

Modular Verification of Systems via Service Coordination^{*}

Gianluigi Ferrari, Ugo Montanari, and Emilio Tuosto

Dipartimento di Informatica, Università di Pisa
{giangi,ugo,etuosto}@di.unipi.it

Abstract. We present a service-oriented approach to the verification of properties of distributed systems specifies in dialects of the π -calculus. Using our verification methodology it is possible to program the coordination rules which are used to specify how the sub-tasks within any verification run are to be carried out, in which order and which are the different toolkits involved. The methodology is supported by a Web-service infrastructure integrating several verification toolkits for checking properties of specifications. Our experimental results have confirmed the potential usefulness of the approach

1 Introduction

In the last years distributed applications over the World-Wide Web, e.g. peer-to-peer file sharing, have attained wide popularity. Several technologies have been developed for handling computing problems which involve a large number of heterogeneous components that are physically distributed and (inter)operate autonomously. These efforts have begun to coalesce around a paradigm where the Web is exploited as a *service distributor*. A service in this sense is not a monolithic Web server but rather a component available over the Web that others might use to develop other services. Conceptually, Web services are stand-alone components in the Internet. Each Web service has an interface accessible through standard protocols and, at the same time, describing the interaction capabilities of the service. Applications over the Web are developed by combining and integrating Web services. Moreover, no Web service has pre-existing knowledge of what interactions with other Web services may occur. The Web service framework has emerged as the standard and natural architecture to realize the so called *Service Oriented Computing* (SOC) [12, 22] paradigm where services are the basic building blocks to construct applications and service coordination becomes the main concern of the whole development process.

In [2, 14] we demonstrate that the SOC paradigm is very effective in addressing the integration issues of verification toolkits. In particular, we developed a Web-service infrastructure integrating verification toolkits for checking properties of mobile processes (eg. [13, 15, 28]) in the sense of the π -calculus [21] and

^{*} Work supported by European Union project PROFUNDIS, Contract No. IST-2001-33100.

related higher-level toolkits for verifying security protocols (e.g. [5, 26]). The development of the verification infrastructure has been performed inside the Profundis project (see URL <http://www.it.uu.se/profundis>) within the Global Computing Initiative of the European Union. For this reason we called it the *Profundis WEB*, PWeb for short.

The main idea of the approach is to make semantic-based verification toolkits available as Web services, and to establish directories for publishing such Web services. This facilitates the easy integration and maintenance of heterogeneous verification toolkits having complementary functionalities.

In the PWeb infrastructure a verification session takes the form of *service coordination*: describing the coordination patterns that a set of verification services have to follow to achieve a certain goal. In other words, the coordination rules are used to specify how the sub-tasks within any verification run are to be carried out, in which order and which are the different toolkits involved. Moreover, there are mechanisms for assigning verification sub-task to the specialized toolkits that are most appropriate to solving them.

Beyond the current prototype implementation, we envisage the important role that will be played by PWeb service coordination. Indeed, service coordination provides several benefits:

- *Model-based verification.* The coordination rules impose constraints on the execution flow of the verification session thus enabling a *model-based* verification methodology where several descriptions are manipulated together. Notice that there is a sound conceptual basis for model-based verification since verification toolkits provide an implementation of well understood semantic theories.
- *Modularity.* The verification of the properties of a large software system can be reduced to the verification of properties over subsystems of manageable complexity: the coordination rules reflect the semantic modularity of system specifications.
- *Flexibility.* The choice of the verification toolkits involved in the verification session may depend on the specific verification requirements.

We argue that service-based approaches have the potential to tackle the tool integration issues of the software engineering process.

The rest of this paper reports our experience in exploiting the facilities of the PWeb in the verification of properties of distributed systems specified in some dialect of the π -calculus. To illustrate the effectiveness and usability of our approach, we consider a case study, the verification of a cryptographic protocol, which allows us to demonstrate how service coordination supports and facilitates modular verification techniques.

2 Preliminaries

2.1 Web Services

A Web service consists of an interface describing operations accessible by message exchange over the Internet protocol stack. The description of a Web service must cover all details needed to interact with it: the message formats, the transport protocols, and son on. Hence, Web services are a programming technology for distributed systems based on Internet standards. However, Web services are not just another object-based paradigm for distributed systems. Indeed, they promote a *service-oriented* programming style which is different from the standard user-to-program style [24, 29]. The service oriented programming metaphor is usually characterised in terms of *publishing*, *finding* and *binding* cycle.

To publish-find-bind in an interoperable way Web services rely on a stack of network protocols. The building block of this protocol is the Simple Object Access Protocol (SOAP) [7]. SOAP is an XML-based messaging protocol defining standard mechanism for remote procedure calls. The Web Service Description Language (WSDL) [10] defines the interface and details service interactions. The Universal Description Discovery and Integration (UDDI) protocol supports publication and discovery facilities [30]. Finally, the Business Process Execution Language for Web Services (BPEL4WS) [25] is exploited to produce a Web service by composing other Web services

2.2 Verification Toolkits

Over the years several semantic-based verification toolkits have been designed and experimented to formally address some issues raised by software development. The *Concurrency Workbench* [11], for example, performs analysis on the Calculus for Communicating Systems (CCS). The Mobility Workbench (MWB) [28] does similar analysis but on the π -calculus. The *History-Dependent Automata Laboratory* (HAL) [13] supports verification of logical formulae expressing properties of the behaviour of π -calculus agents.

Most of the semantic-based verification environments have been developed independently of each other and there is no guarantee that they can interoperate so that the verification of certain properties is the result of a collaboration among the toolkits.

In the verification community the standard approach to deal with the integration issue is to provide a coordination infrastructure based on common format. The FC2 formal [6] is an illustrative example of this approach. The FC2 format provides a language to represent automata. An automaton is represented in the FC2 format by means of a set of tables that keep the information about state names, arc labels, and transition relations between states. FC2 allows interoperability among verification toolkits. The intermediate language approach has been further developed in the design of the VeriTech framework [17]. In this framework the integration among verification toolkits is obtained by suitable functions providing faithful translations among models and properties. A key

role is played by the *core design language* (CDL): each specification is compiled to and from the CDL representation.

A different approach is exploited by the *Electronic Tool Integration Platform* (ETI) initiative [9, 8]. ETI is a web-based infrastructure for the interactive experimentation of verification toolkits. The coordination middleware (HLL) provides the "glue" to integrate the different verification toolkits.

The PWeb proposes itself as an experiment to address the integration issue by exploiting Web services. The PWeb prototype implementation has been conceived to support reasoning about the behaviour of systems specified in some dialect of the π -calculus. The PWeb integrates and coordinate the facilities of some verification toolkits provided as Web services. The MWB and HAL are two of the services of the PWeb. Hereafter, we briefly list the main features of the other services of the PWeb.

TRUST The TRUST toolkit [27, 26] relies on an exact symbolic reduction method, combined with several techniques aiming at reducing the number of interleaving that have to be considered. Authentication and secrecy properties are specified in a very natural way, and whenever an error is found an intruder attacking the protocol is given.

MIHDA The MIHDA toolkit [15, 16] performs state minimisation of History-Dependent (HD) automata. HD automata are made out of states and labeled transitions; their peculiarity resides in the fact that states and transitions are equipped with names which are no longer dealt with as syntactic components of labels, but become explicit part of the operational model. This allows one to model explicitly name creation/deallocation, and name extrusion: these are the distinguished mechanisms of name passing calculi. MIHDA has been exploited to perform finite state verification of π -calculus specifications.

STA STA (Symbolic Trace Analyzer) [5] implements symbolic execution of cryptographic protocols. A successful attack is reported in the form of an execution trace that violates the specified property.

ASPASYA ASPASYA (Automatic Security Protocol Analysis via a SYmbolic model checking Approach) [4] relies on a symbolic technique to model check properties of cryptographic protocols. Security properties are expressed via a logic that predicates over data exchanged in the protocol and observed by an intruder in the execution environment, and also over the "presumed" identities of the protocol principals. ASPASYA allows varying the intruder's knowledge, the portion of the state space to be explored, and the specification of implicit assumptions that are very frequent in security. The user can opportunely mix those three ingredients for checking the correctness of the protocol without modifying neither the protocol specification nor the specification of the desired properties.

3 The PWeb Directory Service

The PWeb implementation has been conceived to support reasoning about the behaviour of systems specified in some dialect of the π -calculus. It supports the dynamic integration of several verification techniques (e.g. standard bisimulation checking and symbolic techniques for cryptographic protocols). The PWeb has been designed by targeting also the goal of extending available verification environments (Mobility Workbench [28], HAL [13]) with new facilities provided as Web services. This has given us the opportunity to verify the effective power of the Web service approach to deal with the reuse and integration of “existing” modules.

The core of the PWeb is a *directory service*. A PWeb directory service is a component that maps the description of the Web services into the corresponding network addresses. Moreover, it supports the binding of services.

The PWeb directory maintains references to the toolkits it works with. Every toolkit has an end-point in the directory service through the WSDL specification. As expected, the WSDL specification describes the interaction capabilities of the toolkit; namely which methods are available and the types of their inputs and outputs. In other words, the WSDL specification describes what a service can do, how to invoke it and the supported XML types (more precisely the XML Schema definitions XSD).

For instance, the WSDL-specification of MIHDA provides the description of the **reduce** service. The description of the **reduce** service refers to the XML description of the HD-automaton describing the behaviour of a π -calculus agent. The invocation of this service on a given HD-automata performs the state minimisation of the HD-automata. The WSDL-description of the **reduce** service of the MIHDA toolkit is displayed in Figure 1.

Notice that there is nothing preventing several directory services to connect to the same toolkits, or to include references to other directory services. Hence, the PWeb is basically a peer-to-peer system.

The PWeb directory service has two main facilities. The **publish** facility is invoked to make a toolkit available as Web service. The **query** facility, instead, is used to discover which are the services available. The **query** provides the service discovery mechanism: it yields the list of services that match the parameter (i.e. the XSD type describing the kind of services we are interested in).

The service discovery mechanisms is exploited by the **trader** engine. The trader engine manipulates pools of services distributed over several PWeb directory services. It can be used to obtain a Web service of a certain type and to bind it inside the application. The **trader** engine gives to the PWeb directory service the ability of finding and binding at run-time web services without “hard-coding” the name of the web service inside the application code. In other words, the **trader** engine provides the resource discovery mechanism for PWeb directory services.

The following code describes the implementation of a simple trader for the PWeb directory.

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions
  name="Mihda"
  targetNamespace="http://jordie.di.unipi.it:8080/pweb/Mihda.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://jordie.di.unipi.it:8080/pweb/Mihda.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://jordie.di.unipi.it:8080/pweb/schemas">
  <import
    namespace='http://http://jordie.di.unipi.it:8080/pweb/schemas''
    location='http://jordie.di.unipi.it:8080/pweb/hds_over_pi.xsd' />
  <types>
    <xsd:schema
      targetNamespace="http://jordie.di.unipi.it:8080/pweb/Mihda.xsd"
      xmlns="http://schemas.xmlsoap.org/wsdl/"
      xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xsd1="http://jordie.di.unipi.it:8080/pweb/Mihda.xsd">
    </xsd:schema>
  </types>
  <message name="ReduceRequest">
    <part name="contents" type="xsd1:hds_over_pi"/>
  </message>
  <message name="ReduceResponse">
    <part name="return" type="xsd1:hds_over_pi"/>
  </message>
  <portType name="MihdaPortType">
    <operation name="Reduce">
      <documentation>Minimize the automata</documentation>
      <input message="tns:ReduceRequest"/>
      <output message="tns:ReduceResponse"/>
    </operation>
  </portType>
  <binding name="MihdaBinding" type="tns:MihdaPortType">
    <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="Reduce">
      <soap:operation
        soapAction="connect:Mihda:MihdaPortType#Reduce"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="Mihda">
    <port binding="tns:MihdaBinding" name="MihdaPort">
      <soap:address
        location="http://jordie.di.unipi.it:8080/pweb/mihda"/>
    </port>
  </service>
</definitions>

```

Fig. 1. The MIHDA WSDL-specification

```

import Trader

offers = Trader.query( "reducer" )

mhda = offers[ 0 ]          # choose the first

offers = Trader.query( "model-checking" )

hal = find_neighbor(offers)  # choose the service only among neighbors

offers = Trader.query( "bisimulation-checking" )

mwb = offers[ 0 ]          # choose the first

```

The `trader` engine allows one to hide network details in the service coordination code. A further benefit is given by the possibility of replicating the services and maintaining a standard access modality to the Web services under coordination. For instance, the code

```
offers = Trader.query("security checker" )
```

can be used to obtain a coordination code that, at run-time, is able to find, bind and finally invoke any service registered as “security checker”. In the PWeb prototype implementation both TRUST and STA are registered as security checkers.

The PWeb directory service is built using Zope [31]. Zope is a (open source) framework for building web applications and is designed to allow administrators to build complex and easily maintainable web servers with a minimum amount of work. Dynamic content is supported through the use of databases which in turn can be updated through web interfaces. Zope is also highly configurable and fully object oriented. New objects can be added and inherited from if the need arises allowing for existing features to be tailored to user needs. There is also a robust security system which allows administrators to manage user privileges.

As a final remark we want to point out that the `trader` engine provides facilities which are similar to the CORBA trader. The CORBA trader is used to query object infrastructures for specific applications and components.

The current prototype implementation of the PWeb directory service can be exercised on-line at the URL <http://jordie.di.unipi.it:8080/pweb>.

3.1 Service Coordination

The fundamental technique which enables the dynamic integration of services is the separation between the service facilities (what the service provides) and the mechanisms that coordinate the way services interact (service coordination). In our experiment, the service coordination language is PYTHON. PYTHON is an interpreted object oriented scripting language which is widely used to connect existing components together.

An example of service coordination is illustrated in Figure 2 to verify a property of a specification, i.e. to test whether a π -calculus process A is a model for a formula F .

```

:
try:

    hd = mihda.compile( A )

    reduced_hd = mihda.reduce( hd )

    reduced_hd_fc2 = mihda.Tofc2( reduced_hd )

    aut = hal.unfold( reduced_hd_fc2 )

    if hal.check( aut, F ):
        print 'ok'
    else:
        print 'ko'

except Exception, e:
    print "*** error ***"

```

Fig. 2. Coordinating HAL and Mihda services

We can briefly comment on the orchestration code of Figure 2. Variables `mihda` and `hal`, have been linked by the trader engine to the required services. Now, a service of Mihda is invoked. More precisely, the result of executing the service `compile` is stored in the variable `hd`.

Next, `hd` is minimized, by invoking the service `reduce` of Mihda; and, by applying the Mihda service `Tofc2`, the minimal automaton is transformed into the FC2 format. Variable `reduced_hd_fc2` contains a HD-automaton in a format suitable for being processed by the HAL service `unfold` that generate an ordinary automaton from a HD-automaton represented in FC2 format.

Finally, a message on the standard output is printed. The message depends on whether π -calculus process A satisfies the formula F or not. This is obtained by invoking the HAL model checking facility `check`.

4 Modular Verification: A Case Study

To illustrate the effectiveness and usability of our approach, we consider a case study, the verification of a cryptographic protocol, which allows us to demonstrate how service coordination supports and facilitates modular verification techniques.

4.1 The KSL protocol

The cryptographic protocol we consider is the KSL protocol [18]. KSL provides an abstract representation of Kerberos [19] and has been conceived for the repeated authentication between principals A and B through a trusted server S . It is assumed that the trusted server shares symmetric key k_{as} and k_{bs} with principals A and B , respectively. Repeated authentication is performed by means of an expiring ticket generated by B for A . The secure communication of the ticket relies on a session-key that A and B establish with the help of S . Until the ticket is valid (not expired), A can re-authenticate itself with B (without requesting a new session key from S).

We briefly describes the informal specification of KSL as list of exchanged messages of the form Source \rightarrow Destination : Payload

- | | | |
|--|---|----------|
| 1. $A \rightarrow B : na, A$ | } | I phase |
| 2. $B \rightarrow S : na, A, nb, B$ | | |
| 3. $S \rightarrow B : \{nb, A, k_{ab}\}_{k_{bs}}, \{na, B, k_{ab}\}_{k_{as}}$ | | |
| 4. $B \rightarrow A : \{na, B, k_{ab}\}_{k_{as}}, \{Tb, A, k_{ab}\}_{k_{bb}}, nc, \{na\}_{k_{ab}}$ | | |
| 5. $A \rightarrow B : \{nc\}_{k_{ab}}$ | | |
| 6. $A \rightarrow B : ma, \{Tb, A, k_{ab}\}_{k_{bb}}$ | } | II phase |
| 7. $B \rightarrow A : mb, \{ma\}_{k_{ab}}$ | | |
| 8. $A \rightarrow B : \{mb\}_{k_{ab}}$ | | |

Looking at the structure of the protocol, we can distinguish two parts:

- messages 1 \div 5 constitutes the initial session-key exchanging phase whereas
- messages 6 \div 8 are the repeated authentication part, namely, each further interaction between A and B starts from message 6.

Notice that S does not play any role in the second phase: the trusted server has a role only in the first phase when the session key (see messages 1 \div 5) is generated and communicated.

Initiator A generates a nonce na , and sends it to B which, on turn, asks S for a new session key. In message 3, S generates the session key k_{ab} and encrypts it into two cryptograms $\{nb, A, k_{ab}\}_{k_{bs}}$ and $\{na, B, k_{ab}\}_{k_{as}}$ sent to B . After decrypting $\{nb, A, k_{ab}\}_{k_{bs}}$, B assumes that k_{ab} is the fresh session key generated by S and meant to be shared with A .

Message 4 is rather involved and crucial to establish correctness of the protocol. In this step, principal B sends to A a message containing: (i) the cryptogram $\{na, B, k_{ab}\}_{k_{as}}$ generated by S , (ii) the “ticket” $\{Tb, A, k_{ab}\}_{k_{bb}}$, (iii) a new nonce nc and (iv) the nonce na encrypted with k_{ab} .

The ticket is a cryptogram encrypted with a key k_{bb} that *only* B knows and will be used in the second part of KSL for achieving repeated authentication. Apart from the identity of A , the ticket contains a time-stamp and the session key so that B can check the validity of the ticket itself. The nonce nc will be used to prove to B that A really asked for the session key k_{ab} , while the cryptogram $\{na\}_{k_{ab}}$ is generated to witness A that B has acquired k_{ab} . Message 5 closes the first part of KSL: A sends back nc encrypted with k_{ab} so that B is granted that A acquired the session key.

Principal A knowing k_{ab} and the ticket issued by B can re-authenticate itself performing messages 6, 7 and 8. In message 6, B receives a nonce, ma , and the ticket that B has previously generated for A . If the ticket is valid, B sends ma encrypted with k_{ab} to A together with a new nonce mb , used to ensure the identity of A (message 8).

4.2 Verifying KSL

Model checking techniques have been exploited in the verification of cryptographic protocols because they can provide a counterexample when some property fails to hold: *attack generation* by counterexample. However, model checkers usually require to limit various “quantity” of the protocol (e.g., the number of participants, the length of the messages) in order to maintain the search space finite.

Traditional model checking techniques can hardly afford the complexity of protocols with many steps (as the KSL) because of the state explosion problem. In the case of KSL, also the use of symbolic techniques results harmless and can only handle sessions with a very limited number of participants.

To cope with this problem we exploit the modularity of the specification by splitting the verification session of KSL into two parts (reflecting the two phases of the protocol). Standard semantic arguments ensures that the second phase of KSL can be checked under the hypothesis that the first phase is safe. More precisely, the repeated authentication property relies on the secrecy of the session key and the tickets exchanged in the first phase. Hence, the second phase can be verified under the assumption that the session-key (and the validity ticket) are not corrupted. We, therefore, check the repeated authentication phase only when the secrecy of k_{ab} has been assessed.

The KSL verification session can be roughly described by means of the following *pseudo-code*.

```

1. safe:= false;
2. while not safe [
3.   get(property);
4.   safe:= test(property, KSL[1-5]);
   ]
5. return test(repeat_auth, KSL[6-8]).

```

This scheme can be easily implemented in the PWeb coordination language.

Notice that the coordination schemata allows us to combine two different verification techniques, namely bisimulation checking and model checking. The first phase of KSL is verified by means of a bisimulation checking technique and then model checking is exploited in the second phase. More precisely, we express (an abstraction) of KSL as π -calculus processes. Then, we prove secrecy by checking the bisimilarity of the processes with a “magic” version where the session-key cannot be compromised. The magic approach (introduced in [1] for the spi-calculus) can be briefly described as follows. If the π -calculus specification

is bisimilar to the ideal version of the protocol, then the secrecy of k_{ab} is ensured because in the “magic” version the correct key-exchange is forced. Once the secrecy of the first phase of KSL is guaranteed, we check the second phase with ASPASyA.

4.3 Encoding cryptography in π -calculus

The bisimulation checking tools of the PWeb can only deal with the π -calculus where only names and not terms can be exchanged. Hence, to model KSL, the exchange of tuples and cryptograms must be encoded. For the sake of simplicity, we do not consider here a mapping from the spi-calculus to π -calculus (like [3]) and prefer to use a simpler and more specific mapping.

Communication of tuples is simply regarded as the separated communication of each field of the tuple while the case of cryptograms is more involved and we limit ourselves to symmetric cryptography which is the only type of encryption used in KSL. Sending a cryptogram $\{M\}_k$, where k is a symmetric key, can be interpreted as sending M over k ; this guarantees that a process can acquire M only if it knows k . The delicate modelling of secret sharing/communicating is resolved by exploiting restriction of names together with *scope extrusion* (the interested reader is referred to [21, 20, 23] for a detailed introduction to π -calculus, here we only give an informal description of the calculus).

The π -calculus is a nominal calculus, namely names model communication ports along which process send/receive other (port) names. Conventionally, output of a name x on port a is written as $\bar{a}\langle x \rangle$ while $a(y)$ is the input action. Consider the process $(\nu x)(\bar{a}\langle x \rangle.P) \mid a(y).Q$; it represents the parallel composition of $(\nu x)(\bar{a}\langle x \rangle.P)$ and $a(y).Q$, the former outputs on a a freshly generated name x and continues as P , while the latter receives a on a and continues as P where the received name replaces y . The reduction representing this behaviour is written as

$$(\nu x)(\bar{a}\langle x \rangle.P) \mid a(y).Q \rightarrow (\nu x)(P \mid Q[x/y]).$$

Note that, after the communication, the scope of the binder ν also contains the continuation of the input process. This is called *scope extrusion* and is one of the main features of π -calculus.

We now describe the calculus by commenting the process representing the server of the KSL protocol. Hereafter, we use $\bar{a}\langle x_1, \dots, x_n \rangle$ (resp. $a(x_1, \dots, x_n)$) as a shorthand for $\bar{a}\langle x_1 \rangle \dots \bar{a}\langle x_n \rangle$ (resp. $a(x_1) \dots a(x_n)$).

$$\begin{aligned} S(p, a, k_{as}, b, k_{bs}) &\triangleq p(na, a', nb, b').[a' = a][b' = b] \\ &\quad (\nu k_{ab}, e_b, e_a) \\ &\quad (\bar{p}\langle e_b, e_a \rangle). \\ &\quad Enc_3[eb, k_{bs}, nb, a, k_{ab}]. \\ &\quad Enc_3[ea, k_{as}, nb, a, k_{ab}]. \\ &\quad S(p, a, k_{as}, b, k_{bs}) \end{aligned}$$

where $Enc_n[i, k, x_1, \dots, x_n]$ is a macro for $i(y).\bar{k}\langle x_1, \dots, x_n \rangle$ and represents a cryptogram encryption of n messages under the key k . The intuition is that, when

triggered on channel i , $Enc_n[i, k, x_1, \dots, x_n]$ forwards the messages x_1, \dots, x_n on the channel k representing the encryption key. The server process is parameterised with respect to six names: p represents the public channel over which all public messages are sent/received; a and b does not represent communication channels but are used just to denote the principals A and B of KSL; finally, k_{as} and k_{bs} are the symmetric keys that S shares with A and B , respectively. The server $S(p, a, k_{as}, b, k_{bs})$ waits on the public channel p for the request of B (message 2. of KSL). It reads in two variables na and nb the values of the nonces sent by B while in a' and b' the identities of the principals are stored. Then, $S(p, a, k_{as}, b, k_{bs})$ checks the identities (the construct $[a' = a]$ can be read as **if** $a' = a$ **then** \dots) and if both are correct, then three new names are generated, namely the session key k_{ab} and two auxiliary names e_b and e_a intended for the communications with the cryptograms $Enc_3[eb, k_{bs}, nb, a, k_{ab}]$ and $Enc_3[ea, k_{as}, nb, a, k_{ab}]$, respectively. Both e_b and e_a are extruded on p and the server recursively starts from the beginning.

The other principals of KSL can be specified using similar techniques. Their description are reported in Table 4.3. Finally, the session of (first phase of) the protocol is described by the following process:

$$KSL(p, a, b) \triangleq (\nu k_{as}, k_{bs})(S(p, a, k_{as}, b, k_{bs}) \mid A_I(p, a, k_{as}, b) \mid B_I(p, b, k_{bs})).$$

Note that the only non-restricted names are the public channel p and the principal identities a and b .

$A_I(p, a, k_{as}, b) \triangleq$ $(\nu na)($ $\bar{p}\langle na, a \rangle.$ $p(e).$ $\bar{e}\langle e \rangle.k_{as}(n, r, k_{ab}).$ $[n = na][r = b]$ $p(t, nc, e_1).$ $\bar{e}_1\langle e_1 \rangle.$ $k_{ab}(n).[n = na]$ $\bar{k}_{ab}\langle nc \rangle.$ $A_{II}(p, a, k_{as}, b, t, k_{ab}))$	$B_I(p, b, k_{bs}) \triangleq$ $p(na, a).(\nu nb)($ $\bar{p}\langle na, a, nb, b \rangle.$ $p(e).$ $\bar{e}\langle e \rangle.k_{bs}(n, i, k_{ab}).[n = na][i = a]$ $p(e).$ $(\nu nc, k_{bb}, Tb, e_1, e_2)(\bar{p}\langle e, e_1, nc, e_2 \rangle.$ $p(e).$ $\bar{e}\langle e \rangle.$ $k_{ab}(n).[n = nc]$ $Enc_3(e_1, k_{bb}, Tb, a, k^{ab}).$ $Enc_1(e_2, k^{ab}, na).$ $B_{II}(p, b, k_{bs}, k_{ab}, Tb, k_{bb}))$
--	---

Table 1. Principals of KSL in π -calculus

The secrecy of k_{ab} is checked by contrasting $KSL(p, a, b)$ with a slightly different version of the protocol where the processes “knows” in advance the session key. Indeed, consider

$$\widehat{KSL}(p, a, b) \triangleq (\nu k_{as}, k_{bs})(S_{k_{ab}}(p, a, k_{as}, b, k_{bs}) \mid A_{I, k_{ab}}(p, a, k_{as}, b) \mid B_{I, k_{ab}}(p, b, k_{bs})),$$

where $S_{k_{ab}}$, $A_{I,k_{ab}}$ and $B_{I,k_{ab}}$ are reported in Table 4.3. The components of \widehat{KSL} differs from those in KSL only because they share k_{ab} in advance hence $A_{I,k_{ab}}$ and $B_{I,k_{ab}}$ will use the it for their communications even if an anomalous execution might alter either k_{ab} or data sent along it. Therefore, the secrecy and integrity of KSL_I is checked by verifying that KSL and \widehat{KSL} are bisimilar.

$ \begin{aligned} &A_{I,k_{ab}}(p, a, k_{as}, b) \triangleq \\ &\quad (\nu na)(\\ &\quad \quad \bar{p}\langle na, a \rangle. \\ &\quad \quad p(e). \\ &\quad \quad \bar{e}\langle e \rangle.k_{as}(n, r, x). \\ &\quad [n = na][r = b] \\ &\quad \quad p(e, nc, e_1). \\ &\quad \quad \bar{e}_1\langle e_1 \rangle. \\ &\quad \quad k_{ab}(n).[n = na] \\ &\quad \quad k_{ab}\langle nc \rangle. \\ &\quad A_{II}(p, a, k_{as}, b, e, k_{ab}, na, nc)) \end{aligned} $	$ \begin{aligned} &B_{I,k_{ab}}(p, b, k_{bs}) \triangleq \\ &\quad p(na, a).(\nu nb)(\\ &\quad \quad \bar{p}\langle na, a, nb, b \rangle \\ &\quad \quad p(e). \\ &\quad \quad \bar{e}\langle e \rangle.k_{bs}(n, i, x).[n = na][i = a] \\ &\quad \quad p(e). \\ &\quad \quad (\nu nc, k_{bb}, Tb, e_1, e_2)(\bar{p}\langle e, e_1, nc, e_2 \rangle. \\ &\quad \quad \quad p(e). \\ &\quad \quad \quad \bar{e}\langle e \rangle. \\ &\quad \quad \quad k_{ab}(n).[n = nc] \\ &\quad \quad \quad Enc_3(e_1, k_{bb}, Tb, a, k^{ab}). \\ &\quad \quad \quad Enc_1(e_2, k^{ab}, na). \\ &\quad \quad B_{II}(p, b, k_{bs}, na, nb, nc, k_{ab}, k_{bb})) \end{aligned} $
$ \begin{aligned} &S_{k_{ab}}(p, a, k_{as}, b, k_{bs}) \triangleq p(na, a', nb, b').[a' = a][b' = b] \\ &\quad (\nu e_b, e_a) \\ &\quad \quad (\bar{p}\langle e_b, e_a \rangle. \\ &\quad \quad \quad Enc_3[eb, k_{bs}, nb, a, k_{ab}]. \\ &\quad \quad \quad Enc_3[ea, k_{as}, nb, a, k_{ab}]. \\ &\quad \quad S_{k_{ab}}(p, a, k_{as}, b, k_{bs})) \end{aligned} $	

Table 2. The magic version of KSL

5 Concluding Remarks

We started our experiment with the goal of understanding whether the SOC paradigm could be effectively exploited to integrate verification toolkits. In this respect, the prototype implementation of the PWeb is a significant example. The main advantage of our coordination model resides in providing an abstract layer to support semantic-based verification methodologies. The experiments we have performed, including the one reported in this paper, have confirmed the potential usefulness of the approach.

The basic PWeb framework can be extended in several directions. In conclusion, we list some of the area of future research *(i)* abstraction techniques for automatic decomposition, *(ii)* advanced discovery mechanisms, *(iii)* advanced coordination mechanisms and trading facilities.

References

1. M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999.
2. M. Baldamus, J. Bengston, G. Ferrari, and R. Raggi. Web services as a new approach to distributing and coordinating semantics-based verification toolkits. In *Web Services and Formal Methods*, ENTCS. Elsevier, 2004.
3. M. Baldamus, J. Parrow, and B. Victor. Spi calculus translated to p-calculus preserving may-tests. In *Annual Symposium on Logic in Computer Science LICS*, volume 19th, pages 22–31. IEEE Computer Society, July, 14 – 17 2004.
4. G. Baldi, A. Bracciali, G. Ferrari, and E. Tuosto. A coordination-based methodology for security protocol verification. In *WISP 2004 (Busi, Gorrieri, Martinelli eds)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.
5. M. Boreale and M. Buscemi. *STA, a Tool for the Analysis of Cryptographic Protocols (Online version)*. Dipartimento di Sistemi ed Informatica, Università di Firenze, and Dipartimento di Informatica, Università di Pisa,, <http://www.dsi.unifi.it/boreale/tool.html>, 2002.
6. A. Bouali, A. Ressouche, V. Roy, and R. D. Simone. The fc2tools set. In *CAV*, volume 1102 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
7. D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, M. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. *Simple Object Access Protocol (SOAP) 1.1*. WRC Note, <http://www.w3.org/TR/2000/NOTE-SOAP-2000058/>, 2000.
8. V. Braun, J. Kreiler, T. Margaria, and B. Steffen. The eti online service in action. In *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 439–443. Springer-Verlag, 1999.
9. V. Braun, T. Margaria, and B. Steffen. The electronic tool integration platform: Concepts and design. *Software Tools for Technology Transfer*, 1(1-2):31–48, 1997.
10. R. Chinnici, M. Gudgina, J. Moreau, and S. Weerawarana. Web service description language (wsdl), version 1.2. Technical report, 2002.
11. R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.
12. F. Curbera, K. R. N. Mukhi, S. Tai, and S. Weerawarana. The next step in web services. *Com. ACM*, 46(10), 2003.
13. G. Ferrari, S. Gnesi, U. Montanari, and M. Pistore. A model checking verification environment for mobile processes. *ACM Transactions on Software Engineering and Methodology*, 12(4), 2004.
14. G. Ferrari, S. Gnesi, U. Montanari, R. Raggi, G. Trentanni, and E. Tuosto. Verification on the web. In J. Augusto and U. Ultes-Nitsche, editors, *2nd International Workshop on Verification and Validation of Enterprise Information Systems, VVEIS 2004*, pages 72 – 74, Porto, Portugal, April 2004. INSTICC Press. In conjunction with ICEIS 2004.
15. G. Ferrari, U. Montanari, R. Raggi, and E. Tuosto. From co-algebraic specification to implementation: the mihda toolkit. In *First International Workshop on Methods for Components and Objects (FMCO)*, Lecture Notes in Computer Science, pages 428–440. Springer-Verlag, 2003.
16. G. Ferrari, U. Montanari, and E. Tuosto. Coalgebraic minimisation of HD-automata for the π -calculus in a polymorphic λ -calculus. *Theoretical Computer Science*, 2003. To appear.

17. S. Katz and O. Grumberg. A framework for translating models and specifications. In P. Butler and Sere, editors, *IFM 2002*, volume 2335 of *Lecture Notes in Computer Science*, pages 145–164. Springer-Verlag, 2002.
18. A. Kehne, J. Schönwälder, and H. Langendörfer. Multiple authentications with a nonce-based protocol using generalized timestamps. In *Proc. ICCS '92*, Genua, 1992.
19. J. Kohl and B. Neuman. The kerberos network authentication service (version 5). Internet Request for Comment RFC-1510, 1993.
20. R. Milner. *Communicating and Mobile Systems: the π -calculus*. Cambridge University Press, 1999.
21. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.
22. M. P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *WISE 2003*, Lecture Notes in Computer Science, pages 3–12, 2003.
23. D. Sangiorgi and D. Walker. *The π -calculus: a Theory of Mobile Processes*. Cambridge University Press, 2002.
24. M. Stal. Web services: Beyond component-based computing. *Communications of ACM*, 55(10):71–76, 2002.
25. A. T. and et al. Business process execution language for web services (bpel4ws), version 1.1. Technical report, 2003.
26. V. Vanackere. *The TRUST protocol analyser*. Lab. Informatique de Marseille, <http://www.cmi.univ-mrs.fr/vvanacke/trust.html>, 2002.
27. V. Vanackere. The trust protocol analyser, automatic and efficient verification of cryptographic protocols. In *Verification Workshop - Verify02*, 2002.
28. B. Victor and F. Moller. The Mobility Workbench — a tool for the π -calculus. In D. Dill, editor, *CAV'94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.
29. W. Vogels. Web services are not distributed objects. *IEEE Internet Computing*, 7(6):59–66, 2003.
30. W3C. UDDI Technical White Paper. Technical report, 2000.
31. Zope, <http://www.zope.org>.