Constructing Secure Service Compositions with Patterns

Luca Pino
School of Informatics
City University London
London, United Kingdom
e-mail: Luca.Pino.1@city.ac.uk

George Spanoudakis
School of Informatics
City University London
London, United Kingdom
e-mail: G.E.Spanoudakis@city.ac.uk

Abstract— **In service based applications, it is often necessary to construct compositions of services in order to provide required functionality in cases where this is not possible through the use of a single service. Whilst creating service compositions, it is necessary to ensure not only that the functionality required of the composition is achieved but also that certain security properties are preserved. In this paper, we describe an approach to constructing secure service compositions. Our approach is based on the use of composition patterns and rules that determine the security properties that should be preserved by the individual services that constitute a composition in order to ensure that security properties of the overall composition are also satisfied. Our approach extends a framework developed to support the runtime service discovery.**

Software service security; secure service composition

I. INTRODUCTION

The problem of constructing service compositions has received considerable attention in the literature (e.g. [25][26][27]). This is because service composition is necessary when a functionality required by a service based application (SBA) cannot be provided by any single service.

Existing approaches focus on the generation of compositions that provide required functional and quality of service properties, and/or on checks of the compatibility of the behavioural models of services that are to be composed. The satisfaction of required functional and QoS properties are necessary conditions for generating usable service compositions. However, they are not sufficient when a service composition needs also to satisfy given security properties. Addressing security properties in service composition has not received significant attention in the literature and – to the best of our knowledge – existing work (e.g., [9][10]) does not offer adequate solutions to this problem. To address this gap, in this paper, we describe our approach to constructing secure service compositions.

Our approach is based on the use of *composition patterns* and *security implication rules* to determine which security properties the individual services that constitute a composition should have in order to ensure that security properties required of the overall composition are satisfied. More specifically, the composition patterns provide abstract and parametric specifications of service workflows. Pattern specifications describe the flows of control and data within a workflow and preconditions determining when a pattern can be applied. The security rules express the consequences that, particular actions of individual services and service compositions, have on security properties. For example, one of the rules used in our approach expresses that if the confidentiality of the inputs to a given activity in a workflow must be preserved and this activity is bound to a service that produces an output containing information about the input, the confidentiality of the service output must also be preserved. The security rules express formally proven relations between security properties.

During the composition process, the security rules are used to determine the security properties that need to be satisfied by individual services, in order to guarantee the security properties required of the entire composition. The security properties identified for individual services are fed into a discovery tool, which subsequently finds suitable candidate services for the composition.

Our approach extends a discovery framework that has been developed at City University to support the discovery of services at runtime [16]. Originally, this framework supported the discovery of single services based on criteria regarding the interface, behaviour and quality of services, in a *reactive* or *a proactive* mode, i.e., when a need for finding a service at arises (reactive mode) or continually in order to maintain up-to-date sets of candidate services that could be used to replace the constituent services of an SBA when any of these services fails (proactive mode). The work that we describe in this paper extends the capabilities of this framework by enabling the construction of secure service compositions.

The rest of this paper is structured as follows. Section II outlines a scenario for service composition, which is used in the rest of the paper to exemplify our approach. Section III gives an overview of the overall service discovery framework within which our approach is used. Section IV and V describe the composition patterns and the security reasoning rules underpinning our approach, respectively. Section VI describes the service composition algorithm and gives an example of applying it. Finally, Section VII overviews related work and Section VIII provides some summarising remarks and outlines directions for future work.

II. SCENARIO

A scenario that we use in the rest of the paper to exemplify our approach involves a financial market SBA offered by a *stockbroker* to *stock investors* wishing to buy and/or sell stocks in different stock exchanges. Stock investors don't have direct access to exchanges and must rely on stockbrokers for carrying trades. The stockbroker's SBA provides a set of useful operations to stock investors to enable trades. Some of these operations are based on services available from third party service providers (e.g., provision of financial data to enable trading decisions). As some of these operations might become unavailable at runtime, however, the stockbroker's SBA should incorporate mechanisms for searching and replacing such services and operations, if the need arises at runtime.

Suppose, for example, that the stockbroker SBA uses an operation called *getStockHeadlines* provided by a third-party service S_1. Given a stock symbol (*Symbol*) and some customer details (*CustData*) as inputs, this operation returns a set of recent news related to the identified stock and uses the customer details to get paid for the offered service. The output of *getStockHeadlines* is an array of *StockNews*, containing the headline, time and source of each news item about the input stock. The required security conditions for this service are that *CustData* and *Symbol* should be confidential during the transmission, to avoid external parties seeing who is requesting information about particular stocks and the credentials used for news payments.

When S_1 becomes unavailable, the stockbroker SBA must look for replacement services for it. If these are no such individual services, an alternative could be to use the composition C shown in Fig. 1. This composition uses service S_2 that provides a different operation to retrieve stock news, called *fetchNewsHeadlines*. This operation returns an array of *MarketNews* containing all the financial news, indexed by a stock's ISIN (i.e., a different ID from Symbol), after receiving proof of prior payment by the requesting party (*PaymToken*). In fact, to be able to use an operation of S_2, a customer must first pay through the operation *Payment*. This operation is offered by S_2 and generates a *PaymToken* (output) from *CustData* (input) after a payment has been made.

The output *MarketNews* must, then, be filtered through the Stock ID. For this reason the composition includes also a *getStockDetails* operation that returns a *StockMap*, containing the *ISIN* corresponding to *Symbol*.

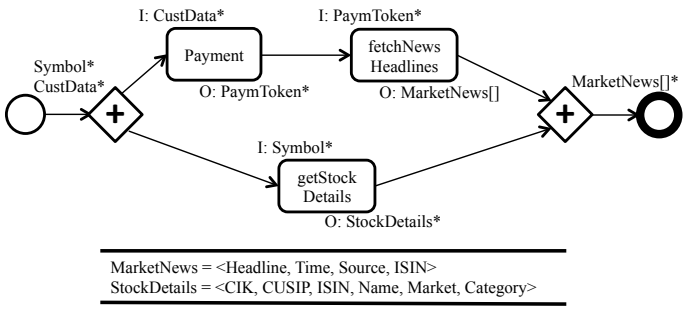

Fig. 1. Composition C – Data marked with (*) should be confidential

The composition of Fig. 1 creates requirements for new security properties that arise from the original security conditions. In particular, since *CustData* must be confidential, then its related payment token, *PaymToken*, must also be confidential. Furthermore, to guarantee that *Symbol* remains confidential, *StockMap* (and *ISIN*) should be confidential as well, since they can also provide the same information about the stock that the investor. Furthermore, the filtered news should be pruned of the *ISIN* information, or the *ISIN* should be confidential.

III. DISCOVERY FRAMEWORK

A. Overall Architecture

Our approach to secure service composition is part of the discovery framework shown in Fig. 2. This framework accepts service discovery queries from SBAs, and finds services in external service registries that satisfy the conditions of the queries. Queries can be submitted for execution in reactive (PULL) or proactive (PUSH) mode.

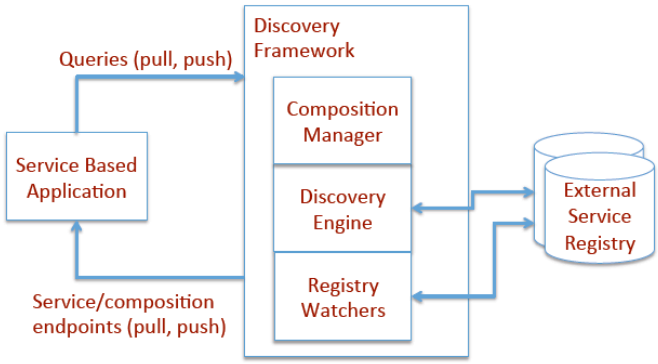

Fig. 2. Discovery Framework

The framework includes a Discovery Engine that is responsible for the retrieving individual service descriptions data from external service registries and matching them with the queries. It also includes Registry Watchers which poll external registries periodically to check if there are new services or amended service descriptions that would alter the candidate sets of services that are maintained for queries executed in proactive mode. The new component of the framework that is based on the work of this paper is the Composition Manager. This component is responsible for the creation of secure service compositions to meet queries in cases where the latter do not match with any single service.

B. Discovery process

The overall discovery process realised by the discovery framework is shown in Fig. 3. The discovery process starts when the Discovery Engine receives a query that should be used for discovering replacement services for one of the partner services of an SBA. Queries are expressed in an XML based language, called *A-SerDiQueL*. Following the parsing of a query, the parts of it that refer to security related discovery criteria (referred to as *Ce query* in Fig. 3) are separated from the parts referring to other functional and quality discovery criteria (referred to as *N query* in the figure). This distinction is

necessary as the part of the query that refers to security related discovery criteria is used in order to identify the composition patterns that could be applied in identifying service compositions that can ensure these criteria (cf. the activity *Identify CPatterns* in the process).

Subsequently, the N query and the composition patterns are either sent to the discovery engine for an one-off execution (if the execution mode of the query is PULL) or are subscribed to it, for continual executions if the execution mode of the query is PUSH. In PUSH mode, multiple executions may be triggered by changes in the descriptions of services already identified as possible matches with a query or due to the emergence of new services in registries fitting with the query.

In both the PUSH and the PULL mode of query execution, the discovery engine executes the received query at least once and returns any services and service compositions that match the discovery criteria of the query (see the activity *Execute N-Query/Cpatterns*). Any services and/or service compositions that match with the discovery criteria of the query at this stage are used to update a *Candidate Service Set*. This set is used as a cache of replacement services for the partner service that was associated with the query in the first place and any subsequent service replacement request will retrieve the first service from this set. In the case of candidate service compositions, the framework generates a virtual service pointer that can be used by the SBA to invoke the composition through the framework.

It should also be noted that the initial formation of the *Candidate Service Set* is followed by ordering the elements of this set in descending order of the degree of match that they have with these criteria (see the activity *OrderRS wrt Ce*). Following this stage in the overall process, any candidate service/service composition that does satisfy the security related criteria is removed from the *Candidate Service Set* and a new discovery process is initiated in order to try to identify further services that could meet first the non security criteria of the query and then the security related criteria. The reason for re-attempting to find services/service compositions meeting the criteria of the query is because there is a possibility that new services might have been published in the service registry meanwhile.

Certain parts of the overall discovery process described in Fig. 3 can be also triggered by events other than a request for the execution of a query. These events are:

- service replacement requests resulting in removal of the first service in the *Candidate Service Set* in order to use it in the SBA;
- publications of new security descriptions (expressed by certificates as we explain in Sect. III.C) for one of the services in the candidate service set that should trigger the re-evaluation of the security related criteria for a candidate set that has been built for a query executed in PUSH mode and possibly a re-ordering of this set; and
- changes in the descriptions of services in the service registries or the publication of new services in them that can lead to the execution of the non security related parts of queries executed in the PUSH mode in the first place and potentially the re-execution of the security related parts if the candidate services set of the query would need to be altered given the results produced by the non security related part of the query.

C. *Query language*

The queries of the discovery framework are expressed in *A-SerDiQueL*, an XML-based language that allows the specification of interface, behavioural, QoS and security conditions about the services to be discovered. *A-SerDiQueL* is an extension of *SerDiQueL* (see [24] for a detailed account) that we have developed to support the specification of security conditions as part of service discovery queries.

The specification of security conditions in *A-SerDiQueL* assumes that the security properties of services are described by security certificates, called *asserts*. An assert certificate certifies that a given security property is preserved by a service. In addition to specifying the relevant security property, certificates may include descriptions of the *evidence* justifying the certification of the property, the *authority* that has issued the certificate, and the *validity period* of the certificate.

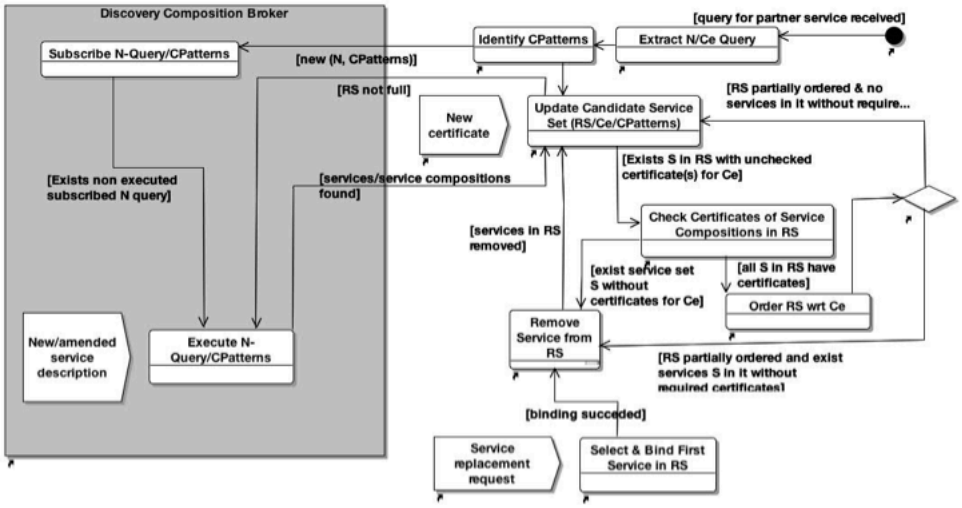

Fig. 3. Overall Discovery Process

TABLE I
A-SERDIQUEL QUERY EXPRESSING A CONFIDENTIALITY CONDITION

```
<AssertQuery name="A1" type="HARD">
  <LogicalExpression>
    <Condition relation="EQUAL-TO">
      <Operand1>
        <AssertOperand facetName="Assert"
           facetType="Assert">
          //AbstractASSERTStructure/Property
        <AssertOperand>
      </Operand1>
      <Operand2>
        <Constant type="STRING">
          Confidentiality(O, X)
        </Constant>
      </Operand2>
    </Condition>
  </LogicalExpression>
</AssertQuery>
```

Certificates are represented in XML according to a specific XML schema, and are published in service registries as a special facet of service descriptions. An example of an *A-SerDiQueL* query regarding the confidentiality of the input X of an operation O of a service S is shown in Table I.

IV. SECURE COMPOSITION PATTERNS

Our approach to constructing secure compositions of services is driven by *secure composition patterns*. A secure composition pattern is a template specifying a service orchestration workflow with activities that can be bound to concrete service operations. Patterns describe the control and data flows connecting the activities. For each activity placeholder in a pattern, it is possible to automatically build a service discovery query in order to find a service operation to instantiate the activity. An example of a simple workflow is the one obtained by invoking sequences of services in a chain to transform an initial input into a required final output [1][2].

Two examples of secure composition patterns are provided in Fig. 4. The first one is the *Secure Sequence Pattern* (SSP). This pattern represents an elementary control flow with two activities, A and B, that must be executed one after the other in this specific order (the order of A and B is represented as a solid arrow in the picture). The data flow of this pattern is summarized in the IO dependencies of the pattern: the input passed to A (in_A) must be a subset of the available input (IN) described in the query, the input to B (in_B) should be a subset of IN together with the output of the first activity (IN + out_A) and the final output (OUT) should be a subset of the output of B (out_B). Note that this is just one of the possible data flows for this workflow; another one can require that the final output should be a subset of the output of A and B. To represent alternative data flows in this case, another variant of the SSP pattern with the same control flow but different data flows would be required in our approach.

The second example is the *Secure Parallel Filter Pattern* (SPFP). This pattern specifies the execution of two activities, A and B, in parallel, and filters their outputs by an attribute value. The data flows of the pattern in Fig. 4 specify that the output of A (out_A) should be a list of some data type $Type_X$ that is to be filtered and B should return as output (out_B) the value used to filter through an attribute of $Type_X$. The final output of the pattern (OUT) is then filtered from these two outputs by an internal activity (filter) which is part of the pattern.

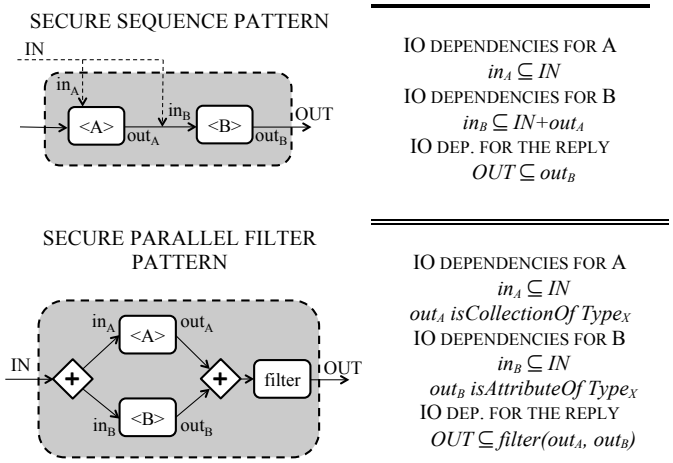

Fig. 4. Examples of secure composition patterns

The composition patterns are expressed internally in OWL-S [20]. Our pattern representation schema uses the Process Model portion of OWL-S that allows to specify service Inputs/Outputs, preconditions and results, as well as compositions of services. Each pattern is described as a `CompositeProcess`, and contains: (a) control flows expressed by `ControlConstructs` (e.g. `Sequence`, `Split-Join`, `Choice`) that contain activity Perform elements, and (b) data flows specified as `Bindings` between variables of different activities. Note that in our specification if a `Binding` has a variable X in the `toVar` element and a variable Y in the `valueSource/ValueOf`, it implies that X is a subset of Y (X ⊆ Y) unlike standard OWL-S which assumes that X = Y).

Table II shows the specification of the SPFP pattern in OWL-S. This specification describes the control flow of the two activities (rows 18-27 for A and 28-37 for B) through the `Split-Join` element (rows 17-38) and the dependencies on the inputs of A (rows 20-26) and B (rows 30-36). Furthermore, it describes the final output OUT (i.e. `ProcessOutput` in row 6) as a subset of the output of a `BasicFilter` activity (rows 7-10). This activity is part of an internal library of available data transformation services (or aggregators). These data transformation services are basic pieces of code that are called from a pattern as a form of data mediation. Since web services data is transmitted in XML, a data transformation service can usually be encoded in few XSLT lines, with some placeholders that will be instantiated during the pattern instantiation.

Table III shows the encoding of the `BasicFilter` used in the SPFP pattern. This is based on the XSLT identity transform [22], and it is applied on the $Type_X$ list to filter it. In particular during the instantiation of the pattern, the `$basicType` in the filter is instantiated to $Type_X$ element name, `$attribute` to the attribute path and `$valueToFilter` to a parameter that receives the value from the `ValueToFilter` in the pattern. In

particular in the SPFP pattern the `ValueToFilter` is obtained from out_B, but it is also possible to use the `BasicFilter` in different patterns with other mappings.

TABLE II
SECURE PARALLEL FILTER PATTERN SNIPPET

```
<p:CompositeProcess rdf:ID="ParallelFilter">
 <rdfs:label>Parallel filter pattern</rdfs:label>
 <p:hasResult><p:Result>
  <p:inCondition rdf:resource="&expr;#AlwaysTrue" />
  <p:withOutput><p:OutputBinding>
   <p:toVar rdf:resource="#ProcessOutput" />
   <p:valueSource><p:ValueOf>
    <p:theVar rdf:resource="&aggreg;#Output_BF" />
    <p:fromProcess rdf:resource="#PerformFilter" />
   </p:ValueOf></p:valueSource>
  </p:OutputBinding></p:withOutput>
 </p:Result></p:hasResult>
 <p:invocable rdf:datatype="&xsd;#boolean">
  false
 </p:invocable>
 <p:composedOf><p:Sequence><p:components>
  <p:Split-Join rdf:parseType="Collection">
   <p:Perform rdf:ID="Perform_A">
    <p:process rdf:resource="#A" />
    <p:hasDataFrom><p:InputBinding>
     <p:toVar rdf:resource="#Input_A" />
     <p:valueSource><p:ValueOf>
      <p:theVar rdf:resource="#ProcessInput" />
      <p:fromProcess
         rdf:resource="&p;#TheParentPerform" />
     </p:ValueOf></p:valueSource>
    </p:InputBinding></p:hasDataFrom>
   </p:Perform>
   <p:Perform rdf:ID="Perform_B">
    <p:process rdf:resource="#B" />
    <p:hasDataFrom><p:InputBinding>
     <p:toVar rdf:resource="#Input_B" />
     <p:valueSource><p:ValueOf>
      <p:theVar rdf:resource="#ProcessInput" />
      <p:fromProcess
         rdf:resource="&p;#TheParentPerform" />
     </p:ValueOf></p:valueSource>
    </p:InputBinding></p:hasDataFrom>
   </p:Perform>
  </p:Split-Join>
  <p:Perform rdf:ID="PerformFilter">
   <p:process rdf:resource="&aggreg;#BasicFilter" />
   <p:hasDataFrom><p:InputBinding>
    <p:toVar
         rdf:resource="&aggreg;#ListToFilter" />
    <p:valueSource><p:ValueOf>
     <p:theVar rdf:resource="#Output_A" />
     <p:fromProcess rdf:resource="#Perform_A" />
    </p:ValueOf></p:valueSource>
   </p:InputBinding><p:InputBinding>
    <p:toVar
         rdf:resource="&aggreg;#ValueToFilter" />
    <p:valueSource><p:ValueOf>
     <p:theVar rdf:resource="#Output_B" />
     <p:fromProcess rdf:resource="#Perform_B" />
    </p:ValueOf></p:valueSource>
   </p:InputBinding></p:hasDataFrom>
  </p:Perform>
 </p:components></p:Sequence></p:composedOf>
</p:CompositeProcess>
```

V. SECURITY PROPERTIES

Security properties are represented in our approach as relations on service operations and, in some cases, on the data exchanged by them (i.e., the input, output and persistent internal service data). These relations are expressed as properties in the form *<relation>(<operation>, <data>)*. The relation that expresses the confidentiality of the customer detail *CustData* of *getStockHeadlines* inputs, for example, is:

Confidentiality(CustData, getStockHeadlines).

A security property P is guaranteed by a service S if S can provide a certificate from an appropriate certification authority containing P. To be able to guarantee a security property in a composition, it may be necessary to check and propagate different properties. In particular, when there is some data involved in a security property, any action that an activity may perform on the secure data may lead to the need to check further properties.

To ensure the confidentiality of some input data, for example, it is necessary to ensure both the confidentiality of data transmission and storage confidentiality. The confidentiality in transmission is required on the input of an activity. However, if the output of the same activity also discloses any kind of information about the input, to preserve confidentiality the output should be transmitted confidentially as well. In a similar manner, if data that can disclose information about the input is stored, the confidentiality of storage should be assured. Such properties can be provided by some encryption mechanism, or by a certified property that the output/stored data doesn't disclose any information about the input (or that no data is being stored). Thus, the ability of a composition to guarantee security properties depends on the certified actions that the individual services perform. Such dependencies between security properties and service actions are expressed by *security implication rules*.

Back to the confidentiality example, suppose that we have some data D used in an activity A as input or output. A security implication rule can express that D should be transmitted confidentially, if it is derived from some data D' that are confidential for some activity A', or otherwise it must be certified that D does not disclose information about D'.

This can be specified by a rule (Rule-1) stating that if some data D', which is required to be confidential during transmission for an activity A', is used as input of another activity A, then the output of A should be confidential on transmission as well or there should be a certificate verifying that A does not disclose information about its input D'.

The security implication rules should be formally proven offline before used in our approach. This is required because

TABLE III
TEMPLATE FOR XSLT BASICFILTER

```
<stylesheet version="2.0"
    xmlns="http://www.w3.org/1999/XSL/Transform">
<template match="@*|node()">
  <copy><apply-templates select="@*|node()"/></copy>
</template>
<template match="{$basicType}">
  <if test="{$attribute} = {$valueToFilter}">
    <copy><apply-templates select="@*|node()"/></copy>
  </if>
</template>
</stylesheet>
```

the process of constructing rules at runtime, or of deriving dependencies between security properties of services, is computationally expensive. The process of constructing proofs of security implication rules is beyond the scope of this paper but interested readers may find examples of such proofs in [4].

The security implication rules are specified in *Situation Calculus* (SC) [5][1]. SC is a first order logic language that supports specifications and reasoning for domains that change dynamically. SC uses predicates called "fluents" that describe the state of a domain. Fluents are evaluated against sequences of actions, called "situations". In the SC model of the security implication rules we use:

- the fluent *next(A,A')* to specify that an activity A is followed by an activity A' in the workflow of a pattern
- the fluent *input(A,D) (output(A,D))* to specify that D is an input (output) of activity A
- the fluent *known(P, S)* to specify that the security property P is already known to be satisfied (certified) in situation S
- the property $conf_T(A,D)$ to specify that there is a certificate for action A stating that data D is confidential on transmission
- the property $derive_{ND}(D, D')$ to specify that data D is derived from D' and there is a certificate stating that D doesn't disclose any information about D'.

The control and data flows of workflows created from patterns, and descriptions of the services that instantiate the activities in them are also mapped in SC as initial fluents for the security implication rules. Furthermore, we represent the traces of the workflow as situations, where *currAct(A)* is the fluent describing that the reasoning step is on activity A. The reasoner navigates stepwise the workflow and, at each step, it collects the required security properties, expressed by the fluent *requires(P, S)*.

TABLE IV
EXAMPLE OF A SECURITY IMPLICATION RULE (RULE-1)

PRECONDITION AXIOMS
$poss(step(A),S) \leftrightarrow currAct(A',S) \wedge next(A,A')$
SUCCESSOR STATE
$currAct(A,do(\alpha,S)) \leftrightarrow \alpha = step(A)$
$requires(conf_T(A,D),do(\alpha,S)) \leftrightarrow [\alpha = step(A) \wedge output(A,D) \wedge input(A,D')$
$\wedge\ known(conf_T(A',D'), S) \wedge known(derive_{ND}(D',D)) \wedge A' \neq A]$
$\vee\ [\alpha \neq step(A) \wedge requires(conf_T(A,D), S)]$

Table IV shows the specification of the security implication rule *Rule-1* that we introduce above in SC. The first two formulae in the table are common rules enabling the SC reasoner to take into account one step at time. The actual formula for *Rule-1* is the third formula (the only difference from the previous explanation is the disjunctive condition in the formula, which is required to solve the "frame problem" in SC).

The instantiation of the activities in a composition pattern is based not only on the IO dependencies of the pattern but also on security properties. The security conditions are inferred from: (a) the security conditions of the query, (b) the security properties that the already instantiated services provide, and (c) the security implication rules. The rules provide a list of the security properties that must be certified for the single activities to satisfy the given security property for the whole composition.

VI. COMPOSITION PROCESS

Service compositions are obtained by constructing workflows through a step-wise instantiation of the composition patterns. When no single replacement service is found for a service S that needs to be discovered, then the discovery query that is associated with the service to be replaced (Q_S) is sent to the Composition Algorithm shown in Table V.

TABLE V
COMPOSITION ALGORITHM

Algorithm: *SecureComposition(Q_S)*
Input: Q_S query for required service
Output: WFSet – set of instantiated workflows

```
1  for each pattern Patt such that applicable(Patt, Q_S)=true do
2    Create the workflow WF* from Patt and put it in WFQueue
3    while there are more workflows in WFQueue do
4      Get the first WF in the WFQueue
5      Take an unassigned activity α in the WF
6      Build a query Q* for α from:
         • the IO dependencies of Patt
         • the conditions inferred by the SecRules
         • any instantiated part of Patt and Q_S
7      Res = execute single service discovery of query Q*
8      if Res = ø then
9        Res = SecureComposition(α,Q*)
10     endif
11     Res' = filter Res based on SecRules on the security properties
       guaranteed by each candidate service
12     for service S* in Res' do
13       WF_S* = WF[α \ S*] //substitute S* for α in the workflow
14       if there is another unassigned activity in WF_S* then
15         Put WF_S* in WFQueue
16       else
17         Add WF_S* to WFSet
18       endif
19     end
20   end
21 end
22 Return WFSet
```

Initially the algorithm identifies the applicable patterns based on the applicability conditions expressed in them. For each of the patterns that are applicable the algorithm attempts to build a workflow by instantiating the activities of the pattern. In particular, each activity in a pattern can be bounded to a single service or, if no single services are found, to a service composition generated recursively by the same process.

To instantiate an activity in a pattern, the algorithm builds a query Q* based on the specification of the pattern and the input service discovery query Q_S. Q* includes any IO dependencies that have been specified in the pattern and security conditions inferred by the security implication rules. The list of the candidate services obtained by executing the query that is constructed *on-fly* is then refined by analyzing the security conditions thanks to the additional inferences that the security implication rules can provide.

Consider, for example, the case where the confidentiality on

[1] We don't use the markup rule language for OWL (Semantic Web Rule Language, SWRL) to specify the security implication rules because of the limitation of this language wrt its decidability [23].

transmission of the input IN is required when the SSP pattern is applied. If a candidate service S for the activity A in the pattern is found, then S should already provide the requested security conditions for its input in_A as evidenced through a certificate of S.

For the output data out_A, however, there are three cases: (i) S has a certificate that guarantees the confidentiality of the transmission of out_A, (ii) S has a certificate that guarantees that out_A doesn't disclose any information about IN, or (iii) S doesn't have any of the two certificates descripted in (i) and (ii). The first two cases guarantee the confidentiality of IN but the third doesn't, and, if this is the case, S will be discarded from the list of candidate services.

Furthermore, if a confidentiality on storage, then S should provide a certificate stating either that: (i) all the stored data comply with the confidentiality on storage requirement, (ii) all the stored data doesn't disclose any information about IN or (iii) there's no data stored from the service. If none of these certificates is provided, then S would not be a valid candidate.

D. Example

As an example of applying the algorithm of Table V consider the scenario introduced in Sect. II. In this scenario suppose that the original service *getStockHeadlines* becomes unavailable and thus it becomes necessary to find a replacement for it. As, however, no single replacement service can be located, it becomes necessary to attempt to find a secure composition of different services.

An applicable pattern in this scenario is the SPFP (Fig. 5(b)), since it can filter arrays and it provides the required confidentiality for the stock *Symbol* and customer details *CustData* given as input. The query Q_A to instantiate the first activity (A in Fig. 5) of this pattern requires a service whose input is a subset of {*Symbol*, *CustData*} and output a subtype of *StockNews[]* containing at least the news related to the Symbol (this last condition is the difference from the query for S_1). Furthermore the query contains the confidentiality conditions about *Symbol* and *CustData*, which are specified by the original query.

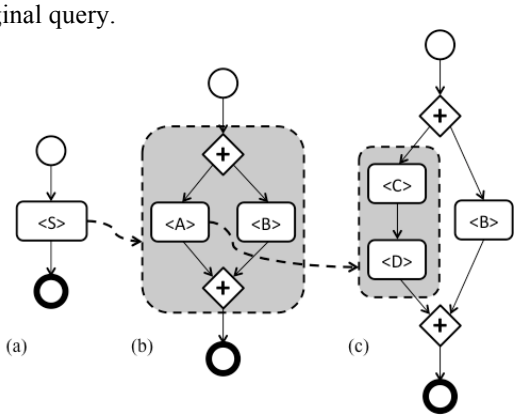

Fig. 5. Example of a complex workflow obtained through the recursive application of composition patterns.

The alternative service, S_2, doesn't match the query, so the algorithm is called recursively to find a composition satisfying the query. A pattern that guarantees the security condition is the SSP (Fig. 5(c)). The query for the last activity (D in Fig. 5) of this pattern is similar to Q_A, except that it no longer incorporates any condition about the input.

In this case the query matches the operation *fetchNewsHeadlines* from S_2 that provides an array of all the *MarketNews* from a *PaymToken*. Up to this point, no security conditions are involved.

After this instantiation, the query for the remaining activity of the SSP (C in Fig. 5) is built. This query requires a service that can derive the missing data. Thus, it requires a service that given a subset of {*Symbol, CustData*}, returns a payment token *PaymToken*, and would satisfy the security conditions about *Symbol* and *CustData* confidentiality. An operation that matches this query is the *Payment* from S_2. This operation consumes the *CustData* input and it guarantees confidentiality for this data. Since this input is confidential, then the security implication rules require that each output must be certified unrelated to the input or confidential as well. In this case *PaymToken* is confidential, so this new requirement is checked also against the already instantiated activities. In particular *fetchNewsHeadlines* must guarantee confidentiality on *PaymToken* too, otherwise the instantiation of *Payment* in C can't take place and the algorithm discards this operation from the candidate operations.

After C is instantiated, the workflow for the SSP is complete, so the recursion is done and this sub-composition instantiates the first activity A of the SPFP. At this point the output of A is a subtype of *StockNews[]* that provides the additional attribute *ISIN* and that can contain also news not related with the requested stock.

The query for the second activity of the Parallel Filter pattern (B in Fig. 5) requires a service whose input is a subset of {*Symbol, CustData*}. The filter activity also requires that the output of B is a subtype of the ID needed to filter the news for the requested stock, in this case the *ISIN*. The query also contains the security conditions about *Symbol* and *CustData* being confidential.

A service that offers the operation *getStockDetails* is found. Given a *Symbol*, this service returns the *StockDetails*, which also contain *ISIN*. Since this service uses *Symbol*, it must comply with the security condition related to it thus, it must provide confidentiality for *Symbol*. The security inference rules are then used to check if there are other security conditions that are derived from the requested condition. Since the *Symbol* of a stock should remain confidential, then all the data that can give information about it should be confidential as well. The outputs of this service, then, should have a certified property of not being related to the *Symbol* or being confidential as well. In our case the *StockDetails* are related to the *Symbol* so this output needs to be confidential as well. Furthermore, the final filtered news can give out information about the searched stock through the *ISIN* attribute, so the output of the composition must be confidential as well.

VII. Related Work

Research dealing with security in service composition has focused on the verification of the security of existing compositions through model checking [6][7][8]. Our focus,

however, is different since we are looking into applying composition patterns and security implication rules that are proven to guarantee security properties as part of a runtime service discovery and composition process.

A work that is more related to ours is [9], where planning techniques are used to compose workflows that are compliant with some lattice-based access control models (e.g. multi-level secure systems). The focus of [9] is how to find efficient algorithms for sequential workflow planning whilst our approach is more general w.r.t both the types of workflows and the security properties that it covers.

In [10] the authors describe an approach to security conscious web service composition through matching security constraints required for service provision and constraints declared by service providers. The security constraints in this approach are specified in SAML [11]. In [10], secure service compositions are generated based upon some pre-defined domain specific business workflows, whilst our approach allows the generation of arbitrary workflows.

Other works on automatic service composition (e.g. [2][12][13][14]) allow the expression of security properties in discovery queries, usually as non-functional properties. These approaches focus on specific types of security properties and check them only against single services in compositions, without addressing the overall security of a composition.

In literature [15][17][18][19], the security patterns are usually defined as design patterns that guarantee some security goal. These patterns are used to secure software during the design and develop phase, which are usually human based. Our approach, instead, is to dynamically compose services while assuring security properties; this requires automated processing of the patterns during the integration of the services.

Finally, our secure service composition patterns are similar to the workflow patterns in [3] as they specify elementary workflows that can be used to generate service compositions. However, our patterns include additional data flow and applicability specifications. In particular some of them also include some data transformation activities.

VIII. Conclusion

In this paper, we have presented an approach that supports the generation of secure compositions of services, as part of runtime service discovery. Our approach is based on composition patterns and security implication rules to determine the security properties that need to be satisfied by the individual services that participate in a composition in order for the composition to satisfy global security properties. The patterns are specified in OWL-S and describe the flows of control and data within a workflow as well as preconditions determining when a pattern can be applied. The security implication rules are specified in Situation Calculus and express the consequences that, particular actions of individual services and service compositions, have on security properties.

Service compositions are built through a stepwise and possibly recursive instantiation of patterns based on an algorithm that we have introduced in the paper. In this process, the logical connections between service and composition level security properties are determined by reasoning based on the security implication rules.

Currently, we are investigating the use of composition patterns that are more complex than primitive service workflows, and focus on an experimental evaluation of our approach.

References

[1] A. Zisman, K. Mahbub and G. Spanoudakis, "A service discovery framework based on linear composition," in *Proc. IEEE Int. Service Computing Conference (SCC 2007)*, pp.536-543, 2007

[2] F. Lécué, E. Silva and L. F. Pires, "A framework for dynamic web services composition," in *Proc. 2nd Work. on Emerging Web Services Technology (WEWST07)*, 2007.

[3] W. M. P. Van Der Aalst et al., "Workflow patterns," *Distrib. Parallel Databases* 12(1): 5-51, 2003.

[4] ASSERT4SOA Project, "D5.1– Formal models and model composition". Available: http://www.assert4soa.eu/deliverable/D5.1.pdf, 2011.

[5] J. McCarthy and P. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," *Machine Intelligence*, 4:463–502, 1969.

[6] M. Deubler, et al., "Sound development of secure service-based systems," in *Proc. of the 2nd Int. Conf. on Service oriented computing (ICSOC '04)*, pp. 115-124, 2004.

[7] Jing Dong, Tu Peng and Yajing Zhao, "Automated verification of security pattern compositions," *Inf. Softw. Techn.* 52(3):274- 295, 2010.

[8] M. Bartoletti, P. Degano and G. L. Ferrari, "Enforcing secure service composition," *18th Work. on Computer Security Foundations,* 2005.

[9] M. Lelarge, Z. Liu and A. Riabov, "Automatic composition of secure workflows," in *Proc. of ATC'2006*, 2006.

[10] B. Carminati, et al., "Security conscious web service composition," in *Proc. of the Int. Conf. on Web Services (ICWS)*, 2006.

[11] OASIS. SAML 1.0 Specification Set [Online]. Available: http://saml.xml.org/saml-specifications, 2002.

[12] Keita Fujii and Tatsuya Suda, "Semantics-based dynamic web service composition," *IEEE J. Sel. Areas Commun*, 23: 2361- 2372, Dec. 2005.

[13] B. Medjahed, A. Bouguettaya and A. K. Elmagarmid, "Composing web services on the semantic web," *The VLDB Journal*, 12(4):333-351, 2003.

[14] M. C. Jaeger, G. Rojec-Goldmann and G. Muhl, "QoS aggregation for web service composition using workflow patterns," in *Proc. of the 8th Int. Conf. on Enterprise Distributed Object Computing,* 2004.

[15] Aniketos, "D3.1 – Design-time support techniques for secure composition and adaptation". Available: http://www.aniketos.eu/, 2011.

[16] A. Zisman, G. Spanoudakis, J. Dooley. "A framework for dynamic service discovery", In *Proc. of 23rd Int. ACM/IEEE Conf. on Automated Software Engineering, 2008*

[17] J. W. Yoder and J. Barcalow, "Architectural patterns for enabling application security," in *Proc. of Pattern Languages of Programs,* 1997.

[18] E. B. Fernandez and R. Y. Pan, "A pattern language for security models," in *Proc. of Pattern Language of Programs (PLoP'01)*, 2001.

[19] N. Yoshioka, H. Washizaki, K. Maruyama, "A survey on security patterns," *Progress in Informatics*, No. 5 pp. 35-47, 2008.

[20] OWL-S Coalition. OWL-S 1.2 Technical Overview [Online]. Available: http://www.ai.sri.com/daml/services/owl-s/1.2/overview/, 2008.

[21] W3C Web Ontology Working Group. OWL Web Ontology Language Reference [Online]. Available: http://www.w3.org/TR/owl-ref/, 2004.

[22] W3C XSL Working Group. XSL Transformations (XSLT) Version 2.0 [Online]. Available: http://www.w3.org/TR/xslt20/, 2007.

[23] B. Motik, U. Sattler, and R. Studer. "Query Answering for OWL-DL with rules," *J. Web Semant. 3*, 1 41-60. 2005.

[24] G. Spanoudakis, and A. Zisman, "Designing and Adapting Service-based Systems: A Service Discovery Framework," *In Service Engineering: European Research Results*, S. Dustdar, F. Li (eds), Springer, 2010

[25] B. Raman, et al. "The SAHARA model for service composition across multiple providers." In *Proc. of the 1st Int. Conf. on Pervasive Computing*, LNCS 2414, 2002.

[26] Shankar R. et al. "SWORD: A developer toolkit for web service composition." In *Proc. of the 11th Int. WWW Conference*, 2002.

[27] S. Majithia, D. Walker and W. A. Gray. A Framework for Automated Service Composition in Service-Oriented Architectures. In *Proc of 1st European Semantic Web Symposium*, LNCS 3053, pp. 269-283, 2004.