Composability of Specifications: Patterns and Properties

(DRAFT)
V0.3

Authors:
Jacques Durand
Alan Weissberger

Table of Contents

1.1 Supportive Composition of Specifications...............................3
1.2 Constitutive Composition of Specifications...............................4
1.3 Functional Composition of Specifications.................................5
2.1 The Fault Perspective..............................................................10
2.2 The Contract Perspective..........................................................10
1. Introduction

The ability to compose specifications is becoming more critical as the many functions that can cooperate in an infrastructure or in a large application are being standardized, while application integration is still a major IT issue. It may be surprising that, although the notion of composition in the area of standards has increasingly been given attention, no more in-depth analysis of what really makes a specification “composable” has been done.

To be more precise, the ability to compose – also named “composability” - is as much a property of implementations as it is a property of specifications. Even if a specification allows for composability, two implementations of it 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, 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 a specification, rather than of the composability of this specification. This is true in particular for functional compositions defined later.

This investigation of composability identifies what properties make specifications and their implementations composable. Its focus will be on – though not restricted to – SOAP-based specifications, meaning specifications that define SOAP extensions.
2. Some Composability Patterns

This section analyzes three ways a specification S2 can compose with a specification S1, so that their implementations can operate jointly. This 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 a specification S2. These can be defined as properties affecting the composability of S1 and S2:

- "Closed" composability. The composition of specifications S1 and S2 is closed when S1 explicitly refers to or requires S2 in a way that makes the use of S2 mandatory in an implementation of S1, at least in cases where features described by S2 are required (even if alternatives to S2 exist).

- "Open" composability. The composition of specifications S1 and S2 is open when S1 relates abstractly to features that can be specified by S2, in such a way that the specification remains open to binding these features with an alternative to S2 (say S3). If S1 refers to S2, it is done in form of a binding to these 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.

Although these different ways S1 can relate to S2 make no difference regarding how their implementations may cooperate, they have significant consequences on the ability to not compose with each other (or their ability to compose with other specifications instead).

1.1 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 (is used by S1), yet the standardization of which is not critical. Although not necessary, this standardization has 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 above representations. A binding to one of them may be proposed by S1, but is 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 of finalized, yet there is value in releasing S1 before S2 is ready.
- There might not be a consensus on which supportive specification to prescribe, as different user communities may prefer to use different supportive specifications for the same abstract features.

Closed composability with a supporting specification S2 removes any indeterminism about how to represent composed features, therefore reducing interoperability hazards, without further profiling needed. But core updates to the specification will be required if an alternative specification is preferred later.

Example 1: The use of extensibility points in XML schemas is an example of open composibility, where the type of an element is abstracted (xs:any and anyAttribute). This is the option taken by WS-Reliability 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.

1.2 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 critical to an implementation of S1. The feature is essential in the sense that agreeing on a precise
representation – hence a specification (S2) of it - is key to implementing and deploying S1. The typical example is of layered protocols.

For example, a specification (S1) about SOAP processing usually defines how it makes use of an underlying protocol (S2) (e.g. HTTP, SMTP). Such a protocol would not be considered “supportive specification” because it is not accessory to implementations of S1. It is essential to an operational (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 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 modular design that clearly delimits in S1 the boundaries of the artifacts specified by S2 or a similar specification, while S1 requirements refer to these artifacts in an abstract way, rather than in S2 terms.

Example 3: SOAP 1.2 introduces the notion of SOAP Message Exchange Patterns. It is possible in any specification based on SOAP 1.2 to make abstraction of underlying protocols, even of their MEPs (like HTTP Request-response). Yet it is clear that implementations of a specification based on SOAP 1.2 that use these MEPs must use an underlying protocol. A binding to this underlying protocol must be defined, including how SOAP MEPs map to it. Although this binding may define an open composition – not exclusive of other bindings - to a particular transport protocol, such a protocol is constitutive to any implementation of the SOAP-based specification. The specification of this protocol is said to be constitutive to the SOAP-based specification.

1.3 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 functions are required. This means for their respective implementations to be able to perform jointly – at a minimum to not impede each other functionality. Unlike previous composition patterns, it does not necessarily translate into explicit bindings or references from one specification to the other. It has more to do with the way their implementations can compose, usually within an environment implicitly assumed,
such as the SOAP processing model.

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, then consuming SOAP headers – it is possible to model their composition as a functional composition in the mathematical sense. 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.

The processing of a message by a node implementing S1 can be modeled as a function that modifies the message (m) by adding related headers: \( m' = s_1(m) \). Conversely, the implementation of S1 on the receiver side usually transforms again the message (\( m' \)) by removing these headers. To distinguish these two roles, we will use two functions for modeling the effect of S1:

- \( s_{1_p} \) represents the effect of the "producer" role of S1 on a message (m) (adding SOAP headers)
- \( s_{1_c} \) represents the effect of the "consumer" role of S1 on a message (m) (consuming SOAP headers)

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:

\[
m'' = s_{1_c}(s_{1_p}(m))
\]

This composition corresponds to a sequence of processing steps \( \{ S_{1p} + S_{1c} \} \) on the message (m), that is usually performed by different SOAP nodes:

\[
(m)\rightarrow S_{1p} + S_{1c} \rightarrow (m'')
\]

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 endpoint. WS-Reliability makes it clear that only two nodes (producer and consumer) 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: \( s_{1_c} = s_{1_p}^{-1} \), 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 sender side for message (m). It will result in a new modified message m’ with a new wsse:Security header. On the receiver 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 :

s1_p ( s2_p (m)) = s2_p ( s1_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:
Such a composition is message-invariant if:
\[ s_1c \ (s_2c \ (s_2p \ (s_1p \ (m)))) = m \]

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:
\[ s_3c \ (s_1c \ (s_2c \ (s_2p \ (s_1p \ (s_3p(m)))))) = s_3c \ (s_3p(m)) \]

**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:

(i) \( S_1p + S_2p + S_2c + S_1c \)

This corresponds to the functional composition: \( s_1c \ (s_2c \ (s_2p \ (s_1p \ (m)))) \)

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

(ii) \( S_1p + S_3p + S_2p + S_3c + S_2c + S_1c \)

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:

(iii) \( S_1p + S_3p + S_2p + S_2c + S_3c + S_1c \)

It is not surprising that WS-I BSP 1.0 is prohibiting enveloping signatures, discouraging enveloped signatures and promoting detached signatures:

- 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. It is then wise to prohibit this feature from a composability perspective.

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.
2. Other Perspectives on Composability

2.1 The Fault Perspective

2.2 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.