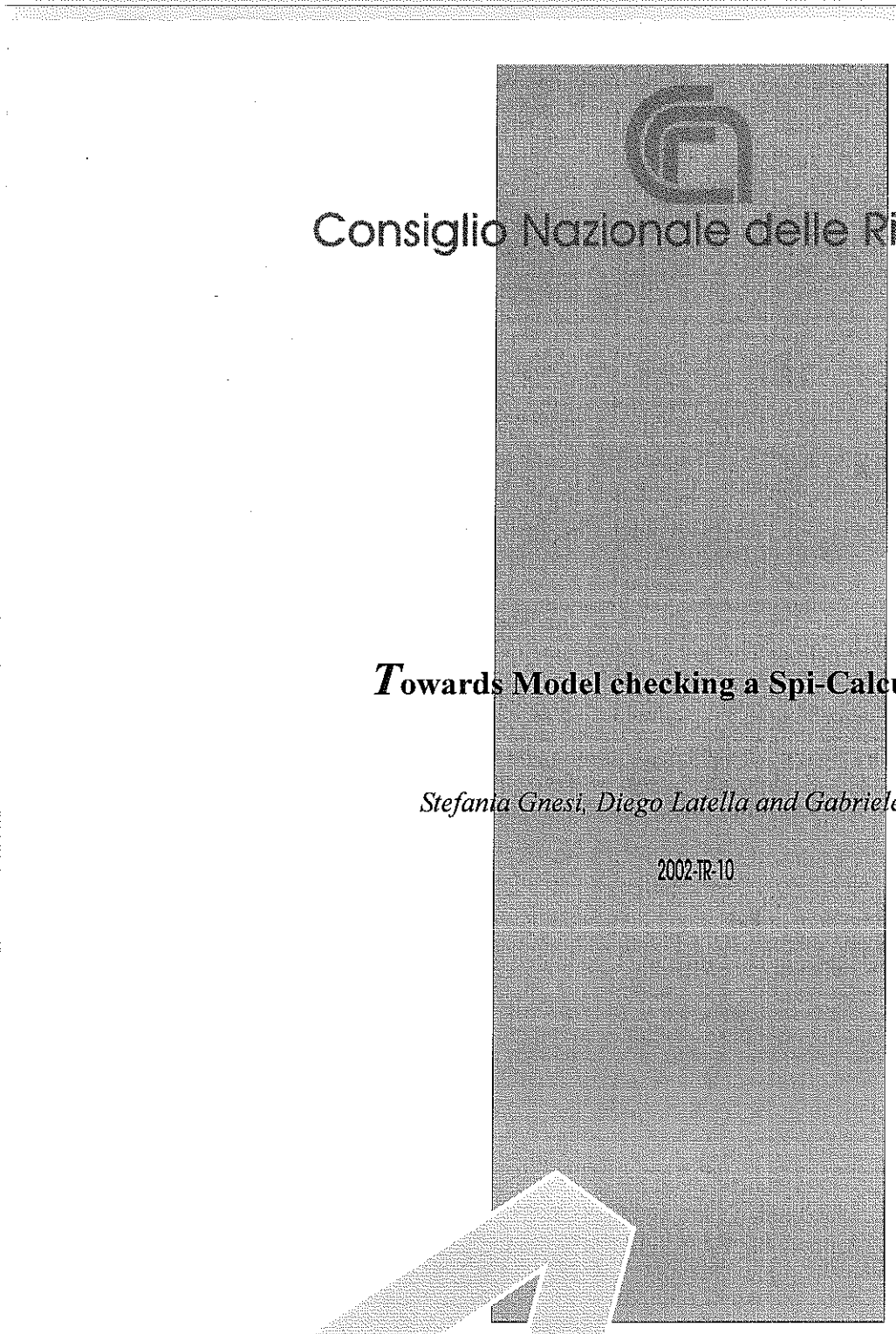
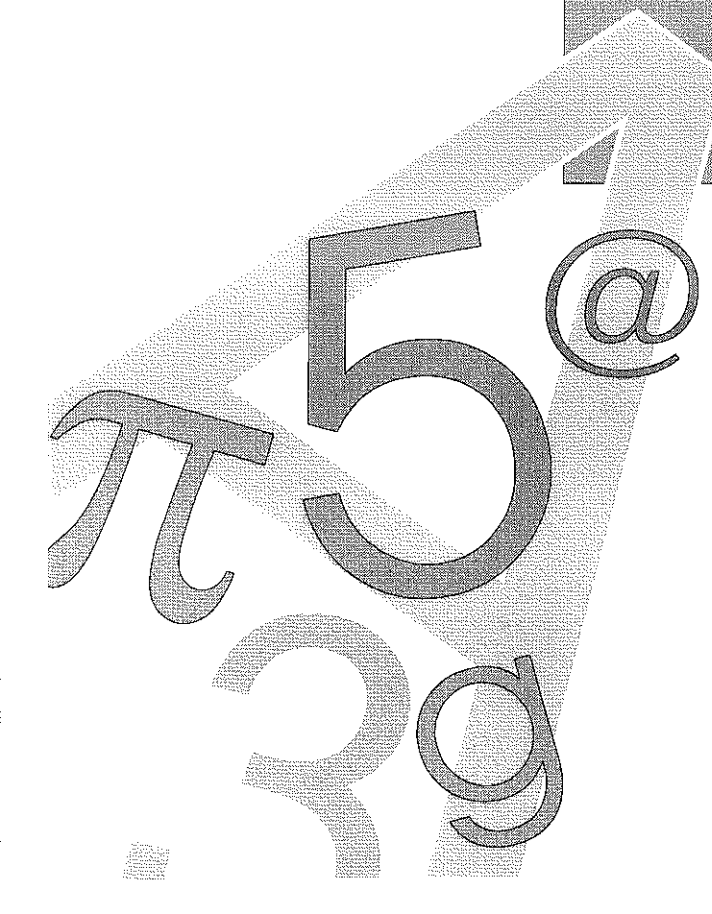


*T*owards Model checking a Spi-Calculus Dialect

Stefania Gnesi, Diego Latella and Gabriele Lenzini

2002-TR-10



Towards Model Checking a Spi-Calculus Dialect

S. Gnesi[1], D. Latella[2], and G. Lenzini[1]

[1] IEI-CNR, Pisa (Italy), ({gnesi,lenzini}@iei.pi.cnr.it)
[2] CNUCE-CNR, Pisa (Italy), (d.latella@cnuce.pi.cnr.it)

Abstract. In this paper we present a model checking framework for a spi-calculus dialect which uses a linear time temporal logic for expressing security properties. We have provided our spi-calculus dialect, called SPID, with a semantics based on labeled transition systems (LTS), where the intruder is modeled in the Dolev-Yao style as an active environment controlling the communication and able to compose new messages by pairing, splitting, encryption or decryption. The LTS coming from protocols specification, usually infinite state because of the intruder's capability to generate an infinite amount of messages, are the models over which the satisfiability of the logic formulae has been defined. As a logic we have used the one defined by Clarke, Jha and Marrero, for their BRUTUS model checker. We call it BRUTUS logic, which is known to be suitable to express a wide class of security properties, such as secrecy, integrity, authenticity, some weak form of anonymity and general safety properties. Although providing a satisfiability procedure over models coming from SPID protocols is the main contribution of this paper, we also have, as long term target, the development of an automatic verification tool. In this case we need finite models, and in this paper as a first solution we have limited, in the transition rules defining the transitions of our LTS, the length of messages the intruder can generate. Although simple, this strategy suffices for obtaining finite models using which it is possible to discover known attacks over protocols commonly used as test cases. [1]

Key words: security protocols, security properties, spi-calculus, temporal logics, model checking.

ACM Computing Classification: D.2.4 Software/Program Verification - F.3.1 Specifying and Verifying and Reasoning about Programs.

[1] This work has been supported by "Mefisto - Metodi Formali per la Sicurezza e il Tempo", Progetto MUST 2001, and by the CNR grant n. 201.02.60.

1 Introduction

Past experiences have shown how formal methods can be successfully applied to the analysis of security protocol problems (e.g., see [5, 24, 26, 1, 15]). In this paper we are interested in defining a verification framework for the *spi-calculus* [3], a process algebra derived from the π-calculus [19, 20] with operators to encrypt and decrypt messages. The spi-calculus syntax is expressive enough to describe all the shared-key cryptographic protocols and, with light modifications [3], also other cryptographic paradigms such as public-keys, hash functions and digital signatures.

One of the main attractive features of the spi-calculus is that secrecy and integrity properties can be naturally formalized [2] as may-testing equivalences [22]. Later symbolic trace analysis strategies [12, 4, 13] have been proposed to verify secrecy, integrity and authenticity properties on the spi calculus or on some dialects derived from it, and recently secrecy and authenticity have been reduced to type checking problems [1, 15] on a typed version of the spi-calculus.

Here we propose a *logic-based model checking* [6] approach to the verification of spi-calculus protocols, by developing a model checker for a spi-calculus dialect which uses a linear time temporal logic suitable for expressing a wide class of security properties. In the model checking the common strategy consists in formulating properties as predicates of a modal or temporal logic and trying to check their satisfiability on a finite model of the system, using algorithms called *model checkers*. In this papers the logic considered is the BRUTUS logic, a linear time temporal logic proposed by Clarke, Jha and Marrero in [7]. By using the BRUTUS logic it is possible to express not only secrecy, integrity and authenticity properties, but also some weak form of anonymity [7] and generic safety properties, and this makes it interesting to be use within a process algebras framework considering that in [7] no theoretical calculus is used to formalize protocols.

The model checking approach applied to spi-calculus is rather new, although model checking had been already applied in protocol security analysis (e.g., see [18, 16, 21, 25, 11, 9, 14]).

In Section 2 and Section 3 we present syntax and semantics of our spi calculus dialect, called SPID, and the BRUTUS logic. In particular we provide SPID with a semantics based on labeled transition systems. These transition systems will be the models on which the satisfiability of the logic is, in turn, defined. In Section 4 we explain, through an example, the use of the calculus and the BRUTUS logic in cryptographic protocol analysis. In Section 5 we describe a model checker algorithm, and some limitations to make finite the underlined model we have introduced. Finally in Section 6 we draw some conclusion.

2 SPID: a Spi-Calculus Dialect

In this section we describe the syntax of SPID, also synthesized in Figure 1. The language definition is based on: (1) a set $\mathcal{N}$ of *atomic names* which can

be agent's names, atomic messages, encryption keys, nounces etc., (2) a set $\mathcal{A}$ of *labels* used for distinguishing different communication actions, (3) a set $\mathcal{V}$ of *variables* and (4) a set $\mathcal{I}$ of *identifiers*[2], used for uniquely identifying process instances. We let m, k and p range over $\mathcal{N}$, a range over $\mathcal{A}$, x range over $\mathcal{V}$ and i range over $\mathcal{I}$.

Atomic names are used to build the set $\mathcal{M}$ of *messages* by pairing or encryption. The set $\mathcal{T}$ of *message terms* is built, again via pairing or encryption, from sets $\mathcal{N}$ and $\mathcal{V}$. We let M, N range over $\mathcal{M}$ and S, T range over $\mathcal{T}$.

2.1 SPID Syntax

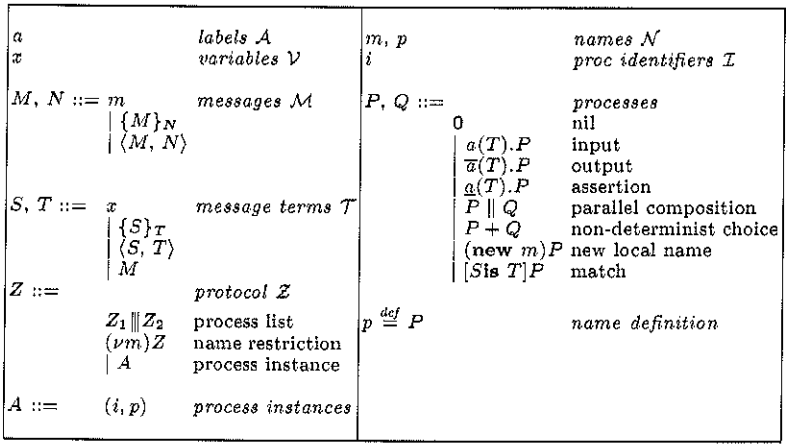
In our calculus a *protocol* Z is the parallel composition of n process instances $(1, p_1) \| \ldots \| (n, p_n)$, where the identifiers $1, \ldots, n$ are all distinct. The intruder is described implicitly, in the style of Dolev-Yao [10], as an *environment* having complete control of any communication channels: it can observe, intercept or exchange any message passing through the net, and by using the messages it knows, it can compose new messages. The restriction primitive (νm), is used for hiding a name m from what the environment initially knows.

A *process instance* (i, p), is composed by a name p, specifying a process, and by a unique identifier, i. More instances of the same process p, $(i_1, p), \ldots, (i_k, p)$, can be used to represent, as usual in security protocol analysis, a process running more sessions of the protocol.

A *process* can be: (1) $\mathbf{0}$, the process that does nothing; (2) $a(T).P$, the input process that is ready to perform an input action, labeled a, allowing a message M to be received from the environment via pattern matching with the message term T; (3) $\overline{a}(T).P'$, the process ready to perform an output action, labeled a, which sends to the environment the message M, obtained instantiating all the variables in T; (4) $\underline{a}(T).P'$, the process who is ready to perform an assertion action, labeled a, over the message M, instance of T. *Assertions*, presented the first time by Woo and Lam in [27], are used here to introduce begin-events and end-events useful for specifying protocol authenticity properties; (5) $P + Q$ and (6) $P \parallel Q$, which are respectively the non-deterministic choice and the parallel composition of processes P and Q; (7) (**new** m)P, which indicates the generation of a new name m then used within P; finally (8) $[S \textbf{is } T]P$, the process that performs an equality test over ground messages obtained instantiating the variables in S and T. If the test succeeds the process behaves as P else it stops.

In input action $a(T).P$ all variables appearing in T are bound in P, while a variable is free if it is not bound. Considering that input action is the only operator which binds variables, notions of bound and free variables can be defined over process instances and protocols in the usual way. From now on we assume to work on closed protocols, that is protocols with no free variables. Similarly in $(\nu m)Z$ and in (**new** m)P the name m is bound respectively in Z and in P, while a name is free if it is not bound.

[2] Without loss of generality, $\mathcal{I}$ is the set of natural numbers $\{1, 2, \ldots\}$.



a		*labels* $\mathcal{A}$	m, p		*names* $\mathcal{N}$
x		*variables* $\mathcal{V}$	i		*proc identifiers* $\mathcal{I}$
$M, N ::=$	m	*messages* $\mathcal{M}$	$P, Q ::=$		*processes*
	$\mid \{M\}_N$			0	nil
	$\mid \langle M, N \rangle$			$\mid a(T).P$	input
				$\mid \overline{a}(T).P$	output
$S, T ::=$	x	*message terms* $\mathcal{T}$		$\mid \underline{a}(T).P$	assertion
	$\mid \{S\}_T$			$\mid P \parallel Q$	parallel composition
	$\mid \langle S, T \rangle$			$\mid P + Q$	non-determinist choice
	$\mid M$			$\mid (\mathbf{new}\ m)P$	new local name
$Z ::=$		*protocol* $\mathcal{Z}$		$\mid [S\mathbf{is}\ T]P$	match
	$Z_1 \parallel Z_2$	process list	$p \stackrel{def}{=} P$		*name definition*
	$(\nu m)Z$	name restriction			
	$\mid A$	process instance			
$A ::=$	(i, p)	*process instances*			

Fig. 1. Syntax of the spi-calculus dialect.

As claimed, our calculus is a dialect of the spi-calculus and the most evident differences concern the communication paradigm:

(a) our calculus does not allow mobility. On the contrary there is one public channel and messages cannot be hidden to an external observer when transmitted.

(b) instead of a symmetric and synchronous communication paradigm, via shared public channels, our calculus uses asynchronous communications via the environment, which is a pool of messages that processes can freely access to. In this way we want to simulate any possible unpredictable intruder's attack. In fact when a process sends a message, the message is put into the environment, and when a process receives a message, the message is retrieved from the *knowledge* of the environment, that is from all the messages the environment can compose starting from the ones it has already got. Precisely, with knowledge or *synthesis* [23] we intend those messages that can be inferred using a proof system starting from an initial set of messages (axioms). Supposing the derivation symbols defined by the rules of the proof system was written $\vdash$, we have:

Definition 1 (Synthesis). *Let $W \subseteq \mathcal{M}$ be a finite set of messages. The* synthesis *of W, written $KS(W)$, is the set $W \cup \{M : W \vdash M\}$, that is the set of messages M for which there exists a proof whose premises are contained in W.*

In the following, when talking about $KS(W)$, we will refer to the proof system in Figure 2, whose rules represent the common operations over messages used in cryptographic protocol analysis. Moreover we have:

(c) our calculus does not allow unbound replication of processes (usually written as $!P$) usually used to describe multiple runs of a protocol. Here

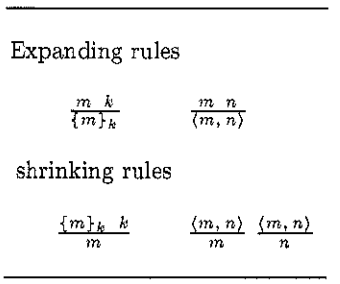


Fig. 2. Inference rules defining $\vdash$.

we assume to have a bound number of runs (e.g., as done in [25, 12, 4]), and multiple runs are described using multiple instances of each process. In addition in this paper we are interested only in acyclic (finite) processes, because this choice both suffices to describe agent behaviors running a security protocol, and avoids infinite problems due to recursive processes.

(d) constructs for pair splitting and for decryption are embedded into the input primitives. In this way the receiving of a messages and its successive analysis (splitting or decrypting) happen atomically in a single action. This choice is considered a syntactic sugar and simplifies part of the theory.

2.2 SPID Semantics

The operational semantics of our calculus is based on labeled transition systems, where transitions are defined at *two* level: an outer and an inner level.

Outer transitions (see Figure 3) have form $\langle \mathcal{G},\ Z\rangle \xrightarrow{\alpha} \langle \mathcal{G}',\ Z'\rangle$ and they describe how a protocol Z, and the global state $\mathcal{G}$, change.

Inner transitions (see Figure 4) have form $\langle \mathcal{G},\ (i,P)\rangle \overset{\alpha}{\rightsquigarrow} \langle \mathcal{G}',\ (i,P')\rangle$ and they describe how a process instance (i,P), and the global state $\mathcal{G}$, evolve as a consequence of the action α.

An action α, can be (a) an input action $i.a(M)$, indicating that process instance i has received the message M performing an action labeled a, (b) an output actions $i.\overline{a}(M)$, saying that instance i has sent the message M with action labeled a, (c) $i.\underline{a}\langle M\rangle$ showing that instance i has executed an assertion labeled a over the message M and (d) τ an internal action. We write Act to indicate the set of all actions.

A *global state* $\mathcal{G}$ is a composition of the local states of process instances and the local state of the environment. Formally, supposing that at most n process instances are involved in a protocol, the global state can be thought as an array of $n+1$ local states, where $\mathcal{G}(0)$ is the local state of the environment, and $\mathcal{G}(i)$ the local state of process instance identified by i. We write *Glob* to indicate the

set of all global states. In turn, a *local state* l, is triple (p, W, σ) where $p \in \mathcal{N}$ is a name, $W \subseteq \mathcal{M}$ is a set of messages and $\sigma : \mathcal{V} \longrightarrow \mathcal{M}$ is a function which binds variables to messages. It is worth to underline that W and σ have a monotone grown rate along the protocol execution, and this implies that when a variable is bound to a ground message as a consequence of an input actions, that variable remains bound afterwards. In other words a variable is indeed bound to the outermost input action instantiates it.

Talking about *binding functions*, with the symbol $\perp$ we indicate the function undefined everywhere, while we write $\sigma' \sqsupseteq \sigma$ whenever the function σ' coincides with σ in every values of the domain where σ is defined. Moreover we write $\sigma(x) = \sigma'(x)$ if both functions are undefined over x or coincide in x, and finally with $\widehat{\sigma}$ we indicate the obvious extension over message terms, where $\widehat{\sigma}(T)$ is obtained substituting each variable x in T with the corresponding value $\sigma(x)$. The test $\widehat{\sigma}(T) = \widehat{\sigma}(S)$ evaluates true if and only if both functions return the same ground message, false otherwise. Talking about a local state l, we indicate with $\mathsf{Name}(l)$, $\mathsf{Know}(l)$ and $\mathsf{Bind}(l)$ respectively the first component p, the second component W, and the third component σ of l.

In Definition 2 and in Figures 3 and 4 we formalize the whole operational semantics.

Definition 2 (Operational Semantics).
Let $Z = (1, p_1) \| \ldots \| (n, p_n)$, be a protocol. The associated labeled transition system is a tuple $(\mathcal{Q}_Z, \langle \mathcal{G}_0, Z_0 \rangle, Act, \mathcal{R}_Z)$, where:

- *$\mathcal{Q}_Z \subseteq Glob \times \mathcal{Z}$ is the set of states;*
- *$\langle \mathcal{G}_0, Z_0 \rangle$ is the initial state so defined*
  - *$Z_0 = Z$;*
  - *$\mathcal{G}_0(0) = (\Omega, W_\Omega, \perp)$, is the initial local state of the environment, where Ω is the name of the environment, W_Ω is the initial set of messages known by the environment composed by all the free names in Z, and $\perp$ is the empty set of bindings of the environment;*
  - *$\mathcal{G}_0(i) = (p_i, W_i, \perp)$ for all $i \in \mathcal{I}$, is the initial local state of the process instance i. More precisely, p_i is the name of the process P if $p_i \stackrel{def}{=} P$, W_i is the set of messages it knows, composed by all the free names in Z and by all the free names in Z and in P_i, and $\perp$ is its initial set of bindings.*
- *Act is the set of actions.*
- *$\mathcal{R}_Z \subseteq \mathcal{Q}_Z \times Act \times \mathcal{Q}_Z$ is the transition relation defined by the rules in Figure 3 and 4. Whenever $(q, \alpha, q') \in \mathcal{R}_Z$ we write $q \stackrel{\alpha}{\longmapsto} q'$.*

We briefly explain the transition rules reported in Figures 3 and Figure 4. Rules (P-PAR1) and (P-PAR2) define transitions for the parallel composition of process instances. The (P-AGE1) and (P-AGE2) rules make a single process transition rise up at protocol transition level. The (P-RES) just consumes the restriction. Anyway restriction plays it main role in the definition of the initial knowledge of agents of the intruder (see Definition 2). The (A-NEW) rule is

$$\text{(P-PAR1)}\ \frac{\langle \mathcal{G},\ Z_1\rangle \overset{\alpha}{\longmapsto} \langle \mathcal{G}',\ Z_1'\rangle}{\langle \mathcal{G},\ Z_1 \| Z_2\rangle \overset{\alpha}{\longmapsto} \langle \mathcal{G}',\ Z_1' \| Z_2\rangle} \qquad \text{(P-PAR2)}\ \frac{\langle \mathcal{G},\ Z_2\rangle \overset{\alpha}{\longmapsto} \langle \mathcal{G}',\ Z_2'\rangle}{\langle \mathcal{G},\ Z_1 \| Z_2\rangle \overset{\alpha}{\longmapsto} \langle \mathcal{G}',\ Z_1 \| Z_2'\rangle}$$

$$\text{(P-AGE1)}\ \frac{\langle \mathcal{G},\ (i,P)\rangle \overset{\alpha}{\Rightarrow} \langle \mathcal{G}',\ (i,P')\rangle}{\langle \mathcal{G},\ (i,P)\rangle \overset{\alpha}{\longmapsto} \langle \mathcal{G}',\ (i,P')\rangle} \qquad \text{(P-RES)}\ \frac{\langle \mathcal{G},\ Z\rangle \overset{\alpha}{\longmapsto} \langle \mathcal{G}',\ Z'\rangle}{\langle \mathcal{G},\ (\nu m)Z\rangle \overset{\alpha}{\longmapsto} \langle \mathcal{G}',\ Z'\rangle}$$

$$\text{(P-AGE2)}\ \frac{\langle \mathcal{G},\ (i,P)\rangle \overset{\alpha}{\Rightarrow} \langle \mathcal{G}',\ (i,P')\rangle}{\langle \mathcal{G},\ (i,p)\rangle \overset{\alpha}{\longmapsto} \langle \mathcal{G}',\ (i,P')\rangle}$$

where $p \overset{def}{=} P$

Fig. 3. Transition rules for protocols

used to define a new local name. Here $P[n/m]$ indicate the process obtained substituting, in P, all the free occurrences of name m with name n. The (A-INP) transition describes what happens when a process instance (i, P) is ready to perform an input action. Before going on we need the following definition, involving message terms:

Definition 3 (Honest Message Terms). *Supposing the following predicate $\mathcal{P}$, defined over message terms*

$$\mathcal{P}(T) = \begin{cases} \texttt{true} & \text{if } T = M \in \mathcal{M} \\ \texttt{true} & \text{if } T = x \in \mathcal{V} \\ \mathcal{P}(T') & \text{if } T = \{T'\}_M \\ \mathcal{P}(T_1) \wedge \mathcal{P}(T_2) & \text{if } T = \langle T_1, T_2\rangle \\ \texttt{false} & \text{otherwise .} \end{cases}$$

a message term T is honest *if and only if $\mathcal{P}(T)$ evals true.*

Informally an honest message terms has no variables in key position, when the term contains encrypted subterms. In fact we check if σ, the local binding function, can be extended with a $\sigma' \sqsupseteq \sigma$ such that the input term $\widehat{\sigma'}(T)$ is a message $M \in KS(W_\Omega)$, and in this case message M is added to the set of messages locally known by the process instance (i, P), while the local binding σ is substituted with σ'. If $\mathcal{P}(\widehat{\sigma}(T))$ was not true, an encrypted messages (e.g., $\{M\}_K$) could be decrypted, via pattern matching with a term (e.g., $\{x\}_y$) without knowing the decryption key.

In addition it is worth to underline that, although $KS(W_\Omega)$ is generally infinite, to check whether $\widehat{\sigma'}(T) \in KS(W_\Omega)$ is decidable [17]. Then the premises of (A-INP) are decidable, even though the number of possible extentions of σ is generally infinite, which produces infinite branching in absence of limiting strategies. The (A-OUT) rule describes what happens when a process instance is ready to perform an output action. The output sends the message $M = \widehat{\sigma}(T)$, that becomes part of the set of messages W_Ω known by the environment. Note that if Z is a closed protocol, then $\widehat{\sigma}(T)$ is necessarily a ground message. Finally (A-ASS) rule represents transitions related to a process assertion. The action depicting the assertion is visible in the transition system, but no message is transmitted reflecting the fact that assertions are not communication actions.

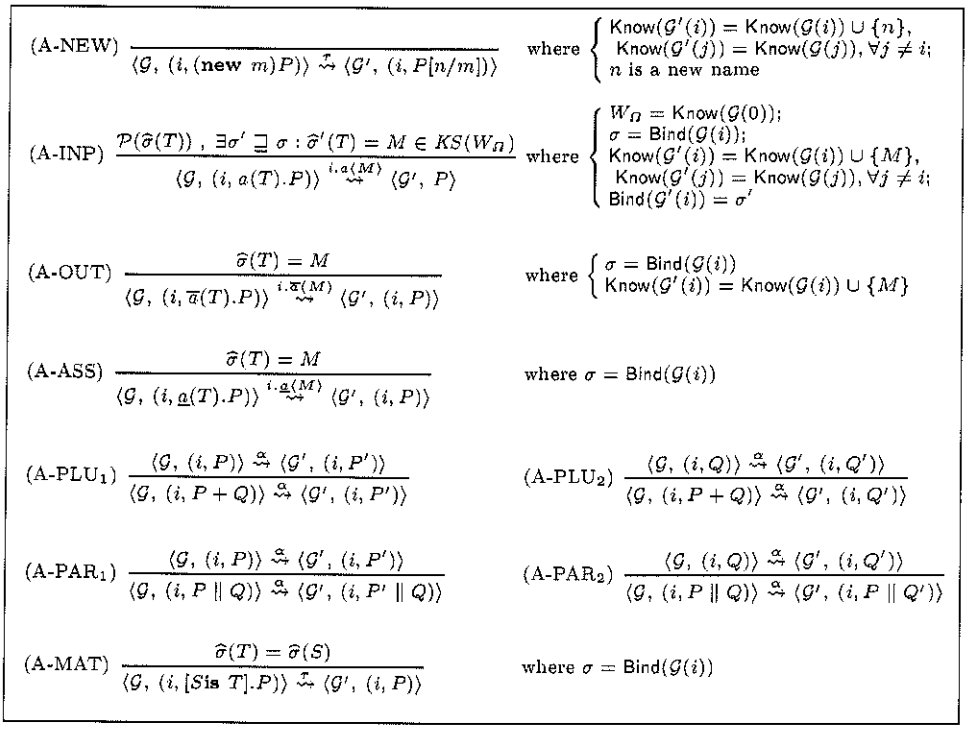


Fig. 4. Transition rules for process instances.

In the following we write LTS_∞ to indicate the class of labeled transition systems defined by the rules in Figure 3 and Figure 4 and coming from closed protocols.

3 BRUTUS Logic

The syntax of BRUTUS logic [8], is presented in Figure 5. Alphabet symbols are the same of the calculus syntax: (1) a set $\mathcal{M}$ of *messages*, (2) a set $\mathcal{A}$ of *labels*, (3) a set $\mathcal{V}$ of *variables* and (4) a set $\mathcal{I}$ of *identifiers*. In addition we have (5) a set $\mathcal{V}_s$ of *process instance variables*.

Alphabet symbols are used to build process instance terms $\mathcal{T}_s$, message terms $\mathcal{T}_m$, terms, atomic propositions and formulae that constitute the syntax of the BRUTUS logic. We let s range over $\mathcal{V}_s$, pt over $\mathcal{T}_s$ and mt range over $\mathcal{T}_m$.

A *term* t, can be: (a) $\mathbf{name}(pt)$, the name of the process instance pt, (b) $pt.mt$ the message term mt interpreted in the local state of the process instance pt or (c) a ground message $M \in \mathcal{M}$.

An *atomic propositions* ρ, is: (a) $\mathbf{Knows}(pt,t)$, which is a predicate on the knowledge of the process instance pt of the term t; (b) $\mathbf{Acts}_\lambda(pt,t)$ which is a

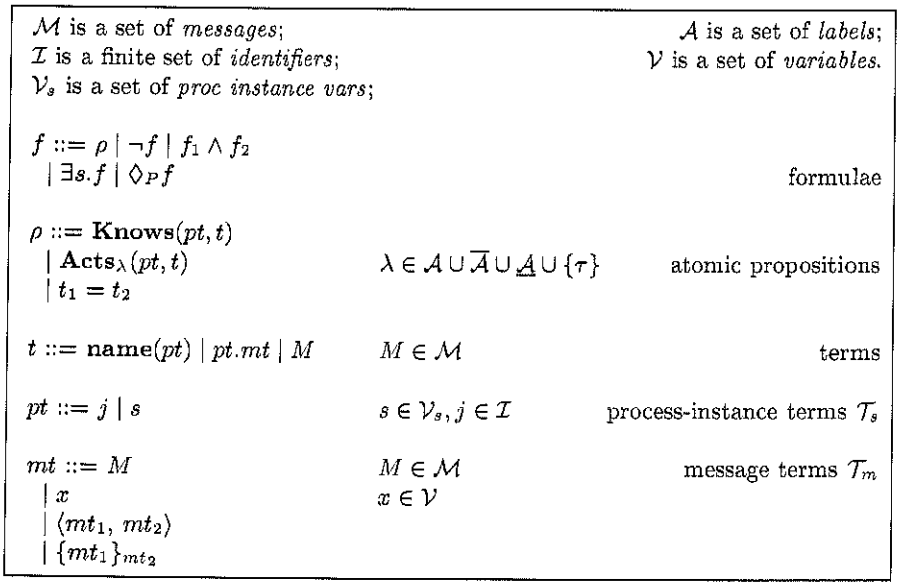


Fig. 5. The Syntax of the BRUTUS Logic

predicate on the action λ, performed by pt, involving the term t; (c) $t_1 = t_2$, which is an equality test over terms.

A *formula* f, can be any propositional logic formula, or the modal formula $\Diamond_P f$, where the symbol $\Diamond_P$ is the modal operator *eventually* in its past interpretation. The formula $\exists s.f$ binds the process-instance variable s within the formula f.

The interpretation of atomic propositions and formulae is defined over *traces* $\pi = q_0 \cdot \alpha_1 \cdot q_1 \cdot \ldots \cdot \alpha_n \cdot q_n$ coming from a labeled transition system belonging to LTS_∞, where $q_i = \langle \mathcal{G}_i, Z_i \rangle$, and $q_i \overset{\alpha_{i+1}}{\longmapsto} q_{i+1}$ is a possible transition from q_i. In other words a trace is the temporal structure over which the satisfiability of a formula is checked. We start defining the interpretation of terms over a single global state $\mathcal{G}$.

Definition 4 (Message Term Interpretation). *Given a global state $\mathcal{G}$, and a message term pt, the* message term interpretation, *is the function*[3] $\mathbb{M} : Glob \rightarrow \mathcal{T}_m \rightarrow \mathcal{M} \cup \{\bot\}$ *given below:*

$$\begin{array}{lll} \mathbb{M}(\mathcal{G})(M) & = M, & \textit{where } M \in \mathcal{M} \\ \mathbb{M}(\mathcal{G})(j.mt) & = \widehat{\sigma}(mt), & \textit{where } \sigma = \mathsf{Bind}(\mathcal{G}(j)) \\ \mathbb{M}(\mathcal{G})(\mathbf{name}(j)) & = p, & \textit{where } p = \mathsf{Name}(\mathcal{G}(j)) \end{array}$$

Informally, the interpretation of $j.mt$ is the ground message obtained instantiating all the variables appearing in mt, using the set of bindings of the process

[3] With *Glob* we intend the set of all global states.

instance whose identifier is j. If not all the variables in mt can be instantiated then $j.mt$ is undefined; the interpretation of $\mathbf{name}(j)$ is the message which represents the name of the process instance whose identifier is j.

Now we define the interpretation of atomic propositions and formulae over a single state. We will write $\langle \pi,\ i\rangle \models \rho$ to mean that the atomic proposition ρ is satisfied on the state $q_i = \langle \mathcal{G}_i,\ Z_i\rangle$ of the trace π. We will write $\langle \pi,\ i\rangle \models f$ to say that formula f is satisfied on the state q_i.

Definition 5 (Atomic Proposition Interpretation). *Given a trace $\pi = q_0 \cdot \alpha_1 \cdot q_1 \cdot \ldots \cdot \alpha_n \cdot q_n$, we have, for $1 \leq i \leq n$:*

$$\begin{array}{ll}
\langle \pi,\ q_i\rangle \models t_1 = t_2 & \textit{iff } \mathbb{M}(\mathcal{G}_i)(t_1) = \mathbb{M}(\mathcal{G}_i)(t_2) \\
\langle \pi,\ q_i\rangle \models \mathbf{Knows}(j,t) & \textit{iff } \mathbb{M}(\mathcal{G}_i)(t) \in KS(\mathsf{Know}(\mathcal{G}_i(j))) \\
\langle \pi,\ q_i\rangle \models \mathbf{Acts}_\lambda(j,t) & \textit{iff } \alpha_i = j.\lambda(M),\ \textit{where } M = \mathbb{M}(\mathcal{G}_i)(t)
\end{array}$$

Informally, the proposition $t_1 = t_2$ is true over the global state $\mathcal{G}_i$, where $q_i = \langle \mathcal{G}_i,\ Z_i\rangle$, if the interpretation of the two message terms over $\mathcal{G}_i$, is equal. We remind from Section 2 that if at least one of the interpretation is undefined the equality results false. The proposition $\mathbf{Knows}(j,t)$ is true, over the global state $\mathcal{G}_i$, where $q_i = \langle \mathcal{G}_i,\ Z_i\rangle$, if and only if the interpretation of the term t over a state $\mathcal{G}_i$, belongs to $KS(W_j)$, where W_j is the set of messages in the local state of the process identified with j. Finally the proposition $\mathbf{Acts}_\lambda(j,t)$ is true, over the global state $\mathcal{G}_i$, where $q_i = \langle \mathcal{G}_i,\ Z_i\rangle$, if and only if the action α_i, the transition taken to enter in the current state, is $j.\lambda(M)$ where M is the message obtained interpreting the term t over $\mathcal{G}_i$.

Definition 6 (Formulae Interpretation). *Given a formula f and a trace $\pi = q_0 \cdot \alpha_1 \cdot \ldots \cdot \alpha_n \cdot q_n$, we have that for $1 \leq i \leq n$:*

$$\begin{array}{ll}
\langle \pi,\ q_i\rangle \models \rho & \textit{iff } \langle \pi,\ q_i\rangle \models \rho \\
\langle \pi,\ q_i\rangle \models \neg f & \textit{iff } \langle \pi,\ q_i\rangle \not\models f \\
\langle \pi,\ q_i\rangle \models f_1 \wedge f_2 & \textit{iff } \langle \pi,\ q_i\rangle \models f_1 \textit{ and } \langle \pi,\ i\rangle \models f_2 \\
\langle \pi,\ q_i\rangle \models \exists s.f & \textit{iff } \textit{there exists } s_0 \in \mathcal{I} : \langle \pi,\ q_i\rangle \models f[s_0/s] \\
\langle \pi,\ q_i\rangle \models \Diamond_P f & \textit{iff } \textit{there exists } j, 0 \leq j \leq i : \langle \pi,\ q_j\rangle \models f
\end{array}$$

The obvious extension of satisfiability over a trace is defined as follows $\pi \models f$ iff $\langle \pi,\ q_i\rangle \models f, \forall i : 0 \leq i \leq \text{length}(\pi)$. Finally we have that a formula f is satisfied over a model $\mathcal{M}$ of a SPID protocol, written $\mathcal{M} \models f$, if and only if f is satisfied over all the traces of the model.

4 Intermezzo

In this section we present as an example the formalization in SPID, a simple protocol. Then we will show how to express a security property using the BRUTUS logic, and provide some intuition on its satisfiability over the model of the protocol.

As a protocol example let us consider the following key-exchange protocol where two principals A and B and a trusted server S are involved:

Protocol 1 (Key Exchange)

$$
\begin{aligned}
&1.\ A \rightarrow S : A, B \\
&2.\ S \rightarrow A : \{K_{AB}\}_{K_{AS}}, \{A, K_{AB}\}_{K_{BS}} \\
&3.\ A \rightarrow B : \{A, K_{AB}\}_{K_{BS}}
\end{aligned}
$$

Informally, in the first step A sends to S a message composed of its name and the name of B. With that message A intends to request a new session key to communicate with B. The server S generates a fresh session key K_{AB} and encrypts two copies of it: one using the key K_{AS}, shared with A, and the other (where there is additional information about A's identity) using the key K_{BS}, shared with B. Then, S sends (step 2) both copies to A, which in turn forwards (step 3) to B the message reserved to it. Once received, B can determine that A wants to start a confidential communication with it, and retrieve the session key it wants to use. One session of the protocol written in SPID appears as in the following:

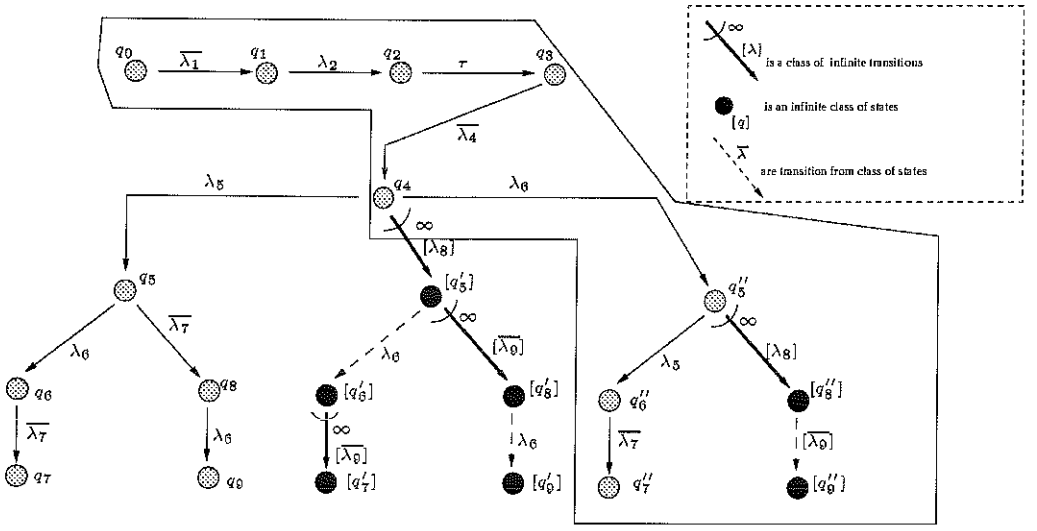
$$
\begin{aligned}
KE &\stackrel{def}{=} (\nu K_{as})(\nu K_{bs})(\nu A)(\nu B)(\nu S)(1, A) \| (2, B) \| (3, S) \\
A &\stackrel{def}{=} \overline{c_{as}}(\langle A, B \rangle).c_{as}(\langle \{x_1\}_{K_{as}}, y_1 \rangle).\overline{c_{ab}}(y_1).\mathbf{0} \\
B &\stackrel{def}{=} c_{ab}(\{\langle A, x_2 \rangle\}_{K_{bs}}).\mathbf{0} \\
S &\stackrel{def}{=} c_{as}(\langle A, B \rangle). \\
&\qquad (\mathbf{new}\ K_{ab})\overline{c_{as}}(\langle \{K_{ab}\}_{K_{as}}, \{\langle A, K_{ab} \rangle\}_{K_{bs}} \rangle).\mathbf{0}
\end{aligned}
$$

The protocol specification shows three process instances, one for each process running the protocol. Three different labels c_{ab}, c_{as} and c_{bs} are used to distinguish actions over the public channel. The two shared keys K_{as} and K_{bs}, and the names of participants A, B and S are initially hidden to the environment.

In Figure 6 we have drawn an abstract representation of the labeled transition system, model of the protocol specification KE. In particular with tight arrows and with $[\lambda]$, we have given a compact representation of an infinite class of transitions and the related actions, caused by receiving different messages from the environment's synthesis. Consequently with $[q]$ we have indicated the infinite set of states reached by $[\lambda]$. Finally with $[\overline{\lambda}]$ and dashed arrows we have represented the infinite class of transitions outgoing from $[q]$ states.

In Figure 7 we have reported a table contained detailed information about most significant states of the transition system (the ones circled in Figure 6). In the first column state names q are reported, in the second the arrays representing the global states $\mathcal{G}$ (each element of the array lays in a different row), and finally in the third column the fragment of the calculus Z representing the protocol evolution.

BRUTUS logic can be used to express security properties, for example the following authenticity property:



LEGEND ('$\star$' stands for a generic message $M \neq \{\langle A, K\rangle\}_{K_{bs}}$)

$\overline{\lambda_1} = 1.\overline{c_{as}}\langle\langle A, B\rangle\rangle$

$\lambda_2 = 3.c_{as}\langle\langle A, B\rangle\rangle$

$\overline{\lambda_4} = 3.\overline{c_{as}}\langle\langle\{K\}_{K_{ab}}, \{\langle A, K\rangle\}_{K_{bs}}\rangle\rangle$

$\lambda_5 = 1.c_{as}\langle\langle\{K\}_{K_{as}}, \{\langle A, K\rangle\}_{K_{bs}}\rangle\rangle$

$\lambda_6 = 2.c_{ab}\langle\{\langle A, K\rangle\}_{K_{bs}}\rangle$

$\overline{\lambda_7} = 1.\overline{c_{ab}}\langle\{\langle A, K\rangle\}_{K_{bs}}\rangle$

$[\lambda_8] = 1.c_{as}\langle\langle\{K\}_{K_{ab}}, \star\rangle\rangle$

$[\overline{\lambda_9}] = 1.\overline{c_{ab}}\langle\star\rangle$

Fig. 6. Example of the LTS

In the following $W_1 = \{A, B, S, K_{as}\}$, $W_2 = \{A, B, S, K_{bs}\}$ and $W_3 = \{A, B, S, K_{as}, K_{bs}\}$.

q	$\mathcal{G}$	Z
q_0	$\mathcal{G}_0 = \begin{array}{l}(\Omega, \emptyset, \bot)\\ (A, W_1, \bot)\\ (B, W_2, \bot)\\ (S, W_3, K_{bs}\}, \bot)\end{array}$	$Z_0 = \begin{array}{l}(\nu K_{as})(\nu K_{bs})(\nu A)(\nu B)(\nu S)\\ (1, A)\\ \parallel (2, B)\\ \parallel (3, S)\end{array}$
q_1	$\mathcal{G}_1 = \begin{array}{l}(\Omega, \{\langle A, B\rangle\}, \bot)\\ (A, W_1, \bot)\\ (B, W_2, \bot)\\ (S, W_3, \bot)\end{array}$	$Z_1 = \begin{array}{l}c_{as}(\langle\{x_1\}_{K_{as}}, y_1\rangle).\overline{c_{ab}}(y_1).\mathbf{0}\\ \parallel (2, B)\\ \parallel (3, S)\end{array}$
q_2	$\mathcal{G}_2 = \begin{array}{l}(\Omega, \langle A, B\rangle, \bot)\\ (A, W_1, \bot)\\ (B, W_2, \bot)\\ (S, W_3 \cup \{\langle A, B\rangle\}, \bot)\end{array}$	$Z_2 = \begin{array}{l}c_{as}(\langle\{x_1\}_{K_{as}}, y_1\rangle).\overline{c_{ab}}(y_1).\mathbf{0}\\ \parallel (2, B)\\ \parallel (3, (\mathbf{new}\ K_{ab})\overline{c_{as}}(\langle\{K_{ab}\}_{K_{as}}, \{\langle A, K_{ab}\rangle\}_{K_{bs}}\rangle).\mathbf{0})\end{array}$
q_3	$\mathcal{G}_3 = \mathcal{G}_2$	$Z_3 = \begin{array}{l}c_{as}(\langle\{x_1\}_{K_{as}}, y_1\rangle).\overline{c_{ab}}(y_1).\mathbf{0}\\ \parallel (2, B)\\ \parallel (3, \overline{c_{as}}(\langle\{K\}_{K_{as}}, \{\langle A, K\rangle\}_{K_{bs}}\rangle).\mathbf{0})\\ \text{where } K \text{ is a new name}\end{array}$
q_4	$\mathcal{G}_4 = \begin{array}{l}(\Omega, \{\langle A, B\rangle,\\ \langle\{K\}_{K_{as}}, \{\langle A, K\rangle\}_{K_{bs}}\rangle\}, \bot)\\ (A, W_1, \bot)\\ (B, W_2, \bot)\\ (S, W_3 \cup \{\langle A, B\rangle\}, \bot)\end{array}$	$Z_4 = \begin{array}{l}c_{as}(\langle\{x_1\}_{K_{as}}, y_1\rangle).\overline{c_{ab}}(y_1).\mathbf{0}\\ \parallel (2, B)\\ \parallel (3, 0)\end{array}$
q_5''	$\mathcal{G}_5'' = \begin{array}{l}(\Omega, \{\langle A, B\rangle,\\ \langle\{K\}_{K_{as}}, \{\langle A, K\rangle\}_{K_{bs}}\rangle\}, \bot)\\ (A, W_1, \bot)\\ (B, W_2 \cup \{\{\langle A, K\rangle\}_{K_{bs}}\}, \sigma_5'')\\ (S, W_3 \cup \{\langle A, B\rangle\}, \bot)\\ \text{where } \sigma_5'' = [x_2/K]\end{array}$	$Z_5'' = \begin{array}{l}c_{as}(\langle\{x_1\}_{K_{as}}, y_1\rangle).\overline{c_{ab}}(y_1).\mathbf{0}\\ \parallel (2, 0)\\ \parallel (3, 0)\end{array}$
q_6''	$\mathcal{G}_6'' = \begin{array}{l}(\Omega, \{\langle A, B\rangle,\\ \langle\{K\}_{K_{as}}, \{\langle A, K\rangle\}_{K_{bs}}\rangle\}, \bot)\\ (A, W_1 \cup \{\{\langle A, K\rangle\}_{K_{bs}}\}, \sigma_6'')\\ (B, W_2 \cup \{\{\langle A, K\rangle\}_{K_{bs}}\}, \sigma_5'')\\ (S, W_3 \cup \{\langle A, B\rangle\}, \bot)\\ \text{where } \sigma_6'' = [x_1/K]\end{array}$	$Z_6'' = \begin{array}{l}(1, \overline{c_{ab}}(y_1).\mathbf{0})\\ \parallel (2, 0)\\ \parallel (3, 0)\end{array}$
q_7''	$\mathcal{G}_7'' = \begin{array}{l}(\Omega, \{\langle A, B\rangle,\\ \langle\{K\}_{K_{as}}, \{\langle A, K\rangle\}_{K_{bs}}\rangle,\\ \{\langle A, K\rangle\}_{K_{bs}}\}, \bot)\\ (A, W_1 \cup \{\{\langle A, K\rangle\}_{K_{bs}}\}, \sigma_6'')\\ (B, W_2 \cup \{\{\langle A, K\rangle\}_{K_{bs}}\}, \sigma_5'')\\ (S, W_3 \cup \{\langle A, B\rangle\}, \bot)\end{array}$	$Z_7'' = \begin{array}{l}(1, \mathbf{0})\\ \parallel (2, 0)\\ \parallel (3, 0)\end{array}$
$[q_8'']$	$[\mathcal{G}_8''] = \begin{array}{l}(\Omega, \{\langle A, B\rangle,\\ \langle\{K\}_{K_{as}}, \{\langle A, K\rangle\}_{K_{bs}}\rangle\}, \bot)\\ (A, W_1 \cup \{\{\langle\{K\}_{K_{as}}, M\rangle\}, \bot)\\ (B, W_2 \cup \{\{\langle A, K\rangle\}_{K_{bs}}\}, \sigma_5'')\\ (S, W_3 \cup \{\langle A, B\rangle\}, \bot)\\ \text{for all } M \in KS(W_\Omega) - \{\{\langle A, K\rangle\}_{K_{bs}}\}\\ \text{where } W_\Omega = \mathsf{Know}(\mathcal{G}_5''(0))\end{array}$	$Z_8'' = \begin{array}{l}(1, \overline{c_{ab}}(y_1).\mathbf{0})\\ \parallel (2, 0)\\ \parallel (3, 0)\end{array}$
$[q_9'']$	$[\mathcal{G}_9''] = \begin{array}{l}(\Omega, \{\langle A, B\rangle,\\ \langle\{K\}_{K_{as}}, \{\langle A, K\rangle\}_{K_{bs}},\\ M'\rangle\}, \bot)\\ (A, W_1 \cup \{\{\langle\{K\}_{K_{as}}, M\rangle\}, \bot)\\ (B, W_2 \cup \{\{\langle A, K\rangle\}_{K_{bs}}\}, \sigma_5'')\\ (S, W_3 \cup \{\langle A, B\rangle\}, \bot)\\ \text{for all } M' \in KS(W_\Omega) - \{\{\langle A, K\rangle\}_{K_{bs}}\}\\ \text{where } W_\Omega = \mathsf{Know}(\mathcal{G}_8''(0))\end{array}$	$Z_9'' = \begin{array}{l}(1, \mathbf{0})\\ \parallel (2, 0)\\ \parallel (3, 0)\end{array}$

Fig. 7. Details of the LTS.

$$f \stackrel{def}{=} \forall b.\, \exists a.\, \mathbf{name}(b) = B \wedge \mathbf{name}(a) = A \wedge \\ \mathbf{Acts}_{c_{ab}}(b, \{\langle A, x_2\rangle\}_{K_{BS}}) \rightarrow \Diamond_P \mathbf{Acts}_{\overline{c_{ab}}}(a, y_1) \wedge \\ a.y_1 = b.\{\langle A, x_2\rangle\}_{K_{BS}} \tag{1}$$

Informally the formula (1) says that whenever a process instance, of the principal B, receives the message $\{\langle A, x_2\rangle\}_{K_{BS}}$ then there exists a process instance, of principal A, which has previously sent a message y_1 to B, and y_1 is exactly the same message $\{\langle A, x_2\rangle\}_{K_{BS}}$ which B has received. The satisfiability of the formula (1) is checked over the labeled transition system in Figure 6. In this case the formula is not satisfied. In fact, is easy to check that on trace $q_0 \cdot \overline{\lambda_1} \ldots q_7''$, for example, process B receives the message before A sends it, proving that the environment has maliciously assumed A's identity.

5 Model Checking

As we have seen in Section 2, the use of KS generally creates an infinite amount of input transitions. This is due to the possibility of processes to receive in input one of the infinite amount of messages composed by the environment, among which there would be the message able to break protocol security.

Anyway because our purpose is to define a model checking environment for SPID, it is we necessarily need to bound the number of messages the environment can synthesize, in order to have finite SPID protocol models. In this Section we will follow a very simple idea. In particular we introduce a bounded definition of knowledge, we call *bound synthesis*, written $KS^{(d)}(W)$. Informally the $KS^{(d)}(W)$ is composed only of those messages in $KS(W)$ of depth d, where the *depth* of a message can be the following:

Definition 7 (Depth of a Message). *Let M be a message. The* depth *of M, $Type(M)$, is defined on the structure of M as follows:*

$$\begin{array}{ll} 1.\ Type(m) & = 1 \\ 2.\ Type(\{M\}_N) & = Type(M) + Type(N) \\ 3.\ Type(\langle M, N\rangle) & = Type(M) + Type(N). \end{array}$$

Using the depth of Definition 7 we can formalize the bound synthesis as follows:

Definition 8 (Bounded Synthesis). *Let $W \subset \mathcal{M}$ be a finite set of messages, and d a positive integer. Then $KS^{(d)} : \mathcal{M} \rightarrow \mathcal{M}$, called the* d-synthesis *is a function from message sets to message sets, such that:*

$$KS^{(d)}(W) = \{M \in KS(W) : Type(M) = d\}$$

About bound synthesis the following results hold:

Lemma 1. *Let W a finite set of messages. Then for every $d \geq 0$, $KS^{(d)}(W)$ is a finite set.*

Theorem 1. *Let W be finite. Given a message M, the question $M \in KS^{(d)}(W)$ can be answered in time $O(|KA(W)| \cdot Type(M))$, where $KA(W)$ is the finite set of messages that can be inferred from W using only shrinking rules (see Figure 2).*

Proof. We can compute the $Type(M)$ in linear time in the size of M. If $Type(M) \neq d$ then $M \notin KS^{(d)}(W)$ and we have finished. Otherwise $Type(M) = d$ and we know from [7] that $M \in KS(W)$ can be answered in time $O(|KA(W)| \cdot Type(M))$.

By using $KS^{(d)}$, for some $d > 0$, instead of KS, the rule (A-IN) of Figure 4, can be redefined and the models for each SPID protocol so obtained are finite. We will call $\mathcal{R}^d$ the new set of transition rules using the bound synthesis.

The model checker is implemented by Algorithm 1, a simple procedure which visits the labeled transition system in a depth first mode. As parameters the algorithm requires a close protocol Z, the formula f to be checked, and an integer $d > 0$ used to define transitions $\mathcal{R}^d$. Informally Algorithm 1 uses two stacks: $\mathcal{S}_A$ and $\mathcal{S}_\Pi$. The former is used for implementing a depth first visit of the transition system, and it contains enabled transitions; the latter is used for storing prefixes of traces. In particular during the depth first traversal of transitions, prefixes $q_0 \cdot \alpha_1 \cdots \alpha_i \cdot q_i$ are built and the satisfiability of the formula f is checked over the state q_i using the satisfiability relation defined in Section 3.

The procedure stops with a counterexample if f is discovered to be unsatisfied in some q_i. Algorithm 1 has time complexity $O(|V| \cdot (|f| + |KA(W_{max})| \cdot |Type(M_{max})|))$ where $|V|$ is the number of states of the transition system, $|f|$ is the length of formula f, W_{max} is the greatest W_Ω and M_{max} is the longest message used within the protocol. In fact the main loop is executed $|V|$ times, and within a loop $|f|$ time is required to check the satisfiability, or $|KA(W_{max})| \cdot |Type(M_{max})|$ are in the worst case the possible enabled transitions.

It is worth to underline that use of $KS^{(d)}$ limits the number of traces of the model to a finite number, and as a consequence a formula which results to be unsatisfied is obviously unsatisfied over the general infinite branching model LTS_∞. On the contrary nothing can be said when a formula results to be satisfied.

6 Conclusions

In this paper we have presented a model checking environment for a spi-calculus dialect, called SPID. In SPID, a protocol is described as a parallel composition of a finite number of process instances, each representing a finite-behaviored agent running the protocol. More run a protocol can be described by instantiating more copies of each agent.

The SPID calculus has been provided with an operational semantics based on labeled transition systems (LTS), where the intruder is described in the Dolev-Yao style, that is by assuming the presence of an implicit environment which

Algorithm 1 Model Checking (Z a closed protocol, f formula, $d \geq 0$)

1: $q_0 = \langle \mathcal{G}_0,\ Z_0 \rangle$ {initial state, see Definition 2}
2: $\mathcal{S}_\Lambda \leftarrow \emptyset$ {an empty stack for containing transitions}
3: $\mathcal{S}_\Pi \leftarrow \emptyset$ {an empty stack containing actions and states (i.e., prefix of traces)}
4: $\texttt{push}(\epsilon \cdot q_0, \mathcal{S}_\Pi)$ {Here ϵ is used just for consistence}
5: $\forall q$, $mark(q) \leftarrow \texttt{false}$ {set all states not marked ($mark$ is an attribute of q).}
6: **repeat**
7: $\quad q \leftarrow \texttt{head}(\mathcal{S}_\Pi)$ {retrieve the element in the top of the stack (only q is important}
8: $\quad$ **if not** $mark(q)$ **then** {if q has not visited yet}
9: $\quad\quad mark(q) \leftarrow \texttt{true}$ {mark it}
10: $\quad\quad$ **if** $\langle \mathcal{S}_\Pi,\ q \rangle \not\models f$ **then** {if f is not satisfied over q along trace $\mathcal{S}_\Pi$ (Def. 6}
11: $\quad\quad\quad$ **return** $\texttt{false}, \mathcal{S}_\Pi$ {return counterexample}
12: $\quad\quad$ **else** {f is satisfied}
13: $\quad\quad\quad \Lambda \leftarrow \{\alpha : \alpha$ is an enabled transition from state q w.r.t. $\mathcal{R}^d\}$
14: $\quad\quad\quad$ **if** $\Lambda = \emptyset$ **then** {no transition is possible}
15: $\quad\quad\quad\quad \texttt{pop}(\mathcal{S}_\Pi)$ {delete head element $\alpha \cdot q$ from trace stack (i.e., backtrack)}
16: $\quad\quad\quad$ **else**
17: $\quad\quad\quad\quad \forall \alpha \in \Lambda$, $\texttt{push}(\alpha, \mathcal{S}_\Lambda)$ {push the enabled transition into $\mathcal{S}_\Lambda$}
18: $\quad\quad\quad$ **end if**
19: $\quad\quad$ **end if**
20: $\quad$ **end if**
21: $\quad$ **if** $\mathcal{S}_\Lambda \neq \emptyset$ **then** {if some transition remains}
22: $\quad\quad \alpha \leftarrow \texttt{pop}(\mathcal{S}_\Lambda)$ {retrieve next transition (i.e., depth first search)}
23: $\quad\quad$ **let** $q' : q \xrightarrow{\alpha} q'$ **in** $\texttt{push}(\alpha \cdot q', \mathcal{S}_\Pi)$
24: $\quad\quad q \leftarrow q'$
25: $\quad$ **else**
26: $\quad\quad \texttt{pop}(\mathcal{S}_\Pi)$ {backtrack}
27: $\quad$ **end if**
28: **until** $\mathcal{S}_\Pi = \emptyset$ {all states have been visited}
29: **return** $\texttt{true}$

has control over the net and which is able to compose new messages, by pairing, splitting, decryption and encryption, starting from the ones passing through the net. As a consequence any communication happens via the environment-intruder, which plays as a dynamic pool of messages: when a process sends a message this becomes part of the environment, and when a process receives a messages this is retrieved from the ones the environment can generate.

The presence of an intruder that synthesizes an infinite amount of messages during receiving actions of processes, makes our LTS infinite branching even if all processes have a finite behavior. In fact, infinite transitions arise when a process instance tries to receive a message among the infinite ones the environment can generate.

The LTS coming from SPID protocols are also the models over which the satisfiability of a linear time temporal logic, we called BRUTUS logic, has been defined. Using the BRUTUS logic, defined by Clarke, Jha and Marrero in their BRUTUS model checker for security protocols, it is possible to express secrecy, integrity, authenticity properties, some weak form of anonymity and general safety properties. Defining the satisfiability over these LTS, models of SPID protocols, is the main contribution of this paper.

Anyway, in order to provide an effective model checker procedure, possible only over finite, or finite state, we need some useful and limiting strategy. In this paper, as a first solution, we have chosen to modify the rule defining input transitions by imposing that the length of messages built by the intruder was fixed a priori. In other words the model checker algorithm takes as parameters, in addition to a closed protocol P and a BRUTUS formula f, an integer parameter $d > 0$, and in the an on-the-fly generation of the model for P, no messages of length greater than d are considered when expanding input transitions.

In this way, because the set of traces of the bound LTS is obviously a subset of the set of traces of the general LTS, an attack found over a finite model is an attack over the infinite model of a protocol. In practice, this simple strategy is sufficient for finding most common flaws, as previous works shown (e.g., see [9]).

References

1. M. Abadi. Secrecy by Typing in Security Protocols. *Journal of the ACM*, 46(5):749–786, 1999.
2. M. Abadi and A. D. Gordon. Reasoning about Cryptographic Protocols in the Spi Calculus. In *Proc. of CONCUR '97: Concurrency Theory, 8th International Conference*, volume 1243 of *Lecture Notes in Computer Science*, pages 59–73. Springer-Verlag, 1997.
3. M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols. The Spi Calculus. Technical Report 149, Digital Equipment Corporation Systems Research Center, Palo Alto, California, 1998.
4. M. Boreale. Symbolic trace analysis of cryptographic protocols in the spi-calculus. In *Proc. of ICALP 2001*, 2001.
5. M. Burrows, M. Abadi, and R. Needham. A logic of authentication. In *Proc. of the Royal Society of London*, volume 426 of *Lecture Notes in Computer Science*, pages 233–271. Springer-Verlag, 1989.
6. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specification. *ACM Transaction on Programming Languages and Systems*, 8(2):244–263, 1986.
7. E. M. Clarke, S. Jha, and W. Marrero. A Machine Checkable Logic of Knowledge for Protocols. In *Proc. of Workshop on Formal Methods and Security Protocols*, 1998.
8. E. M. Clarke, S. Jha, and W. Marrero. Partial Order Reductions for Security Protocol Verification. In *Proc. of the International Conference for the Construction and Analysis of Systems - TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 503–518, 2000.
9. E. M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with Brutus. *ACM Transactions on Software Engineering and Methodology*, 9(4):443–487, 2000.
10. D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transaction on Information Theory*, 29(2):198–208, 1983.
11. A. Durante, R. Focardi, and R. Gorrieri. A Compiler for Analyzing Cryptographic Protocols Using Noninterference. *ACM Transactions on Software Engineering and Methodology*, 9, 2000.
12. L. Durante, R. Sisto, and A. Valenzano. A State-Exploration Technique for Spi-Calculus Testing-Equivalence Verification. In *Proc. of the IFIP TC6 WG6.1 Joint International Conference FORTE XIII and PSTV XX*, pages 155–170. Kluwer Academic Publishers, 2000.
13. M. Fiore and M. Abadi. Computing Symbolic Models for Verifying Cryptographic Protocols. In *Proc. of the 14th Computer Security Foundation Workshop (CSFW-14)*, pages 160–173. IEEE, Computer Society Press, 2001.
14. A. Giani, F. Martinelli, M. Petrocchi, and A. Vaccarelli. A Case Study with PaMoChSA: A Tool for the Automatic Analysis of Cryptographic Protocols. In *Proc. the World Multiconference on Systemics,Cybernetics and Informatics and International Conference on Information Systems, Analysis and Synthesis (SCI/ISAS)*, pages 108–116, 2001.
15. A. D. Gordon and A. Jeffrey. Authenticity by Typing for Security Protocols. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW 2001)*, pages 145–159. IEEE Computer Society, 2001.
16. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. of 19th IEEE Computer Security Foundations Workshop*

(CSFW'96), volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.
17. W. Marrero, E. Clarke, and S. Jha. Model Checking for Security Protocols. In *Proc. of the DIMACS Workshop on Design and Formal Verification of Secutity Protocols*, 1997.
18. C. A. Meadows. The NRL protocol analyzer: an overview. In *Proc. of the 2nd International Conference on the Practical Application of PROLOG*, 1994.
19. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I. *Information and Computation*, 100(1):1–40, 1992.
20. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, II. *Information and Computation*, 100(1):41–77, 1992.
21. J. C. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Murϕ. In *Proceeding of the IEEE Symposium on Security and Privacy*, pages 141–151, 1997.
22. R. De Nicola and M. C. B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.
23. L. C. Paulson. Proving Properties of Security Protocols by Induction. Technical Report 409, Computer Laboratory, Univ. of Cambridge, 1996.
24. L. C. Paulson. Proving Properties of Security Protocols by Induction. In *Proc. of The 10th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.
25. S. Schneider. Verifying Authentication Protocols in CSP. *IEEE Transaction on Sofware Engineering*, 24(8):743–758, 1998.
26. J. Thayer, J. Herzog, and J. Guttman. Strand spaces: Why is a security protocol correct? In *Proc. of the 19th IEEE Computer Society Symposium on Research in Security and Privacy*, 1998.
27. T. Woo and S. Lam. A Semantic Model for Authentication Protocols. In *Proc. of the IEEE Symposium on Research in Security and Privacy*, 1993.