Analysing the BKE-security protocol with μCRL

Jan Friso Groote Sjouke Mauw Alexander Serebrenik
Laboratory of Quality of Software (LaQuSo), Division of Computer Science
Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands.
E-mail: {J.F.Groote,S.Mauw,A.Serebrenik}@tue.nl

Abstract
The μCRL language and its corresponding tool set form a powerful methodology for the verification of distributed systems. We demonstrate its use by applying it to a well-known security protocol: the Bilateral Key Exchange protocol.

1 Introduction

Contemporary tools for the analysis of communicating processes are all essentially based on an explicit state space representation [7, 10, 12, 19, 23, 25]. In some cases specific data types have an explicit encoding (e.g. time in [10]), and in other cases compression techniques have been applied to the representation of states [7, 19]. The tools have matured to the level where systematically errors can be found in relatively small process descriptions. Applying the tools to larger and real-life systems is a continuous battle with the so-called state space explosion problem. The essence is that basically every communicating process that is analysed has more states than can be represented within such tools. The consequence is that behavioural analysis must be restricted to small instances under restricted communication scenario's.

By proceeding along the current lines a steady progress can be expected. Computers become larger and faster, and better state space compression and analysis techniques will be invented. However, we are convinced that in order to make a substantial step forward, a more radical development is necessary. We believe that using the μCRL tool set [14] we made such a step.

Using techniques described in [26] it is possible to transform all μCRL processes to linear form. The tools that do this are able to perform this operation for systems consisting of hundreds of parallel processes. A linear process allows all kinds of manipulations that can easily be automated. The crucial point is that (a) human ingenuity can determine which operations must be applied and (b) computer programs are available to carry these out. A very similar situation can be found in engineering mathematics where Maple, Mathlab and Mathematica have become tools to interactively manipulate large mathematical formulae. We believe it is essential that a tool set is grounded in a sound theoretical framework. For μCRL a good summary of the state of the art is given in [15].

In the current paper we will demonstrate the use of μCRL to a popular application domain, viz. the verification of security protocols. The importance of formal methods in the analysis of security protocols is evident in the fact that developing correct security protocols is notoriously difficult. Many proposed and even many implemented security protocols have been found to contain errors. The reason for this is that it is hard to predict all possible attacks that an intruder can use to break security.

The Dolev-Yao intruder model (see [9]) is the most widely accepted way of modelling an intruder. In this model the intruder has complete control over the network. He can intercept messages and construct new messages from all information that he can deduce from the messages sent by agents. The growth of the intruder knowledge, and thus the number of different messages that the intruder can construct, is the major reason for the exponential growth in the number of possible behaviours of the system.

Model checking and theorem proving are the most used formal methods for verifying security protocols. Theorem proving requires the development of a proof logic for security properties (see e.g. [6]) and in general needs user interaction. Model checking (see e.g. [20]) suffers from the state-space explosion problem, but can be done automatically. We will demonstrate the powerful state reduction techniques of the μCRL model checker by analysing the Bilateral Key Exchange (BKE, see [8]) protocol. The purpose of this protocol is to establish a fresh cryptographic key between two agents which should remain secret from a possible intruder. In order to keep our focus on the modelling of a security protocol, rather than on the modelling of security properties, we will concentrate on the secrecy requirement only.

This paper is structured as follows. In Section 2 we will explain the μCRL language and its tool set. The Bilateral Key Exchange protocol and its specification are treated in Section 3. Modelling of the he adversary is discussed in Section 4. Optimisations are the subject of Section 5. Finally, the results are evaluated in Section 6.

Acknowledgements. We thank Cas Cremers for the fruitful discussions on modelling the BKE protocol. We also thank Stefan Blom for helping us with performing the experimental evaluation.

2 A short primer in μCRL

The language μCRL (*micro Common Representation Language*) [14, 15] has been defined in 1990. At that time it had been understood that process algebras were a very suitable way to describe the behaviour of systems. Yet, process algebras did not comprise data and therefore lacked practical expressivity. With exactly this purpose in mind the languages PSF [21] and LOTOS [5] were designed. Their major goal was to make a specification language, whereas the purpose of μCRL is to constitute a practical, concise, mathematical language geared towards verification and analysis of behaviour of realistic communicating systems.

The consequence is a bare language of sufficient expressivity with a precisely defined semantics, basic axioms, proof methodology and a full range of analysis tools. In this section we shortly explain the language and the main ideas behind the verification methodologies and tools. For a much more extensive treatment, we refer to [15].

2.1 The data language

Basically, μCRL consists of a data part and a process part. The data part is built up from very simple equational abstract data types. All data sorts that are used in a specification must be declared. Sorts can be declared by

sort **Bool** $\mathbb{N}$ *List*

This declares three sorts. All sorts are non empty. The sort **Bool** is special in the sense that it must have exactly two different elements denoted by t and f. The reason for this is that terms of sort **Bool** are used in conditions in process terms. Note that the difference of t and f can induce that other terms must be different, too. If an assumption that terms t and u are equal, leads to the conclusion that $\mathsf{f} = \mathsf{t}$, then t and u must be different. This technique is a form of *reductio ad absurdum* and is the only available technique in the data part of μCRL to prove that terms must be different.

Constructor functions for each sort can be declared as follows:

func $\mathsf{t}, \mathsf{f} : \to \mathbf{Bool}$
$0 : \to \mathbb{N}$
$succ : \mathbb{N} \to \mathbb{N}$

We let t denote true, f denote false. The constructor 0 represents zero, and *succ* stands for the successor function on the natural numbers. If a sort D has constructor functions with D as its target sort, then all elements of that sort can be denoted using the constructor functions. Given the declaration above, t and f denote all booleans and each element of $\mathbb{N}$ can be written as $succ(\ldots(succ(0)\ldots))$, i.e. the application of zero or more times the application of *succ* to 0.

Auxiliary functions can be defined using the keyword **map**. For instance the standard functions on the sort **Bool** can be declared as follows:

map $\wedge, \vee : \mathbf{Bool} \times \mathbf{Bool} \rightarrow \mathbf{Bool}$
$\neg : \mathbf{Bool} \rightarrow \mathbf{Bool}$

In a textual exposition we use symbols such as $\wedge$ and $\vee$ and use these in the normal mathematical way. The language *μ*CRL which the tools can handle assume that all these functions are written in standard ASCII symbols and are all prefix. Thus a term $t \wedge u$ is written as '`and(t,u)`' to be consumable by the tool set.

Auxiliary functions do not characterise the structure of sorts. A domain D that has no constructor functions with D as target sort can in principle have any number (> 0) of elements, even uncountably many. This means that it is not possible to denote all elements of such a sort with a term.

Properties of functions can be determined by simple unconditional equations. A well-known characterisation of the functions for booleans is written as follows:

var $b : \mathbf{Bool}$
rew $\mathsf{t} \wedge b = b$
$\mathsf{f} \wedge b = \mathsf{f}$
$\mathsf{t} \vee b = \mathsf{t}$
$\mathsf{f} \vee b = b$
$\neg \mathsf{t} = \mathsf{f}$
$\neg \mathsf{f} = \mathsf{t}$

With the keyword **var** variables are declared to be used in the equations. The keyword **rew** comes from the verb *rewrite* and was chosen because rewriting technology was considered the way to prove data terms in *μ*CRL equal. This term is somewhat misleading. Despite the abundant use of rewriting technology in the tools, the equations following the keyword **rew** must be considered as a set of plain unordered equations. Its meaning does not change if left and right hand side of an equation is exchanged or if the relative position of each equation in the **rew** section is altered.

This is basically all to know about the data part of *μ*CRL. Despite its simplicity, the language has proven itself totally apt for the specification of all conceivable data types. Its conciseness has a few advantages, namely that the data specification language can be quickly and easily understood and that building tools for it is a relatively easy affair, allowing to concentrate on making the tools very efficient.

There are also disadvantages, namely that basic data types must be defined for each specification. The extra work that it induces is not the real problem. The real problem is the lack of standardisation that it stimulates, i.e. before understanding the intricacies of a process, first the particular way the data types have been defined for this specification must be studied. This also hampers the development of meta knowledge on the data types and the use of dedicated tools.

2.2 The process language

The process language of *μ*CRL is based on the language ACP, Algebra of Communicating Processes [1], which is very similar to other process specification languages, such as for instance CCS, Calculus of Communication Processes [22]. One of the most important concepts of these languages is an action. An *action* is an atomic event of a process that indicates a communication with the outside world or with another process. In the last case this communication takes place synchronously and is also called an interaction. Actions are typically written as a, b, c but can also have more comprehensive names such as *timeout*, *send* and *receive*.

Using the two main operators, namely a dot for *sequential composition* and a plus for *alternative composition*, behaviour of systems can be denoted. The process term $a{\cdot}b{\cdot}c$ indicates the process that can consecutively perform actions a, b and c. The process $a{\cdot}b + c{\cdot}d$ indicates the process that can perform either an action a followed by an action b, or an action c followed by an action d. The choice between the two is made when the first action takes place. The sequential composition operator binds the strongest and is often omitted if the meaning becomes clear from the context.

Using the alternative composition operator nondeterministic processes can be described. For instance the process $a{\cdot}b + a{\cdot}c$ can initially do an a action. But it is undetermined whether it would be able to do a b or a c action afterwards. Expressing nondeterminism is of great value. It allows to abstractly describe processes of which the actual behaviour depends on factors that cannot sufficiently, concisely or conveniently be described.

A special action, denoted by τ. is called the *internal* action. It is an action that cannot directly be observed, for instance because it takes place inside a computer or because it represents a communication between two processes that has been shielded from observers. In order to indicate that actions are shielded the *hiding operator* τ_I where I is a set of actions can be used. The process $\tau_{\{a,c\}}(a{\cdot}b{\cdot}c)$ equals the process $\tau{\cdot}b{\cdot}\tau$. The actions a and c are hidden.

There is a special process called *inaction* or *deadlock*, which is denoted as δ. This is the process that cannot perform any activity. It is used for several purposes. For instance, if it is shown that a process behaves as deadlock, this is often an indication that there is something terribly wrong with the interaction between processes. In order to declare that actions cannot take place, the *encapsulation operator* ∂_H with H a set of actions is used. The process $\partial_{\{a,c\}}(a{\cdot}b{\cdot}c)$ equals $\delta{\cdot}b{\cdot}\delta$. And because δ satisfies the process equation $\delta{\cdot}x = \delta$, this process equals δ.

The *parallel operator* is another feature of the language. The process $a{\cdot}b \parallel c{\cdot}d$ says that the actions $a{\cdot}b$ and $c{\cdot}d$ can be executed in any order (as long as a is before b and c before d). This is called interleaving. In general, if p and q are processes, $p \parallel q$ denotes the process where the actions of p can happen concurrent to the actions of q in an interleaved fashion.

Using a communication declaration it can be indicated how actions can communicate. So, if a, b and c are actions

comm $a \mid b = c$

indicates that the actions a and b in two parallel processes can happen synchronously, and the result is called c. So, to be explicit, the process $a \parallel b$ behaves as $a{\cdot}b + b{\cdot}a + c$. In general, synchronisation must be enforced, i.e. it is not desirable that in $a \parallel b$, the actions a and b can still happen in isolation. The encapsulation operator ∂_H explained above can be used to block the actions a and b. More concretely, $\partial_{\{a,b\}}(a \parallel b)$ behaves as c.

Although we have not yet explained the combination of processes with data, it is useful to know that actions can carry data arguments. A communication between actions can only take place if the data arguments are exactly the same. Furthermore, for multi-party communication, the communication declaration must be commutative and associative. Consult [13] for precise details.

In order to specify iterative behaviour, process equations are used. A process equation of the form

$$X = a{\cdot}X$$

indicates that a process X can perform action a and then behave as X again. In other words, the process X can perform an infinite sequence of a actions. More complex equations can be written down. The process equation $Y = a{\cdot}Y{\cdot}b + c$ characterises a process that can do an arbitrary number of a's, a c and then as many b's as it performed a's. The process equation $Z = a{\cdot}(b \parallel Z)$ characterises a process that can continually execute a's and b's where the number of executed b's never exceeds the number of a's that took place.

2.3 Processes combined with data

The essence of μCRL is that data and processes have been combined. This boils down to four extensions of the process language.

The first extension is that actions can have data. Actions must be declared indicating which data they must have. Assuming the existence of the data sorts **Bool** and $\mathbb{N}$ this can be done as follows:

act $a : \mathbf{Bool} \times \mathbb{N}$
$a, b : \mathbb{N}$
c

Here, three actions are declared. The action a must carry either two parameters of sorts **Bool** and $\mathbb{N}$, or one of sort $\mathbb{N}$. The action b has a parameter of sort $\mathbb{N}$ and the action c has no parameters.

The second extension is that data can be used in process equations. Formally spoken, the process variables now become higher order variables, which induces a whole world of mathematical complexity to the language. However, from a practical perspective, these equations resemble procedure declarations in standard programming language, and their use does not impose any problem.

The data is simply added to the process variable. So, a simple counter can be described as follows:

proc $\mathit{Count}(n{:}\mathbb{N}) = \mathit{up}\cdot\mathit{Count}(\mathit{succ}(n))$

The third extension is needed to let the data influence the flow of events. Therefore the *conditional operator then-if-else*, denoted by $\_\lhd\_\rhd\_$ is added to the language. To show its use, we can extend the counter above to also count down. The function *pred* subtracts one.

$$
\begin{aligned}
\textbf{proc}\quad \mathit{Count}(n{:}\mathbb{N}) = {} & \mathit{up}\cdot\mathit{Count}(\mathit{succ}(n)) + \\
& \mathit{down}\cdot\mathit{Count}(\mathit{pred}(n)) \lhd n>0 \rhd \delta
\end{aligned}
$$

This process has two *summands*, separated by a $+$. Both summands indicate actions that can occur independently. The δ at the else branch of the condition indicates that if $n = 0$ no action can be performed in the second summand.

The fourth extension is the *sum operator*, which in a particular case can look as follows $\sum_{n:\mathbb{N}} a(n)$. This indicates a choice between actions $a(n)$ for all n. This is the same as writing $a(0)+a(\mathit{succ}(0))+a(\mathit{succ}(\mathit{succ}(0)))+\cdots$. However using the binary operator $+$ it is not possible to indicate a choice between an infinite number of options. This is exactly what the sum operator has been designed to do.

Just as an illustration, we can extend the counter above with a set action, that allows to set the counter to any arbitrary value.

$$
\begin{aligned}
\textbf{proc}\quad \mathit{Count}(n{:}\mathbb{N}) = {} & \mathit{up}\cdot\mathit{Count}(\mathit{succ}(n)) + \\
& \mathit{down}\cdot\mathit{Count}(\mathit{pred}(n)) \lhd n>0 \rhd \delta + \\
& \textstyle\sum_{m:\mathbb{N}} \mathit{set}(m)\cdot\mathit{Count}(m)
\end{aligned}
$$

Observe that the sum operator acts as binder, similarly to the λ in the lambda calculus, or quantifiers $\exists,\forall$ in logic.

Except for a few rarely used constructs, we have seen all language elements of the μCRL language. A useful feature that has not been indicated yet, is the **init** keyword, using which the initial state is indicated. For the counter this could work as follows:

init $\mathit{Count}(0)$

2.4 Theory and tools

The μCRL tool set consists of a number of tools that have been used to transform and optimise the μCRL specification of the protocol, and to generate the state space. Names of the tools are given in italics.

The most important tool, called *mcrl*, checks whether a specification is a well-formed μCRL expression. In addition, it transforms the specification to a linear process format. In essence this format consists of a vector of data parameters, encoding the state of the process, and a set of condition-action-effect rules. These rules say under which condition on the data vector the action can be executed. The effect part of the rules indicate how the data must be updated when executing the action.

The linear process format does not contain any parallelism. Parallel operators can be translated away, without substantially blowing up the size of linear processes. The standardized linear form makes it easy to define and implement reduction and analysis tools to manipulate and understand processes.

A series of such reduction tools is applied to simplify the transformed program. One of the tools, called *constelm* eliminates parameters constant through any run of the process. A similar tool, *parelm*, removes parameters that do not influence the behaviour of the system. For instance, in the following definition $P(x{:}\mathbb{N}) = a \cdot P(succ(x))$ the parameter x is superfluous and can be removed. The tool *rewr* simplifies data terms in the linear process by rewriting them using the equations of the data types. The tool *structelm* replaces a term with a constructor as head symbol by a name of the constructor and its arguments. Finally, *sumelm* replaces a variable which are bound by a summation by a data term if the variable is invariantly equal to this data term. For example, $\sum_{d:D}\sum_{f:F} read(d,f) \cdot Q(d,e) \triangleleft f = e \triangleright \delta$ is transformed to $\sum_{d:D} read(d,e) \cdot Q(d,e)$. To improve the effect of the transformations some of them can be repeated. The tool *tbfupdate* replaces action labels in a linear process by others according to some translation file. In Appendix B such a file can be found, prescribing for instance that an action $c_1(t)$ for any term t is replaced by an action *ctau* without parameters. The tool *stategraph* makes a high level analysis of the dependencies between the parameters in states, and makes adaptions to reduce the state space and to make parelm and constelm more effective. These adaptions preserve the behaviour of the process in the sense of strong bisimulation.

The main purpose of the simplifications above is improving the speed of the *instantiator*, a tool generating the state space. Reducing the execution time of the instantiator is essential, since it takes lion's share of the total execution time. In our tests the instantiator has been invoked with the following parameters: *monitor*, allowing us to keep track of the instantiator progress, and *trace NOT_SECRET*, reporting whether the action *NOT_SECRET* corresponding to security violation has been performed. Each test has been performed twice: with *confluent* flag and without it. This flag forces the instantiator to attempt proving that certain actions can be performed in an arbitrary order as their result does not depend on the particular execution order. In such a case, states connected by a confluent hidden action can be identified.

3 Bilateral Key Exchange protocol with public keys

As explained in the introduction, the goal of the Bilateral Key Exchange protocol is that two parties agree upon a freshly generated secret key. In this section we will explain this protocol in detail and discuss the μCRL specification of the protocol.

Figure 1 shows this protocol in a Message Sequence Chart. The two vertical axes represent the two *roles* of the protocol, which are the initiator role I and the responder role R. We list the initial knowledge of each of the roles above the headers of the roles. Thus, the initiator has an asymmetric key pair (SKi, PKi) and knows the public key of the responder PKr. Likewise, the responder has asymmetric key pair (SKr, PKr) and knows the public key of some initiator PKi. The way in which this initial knowledge was established is not made explicit.

The initiator starts by creating a fresh nonce ni. This is a random, unpredictable value which is used to make the exchanged messages unique and thus helps to counter replay attacks. The first message sent by the initiator consists of the pair ni, I which is encrypted with the public key of the intended responder, denoted by $\{ni, I\}_{PKr}$. Encryption is used to guarantee that only the intended recipient can unpack the message. The only messages that the responder accepts have a form $\{ni, I\}_{PKr}$, i.e., they are encrypted by his public key, and they contain a nonce and the identity of the initiator. Upon receipt of the message, the responder creates his own fresh nonce nr and a fresh symmetric key kir that he wants to share with the initiator. The goal of the protocol is to transfer this key to the initiator in a secret way. Therefore, the responder replies with the message $\{h(ni), nr, kir\}_{PKi}$. With this message he proves that he was able to unpack the previous message (by showing that he knows nonce ni, witnessed by sending a hash $h(ni)$ of this nonce). Furthermore, this message contains the key kir and a challenge nr. The complete message is encrypted with the public key of the initiator to ensure that only I can unpack the message. As above, this is the only type of messages accepted by the initiator. Finally, the initiator responds to R's message by sending a hash of nonce nr encrypted with key kir. Herewith he acknowledges receipt of the previous message. At the end of the two roles we have listed the security claims as a special kind of event. Both participants claim that whenever they reach the end of their protocol the value of kir cannot be known to an intruder.

A system executing this protocol consists of a number of agents, each of which may execute one or more instances of both roles (in parallel). When an agent executes a role from a protocol, we call this a run. Therefore, a system consists of a collection of runs exchanging messages to each other.

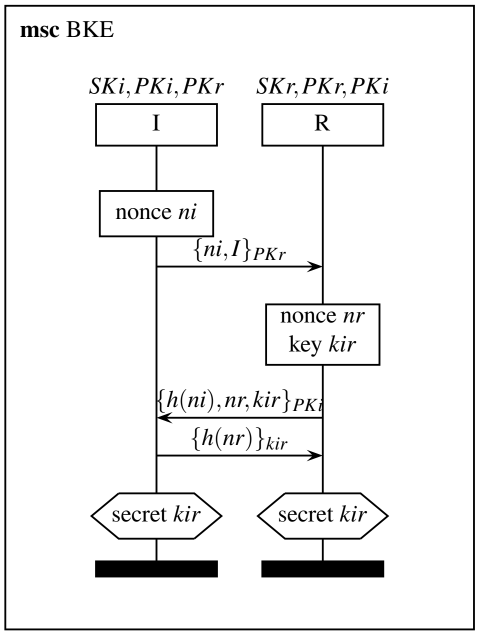

Figure 1: The Bilateral Key Exchange protocol with public keys

The specification of the protocol in μCRL is rather straightforward and is given in Table 1. We require data sorts *Agent*, natural numbers ($\mathbb{N}$), *Nonce* and *Key*. Agents can play different roles simultaneously. To distinguish these roles, each role gets a unique natural number n. Moreover, both roles get an initial agent a with whom the role wants to establish a symmetric key. We use s_I (s_R) and r_I (r_R) to denote sending and receiving messages by the initiator (responder), respectively. The specification is quite straightforward and directly follows the message sequence chart.

$$
\begin{aligned}
&\textbf{proc}\quad I(self{:}Agent, n{:}\mathbb{N}, a{:}Agent) = \\
&\qquad s_I(\{nonce(n), addr(self)\}_{PK(a)}) \cdot \\
&\qquad \textstyle\sum_{nr:Nonce,key:Key} r_I(\{h(nonce(n)), nr, key\}_{PK(self)}) \cdot \\
&\qquad s_I(\{h(nr)\}_{key}) \\
\\
&\textbf{proc}\quad R(self{:}Agent, n{:}\mathbb{N}, a{:}Agent) = \\
&\qquad \textstyle\sum_{ni:Nonce,a:Agent} r_R(\{ni, addr(a)\}_{PK(self)}) \cdot \\
&\qquad s_R(\{h(ni), nonce(n), K(n)\}_{PK(a)}) \cdot \\
&\qquad r_R(\{h(nonce(n))\}_{K(n)})
\end{aligned}
$$

Table 1: Specification of the Initiator and Responder role

4 The intruder

In order to verify correctness of the protocol we need to extend the μCRL specification above by adding an intruder. As explained before, we assume the so-called Dolev-Yao intruder model (see [9]), which is considered the most general model of an adversary. This model implies that the intruder has complete control over the network and that he can derive new messages from his initial knowledge and messages received from honest agents. Hereby we assume that the intruder can only decrypt messages if he is in possession of the appropriate cryptographic key. Furthermore, we assume that a number of agents may conspire with the intruder and try to mislead the honest agents as to learn their secrets. Due to the capabilities of the intruder to intercept any sent message and to insert any message which can be constructed from his knowledge, we can model the existence of conspiring agents by assuming that their secret keys are in the initial knowledge of the intruder.

Now we come back to the Bilateral Key Exchange protocol. The specification from Figure 1 is correct if for any number of agents, executing any number of runs, in the presence of a Dolev-Yao intruder, whenever an honest run enters a secrecy claim, the corresponding key *kir* is never exposed to the intruder.

To verify the protocol the system executes in parallel a number of runs of the initiator, a number of runs of the responder and exactly one run of the intruder. Every run is identified by an agent performing it and a number. Communication between the parties proceeds along four communication channels: from the initiator to the intruder, from the intruder to the initiator, from the responder to the intruder and from the intruder to the responder. In other words, the initiator (the responder) sends messages by performing action s_I (s_R) and reads them by performing action r_I (r_R) (cf. Table 1). Unlike them, the intruder broadcasts messages to all agents (action s_E) and reads any messages present (action r_E). Communication is possible only between s_E and r_I, s_E and r_R, s_I and r_E, and s_R and r_E. In particular, this means that the intruder gets access to all messages circulating in the network.

We also assume that the intruder can always compute the hash function h but given the value $h(x)$ he cannot find x.

In order to start the Bilateral Key Exchange protocol every participating run should know the name of the partner with whom it intends to share the secret. To implement this we require the following preparatory step to be performed: the intruder chooses the identity of the partner for each run and sends it to the corresponding agent. Agents use this information to choose public keys for encryption. It should be noted that, generally speaking, this decision could have been made by the agents themselves. However, delegating this to the intruder makes the process τ-confluent [16] and significantly reduces a number of states. Hence, specifications presented in Table 1 are extended by a preliminary step of reading a name of the intended partner.

Finally, we review different actions that can be performed by the intruder. First of all, it listens on the incoming channels and every time a message arrives, the intruder's knowledge is updated. At each moment of time it consists of two lists: a list of an unencrypted information obtained, and a list of pending messages waiting for decryption. Initially the first list contains addresses of all the agents, their public keys, the secret key of the intruder self and one nonce. One can show that one nonce is sufficient to simulate an intruder having infinitely many nonces. The second list is initially empty. Every time a new message is read, the intruder checks whether it has the corresponding key for decryption. If this is the case the message is decrypted, its contents is added to the list of unencrypted information and the intruder tries to use the contents to decrypt additional messages pending. If the message cannot be decrypted it is simply added to the list of pending messages. To simplify the retrieval of messages given a decryption key we organise the pending messages list as a list of (key, lom)-pairs, where *lom* is a list of messages that require *key* for decryption.

Next, the intruder can claim that the secrecy has been violated, if he possesses some information that has been claimed to be secret by one of the other parties. Finally, he can send a number of different messages:

1. Identity of the agent and a natural number p. This message is used to establish the partner for the run identified by p.
2. An unchanged message received at one of the previous steps.
3. A message imitating the first message of the initiator. It encrypts a nonce and an address.

4. A message imitating the message of the responder. It encrypts a hash of a nonce, a nonce and a symmetric key.

5. A message imitating the second message of the initiator. It contains a hash of a nonce encrypted by a symmetric key.

5 Optimisation and state space generation

Recall that in order to verify the protocol we need to generate the state space. To reduce the size of the state space a number of optimisations have been performed. The first optimisation consists of the use of types, i.e., we assume that a nonce can be always distinguished from an address, etc. Formally, we introduce four types of entities (data sorts): nonces, addresses, symmetric keys, and functional keys. The latter group includes asymmetric keys and hashes. This means that instead of two lists (unencrypted information and pending messages) at each moment of time the intruder has to remember six different lists: four corresponding to four different types of (unencrypted) entities and two lists of pending messages: those encrypted by symmetric keys and those encrypted by functional keys.

Second, scrutinising the protocol we observe that the hash function is applied exclusively to nonces, and that security claims are performed only on symmetric keys.

Third, if the responder communicates with a compromised agent, the last communication step can be omitted. Indeed, it neither changes the knowledge of the adversary, nor can it lead to an additional security claim of the responder.

Fourth, we keep track of a number of steps the intruder can perform. Each initiator or responder run can perform at most three read and send actions in the original Bilateral Key Exchange protocol and one more read action during the preliminary step. In other words, given a number of processes n there is no need for the intruder to send more than $4n$ messages.

The final group of the optimisations considers the preliminary step. First of all, a number of messages during this step is limited by the number of participating processes. Hence, we add a new parameter p to our implementation of the intruder. We initialise p by the number of participating processes and decrement it at each iteration.

Next, recall that in order for a secrecy claim to be made the correspondent of a run should be different from the intruder. Since we are interested in finding secrecy violations, we require that the correspondent of *at least one* run is not the intruder. We enforce this by requiring that the last considered run ($p = 1$) communicates with a trusted party.

The last optimisation makes use of the particular form of the verification tests. Our first assumption is that there are three different agents: the one that performs the role of the intruder (E), the one that performs the role of the initiator (A) and the one that performs the role of the responder (B). Observe that the protocol is essentially symmetric, i.e., it is of no importance whether the initiator run performed by A with the identifier 1 communicates with B and the initiator run performed by A with the identifier 3 communicates with E or vice versa. Hence, we can order the agents ($E < A < B$) and impose the requirement that communication partners of the initiator runs of the same agent increase with respect to this ordering. The same can be said for the respondent runs of the same agent. To implement this observation we added two more parameters to our implementation of the intruder, namely aI and aR, such that the communication partner intended for an initiator (a responder) run should be greater or equal to aI (aR). In our tests, processes with odd numbers were performed by agent A and played the initiator role, while processes with even numbers were performed by B and played the responder role. Hence, $even(p)$ allows the intruder to distinguish between initiator and responder runs.

The following fragment of the intruder implementation summarises the discussion above:

$$
\begin{array}{l}
\textbf{proc} \quad E(K{:}Knowledge, p{:}\mathbb{N}, aI{:}Agent, aR{:}Agent) = \\
\qquad \ldots \\
\qquad \sum_{a:Agent}(\\
\qquad\qquad (s_E(a,p) \cdot E(K, P(p), aI, \mathit{if}(p=2,E,a)) \lhd 0 < p \wedge a \geq aR \rhd \delta) \\
\qquad \lhd even(p) \rhd \\
\qquad\qquad (s_E(a,p).E(K, P(p), \mathit{if}(p=1,E,a), aR) \\
\qquad\qquad\qquad \lhd 0 < p \wedge a \geq aI \wedge ((p=1 \wedge aI = E) \rightarrow a \neq E) \rhd \delta) \\
\qquad)+ \\
\qquad \ldots
\end{array}
$$

Table 2: Specification of the Intruder role (fragment)

Machine	Confluence	Max. number of processes	Number of states
Stand-alone	no	6	2094438
Stand-alone	dynamic	7	2223873
Stand-alone	static	8	14229495
Parallel	dynamic	7	2223873
Parallel	static	9	111655379

Table 3: Summary of results

6 Experimental evaluation

In this section we evaluate our approach experimentally. To do so, we have performed a series of tests. In these tests two questions have been studied. First, we were interested in the feasibility of our approach in terms of the state space size. Second, the impact of confluence on the state space size has been considered. To do so, we have performed five groups of tests: without confluence elimination on a stand-alone machine, with dynamic confluence elimination on a stand-alone machine, with static confluence elimination of a stand-alone machine, with dynamic confluence elimination on a parallel system and with static confluence elimination on a parallel system. The stand-alone machine had an Intel®Xeon CPU 2.40GHz processor with 4GB memory running Linux version 2.4.20. As a parallel system dual Athlon MP1600 nodes were used[1].

Tests of the following form have been performed for a number of processes ranging from one to nine:

$$
\partial_{\{r_E, s_I, r_I, s_E, s_R, r_R, secret, secret\_\}} \quad (I(A,1) \parallel R(B,2) \parallel I(A,3) \parallel R(B,4) \parallel \ldots \\
\parallel E(initialKnowledge, n, E, E)),
$$

where *secret* and *secret_* are used to express the secrecy claims. Results are summarised in Table 3.

The way μCRL tools have been applied to produce the state space and to verify correctness of the protocol is represented in Table 4. The μCRL implementation of the BKE and the updatefile can be found in Appendices A and B. The following should be observed. The tool *structelm* is called to expand the sort *Message*. This is the sort implementing a number of different kinds of messages circulating in the network. In combination with *rewr -case structelm* allows the system to distinguish between them and to generate simplified expressions for each one of the cases.

It is well known [16] that confluent sets of transitions can be used to trim the state space. We aimed to understand the significance of the space reduction introduced by confluence exploration. Hence, for each run

[1]Tests on a parallel system have been perfromed by Stefan Blom at CWI.

No confluence elimination

```
mcrl -regular -nocluster -newstate security
tbfupdate -edit updatefile security | rewr | stategraph | constelm | parelm > temp0.tbf
structelm -expand Message temp0.tbf | rewr -case | sumelm |
        parelm | stategraph | constelm | parelm > temp1.tbf
time instantiator -trace NOT_SECRET -monitor temp1
```

Static confluence elimination

```
mcrl -regular -nocluster -newstate security
tbfupdate -edit updatefile security | rewr | stategraph | constelm | parelm > temp0.tbf
structelm -expand Message temp0.tbf | rewr -case | sumelm |
        parelm | stategraph | constelm | parelm > temp1.tbf
time instantiator -confluent ctau -monitor -trace NOT_SECRET temp1
```

Dynamic confluence elimination

```
#!/bin/bash

for i in 1 2 3 4 5 6 7 8 9 10 ; do
        echo processing system$i.mcrl
        m4 -DSYSTEM=System$i security24.m4 > system$i.mcrl
        mcrl -regular -nocluster -newstate system$i.mcrl
        tbfupdate -edit updatefile20 system$i.tbf | rewr | stategraph | constelm | parelm > system$i-1.tbf
        confelm system$i-1.tbf | stategraph | constelm | confelm | stategraph | constelm > system$i-2.tbf
```

Table 4: Tools applied

we recorded the size of the state space obtained without confluence exploration, with dynamic confluence exploration (*-confluent*), and static confluence elimination (*confelm*). As expected, the size of the state space was independent of a machine it has been generated upon. However, using a parallel system enlarged the memory size. Hence, we were able to verify the protocol for nine runs—a task for which the same algorithm on a stand-alone machine runs out of memory. Experimental results are summarised in Figure 2. Asterisks mark reference points obtained when no confluence has been explored, circles correspond to dynamic confluence elimination, and triangles to static confluence elimination. Note that a logarithmic scale is used.

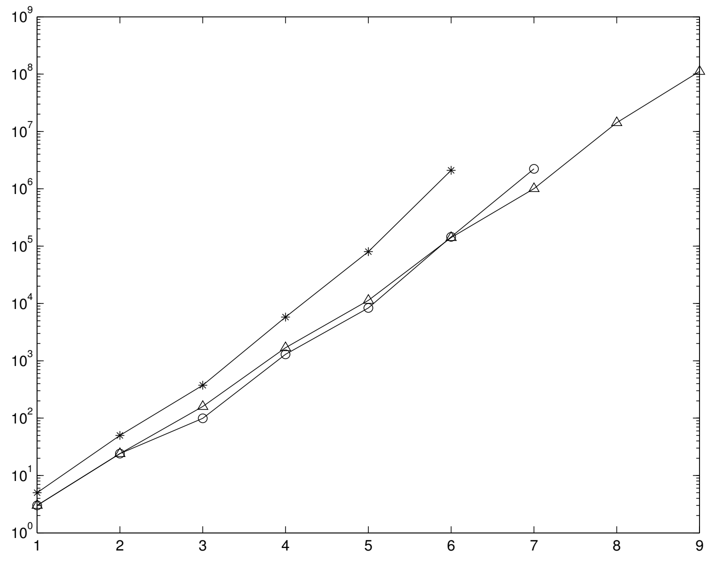

Figure 2: State spaces as function of the number of runs

We observe that in all cases the state space turns out to be exponential in the number of runs. Approximation functions found by Matlab exponential fitting are $10^{1.117x-0.5647}$, $10^{0.9677x-0.6716}$, $10^{0.952x-0.5717}$ for no confluence, dynamic confluence and static confluence, respectively. These results also show that the state size reduction introduced by using confluence elimination techniques is exponential.

7 Conclusions

In this section we summarise the main lessons learned from our experience with implementing the BKE algorithm in μCRL and evaluating it experimentally. First of all, discovering knowledge of the experts remains a difficult task. The μCRL implementation has been modified a number of times to comply to it. Second, we have observed that general modeling tools are well-suited to verify correctness of communication protocols. Finally, our experiments suggest that eliminating confluent transactions can be essential for verification, and that static confluence elimination can outperform dynamic confluence elimination for large state spaces.

References

[1] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science* 37(1):77–121, 1985.

[2] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specifications. In proceedings CAV'01. LNCS 2102, pages 250–254, 2001.

[3] S.C.C. Blom and S. Orzan. Distributed branching bisimulation reduction of state spaces. In L. Brim and O. Grumberg, editors, PDMC 2003: Parallel and Distributed Model Checking, volume 89 of Electronic Notes in Theoretical Computer Science. Elsevier, 2003.

[4] S.C.C. Blom and J.C. van de Pol, State space reduction by proving confluence. In E. Brinksma and K.G. Larsen, editors, Proceedings of CAV'02, LNCS 2404, Springer Verlag, 2002.

[5] E. Brinksma, A tutorial on LOTOS. In M. Diaz, editor, proceedings of the IFIP WG6.1 Fifth International Conference on Protocol Specification, Testing and Verification, pages 171–194, Elsevier, 1986.

[6] M. Burrows, M. Abadi, and R. Needham, A logic of authentication. *ACM Transactions on Computer Systems* 8(1):18–36, 1990.

[7] E. Clarke, O. Grumberg and D. Long. Verification tools for finite-state concurrent systems. In A Decade of Concurrency – Reflections and Perspectives. LNCS 803, Springer Verlag, 1994.

[8] J.A. Clark and J.L. Jacob. A survey of authentication protocol literature. Technical Report 1.0, 1997.

[9] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory* 29(12):198–208, 1983.

[10] A. David, G. Behrmann, K.G. Larsen and Wang Yi. A tool architecture for the next generation of UPPAAL. Technical report. Uppsala University 2003.

[11] H. Garavel. OPEN/CAESAR: An open software architecture for verification, simulation, and testing. Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98, 1998.

[12] H. Garavel, F. Lang, R. Mateescu. An overview of CADP 2001. European Association for Software Science and Technology (EASST) Newsletter volume 4, pages 13–24, 2002.

[13] J.F. Groote and J.C. van de Pol. Equational binary decision diagrams. In M. Parigot and A. Voronkov, Logic for Programming and Reasoning, LPAR2000, LNAI 1955, Springer Verlag, pages 161–178, 2000.

[14] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef and S.F.M. van Vlijmen, editors, Algebra of Communicating Processes, Workshops in Computing, pages 26–62, 1994.

[15] J.F. Groote and M. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse and S.A. Smolka. Handbook of Process Algebra, pages 1151–1208, Elsevier, Amsterdam, 2001.

[16] J.F. Groote and M.P.A. Sellink. Confluence for process verification. *Theoretical Computer Science B (Logic, semantics and theory of programming)* 170(1-2):47–81, 1996.

[17] J.F. Groote and T.A.C. Willemse. A Checker for modal formulas for processes with data, Eindhoven University of Technology, Department of Computer Science, CSR 02-16, 2002. To appear in Theoretical Computer Science.

[18] F. van Ham, H. van de Wetering and J.J. van Wijk. Interactive visualization of state transition systems. IEEE Transactions on Visualization and Computer Graphics 8(4):319–329, 2002.

[19] G.J. Holzmann. The spin model checker: Primer and reference manual. Addison-Wesley, 2003.

[20] G. Lowe, Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS 1055, Springer-Verlag, March 1996.

[21] S. Mauw and G.J. Veltink. A process specification formalism. In Fundamenta Informaticae XIII (1990), pages 85–139, IOS Press, 1990

[22] R. Milner, Communication and concurrency, Prentice Hall, 1989.

[23] F. Moller and P. Stevens. The Edinburgh concurrency workbench user manual. http://www.dcs.ed.ac.uk/home/cwb. 1996.

[24] J.C. van de Pol and M.A. Valero Espada. Modal abstractions in μCRL. In C. Rattray, S. Maharaj and C. Shankland editors, Proceedings of Algebraic Methodology and Software Technology (AMAST'04), Stirling, Scotland, LNCS 3116, Springer Verlag, 2004.

[25] A. Stump, C.W. Barrett, and D.L. Dill. CVC: a cooperating validity checker. In E. Brinksma and K.G. Larsen, editors, Proceedings of CAV 2002, LNCS 2404, Springer Verlag, 2002.

[26] Y.S. Usenko. Linearization in μCRL. PhD. Thesis. Eindhoven University of Technology, 2002.

A μCRL-implementation of the BKE protocol

```
% This mCRL model consists of two major parts: a data description
% and a processes description.
% First we introduce the Booleans including the basic operations.

sort Bool
func T,F:->Bool
map  and,or:Bool#Bool->Bool
     not:Bool->Bool
     eq:Bool#Bool->Bool
     if:Bool#Bool#Bool->Bool
     imply:Bool#Bool->Bool

var  x,y:Bool
rew  and(T,x)=x      or(T,x)=T     imply(T,T)=T
     and(x,T)=x      or(x,T)=T     imply(T,F)=F
     and(x,F)=F      or(x,F)=x     imply(F,T)=T
     and(F,x)=F      or(F,x)=x     imply(F,F)=T

     eq(x,T)=x       if(T,x,y)=x   not(F)=T
     eq(T,x)=x       if(F,x,y)=y   not(T)=F
     eq(F,x)=not(x)  if(x,y,y)=y
     eq(x,F)=not(x)

% Secondly, we require the natural numbers with 0 and the successor
% S as constructors. This means that each natural number can be
% written as S(S(...(0)..)).
% As a shorthand we allow to write digits 1, 2, etc. eq stands for
% equality, and sm for smaller than. P is the predecessor.

sort Nat
func 0:->Nat
     S:Nat->Nat
map  eq:Nat#Nat->Bool
     sm:Nat#Nat->Bool
     plus:Nat#Nat->Nat
     1,2,3,4,5,6,7,8,9:->Nat
     P:Nat->Nat
     even:Nat->Bool

var  n,m:Nat
rew  eq(n,n)=T         sm(n,n)=F         eq(S(n),S(m))=eq(n,m)
     eq(S(n),0)=F      sm(n,0)=F         sm(S(n),S(m))=sm(n,m)
     eq(0,S(m))=F      sm(0,S(m))=T      plus(S(n),m)=S(plus(n,m))
     plus(0,n)=n       plus(n,0)=n       plus(n,S(m))=S(plus(n,m))

     even(0)=T
     even(S(n))=not(even(n))

     P(S(n))=n

     1=S(0)     4=S(3) 7=S(6)
     2=S(1)     5=S(4) 8=S(7)
     3=S(2)     6=S(5) 9=S(8)

% Agents. There are exactly three agents - A, B, E.
% We use an ordering on the agents which is defined as E < A < B.
% It is used to reduce the state space which is generated.

sort  Agent
func  A,B,E:->Agent
map   eq:Agent#Agent->Bool
      sm:Agent#Agent->Bool
      if:Bool#Agent#Agent->Agent

var   a,a':Agent
rew   eq(a,a)=T
      eq(A,B)=F      eq(B,A)=F      eq(E,A)=F
      eq(A,E)=F      eq(B,E)=F      eq(E,B)=F

      if(T,a,a')=a
      if(F,a,a')=a'

      sm(A,B)=T      sm(B,B)=F      sm(E,B)=T
      sm(A,E)=F      sm(B,E)=F      sm(E,E)=F
      sm(A,A)=F      sm(B,A)=F      sm(E,A)=T

% There are two kinds of keys used in the protocol: symmetric and
% asymmetric ones (functional keys).
% Symmetric keys have form K(n) where n is a natural number.
% On symmetric keys we define equality eq and smaller than sm.

sort  symKEY
map   K:Nat->symKEY
      eq:symKEY#symKEY->Bool
      sm:symKEY#symKEY->Bool

var   n,m:Nat
rew  eq(K(n),K(m)) = eq(n,m)
     sm(K(n),K(m)) = sm(n,m)

% A nonce is a random, unpredictable value which is used to make the
% exchanged messages unique and thus helps to counter replay attacks.
```

```
sort  NONCE
map   nonce:Nat->NONCE
      eq:NONCE#NONCE->Bool
      sm:NONCE#NONCE->Bool

var   n,m:Nat
rew   eq(nonce(n),nonce(m)) = eq(n,m)
      sm(nonce(n),nonce(m)) = sm(n,m)

% Address is either addr(a) for some agent a or
% a special value badAddr, which is required to make address:funKEY->ADDR
% to a total function (see below). We require again the equality and
% smaller than functions.

sort  ADDR
map   addr:Agent->ADDR
      badAddr:->ADDR
      eq:ADDR#ADDR->Bool
      sm:ADDR#ADDR->Bool

var   a1,a2:Agent
rew   eq(badAddr,badAddr)  = F
      eq(badAddr,addr(a2)) = F
      eq(addr(a1),badAddr) = F
      eq(addr(a1),addr(a2))= eq(a1,a2)

      sm(badAddr,badAddr)  = F
      sm(badAddr,addr(a2)) = T
      sm(addr(a1),addr(a2))= sm(a1,a2)
      sm(addr(a1),badAddr) = F

% The sort funKEY contains the second kind of keys, functional keys.
% Functional keys are public keys (PK) and secret keys (SK). Furthermore,
% functional keys can be the result of applying a hash function (h) to
% some nonce. For a public key k, the function decode_key returns the
% corresponding secret key. For a given secret key, it returns the
% corresponding public key. Otherwise decode_key returns a special value
% NOKEY. For a public key, the function address returns the address of the
% owner of the key. Otherwise it returns a special value badAddr.
% The constant NOKEY is needed to make decodeKEY:funKEY->funKEY to a
% total function as stated above.

sort funKEY
map   PK,SK:Agent->funKEY
      h:NONCE->funKEY
      eq:funKEY#funKEY->Bool
      sm:funKEY#funKEY->Bool
      decodeKEY:funKEY->funKEY
      address:funKEY->ADDR
      NOKEY :->funKEY

var   a1,a2:Agent
      n1,n2: NONCE
rew   eq(NOKEY,NOKEY)   = T         eq(PK(a1),NOKEY)  = F
```

```
      eq(NOKEY,PK(a2))  = F         eq(PK(a1),PK(a2)) = eq(a1,a2)
      eq(NOKEY,SK(a2))  = F         eq(PK(a1),SK(a2)) = F
      eq(NOKEY,h(n2))   = F         eq(PK(a1),h(n2))  = F

      eq(SK(a1),NOKEY)  = F         eq(h(n1),NOKEY)   = F
      eq(SK(a1),PK(a2)) = F         eq(h(n1),h(n2))   = eq(n1,n2)
      eq(SK(a1),SK(a2)) = eq(a1,a2) eq(h(n1),PK(a1))  = F
      eq(SK(a1),h(n2))  = F         eq(h(n1),SK(a1))  = F

      sm(NOKEY,NOKEY)   = F         sm(PK(a1),NOKEY)  = F
      sm(NOKEY,PK(a2))  = T         sm(PK(a1),PK(a2)) = sm(a1,a2)
      sm(NOKEY,SK(a2))  = T         sm(PK(a1),SK(a2)) = T
      sm(NOKEY,h(n2))   = T         sm(PK(a1),h(n2))  = T

      sm(SK(a1),NOKEY)  = F         sm(h(n1), NOKEY)  = F
      sm(SK(a1),PK(a2)) = F         sm(h(n1),h(n2))   = sm(n1,n2)
      sm(SK(a1),SK(a2)) = sm(a1,a2) sm(h(n1),PK(a1))  = F
      sm(SK(a1),h(n2))  = T         sm(h(n1),SK(a1))  = F

      decodeKEY(PK(a1)) = SK(a1)    address(PK(a1)) = addr(a1)
      decodeKEY(SK(a1)) = PK(a1)    address(h(n1)) = badAddr
      decodeKEY(h(n1)) = NOKEY      address(SK(a1)) = badAddr
      decodeKEY(NOKEY) = NOKEY      address(NOKEY) = badAddr

% The eavesdropper's knowledge (see sort Knowledge below) consists of
% six different lists of information. One of them is a sorted list of
% symmetric keys obtained by the intruder, given in the sort symKEYList.
% Initially the list is empty. The function add adds a new element to
% the list, isIn is a membership test (required only for keys).
% The expression get(n,l) gets the n'th symmetric key of the symmetric
% key list l

sort  symKEYList
func  emptysymKEYList:->symKEYList
      insymKEY:symKEY#symKEYList->symKEYList
map   add:symKEY#symKEYList->symKEYList
      isIn:symKEY#symKEYList->Bool
      isInAUX:Bool#symKEY#symKEYList->Bool
      if:Bool#symKEYList#symKEYList->symKEYList
      length:symKEYList->Nat
      get:Nat#symKEYList->symKEY
      initialsymKEYList:->symKEYList

var   k1,k2:symKEY
      l1,l2:symKEYList
      n:Nat

rew   add(k1,emptysymKEYList)=insymKEY(k1,emptysymKEYList)
      add(k1,insymKEY(k2,l1))=
          if(sm(k1,k2),insymKEY(k1,insymKEY(k2,l1)),
             if(eq(k1,k2),insymKEY(k2,l1),
                          insymKEY(k2,add(k1,l1))))

      isIn(k1,emptysymKEYList)=F
      isIn(k1,insymKEY(k2,l1))=
           if(sm(k1,k2),F,isInAUX(eq(k1,k2),k1,l1))

      isInAUX(T,k1,l1) = T
      isInAUX(F,k1,l1) = isIn(k1,l1)

      if(T,l1,l2)=l1
      if(F,l1,l2)=l2

      length(emptysymKEYList)=0
      length(insymKEY(k1,l1))=S(length(l1))

      get(0,insymKEY(k1,l1))=k1
      get(S(n),insymKEY(k1,l1))=get(n,l1)

      initialsymKEYList = emptysymKEYList

% A sorted list of nonces is also a part of the eavesdropper's knowledge.
% Initially the eavesdropper posesses one nonce. One can show that
% this is sufficient to simulate an eavesdropper with infinitely many
% nonces.

sort  NONCEList
func  emptyNONCEList:->NONCEList
      inNONCE:NONCE#NONCEList->NONCEList
map   add:NONCE#NONCEList->NONCEList
      if:Bool#NONCEList#NONCEList->NONCEList
      length:NONCEList->Nat
      get:Nat#NONCEList->NONCE
      initialNONCEList:->NONCEList

var   k1,k2:NONCE
      l1,l2:NONCEList
      n:Nat
rew   add(k1,emptyNONCEList)=inNONCE(k1,emptyNONCEList)
      add(k1,inNONCE(k2,l1))=
         if(sm(k1,k2),inNONCE(k1,inNONCE(k2,l1)),
                if(eq(k1,k2),inNONCE(k2,l1),
                             inNONCE(k2,add(k1,l1))))

      if(T,l1,l2)=l1
      if(F,l1,l2)=l2

      length(emptyNONCEList)=0
      length(inNONCE(k1,l1))=S(length(l1))

      get(0,inNONCE(k1,l1))=k1
      get(S(n),inNONCE(k1,l1))=get(n,l1)

      initialNONCEList = add(nonce(0), emptyNONCEList)

% A sorted list of addresses is also a part of intruder's knowledge.
% All addresses are public. Hence, the initial knowledge of the
% intruder consists of the addresses of all the agents.

sort  ADDRList
```

```
func  emptyADDRList:->ADDRList
      inADDR:ADDR#ADDRList->ADDRList
map   add:ADDR#ADDRList->ADDRList
      if:Bool#ADDRList#ADDRList->ADDRList
      length:ADDRList->Nat
      get:Nat#ADDRList->ADDR
      initialADDRList:->ADDRList

var   k1,k2:ADDR
      l1,l2:ADDRList
      n:Nat
rew   add(k1,emptyADDRList)=inADDR(k1,emptyADDRList)
      add(k1,inADDR(k2,l1))=
         if(sm(k1,k2),inADDR(k1,inADDR(k2,l1)),
                 if(eq(k1,k2),inADDR(k2,l1),
                              inADDR(k2,add(k1,l1))))

      if(T,l1,l2)=l1
      if(F,l1,l2)=l2

      length(emptyADDRList)=0
      length(inADDR(k1,l1))=S(length(l1))

      get(0,inADDR(k1,l1))=k1
      get(S(n),inADDR(k1,l1))=get(n,l1)

      initialADDRList = add(addr(A), add(addr(B),
                             add(addr(E), emptyADDRList)))

% The intruder also keeps a sorted list of functional keys.
% Initially all public keys are known to the intruder. And
% naturally the eavesdropper initially knows its own
% secret key.

sort  funKEYList
func  emptyfunKEYList:->funKEYList
      infunKEY:funKEY#funKEYList->funKEYList
map   add:funKEY#funKEYList->funKEYList
      isIn:funKEY#funKEYList->Bool
      isInAUX:Bool#funKEY#funKEYList->Bool
      if:Bool#funKEYList#funKEYList->funKEYList
      length:funKEYList->Nat
      get:Nat#funKEYList->funKEY
      initialfunKEYList :->funKEYList

var   k1,k2:funKEY
      l1,l2:funKEYList
      n:Nat
rew   add(k1,emptyfunKEYList)=infunKEY(k1,emptyfunKEYList)
      add(k1,infunKEY(k2,l1))=
         if(sm(k1,k2),infunKEY(k1,infunKEY(k2,l1)),
                 if(eq(k1,k2),infunKEY(k2,l1),
                              infunKEY(k2,add(k1,l1))))

      isIn(k1,emptyfunKEYList)=F
      isIn(k1,infunKEY(k2,l1))=
         if(sm(k1,k2),F,isInAUX(eq(k1,k2),k1,l1))

      isInAUX(T,k1,l1) = T
      isInAUX(F,k1,l1) = isIn(k1,l1)

      if(T,l1,l2)=l1
      if(F,l1,l2)=l2

      length(emptyfunKEYList)=0
      length(infunKEY(k1,l1))=S(length(l1))

      get(0,infunKEY(k1,l1))=k1
      get(S(n),infunKEY(k1,l1))=get(n,l1)

      initialfunKEYList = add(PK(A), add(PK(B),
          add(PK(E), add(SK(E), emptyfunKEYList))))

% The sort Message contains all the messages that are exchanged
% in the BKE protocol. There three kinds of messages:
% two sent by the initiator and one sent by the responder.
% The second message sent by an initiator consists of a functional
% key encrypted by a symmetric key. The first message sent by an
% initiator contains a nonce and an address encrypted by a
% functional key. The message sent by the receiver contains a
% functional key, a nonce, a symmetric key and it is encrypted by
% a functional key. The functions get... are projections to extract
% information from the messages.
% The functions encryptedN (N=1,2,3) indicate whether a message
% has a particular form. The function encrypted1 is true if its
% first argument has the form of the second I-message and the key
% used during the encryption is its second argument.
% The function encrypted2 is true if its first argument has the form
% of the first I-message and the key used during the encryption
% is its second argument. The function encrypted3 is true if its
% first argument has the form of the R-message and the key used
% during the encryption is its second argument. Two messages are equal,
% expressed using the function eq iff they are of the same kind
% and all components are equal.
% The function sm defines an ordering on messages. This ordering is
% defined as follows:
% for any fk1,sk2,n1,a2,fk3,fk1',n2,sk3 and fk4
%       encrypt(fk1,sk2) < encrypt(n1,a2,fk3) < encrypt(fk1',n2,sk3,fk4).
% Messages of the same kind are compared lexicographically.


sort  Message
func  encrypt:funKEY#symKEY->Message
      encrypt:NONCE#ADDR#funKEY->Message
      encrypt:funKEY#NONCE#symKEY#funKEY->Message
map   eq:Message#Message->Bool
      get1N:Message->NONCE
      get1f:Message->funKEY
      get2N:Message->NONCE
      get2A:Message->ADDR
      get3s:Message->symKEY
      sm:Message#Message->Bool
      encrypted1:Message#symKEY->Bool
      encrypted2:Message#funKEY->Bool
      encrypted3:Message#funKEY->Bool

var   fk1,fk1',fk3,fk3',fk4,fk4',fk5 :funKEY
      sk2,sk2',sk3,sk3',sk4,sk5:symKEY
      n1,n1',n2,n2':NONCE
      a2,a2':ADDR
rew   eq(encrypt(fk1,sk2),encrypt(n1',a2',fk3'))=F
      eq(encrypt(fk1,sk2),encrypt(fk1',n2',sk3',fk4'))=F
      eq(encrypt(n1,a2,fk3),encrypt(fk1',sk2'))=F
      eq(encrypt(n1,a2,fk3),encrypt(fk1',n2',sk3',fk4'))=F
      eq(encrypt(fk1,n2,sk3,fk4),encrypt(fk1',sk2'))=F
      eq(encrypt(fk1,n2,sk3,fk4),encrypt(n1',a2',fk3'))=F
      eq(encrypt(fk1,sk2),encrypt(fk1',sk2'))=
        and(eq(fk1,fk1'),eq(sk2,sk2'))
      eq(encrypt(n1,a2,fk3),encrypt(n1',a2',fk3'))=
        and(eq(n1,n1'),
        and(eq(a2,a2'), eq(fk3, fk3')))
      eq(encrypt(fk1,n2,sk3,fk4),encrypt(fk1',n2',sk3',fk4'))=
        and(eq(fk1,fk1'),
        and(eq(n2,n2'),
        and(eq(sk3,sk3'), eq(fk4, fk4'))))

      get1f(encrypt(fk1,sk2))=fk1
      get1f(encrypt(fk1,n2,sk3,fk4))=fk1
      get1N(encrypt(n1,a2,fk3))=n1
      get2A(encrypt(n1,a2,fk3))=a2
      get2N(encrypt(fk1,n2,sk3,fk4))=n2
      get3s(encrypt(fk1,n2,sk3,fk4))=sk3

      sm(encrypt(fk1,sk2),encrypt(fk1',sk2'))=
        or(sm(fk1,fk1'),
              and(eq(fk1,fk1'),sm(sk2,sk2')))
      sm(encrypt(fk1,sk2),encrypt(n1',a2',fk3'))=T
      sm(encrypt(fk1,sk2),encrypt(fk1',n2',sk3',fk4'))=T
      sm(encrypt(n1,a2,fk3),encrypt(fk1',sk2'))=F
      sm(encrypt(n1,a2,fk3),encrypt(n1',a2',fk3'))=
          or(sm(n1,n1'),
             and(eq(n1,n1'),
               or(sm(a2,a2'),
                    and(eq(a2,a2'),
                    sm(fk3,fk3')))))
      sm(encrypt(n1,a2,fk3),encrypt(fk1',n2',sk3',fk4'))=T
      sm(encrypt(fk1,n2,sk3,fk4),encrypt(fk1',sk2'))=F
      sm(encrypt(fk1,n2,sk3,fk4),encrypt(n1',a2',fk3'))=F
      sm(encrypt(fk1,n2,sk3,fk4),encrypt(fk1',n2',sk3',fk4'))=
          or(sm(fk1,fk1'),
             and(eq(fk1,fk1'),
                 or(sm(n2,n2'),
                      and(eq(n2,n2'),
                          or(sm(sk3,sk3'),
                             and(eq(sk3,sk3'),
                           sm(fk4,fk4')))))))

      encrypted1(encrypt(fk1,sk2),sk3)=eq(sk2,sk3)
      encrypted1(encrypt(n1,a2,fk3),sk4)=F
      encrypted1(encrypt(fk1,n2,sk3,fk4),sk5)=F

      encrypted2(encrypt(fk1,sk2),fk3)=F
      encrypted2(encrypt(n1,a2,fk3),fk4)=eq(fk3,fk4)
      encrypted2(encrypt(fk1,n2,sk3,fk4),fk5)=F

      encrypted3(encrypt(fk1,sk2),fk3)=F
      encrypted3(encrypt(n1,a2,fk3),fk4)=F
      encrypted3(encrypt(fk1,n2,sk3,fk4),fk5)=eq(fk4,fk5)

% The sort MessageList contains sorted list of messages obtained by
% the intruder. It has the same operations as the lists of nonces
% and addresses. We do not store the messages directly in the intruder's
% knowledge but distinguish between those encoded by symmetric keys and
% and those encoded by the functional keys (see ConditionalfunKEYList and
% ConditionalsymKEYList below).

sort  MessageList
func  emptyMessageList:->MessageList
      inMessage:Message#MessageList->MessageList
map   add:Message#MessageList->MessageList
      if:Bool#MessageList#MessageList->MessageList
      length:MessageList->Nat
      get:Nat#MessageList->Message

var   m1,m2:Message
      l1,l2:MessageList
      n:Nat
rew   add(m1,emptyMessageList)=inMessage(m1,emptyMessageList)
      add(m1,inMessage(m2,l1))=
          if(sm(m1,m2),inMessage(m1,inMessage(m2,l1)),
             if(eq(m1,m2),inMessage(m2,l1),
                          inMessage(m2,add(m1,l1))))

      if(T,l1,l2)=l1
      if(F,l1,l2)=l2

      length(emptyMessageList)=0
      length(inMessage(m1,l1))=S(length(l1))

      get(0,inMessage(m1,l1))=m1
      get(S(n),inMessage(m1,l1))=get(n,l1)

% The sort funEncaps is an auxiliary sort, needed in ConditionalfunKEYList.
% It represents a pair of (ConditionalfunKEYList,MessageList).

sort funEncaps
func  encaps:ConditionalfunKEYList#MessageList->funEncaps
map  pr1: funEncaps->ConditionalfunKEYList
     pr2: funEncaps->MessageList
     if:Bool#funEncaps#funEncaps->funEncaps
```

```
var  fcl:ConditionalfunKEYList
     ml:MessageList
     fe1,fe2:funEncaps
rew  pr1(encaps(fcl,ml)) = fcl
     pr2(encaps(fcl,ml)) = ml
     if(T,fe1,fe2)=fe1
     if(F,fe1,fe2)=fe2

% The list of the messages received by the intruder that are indexed by
% functional keys is given in the sort ConditionalFunKEYList. The list has
% the form <fk1,l1,fk2,l2,...> where each li is a list of messages
% received so far that are encrypted by the preceeding
% functional key fki. The function addCond adds a pair (funKEY,Message)
% to an existing list ConditionalfunKEYList and returns an updated list.
% The function removeCond removes messages encoded by funKEY from the
% list ConditionalfunKEYList. It returns a pair encoded by the sort
% funEncaps: an updated list ConditionalfunKEYList and a list of messages
% corresponding to funKEY in the original list. If no messages encoded
% by funKEY are present, it returns an unchanged list ConditionalfunKEYList
% and an empty list of messages. The second equation for the function
% addCond is quite complex. It makes a distinction between the values of
% k1 and k2. If k1<k2 then (k1,m) should be added at the front of the
% list of messages. If k1 = k2 then the intruder has already some
% messages encrypted by k1. Therefore, m has to be added to the
% corresponding list of messages. If k1>k2 then, since the list is
% ordered, we proceed with adding (k,m) to its tail. A similar distinction
% is made for removeCond.

sort ConditionalfunKEYList
func emptyCfunKEYList:->ConditionalfunKEYList
     inCfunKEY:funKEY#MessageList#ConditionalfunKEYList->
            ConditionalfunKEYList
map  addCond:funKEY#Message#ConditionalfunKEYList->
        ConditionalfunKEYList
     removeCond:funKEY#ConditionalfunKEYList->funEncaps
     if:Bool#ConditionalfunKEYList#ConditionalfunKEYList->
        ConditionalfunKEYList
     getMessageList:Nat#ConditionalfunKEYList->MessageList
     length:ConditionalfunKEYList->Nat

var  k1,k2:funKEY
     m:Message
     L:MessageList
     CL,CL1:ConditionalfunKEYList
     n:Nat
rew  addCond(k1,m,emptyCfunKEYList)=
        inCfunKEY(k1,inMessage(m,emptyMessageList),
                emptyCfunKEYList)
     addCond(k1,m,inCfunKEY(k2,L,CL))=
        if(sm(k1,k2),
            inCfunKEY(k1,inMessage(m,emptyMessageList),inCfunKEY(k2,L,CL)),
          if(eq(k1,k2),
              inCfunKEY(k2,add(m,L),CL),
              inCfunKEY(k2,L,addCond(k1,m,CL))))
     removeCond(k1,emptyCfunKEYList)=
        encaps(emptyCfunKEYList,emptyMessageList)
     removeCond(k1,inCfunKEY(k2,L,CL))=
        if(sm(k1,k2),
            encaps(CL,emptyMessageList),
          if(eq(k1,k2),
              encaps(CL,L),
              encaps(inCfunKEY(k2,L,pr1(removeCond(k1,CL))),
                      pr2(removeCond(k1,CL)))))

      if(T,CL,CL1)=CL
      if(F,CL,CL1)=CL1

      length(emptyCfunKEYList) = 0
      length(inCfunKEY(k1,L,CL)) = S(length(CL))

      getMessageList(n,emptyCfunKEYList) = emptyMessageList
      getMessageList(0,inCfunKEY(k1,L,CL)) = L
      getMessageList(S(n),inCfunKEY(k1,L,CL)) = getMessageList(n,CL)

% The sort symEncaps is also auxiliary, similar to funEncaps.

sort symEncaps
func  encaps:ConditionalsymKEYList#MessageList->symEncaps
map  pr1: symEncaps->ConditionalsymKEYList
     pr2: symEncaps->MessageList
     if:Bool#symEncaps#symEncaps->symEncaps

var  fcl:ConditionalsymKEYList
     ml:MessageList
     se1,se2:symEncaps
rew  pr1(encaps(fcl,ml)) = fcl
     pr2(encaps(fcl,ml)) = ml
     if(T,se1,se2)=se1
     if(F,se1,se2)=se2

% The sort ConditionalsymKEYList contains a list of messages received
% en that have been encoded by symmetric keys. For
% explanations see ConditionalfunKEYList above, which has exactly
% the same structure.

sort  ConditionalsymKEYList
func  emptyCsymKEYList:->ConditionalsymKEYList
      inCsymKEY:symKEY#MessageList#ConditionalsymKEYList->
         ConditionalsymKEYList
map   addCond:symKEY#Message#ConditionalsymKEYList->
         ConditionalsymKEYList
      removeCond:symKEY#ConditionalsymKEYList->symEncaps
      if:Bool#ConditionalsymKEYList#ConditionalsymKEYList->
                ConditionalsymKEYList
      getMessageList:Nat#ConditionalsymKEYList->MessageList
      length:ConditionalsymKEYList->Nat

var   k1,k2:symKEY
```

```
      m: Message
      L:MessageList
      CL,CL1:ConditionalsymKEYList
      n:Nat
rew   addCond(k1,m,emptyCsymKEYList)=
        inCsymKEY(k1,inMessage(m,emptyMessageList),
                emptyCsymKEYList)

      addCond(k1,m,inCsymKEY(k2,L,CL))=
         if(sm(k1,k2),
             inCsymKEY(k1,inMessage(m,emptyMessageList),
                inCsymKEY(k2,L,CL)),
           if(eq(k1,k2),
               inCsymKEY(k2,add(m,L),CL),
               inCsymKEY(k2,L,addCond(k1,m,CL))))

      removeCond(k1,emptyCsymKEYList)=
         encaps(emptyCsymKEYList,emptyMessageList)
      removeCond(k1,inCsymKEY(k2,L,CL))=
         if(sm(k1,k2),
             encaps(CL,emptyMessageList),
           if(eq(k1,k2),
               encaps(CL,L),
               encaps(inCsymKEY(k2,L,pr1(removeCond(k1,CL))),
                      pr2(removeCond(k1,CL)))))

      if(T,CL,CL1)=CL
      if(F,CL,CL1)=CL1

      length(emptyCsymKEYList)=0
      length(inCsymKEY(k1,L,CL))=S(length(CL))

      getMessageList(n,emptyCsymKEYList)=emptyMessageList
      getMessageList(0,inCsymKEY(k1,L,CL))=L
      getMessageList(S(n),inCsymKEY(k1,L,CL))=getMessageList(n,CL)

% The sort Knowledge of the intruder. This sort implements the entire
% knowledge of the intruder, consisting of six different lists: two
% lists of messages (encoded by symmetric and functional keys), and
% lists of symmetric keys, functional keys, nonces and addresses. The
% most important function of this sort is updateKnowledge that given
% a (list of) message(s) and the current state of intruder's knowledge
% updates it and returns the updated state of knowledge. Updating knowledge
% is a recursive process since decrypting a message can result in obtaining
% new symmetric or functional keys that in their turn can be used to
% decrypt further messages. This propagation step is modeled by two
% auxiliary functions propagates and propagatef.

sort Knowledge
func wrap:ConditionalsymKEYList#ConditionalfunKEYList#
     symKEYList#funKEYList#NONCEList#ADDRList->Knowledge
map  fEncMessages: Knowledge->ConditionalfunKEYList
     sEncMessages: Knowledge->ConditionalsymKEYList
     fKEYs: Knowledge->funKEYList
     sKEYs: Knowledge->symKEYList
     addresses: Knowledge->ADDRList
     nonces: Knowledge->NONCEList
     if:Bool#Knowledge#Knowledge->Knowledge
     updateKnowledge:Message#Knowledge->Knowledge
     updateKnowledge:MessageList#Knowledge->Knowledge
     propagates:symKEY#Knowledge->Knowledge
     propagatef:funKEY#Knowledge->Knowledge
     initialKnowledge:->Knowledge

var  scl: ConditionalsymKEYList
     fcl:ConditionalfunKEYList
     sl: symKEYList
     fl: funKEYList
     nl:NONCEList
     al:ADDRList
     K1,K2:Knowledge
     f,f1:funKEY
     s:symKEY
     n:NONCE
     a:ADDR
     ml:MessageList
     m: Message
rew  sEncMessages(wrap(scl,fcl,sl,fl,nl,al)) = scl
     fEncMessages(wrap(scl,fcl,sl,fl,nl,al)) = fcl
     sKEYs(wrap(scl,fcl,sl,fl,nl,al)) = sl
     fKEYs(wrap(scl,fcl,sl,fl,nl,al)) = fl
     addresses(wrap(scl,fcl,sl,fl,nl,al)) = al
     nonces(wrap(scl,fcl,sl,fl,nl,al)) = nl
     if(T,K1,K2) = K1
     if(F,K1,K2) = K2

     updateKnowledge(encrypt(f,s),wrap(scl,fcl,sl,fl,nl,al)) =
        if(isIn(s,sl),
           propagatef(f,wrap(scl,fcl,sl,add(f,fl),nl,al)),
           wrap(addCond(s,encrypt(f,s),scl),fcl,sl,fl,nl,al))
     updateKnowledge(encrypt(n,a,f),wrap(scl,fcl,sl,fl,nl,al)) =
        if(isIn(decodeKEY(f),fl),
           wrap(scl,fcl,sl,fl,add(n,nl),add(a,al)),
           wrap(scl,addCond(f,encrypt(n,a,f),fcl),sl,fl,nl,al))
     updateKnowledge(encrypt(f1,n,s,f),wrap(scl,fcl,sl,fl,nl,al)) =
        if(isIn(decodeKEY(f),fl),
           propagates(s,
             propagatef(f1,
                wrap(scl,fcl,add(s,sl),add(f1,fl),add(n,nl),al))),
                wrap(scl,addCond(f,encrypt(f1,n,s,f),fcl),sl,fl,nl,al))

     updateKnowledge(emptyMessageList,K1) = K1
     updateKnowledge(inMessage(m,ml),K1) =
        updateKnowledge(ml,updateKnowledge(m,K1))

     propagates(s,wrap(scl,fcl,sl,fl,nl,al)) =
        updateKnowledge(pr2(removeCond(s,scl)),
                wrap(pr1(removeCond(s,scl)),fcl,sl,fl,nl,al))

     propagatef(f,wrap(scl,fcl,sl,fl,nl,al)) =
        updateKnowledge(pr2(removeCond(f,fcl)),
                wrap(scl,pr1(removeCond(f,fcl)),sl,fl,nl,al))

     initialKnowledge =
        wrap(emptyCsymKEYList,emptyCfunKEYList,
                initialsymKEYList, initialfunKEYList,
                initialNONCEList, initialADDRList)


% Here we are finished with the required datatypes. Below we
% define the processes. We start out with declaring the actions
% that the processes use to communicate. Using the comm declaration
% section, it is defined how these action communicate. Then we
% model three processes, the initiator, the responder and the
% eavesdropper (Iproc, Rproc and Eproc). Then we define different
% constellations of initiators, responder and eavesdroppers that
% we have analysed. The concrete selection is made using the
% keyword init at the end.

act  sE          :Message     % send of the eavesdropper.
     sE          :Agent#Nat   % send of the eavesdropper
                              % (the preliminary step).
     rE          :Message     % receive of the eavesdropper.
     sI          :Message     % send of the initiator.
     rI          :Message     % receive of the initiator.
     rI          :Agent#Nat   % receive of the initiator
                              % (the preliminary step).
     sR          :Message     % send of the responder.
     rR          :Message     % receive of the responder.
     rR          :Agent#Nat   % receive of the responder
                              % (the preliminary step).
     c1,c2,c3,c4:Message      % communication between sends
                              % and receives. See "comm" below.
     c2,c4       :Agent#Nat   % communication between sends and
                              % receives. See "comm" below
                              % (the preliminary step).
     secret      :symKEY      % secrecy claims of the initiator
                              % and the responder.
     secret_     :symKEY      % the receipt of a claim by the
                              % eavesdropper to check whether
                              % it knows the secret.
     secret__    :symKEY      % communcation of secrecy claims.
     NOT_SECRET               % secrecy violation indicator.

% Below it is defined on which channels the different
% processes synchronize to exchange information.

comm  sI|rE = c1              % from the initiator to the intruder
      sE|rI = c2              % from the intruder to the initiator
      sR|rE = c3              % from the responder to the intruder
      sE|rR = c4              % from the intruder to the responder
      secret|secret_ = secret__

% Below we model the initiator. The initiator starts by creating a
% fresh nonce. The first message sent by the initiator consists of
% the nonce and his address encrypted with the public key of the
% intended responder. Then, the initiator reads a message, verifies
% that it has the expected form (a functional key and a hash of a
% nonce encrypted by the initiator's public key), and acknowledges
% receipt of the message by sending hash of the shared nonce
% encrypted by the shared symmetric key. Finally, the initiator
% claims security of the shared symmetric key. For further details
% see the main text.

proc Iproc(self:Agent,n:Nat)=
       sum(a:Agent,rI(a,n).
         sI(encrypt(nonce(n),addr(self),PK(a))).
       sum(m:Message,rI(m).
         (sI(encrypt(h(get2N(m)),get3s(m))).
            (secret(get3s(m))<|not(eq(a,E))|>delta).delta)
            <|and(encrypted3(m,PK(self)),eq(get1f(m),h(nonce(n))))|> delta
         ))

% The responder is similarly to the initiator. The responder starts
% by receiving a message, verifies that it is in the expected form
% (contains the address of the presumed counterpart encrypted by the
% responder's public key), generates a new nonce and sends a message
% consisting of the hashed nonce of the counterpart, the newly
% created nonce and the symmetric key to be shared, encrypted by the
% public key of the counterpart. Next, the responder reads a new
% message, verifies that it has been encrypted by the shared symmetric
% key and contains the hash function of his nonce, and claims secrecy
% of the shared key. Additional information can be found in the main text.

proc Rproc(self:Agent,n:Nat)=
       sum(a:Agent,rR(a,n).
       sum(m1:Message,rR(m1).
         sR(encrypt(h(get1N(m1)),nonce(n),K(n),PK(a))).
             (sum(m2:Message,rR(m2).
                 secret(K(n)).delta
                     <| and(not(eq(a,E)),and(encrypted1(m2,K(n)),
                     eq(get1f(m2),h(nonce(n))))) |> delta
                     ))
                   <| and(eq(get2A(m1),addr(a)),
                    encrypted2(m1,PK(self))) |> delta
                 ))

% The process Eproc models the eavesdropper. The first summand (action rE)
% describes the receipt of a message. This message is immediately added to
% the knowledge that the eavesdropper already has. The next summands, except
% the last, describe the sending of an arbitrary message that can be
% constructed with the knowledge of the eavesdropper. In this way all
% possible attacks on the protocol are modelled. The last summand consists
% of receiving a supposed to be secret. If the eavesdropper knows it,
% it signals a NOT_SECRET action. Please refer to Section 4 of
% the main text for further details.

proc Eproc(K:Knowledge,p:Nat,aI:Agent,aR:Agent)=

       sum(m:Message,rE(m).Eproc(updateKnowledge(m,K),p,aI,aR)
         <| eq(0,p) |> delta) +

       sum(a:Agent,
         (sE(a,p).Eproc(K,P(p),aI,if(eq(p,2),E,a))
                 <| and(sm(0,p),not(sm(a,aR))) |> delta)
         <| even(p) |>
         (sE(a,p).Eproc(K,P(p),if(eq(p,1),E,a),aR)
                 <| and(and(sm(0,p),not(sm(a,aI))),
                         imply(and(eq(p,1),eq(aI,E)),
                         not(eq(a,E)))) |> delta))+

       sum(n1:Nat,sum(n2:Nat,
         sE(encrypt(h(get(n1,nonces(K))),
                 get(n2,sKEYs(K)))).Eproc(K,p,aI,aR)
            <|and(eq(0,p),
              and(sm(n1,length(nonces(K))),
                  sm(n2,length(sKEYs(K)))))
            |>delta)) +

       sum(n1:Nat,sum(n2:Nat,
         sE(encrypt(get(n1,fKEYs(K)),
                 get(n2,sKEYs(K)))).Eproc(K,p,aI,aR)
            <|and(eq(0,p),
              and(sm(n1,length(fKEYs(K))),
                  sm(n2,length(sKEYs(K)))))
            |>delta)) +

       sum(n1:Nat,sum(n2:Nat,sum(n3:Nat,
          sE(encrypt(get(n1,nonces(K)),get(n2,addresses(K)),
                 get(n3,fKEYs(K)))).Eproc(K,p,aI,aR)
            <|and(eq(0,p),
              and(sm(n1,length(nonces(K))),
              and(sm(n2,length(addresses(K))),
                  sm(n3,length(fKEYs(K))))))
            |>delta)))+

       sum(n1:Nat,sum(n2:Nat,sum(n3:Nat,sum(n4:Nat,
         sE(encrypt(h(get(n1,nonces(K))),get(n2,nonces(K)),
                 get(n3,sKEYs(K)),get(n4,fKEYs(K)))).
                 Eproc(K,p,aI,aR)
            <|and(eq(0,p),
              and(sm(n1,length(nonces(K))),
              and(sm(n2,length(nonces(K))),
              and(sm(n3,length(sKEYs(K))),
                  sm(n4,length(fKEYs(K)))))))
            |>delta))))+

       sum(n1:Nat,sum(n2:Nat,sum(n3:Nat,sum(n4:Nat,
         sE(encrypt(get(n1,fKEYs(K)),get(n2,nonces(K)),
                 get(n3,sKEYs(K)),get(n4,fKEYs(K)))).
                 Eproc(K,p,aI,aR)
             <|and(eq(0,p),
               and(sm(n1,length(fKEYs(K))),
               and(sm(n2,length(nonces(K))),
               and(sm(n3,length(sKEYs(K))),
                 sm(n4,length(fKEYs(K)))))))
           |>delta))))+

       sum(n1:Nat,sum(n2:Nat,
            sE(get(n2,getMessageList(n1,sEncMessages(K)))).
                 Eproc(K,p,aI,aR)
           <|and(eq(0,p),
             and(sm(n1,length(sEncMessages(K))),
                 sm(n2,length(getMessageList(n1,sEncMessages(K))))))
           |>delta))+

       sum(n1:Nat,sum(n2:Nat,
            sE(get(n2,getMessageList(n1,fEncMessages(K)))).
                 Eproc(K,p,aI,aR)
           <|and(eq(0,p),
             and(sm(n1,length(fEncMessages(K))),
                 sm(n2,length(getMessageList(n1,fEncMessages(K))))))
           |>delta))+

       sum(k:symKEY, secret_(k).
          (NOT_SECRET.Eproc(K,p,aI,aR)
         <| isIn(k,sKEYs(K)) |>
            Eproc(K,p,aI,aR)
          ))

% System1..System9 are used to test the protocol for 1..9 processes.
% To generate the state space: replace 'init System5' below by 'init
% SystemN', where N is the desired number of processes.

proc System1=
      encap({rE,sI,rI,sE,sR,rR,secret,secret_},
             Iproc(A,1)||
             Eproc(initialKnowledge,1,E,E))

proc System2=
      encap({rE,sI,rI,sE,sR,rR,secret,secret_},
             Iproc(A,1)||Rproc(B,2)||
             Eproc(initialKnowledge,2,E,E))


proc System3=
      encap({rE,sI,rI,sE,sR,rR,secret,secret_},
             Iproc(A,1)||Rproc(B,2)||
             Iproc(A,3)||
             Eproc(initialKnowledge,3,E,E))

proc System4=
      encap({rE,sI,rI,sE,sR,rR,secret,secret_},
             Iproc(A,1)||Rproc(B,2)||
             Iproc(A,3)||Rproc(B,4)||
             Eproc(initialKnowledge,4,E,E))

proc System5=
      encap({rE,sI,rI,sE,sR,rR,secret,secret_},
             Iproc(A,1)||Rproc(B,2)||
             Iproc(A,3)||Rproc(B,4)||
             Iproc(A,5)||
             Eproc(initialKnowledge,5,E,E))

proc System6=
     encap({rE,sI,rI,sE,sR,rR,secret,secret_},
             Iproc(A,1)||Rproc(B,2)||
             Iproc(A,3)||Rproc(B,4)||
             Iproc(A,5)||Rproc(B,6)||
             Eproc(initialKnowledge,6,E,E))

proc System7=
     encap({rE,sI,rI,sE,sR,rR,secret,secret_},
             Iproc(A,1)||Rproc(B,2)||
             Iproc(A,3)||Rproc(B,4)||
             Iproc(A,5)||Rproc(B,6)||
             Iproc(A,7)||
             Eproc(initialKnowledge,7,E,E))

proc System8=
     encap({rE,sI,rI,sE,sR,rR,secret,secret_},
             Iproc(A,1)||Rproc(B,2)||
             Iproc(A,3)||Rproc(B,4)||
             Iproc(A,5)||Rproc(B,6)||
             Iproc(A,7)||Rproc(B,8)||
             Eproc(initialKnowledge,8,E,E))

proc System9=
     encap({rE,sI,rI,sE,sR,rR,secret,secret_},
             Iproc(A,1)||Rproc(B,2)||
             Iproc(A,3)||Rproc(B,4)||
             Iproc(A,5)||Rproc(B,6)||
             Iproc(A,7)||Rproc(B,8)||
             Iproc(A,9)||
             Eproc(initialKnowledge,9,E,E))


init System5
```

B Updatefile

The file `updatefile` contains renamings of actions. E.g. in the description below the action `c1(X)` is renamed to `ctau` where `X` is any parameter. It is possible to formulate conditions on these parameters to perform conditional renaming. This is not used below. The tool `tbfupdate` is needed to carry out the actual renaming on a linear process.

```
c1(X) -> ctau
c2(X) -> tau
c3(X) -> ctau
c4(X) -> tau
c2(X,Y) -> tau
c4(X,Y) -> tau
secret__(X) -> tau
```