Composability of Specifications: Patterns and Properties

(DRAFT)
V0.5
March 18, 2004

Authors:
Jacques Durand
Alan Weissberger

Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Some Properties of Composability</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Supportive Composition of Specifications</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Constitutive Composition of Specifications</td>
<td>5</td>
</tr>
<tr>
<td>2.4 Functional Composition of Specifications</td>
<td>6</td>
</tr>
<tr>
<td>3.1 The Fault Perspective</td>
<td>8</td>
</tr>
<tr>
<td>3.2 The Contract Perspective</td>
<td>9</td>
</tr>
</tbody>
</table>
1. Introduction

The ability to compose Web services specifications is becoming a critical issue for end-users as more combinable functions are being standardized (by OASIS, W3C, etc.) A typical example is a secure, reliable SOAP message exchange that relies on a general addressing support for identifying the endpoints. Although the notion of composition in the area of Web services standards has increasingly been given attention [WScomposition], no in-depth analysis of what really makes a specification “composable” has been done by any known standard body.

To be more precise, the “composability” of separate specifications, defined as the ability to integrate or operate implementations of these - is as much a property of implementations as it is a property of specifications. Even if a specification allows for composability, different implementations may vary greatly in their ability to compose with other functions implementing other specifications. Also, different features within a specification may offer different composability prospects. In a SOAP processing context, detached signatures, as defined in XML-Signature [XMLSig], will offer better composability with other message-processing functions than enveloped signatures or enveloping signatures. It is often more appropriate to refer to the composability of implementations of specifications, rather than the composability of the constituent specifications. This is true in particular for functional compositions defined later.

The investigation of composability in this document identifies the properties which make specifications and their implementations composable. Its focus will be on SOAP-based specifications; meaning specifications that define SOAP header extensions [SOAP1.2].
2. Composability Patterns

2.1 Some Properties of Composability

This section analyzes three ways that two different specifications (S1 and S2) can compose with one another. The objective is to make their implementations operate jointly or cooperatively. This exercise leads to defining three forms of composition:

- **Supportive** composition,
- **Constitutive** composition,
- **Functional** composition

Before this, it is helpful to distinguish two basic ways a specification (S1) can relate to another specification (S2). These can be defined as properties affecting the composability of S1 and S2.

- **Closed composability.** The specification S1 composes with S2 in a closed way when S1 explicitly refers to or requires S2 in a way that excludes alternatives to S2.

- **Open composability.** The specifications S1 composes with S2 in an open way when S1 explicitly refers to S2 but in non-exclusive terms, or when S1 only relates abstractly to features that can be specified by S2, as well as by an alternative to S2 (say S3). Such a composition is usually done in form of a binding to these abstract features. Replacing S2 with another specification S3 would not require changes in the main body of the specification S1, but only the introduction of another binding for S3.

The use of extensibility points in XML schemas is an example of open composability practice, where the type of an element is abstracted (xs:any and anyAttribute). If this element is described in another schema, and if its type was explicitly referred to in the primary schema, the latter would be exclusively tied to the former, and these schemas (understood here as elements of specifications) would compose in a closed way. There is no judgement here about which way is better.

Although these different ways S1 can relate to S2 may not effect how their implementations cooperate, they have significant consequences on the ability to use alternatives (or their ability to compose with other specifications). Open and closed composability are illustrated and discussed later in this study.

- **Colliding Features.** A specification (S1) may specify a feature (f1) that is functionally equivalent to (f2) specified by another specification (S2). When S1 and S2 are composed, these two features are colliding: an implementation could achieve equivalent functions by using either one. Implementing both
would lead to a conflict.

It is expected that some features collide when composing specifications. Feature collisions are discussed later in this study.

2.2 Supportive Composition of Specifications

A specification (S2) is said to be supportive of a specification (S1) if it specifies an artifact - data structure, concept or function - that plays an enabling role for S1 to use. The standardization of that artifact is not critical, but it does have benefits: it allows implementors to use tools that support the standard and it increases the portability and reusability of related artifact representations.

For example, a business document standard S1 describes the structure of XML documents, the content of which needs to conform to some rules. Such rules could be expressed informally in S1 (and implemented by developers as they wish), or S1 may require that they be implemented in a popular representation (XML schema, XPath / XSL, RuleML, etc.) for better portability and tool support. Such a standard rule representation is not critically required by S1: it would only play a supportive role. Composability of S1 with one of these representations may be either open or closed:

- The open-composable way: By stating these rules in an abstract way and not mandating a particular representation, S1 would remain open-composable with any of the representations mentioned above. A binding to one of them may be proposed by S1, not exclusive from other bindings.
- The closed-composable way: By mandating a particular representation for these rules – say RuleML - at the exclusion of others, S1 is composable with RuleML but in a dependent and exclusive way.

Open composability of a specification S1 with a supportive specification S2 is appropriate in the following cases:
- S2 is not stable or finalized, yet there is value in releasing S1 before S2 has been completed.
- There might not be a consensus on which supportive specification to use. Different user communities may prefer to select different supportive specifications for the same abstract features.

Closed composability with a supporting specification removes any indeterminism about how to represent composed features, therefore reducing interoperability hazards, without requiring further profiling. But core updates to the specification will be required if an alternative specification is preferred later.

Example 1: WS-Addressing (not finalized at the time this is written) will be a
supportive specification for several other specifications, in particular as a support for a callback mechanism. Without WS-Addressing, specifications describe their reply address in an ad-hoc way. This is the option taken by WS-Reliability [WSR1.1] for the "ReplyTo" element.

Example 2: In WS-Reliability the reliability agreement (RM-Agreement) is described abstractly, yet in a way that is precise enough for defining the various quality of services in reliable messaging. A generally agreed-upon representation for agreement, capabilities or policies would certainly improve the ability of WS-Reliability implementations to share these agreements, yet is not critical to the implementation of WS-Reliability features. It would be a supportive specification for WS-Reliability. At the time a stable standard emerges for these – which was not the case when WS-Reliability 1.1 was designed, a normative binding can be released for the RM-Agreement.

2.3 Constitutive Composition of Specifications

In this pattern of composition, specification S1 is making use of specification S2 so that S2 specifies a feature component that is essential to an implementation of S1. The selected feature (of S2) is essential (to S1) in the sense that agreeing on a precise representation of it is vital to implementing and deploying S1. The typical example is of layered protocols.

For example, a specification (S1) dealing with SOAP header processing often defines how it makes use of an underlying protocol (S2) (e.g. HTTP, SMTP). Such a lower layer protocol would not be considered “supportive specification”, because it is not an accessory to implementations of S1. It is essential in an operational sense (here, interoperable implementation of S1).

The composition of S1 and S2 may be open or closed, as previously defined. It is achieved by a binding that may be part of the main body of specification S1, or it may be specified separately.

In some cases the binding S1-S2 is not described in S1. This does not mean that features of S2 are unessential to S1, or can be left to the developer's choice. It may just mean that these features and their binding to S1 have already been specified somewhere else. For example, a SOAP-based specification may just rely on the standard SOAP-HTTP binding as defined in the original SOAP 1.1 specification.

Open composability is achieved by a design that clearly delimits in S1 the boundaries of the features specified by S2 or a similar specification, while S1 requirements refer to these features in an abstract way, rather than in S2 terms.

For example, the SOAP transport binding framework, as well as SOAP message exchange patterns [SOAPadjuncts], allow for making abstraction of the
underlying protocols in a SOAP-based specification (S1). It is not only necessary for any implementation of S1 to include an implementation of such an underlying protocol, but the specification of the latter (S2) is essential to the interoperability of S1 implementation.

### 2.4 Functional Composition of Specifications

This type of composition is about the ability of two specifications to be used together in a common deployment context where both sets of functions are required. Functional composition requires the respective implementations to be able to perform jointly – at a minimum to not impede each other's functionality. Unlike previous composition patterns, it does not necessarily translate into explicit bindings or references from one specification to the other. Functional composition has more to do with the way the respective implementations can cooperate, usually within a common environment implicitly assumed, such as the SOAP processing model [SOPA1.2].

The SOAP processing model is designed so that SOAP headers can be consumed by different SOAP nodes implementing different SOAP-based specifications. Because the functions implemented by these specifications process the same messages – by inserting (for transmission) then consuming (on reception) SOAP headers – it is possible to model their composition as a functional composition in the mathematical sense.

For example, WS-Reliability and WS-Security [WSS1.0] do not reference each other, but are functionally composable. The effect of WS-Reliability on the message \( m \) on the sending side – adding a header - can be modeled as a function that transforms the message: \( r(m) \). Similarly, the message resulting from adding to \( m \) a wsse:Security header can be represented as: \( s(m) \). When both reliability and security functions are applied in this order on sending side, the resulting message is:

\[
m' = s(r(m))
\]

When implemented by two different SOAP nodes, this composition corresponds to a sequence of processing steps \( \{ R + S \} \) on the message \( m \):

\[
(m) \rightarrow R + S \rightarrow (m')
\]

Reverse functions associated with the consumption of these SOAP headers will generally be applied on the receiving side, in the reverse order:

\[
m = r'(s'(m'))
\]

In this pattern of functional composition, the processing of the security function is said to be *nested* inside the processing of the reliability function.
Although this modeling ignores some aspects of SOAP processing (including faults, and other side-effects) by exclusively focusing on the effect on the message, it provides some insights on the composability of such processing. These are studied in more detailed in Appendix A.

In the case of WS-Reliability and WS-Security, the semantics of the composition will be different depending on the order of the processing:

• If Security is “nested” in Reliability processing (processed after Reliability on sending side and before on receiving side) then the integrity of Reliability headers can be guaranteed.
• If Reliability is “nested” in Security processing, duplicate messages or out-of-order messages will not uselessly be submitted to security processing, but the integrity of Reliability headers is not guaranteed.
3. Other Perspectives on Composability

3.1 The Fault Perspective

In the SOAP processing model, a SOAP Fault is generated when the processing of a headers fails. The faulting may then appear at different stages in the message processing. Because the requirement is to not process further a faulted message, the effect of a fault will depend on the order of this processing – and of the functional composition corresponding to this processing, as shown in these two examples:

(a) A SOAP node is logging received messages. The semantics of the logging function depends on where logging is done in a SOAP processing chain. If some SOAP extensions are processed after the logging, the log may also contain failed messages that will never reach their ultimate destination.

(b) In a composition where security processing is nested within reliability processing, a failed signature validation will cause the message to not be acknowledged. A useless resending mechanism may be triggered if the security fault is not properly interpreted by the reliability processing. This would not be the case if these functions were composed otherwise.

The composability of a SOAP-based specification is enhanced by the ability to interpret external faults. For example, error handling in WS-Reliability 1.1 identifies two types of errors: unrecoverable errors for which resending is useless, and recoverable errors for which resending maybe helpful. A failed signature validation should preferably be interpreted as unrecoverable error from the reliability processor viewpoint. If not, useless resending will occur, increasing the overhead of message reliability.

For better composability, a specification must decide how to handle 3rd party faults, yet is not supposed to know their detailed semantics if one wants composability to remain open. There are two ways SOAP-based specifications can enable open composability from a Fault perspective:

1) By supporting a Fault binding mechanism. For example, if the reliability specification defines two generic faults: “recoverableFault” and “unrecoverableFault”, then a signature validation fault can be bound to unrecoverableFault.

2) By using a more refined severity scale to be used across SOAP-based specifications. For example, an external fault of severity level “fatal” would abort the resending mechanism in the reliability function, while “high” or “medium” would not.

The use of customized faults (non-SOAP Faults) in a specification will also affect composability. Such faults are only intended for processing within the scope of a
single specification: they will be transparent to the SOAP processing model and therefore to other specifications. If these faults need be escalated beyond this scope, SOAP faults should be generated. Using non-SOAP faulting in a composition gives a better control on whether the message should be processed further or not in spite of the faulting.

3.2 The Contract Perspective:

A SOAP-based specification enforces a contract that takes effect between the two processing nodes that implement this specification, on sending and receiving sides. But more importantly, an implicit contract takes place between the parties that are before and after this processing chain, and the processing chain itself.

For example, persistent security is such a contract: WS-Security will guarantee integrity of a message at the time it is signed by a SOAP node, and until another SOAP node validates the signature on receiver side. Persistent security is then guaranteed between whoever (an application, or another node) passes the message to the Security producer role, and whoever gets the message (an application, or another node) just after it has been consumed by the Security consumer role. We call these applications or other nodes, the parties that contract with the Security function.

Depending on how specification-related functions are composed, their contracting parties will vary. For example, if Security is composed in a nested way with another function – say Reliability – then persistent security will no longer be guaranteed end-to-end between the originator (e.g. application) that sent a message, and the ultimate receiver. The contract supported by the Security specification will then be affected.
Appendix A: Properties of Functional Compositions

The processing of a message by a node implementing a SOAP-based specification (S1) can be modeled as a function that modifies the message (m) by adding related headers: m' = s1(m). Conversely, the implementation of S1 on the receiving side usually transforms the message (m') by removing those headers. To distinguish these two roles, we will use two functions for modeling the effect of S1:

- s1_p represents the effect of the “producer” role of S1 on a message (m). A SOAP node S1p will perform this function.
- s1_c represents the effect of the “consumer” role of S1 on a message (m). A SOAP node S1c will perform this function.

Consequently, when S1 alone is affecting a SOAP message, the resulting transformation will be a composition of the functions associated with these two roles of S1 on the message:

\[ m'' = s1_c ( s1_p (m)) \]

This composition corresponds to a sequence of processing steps \{ S1p + S1c \} on the message (m), that is usually performed by different SOAP nodes:

\[(m) --> S1p + S1c --> (m'')\]

This kind of composition is called here “end-to-end [functional] composition of S1 roles”. Usually, SOAP-based specifications clearly assign roles to the origin and destination end-points, while preventing intermediary nodes from preempting these roles. For example, WS-Security requires that a security header block without a specified actor or role may be processed by any node, but must not be removed prior to the final destination or end-point. WS-Reliability makes it clear that only two nodes (sending and receiving processors) can implement the reliability contract, and that intermediaries have no qualitative role that could affect this contract.

Message-invariant composition: If the end-to-end composition of two roles – a producer and a consumer – of a specification S1 is so that in their implementation, the consuming side is removing all headers added by the producing side without affecting the rest of the message, then these two roles can be seen as performing inverse functions: \( s1_c = s1_p^{-1} \), and:

\[ s1_c ( s1_p (m)) = m. \]

In such a case we will say that the composition is message-invariant, and that the end-to-end composition of S1 roles (or of their implementations) is message-invariant. This also applies to the case where an S1 implementation does not create new headers, but only adds elements to an existing header block in its producer role, and removes these elements in its consumer role.
Example: An implementation of WS-Security has a producer role (function S) on the sending side for message (m). It will result in a new modified message m’ with a new wsse:Security header. On the receiving side, this implementation of WS-Security has a consumer role: as a SOAP node it will consume this header. This normally results in removing the header block and forwarding the message in the same state as it was before it was processed, if no Fault is generated. When an implementation of WS-Security behaves that way, then the composition of its roles is message-invariant. Note that the effect of an implementation may be an update of the header block, instead of an addition or removal (e.g. add or remove new elements related to security tokens, within a wsse:Security header).

Commutative composition: When the implementations of two SOAP-based specifications S1 and S2 are composed, their composition is commutative if the processing order does not matter. Assuming that S1 and S2 have each both a producer role and a consumer role, then S1 and S2 are commutative on their producer role if:

\[ s_1^p( s_2^p(m)) = s_2^p( s_1^p(m)) \]

Meaning that the order of processing of the headers related to these specifications (producer side) does not matter to the consumer roles. Concretely, this means that SOAP nodes implementing these specifications may be serialized in any order. Commutativity on the producer roles will entail commutativity on the consumer roles and vice-versa, provided that each one of S1 and S2 implementations is message-invariant (on its end-to-end roles composition).

However, when the header updated or added by a specification depends on other headers – as it is often the case with digital signatures – then commutativity is no longer a property of the specifications S1 and S2 involved, but depends on which features of S1 and S2 are used and how. For example, when both WS-Reliability (function \( r_p \) for the producer role) and WS-Security (fonction \( s_p \)) are applied, the resulting composition on producer side is commutative only if reliability headers are excluded from signatures and digests.

Nested composition: In a nested composition of two specifications S1 and S2 which both support two roles (producer and consumer), the functions associated with the producer are processed on a message in reverse order of the functions associated with the consumer. An S2 implementation is “nested” in an S1 implementation if S1 is applied before S2 on producer side, and S1 after S2 on the consumer side. The end-to-end sequence of processing amounts to the following composition:

\[ s_1^c( s_2^c( s_2^p( s_1^p(m))))) \]

Such a composition is message-invariant if:

\[ s_1^c( s_2^c( s_2^p( s_1^p(m)))) = m \]

When some implementations of specifications S1 and S2 are already individually
message-invariant, then it is easy to see that their nested composition is also message-invariant. Consider a message-invariant composition, i.e. involving two or more implementations of some specification(s), with both producer and consumer roles. It is always possible to "insert" such a processing chain within an end-to-end composition of S3 on its producer and consumer roles:

\[
s3 \circ (s1 \circ (s2 \circ (s2 \circ (s1 \circ (s3 \circ (m))))) = s3 \circ (s3 \circ (m))
\]

**Example:** If a node implementing WS-Security is generating an enveloped signature, then a message-invariant processing composition generating and consuming extra headers - like WS-Reliability headers or another wsse:Security header - can always be nested before this signature is validated, in spite of the signature being not valid all the time the message contained extra headers.

However, nesting a message-invariant producer-consumer roles composition within another message-invariant composition cannot be done without consideration for other functions (intermediaries) possibly occurring between the producer and consumer roles of the nested specification.

For example, consider the following processing sequence that composes two implementations S1 and S2 in a nested way, on their producer (p) and consumer (c) roles:

1. \(S1p + S2p + S2c + S1c\)

This corresponds to the functional composition: \(s1 \circ (s2 \circ (s2 \circ (s1 \circ (m))))\)

Assuming S1 and S2 implementations are message-invariant, a third specification S3 – even if itself message-invariant - cannot be composed in the following way:

2. \(S1p + S3p + S2p + S3c + S2c + S1c\)

unless S2 and S3 are commutative either on their producer role, or on their consumer role. A case where these would not be commutative, is when S3p is generating an enveloped signature. Otherwise, a correct composition would be:

3. \(S1p + S3p + S2p + S2c + S3c + S1c\)

WS-I Basic Security Profile 1.0 [BSP1.0] is prohibiting enveloping signatures, discouraging enveloped signatures and promoting detached signatures. This position can be understood from the functional composability viewpoint:

- Functions implementing detached signatures have the greatest chances of being composable in a commutative way (and always are, if signature elements are not themselves signed), meaning that they put the least amount of constraints on the processing order – and to the use of intermediaries.
- Enveloped signatures only allow for non-commutative compositions, yet if S2p in (ii) was implementing such a signature, the message processing would not break before its consumer role is processed (S2c) meaning that the processing of other headers is still possible. This somehow limits the impact of a bad composition, and makes error detection easier.
As for enveloping signatures, they would clearly make a composition like (ii) above impossible if the signature is done by S2p on a header that must be recognized by S3c. In addition they would generally prevent the consumption of any enveloped header taking place before role S2c, making it harder to detect why the composition failed. This feature has the poorest composability prospects.

As a conclusion, the functional composability of specifications depends not only on which features and options are exercised, but on how implementations are deployed. Although the functional modeling presented here cannot render all aspects of some processing schemes (e.g. cases where some intermediaries need to “check” headers without removing them, fault handling), it captures some essential aspects of composability.
References

[BSP1.0] “Basic Security Profile 1.0

http://www.w3.org/TR/2003/REC-soap12-part2-20030624/

http://www.w3.org/TR/soap12-part1/


[XMLSig] “XML Signature Syntax and Processing”, W3C recommendation.
http://www.w3.org/TR/xmldsig-core/