Journal of Universal Computer Science, vol. 3, no. 12 (1997), 1337-1381
submitted: 20/11/96, accepted: 22/6/97, appeared: 28/12/97

Formal Analysis of the Kerberos Authentication System

Giampaolo Bella
(Computer Laboratory, University of Cambridge
New Museums Site - Pembroke Street - CB2 3QG Cambridge (UK)
Giampaolo.Bella@cl.cam.ac.uk)

Elvinia Riccobene
(Dipartimento di Matematica, Università di Catania
Viale A.Doria, 6 - I-95125 Catania (ITALY)
riccobene@dipmat.unict.it)

Abstract: The Gurevich's Abstract State Machine formalism is used to specify the well known *Kerberos Authentication System* based on the Needham-Schroeder authentication protocol. A complete model of the system is reached through stepwise refinements of *ASM*s, and is used as a basis both to discover the minimum assumptions to guarantee the *correctness* of the system and to analyse its *security* weaknesses. Each refined model comes together with a correctness refinement theorem.

Key Words: Formal Methods, Security, Protocol specification, Refinement, Protocol verification, Key distribution protocol, Gurevich's Abstract State Machine, Kerberos.
Category: C.2.2, D

1 Introduction

When computers were stand-alone entities, security was provided by physical means: computer rooms were sealed and locked, and punched cards went in one window with line printer listings out another.

But since the early 1970s, *networking* brings up the necessity to communicate securely over an insecure network of computers, where multiple users can gain access to the network services. Therefore, the ability to accurately identify each user making a request becomes essential.

In general terms, the problem is how the provider of a service (a "server" in brief) can determine whether a client's request for the service is to be honoured or not. A good solution is the server's capability to verify the user's identity. This process is called *authentication* and goes through some steps which constitute what is called an *authentication protocol*.

Although protocols are generally made up of a few messages sent on the network, they can hide terribly subtle errors. It is not unusual for errors to be discovered on protocols that have been used for years (e.g. see [Lowe 96b]). Protocol errors "... are unlikely to be detected in normal operations. The need for techniques to verify the correctness of such protocols is great ..." [Needham, Schroeder 78].

A protocol should be *correct*, i.e. it should allow authorised users from gaining the services they require, and hopefully *secure*, i.e. it should prevent any unauthorised user to get access to any service. Unfortunately, unlike the former notion, *security* "... is not a simple Boolean predicate" [Anderson 95]. Today, after thirty years of networking, researchers have proved many properties of different protocols, but claims like '*this protocol is secure*' are still difficult to make.

Although errors might arise at the *implementation level*, when writing the compilable code for a protocol, the flaws which are more difficult to detect seem to be hidden at the *design level*. To tackle them, the use of *formal methods* has recently had good results ([Bellare, Rogaway 95], [Burrows et al. 90], [Lowe 96a], [Bolignano 96], [Paulson 96], etc.). This paper shows how the *Gurevich's Abstract State Machine* (ASM in brief) formalism can model a complex protocol like *Kerberos*, and allow us to reason about its properties.

The Kerberos Authentication System is based on the Needham and Schroeder authentication protocol, which seems to be secure under the naive assumption that no session keys may ever be compromised [Needham, Schroeder 78]. The accidental loss of a session key is supposed to be overcome by the presence of timestamps [Denning, Sacco 81]. Nevertheless, Kerberos still seems to suffer from some limitations [Bellovin, Merritt 90], which were partially corrected in the last version, Version 5, and are highlighted here.

In this paper an abstract model of the complete authentication system is reached through stepwise refinement of ASMs, and is used as a basis to discover the minimum assumptions to guarantee the *correctness* of the system. The *security* of the system is also taken into account by showing how an eavesdropper might exploit a stolen session key to obtain a resource, despite the addition of timestamps.

The stepwise refinement strategy makes it easier to understand a complex system as Kerberos, because the details are presented step by step. Our formalism proves to be suitable to this purpose, due to its versatility. In fact, the abstraction mechanism built into the notion of ASM allows the systematic use of strong information hiding and modularisation techniques. Stepwise refinements and extensions are straightforwardly achieved by function and module refinements.

To our knowledge, this is the first attempt to get a formal specification of the whole Kerberos architecture, and we believe that our work might serve as a basis for both implementation and further tool-assisted analyses. As a matter of fact, the work of [Bella, Paulson 97] is based on our specification.

The specification of the whole architecture comes together with the proofs of correctness under reasonable regularity conditions, and the description of a real hostile environment. These features make our specification more reliable than the theoretical work of [Burrows et al. 90] and of [Schumann 97], and more realistic than the work of [Mitchell et al. 97], which is based on a very idealised version of Kerberos without timestamps, and suffer from the intrinsic limits of state enumeration.

The paper is organised as follows. Basic ideas of the operation of Kerberos are introduced in [Section 2]; [Section 3] briefly introduces the reader to the *Gurevich's Abstract State Machine* and presents the specification reached through a sequence of four refinement steps, starting with a model which directly reflects the basic idea of the Kerberos operation, and moving down to a model describing the complete system. Liveness conditions and proofs of correctness are presented in [Section 3.4]; [Section 4] concerns the external threats, and [Section 5] concludes the paper.

2 Kerberos

Kerberos [Miller et al. 89] was developed in the mid eighties as part of project Athena at MIT. It is a distributed authentication system that allows a *client* – a process running on behalf of a user – to prove its identity to a *server* without sending data across the network that might allow an attacker to subsequently impersonate that principal.

Rather than building in elaborate, error-prone authentication protocols at each server, Kerberos provides a centralised authentication server whose function is to authenticate clients to servers and servers to clients, and provide both client and server with a secret key that they may use to encrypt/decrypt their messages.

Based on the [Needham, Schroeder 78] key distribution protocol modified with the addition of timestamps [Voydock, Kent 83], Kerberos relies exclusively on *private-key* (also called conventional) encryption, rather than *public-key* encryption.

In this paper we refer to the original Kerberos, Version 4 [Miller et al. 89], which is the most widely used.

2.1 How Kerberos Works

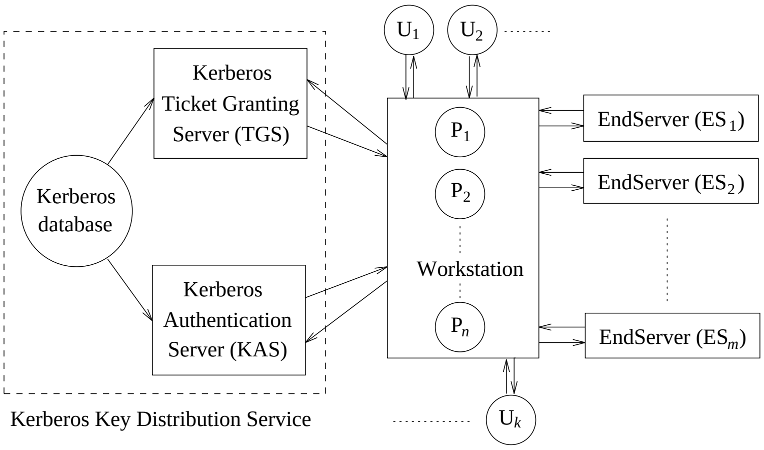

Figure 1: Kerberos Authentication Architecture

In [Fig. 1], we can see the complete layout of the Kerberos architecture. It consists of the following parties: the *Workstation* with a set of processes $P_1, \ldots, P_n$ running on it on behalf of a set of users $U_1, \ldots, U_k$, the *Kerberos Key Distribution Service* (or *Kerberos* in brief) and a set of end servers $ES_1, \ldots, ES_m$. The Kerberos Key Distribution Service is in turn composed of two servers, the *Kerberos Authentication Server – KAS –* and the *Ticket Granting Server – TGS.*

Kerberos keeps a database of the clients' and servers' secret keys. A client's key is the DES-coded version of the password of the user who owns the client.

Names (identifiers) are the only entities exchanged in cleartext on the network, everything else is encrypted.

The basic idea of the Kerberos authentication scheme is the following: to use a service, a client must supply the end server which provides that service with a *ticket*, previously obtained from Kerberos. A ticket for a service is a string of bits which has been encrypted using the providing server's private key. That private key is known only to the server itself and to Kerberos. As a result, the server can be confident that any information found inside the ticket originated from Kerberos. Since the identity of the client has been declared inside the ticket, the server that receives a ticket has a Kerberos-authenticated evidence of the client's identity.

A distinct ticket is needed for each service, but not all the services that a client might wish to use are known at the beginning of a login session. Therefore, to overcome the need to save on the workstation the tickets for all the services – some of which might even never be required – the client obtains a single ticket from KAS at login time, called *authentication ticket*, to be used only with TGS. Whenever the client wishes to use a network service, it asks TGS for a new, specific ticket, called *service ticket*, to be used with the end server that provides the service.

To help ensure that an intruder does not steal and reuse another user's ticket, the client accompanies the ticket with an *authenticator*, as will be elaborated later. Tickets and authenticators are the two types of the client's credentials.

Together with the ticket, a client also receives from Kerberos a session key to share with a server during a communication session. More precisely, a client receives the *authentication key* from KAS to communicate with TGS, and the *service key* from TGS to communicate with an ES.

Therefore, broadly speaking, Kerberos comprises two orthogonal phases: the *authentication phase*, during which KAS provides the client with the authentication ticket and the authentication key; and the *authorisation phase*, during which TGS provides the client – already authenticated by KAS – with a service ticket and a service key to share with an ES.

This short introduction simply aims to give an insight into the complex mechanisms of the protocol. A complete explanation will be reached through a sequence of steps in the following section.

3 The Specification

In this section a complete, formal model of the whole authentication system is reached through a hierarchy of more and more detailed models. Each model is a distributed Gurevich's Abstract State Machine. The basic definitions of this framework can be found in [Section 3.1], while for the mathematical foundation we refer to [Gurevich 95, Börger 95b].

The first model, *MessagePassing*, describes the message exchange layout between agents and considers only one client process. Elements like tickets and authenticators are considered as atomic objects provided by monitored functions. It gives the guidelines of the protocol.

Tickets, authenticators and all the abstract functions providing them are refined in the second model, *EncryptionDecryption*, by explicit formalisation of the encryption-decryption procedures at the basis of the mechanism to hide information. This model contains the bulk of the operational details of the protocol.

The third model, *MultipleClients*, allows several client processes to run on the workstation, and finally, by the fourth model, *MultipleEndServers*, the description of the complete system is reached through the consideration of several network end servers.

Each refined model comes together with a refinement theorem.

3.1 Gurevich's Abstract State Machines

We assume that the reader is familiar with the semantics of the Abstract State Machine defined in [Gurevich 95], and we quote here only the essential definitions.

A *Gurevich's Abstract State Machine* $\mathcal{A}$ is defined by a program *Prog* – consisting of a finite number of *transition rules* – and a (class of) *initial state*(s) S_0. $\mathcal{A}$ models the operational behaviour of a real dynamic system $\mathcal{S}$ in terms of state transitions.

A *state* S is a first-order structure over a fixed *signature* which is also regarded as the signature of $\mathcal{A}$, representing the instantaneous configuration of $\mathcal{S}$. The value of a term t at S is denoted by $[t]_S$.

The basic transition rule is the following *function update*

$$f(t_1, \ldots, t_n) := t$$

where f is an arbitrary n-ary function and $t_1, \ldots, t_n, t$ are first-order terms. To fire this rule to a state S evaluate all terms $t_1, \ldots, t_n, t$ at S and update the function f to t on parameters $t_1, \ldots, t_n$. This produces another state S' which differs from S only in the new interpretation of the function f.
Note that no rule can change the signature of $\mathcal{A}$.

There are some rule constructors.

- The *conditional constructor* which produces "guarded" transition rules of the form:

  $$\texttt{if } g \texttt{ then } R_1 \texttt{ else } R_2$$

  where g is a ground term (the *guard* of the rule) and R_1, R_2 are transition rules. To fire that new rule to a state S evaluate the guard; if it is true, then execute R_1, otherwise execute R_2. The `else` part may be omitted.
- The *block constructor* which produces transition rules of the form:

  $$\begin{array}{l}\texttt{block}\\ \quad R_1\\ \quad\ \ \vdots\\ \quad R_n\\ \texttt{endblock}\end{array}$$

  to apply R_1, …, R_n simultaneously.
- The *parallel constructor* which produces the "parallel" synchronous rule of form:

  $$\begin{array}{l}\texttt{var } x \texttt{ ranges over } U\\ \quad R(x)\end{array}$$

where $R(x)$ is a generalised basic transition rule with a variable x ranging over the universe U. To execute the new rule, execute $R(x)$ for every $x \in U$.

We also make use of the following construct to let universes grow:

$$\texttt{extend}\ U\ \texttt{by}\ x_1, \ldots, x_n\ \texttt{with}\ \mathit{Updates}\ \texttt{endextend}$$

where *Updates* may (and should) depend on the x_i and are used to define certain functions for (some of) the new objects x_i of the resulting universe U.

State transitions of $\mathcal{A}$ may be influenced in two ways: through the rules of the program *Prog*, or through the modifications of the environment. A function in the signature of $\mathcal{A}$ is called *controlled* if it is updated by (and only by) a transition rule in *Prog*, otherwise, if its values come by modifications of the environment, it is called *monitored* (see [Börger, Mearelli 97]). The symbol S^- denotes the reduct of the state S to the controlled functions.

A computation of $\mathcal{S}$ is modelled by a finite or infinite *run* ϱ of $\mathcal{A}$ as a sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of $\mathcal{A}$, where S_0 is an initial state and each S_{n+1}^- is obtained from S_n by firing simultaneously all transition rules of *Prog* to S_n.

In a *distributed Gurevich's Abstract State Machine* $\mathcal{A}$, multiple autonomous agents cooperatively model a concurrent computation of a system $\mathcal{S}$. Each agent a executes its own *single-agent program* $\mathit{Prog}(a)$ as specified by the *module* associated with a by the function *Mod*. More precisely, an agent a has a partial view $\mathit{View}(a, S)$ of a given global state S as defined by its sub-vocabulary $\mathit{Fun}(a)$ (i.e. the function names occurring in $\mathit{Prog}(a)$) and it can *make a move* at S by firing $\mathit{Prog}(a)$ at $\mathit{View}(a, S)$ and changing S accordingly. The underlying semantic model ensures that the order in which the agents of $\mathcal{A}$ perform their operations is always such that no conflicts between the update sets computed for distinct agents can arise. The global program *Prog* is the union of all single-agent programs.

Self does not belong to $\mathit{Fun}(a)$ for any agent a and is a special nullary function name which is differently interpreted by different agents (an agent a interprets *Self* as a) and allows an agent to identify itself among other agents. It cannot be the subject of an update instruction and is used to parameterise the agent's specific functions.

A *sequential run* of a distributed Gurevich's Abstract State machine $\mathcal{A}$ is a (finite or infinite) sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of $\mathcal{A}$, where S_0 is an initial state and every S_{n+1}^- is obtained from S_n by executing a move of *an* agent.

The *partially ordered run*, defined in [Gurevich 95], is the most general definition of run for a distributed ASM. In order to prove properties on a partially ordered run, the attention may be restricted to a *linearisation* of it, which is, in turn, a sequential run (see [Gurevich 95] for more explanations).

In the context of a distributed ASM, the functions are better classified from a specific agent's point of view. For an agent a, a function can be *controlled* if updated by (and only by) a rule in $\mathit{Prog}(a)$, and *monitored* if updated non deterministically by (and only by) its external environment (i.e. by some other agent than a or by the environment of the whole distributed system). A function is called *interaction* if updated by rules of $\mathit{Prog}(a)$ and by the environment; such functions do not occur in our specification of Kerberos.

On a generic term t we use the abbreviations $\mathit{defined}(t)$ for $t \neq \mathit{undef}$, $\mathit{undefined}(t)$ for $t = \mathit{undef}$, $\mathit{clear}(t)$ for $t := \mathit{undef}$. Incidentally, recall that $f(t) = \mathit{undef}$ if $t = \mathit{undef}$.

3.2 The *MessagePassing* Model

We start with a simplified version of the system, which is depicted in [Fig.2]. The processes running on the workstation are formalised by a client agent C.

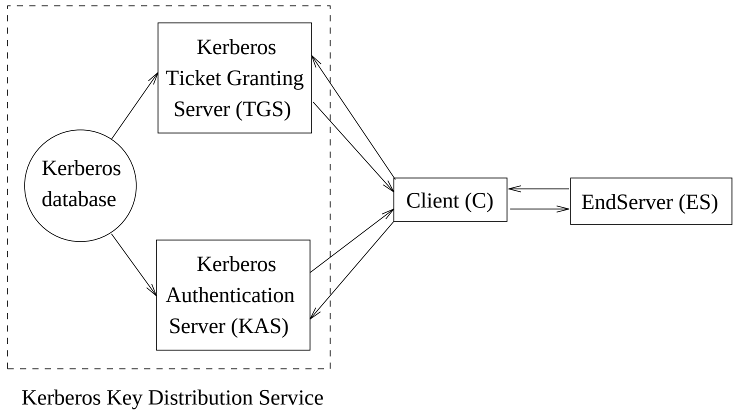

Figure 2: Basic Kerberos Authentication Scheme

The aim of this model is formalising the layout of message exchange between agents. Tickets and authenticators are considered as atomic objects without any insight of how they are built or of their components.

This model shows how the two phases of Kerberos are instances of the Needham and Schroeder private-key protocol modified with the addition of timestamps: the authentication phase involves KAS as a trusted third party and C and TGS as principals, skips the final handshake between the principals, and uses identifiers and timestamps as nonces; the authorisation phase regards TGS as a trusted third party and C and ES as principals, and makes use of authenticators and timestamps as nonces.

In the sequel we use small italics letters for a function *name*, except when we use a *CompoundName*. We use the capital initial when a function represents an agent's private information. We use the slanted font for the names of *macros*.

3.2.1 Global Signature

The most abstract model is a distributed ASM made up of four modules, one for each of the agents sitting on the network: the client C, the Kerberos Authentication Server KAS, the Ticket Granting Server TGS, the end server ES. We then define the universe $AGENT = \{C,\ KAS,\ TGS,\ ES\}$.

Our agents send messages over the network. This leads to the definition of the universe $MESSAGE$ and of the functions

$$sender,\ receiver : MESSAGE \longrightarrow AGENT$$

$MESSAGE$ contains messages of two types: compound messages (concatenation of data) and messages encrypted by a key. We denote the former by $\{X, X'\}$, the latter as $\{X, X'\}_{key}$. To know the message type we use the function

$$type : MESSAGE \longrightarrow \{cleartext,\ encrypted\}$$

Our rules use guards of the form $X\_ReceiveFrom\_Y : mssg$ as abbreviation for '$sender(mssg) = Y$ & $receiver(mssg) = X$ & $type(mssg) = cleartext$', while $X\_ReceiveFrom\_Y : CryptedMssg$ abbreviates '$sender(CryptedMssg) = Y$ & $receiver(CryptedMssg) = X$ & $type(CryptedMssg) = encrypted$'. The following macro:

X_SendTo_Y: $mssg$
  $\equiv$ `extend` $MESSAGE$ `by` $mssg$ `with`
    $sender(mssg) := X$
    $receiver(mssg) := Y$
    $type(mssg) := cleartext$
  `endextend`

allows to perform the communication between two agents. It creates a new cleartext message, given its sender and receiver, by extending the universe $MESSAGE$. A similar macro builds an encrypted message, replacing $mssg$ with $CryptedMssg$ and $cleartext$ with $encrypted$.
We often use $clear(mssg)$ and $clear(CryptedMssg)$ to empty a communication channel from the message.

Each agent operates strictly sequentially, so that each module is a sequential ASM which executes its rules as soon as they become applicable (i.e. when the guards of the rules have true interpretation in the current state). The sequential operation is formalised using the function

$$mode : AGENT \longrightarrow \{\mathsf{ReadyToSend},\ \mathsf{ReadyToReceive},\ \mathsf{ReadyToStart}\}$$

In ReadyToSend mode, an agent is going to send a message, while in ReadyToReceive mode, an agent is able to receive a message and eventually store the information it contains. Only after the possible handshake and the mutual authentication with ES, C is ReadyToStart the communication with it. We often write $mode$ for $mode(Self)$ when the agent $Self$ is uniquely determined from the context.

The agents must be synchronized with each other, so we have a unique clock for all of them and the nullary function name CT giving the *current time* of the clock. If $REAL$ is the set of the real numbers, CT is a real-valued function of real time. As an integrity constraint on CT, we require that the value of CT increases monotonically to the limit ∞.

Keys, tickets and authenticators are atomic objects belonging to the universes KEY, $TICKET$ and $AUTH$ respectively. The unary function

$$K : AGENT \longrightarrow KEY$$

yields the agents' private key known only to the agent itself and saved in the Kerberos database, to which only KAS and TGS have access. The monitored predicate $expired(Key)$, Key: KEY, is *true* if Key has expired.

Keys, tickets and authenticators are sent inside messages. To extract them we use the functions

$$ExtractKey : MESSAGE \longrightarrow KEY$$
$$ExtractTicket : MESSAGE \longrightarrow TICKET$$
$$ExtractAuth : MESSAGE \longrightarrow AUTH$$

3.2.2 Local Signature

Each agent has to save session keys and sealed tickets on its own private locations, so we define

$$K(C,S,Self)\colon KEY \text{ and } Ticket(C,S,Self)\colon TICKET$$

respectively, being $S \in \{TGS, ES\}$. We omit for brevity the argument *Self* in the module of *Self* – recall that each agent a interprets *Self* as a –, so we write $K(C, S)$ for the session key shared by C and S, and $Ticket(C, S)$ for the credential used by C to authenticate ifself to S.

We call a client *authenticated* if it has got an authentication ticket and an authentication key, and we define in the signature $Fun(C)$ of C

$$authenticated \equiv defined(Ticket(C,TGS,C)) \ \& \ defined(K(C,TGS,C))$$

Similarly, we call a client *authorised* if it has got a service ticket and a service key, so that in $Fun(C)$

$$authorised \equiv defined(Ticket(C,ES,C)) \ \& \ defined(K(C,ES,C))$$

$Auth(C,Key)\colon AUTH$ yields the client's authenticator sealed with key *Key*. The client builds it every time it wishes to contact a server by the function

$$BuildAuth : \{C\} \times KEY \longrightarrow AUTH$$

From the other side, *ES* uses the function

$$ExtractTs : AUTH \longrightarrow REAL$$

to extract a timestamp from a given authenticator.

To build keys and tickets we introduce for the agents *KAS* and *TGS* the functions

$$ProvideKey : \{C\} \times \{TGS, ES\} \longrightarrow KEY$$
$$ProvideTicket : \{C\} \times \{TGS, ES\} \longrightarrow TICKET$$

To improve security, Kerberos provides the servers with a procedure to validate the client and the client with a mechanism to check for the authenticity of an expected message. In the modules of *TGS* and *ES*, this validity procedure is abstractly formalised by the predicate

$$SuccValid(C,\ Ticket,\ Auth)$$

which is *true* if C is successfully validated by way of its credentials *Ticket*: *TICKET* and *Auth*: *AUTH*. The client follows a different procedure. Whenever it sends a message, it keeps some useful information and uses it later to verify the authenticity of the reply. Therefore, we introduce the function

$$mark : REAL \cup AGENT \times REAL$$

to keep such information and the predicate

$$CheckValid(CryptedMssg,\ mark)$$

which is *true* if C believes that *CryptedMssg*: *MESSAGE* is *recent* according to the data saved in *mark*.

When receiving a(n encrypted) message from *KAS*, not only must C check for the validity of that message, but it must also control whether it is able to open the message by its password coded using the standard Unix one-way encryption algorithm (which should equal the client's private key saved in the Kerberos database). This prevents a bogus user from stealing data. Such a test on the password is made by the predicate

$$CheckPassword(CryptedMssg,\ CodePsw(C))$$

being *CryptedMssg*: *MESSAGE* and $CodePsw(C)$: *KEY* the function that yields the coded client's password.

We formalise the existing network services by means of the universe *SERVICE*, which is supposed at this level to contain as elements the services that can be provided by *ES*. The monitored function *RequiredService*(C): *SERVICE* represents the client's request for a service.

For some network services, C is required to check the authenticity of the reply from *ES* – the predicate *RequiredMutualAuthentication*(*RequiredService*(C)) would be true. C performs this test by the predicate

MutualAuthentication(*CryptedMssg, mark*)

which is *true* if C successfully authenticates *ES*. This is still abstract at this level.

3.2.3 Module Specifications

The full program is listed in [Fig.3]. Each of the four agents has its own program and in any state the function *Mod* is defined as *Mod*(A) = A_MODULE, $A \in$ *AGENT*. For convenience's sake, in every rule of the A_MODULE we use the agent name A, which must be read as *Self*.

```
Authentication Phase                 Authorisation Phase

[C_MODULE_AUTHENTICATION]            [C_MODULE_AUTHORISATION]

   RequireAuthentication                RequireAuthorisation
   GetAuthentication                    NegotiateService
   RevokeAuthentication                 ≡ block
                                            GetAuthorisation
                                            ShowAuthorisation
[KAS_MODULE]                              endblock
                                        MutualAuthentication
   KasReply                             RevokeAuthorisation
   ≡ block
       CheckIdentity                 [TGS_MODULE]
       ProvideAuthentication
     endblock                           TgsReply
   RevokeAuthentication                 ≡ block
                                            CheckAuthentication
                                            ProvideAuthorisation
                                          endblock
                                        RevokeAuthorisation

                                     [ES_MODULE]

                                        Handshake
                                        ≡ block
                                            CheckAuthorisation
                                            ProvideHandshake
                                          endblock
```

Figure 3: The Program

The C_MODULE is divided into two subprograms C_MODULE_AUTHENTICATION and C_MODULE_AUTHORISATION to emphasise that C is an active party during both phases.

We now examine each rule in detail. Variables *mssg* and *CryptedMssg* range over *MESSAGE*.

Authentication Phase

To become authenticated, C sends a message containing its identifier, the TGS identifier and the current time to KAS. It also stores information that will be later used to check the validity of the KAS reply.

RequireAuthentication rule of C_MODULE_AUTHENTICATION

```
if ¬authenticated & mode = ReadyToSend
then
    C_SendTo_KAS : {C, TGS, CT}
    mark := ⟨TGS,CT⟩
    mode := ReadyToReceive
```

The following **KasReply** rule describes the KAS operation when it is invocated for authentication. As soon as KAS receives a message from C, it checks whether C is legitimate by looking up if $K(C)$ is defined in the Kerberos database (**CheckIdentity** rule). If C is legitimate, KAS generates a random session key to be used by C as an authentication key, and an authentication ticket that C will use as a credential with TGS. Ticket and key, along with the TGS identifier and the current time, are then sent to C inside a message encrypted by the client's key (**ProvideAuthentication** rule).

KasReply rule of KAS_MODULE

```
block

    [CheckIdentity rule]
    if mode = ReadyToReceive & KAS_ReceiveFrom_C : mssg & defined(K(C))
    then
        clear(mssg)
        mode := ReadyToSend

    [ProvideAuthentication rule]
    if mode = ReadyToSend
    then
        if defined(K(C,TGS)) & defined(Ticket(C,TGS))
        then
            KAS_SendTo_C : {K(C,TGS), Ticket(C,TGS), TGS, CT}_K(C)
            mode := ReadyToReceive
        else
            K(C,TGS) := ProvideKey(C,TGS)
            Ticket(C,TGS) := ProvideTicket(C,TGS)

endblock
```

Hence, C receives the KAS reply. If the test on the password succeeds and the message is considered recent, C extracts from the message the authentication key and the authentication ticket, and saves its own copies.

GetAuthentication rule of C_MODULE_AUTHENTICATION

$$
\begin{array}{l}
\texttt{if}\ \neg authenticated\ \&\ mode = \mathsf{ReadyToReceive}\\
\quad \&\ C\_ReceiveFrom\_KAS : CryptedMssg\\
\quad \&\ CheckPassword\,(CryptedMssg,\ CodePsw\,(C\,))\\
\quad \&\ CheckValid\,(CryptedMssg,\ mark\,)\\
\texttt{then}\\
\quad K(C,TGS) := ExtractKey\,(CryptedMssg)\\
\quad Ticket(C,TGS) := ExtractTicket\,(CryptedMssg)\\
\quad clear\,(CryptedMssg)\\
\quad mode := \mathsf{ReadyToSend}
\end{array}
$$

This ends the Authentication Phase.

Authorisation Phase

When a client is *authenticated*, it may wish to access some network services. Such a situation is formalised by *defined*(*RequiredService*(C)), abbreviated as *RequiredService*. This starts the authorisation phase.

C must initially get a service key and a service ticket from TGS to contact *ES*, the server able to perform the required service. C builds an authenticator sealed with its authentication key, and sends it to TGS along with its authentication ticket and the *ES* identifier. It also stores the *ES* identifier and the current time to verify later the TGS reply. The authenticator is erased because it is to be used only once.

RequireAuthorisation rule of C_MODULE_AUTHORISATION

$$
\begin{array}{l}
\texttt{if}\ authenticated\ \&\ RequiredService\ \&\ \neg authorised\ \&\ mode = \mathsf{ReadyToSend}\\
\texttt{then}\\
\quad \texttt{if}\ defined\,(Auth\,(C,\ K\,(C,TGS)))\\
\quad \texttt{then}\\
\qquad C\_SendTo\_TGS : \{Auth\,(C,\ K\,(C,TGS)),\ Ticket\,(C,TGS),\ ES\}\\
\qquad mark := \langle ES,\ CT\,\rangle\\
\qquad clear\,(Auth\,(C,\ K\,(C,TGS)))\\
\qquad mode := \mathsf{ReadyToReceive}\\
\quad \texttt{else}\\
\qquad Auth\,(C,\ K\,(C,TGS)) := BuildAuth\,(C,\ K\,(C,TGS))
\end{array}
$$

Rule **TgsReply** concerns how TGS deals with the client's request for a service. If C is validated by the ticket and the authenticator received inside the message, TGS saves its own copy of the authentication key (**CheckAuthentication** rule), and begins the creation of a reply. It builds an authorisation key and an authorisation ticket to be sent to C along with the *ES* identifier and the current time. The message is sealed with the authentication key. If such authorisation credentials have been already defined – i.e. C has already required the service in the past – and have still not expired, TGS does not define new ones. The authentication key is always deleted from memory because TGS gets it every time from the received message (**ProvideAuthorisation** rule).

TgsReply rule of TGS_MODULE

```
block

    [CheckAuthentication rule]
    if mode = ReadyToReceive & TGS_ReceiveFrom_C : mssg
       & SuccValid (C, ExtractTicket (mssg), ExtractAuth (mssg))
    then
       K(C,TGS) := ExtractKey (mssg)
       clear (mssg)
       mode := ReadyToSend

    [ProvideAuthorisation rule]
    if mode = ReadyToSend
    then
       if defined (K (C,ES)) & defined (Ticket (C,ES))
       then
          TGS_SendTo_C : {K (C,ES), Ticket (C,ES), ES, CT}_K(C,TGS)
          clear (K(C,TGS))
          mode := ReadyToReceive
       else
          K(C,ES) := ProvideKey(C,ES)
          Ticket(C,ES) := ProvideTicket(C,ES)

endblock
```

C verifies the authenticity of the TGS reply and saves its own copies of the authorisation credentials. Incidentally, the authorisation ticket remains indecipherable to C because it is sealed with the ES private key (**GetAuthorisation** rule). The authorisation ticket along with a new authenticator sealed with the authorisation key is what C then sends ES. If the required service requires the mutual authentication, then C stores information, by *mark*, to be used later for the test and gets mode ReadyToReceive the handshake signal from ES, otherwise it takes mode ReadyToStart the communication with ES (**ShowAuthorisation** rule).

NegotiateService rule of C_MODULE_AUTHORISATION

```
block

    [GetAuthorisation rule]
    if authenticated & RequiredService & ¬authorised
       & mode = ReadyToReceive & C_ReceiveFrom_TGS : CryptedMssg
       & CheckValid (CryptedMssg, mark)
    then
       K(C,ES) := ExtractKey (CryptedMssg)
       Ticket(C,ES) := ExtractTicket (CryptedMssg)
       clear (CryptedMssg)
       mode := ReadyToSend

    [ShowAuthorisation rule]
    if authenticated & RequiredService & authorised
       & mode = ReadyToSend
    then
       if defined (Auth (C, K (C,ES)))
       then
          C_SendTo_ES : {Auth (C, K (C,ES)), Ticket (C,ES)}
          clear (Auth (C, K (C,ES)))
          if RequiredMutualAuthentication (RequiredService (C))
```

```
            then
                mark := ⟨CT⟩
                mode := ReadyToReceive
            else
                mode := ReadyToStart
        else
            Auth(C, K(C,ES)) := BuildAuth(C, K(C,ES))

endblock
```

Rule **Handshake** explains how *ES* manages the service requests. If the client's credentials check out, it saves its own copy of the authorisation key. From now on, *C* and *ES* share the same session key (**CheckAuthorisation** rule), as will be proved later. If the client's service request requires the mutual authentication, then *ES* replies the client with the *handshake function*, i.e. the timestamp of the authenticator incremented by one, encrypted by the authorisation key (**ProvideHandshake** rule).

Handshake rule of ES_MODULE

```
block

    [CheckAuthorisation rule]
    if mode = ReadyToReceive & ES_ReceiveFrom_C : mssg
        & SuccValid(C, ExtractTicket(mssg), ExtractAuth(mssg))
    then
        K(C, ES) := ExtractKey(mssg)
        clear(mssg)
        mode := ReadyToSend

    [ProvideHandshake rule]
    if mode = ReadyToSend
    then
        if RequiredMutualAuthentication(RequiredService(C))
        then
            ES_SendTo_C : {ExtractTs(ExtractAuth(mssg)) + 1}K(C,ES)
        mode := ReadyToReceive

endblock
```

On reception of the handshake signal from *ES*, *C* performs the mutual authentication test. If it succeeds, *C* becomes ReadyToStart the expected communication with *ES* (**MutualAuthentication** rule).

MutualAuthentication rule of C_MODULE_AUTHORISATION

```
if authenticated & RequiredService & authorised
    & mode = ReadyToReceive & C_ReceiveFrom_ES : CryptedMssg
    & RequiredMutualAuthentication(RequiredService(C))
    & MutualAuthentication(CryptedMssg, mark)
then
    clear(CryptedMssg)
    mode := ReadyToStart
```

This ends the authorisation phase of Kerberos.

We are not interested in describing how the communication between *C* and

ES carries on after the handshake, because this does not belong to Kerberos. Nevertheless, our specification has to be completed by the following two rules which drive the deletion of the expired session keys and tickets.

RevokeAuthentication rule of C_MODULE_AUTHENTICATION and KAS_MODULE

if $expired(K(C,TGS))$
then
  $K(C,TGS) := undef$
  $Ticket(C,TGS) := undef$

RevokeAuthorisation rule of C_MODULE_AUTHORISATION and TGS_MODULE

if $expired(K(C,ES))$
then
  $K(C,ES) := undef$
  $Ticket(C,ES) := undef$

We have now specified the message passing layout of Kerberos. Those entities requiring knowledge of the encryption mechanism have been left abstract and they are treated in the next section.

3.3 The *EncryptionDecryption* Model

The abstract objects of the previous model are here refined through explicit formalisation of the encryption-decryption procedures that serve to build the client's credentials.

3.3.1 Signature Refinement

Kerberos allows the communication between parties by way of encrypted messages and compound messages of encrypted data. Encryption-decryption procedures use the Data Encryption Standard (DES). We formalise them by the functions

$$encrypt : DATA \times KEY \longrightarrow CRYPTDATA$$
$$decrypt : CRYPTDATA \times KEY \longrightarrow DATA$$

such that

$$decrypt(encrypt(t,k),k') = \begin{cases} t & \text{if } k = k' \\ undef & \text{if } k \neq k' \end{cases}$$

DATA is the set of any type of data (structured or not) in cleartext, *CRYPTDATA* is the set of encrypted data. We consider *MESSAGE* as union of a subset of *DATA* and a subset of *CRYPTDATA*. Encrypted messages of the form $\{X, X'\}_{key}$ are now formally defined by $encrypt(\{X,X'\}, key)$. Universes *TICKET* and *AUTH* are both subsets of *CRYPTDATA*.

An element of *MESSAGE* may be a cleartext record – if sent by the client – or a sealed record – if sent by a server – of data, some of which can be encrypted. To read the data stored inside an encrypted message, first this has to be decrypted by the right key, and then data can be obtained as projections on the fields of the record. We thus define the following functions

$key(mssg)$: *KEY*, $ticket(mssg)$: *TICKET*, $auth(mssg)$: *AUTH*

yielding the corresponding fields of the record $mssg = \{key,\ ticket,\ auth\}$.

Once the actual mechanism to get information from a message is known, we can refine the abstract functions used for that purpose in [Section 3.2.1]. Recall that $\mathit{Fun}(a)$ denotes the set of function names occurring in the program of the agent a.

In the sequel of the section, S denotes an element of $\{\mathit{TGS},\ \mathit{ES}\}$.

ExtractKey was used to get a session key from a message. This operation consists now of different steps whether it is performed by C or by S, so we distinguish two cases. C receives the session key from a server inside an encrypted message, so that in $\mathit{Fun}(C)$

$$\mathit{ExtractKey}(\mathit{CryptedMssg}) = \mathit{key}(\mathit{decrypt}(\mathit{CryptedMssg},\ K))$$

where K is the key that was used to create *CryptedMssg*. S gets the session key from the ticket found inside the (cleartext) message received from C. This ticket is encrypted by the server's private key. Thus, in $\mathit{Fun}(S)$

$$\mathit{ExtractKey}(\mathit{mssg}) = \mathit{key}(\mathit{decrypt}(\mathit{ticket}(\mathit{mssg}), K(S)))$$

(The *key* function on tickets is explained below).

Also to extract a ticket from a message, C and S follow different steps. S simply receives the ticket as a component of a message in cleartext from C, so that in $\mathit{Fun}(S)$ the refinement of *ExtractTicket* is just the function rewriting

$$\mathit{ExtractTicket}(\mathit{mssg}) = \mathit{ticket}(\mathit{mssg})$$

By contrast, C receives the ticket inside an encrypted message, so that in $\mathit{Fun}(C)$

$$\mathit{ExtractTicket}(\mathit{CryptedMssg}) = \mathit{ticket}(\mathit{decrypt}(\mathit{CryptedMssg},\ K))$$

where K is the key that was used to create *CryptedMssg*.

Servers are the only parties that receive authenticators, which are always sent by C as components of cleartext messages; as a consequence, $\mathit{ExtractAuth}(\mathit{mssg})$ is just rewritten as $\mathit{auth}(\mathit{mssg})$.

We now explicitly formalise the internal structure of tickets and authenticators, which have been considered so far as atomic objects of the universes TICKET and AUTH respectively.

A ticket is good for a single client C and a single server S. It is a record of the form

$$\{C,\ S,\ \mathit{address}(C),\ K(C,S),\ \mathit{ts},\ \mathit{lifetime}\}_{K(S)}$$

It contains the client's name and network address, the server's name, the session key between client and server, a timestamp and the lifetime of the key. This information is encrypted with the server's secret key. Once the client gets this ticket, it can use it several times to access the server, up until the ticket expires. The client cannot decrypt the ticket (it does not know the server's secret key), but it can present it to the server in its encrypted form. Thus, no one listening on the network can read or modify the ticket as it passes through the communication channels. The structure of the ticket brings the case for the definition of a universe $\mathit{ADDRESS}$ of network addresses and of a function $\mathit{address}(C)$: $\mathit{ADDRESS}$. To extract the components of a ticket $\mathit{Ticket}(C,S)$: TICKET after decryption, we introduce the functions

$\mathit{client}(\mathit{Ticket})$: AGENT, $\mathit{server}(\mathit{Ticket})$: AGENT,
$\mathit{address}(\mathit{Ticket})$: $\mathit{ADDRESS}$, $\mathit{key}(\mathit{Ticket})$: KEY,
$\mathit{ts}(\mathit{Ticket})$: REAL, $\mathit{lifetime}(\mathit{Ticket})$: REAL

yielding the corresponding fields of $\mathit{Ticket} = \{\mathit{client},\ \mathit{server},\ \mathit{address},\ \mathit{key},\ \mathit{ts},\ \mathit{lifetime}\} = \mathit{decrypt}(\mathit{Ticket}(C,S),\ K(S))$.

The function *ProvideTicket* is refined as follows:

$$\mathit{ProvideTicket}(C,S) = \mathit{encrypt}(\{C,S,\mathit{address}(C),K(C,S),\mathit{ts},\mathit{lifetime}\},\ K(S))$$

The session keys are randomly generated by Kerberos. Recall that CT is the current time. The function

$$random : \{C\} \times \{TGS, ES\} \times TIME \longrightarrow KEY$$

yields a random number and refines the function $ProvideKey$ as follows:

$$ProvideKey(C,S) = random(C,S,CT)$$

Each session key expires after a predetermined lifetime, which is usually of the order of several (eight) hours for authentication keys, and of some minutes for authorisation keys. This leads to the definition of the parameters

$AuthLife$: $REAL$, upper bound on the lifetime of an authentication key;

$ServLife$: $REAL$, upper bound on the lifetime of a service key.

Although the value of the lifetime might differ from ticket to ticket, only the two upper bounds described above are commonly used.

Since session keys are sent inside tickets, tickets have in practice the same lifetimes as keys. So, when we say that a ticket has not expired (or also a ticket is fresh), we mean that the key it contains has not expired. We refine accordingly the predicate *expired* as follows:

$expired(K(C,S)) \equiv$

CT - $ts(decrypt(Ticket(C,S),\ K(S))) > lifetime(decrypt(Ticket(C,S),\ K(S)))$

the lifetime on the right being $AuthLife$ if $S = TGS$, $ServLife$ if $S = ES$. This refinement of *expired* updates only $Fun(KAS)$ and $Fun(TGS)$ because KAS and TGS are the only agents having access to the components of the tickets they have built. Thus, in $Fun(C)$ *expired* is a monitored predicate which is true on a key K if and only if the copy of K saved in the memory of the server that generates it, has expired. Such a differentiation comes from a lack of detail in [Miller et al. 89], which is better explained in the next section.

Each time C wishes to contact S, it generates an authenticator that contains its own name and address and a timestamp, sealed with the session key:

$$\{C,\ address(C),\ ts\}_{K(C,S)}$$

To extract the components, we introduce the functions

$client(Auth)$: $AGENT$, $address(Auth)$: $ADDRESS$, $ts(Auth)$: $REAL$

yielding the corresponding fields of

$$Auth = \{client,\ address,\ ts\} = decrypt(Auth(C,K(C,S)),\ K(C,S)).$$

Therefore, the function $ExtractTs$ is rewritten by ts.

The function $BuildAuth$ is refined as follows:

$$BuildAuth(C,\ K(C,S)) = encrypt(\{C,\ address(C),\ CT\},\ K(C,S))$$

Authenticators expire after a short time of a few minutes [Miller et al. 89], so that we introduce the suitable parameter

$DeltaAuth$: $REAL$, upper bound on the lifetime of an authenticator.

On reception of a message from C, S has to authenticate C using as credentials the ticket and the authenticator found inside the message. S decrypts the ticket using its private key and uses the session key found inside the ticket to decrypt the authenticator. If the client's name and network address in the ticket match those in the authenticator, the timestamp in the ticket has not expired, the network address in the authenticator matches that of the sender, and the timestamp in the authenticator is sufficiently recent, then the request is taken as legitimate. ES also makes sure that the authenticator is not a reply of an old one by checking that the timestamp it contains has never been received before inside any authenticator. For this purpose, ES keeps a list of the timestamps of the authenticators received from C in $TimeStamp$: $REAL^*$, updated by the function

$$append : REAL \times REAL^* \longrightarrow REAL^*$$

This test is formally specified by the predicate $SuccValid(C, Ticket,\ Auth)$, $Ticket$: $TICKET$ and $Auth$: $AUTH$, now refined as follows (between square brackets the test by ES only):

$$\begin{aligned}
&SuccValid(C, Ticket,\ Auth)\\
&\quad \equiv defined(ticket)\\
&\qquad \&\ defined(auth)\\
&\qquad \&\ client(ticket) = client(auth)\\
&\qquad \&\ address(ticket) = address(auth) = address(C)\\
&\qquad \&\ (CT\ \text{-}\ ts(ticket)) \leq lifetime(ticket)\\
&\qquad \&\ (CT\ \text{-}\ ts(auth)) \leq DeltaAuth\\
&\qquad [\&\ ts(auth) \notin\ TimeStamp]
\end{aligned}$$

where $ticket = decrypt(Ticket, K(Self))$
and $auth = decrypt(Auth,\ key(decrypt(Ticket, K(Self))))$.

At the other end of the communication, C performs some tests on the received messages as well; they are now reconsidered in turn.

C checks that the message received from KAS and TGS is recent by the predicate $CheckValid(CryptedMssg,\ mark)$ which is true when $CryptedMssg$ is recent. We need now to formalise this concept of "recent message" which is not stated more precisely in [Miller et al. 89]'s informal description. Our model assumes the underlying communication protocol to be *reliable* (if agent X sends agent Y message $mssg$, Y receives $mssg$ with no alterations). Without loss of generality, the model also assumes the transmission of a message to be instantaneous and formalises it as an atomic operation by extension of the universe $MESSAGE$; as a matter of facts, for any transmission both sender and receiver are simultaneously defined. Otherwise, further assumptions on the time of delivery over the network would have been needed, which go well beyond the aims of the paper. By contrast, we are interested in modelling the *time of reaction* of a server, i.e. how long a server does take to reply to a message, and we define

d_r: $REAL$, upper bound on the time of reaction of a server.

Therefore, we call *recent* the reply to a message if it has been issued within time d_r from the issue of the original message. In the light of these considerations, $CheckValid$ can be refined as follows:

$$\begin{aligned}
&CheckValid(CryptedMssg,\ mark)\\
&\quad \equiv ts(decrypt(CryptedMssg,\ K))\ \text{-}\ ts(mark) < d_r
\end{aligned}$$

where K is the client's coded password or its authentication key according to whether the $CryptedMssg$ has been received from KAS or TGS.

The test on the password, abstractly specified by the predicate $CheckPassword(CryptedMssg,\ CodePsw(C))$, consists in checking if the coded password of the client can decrypt the message received from KAS. Therefore,

$$\begin{aligned}
&CheckPassword(CryptedMssg,\ CodePsw(C))\\
&\quad \equiv defined(decrypt(CryptedMssg,\ code(AskPsw(C))))
\end{aligned}$$

where $code(AskPsw(C))$: KEY yields the coded version, using the standard Unix one-way encryption algorithm, of the client's password in cleartext given by the monitored function $AskPsw$.

The mutual authentication test performed by C after the handshake with *ES*, consists in checking the value of the handshake function sent by *ES*. Thus,

$$\begin{aligned}&\mathit{MutualAuthentication}(\mathit{CryptedMssg},\ \mathit{mark})\\&\quad\equiv \mathit{ts}(\mathit{decrypt}(\mathit{CryptedMssg},\ K(C{,}ES))) = \mathit{ts}(\mathit{mark}) + 1\end{aligned}$$

3.3.2 Module Refinement

Some of the actions that were done atomically by abstract functions in the previous model are now carried out in more than one step. So, we now comment on those (sub)rules that involve new definitions. The new, complete program can be found in the next section.

Both rule **ProvideAuthentication** and rule **ProvideAuthorisation** comprise a further step because the ticket can be built only after the creation of the session key that it has to include. The lifetime of the session key inside the messages sent to C in both the rules seems redundant information to the authors. In fact, it is not clear in [Miller et al. 89] how C might use such information to check the expiration time of a session key. This is why *expired* is still a monitored predicate in $\mathit{Fun}(C)$.

The test on the password in rule **GetAuthentication** is done in more than one step. As soon as C receives the *KAS* reply, it prompts the user for the password (in cleartext) – recall that C formalises a process running on behalf of a user – by the monitored function *AskPsw*. If the password is right, its coded version $\mathit{code}(\mathit{password})$ matches the client's private key used by *KAS* to encrypt the message, and can serve successfully for decryption. The user password is then erased to prevent possible leaks.

The last remark concerns the refinements of rule **RevokeAuthentication** and of rule **RevokeAuthorisation**. They remain unchanged in the module of C, whereas they are refined in the modules of *KAS* and *TGS* by the new definition of the predicate *expired*. The reason of this differentiation is that only these two servers know, and therefore can control, the lifetimes of the session keys.

The following theorem expresses the refinement correctness.

Theorem 1 (Refinement) *The runs of the 'MessagePassing' model and the runs of the 'EncryptionDecryption' model are in one-to-one correspondence.*

The proof is trivial. It is sufficient to verify that refined functions take the same value as the corresponding abstract functions and homonymous rules model the same agent behaviour. □

Remark. By the Thoeorem 1, the refined model 'EncryptionDecryption' is a *correct implementation* of the 'MessagePassing' model.

3.3.3 The Program

Variables *mssg* and *CryptedMssg* range over *MESSAGE*.

Authentication Phase

$\boxed{\text{C\_MODULE\_AUTHENTICATION}}$

RequireAuthentication rule

```
if ¬authenticated & mode = ReadyToSend
then
    C_SendTo_KAS : {C, TGS, CT}
    mark := ⟨TGS,CT⟩
    mode := ReadyToReceive
```

GetAuthentication rule

```
if ¬authenticated & mode = ReadyToReceive
   & C_ReceiveFrom_KAS : CryptedMssg
then
    if defined(password)
    then
       if defined(decrypt(CryptedMssg, code(password)))
          & CheckValid(CryptedMssg, mark)
       then
          K(C,TGS) := key(decrypt(CryptedMssg, code(password)))
          Ticket(C,TGS) := ticket(decrypt(CryptedMssg, code(password)))
          clear(password)
          clear(CryptedMssg)
          mode := ReadyToSend
    else
       password := AskPsw(C)
```

RevokeAuthentication rule

```
if expired(K(C,TGS))
then
    K(C,TGS) := undef
    Ticket(C,TGS) := undef
```

$\boxed{\text{KAS\_MODULE}}$

KasReply rule

```
block

    [CheckIdentity rule]
    if mode = ReadyToReceive & KAS_ReceiveFrom_C : mssg & defined(K(C))
    then
        clear(mssg)
        mode := ReadyToSend

    [ProvideAuthentication rule]
    if mode = ReadyToSend
    then
        if defined(K(C,TGS))
        then
           if defined(Ticket(C,TGS))
           then
              KAS_SendTo_C :
                encrypt({K(C,TGS),Ticket(C,TGS),TGS,CT,AuthLife}, K(C))
              mode := ReadyToReceive
           else
              Ticket(C,TGS) :=
                 encrypt({C,TGS,address(C),K(C,TGS),CT,AuthLife}, K(TGS))
        else
           K(C,TGS) := random(C, TGS, CT)

endblock
```

RevokeAuthentication rule

```
if (CT − ts(decrypt(Ticket(C,TGS), K(TGS)))) > AuthLife
then
    K(C,TGS) := undef
    Ticket(C,TGS) := undef
```

Authorisation Phase

C_MODULE_AUTHORISATION

RequireAuthorisation rule

```
if authenticated & RequiredService & ¬authorised & mode = ReadyToSend
then
    if defined(Auth(C, K(C,TGS)))
    then
        C_SendTo_TGS : {Auth(C, K(C,TGS)), Ticket(C,TGS), ES}
        mark := ⟨ES, CT⟩
        clear(Auth(C, K(C,TGS)))
        mode := ReadyToReceive
else
    Auth(C, K(C,TGS)) := encrypt({C, address(C), CT}, K(C,TGS))
```

NegotiateService rule

```
block

    [GetAuthorisation rule]
    if authenticated & RequiredService & ¬authorised
        & mode = ReadyToReceive & C_ReceiveFrom_TGS : CryptedMssg
        & CheckValid(CryptedMssg, mark)
    then
        K(C,ES) := key(decrypt(CryptedMssg, K(C,TGS)))
        Ticket(C,ES) := ticket(decrypt(CryptedMssg, K(C,TGS)))
        clear(CryptedMssg)
        mode := ReadyToSend

    [ShowAuthorisation rule]
    if authenticated & RequiredService & authorised
        & mode = ReadyToSend
    then
        if defined(Auth(C, K(C,ES)))
        then
            C_SendTo_ES : {Auth(C, K(C,ES)), Ticket(C,ES)}
            clear(Auth(C, K(C,ES)))
            if RequiredMutualAuthentication(RequiredService(C))
            then
                mark := ⟨CT⟩
                mode := ReadyToReceive
            else
                mode := ReadyToStart
        else
            Auth(C, K(C,ES)) := encrypt({C, address(C), CT}, K(C,ES))

endblock
```

MutualAuthentication rule

```
if authenticated & RequiredService & authorised
   & mode = ReadyToReceive & C_ReceiveFrom_ES : CryptedMssg
   & RequiredMutualAuthentication (RequiredService (C))
   & MutualAuthentication (CryptedMssg, mark)
then
   clear (CryptedMssg)
   mode := ReadyToStart
```

RevokeAuthorisation rule

```
if expired (K (C,ES))
then
   K (C,ES) := undef
   Ticket (C,ES) := undef
```

$\boxed{\text{TGS\_MODULE}}$

TgsReply rule

```
block

   [CheckAuthentication rule]
   if mode = ReadyToReceive & TGS_ReceiveFrom_C : mssg
      & SuccValid (C, ticket (mssg), auth (mssg))
   then
      K(C, TGS) := key (decrypt (ticket (mssg), K (TGS)))
      clear (mssg)
      mode := ReadyToSend

   [ProvideAuthorisation rule]
   if mode = ReadyToSend
   then
      if defined (K (C,ES))
      then
         if defined (Ticket (C,ES))
         then
            TGS_SendTo_C :
              encrypt ({K (C,ES), Ticket (C,ES), ES, CT, ServLife}, K(C, TGS))
            clear (K(C, TGS))
            mode := ReadyToReceive
         else
            Ticket(C, ES) :=
               encrypt ({C, ES, address (C), K (C,ES), CT, ServLife}, K (ES))
      else
         K(C, ES) := random (C, ES, CT)

endblock
```

RevokeAuthorisation rule

```
if (CT − ts (decrypt (Ticket (C,ES), K (ES)))) > ServLife
then
   K (C,ES) := undef
   Ticket (C,ES) := undef
```

$\boxed{\text{ES\_MODULE}}$

Handshake rule

```
block

    [CheckAuthorisation rule]
    if mode = ReadyToReceive & ES_ReceiveFrom_C : mssg
       & SuccValid (C, ticket (mssg), auth (mssg))
    then
       K(C, ES) := key (decrypt (ticket (mssg), K (ES)))
       clear (mssg)
       mode := ReadyToSend

    [ProvideHandshake rule]
    if mode = ReadyToSend
    then
       if RequiredMutualAuthentication (RequiredService (C))
       then
          ES_SendTo_C :
             encrypt ( {ts (decrypt (auth (mssg), K (C,ES))) + 1}, K (C,ES))
       TimeStamp :=
          append (ts (decrypt (auth (mssg), K (C,ES))), TimeStamp)
       mode := ReadyToReceive

endblock
```

3.4 Correctness

The aim of this section is showing that the Kerberos Authentication System is correct, i.e. it allows a client to obtain a session key for the required service. All of the correctness properties are proved over runs called *regular*, defined below.

The correct operation of Kerberos relies on the existence of certain conditions, some to hold in any state, some to characterise the starting up environment, others to guarantee that the system can work as it is meant to do. We call them *global*, *initial* and *working*, respectively.

3.4.1 Global Conditions

Kerberos assumes the external condition that each agent is provided with a private key known only to the agent itself and saved in the Kerberos database. At the specification level, this gives rise to the following *global condition* to hold in any state:

G1. $\mathit{defined}(K(C))$ & $\mathit{defined}(K(TGS))$ & $\mathit{defined}(K(ES))$

KAS does not have its own key because it communicates with C using the client's private key $K(C)$.

The condition that a client can not be authorised if not authenticated also must hold as *global condition* in any state:

G2. $\neg \mathit{authenticated}(C) \longrightarrow \neg \mathit{authorised}(C)$

3.4.2 Initial Conditions

A set of initial conditions is formalised below for each of the two phases of the protocol. The second set differs from the first in the existence of an authentication ticket and an authentication key for C and in the presence of an external request for a service. Either I1 or I2 must hold in any initial state.

I1. [*Authentication phase*]
- $Ticket(C,TGS,Self) = undef,\ Self \in \{C,\ KAS\}$
- $K(C,TGS,Self) = undef,\ Self \in \{C,\ KAS\}$
- $mode(C) = \mathsf{ReadyToSend}$
- $mode(KAS) = \mathsf{ReadyToReceive}$
- $sender(X) = receiver(X) = undef,\ \forall X \in MESSAGE$

I2. [*Authorisation phase*]
- $Ticket(C,TGS,Self) \neq undef,\ Self \in \{C,\ KAS\}$
- $Ticket(C,ES,Self) = undef,\ Self \in \{C,\ TGS\}$
- $K(C,TGS,Self) \neq undef,\ Self \in \{C,\ KAS\}$
- $K(C,ES,Self) = undef,\ Self \in \{C,\ TGS\}$
- $mode(C) = \mathsf{ReadyToSend}$
- $mode(Self) = \mathsf{ReadyToReceive},\ Self \in \{TGS,ES\}$
- $sender(X) = receiver(X) = undef,\ \forall X \in MESSAGE$
- $RequiredService(C) \neq undef$

By the global condition G2, a client is not authorised in any state satisfying conditions I1.

3.4.3 Working Conditions

Not all the runs starting from a state which satisfies a set of initial conditions, are guaranteed to end successfully – in some of those runs the client might be unable to gain authentication or authorisation. Intuitively, suppose the coded version of the client's password does not match the client's key saved in the Kerberos database, then the protocol would block. Suppose the agents are not synchronized: a server could take as fresh the client's credentials (tickets and authenticators) when they are not; or, if a server's reply is too slow, a client could reject it as an intruder's faked message. Obviously, the protocol is never likely to work properly in any of such situations.

This brings up the necessity to guarantee some minimal conditions on our model, that we call *working conditions*, to exclude undesirable situations like those described above. Each condition is first informally introduced and then formally described.

The operation of Kerberos relies on the use of timestamps, so the clock system must be reliable.

W1. [*Monotonicity of the clock*]
The values of CT at states $S_0, S_1, S_2, \ldots$ form a strictly increasing sequence. If there is a final state, then $CT = \infty$ in the final state.

The client is provided with regular identifier and password.

W2. [*Password correctness*]
The client's key saved in the Kerberos database matches the coded version of the client's password: $K(C) = code(AskPsw(C))$.

A server always replies to a message of the client within time d_r.

W3. [*Upper bound on a server's reaction time*]
Let $S \in \{KAS, TGS\}$, $m \in MESSAGE$, $i \in \mathcal{N}$. If C sends m to S at state S_i – i.e. $[sender(m) = C]_{S_{i+1}}$ & $[receiver(m) = S]_{S_{i+1}}$ – then S replies by a message m' within time d_r – i.e. there exist $m' \in MESSAGE$, $j \in \mathcal{N}$, $j > i$, such that $[sender(m') = S]_{S_{j+1}}$ & $[receiver(m') = C]_{S_{j+1}}$ & $([CT]_{S_j} - [CT]_{S_i} \leq d_r)$.

Any of the client's requests has to be honoured by the appropriate server, i.e. both *TGS* and *ES* must receive authenticators and tickets that have not expired:

W4. [*Validity of the client's credentials*]
Let *term* be a credential of the client's (ticket or authenticator) and let t be the timestamp of *term*; it is $t = [CT]_{S_i}$ for some state S_i, $i \in \mathcal{N}$. If S receives a message m containing *term* from C at some state S_j, $j > i$ – i.e. $[sender(m) = C]_{S_j}$ & $[receiver(m) = S]_{S_j}$ – then the following relation holds:

$$[CT]_{S_j} - [CT]_{S_i} \leq \Delta$$

where

$$\Delta = \begin{cases} DeltaAuth \text{ if } term = Auth(C,\ K(C,S)),\ S \in \{TGS, ES\} \\ AuthLife \ \text{ if } term = Ticket(C,\ TGS) \\ ServLife \ \text{ if } term = Ticket(C,\ ES) \end{cases}$$

Note that these four conditions are the minimum necessary to guarantee the application of the complete sequence of rules to simulate a complete execution of Kerberos.

3.4.4 Regular Runs

We can take advantage of the fact that agents of our ASM fires only one rule at a time, therefore for our purposes we can consider *sequential runs* of distributed ASMs as defined in [Section 3.1].

Definition 1. An *authentication* (*authorisation*) *regular run* is any sequential run of our distributed ASM such that

(i) global conditions G1 and G2 hold;
(ii) the initial state satisfies initial conditions I1 (I2);
(iii) working conditions W1 to W4 hold.

3.4.5 Proof of Correctness

We now prove two correctness theorems stating that the system works as it is supposed to do: a legitimate client can authenticate itself to KAS, and an authenticated client requiring a service can obtain from TGS a session key to start the communication with ES.

Theorem 2 (Authentication Phase Correctness)
At some state of any authentication regular run, a client becomes authenticated.

Proof. We aim to show that exists $n \in \mathcal{N}$ such that $[authenticated = true]_{S_n}$.

Let S_0 be the initial state satisfying the initial conditions I1. In S_0, C is not authenticated, so that we have the following transition:

$$S_0 \longrightarrow S_1 \qquad \text{by } \mathbf{RequireAuthentication}$$

Integrity constraint G1 allows the transition

$$S_1 \longrightarrow S_2 \qquad \text{by } \mathbf{CheckIdentity}$$

and then, through the transitions

$$S_2 \longrightarrow S_3 \longrightarrow S_4 \longrightarrow S_5 \qquad \text{by } \mathbf{ProvideAuthentication}\text{s}$$

KAS builds and sends C the authentication ticket and the session key. Conditions W2 and W3 (with $i = 0$, $j = 4$, $m = \{C, TGS, CT\}$, $m' = encrypt(\{K(C,TGS), Ticket(C,TGS),\ TGS,\ ts,\ AuthLife\},\ K(C)))$ allow the transitions

$$S_5 \longrightarrow S_6 \longrightarrow S_7 \qquad \text{by } \mathbf{GetAuthentication}\text{s}$$

In S_7 we have

$$[defined(Ticket(C,TGS,C))]_{S_7}, \quad [defined(K(C,TGS,C))]_{S_7}$$

which translates into the conclusion. □

Theorem 3 (Authorisation Phase Correctness) *At some state of any authorisation regular run, an authenticated client shares a session key with the end server and is* ReadyToStart *the communication with it.*

Proof. Let S_{k_0}, be the initial state of the authorisation phase satisfying conditions I2. The thesis is proved through the following intermediate results:

(i) there exists a state S_h, $h > k_0$, such that an authenticated client C is authorised in S_h, i.e. $[authorised = true]_{S_h}$;
(ii) there exists a state S_i, $i > h$, such that an authorised client C:
  - shares a private session key with ES, i.e. $[defined(K(C,ES,C))\ \&\ defined(K(C,ES,ES))\ \&\ K(C,ES,C) = K(C,ES,ES)]_{S_i}$, and
  - is ReadyToStart the communication with ES, i.e. $[mode(C) = \mathsf{ReadyToStart}]_{S_i}$.

Initial and working conditions let the following regular run exist:

$$
\begin{array}{rl}
S_{k_0} \longrightarrow S_{k_0+1} \longrightarrow S_{k_0+2} & \text{by } \textbf{RequireAuthorisations} \\
S_{k_0+2} \longrightarrow S_{k_0+3} & \text{by } \textbf{CheckAuthentication} \\
S_{k_0+3} \longrightarrow S_{k_0+4} \longrightarrow S_{k_0+5} \longrightarrow S_{k_0+6} & \text{by } \textbf{ProvideAuthorisations} \\
S_{k_0+6} \longrightarrow S_{k_0+7} & \text{by } \textbf{GetAuthorisation} \\
S_{k_0+7} \longrightarrow S_{k_0+8} \longrightarrow S_{k_0+9} & \text{by } \textbf{ShowAuthorisations} \\
S_{k_0+9} \longrightarrow S_{k_0+10} & \text{by } \textbf{CheckAuthorisation}
\end{array}
$$

Rule **CheckAuthentication** fires to S_{k_0+2} because guard *SuccValid*(*C, ticket*(*mssg*)*, auth*(*mssg*)) checks out as follows. Its first four conditions holds by construction of the run and constraint G1. Condition W4 guarantees the validity of the ticket for $term = Ticket(C,TGS)$, $S = TGS$, $j = k_0 + 2$, $m = \{Auth(C,K(C,TGS)),\ Ticket(C,TGS),\ ES\}$ and $i = h$, $h < k_0$ such that $[defined(Ticket(C,TGS,KAS))]_{S_{h+1}}$ & $[undefined(Ticket(C,TGS,KAS))]_{S_h}$. For the same values of S, j and m, condition W4 also guarantees the validity of the authenticator for $term = Auth(C,K(C,TGS))$ and $i = k_0$.

Rule **GetAuthorisation** fires to S_{k_0+6} thanks to working condition W3, taking $i = k_0 + 1$, $j = k_0 + 5$, $m = \{Auth(C,K(C,TGS)),\ Ticket(C,TGS),\ ES\}$, $m' = encrypt(\{K(C,ES),\ Ticket(C,ES),\ ES,\ ts,\ ServLife\},\ K(C,TGS))$. In S_{k_0+7} holds

$$[defined(Ticket(C,ES,C))]_{S_{k_0+7}}\ \&\ [defined(K(C,ES,C))]_{S_{k_0+7}},$$

that is $[authorised = true]_{S_{k_0+7}}$, which proves (i) for $h = k_0 + 7$.

Firing of rule **ProvideAuthorisation** to S_{k_0+3} builds what is going to be a service key for the communication session between C and ES. Let $ServiceKey = [K(C,ES,TGS)]_{S_{k_0+4}}$. The fire of rule **GetAuthorisation** causes $[defined(K(C,ES,C))]_{S_{k_0+7}}$ and the fire of rule **CheckAuthorisation** gives rise to $[defined(K(C,ES,ES))]_{S_{k_0+10}}$. It is straightforward to verify that, by construction of the run,

$$[K(C, ES, C)]_{S_{k_0+10}} = [K(C, ES, ES)]_{S_{k_0+10}} = ServiceKey \tag{1}$$

and that these two locations will not be updated on the same run.

If the mutual authentication is not required for the service, then by rule **ShowAuthorisation** fired to S_{k_0+9}, C has taken mode ReadyToStart. Since the location $mode(C)$ is not updated through the transition from S_{k_0+9} to S_{k_0+10}, (ii) is proved for $i = k_0 + 10$.

If the mutual authentication is required, then the run continues with the transitions:

$$
\begin{array}{rl}
S_{k_0+10} \longrightarrow S_{k_0+11} & \text{by } \textbf{ProvideHandshake} \\
S_{k_0+11} \longrightarrow S_{k_0+12} & \text{by } \textbf{MutualAuthentication}
\end{array}
$$

Rule **CheckAuthorisation** fires to S_{k_0+9}, thus starting the handshake between C and ES, because *SuccValid*(*C, ticket*(*mssg*)*, auth*(*mssg*)) checks out as follows. Its first four conditions holds by construction of the run and constraint G1. Condition W4 guarantees the validity of the ticket for $term = Ticket(C,ES)$, $S = ES$, $j = k_0 + 9$, $m = \{Auth(C,K(C,ES)),\ Ticket(C,ES)\}$, $i = k_0 + 4$, and the validity of the authenticator for $term = Auth(C,K(C,ES))$, $i = k_0 + 7$.

Rule **MutualAuthentication** fires to S_{k_0+11} because the mutual authentication test succeeds by construction of the run.

At state S_{k_0+12}, $[mode(C) = \mathsf{ReadyToStart}]_{S_{k_0+12}}$ holds. The relation (1) is still true because it involves locations which are not updated by the transitions from S_{k_0+10} to S_{k_0+12}. Therefore, (ii) is proved for $j = k_0 + 12$. $\square$

3.5 The *MultipleClients* Model

We now go back to the real Kerberos System depicted in [Fig.1].

We have considered so far the interaction between a single client C and a single end server *ES*. In the real environment there are more than one user logging in on a workstation to access a certain number of network services. First, in this section we refine C by the processes running on a workstation, and then in [Section 3.6] we describe how to get a program that allows several end servers.

3.5.1 Signature Refinement

The real system is a network of hosts. For convenience, we make no difference between the host itself and its daemon process, so that we define the universe $DAEMON = \{KAS, TGS, ES, WS\}$. *WS* represents the workstation on which users login and eventually require services.

We call *legitimate* a user who is provided with an account on the workstation, so he owns a network identifier and a private password. The universe *USER* represents the identifiers of the legitimate users.

Each user may own a certain number of processes running on the workstation. This leads to the universe *PROCESS* and to the function

$$owner : PROCESS \longrightarrow USER$$

yielding the identifier of the owner of a given process.

Any process (either one owned by a user, or a daemon process) is an agent with its own operation mode, so we redefine the universe *AGENT* as *DAEMON* $\cup$ *PROCESS*. The agent *WS* is special because it is able to create new agents in *PROCESS* by importing reserve elements. More precisely, *WS* creates a new *login* process agent every time a user tries to login on the workstation, and a *service* process agent every time an authenticated user requires a service.

A login process has to authenticate itself (on behalf of its owner) with *KAS*, thus working as the 'old' agent C during the authentication phase. A service process is born authenticated (it might be envisaged as the child process of a login process already authenticated) and has to get authorisation from *TGS* to start the communication with *ES*; so it works as the 'old' C during the authorisation phase. Therefore, the 'old' C_MODULE is now divided into two modules: the LP_MODULE for the operation of login processes, and the SP_MODULE for the operation of service processes.

If $U \in USER$, the monitored predicate $RequireLogin(U)$ is true when the user U wants to login, and U is associated to its *login* process P by the function

$$login : USER \longrightarrow PROCESS$$

The following implication must hold in any state

$$(owner(P) = U \,\&\, Mod(P) = \text{LP\_MODULE}) \longrightarrow login(U) = P$$

A service process is created for any user's request of a network service. The following functions

$$task : PROCESS \longrightarrow SERVICE$$
$$process : USER \times SERVICE \longrightarrow PROCESS$$

yield respectively the task associated to a given service process, and the service process associated to a given user's request. They have to be thought as one another's inverse in the sense that

$$task(process(U, RequiredService(U))) = RequiredService(U)$$

which we put as a constraint. Therefore, in any state there holds

$$task(P) = RequiredService(U) \rightarrow defined(process(U, RequiredService(U))).$$

To keep track of the process to which the server has to reply, we define the function

$$P\colon DAEMON \longrightarrow PROCESS$$

and we abbreviate $P(Self)$ by P.

The function *Mod* associates *KAS* to KAS_MODULE, *TGS* to TGS_MODULE, *ES* to ES_MODULE, *WS* to WS_MODULE, a login process P to LP_MODULE, a service process P to SP_MODULE.

Because $AGENT$ has different elements at this level, some definitions of functions need to be reconsidered.

The unary function K yielding all the private keys saved in the Kerberos database, is redefined as

$$K : DAEMON \cup USER \longrightarrow KEY.$$

The functions *mark* and *TimeStamp*, the predicates *authenticated* and *authorised* are now parameterised by $P \in PROCESS$.

The functions *ProvideTicket*, *BuildAuth*, *random, address, Ticket, Auth*, the 3-ary function K and predicate *SuccValid* are now defined on *PROCESS* instead of on *C*. Functions *ProvideTicket* and *BuildAuth* are refined as follows:

$$ProvideTicket(P,S) =$$
$$encrypt(\{owner(P),\ S,\ address(P),\ K(P,S),\ ts,\ lifetime\},\ K(S))$$
$$BuildAuth(P,\ K(P,S)) = encrypt(\{owner(P),\ address(P),\ CT\},\ K(P,S))$$

Functions *RequiredService* and *AskPsw* are redefined on *USER*. Note that the function *mode* is not defined on *WS*. The functions not mentioned here remain unchanged.

3.5.2 Module Refinement

Intuitively, each agent $P \in PROCESS$ works as the agent C in the authentication phase if it is a login process, or as C in the authorisation phase if it is a service process. However, in the previous model we made no distinction between an agent and its identifier.

We now distinguish a process running on the workstation from its user's identifier. Thus, the C_MODULE_AUTHENTICATION becomes the LP_MODULE and the C_MODULE_AUTHORISATION becomes the SP_MODULE replacing the occurrences of C by P, except those having the meaning of user's identifier. That is, C has to be replaced by $owner(P)$ when it occurs as argument of the function *AskPsw*, or inside authenticators, or inside the message sent to *KAS*. The condition $defined(RequiredService(C))$ (abbreviated as *RequiredService*) no longer occurs among the guards of the rules of the SP_MODULE because we no longer need distinguish the authentication phase from the authorisation phase of a client, due to the definition of separate modules for a login process and a service process.

KAS_MODULE, TGS_MODULE, ES_MODULE need the occurrences of C to be replaced by P except those having the meaning of user's identifier, i.e. occurring as an argument of the unary function K or inside tickets, according to the function refinements.

Rules **CheckIdentity**, **CheckAuthentication** and **CheckAuthorisation** contain now an update of the form $P := \mathit{sender}(\mathit{mssg})$ to keep track of the process to which the server has to reply. It was not needed at the previous level, where servers only interacted with C.

WS_MODULE is new and describes the actions of the workstation as an interface between a legitimate user and Kerberos. By rule **CreateLoginProcess**, it creates a new login process (agent) per each external request for login. Each login process has to go through the authentication with KAS, so its *Mod* function is updated to LP_MODULE. By rule **CreateServiceProcess**, WS creates a new service process (agent) per each user's request for a service. The new agent is created if the user is already authenticated, i.e. if its login process is still authenticated. Incidentally, the predicate *authenticated* is interactively updatable for WS. Each service process inherits the authentication ticket and the authentication key from the (parent) login process, and then has to go through the authorisation with TGS before beginning the communication with ES. Its *Mod* function is therefore updated to SP_MODULE. The SP_MODULE also contains the **RevokeAuthentication** rule because a service process loses its authentication when the authentication key expires.

The complete program is listed in [Section 3.5.4].

Given a process P, we denote by $\mathcal{A}(P)$ the ASM restricted to the agents involved in a computation of P (i.e. P, KAS, TGS and ES).

Theorem 4 (Refinement)
Let P be a login (service) process. The runs in $\mathcal{A}(P)$ are in one-to-one correspondence with the runs in the authentication (authorisation) phase of the 'EncryptionDecryption' model.

It is straightforward to verify that rules of LP_MODULE (SP_MODULE) model the same agent behavour as the homonymous rules of C_MODULE in the authentication (authorisation) phase. □

Remark. By the Theorem 4 we can state that each login (service) process in the 'MultipleClients' model *implements correctly* the operation of the client agent in the authentication (authorisation) phase of the 'EncryptionDecryption' model.

3.5.3 Correctness Analysis

In the light of the previous refinement theorem, we must suitably reformulate the global, initial and working conditions, and the definition of regular run of [Section 3.4]. In fact, having refined the agent C by the concept of workstation with users and processes, all conditions previously defined on C must be ascribed to the right agents.

Let us fix an element P of $\mathit{PROCESS}$; recall the definition of $\mathcal{A}(P)$ from the previous section.

Definition 2. A (*sequential*) *run of* P is a sequence $\varrho_P = S_0, S_1, \ldots, S_k, \ldots$ of states of $\mathcal{A}(P)$ such that, for each positive k, S^-_{k+1} is obtained from S_k by executing one rule $r(k)$ of $\mathcal{A}(P)$ at S_k.

Note that ϱ_P uniquely determines rules $r(k)$ and each rule uniquely determines the agent whose program contains the rule.

It is straightforward to verify that each run ϱ_P is a linearization of a partially ordered run of $\mathcal{A}$, and is sequential in the sense of sequential runs of distributed ASM as defined in [Section 3.1]. Therefore, to prove correctness properties of the whole model, it is enough to reason on the linearization ϱ_P.

The following *global conditions* hold in any state of $\mathcal{A}(P)$:

G1$'$. $\forall U \in \mathit{USER}\ \mathit{defined}(K(U))\ \&\ \mathit{defined}(K(TGS))\ \&\ \mathit{defined}(K(ES))$
G2$'$. $\neg \mathit{authenticated}(P) \longrightarrow \neg \mathit{authorised}(P)$

I1 or I2 now hold on $\mathcal{A}(P)$ according with $\mathit{Mod}(P) =$ LP_MODULE or $\mathit{Mod}(P) =$ SP_MODULE respectively (C must be replaced by P or $\mathit{owner}(P)$ according to the function refinements of [Section 3.5.1] and $\mathit{RequiredService}(C) \neq \mathit{undef}$ by $\mathit{task}(P) \neq \mathit{undef}$). Therefore:

I1$'$. [*Initial conditions of a login process*]
- $\mathit{Ticket}(P, TGS, \mathit{Self}) = \mathit{undef},\ \mathit{Self} \in \{P,\ KAS\}$
- $K(P, TGS, \mathit{Self}) = \mathit{undef},\ \mathit{Self} \in \{P,\ KAS\}$
- $\mathit{mode}(P) = \mathsf{ReadyToSend}$
- $\mathit{mode}(KAS) = \mathsf{ReadyToReceive}$
- $\mathit{sender}(X) \neq P\ \&\ \mathit{receiver}(X) \neq P,\ \forall X \in \mathit{MESSAGE}$

I2$'$. [*Initial conditions of a service process*]
- $\mathit{Ticket}(P, TGS, \mathit{Self}) \neq \mathit{undef},\ \mathit{Self} \in \{P,\ KAS\}$
- $\mathit{Ticket}(P, ES, \mathit{Self}) = \mathit{undef},\ \mathit{Self} \in \{P,\ TGS\}$
- $K(P, TGS, \mathit{Self}) \neq \mathit{undef},\ \mathit{Self} \in \{P,\ KAS\}$
- $K(P, ES, \mathit{Self}) = \mathit{undef},\ \mathit{Self} \in \{P,\ TGS\}$
- $\mathit{mode}(P) = \mathsf{ReadyToSend}$
- $\mathit{mode}(\mathit{Self}) = \mathsf{ReadyToReceive},\ \mathit{Self} \in \{TGS, ES\}$
- $\mathit{sender}(X) \neq P\ \&\ \mathit{receiver}(X) \neq P,\ X \in \mathit{MESSAGE}$
- $\mathit{task}(P) \neq \mathit{undef}$

The *working conditions* can be stated on any $P \in \mathit{PROCESS}$, replacing C by P or $\mathit{owner}(P)$ according to the function refinements of [Section 3.5.1].

Definition 3. A run ϱ_P of a login (service) process P is called *regular* if it satisfies the following conditions:

(i) global conditions G1$'$ and G2$'$ hold;
(ii) the initial state satisfies initial conditions I1$'$ (I2$'$);
(iii) working conditions W1 to W4 hold in any state of ϱ_P.

Corollary 1 (Correctness)
(i) *At some state of any regular run ϱ_P of a login process, P becomes authenticated.*
(ii) *At some state of any regular run ϱ_P of a service process, P shares a session key with the end server and is* ReadyToStart *the communication with it.*

Proof. Correctness Theorem 2 and Refinement Theorem 4 applied to the login process P proves (i). Correctness Theorem 3 and Refinement Theorem 4 applied to the service process P proves (ii). □

Definition 4. A *run* ϱ in $\mathcal{A}$ is called *regular* if the following conditions are satisfied:

(i) the initial state satisfies the following conditions:
  - $\exists\ U \in USER$: $RequiredLogin(U) \neq undef$ & $\forall\ P \in PROCESS$: $\neg(owner(P) = U)$
  - $mode(Self) =$ ReadyToReceive, $Self \in \{KAS, TGS, ES\}$
  - $sender(X) = receiver(X) = undef$, $\forall X \in MESSAGE$

(ii) $\forall P \in PROCESS$, all runs ϱ_P are regular.

As a consequence of the corollary, it is easy to prove the following main result.

Theorem 5 (Main Theorem) *At some state of any regular run, a legitimate user requiring a service shares a session key with the end server providing that service and is* ReadyToStart *the communication with it.*

Proof. Let $U \in USER$ be the legitimate user satisfying the first of the initial conditions holding in the initial state S_0. We have to show that there exists a state S_k, $k > 0$, and a process $P \in PROCESS$ such that the following relation holds in S_k:

$$Mod(P) = \text{SP\_MODULE} \,\&\, owner(P) = U \,\&\, mode(P) = \mathsf{ReadyToStart} \quad (2)$$

By initial conditions only the rule **CreateLoginProcess** can fire in S_0, and a login process *P1* of owner U is created. On *P1* conditions $I1'$ hold, so by the part (i) of Corollary 1, at some state S_h, $h > 0$, holds *authenticated*(*P1*). This condition, together with the hypothesis $RequiredService(U) \neq undef$, allows rule **CreateServiceProcess** to fire. A service process *P2* already authenticated, with owner U and task $RequiredService(U)$, is created. In the current state initial conditions I2$'$ hold on *P2*, so the part (ii) of Corollary 1 proves the (2) taking P as *P2*. □

Remark. The theorems above show that the 'MultipleClients' model is an effective extension of the 'EncryptionDecryption' model for the presence of the agent *WS* to capture every external request for login or for a network service.

3.5.4 The Program

Variable P ranges over *PROCESS*, variables *mssg* and *CryptedMssg* range over *MESSAGE*.

Authentication Phase

$\boxed{\text{WS\_MODULE}}$

CreateLoginProcess rule

```
var U ranges over USER
   if RequiredLogin (U) & undefined (login (U))
   then
      extend PROCESS by P with
         Mod (P) := LP_MODULE
         owner (P) := U
         mode (P) := ReadyToSend
         K (P, TGS) := undef
         Ticket (P, TGS) := undef
         K (P, ES) := undef
         Ticket (P, ES) := undef
```

CreateServiceProcess rule

```
var U ranges over USER
   if RequiredService (U) & authenticated (login(U))
   & undefined (process (U, RequiredService (U)))
   then
      extend PROCESS by P with
         Mod (P) := SP_MODULE
         owner (P) := U
         task (P) := RequiredService (U)
         mode (P) := ReadyToSend
         K (P, TGS) := K (login (U), TGS))
         Ticket (P, TGS) := Ticket (login (U), TGS))
         K (P, ES) := undef
         Ticket (P, ES) := undef
```

$\boxed{\text{LP\_MODULE}}$

RequireAuthentication rule

```
if ¬authenticated (P) & mode (P) = ReadyToSend
then
   P_SendTo_KAS : {owner (P), TGS, CT}
   mark (P) := ⟨TGS, CT⟩
   mode (P) := ReadyToReceive
```

GetAuthentication rule

```
if ¬authenticated (P) & mode (P) = ReadyToReceive
   & P_ReceiveFrom_KAS : CryptedMssg
then
   if defined (password (P))
   then
      if defined (decrypt (CryptedMssg, code (password (P))))
         & CheckValid (CryptedMssg, mark (P))
      then
         K(P, TGS) := key (decrypt (CryptedMssg, code (password (P))))
         Ticket(P, TGS) := ticket (decrypt (CryptedMssg, code (password (P))))
         clear (password (P))
         clear (CryptedMssg)
         mode (P) := ReadyToSend
   else
      password (P) := AskPsw (owner (P))
```

RevokeAuthentication rule

```
if expired (K (P,TGS))
then
    K (P,TGS) := undef
    Ticket (P,TGS) := undef
```

$\boxed{\text{KAS\_MODULE}}$

KasReply rule

```
block

  [CheckIdentity rule]
    if mode = ReadyToReceive & KAS_ReceiveFrom_P : mssg
       & defined (K (owner (P)))
    then
        P := sender (mssg)
        clear (mssg)
        mode := ReadyToSend

  [ProvideAuthentication rule]
    if mode = ReadyToSend
    then
        if defined (K (P,TGS))
        then
           if defined (Ticket (P,TGS))
           then
              KAS_SendTo_P :
                encrypt ({K (P,TGS), Ticket (P,TGS), TGS, CT, AuthLife},
                         K (owner (P)))
              clear (P)
              mode := ReadyToReceive
           else
              Ticket(P,TGS) :=
                encrypt ({owner (P), TGS, address (P), K (P,TGS), CT, AuthLife},
                         K (TGS))
        else
           K(P,TGS) := random (P, TGS, CT)

endblock
```

RevokeAuthentication rule

```
if (CT − ts (decrypt (Ticket (P,TGS), K (TGS)))) > AuthLife
then
    K (P,TGS) := undef
    Ticket (P,TGS) := undef
```

Authorisation Phase

$\boxed{\text{SP\_MODULE}}$

RequireAuthorisation rule

```
if authenticated (P) & ¬authorised (P)
   & mode (P) = ReadyToSend
then
```

```
        if defined (Auth (P, K (P,TGS)))
        then
            P_SendTo_TGS : {Auth (P, K (P,TGS)), Ticket (P,TGS), ES}
            mark (P) := ⟨ES, CT⟩
            clear (Auth (P, K (P,TGS)))
            mode (P) := ReadyToReceive
    else
        Auth (P, K (P,TGS)) := encrypt ( {owner (P), address (P), CT}, K (P,TGS))
```

NegotiateService rule

```
block

    [GetAuthorisation rule]
        if authenticated (P) & ¬authorised (P)
            & mode (P) = ReadyToReceive & P_ReceiveFrom_TGS : CryptedMssg
            & CheckValid (CryptedMssg, mark (P))
        then
            K(P, ES) := key (decrypt (CryptedMssg, K (P,TGS)))
            Ticket(P, ES) := ticket (decrypt (CryptedMssg, K (P,TGS)))
            clear (CryptedMssg)
            mode (P) := ReadyToSend

    [ShowAuthorisation rule]
        if authenticated (P) & authorised (P)
            & mode (P) = ReadyToSend
        then
            if defined (Auth (P, K (P,ES)))
            then
                P_SendTo_ES : {Auth (P, K (P,ES)), Ticket (P,ES)}
                clear (Auth (P, K (P,ES)))
                if RequiredMutualAuthentication (task (P))
                then
                    mark (P) := ⟨CT⟩
                    mode (P) := ReadyToReceive
                else
                    mode (P) := ReadyToStart
            else
                Auth (P, K (P,ES)) := encrypt ({owner (P),address(P),CT}, K (P,ES))

endblock
```

MutualAuthentication rule

```
    if authenticated (P) & authorised (P)
        & mode (P) = ReadyToReceive & P_ReceiveFrom_ES : CryptedMssg
        & RequiredMutualAuthentication (task (P))
        & MutualAuthentication (CryptedMssg, mark (P))
    then
        clear (CryptedMssg)
        mode (P) := ReadyToStart
```

RevokeAuthentication rule

```
    if expired (K (P,TGS))
    then
        K (P,TGS) := undef
        Ticket (P,TGS) := undef
```

RevokeAuthorisation rule

```
if expired(K(P,ES))
then
    K(P,ES) := undef
    Ticket(P,ES) := undef
```

TGS_MODULE

TgsReply rule

```
block

    [CheckAuthentication rule]
        if mode = ReadyToReceive & TGS_ReceiveFrom_P : mssg
            & SuccValid(P, ticket(mssg), auth(mssg))
        then
            K(P, TGS) := key(decrypt(ticket(mssg), K(TGS)))
            P := sender(mssg)
            clear(mssg)
            mode := ReadyToSend

    [ProvideAuthorisation rule]
        if mode = ReadyToSend
        then
            if defined(K(P,ES))
            then
                if defined(Ticket(P,ES))
                then
                    TGS_SendTo_P :
                      encrypt({K(P,ES),Ticket(P,ES),ES,CT,ServLife}, K(P,TGS))
                    clear(K(P,TGS))
                    clear(P)
                    mode := ReadyToReceive
                else
                    Ticket(P,ES) :=
                      encrypt({owner(P),ES,address(P),K(P,ES),CT,ServLife},
                              K(ES))
            else
                K(P,ES) := random(P, ES, CT)

endblock
```

RevokeAuthorisation rule

```
if (CT − ts(decrypt(Ticket(P,ES), K(ES)))) > ServLife
then
    K(P,ES) := undef
    Ticket(P,ES) := undef
```

ES_MODULE

Handshake rule

```
block

    [CheckAuthorisation rule]
        if mode = ReadyToReceive & ES_ReceiveFrom_P : mssg
            & SuccValid(P, ticket(mssg), auth(mssg))
```

```
    then
        K(P, ES) := key(decrypt(ticket(mssg), K(ES)))
        P := sender(mssg)
        clear(mssg)
        mode := ReadyToSend

  [ProvideHandshake rule]
    if mode = ReadyToSend
    then
        if RequiredMutualAuthentication(task(P))
        then
            ES_SendTo_P :
                encrypt({ts(decrypt(auth(mssg), K(P,ES))) + 1}, K(P,ES))
        TimeStamp(P) :=
            append(ts(decrypt(auth(mssg), K(P,ES))), TimeStamp(P))
        clear(P)
        mode := ReadyToReceive

endblock
```

3.6 The *MultipleEndServers* Model

Consider the complete system with a certain number of end servers, each providing a different network service [Fig.1]. We create an agent for each end server, and define a universe *SERVER* of end servers to extend the universe *DAEMON* and therefore *AGENT*.

To address the end server that provides one particular service, we define the function

$$server : SERVICE \longrightarrow SERVER$$

where *SERVICE* has been extended to the whole set of network services.

Those operations concerning the binding to a client performed by each agent in *SERVER* may be formalised by the ES_MODULE, so that we define $Mod(S)$ = ES_MODULE for all $S \in SERVER$. All of the conditions stated till now about *ES* must be extended to all elements of *SERVER*.

Therefore, the complete specification comes out from the program in [Section 3.5.4] after the following modifications:

1. in SP_MODULE replace each occurrence of *ES* by $server(task(P))$;
2. in TGS_MODULE add the update '$ES := EndServer(mssg)$' to rule **CheckAuthentication** and the update '$clear(ES)$' (below $clear(P)$) to rule **ProvideAuthorisation**.

The function $EndServer(mssg)$: *SERVICE* yields the projection on the third field of the record *mssg* received as message from a (service) process.

As with servers *KAS* and *TGS*, we make no distinction between the host end server and its daemon process. As a consequence, by $server(service)$ we mean either the end server identifier or its daemon process, that gives sense to update (1).

Given an end server *ES*, we denote by $\mathcal{A}(ES)$ the restriction of the current ASM with $SERVER = \{ES\}$.

Theorem 6 (Refinement) *The runs in $\mathcal{A}(ES)$ and in the 'MultipleClients' model are in one-to-one correspondence.*

It is straightforward to verify that the ES_MODULE at both levels model the same agent behaviour. □

Remark. Theorem 6 shows that the 'MultipleEndServers' model is a generalization of the 'MultipleClients model'.

As a Corollary of Theorem 6, the Main Theorem 5 stated in [Section 3.5.2] holds in the 'MultipleEndServers' model. The proof will consider regular runs involving the agent $server(task(P))$ such that $owner(P) = U$ & $Mod(P) =$ SP_MODULE in the role of *ES*.

4 External Threats

This section takes into account the threats coming from the environment in which the Kerberos Authentication System has to work, i.e. a modern network of computers.

Today a computer network must be considered a hostile environment in the sense that eavesdroppers are able to listen to the connection between two parties, thus stealing valuable information. In this context, we use the term *spy* to refer to a network user whose target is accessing a network resource without using her identity, i.e. user identifier and password. For a complete treatment of the concept of spy, see [Voydock, Kent 83].

4.1 The spy

To specify a spy's operation, first of all one has to answer the following questions. How does a spy work? What can a spy learn by observing the network traffic? How can a spy wish to get services she is not allowed to?

Broadly speaking, the problem of how to model a spy is not trivial at all. Many have tried to formalise an even more powerful spy than she could ever be in reality, in order to have better assurances that the protocol may be considered safe in practice. This approach might work well, but could also, e.g., cause overloading problems to state-enumeration methods (as pointed out in [Lowe 96a, Paulson 96]), or prove to be unplausible. On the other hand, many have taken into account too weak, unrealistic attackers. A tradeoff is needed.

The following capabilities seem the most reasonable [Bellovin, Merritt 90]. A spy can **control all of the traffic** over the network (which translates into the total access to the universe *MESSAGE*); **modify her workstation address** in the form of that where the user is logged in; **exploit the accidental loss** of a session key with *TGS* (model 1), or of a session key with an end server (model 2). Thus, we analyse two different models of a spy, which are given as *non-destructive* spies, i.e. they do not alter the messages sent on the network but simply steal the information they contain. In fact, considering *destructive* spies would simply make the description longer – the atomic transmission of a message would become the task of a new agent formalising the message-passing interface – without adding to the analysis of the spy's potentialities.

Remark. There is no need to analyse a spy who is able to intercept the messages sent on the network but unable to steal any session key. Such a spy would never gain any service because, should she reply intercepted tickets or authenticators, she could not handle the servers' encrypted reply.

We formalise the spy as a new element SPY of $AGENT$ and we extend on SPY all of the functions defined for elements of $PROCESS$. We impose $owner(SPY) = SPY$ so that SPY is seen as a service process running on behalf of the spy.

The stolen session key is denoted by K_{stolen}: KEY which is a monitored function, since we are not concerned about how SPY gets hold of it. She might exploit an accidental loss by a legitimate user, or even pay money for it, or something else.

SPY starts to operate as soon as she intercepts a message that she can decrypt by K_{stolen}. The monitored function $mssg_{\mathsf{stolen}}$: $MESSAGE$ yields the message containing an authenticator that can be decrypted by K_{stolen}. This function formalises the interception of a message by SPY. We impose the following *integrity constraints*:

C1. if $defined(K_{\mathsf{stolen}})$, then $K_{\mathsf{stolen}} = K(P,S)$ for some $P \in PROCESS$ and $S \in \{TGS\} \cup SERVER$;
C2. if $defined(K_{\mathsf{stolen}})$ & $defined(mssg_{\mathsf{stolen}})$ & $K_{\mathsf{stolen}} = K(P,S)$ for some $P \in PROCESS$ and $S \in \{TGS\} \cup SERVER$, then $sender(mssg_{\mathsf{stolen}}) = P$ & $receiver(mssg_{\mathsf{stolen}}) = S$ & $defined(decrypt(auth(mssg_{\mathsf{stolen}}), K_{\mathsf{stolen}}))$;
C3. if $defined(K_{\mathsf{stolen}})$ & $K_{\mathsf{stolen}} = K(P,S)$ for some $P \in PROCESS$ and $S \in \{TGS\} \cup SERVER$, then $K(SPY,S) = K_{\mathsf{stolen}}$;
C4. if $defined(K_{\mathsf{stolen}})$ & $K_{\mathsf{stolen}} = K(P,S)$ for some $P \in PROCESS$ & $S \in SERVER$, then $server(task(SPY)) = server(task(P))$.

In the sequel we abbreviate $server(task(SPY))$ by ES.

The two models sketched above are specified by the new modules SPY_MODULE1 and SPY_MODULE2 respectively. They must be added, in turn, to the program of the *MultipleEndServers* model.

4.1.1 The first model of spy

This spy is able to intercept the messages sent on the network and to exploit a stolen session key with TGS. Thus, by constraint C3 we have $K(SPY,TGS) = K_{\mathsf{stolen}} = K(P,TGS)$, being P the service process to which the session key was issued.

This is obviously the most powerful model, since that kind of session key has a lifetime of hours and is used every time a user wishes to access a service.

To model this agent, we define $Mod(SPY)$ = SPY_MODULE1 [see Fig. 4]. The following rule **RequireAuthorisation** describes how SPY impersonates an (authenticated) service process during the request for authorisation to TGS.

SPY controls the connection between the worstation and TGS. When she gets a message she can decrypt using the stolen key, she extracts and saves its components. Note that the second update allows $authenticated(SPY) = true$ (**StealAuthentication** rule).

After that, SPY exploits the stolen key to define a faked authenticator by refreshing the current time, and sends TGS an apparently legal message (**FakeAuthentication** rule).

Note that the existence of two authenticators that differ only in the timestamps is perfectly admissible because they could have been issued by the same

```
                SPY_MODULE1

RequireAuthorisation
≡ block
      StealAuthentication
      FakeAuthentication
  endblock
NegotiateService
≡ block
      GetAuthorisation
      ShowAuthorisation
  endblock
MutualAuthentication
RevokeAuthentication
RevokeAuthorisation
```

Figure 4: The program for the first model of SPY

user at different times. Therefore, they will pass the control on TGS.
RequireAuthorisation rule of SPY_MODULE1

```
block

    [StealAuthentication rule]
        if mode = ReadyToReceive & defined(K(SPY,TGS))
           & defined(mssg_stolen)
        then
           Auth(SPY, K(SPY,TGS)) := auth(mssg_stolen)
           Ticket(SPY,TGS) := ticket(mssg_stolen)
           clear(mssg_stolen)
           mode := ReadyToSend

    [FakeAuthentication rule]
        if mode = ReadyToSend
        then
           if defined(FakedAuth)
           then
              SPY_SendTo_TGS : {Ticket(SPY,TGS),FakedAuth,server(task(SPY))}
              mark(SPY) := ⟨server(task(SPY)), CT⟩
              mode = ReadyToReceive
           else
              FakedAuth :=
                encrypt({client(StolenAuth),address(StolenAuth),CT}, K(SPY,TGS))
        where StolenAuth = decrypt(Auth(SPY, K(SPY,TGS)), K(SPY,TGS))

endblock
```

Rules **NegotiateService**, **MutualAuthentication**, **RevokeAuthorisation** and **RevokeAuthentication** of SP_MODULE rewritten with P updated to SPY complete the module.

Incidentally, it has to be observed that the proof of correctness Corollary 1 and Theorem 5 are not affected from the extension by SPY, because SPY_MO-

DULE1 is an actual non-destructive program (the same will apply to SPY_MODULE2). In fact its only interaction with the external environment is by the macro *SendTo* which simply extends the universe *MESSAGE* without altering its previous contents.

To analyse the spy's potentialities, it is enough to restrict our attention to run involving the agents SPY, TGS and ES. Let $\mathcal{A}(SPY)$ be the restriction of the whole ASM $\mathcal{A}$ to those three agents. Note that the agents of $\mathcal{A}(SPY)$ can fire one rule at a time, thus we simply consider *sequential run*.

Definition 5. A (*sequential*) *run of* SPY is a sequence $\varrho_{_{SPY}} = S_0, S_1, \ldots, S_k, \ldots$ of states of $\mathcal{A}(SPY)$ such that, for each positive k, S^-_{k+1} is obtained from S_k by executing one rule of $\mathcal{A}(SPY)$ at S_k.

The working condition W3 can be stated for SPY (replacing P by SPY) in $\mathcal{A}(SPY)$ because it concerns the right operation of the servers.

Definition 6. A run $\varrho_{_{SPY}}$ of SPY is called *regular* if the following conditions hold:

(i) the initial state conditions are:
- $mode(Self) =$ ReadyToReceive, $Self \in \{SPY, TGS, ES\}$;
- $defined(K_{\mathsf{stolen}})$ and $K_{\mathsf{stolen}} = K(P, TGS)$, for some $P \in PROCESS$;
- $address(SPY) = address(P)$ such that $K_{\mathsf{stolen}} = K(P, TGS)$, $P \in PROCESS$;
- $task(SPY) \neq undef$;
- $sender(X) \neq SPY \wedge receiver(X) \neq SPY$, $\forall X \in MESSAGE$;
- $defined(mssg_{\mathsf{stolen}})$ and $defined(decrypt(auth(mssg_{\mathsf{stolen}}),\ K_{\mathsf{stolen}}))$.

(ii) working conditions W1 and W3 hold at any state of $\varrho_{_{SPY}}$.

When SPY intercepts a message containing the authenticator she can decrypt by $\mathrm{K}_{\mathsf{stolen}}$ and a ticket still fresh – i.e. $\mathrm{K}_{\mathsf{stolen}}$ is still fresh – if she is quick enough to forward the ticket to TGS and to fake the authenticator, she will get the authorisation credentials from TGS. This is formalised by the following

Theorem 7 *Let $\varrho_{_{SPY}}$ be a regular run of SPY. Let t be the timestamp of the stolen ticket and t' the timestamp of the faked authenticator. If $[CT]_{S_0} < t + AuthLife$, and TGS receives a message from SPY at some state S_i, $i > 0$, such that $[CT]_{S_i} < min(t + AuthLife, t' + DeltaLife)$, then*
(i) *authorised(SPY) holds at some state;*
(ii) *if the authorised SPY satisfies the working condition W4 with the required end server, then she shares a session key with that end server and is* ReadyToStart *the communication with it.*

Proof. To prove the part (i) we have to show that SPY is able to get from TGS a service ticket and a service key; after that the proof of (ii) trivially follows the same pattern as in the part (ii) of the Corollary 1 – replacing each rule in SP_MODULE by the corresponding one in SPY_MODULE1 – starting from the application of rule **ShowAuthorisation**.

By initial conditions the guards of rule **StealAuthentication** hold in S_0, so that SPY stores authenticator and ticket contained in the stolen message. Having learnt from it, by rule **FakeAuthentication** he fakes the stolen authenticator using K_{stolen}, and sends TGS a message of the expected form. In the current state, the guards of rule **TgsReply** holds because macro *SuccValid* checks out by hypothesis on $[CT]_{S_i}$ and constraint about the function *address*. Hence, TGS builds a service ticket and a service key for SPY that it considers a legitimate, authenticated process. SPY gets hold of them by rule **GetAuthorisation**. □

4.1.2 The second model of spy

In addition controlling the network, this spy owns a stolen session key with an end server. By constraint C3 we have $K(SPY,ES) = K_{\mathsf{stolen}} = K(P,ES)$ and, by C4, $ES = server(task(P))$, being P the process to which the session key was issued. We define $Mod(SPY)$ = SPY_MODULE2. Once SPY has stolen a

```
                SPY_MODULE2

RequireAuthorisation
≡ block
      StealAuthentication
      FakeAuthentication
  endblock
MutualAuthentication
RevokeAuthorisation
```

Figure 5: The program for the second model of SPY

message containing the service ticket and the service authenticator that she can decrypt, she can gain the service for which the session key was issued only if she is able to fake the authenticator within the validity time of the ticket. Although this time is short, the target is still reasonable because the check within a large number of messages to find the one that can be decrypted by a given key, may be real-time computed.

Once again, a good way to fake the authenticator is updating the timestamp thanks to the stolen session key, and then sending ES a message of the expected form. This is described in the following rule.

RequireAuthorisation rule of SPY_MODULE2

```
block

   [StealAuthentication rule]
      if mode = ReadyToReceive & defined(K(SPY,ES))
         & defined(mssg_stolen)
      then
         Auth(SPY, K(SPY,ES)) := auth(mssg_stolen)
         Ticket(SPY,ES) := ticket(mssg_stolen)
         clear(mssg_stolen)
         mode := ReadyToSend
```

```
[FakeAuthentication]
   if mode = ReadyToSend
   then
      if defined(FakedAuth)
      then
         SPY_SendTo_ES : {Ticket(SPY,ES), FakedAuth}
         mark(SPY) := ⟨CT⟩
         mode = ReadyToReceive
      else
         FakedAuth :=
            encrypt({client(StolenAuth), address(StolenAuth), CT}, K(SPY,ES))
   where StolenAuth = decrypt(Auth(SPY, K(SPY,ES)), K(SPY,ES))

endblock
```

Let now $\mathcal{A}(SPY)$ be the restriction of the whole ASM to the agents SPY and *ES*. The previous definition of a (*sequential*) *run* $\varrho_{_{SPY}}$ *of* SPY is still valid. The working condition W3 can be stated for SPY in $\mathcal{A}(SPY)$.

Definition 7. A run $\varrho_{_{SPY}}$ of SPY is called *regular* if the following conditions hold:

(i) the initial state conditions are:
  - $mode(Self) = \mathsf{ReadyToReceive}$, $Self \in \{SPY, ES\}$;
  - $defined(K_{\mathsf{stolen}})$ and $K_{\mathsf{stolen}} = K(P,ES)$, $P \in PROCESS$, and $ES = server(task(P))$;
  - $address(SPY) = address(P)$ such that $K_{\mathsf{stolen}} = K(P,ES)$, $P \in PROCESS$;
  - $sender(X) \neq SPY \wedge receiver(X) \neq SPY$, $\forall X \in MESSAGE$;
  - $defined(mssg_{\mathsf{stolen}})$ and $defined(decrypt(auth(mssg_{\mathsf{stolen}}), K_{\mathsf{stolen}}))$.

(ii) working conditions W1 and W3 hold at any state of $\varrho_{_{SPY}}$.

When SPY intercepts a message containing the authenticator that she can decrypt by K_{stolen} and a ticket still fresh – i.e. K_{stolen} is still fresh – if she is quick enough to forward the ticket to *ES* and to fake the authenticator, she will get the handshake from *ES*. This is formalised by the following

Theorem 8 *Let $\varrho_{_{SPY}}$ be a regular run of SPY. Let t be the timestamp of the stolen ticket and t' the timestamp of the faked authenticator. If $[CT]_{S_0} < t + ServLife$ and ES receives a message from SPY at some state S_i, $i > 0$, such that $[CT]_{S_i} < min(t + ServLife, t' + DeltaLife)$, then SPY can share a session key with ES and get mode* ReadyToStart *the communication.*

Proof. By initial conditions the guards of rule **StealAuthentication** hold in S_0, so that *SPY* stores authenticator and ticket from the stolen message, and by rule **FakeAuthentication** sends *ES* a message of the expected form, containing the faked authenticator. Rule **Handshake** fires to the current state because *SuccValid* checks out by the hypothesis on $[CT]_{S_i}$ and the condition on the function *address*, in such a way that *ES* starts the handshake with *SPY* if the mutual authentication is required or simply saves the session key. The possible application of the **MutualAuthentication** rule ends the proof. □

5 Conclusion

We have provided a complete analysis of the Kerberos Authentication System. The final ASM model has been reached through stepwise refinements. The first model directly reflects the guidelines of the Kerberos operation. The last one describes the complete system with each legitimate user able to require all the existing network resources.

Throughout this sequence of models, the ASM formalism has proved to be particularly suitable to the stepwise refinement strategy which usually makes it easier to understand complex systems, allowing to discover technical details step by step.

Our models have been used to discover the minimum assumptions to guarantee the *correctness* of the system. Each model ends with a refinement theorem which extends the correctness properties proved at the previous level. The last model may be extended in turn by two different models formalising the possible actions of an eavesdropper. This has shown some weaknesses of the system *security*.

Our proofs are traditional (not formalised) mathematical proofs and are not to be considered in opposition to machine-assisted proofs but as a possible guideline to be used within a specific proof system.

Acknowledgements. The authors are grateful to Egon Börger for his invaluable suggestions and constant encouragement. Rosario Gennaro pointed out the basic layout of Kerberos. Rebecca Marshall generously revised part of the paper. Roger Needham suggested few improvements.

The first author would also like to thank the *Fondazione Bonino-Pulejo* (Messina-ITALY) and *The British Council* (Rome-ITALY) for funding his Ph.D.

The second author has been partially supported by the INTAS project.

References

[Abadi, Needham 96] Abadi, M., Needham, R.: "Prudent engineering practice for cryptographic protocols"; IEEE Transactions on Software Engineering (1996), 22(1), 6-15.

[Anderson 95] Anderson, R.: "Why cryptosystems fail"; Communications of the ACM (1994), 37(11), 32-40.

[Bella, Paulson 97] Bella, G., Paulson, L.C.: "Using Isabelle to Prove Properties of the Kerberos Authentication System"; in Proc. DIMACS Workshop on Design and Formal Verification of Security Protocols (1997).

[Bellare, Rogaway 95] Bellare, M., Rogaway, P.: "Provably Secure Session Key Distribution – The Three Party Case"; Proc. STOCS 1995, ACM Press (1995), 57-66.

[Bellovin, Merritt 90] Bellovin, S.M., Merritt, M.: "Limitations of the Kerberos authentication system"; Computer Comm. Review (1990), 20(5), 119-132.

[Bolignano 96] Bolignano, D.: "An approach to the formal verification of cryptographic protocols"; in Third ACM Conference on Computer and Communication Security, ACM Press (1996), 106-118.

[Börger 95a] Börger, E.: "Specification and Validation Methods", Oxford University Press (1994).

[Börger 95b] Börger, E.: "Why Use Evolving Algebras for Hardware and Software Engineering?"; in M. Bartosek, J. Staudek, J. Wiedermann (Eds.): Proc. SOFSEM'95, Springer LNCS 1012 (1995), 236-271.

[Börger, Mearelli 97] Börger, E., Mearelli, L.: "Integrating ASMs into the Software Development Life Cycle"; in J.UCS (1997), 3(5), 603-665.

[Burrows et al. 90] Burrows, M., Abadi, M., Needham, R.: "A Logic for Authentication"; ACM Transaction on Computer Systems, 8,1(1990), 18-35.

[Denning, Sacco 81] Denning, D.E., Sacco, G.M.: "Timestamps in key distribution protocols"; Comm. of ACM (1981), 24(8), 533-536.

[Gurevich 95] Gurevich, Y.: "Evolving Algebras 1993: Lipari Guide"; in E. Börger (Ed.): Specification and Validation Methods, Oxford University Press (1995).

[Lowe 96a] Lowe, G.: "SPLICE\AS: A case study in using to detect errors in security protocols"; Technical Report, Oxford University Computing Laboratory (1996).

[Lowe 96b] Lowe, G.: "Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR"; in T. Margaria and B. Steffen (Eds.): Tools and Algorithms for the Construction and Analysis of Systems (1996), Second International Workshop, TACAS '96, LNCS 1055, 147-166.

[Miller et al. 89] Miller, S.P., Neuman, J.I., Schiller, J.I., Saltzer, J.H.: "Kerberos Authentication and Authorisation System"; Project Athena Techical Plan, Section E.2.1, MIT (1989), 1-36.

[Mitchell et al. 97] Mitchell, J.C., Mitchell, M., Stern, U.: "Automated Analysis of Cryptographic Protocols Using Murphi"; IEEE Symposium on Security and Privacy (1997), 141-151.

[Needham, Schroeder 78] Needham, R., Schroeder, M.: "Using encryption for authentication in large networks of computers"; Communication of the ACM, 21,12 (1978), 993-999.

[Neuman, Ts'o 94] Neuman, B.C., Ts'o, T: "Kerberos: An authentication service for computer network"; IEEE Communication Magazine, 32,9 (1994), 33-39.

[Paulson 96] Paulson, L.C.: "Proving properties of security protocols by induction"; Technical Report No.409, Cambridge University Computer Laboratory (1996).

[Schumann 97] Schumann, J.: "Automatic Verification of Cryptographic Protocols with Setheo", in Proc. CADE97 Workshop of Automated Theorem Proving in Software Engineering (1997).

[Stalling 95] Stalling, W.S.: "Network and Internetwork Security Principles and Practice", Prentice Hall (1995).

[Stevens 90] Stevens, W.R.: "UNIX Network Programming", PTR Prentice Hall (1990).

[Voydock, Kent 83] Voydock, V.L., Kent, S.T.: "Security mechanisms in high-level network protocols"; Computing Surveys 15 (1983), 135-171.