

Verification on the WEB of Name-Passing Process Calculi*

Gianluigi Ferrari, Ugo Montanari, and Emilio Tuosto

Dipartimento di Informatica, Università di Pisa

Abstract. Verification toolkits are typically of rather special and, if one compares different related toolkits to each other, complementary functionalities. It has therefore been proposed to integrate different related toolkits within common environments. We argue that Web service technologies can serve as an innovative platform for addressing integration issues. Indeed, service integration in general is certainly an essential ingredient of the Web service paradigm. Furthermore, directory-like techniques will help to tackle integration of verification methodologies.

1 Introduction

In the last couple of years distributed applications over the World-Wide Web, Web for short, such as e-commerce, file sharing, have attained wide popularity, spawning an increasing demand for evolutionary paradigms in designing and controlling them. Uniform mechanisms have been developed for handling computing problems which involve a large number of heterogeneous components that are physically distributed and (inter)operate autonomously.

These developments 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.

We argue here that this paradigm can be fruitfully applied to the problem of integrating formal verification toolkits. We are motivated by the following two observations:

- There is a *need* for and a *profit to be gained* from integrating related formal verification tools since despite being related they are often of rather special and, at the same time, at least partly complementary functionality.

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

- There is a *sound conceptual basis* for integration since verification toolkits are semantics-based in the sense that they are implementations of well-understood mathematical theories. In principle it is therefore easy to match (part of) the interfaces of related tools so that they can work together.

These observations have been made before and indeed other concepts for toolkit integration have been put forward (e.g. [5]). Web services are, however, a new approach to the topic.

Our approach is distributed, thereby moving the issue to the realm of coordination. For two fundamental reasons, this way of dealing with the problem seems to be natural:

1. Some tools are but others are not or only to a limited degree portable or even available for download, and this situation is not likely to change anytime soon, for a variety of reasons. Distribution has some intrinsic advantages over centralised approaches also, such as availability, reconfigurability and openness.
2. Distributed tool integration seems to be best-viewed from coordination standpoint, since we are aiming at what we call *deep integration*: There shall be automatic means for assigning sub-tasks belonging to any verification run to those node that are most appropriate for solving them. That in turn requires both a platform and a language that allow users to direct what should be done where, at different levels of abstraction. This requirement can be regarded as almost synonymous with coordination.

Shallow integration would just mean that users could choose between different tools for tackling any verification problem as a whole.

Our immediate focus is on toolkits for verifying mobile processes (eg. [10, 11, 17]) in the sense of the π -calculus [14] and related higher-level toolkits for verifying security protocols (e.g. [2, 15]).

One main idea of the work presented here is to make semantic-based verification toolkits available as Web services, using standards such as XML, WSDL and SOAP. Another main idea is to establish directories for publishing such Web services. On this basis, then, we will be able to play out the fact that verification tools are semantics-based, as that characteristic should allow us to provide powerful automated matching facilities for services.

Verification web services plus automated verification web service directories thus provide a platform within our concept for distributed, deeply integrated and therefore coordinated verification. On top of it, there is a verification scripting facility so that users are able to specify how the sub-tasks within any verification run are to be carried out. Beyond the current prototype, we envisage that an important role in that will eventually be played by the trading functionality embedded in semantics-based directories. Specifically, there should be different levels of abstraction in proof scripting, namely goal-oriented, algorithm-oriented, tool-oriented and node-oriented levels.

The rest of this paper presents the prototype of an environment which integrates and coordinates different verification tools via the Web as a service distributor. The development of the verification environment 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.

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 [13, 18]. 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) [4]. SOAP is an XML-based messaging protocol defining standard mechanism for remote procedure calls. The Web Service Description Language (WSDL) [7] defines the interface and details service interactions. The Universal Description Discovery and Integration (UDDI) protocol supports publication and discovery facilities [19]. Finally, the Business Process Execution Language for Web Services (BPEL4WS) [9] is exploited to produce a Web service by composing other Web services

2.2 Verification Toolkits for Mobility and Security

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* [8], for example, was developed at the University of Edinburgh and performs analysis on the Calculus for Communicating Systems (CCS). The *Mobility Workbench* (MWB) [17], developed at the university of Uppsala, does similar analysis but on the π -calculus. The *History-Dependent Automata Laboratory* (HAL) [10] 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 [3] 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.

A different approach is exploited by the *Electronic Tool Integration Platform* (ETI) initiative [5]. 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 [16,15] 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 [11] 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) [2] 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) [1] 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 [17], HAL [10]) 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 available a toolkit 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" )

mihda = 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 [20]. 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. One of the main advantages of Zope is that it is portable. It runs on a variety of machines infrastructures including Windows 2000/NT/XP, Linux, Solaris and Max OS X.

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`.

3.2 Querying XML Types

The PWeb directory service includes a database of XML types which keeps track of the relationships among the XSD types which can be exploited as arguments of messages. Whenever a new XSD type is added to the PWeb directory service (e.g

as a result of a service registration), it is compared with the existing XSD types and its relationships with the other registered types are stored in the database. Whenever an application performs a query, the trader engine will provide a list of types which are compatible with the type of the query. In the prototype implementation, this is obtained by a simple script code written in Python.

We are currently investigating more expressive and powerful mechanisms for querying XML types. In particular, we started some experiments in using programming languages specifically designed to manipulate and querying XML data [12, 6].

4 Lessons Learned

We started our experiment with the goal of understanding whether the Web service metaphor could be effectively exploited to integrate in a distributed and coordinated fashion semantics-based verification toolkits. In this respect, the prototype implementation of the PWeb is a significant example.

Our approach adopts a service orchestration model whose main advantage resides in reducing the impact of network dependencies and of dynamic addition/removal of Web services by the well-identified notions of directory of services and trader engine. To the best of our knowledge, this is the first verification environment that specifically addresses the problem of exploiting Web services.

The service orchestration mechanisms presented in this paper, however, have some disadvantages. In particular, they do not exploit the full expressive power of SOAP to handle types and signatures. For instance, the so called “version consistency” problem (namely the client program can work with one version of the service and not with others) can be solved by types and signatures.

References

1. 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.
2. 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.
3. A. Bouali, A. Ressouche, V. Roy, and R. De Simone. The fc2tools set. In *CAV*, volume 1102 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
4. 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.
5. Volker Braun, Jurgen Kreileder, Tiziana Margaria, and Bernhard Steffen. The eti online service in action. In *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 439–443. Springer-Verlag, 1999.
6. Luca Cardelli and Giorgio Ghelli. Tql: A query language for semistructured data based on the ambient logic. *To appear in Mathematical Structures in Computer Science*, 2003.

7. R. Chinnici, M. Gudgina, J. Moreau, and S. Weerawarana. Web service description language (wsdl), version 1.2. Technical report, 2002.
8. Rance Cleaveland, Joachim Parrow, and Bernhard 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.
9. T. Andrews et al. Business process execution language for web services (bpel4ws), version 1.1. Technical report, 2003.
10. G. Ferrari, S. Gnesi, Ugo Montanari, and Marco Pistore. A model checking verification environment for mobile processes. *To appear in ACM TOSEM*, 2003.
11. Gianluigi Ferrari, Ugo Montanari, Roberto Raggi, and Emilio 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.
12. Haruo Hosoya and Benjamin C. Pierce. Xduce: A typed xml processing language. *ACM Transactions on Internet Technology* 3(2), pages 117–148, 2003.
13. Stal M. Web services: Beyond component-based computing. *Communications of ACM*, 55(10):71–76, 2002.
14. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts I+II. *Information and Computation*, 100:1–77, 1992.
15. V. Vanackere. *The TRUST protocol analyser*. Lab. Informatique de Marseille, <http://www.cmi.univ-mrs.fr/~vvanacke/trust.html>, 2002.
16. V. Vanackere. The trust protocol analyser, automatic and efficient verification of cryptographic protocols. In *Verification Workshop - Verify02*, 2002.
17. Björn Victor and Faron Moller. The Mobility Workbench — a tool for the π -calculus. In David Dill, editor, *CAV'94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.
18. Vogels W. Web services are not distributed objects. *IEEE Internet Computing*, 7(6):59–66, 2003.
19. W3C. UDDI Technical White Paper. Technical report, 2000.
20. Zope, <http://www.zope.org>.