Formal Modelling and Verification of an Asynchronous Extension of SOAP[*]

Maurice H. ter Beek Stefania Gnesi
Franco Mazzanti
ISTI–CNR, Via G. Moruzzi 1, 56124 Pisa, Italy
{maurice.terbeek,stefania.gnesi,franco.mazzanti}@isti.cnr.it

Corrado Moiso
Telecom Italia, Via Reiss Romoli 274, 10148 Torino, Italy
corrado.moiso@telecomitalia.it

Abstract

Current web services are largely based on a synchronous request-response model that uses the Simple Object Access Protocol SOAP. Next-generation telecommunication networks, on the contrary, are characterised by the need to handle asynchronous interactions among distributed service components, e.g., to deal with long-running computations and with events produced by the network resources. As these worlds are more and more converging into a single application context, several solutions have been proposed to deal with asynchronous events in the context of web services. In this paper we formalise and verify one such approach, viz., an original asynchronous extension of SOAP, and draw some conclusions. The formal model is specified as a set of communicating state machines. The semantics of the model is seen as a doubly-labelled transition system, and its behavioural properties are expressed in the action- and state-based temporal logic μ-UCTL and verified with the on-the-fly model checker UMC.

1. Introduction

Current web services are largely based on a synchronous request-response model that uses the Simple Object Access Protocol SOAP [19], a call-response mechanism which operates in a Client-Server paradigm. The standard setting is to use HTTP as the underlying Internet protocol, which unfortunately means that a Client remains blocked between the events of sending a request and receiving a response. Note that simply using SMTP rather than HTTP is not feasible for real-time protocols. Furthermore, a Client's request fails if the client does not receive a response within a predetermined (limited) period of time. Wireless Internet and next-generation telecommunication networks, on the contrary, are characterised by the need to handle asynchronous interactions among distributed service components, *e.g.*, to deal with long-running computations and with temporary unavailability of service components. The continuing convergence of the telecommunication and the Internet world into a single application context thus requires modern web services to integrate telecommunication features. To this aim, several solutions have been proposed in order to deal with asynchronous Message Exchange Patterns (MEP) in the context of web services.

The Asynchronous Service Access Protocol ASAP was proposed by OASIS [15] to extend the request-response interaction model of SOAP with the possibility to handle requests with long-running computations, and to provide operations to control and monitor the execution status. ASAP consists of a set of interfaces. The results of a request are returned on a web service interface that is provided by the invoking application (according to a call-back model). The main drawback of this approach is the fact that both the request parameters and the results are returned as untyped XML "blobs", which means that one loses all information concerning the typing of the interfaces. Moreover, the approach does not provide any native support for multiple responses, which could be useful to handle notifications. Our below proposal to extend SOAP does not suffer from these two drawbacks.

Further approaches include the protocols that address reliable message exchanges. Examples are WS-Reliable Messaging and WS-Reliability, both specified by OASIS [15]. WS-Reliable Messaging is a protocol that allows messages to be delivered in a reliable way between distributed applications, even in the presence of software component, system or network failures. WS-Reliable Messaging is not defined for SOAP, but a SOAP binding is defined. WS-Reliable

[*] Work partly supported by EU project IST-3-016004-IP-09 SENSORIA (*Software Engineering for Service-Oriented Overlay Computers*).

Messaging does not provide any direct support for the two aforementioned asynchronicity aspects that we consider: It does not provide direct support for asynchronous two-way MEP primitives nor for the case of Server unavailability.

WS-Reliability is a SOAP-based protocol for exchanging SOAP messages with guaranteed delivery, no duplicates and guaranteed message ordering. WS-Reliability is defined as SOAP header extensions and it is independent of the underlying protocol (the specification contains a binding to HTTP). WS-Reliability supports several ways to handle two-way MEP primitives, including one based on callbacks (the "Reliable response" of a "Reliable request" is sent to a Client via a SOAP request) but in the specifications these patterns are used only to handle the exchange of information concerning the support of the reliability (and not to exchange the response to a request of a web service operation). Moreover, analogously to the case of WS-Reliable Messaging, the Server must be available when a request is performed (and therefore the requirement on Server unavailability is not supported). In fact, WS-Reliability assumes HTTP as transport protocol. Finally, both WS-Reliable Messaging and WS-Reliability provide a set of MEP primitives to handle multiple message exchanges (with reliable characteristics) between Client and Server that are however not considered relevant in our scenario.

Yet another approach is the Web Services Notification standard WSN proposed by OASIS [15], which specifies the way web services can interact by means of event subscription and notification. The WSN standard can be thought of as defining publish/subscribe event notification for web services. Other initiatives provide similar approaches (*e.g.*, the Pubscribe mechanism defined by Apache [1]).

The main drawback of all of these approaches, however, is that they operate on the *application level*. In fact, they define specific operations and interfaces to deal with asynchronous interactions. We investigate an alternative approach, *viz.*, to enhance the communication protocol. We aim at a protocol which is backward compatible with SOAP version 1.2 (*i.e.*, it can be used by Clients that perform synchronous interactions) and which does not allow a Client to be in charge of dealing with the protocol mechanisms (we thus want to keep Clients simple). Moreover, we want a protocol that satisfies these requirements but at the same time can use HTTP as underlying transport protocol (since it is widely available for SOAP and the only standard feasible).

We thus signal the need to allow asynchronous interaction on the *protocol level* rather than on the application level, affronting moreover the problem of (temporary) unavailability of the communicating resources. The above protocols do not deal with this latter issue in a satisfactory manner: In most cases, a request simply fails if the sender or the receiver is unavailable for a predetermined (limited) period of time.

To overcome both these issues, in this paper we present a first step in the development of an original asynchronous extension of SOAP, which we call aSOAP. This step consists of the use of formal methods to analyse an initial formalisation of aSOAP. We consequently draw some conclusions with the aim of eventually arriving at a formal proposal defining aSOAP.

We specify the formal model of aSOAP as a set of communicating UML state machines [16]. We use UML as particular formal method since it has become the *de facto* industrial standard for modelling and documenting software systems. The UML semantics associates a state machine to each object in a system design, while the system's behaviour is defined by the possible evolutions of the resulting set of state machines which may communicate by exchanging signals. All these possible system evolutions are formally represented as a doubly-labelled transition system [6] in which the states represent the various system configurations and the edges represent the possible evolutions of a system configuration. Subsequently, we express several behavioural properties of our aSOAP model in the action- and state-based temporal logic μ-UCTL [9] and verify them with the on-the-fly model checker UMC [8] that allows the model checking of UML state machines. UMC is being developed at ISTI–CNR in the context of EU projects AGILE (2002-2004) and SENSORIA (2005-2009).

This paper is organised as follows. We begin by describing SOAP and its asynchronous extension aSOAP, followed by an overview of the basic concepts of the model checker UMC and the logic μ-UCTL. We then discuss our formal specification of aSOAP, after which we verify some properties and discuss the consequences. Finally, we conclude and mention some issues for future work.

2. The Simple Object Access Protocol SOAP

SOAP is a platform- and language-independent communication protocol that defines an XML-based format for applications to exchange information over IP-based protocols by using remote procedure calls—a powerful programming technique to construct distributed, Client-Server applications. This technique requires a Client to initiate a procedure call by sending a request to the Server, and then to wait. In fact, the thread remains blocked until either a response is received or it times out. The Server, on the other side, as soon as it receives the request, calls a dispatch routine that performs the requested service and sends the response to the Client. After the remote procedure call is completed—and only then—the Client may continue. In this setting a Client thus remains blocked between the events of sending a request and receiving the corresponding response. Furthermore, a Client's request fails if the Client does not receive a response within a predetermined (limited) period of time.

In the context of modern web services—integrating more and more telecommunications features—there is a need to handle asynchronous two-way MEP primitives (request-response) in a Client-Server architecture, *e.g.*, to deal with long-running computations on the Server side, and to deal with the temporary unavailability of both the Client and the Server side. To this aim, we investigate an original asynchronous extension of SOAP that is compatible with SOAP version 1.2.

3. aSOAP: asynchronous Extension of SOAP

The two main issues limiting the use of SOAP in the context of web services that integrate a number of telecommunication features thus are:

1. a mechanism to handle temporary unavailabilities of a web service consumer or a web service provider and
2. a mechanism to handle asynchronous invocations by a web service consumer,

   both on protocol level.

We thus set out to design aSOAP in order to overcome these limitations on the protocol level. Regarding the first issue, aSOAP is defined to operate in a Client-Server architecture with an additional web service Proxy placed in between the Client and the Server side. This Proxy must guarantee that various attempts to contact either side are made in case of temporary unavailability of the respective side. Regarding the second issue, aSOAP requires that a Client, whenever it is willing to accept the possibility of an asynchronous response to its request, sends the Proxy not only its request but also the `URL` at which it would like to receive the response. We consider this `URL` to be the address of a generic "SOAP listener" and we assume the application level to be equipped with a mechanism capable of receiving SOAP messages at this `URL`. The Client is not blocked during an asynchronous SOAP invocation. Furthermore, the Proxy is assumed to be always reachable by both Client and Server whenever they have an active connection. The reference model of aSOAP is presented in Fig. 1.

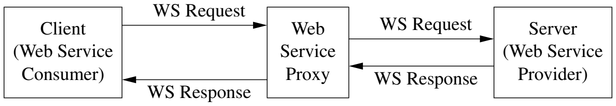

Figure 1. Reference model of aSOAP.

The crucial element to understand aSOAP is the functioning of the Proxy. When the Proxy receives a `SOAP Invocation` from the Client, it forwards it to the Server if the Server is currently reachable. If the Server is momentarily unreachable, then the Proxy applies a retry policy to contact the Server, while at the same time it informs the Client of the Server's unavailability via a `SOAP Unreachable`. If during this retry phase the Server becomes reachable, then the Proxy forwards the Client's `SOAP Invocation` to the Server. If, on the other hand, a predetermined (limited) period of time to retry passes without the Server becoming reachable, then the Proxy sends the Client a `SOAP Failure`, which the Client acknowledges with `SOAP OK`.

If the Client is willing to accept the possibility of an *asynchronous* response to its `SOAP Invocation`, then it inserts the `URL` of the SOAP listener where it would like to receive the response in the SOAP header. When the Proxy receives such a `SOAP Invocation(URL)` from the Client, it generates a Request Identifier `REQ-ID`, which uniquely identifies the Client's `SOAP Invocation`, and adds it to the SOAP header. Consequently, it tries to forward the resulting `SOAP Invocation(REQ-ID)` to the Server. Obviously the Proxy adds this `REQ-ID` to the headers of all SOAP messages it needs to send to the Client regarding this particular request.

When the Server receives a `SOAP Invocation` or a `SOAP Invocation(REQ-ID)` from the Proxy, its reaction depends on the time needed to elaborate a response. If the response is immediate, then the Server sends the Proxy a `SOAP Result`, otherwise it sends a `SOAP Deferred`. First assume that the response is immediate. The Proxy then tries to forward this response to the Client. If the Client is reachable, then its request operation is concluded after it has received the `SOAP Result(REQ-ID)` and acknowledged it with a `SOAP OK` to the Proxy. If, on the contrary, the Client is unavailable, then the Proxy applies a retry policy to contact the Client. If during this retry phase the Client becomes reachable, then the Proxy forwards it the `SOAP Result(REQ-ID)` and the request operation is concluded when the Proxy has received a `SOAP OK` acknowledgement from the Client. If, on the other hand, a predetermined (limited) period of time to retry passes without the Client becoming reachable, then the Proxy irrevocably discards the request operation.

Next assume that the Server response was not immediate, but that a long-running computation is needed to elaborate a response, *i.e.*, the Server has sent the Proxy a `SOAP Deferred`. The Proxy then informs the Client of the delay of the particular request by sending it a `SOAP Deferred(REQ-ID)`. When, after a while, the Server has elaborated the response, it sends a `SOAP Result(REQ-ID)` to the Proxy, which acknowledges with `SOAP OK` and consequently initiates the usual retry policy to forward the Client the `SOAP Result(REQ-ID)`.

A message sequence chart of the particular scenario described above is presented in Fig. 2.

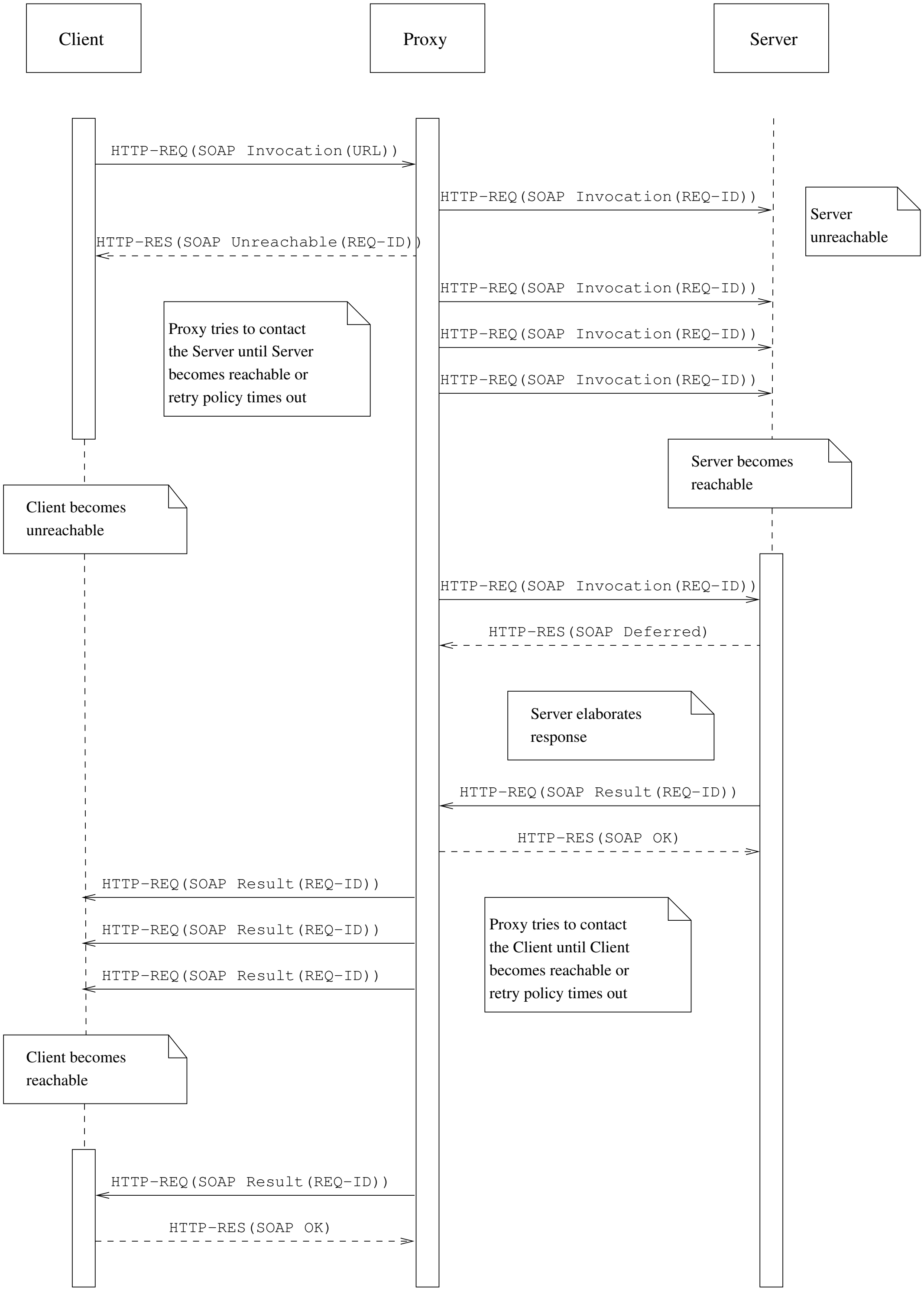
Client
Proxy
Server
HTTP-REQ(SOAP Invocation(URL))
HTTP-REQ(SOAP Invocation(REQ-ID))
Server unreachable
HTTP-RES(SOAP Unreachable(REQ-ID))
HTTP-REQ(SOAP Invocation(REQ-ID))
Proxy tries to contact the Server until Server becomes reachable or retry policy times out
HTTP-REQ(SOAP Invocation(REQ-ID))
HTTP-REQ(SOAP Invocation(REQ-ID))
Server becomes reachable
Client becomes unreachable
HTTP-REQ(SOAP Invocation(REQ-ID))
HTTP-RES(SOAP Deferred)
Server elaborates response
HTTP-REQ(SOAP Result(REQ-ID))
HTTP-RES(SOAP OK)
HTTP-REQ(SOAP Result(REQ-ID))
Proxy tries to contact the Client until Client becomes reachable or retry policy times out
HTTP-REQ(SOAP Result(REQ-ID))
HTTP-REQ(SOAP Result(REQ-ID))
Client becomes reachable
HTTP-REQ(SOAP Result(REQ-ID))
HTTP-RES(SOAP OK)

Figure 2. Message sequence chart of an aSOAP scenario.

It is important to note that aSOAP is fully proprietary and non-standard. It is moreover designed to have a minimal impact on existing architectures by preserving backward compatibility with Clients and Servers using SOAP—a Client that does not require asynchronous invocations need not be modified—and by concentrating the overhead resulting from the extension as much as possible in the Proxy. In particular, SOAP versions 1.1 and 1.2 specifically permit the addition of a SOAP header (in which the web service consumer can indicate its willingness to accept an asynchronous response and the URL at which it is ready to accept this response) to a SOAP message. Hence aSOAP can be implemented by using SOAP version 1.2:

- the headers can be SOAP version 1.2 headers and
- the Proxy may be an intermediary node that processes the aSOAP headers.

4. Modelling and Verification with UMC

In this section we briefly discuss the model checker UMC and the logic μ-UCTL, followed by the formalisation of aSOAP and the verification of some properties.

4.1. Background on UMC and μ-UCTL

Model checking is an automatic technique to verify whether a concurrent, distributed system design satisfies its specifications and certain desired properties [5]. The verification is moreover exhaustive, *i.e.*, all possible input combinations and states are taken into account. Obviously, the level of completeness of a verification depends on the range of properties verified. Compared to testing, model checking generally needs to be performed on an abstract system (design) in order to avoid state-space explosions. However, more problems are usually found by model checking the full behaviour of a scaled-down system than by testing some behaviour of the full system.

UMC is a model checker that creates and traverses the state space *on the fly*. The advantage of on-the-fly model checking is that often only a fragment of the full state space needs to be generated and analysed to obtain a satisfying result. The development of UMC is still in progress [8, 9] and a prototypical version is being used internally at ISTI-CNR for academic and experimental purposes. So far, the focus of the development has been on the design of the kind of qualitative features one would desire for such a tool, experimenting with various logics, system modelling languages, and user interfaces. The quantitative aspects of such a tool, *e.g.*, concerning optimizations aimed at limiting state-space explosions or the complexity of the evaluation algorithms (to their known minimal limits), have not yet been taken into consideration. At the time of writing there has not yet been an official public release of the tool, even if the current prototype can be experimented via a web interface at the address http://fmt.isti.cnr.it/umc/.

UMC can be used to verify models specified as communicating UML state machines. The Unified Modelling Language UML is a graphical modelling language for object-oriented system design [16, 17] that was introduced to visualise, specify, construct and document several aspects—or views—of systems. Different diagrams are used to describe the different views. UMC uses UML statecharts, which describe the dynamic aspects of system behaviour and cover both the state-based and the event-based modelling paradigms. According to the UML paradigm a system is constituted of a set of evolving and communicating objects. Each object has a set of local attributes, an event pool collecting the set of events to be processed and a set of active states inside a corresponding statechart.

The UML semantics then associates the concept of a state machine to each object in a system design, while the system's behaviour is defined by the possible evolutions of the resulting set of state machines which may communicate by exchanging signals. All these possible system evolutions can be formally represented as a doubly-labelled transition system in which the states represent the various system configurations and the edges the possible evolutions of a system configuration. We refer to [16, 17] for the precise definition of state machines and to [6] for that of doubly-labelled transition systems.

The behavioural properties that UMC can verify need to be expressed in the action- and state-based temporal logic μ-UCTL [9], which includes both the branching-time action-based logic ACTL [6] and the branching-time state-based logic CTL [4]. The syntax of μ-UCTL is

$$\phi ::= \text{true} \mid \phi \wedge \phi \mid \neg\phi \mid p \mid <\chi> \phi \mid \min \mathrm{Z}\colon \phi,$$

where ϕ is a *state formula*, p is a *predicate*, $<\chi>\ \phi$ is the *strong next* operator, χ is an *action formula* and $\min \mathrm{Z}\colon \phi$ is the *minimal fixed point* operator.

As said before, the semantics of μ-UCTL is given over doubly-labelled transition systems [6]. Informally, a formula is true on a doubly-labelled transition system if its set of transitions confirms what the formula states. In particular, p is true in a state S if it belongs to the predicates that are true in S and $<\chi>\ \phi$ is true in a state S if ϕ is true in a successor state of S that can be reached by an action satisfying χ. We refer the reader to [9] for the formal definitions of all μ-UCTL operators.

Starting from the basic μ-UCTL operators, one can derive the standard μ-calculus operators such as $\vee$, $\Rightarrow$, $\max \mathrm{Z}\colon \phi$ and $[]$ in the usual way. In particular, $[\,\chi\,]\,\phi$ can be defined as $\neg < \chi > \neg\phi$. Furthermore one can of course also derive the standard CTL/ACTL-like temporal operators like EF ("possibly"), AF ("eventually"), AG ("always") and the various Until operators in the usual way. In particular, EFϕ = min Z : $\phi \vee <\text{true}>$ Z (*i.e.*, EFϕ is true if there is an execution path on which ϕ holds in at least one reachable configuration), AFϕ = min Z : $\phi \vee (\neg$FINAL $\wedge$ [true] Z) where FINAL = $\neg <$ true $>$ true (*i.e.*, AFϕ is true if along all execution paths a certain configuration is reached in which ϕ holds) and AG$\phi = \neg$EF$\neg\phi$ (*i.e.*, AGϕ is true if along all computation paths ϕ holds in all configurations).

Note that μ-UCTL has the same expressive power as the propositional μ-calculus [12] when an arbitrary nesting of μ (minimal) and ν (maximal) fixed points are used. The main difference between μ-UCTL and the μ-calculus lies in the syntactic extension that allows the expression of both state-based properties (*i.e.*, those definable in the propositional μ-calculus) and action-based properties (*i.e.*, those expressible instead in the Hennessy-Milner logic plus recursion [13]). It is well known that logics like μ-UCTL include both linear-time logics (*e.g.*, LTL [14]) and branching-time logics (*e.g.*, CTL and CTL* [2, 3] and ACTL and ACTL* [6]) and that these logics have different expressive power in terms of the properties definable in them.

4.2. Towards a Specification of aSOAP

Before discussing several aspects of our formal specification of aSOAP, we first recall the assumptions that are part of the design of aSOAP.

1. The Proxy is always reachable by both the Client and the Server whenever they have an active connection;

2. If the Client is willing to accept the possibility of an *asynchronous* response to its SOAP Invocation, then it inserts the URL of the SOAP listener where it wants to receive the response in the SOAP header;

3. The URL in the header of an asynchronous SOAP Invocation(URL) is the address of a generic SOAP listener and the application level is equipped with a mechanism capable of receiving SOAP messages at this URL;

4. Upon receiving an asynchronous SOAP Invocation (URL) from the Client, the Proxy generates a Request Identifier REQ-ID that uniquely identifies the Client's SOAP Invocation in further communications.

Regarding the first assumption it is important to note that we consider a scenario in which both the Client and the Proxy reside in the so-called Service Layer of an operator. The application that may invoke web services (*i.e.*, the Client) is thus connected with a high level of stability. In particular, web services are not invoked from a mobile terminal with unstable connectivity. In our scenario it is the Server that might be a mobile terminal. Furthermore, a mobile scenario is only one of the scenarios considered, in the sense that aSOAP might be used also to handle requests that activate long-running computations.

During several sessions between ISTI–CNR and Telecom Italia we have discussed our design and developed our formalisation of aSOAP in detail. To facilitate the discussions about the behaviour of the various use-case scenarios of aSOAP, we decided upon a separate message sequence chart for each such a scenario. These use cases clearly depend on the reachability of both Client and Server and on the type of elaboration of the Server's responses. Let **R** denote reachability, **U** unreachability and **N** and **I** non-immediate and immediate Server response, respectively. Then Table 1 contains all use-case scenarios of aSOAP.

Table 1. The aSOAP use-case scenarios.

General cases	Associated degenerate cases
UUN Both the Client and the Server are unreachable and Server response is not immediate	**RUN** Client is always reachable but Server is unreachable when it must receive the request
	URN Server is always reachable but Client, after effectuating the request, becomes unreachable
	RRN Client and Server are always reachable
UUI Both the Client and the Server are unreachable and Server response is immediate	**RUI** Client is always reachable but Server is unreachable when it must receive the request
	URI Server is always reachable but Client, after effectuating the request, becomes unreachable
	RRI Client and Server are always reachable

Initially, the behaviour of aSOAP in each of its use-case scenarios has been described by means of message sequence charts. The one for use-case scenario **UUN**, *e.g.*, is depicted in Fig. 2. Finally, all these scenarios were translated into an operational model, in which the following concrete modelling choices were adopted.

1. All SOAP invocations are asynchronous, *i.e.*, we abstract from the synchronous SOAP invocations that only serve to guarantee backward compatibility with SOAP versions 1.1 and 1.2;

2. The URL in the header of a SOAP message is identified with the Client, *i.e.*, each Client is seen as just a listener of asynchronous SOAP invocations;
3. A system model is constituted by a Server (and its subthreads), a Proxy (and its subthreads) and a fixed (configurable) number of Clients.
4. The Proxy and the Server may activate at most a fixed (configurable) number of parallel subthreads;
5. With the Client or the Server unreachable, the Proxy attempts to contact them up to maxretries times.
6. The Client issues a single SOAP invocation and then terminates.[1]

As said before, lack of space prohibits the inclusion of the full specification of aSOAP. To still provide the reader with a flavour of our formalisation, we now give the full specification of the Client and its UML statechart and only the minimal information needed to get an idea of the functioning of the Proxy, the Server and the whole system.

```
Class Client is
Operations:
  SOAP_Result(requid:Tokens):Tokens;
  SOAP_Failure(requid:Tokens):Tokens;
Vars:
  status: Tokens := Inactive;
  theproxy: Proxy;
  result: Tokens[] := [];
State top = ready, check, wait, done
Transitions:
  ready -> check // Proxy always reachable
       { -
       / status := Running;
         result := theproxy.PSOAP_Invocation(self)
       }
  check -> wait // Server initially unreachable
       { -[result[0]=Server_Unreachable]
       / result := []; // Wait for deferred results
       }
  check -> wait // Connection with Proxy lost
       { -[result[0]=Client_Unreachable]
       / result := []; // Wait for deferred results
       }
  check -> wait // Explicit notification by Server
       { -[result[0]=Soap_Deferred]
       / result := []; // Wait for deferred results
       }
  check -> done // Immediate result from Server
       { -[result[0]=Soap_Result]
       / status := Done; // Listener execution
         result := [];   // completed
       }
  wait -> done // Issued invocation to Proxy
       { SOAP_Result(requid) // Wait for result from
       / status := Done;     // Proxy to complete
         return Soap_OK;     // listener execution
       }
  wait -> done // In "wait" status not certain that
       { SOAP_Failure(requid) // result will arrive
       / status := Done;
         return Soap_OK;
       }
end Client;
```

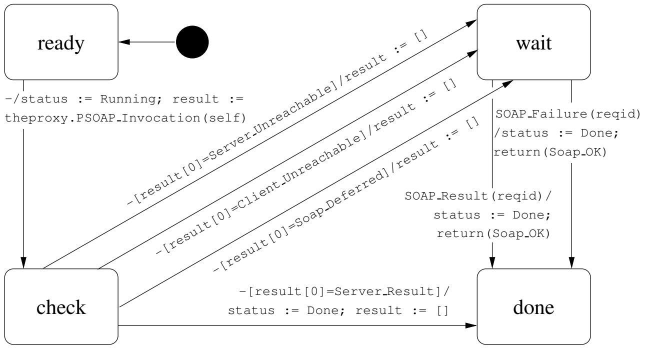
ready
wait
check
done
–/status := Running; result :=
theproxy.PSOAP_Invocation(self)
–[result[0]=Server_Unreachable]/result := []
–[result[0]=Client_Unreachable]/result := []
–[result[0]=Soap_Deferred]/result := []
SOAP_Failure(reqid)
/status := Done;
return(Soap_OK)
SOAP_Result(reqid)/
status := Done;
return(Soap_OK)
–[result[0]=Server_Result]/
status := Done; result := []

Figure 3. Statechart specification of Client.

```
Class Server is
Operations:      // Incoming from Proxy:
  SOAP_Invocation(requid:Tokens): Tokens;
...
end Server;

Class Proxy is
Operations:        // Incoming from Client:
  PSOAP_Invocation(theurl:Client): Tokens[];
...
end Proxy;

Object C1: Client(theproxy=>P1); // System w/ 2 Clients
Object C2: Client(theproxy=>P1); // <3 par. subthreads:
Object P1: Proxy(theserver=>S1,My_Threads=>[PT1,PT2]);
Object PT1: ProxyThread(maxretries=>2); // maxretries=2
Object PT2: ProxyThread(maxretries=>2); //(each thread)
Object S1: Server(My_Threads=>[ST1,ST2]); // <3 par.
Object ST1: ServerThread;                 // subthreads
Object ST2: ServerThread;
```

The reader can consult the full specification online [18].

4.3. Initial UMC Verifications of aSOAP

We now show that the abstractions which we have applied to model aSOAP are sufficient to allow the verification of an initial set of properties with UMC.

All verifications reported in this paper have been performed by running UMC version 3.3 on a SUN workstation with 1 Gigabyte of available physical memory.

To begin with, we studied the state-space complexity of our aSOAP specification. The results are reported in Table 2 for various values of the specification's parameters.

The parameters are the number of Clients (1–3), the maximum number of attempts that the Proxy may try to contact the Client or the Server in case of their unreachability (1 or 2) and the maximum number of parallel subthreads that the Proxy and the Server may activate (both 1 or 2). The entries of the third column indicate these three maxima, *i.e.*, $1-2-1$ means the Proxy may try to contact the Client or Server at most once, the Proxy may activate at most two parallel subthreads and the Server at most one.

We see that the number of states increases exponentially with the number of Clients, which is of course due to the

[1] In the future we intend to consider Clients that perform a loop of SOAP invocations or that issue several SOAP invocations before waiting for the deferred SOAP results.

Table 2. Space complexity of specification.

Model	Clients	Maxima	States[2]
1	1	$1-1-1$	151
2	1	$2-1-1$	319
3	1	$1-2-1$	151
4	1	$2-2-1$	319
5	1	$1-2-2$	151
6	1	$2-2-2$	319
7	2	$1-1-1$	3701
8	2	$2-1-1$	8321
9	2	$1-2-1$	23953
10	2	$2-2-1$	97699
11	2	$1-2-2$	87569
12	2	$2-2-2$	393907
13	3	$1-1-1$	67099
14	3	$2-1-1$	151414
15	3	$1-2-1$	> 500000
16	3	$2-2-1$	> 500000
17	3	$1-2-2$	> 500000
18	3	$2-2-2$	> 500000

explosion of possible interleavings that is inherent to system models with more than one Client. As a result, already many system models with three Clients have state spaces that cannot possibly be traversed in full—let alone reasoned about without using automatic analysis tools. Since UMC is an on-the-fly model checker, however, we will see below that certain properties can nevertheless be verified for such system models and—in fact—even for much larger system models. We also note that the maximum number of parallel subthreads that the Proxy and the Server can activate influences the total number of states of a system model only in system models with more than one Client.

Subsequently we set out to verify several behavioural properties expressed in μ-UCTL. Consider model #12 (*i.e.*, a system with two Clients, with a Proxy that may try at most two times to contact the Client or the Server in case of their unreachability and that may activate at most two parallel subthreads and with a Server that may activate at most two parallel subthreads). We first verified that

All system executions eventually reach a configuration
in which all Clients are in status `Done`. (Property 1)

by verifying the formula

```
AF ((C1.status=Done) and (C2.status=Done))
```

The formula turned out to be false, *i.e.*, Property 1 does not hold. The reason is the fact that the Server's response need not reach the Client: A possible system execution (which can also occur in the minimal system models #1 and #2 with just one Client and no parallelism) is such that the Client's SOAP invocation is being deferred by the Server, but its subsequent final SOAP result never reaches the Client because the Client becomes unreachable for a sufficiently long time for the Proxy to cancel the SOAP invocation.

Interesting enough, Property 1 can easily be verified also for system models with tens of Clients (in case of a system model with five Clients, *e.g.*, only 117 states need to be explored in depth-first mode). This is because UMC is an on-the-fly model checker that, as said before, only generates and analyses the fragment of the full state space that is needed to obtain the result.

The question remains whether Property 1 does hold in a setting in which the Client and the Server are always reachable. Second, we thus verified that

For all execution paths without communication failures
the system will eventually reach a configuration
in which all Clients are in status `Done`. (Property 2)

whose formalisation has the minimal fixed point structure

```
min RECPROPERTY:
"system is in a final correct state"
 or "communication failure occurred"
 or "system is not in a final state"
                  and [] RECPROPERTY
```

The actual formula used in UMC is

```
min Z :
  ( ((C1.status=Done) and (C2.status=Done))
 or((PT1.result=Client_Unreachable)
   or (PT1.result=Server_Unreachable)
   or (PT2.result=Client_Unreachable)
   or (PT2.result=Server_Unreachable))
 or(not FINAL and [true] Z) )
```

This formula is true. Hence Property 2 does hold. To obtain this result, UMC analysed 34735 states. While Property 2 can still be verified for system models with three Clients by exploring up to 96928 states, this is no longer the case for system models with more than three Clients.

Finally, consider the simplest system model #1 (*i.e.*, a system with just one Client, with a Proxy that tries to contact the Client or the Server only once and that may not

[2] $>$ 500000 indicates that the actual number of states is known to be larger than 500000, even if the precise real number has not been computed.

activate any parallel subthread and with a Server that may neither activate any parallel subthread). We verified that

If Client receives `SOAP_Result(ReqId)` *operation call then it received* `[Soap_Deferred,ReqId]` *response to its previous* `PSOAP_Invocation`. (Property 3)

by verifying the formula

```
AG [C1.SOAP_Result(*,ReqId)]
        (C1.result=[Soap_Deferred,ReqId])
```

While this formula should obviously be false, in our current model it actually is true, *i.e.*, Property 3 does not hold. The reason is that the Proxy may find the Client unreachable, and thus be unable to notify the Client of the deferred Server response and of the `REQ-ID` that it has generated for its request. This of course does not prevent the request to proceed its usual course, until eventually the deferred result is produced by the Server. However, in this particular scenario the `REQ-ID` associated to this result will mean nothing to the Client. We are currently studying the gravity of this particularity and whether there is a way to avoid it.

5. Conclusions and Future Work

This paper describes ongoing work on applying academic experience with formal modelling and verification to an industrial case study. Our goal is to use formal methods in the design phase of an asynchronous extension aSOAP of SOAP in order to eventually arrive at a proposal of which we can guarantee that it satisfies certain desirable properties. To this aim, we have shown a formal framework which allows one to analyse and verify behavioural properties of the ongoing aSOAP design. Again, the aSOAP design and verification activity is still in progress. The experience that we have gained so far demonstrates that being able to formally model and verify the reference models for the protocol may actually increase the confidence in the design. In particular, some of the properties that we have checked in this paper show their validity or invalidity already in the minimal case of a system with just one Client, and the effect of composing multiple Clients in parallel did not introduce any behavioural novelties apart from a huge increase in verification complexity. Indeed, the validity of these properties might still be checked by hand. This may seem to reduce the usefulness of the formal modelling effort. However, sometimes having a mechanical verification of a property already considered “intuitively true” might be of interest from the point of view of the validation of a product. Moreover, our approach promises to be well applicable also to quite bigger specifications in which the validity of a property is no longer so easy to judge.

Due to the similarities of both goals (UML verification) and techniques (on-the-fly model checking) it would definitively be of interest to compare our model and results with an equivalent (using the textual UML format) model in the HUGO-RT framework [11]. One of the main differences between our approach and that of HUGO (apart from the real-time aspects, not considered here) probably lies in the kind of logic used to specify the properties. The HUGO/SPIN approach is based on a state-based linear-time logic (LTL), while our approach relies on a UML-oriented action- and state-based full μ-calculus (μ-UCTL). Again, the properties shown in this paper do not exploit well the actual need of a branching-time logic, since they are also easily expressible as LTL formulae. Future work in this direction might provide more insights also regarding these aspects.

Apart from HUGO, we are aware of only one other freely available tool that might allow one to model check a system described in UML terms as a collection of evolving state machines defined through UML statecharts, *viz.*, UMLAUT [10]. In that case the model checking is achieved in the context of the CADP toolset [7]. This is surely another project which we would like to monitor, but unfortunately we have not yet had the time to analyse it in detail.

6. Acknowledgements

We thank Diego Latella, Mieke Massink and Ermes Thuegaz for discussions on some of the issues addressed in this paper.

References

[1] Apache Web Services Project, Pubscribe. http://ws.apache.org/pubscribe/

[2] M. Ben-Ari, A. Pnueli and Z. Manna, The Temporal Logic of Branching Time. *Acta Informatica* 20 (1983), 207–226.

[3] E.M. Clarke, E.A. Emerson, Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Proc. Logic of Programs Workshop*, LNCS 131, Springer, Berlin, 1981, 52–71.

[4] E.M. Clarke, E.A. Emerson and A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Programming Languages and Systems* 8, 2 (1986), 244–263.

[5] E.M. Clarke, O. Grumberg and D.A. Peled, *Model Checking*. MIT Press, Cambridge, MA, 1999.

[6] R. De Nicola and F.W. Vaandrager, Three logics for branching bisimulation. *JACM* 42, 2 (1995), 458–487.

[7] H. Garavel, OPEN/CÆSAR: An OPen Software Architecture for Verification, Simulation, and Testing. In *Proc. Int. Conf. Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, LNCS 1384, Springer, Berlin, 1998, 68–84.
http://www.inrialpes.fr/vasy/cadp/

[8] S. Gnesi and F. Mazzanti, On the fly model checking of communicating UML State Machines. In *Proc. Int. Conf. Software Engineering Research, Management and Applications (SERA'04)*, 2004, 331–338.

[9] S. Gnesi and F. Mazzanti, A Model Checking Verification Environment for UML Statecharts. Presented at AICA'05, 2005.

[10] W.M. Ho, J.-M. Jézéquel, A. Le Guennec and F. Pennaneac'h, UMLAUT: an Extendible UML Transformation Framework. Tech. Rep. 3775, INRIA, Rennes, 1999. http://www.inria.fr/RRRT/RR-3775.html

[11] A. Knapp, S. Merz and Ch. Rauh, Model Checking Timed UML State Machines and Collaborations. In *Proc. Int. Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'02)*, LNCS 2469, Springer, Berlin, 2002, 395-414.
http://www.pst.ifi.lmu.de/projekte/hugo/

[12] D. Kozen, Results on the Propositional μ-Calculus. *TCS* 27 (1983), 333–354.

[13] K. G. Larsen, Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion. *TCS* 72, 2–3 (1990), 265–288.

[14] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems—Specification*. Springer, Berlin, 1992.

[15] OASIS (Organization for the Advancement of Structured Information Standards).
http://www.oasis-open.org/

[16] Object Management Group, UML (Unified Modeling Language). http://www.uml.org/

[17] J. Rumbaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, MA, 1998.

[18] Specification of aSOAP.
http://fmt.isti.cnr.it/umc/examples/0-ASOAPPP.umc

[19] W3C (WWW Consortium), Latest SOAP versions.
http://www.w3.org/TR/soap/