Q'\rangle \in R$).

$\text{Sys}_1 = \langle S_1, A, T_1, s_{0_1}\rangle$ and $\text{Sys}_2 = \langle S_2, A, T_2, s_{0_2}\rangle$ are **branching bisimilar**, denoted $\text{Sys}_1 \approx_{bb} \text{Sys}_2$, iff there exists a branching bisimulation relation $R \subseteq S_1 \times S_2$, such that $\langle s_{0_1},s_{0_2}\rangle \in R$

A relation $R \subseteq S \times S$ is a **weak simulation** iff $\forall \langle P,Q\rangle \in R$,$\forall\ a \in A$:

if $P \xrightarrow{i^*} \xrightarrow{a} P'$, then $\exists\ Q'$ such that $Q \xrightarrow{i^*} \xrightarrow{a} Q'$ and $\langle P',Q'\rangle \in R$.

$\text{Sys}_1 = \langle S_1, A, T_1, s_{0_1}\rangle$ can be simulated by $\text{Sys}_2 = \langle S_2, A, T_2, s_{0_2}\rangle$, denoted $\text{Sys}_1 \preceq_s \text{Sys}_2$, iff there exists a weak simulation relation $R \subseteq S_1 \times S_2$, such that $\langle s_{0_1}, s_{0_2}\rangle \in R$

$\text{Sys}_1 = \langle S_1, A, T_1, s_{0_1}\rangle$ and $\text{Sys}_2 = \langle S_2, A, T_2, s_{0_2}\rangle$ are **safety equivalent**, denoted $\text{Sys}_1 \approx_s \text{Sys}_2$, iff $Sys_1 \preceq_s Sys_2$ and $Sys_2 \preceq_s Sys_1$.