Pattern–Based Design and Verification of Secure Service Compositions

Luca Pino, George Spanoudakis, Maria Krotsiani and Khaled Mahbub

Abstract— Ensuring the preservation of security is a key requirement and challenge for Service-Based Systems (SBS) due to the use of third party software services not operating under different security perimeters. In this paper, we present an approach for verifying the security properties of SBS workflows and adapting them if such properties are not preserved. Our approach uses secure service composition patterns. These patterns encode proven dependencies between *service level* and *workflow level* security properties. These dependencies are used in reasoning processes supporting the verification of SBS workflows with respect to workflow security properties and their adaptation in ways that guarantee the properties if necessary. Our approach has been implemented by extending the Eclipse BPEL Designer and validated experimentally. The experimental evaluation has produced positive results, indicating that even for complex workflows and large sets of secure service composition patterns verification can be performed efficiently.

Index Terms— Design Tools and Techniques, Security and Protection, Services Composition, Systems analysis and design

1 Introduction

Security assurance is important for any software application but acutely so in the case of *service-based systems* (SBSs), i.e., systems composed of distributed software services, which can be deployed on different and heterogeneous infrastructures and operate without common ownership and centralised control.

Assessing and providing assurance about the security of SBSs is a complex problem that has no comprehensive solution to the best of our knowledge. Existing solutions (e.g., [2][3][6][8]) rely on different forms of model checking and theorem proving to verify security properties of service compositions. These approaches typically require the specification of: (a) behavioural models of the software services used by the SBS, (b) the component that orchestrates them to provide the SBS functionality (i.e., the service orchestrator), and (c) the security properties that need to be guaranteed in some temporal logic language. There are two main difficulties with such approaches. The first is that creating accurate specifications of (a)-(c) for realistic SBSs is a non-trivial and time-consuming task. The second is that, even if SBS specifications are available, performing automated static analysis might be computationally intractable.

In this paper, we present an alternative approach for designing, adapting and verifying the security properties of SBSs, which is based on *pattern driven verification*. Our approach assumes that an SBS is designed and implemented by a service orchestration process (aka *service workflow*), which invokes (and receives responses from) the individual services that constitute the SBS (aka *partner services*) and may perform various computations upon the data exchanged with these services.

To support the verification of security properties, our approach uses *secure service composition (SCO) patterns*. These patterns encode proven dependencies between *service level security properties* (i.e., security properties of the individual services of an SBS) and *workflow level security properties* (i.e., security properties of the entire orchestration/workflow of the SBS). The encoding of such dependencies in SCO patterns enables the inference of service level security properties, which – if satisfied by the individual services of the SBS – would guarantee the satisfaction of workflow level security properties for it.

The inference of service level security properties required for verifying workflow level properties of an SBS is the basis of our approach since – once these properties are identified – verification can be based on checking whether the specific services that constitute the SBS satisfy the security properties required of them. Checking this is based on *digital security certificates*, which are assigned to services following a certification process. Such certificates encode: (i) the service that is certified, (ii) the endpoint at which this service can be accessed, (iii) the service level security property that is certified for the service, and (iv) the evidence demonstrating that the security property is satisfied by the service [20].

The inference of service level security properties using the SCO patterns enables also the generation/adaptation of an SBS service workflow in a manner guaranteeing that it will satisfy required workflow level security properties. More specifically, the inference process establishes all the

L. Pino is with Department of Computer Science, City, University of London, Northampton Square, London, EC1V 0HB, UK, E-mail: Luca.Pino.2@city.ac.uk.
M. Krotsiani is with Department of Computer Science, City, University of London, E-mail: Maria.Krotsiani@city.ac.uk.
G. Spanoudakis is with Department of Computer Science, City, University of London, E-mail: G.E.Spanoudakis@city.ac.uk.
K. Mahbub is with School of Computing and Digital Technology, Birmingham City University, Millennium Point, Birmingham B4 7XG, UK, E-mail: khaled.mahbub@bcu.ac.uk. Note: This work was carried whilst the author was with City, University of London.

alternative combinations of security properties of the individual partner services of the SBS (referred to as *security plans* in the rest of the paper) that could ensure the satisfaction of workflow level security properties by it. Driven by such security plans, SBS adaptation is realised as a search process that locates potential partner services for the SBS workflow that satisfy the required security properties. The main benefits of our approach are that:

- It can be used both for the verification of workflow level security properties of existing service workflows (SBSs), and the adaptation of such workflows in ways that are guaranteed to preserve security properties.
- It is computationally feasible since the verification of security properties is based on finding combinations of the pre-specified SCO patterns that would entail them. Although this process may, in principle, have a combinatorial complexity, experimental results have shown promising average performance even for large SCO pattern sets.
- It can be extended through the incorporation of new SCO patterns.

It should be noted that, although our approach provides sound and feasible verification analysis, it is not complete in performing security verification. This is because, there is no guarantee that SCO patterns will encode ALL the combinations of service level security properties that can guarantee a workflow level security property. Hence, a failure of the pattern driven verification process does not mean that the security property in question does not hold for the workflow.

The work we present in this paper extends our previous work described in [24][25][26][28]. The work in [24][25] presented an initial proof-of-concept realisation of our approach for pattern-driven generation of secure service compositions without, however, addressing verification. In our original work, patterns were modelled using OWL-S [19] and Situation Calculus [16]. This, however, turned out to be inefficient for both specifying and applying patterns for secure service composition. In [26], we presented the initial version of the algorithm for inferring service level from workflow level security properties, and provided an overview of how it could support workflow verification/adaptation without any algorithmic details for these processes. In [28], we presented formal proofs of some SCO patterns (e.g., integrity) and an initial prototype of our approach focusing on security-driven service discovery.

The pattern proofs approach introduced in [28] was based on modelling patterns using the Security Modeling Framework SeMF [13]. SeMF provides an adequate formal framework for modelling basic service workflows (e.g., sequential, split-join, OR-orchestrations) in SBS systems and proving pattern properties [27]. However, it cannot support the application of patterns once they are proven.

The main contributions of this paper, with respect to our earlier work, are:

(a) It extends the SCO patterns representation scheme with additional conditions about pattern activity inputs and outputs, which are required for matching patterns with SBS workflows, and provides a scheme for expressing patterns in Drools [9], to enable pattern application (matching).

(b) It introduces the verification algorithm that is based on the extended form of SCO patterns and can verify if a service workflow satisfies specific security properties required of it.

(c) It presents and discusses the outcomes of an experimental evaluation of our approach.

The remainder of this paper is structured as follows. Section 2 presents a scenario showing the need for and the ways of using the pattern driven verification approach. Section 3 introduces SCO patterns and describes their specification in Drools. Sections 4 and 5 present the security verification and the secure workflow generation processes, respectively. Section 6 describes the tool we implemented to realise our approach. Section 7 presents the outcomes of the experimental evaluation of our approach. Section 8 discusses related work. Finally, Section 9 provides conclusions and outlines directions for future work.

2 Scenario

To exemplify our approach, consider the SBS service workflow fragment that is shown in Fig 1. This fragment (called *Checkout*) corresponds to the last part a purchasing process realised by an SBS. More specifically, upon receiving a purchasing request consisting of a list of items to be purchased, the credit card details and address of the purchaser, *Checkout* takes payment (see activity *Payment* in Fig. 1) for the purchased items, places the order in a purchase repository (see activity *PlaceOrder* in Fig. 1), and creates an order report (see activity *WriteReport* in Fig. 1). The activities *Payment*, *PlaceOrder* and *WriteReport* of *Checkout* are realised through the invocation of operations of partner services, which are assumed to have identical names with the relevant workflow activities.

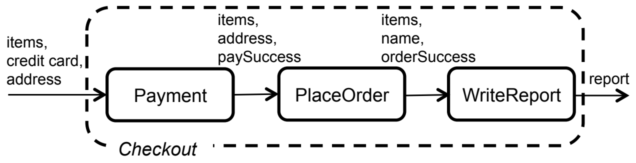


Fig. 1. Example of SBS service workflow – Checkout.

In this scenario, a designer might wish to verify whether the *Checkout* workflow preserves the confidentiality of the *credit card* and *address* information of the user. To ensure this, it is necessary to verify that all the services, which are orchestrated by *Checkout* (i.e., *Payment, PlaceOrder* and *WriteReport*), preserve the confidentiality of credit card and address information, as well as the confidentiality of additional information that is exchanged internally within the workflow (i.e., *paySuccess* and *orderSuccess*) if the latter also includes any information about the *credit card* and *address* information of the user. Furthermore, the preservation of confidentiality will need to be checked against the transmission, processing and storage of any of the information items that need to remain confidential.

3 Secure Composition Patterns

3.1 Overview of pattern structure and semantics

SCO patterns encode proven dependencies between security properties of individual services (i.e., *service level security properties*) and security properties of the entire SBS service workflows (i.e., *workflow level properties*). As discussed in Sect. 1, the encoding of such dependencies enables: (i) the verification that the service workflow of an SBS satisfies certain security properties, and (ii) the generation (and adaptation) of an SBS workflow in a way that is guaranteed to satisfy required workflow level security properties. The specification of an SCO pattern consists of four parts:

(i) The *workflow* (*WF*) part – This part of the pattern defines the form of the workflow (i.e., service orchestration) that the pattern applies to. WF is specified as an orchestration of abstract *activity placeholders*. When a pattern is matched against the workflow of an SBS, the placeholders in its WF may be bound to invocations of operations of specific *partner services* of the SBS or sub-workflows of it.

(ii) The *RSP* properties part – This part of the pattern defines the workflow level security properties (referred to as "RSP properties" in the following) that the pattern can guarantee for the workflow specified in its WF part.

(iii) The *ASP* properties part– This part of the pattern defines the service level security properties (referred to as "ASP properties" in the following), which are required of the activity placeholders in the workflow of the pattern, in order to guarantee the RSP properties specified in the pattern.

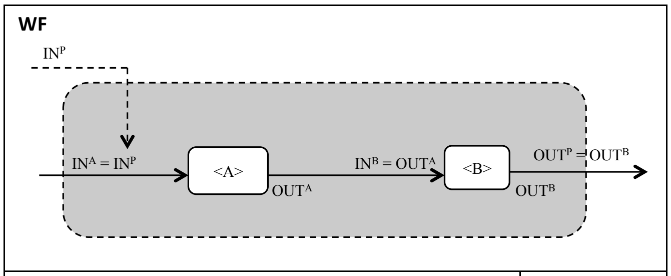
(iv) The *CONDITIONS* part – This part includes conditions, regarding the inputs and outputs of the activity placeholders of the pattern.

The semantic interpretation of an SCO pattern having the above structure is that if the ASP properties, which have been specified for the activity placeholders in the workflow of the pattern, and the conditions of the pattern are true, then the RSP property specified in the pattern is also true for the entire WF of it. Formally, this can be expressed as: $ASP \wedge WF \wedge CONDITIONS \vDash RSP$ where $\vDash$ denotes the entailment relation that has been established by the proof of the pattern.

SCO patterns cover basic control flow patterns suggested by the Workflow Management Coalition [36]), namely the *sequential*, *parallel split* and *exclusive choice* orchestrations, and the security properties of confidentiality and integrity [27][23]. In the following, we give an example of an SCO pattern to demonstrate the use of the above structure.

3.2 Example of an SCO pattern

Our first example is an SCO pattern regarding the security property of confidentiality, i.e., a property requiring that no non-authorised disclosure of information should be possible in a system. Confidentiality has been commonly defined based on the concept of information flow (IF) [34] IF-based definitions of confidentiality stratify the users of a system in classes with different access rights to information, and distinguish the information flows within it according to the class of users that they should be accessible to. Typically, the user classes used in IF-approaches are low-level users with restricted access to information, and high-level users having full access. Based on the above principles, there have been several definitions of properties, whose intent is to express the concept of confidentiality. The definition that we focus on is that of *Perfect Security Property (PSP)* [34]. PSP requires that a low-level user, allowed to access only public information, should not be able to determine anything about high-level (i.e., confidential) information.



ASP:	Conditions:
$PSP(A, V^A, C^A)$ and $V^A \subseteq V^P$ and $C^A \cap V^P = \varnothing$	$IN^P = IN^A$
$PSP(B, V^B, C^B)$ and $V^B \subseteq V^P$ and $C^B \cap V^P = \varnothing$	$OUT^A = IN^B$
RSP: $SecReq^P = PSP(P, V^P, C^P)$	$OUT^B = OUT^P$

Fig. 2. PSP SCO Pattern.

Fig. 2 shows the SCO pattern for preserving PSP on a sequential service workflow P, i.e., a service workflow with two activity placeholders (A and B), in which A is executed before B. The structure of P is shown in the *WF* part of the figure. Further conditions that define P are specified in the *Conditions* part. These are: (a) the inputs of A are the inputs of the workflow ($IN^P = IN^A$), (b) the inputs of B are the outputs of A ($IN^B = OUT^A$), and (c) the outputs of P are the outputs of B ($OUT^P = OUT^B$).

Let us assume that for each x in {P, A, B}

- IN^x and OUT^x are the sets of inputs and outputs of x, and $E^x = IN^x \cup OUT^x$;
- V^x and C^x are two disjoint subsets of E^x, which partition it into public parts V^x (i.e., parts visible to low-level users) and confidential parts C^x (i.e., parts visible only to high-level users)

Then, as proven in [18], PSP holds on the workflow P if, for all activity placeholders $x \in \{A, B\}$: (a) the actions of x that reveal public information are part of the actions of P that reveal public information (i.e., $V^x \subseteq V^P$), and (b) the confidential actions of x that reveal confidential information do not include any action of P that reveals public information (i.e., $C^x \cap V^P = \varnothing$). The conditions (a) and (b) are expressed as ASP properties of the pattern, and entail the PSP property on P. The latter is expressed by $PSP(P,V^P,C^P)$ in the RSP part of the pattern.

3.3 Encoding of SCO patterns in Drools

3.1.1 Overview of rule language

SCO patterns are expressed as Drools production rules [9]. The reasons for choosing this approach to specify patterns is that Drools is supported by rule engine, which realises a rule based reasoning process based on the Rete algorithm [11], i.e., an efficient pattern-matching algorithm known to scale well for large numbers of rules and data sets. Hence, the choice of Drools enabled us to have an efficient implementation of the pattern based reasoning process.

In the following, we introduce the scheme for encoding SCO patterns in Drools and then show how the two patterns introduced in Sect. 3 are encoded in this scheme.

A production rule in Drools has following structure:

```
rule name <attributes>*
when <conditional element>* then <action>* end
```

The antecedent (i.e., `when`) part of the rule specifies a set of conditions and the consequence (i.e., `then`) part of the rule specifies a list of actions. When a rule is applied, the Drools rule engine checks whether its conditions are satisfied by (i.e., match with) the facts in the KB and, if they are, the actions of the rule are executed. Rule actions are typically used to modify the KB by inserting, retracting or updating the objects (facts) in it. Such modifications are encoded through the standard Drools actions "insert", "retract" and "update", respectively. The conditions of a rule are expressed as *patterns* of objects that encode the facts in the Drools KB. These patterns define object types and constraints for the data encoded in objects. These constraints may be atomic or complex. Complex Drool object constraints are defined through logical operators (e.g., `and, or, not, exists, forall`). A presentation of the full grammar of the Drools rule language is beyond the scope of this paper and may be found in-http://docs.jboss.org/drools/release/6.1.0.Final/drools-docs/html_single/. Table 1 provides an overview of the main specification constructs of this language, to enable the reader understand the SCO patterns specifications given in the paper.

TABLE 1: HIGH LEVEL DROOLS RULE SPECIFICATION CONSTRUCTS

Construct	Meaning/Usage
Conditional element: `conditionalElement: and-CE | or-CE | not-CE | exists-CE | forall-CE | from-CE | collect-CE | accumulate-CE | eval-CE`	Conditional elements are used to specify conditions in the *when* part of a rule and in constraint expressions (see Pattern construct). Conditional elements realise basic logical operators (e.g., and, or, not); quantified logic operators (forall and exists); and object collection operators (e.g., collect, accumulate).
Pattern `Top level syntax: Pattern: <patternBinding ":" > PatternType "(" Constraints ")"`	Patterns are matched with elements in the working memory. The pattern binding is typically a variable and the pattern type refers to declared object types that could be matched with the pattern. Constraints are specified by logical expressions. Such expressions can be constructed by logic conditional elements (see above); object collection elements, unification operators (:=); relational (<, >, =<, >=, !=, ==); arithmetic (e.g., +,-,*,/,%); property/list access operators; data accumulation functions (e.g., min, max, average, count, sum); regular expression matching operators (e.g., matches, contains, str); and temporal (Allens) operators (e.g., before, after, coincides).
Actions: Top Level Syntax:	Actions used in our approach are: *Modify* – This action modifies the contents of a fact/object. *Insert* – This action insert a new fact/object into the knowledge base. *Retract* – This action deletes a fact/object.

3.1.2 Extensions of the rule language for specifying SCO patterns

Drools rules are used to encode relations between the ASP and RSP security properties in SCO patterns. The encoding scheme is set to enable the inference of the ASP properties, which are required of the activity placeholders of the workflow of the SCO pattern, for this workflow to have the RSP property guaranteed by the pattern. More specifically: (i) the `when` part of the rule encodes the WF part of the pattern, the conditions regarding the inputs and outputs of the activities of WF, and the RSP property guaranteed by the pattern for WF; and (ii) the `then` part of the rule encodes the service level security properties ASP, which if satisfied by the WF's activity placeholders would guarantee the workflow level property RSP. Hence, a Drools rule expressing an SCO pattern encodes the implications: WF $\wedge$ Conditions $\wedge$ RSP $\Rightarrow$ ASP_i (i=1,...,n) where ASP_i are the ASP properties required of the individual services bound to the activity placeholders of the SCO pattern. Note that this implication expresses the opposite of the dependency relation proven in the pattern, i.e., $ASP \wedge WF \wedge CONDITIONS \vDash RSP$. This encoding enables the inference of the ASP_i properties, which if satisfied by the individual services of a workflow would guarantee RSP for it, during the WF verification and adaptation processes (see Sect. 4 and 5 below).

The specification of SCO patterns in Drools makes also use of *placeholder* and (security) *requirement* objects. The types of these objects are specified as shown in the UML diagram of Fig. 3. Activity placeholder objects (or simply "placeholders") represent the partner services (or compositions of partner services) that are already bound or should be bound to the workflow of an SCO pattern. Placeholders can be of three different types:

- *PartnerLinkActivity (PLA) placeholders* – These are used to represent partner services that are already bound to and can be invoked by the WF of an SCO pattern. PLA placeholders contain information about the services bound to WF and the security properties that have been certified for them.
- *OrchestrationPattern (OP)* placeholders – These are used to represent sub-workflows of the overall WF of the pattern. OP placeholders can be *Sequential, Parallel* (i.e., *AND-OPs*) and *Choice (OR-OPs)* representing activities, which are executed sequentially, in parallel or alternatively, respectively.
- *Unassigned Activity (UA)* placeholders, representing activities, which are yet to be bound to an individual service or a service orchestration in order to have an executable WF. UA placeholders contain the structural

(i.e., WSDL) specification of the service or orchestration that is eligible for instantiating it.

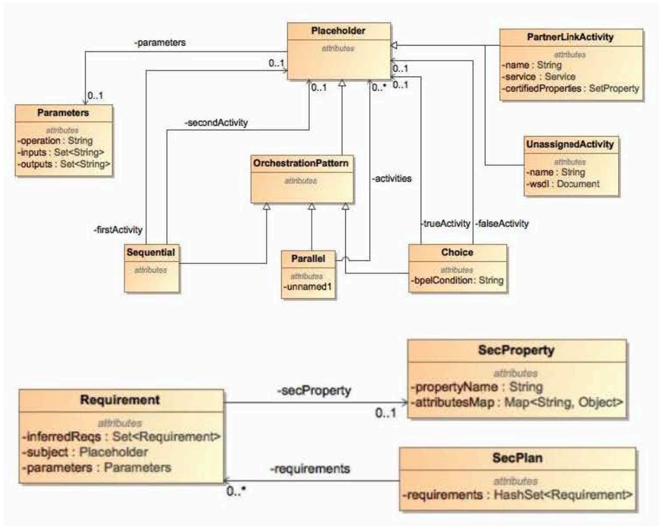


Fig. 3. SCO Pattern specification object types

A security requirement object expresses a security property ASP that is required of a placeholder of the SCO pattern (i.e., an ASP property). The required security property is expressed by the *secProp* opposite association end (field) of security requirement objects, and the placeholder that the property refers to is expressed by the *subject* field of the same object. A security requirement object may also contain: (a) a set of *Parameters* indicating the inputs or outputs of the placeholder that the security property refers to, and (b) further requirements (*inferredReqs* field) that may have been deduced as sufficient for the security property expressed by the requirement to hold. Security property objects contain the security property name (*propertyName* field) and an optional set of attribute-value fields (*attributesMap* field) allowing the expression of extra conditions over the property. The set of all the ASP properties that are inferred for the different services of a workflow by an SCO pattern are aggregated into *SecPlan* objects.

3.1.3 Example of Drools specification of SCO pattern

Based on the above scheme, the SCO pattern expressing the PSP notion of confidentiality that we discussed in Section 3.2, can be represented as shown in Table 2. The when part of this rule specifies: (i) the two activity placeholders A and B of the PSP pattern (see object variables $A and $B in lines 2-3 and 4-5); (ii) the order in which $A and $B should be executed (see variable $WF), (iii) the conditions between the outputs of $A, and the inputs of $B as required by the PSP pattern (lines 6-8); and (iv) the RSP property that can be guaranteed through the application of the pattern, i.e., the PSP property in this case (see variable $RSP in lines 9-10). (i) and (ii) constitute the specification of the WF part of the pattern.

The then part, the rule specifies actions, which generate a security plan indicating the security properties ASP that, if satisfied by the activity placeholders of the pattern's workflow WF, would make WF to satisfy RSP. According to the proof of the pattern, each of the placeholders should

TABLE 2: DROOLS RULE FOR PSP PATTERN

```
1.   rule "PSP on Cascade"
     when
2.     $A:    Placeholder( $input : parameters.inputs,
3.                  $intData : parameters.outputs)
4.     $B:    Placeholder( parameters.inputs==$intData,
5.                  $output : parameters.outputs)
6.     $WF:  Sequential( parameters.inputs == $inputs,
7.                  parameters.outputs == $outputs,
8.                  firstActivity==$A, secondActivity==$B)
9.     $RSP: Req( propertyName == "PSP",
10.                 subject == $WF, satisfied == false)
11.    $SP:   SecPlan( requirements contains $RSP)
     then
12.    SecPlan newSecPlan = new SecPlan($SP);
13.    newSecPlan.removeRequirement($RSP);
14.    Set V_P = $RSP.getAttributesMap().get("V");
15.    Req ASP_A = new Req($RSP, "PSP", $A);
16.    ASP_A.getAttributesMap().put("V", new
       Operation("subset", V_P));
17.    ASP_A.getAttributesMap().put("C", new
       Operation("subset", new Operation("complement",V_P)));
18.    newSecPlan.getRequirements().add(ASP_A);
19.    insert(ASP_A);
20.    Req ASP_B = new Req($RSP, "PSP", $B);
21.    ASP_B.getAttributesMap().put("V", new
       Operation("subset", V_P));
22.    ASP_B.getAttributesMap().put("C", new
       Operation("subset",
         new Operation("complement",V_P)));
23.    newSecPlan.getRequirements().add(ASP_B);
24.    insert(ASP_B);
25.    insert(newSecPlan);
     end
```

satisfy the "PSP" property. Hence, "PSP" is set as the ASP property that should be satisfied ASP_A and ASP_B (see lines 15 and 20, respectively).

The additional conditions needed by the pattern (i.e., the conditions $V^A \subseteq V^P$ and $C^A \cap V^P = \emptyset$ for A, and the conditions $V^B \subseteq V^P$ and $C^B \cap V^P = \emptyset$ for B) are added to ASP_A (see lines 16 and 17) and ASP_B (see lines 21 and 22). The specification of these additional conditions refers to the class Operation (see lines 16-17 and 21-22). This operation is provided by the query language (and the discovery engine implementing it), which is used for finding suitable matches (see Sect. 5 and Sect. 6).

4 VERIFICATION PROCESS

The process for verifying if an SBS workflow SBS-WF satisfies required security properties has two main phases.

In the first of these phases, all the SCO patterns that can guarantee the RSP property required of SBS-WF are identified and, if their workflow structure matches the structure of SBS-WF, they are used to infer the ASP properties, which if satisfied by the individual services of SBS-WF, would guarantee RSP for it. This phase may generate alternative combinations of ASP properties for the services of SBS-WF, which would guarantee RSP. Each of these combinations is what we call a *security plan* in our approach. In the second phase of the verification process, each of the security plans generated in the first phase are used to drive a search process. This process checks if the individual services of SBS-WF referred to in the security plan satisfy the ASP property required of them by the plan. In the following, we present the algorithms that realise the two phases of the verification process.

4.1 Phase 1: Inference of Security Requirements

The algorithm for generating different security plans, i.e., the possible alternative combinations of security properties of activity placeholders of a workflow WF that would make it satisfy a workflow level security property RSP is listed in Table 3.

TABLE 3: INFER SECURITY PLANS ALGORITHM

```
Algorithm: INFERSECURITYPLANS(WF, Req, SecPlans)
Input:  WF /* workflow */
        Req /* security requirement for WF */
Output: SecPlans /* Security plans with ASPs */
1. INFERRECURSION(WF, Req, [], SecPlans)
```

```
Algorithm: INFERRECURSION(PH, Req, InPlans, OutPlans)
Input: PH /*activity placeholder*/
       Req /*security requirement for PH*/
       InPlans /* list of current plans*/
Output: OutPlans /*list of inferred security plans*/
1.  If there is no pattern P such that
       P.RSP matches Req and P.WF matches WF then
2.     OutPlans:= InPlans
3.  Else
4.    For each pattern P such that P.RSP matches Req and
      P.WF matches WF do
5.      SecPlans_P := security requirements inferred
                      by the inference rules of P
6.      Remove Req from InPlans
7.      Add SecPlans_P to InPlans
8.    EndFor
9.    For each R in InPlans where R.subject is a work-
      flow do
10.      INFERRECURSION(R.subject, R, InPlans, OutPlans)
11.   EndFor
12. EndIf
13. EndFor
```

The algorithm is invoked having as input a workflow (*WF*) and a security property (*RSP*) required of it, which is encoded with the security requirement *Req*. Based on these two inputs, the algorithm derives the security requirements (i.e., ASP properties) that should be satisfied by partner services that may be bound to the different activity placeholders in *WF* in order to guarantee RSP. Given *WF* and a security requirement *Req* requiring the property *RSP*, the algorithm tries to apply all the SCO patterns that would be able to guarantee *RSP*. A pattern *P* is applied if its workflow (*P.WF*) matches with the input workflow *WF*. In this case, the security plans that can be derived from the pattern (through the application of its rules) are computed (line 5). These plans replace the initial requirement *Req* (see lines 6-7 in INFERRECURSION). If the updated list of security plans contains only ASP security properties that are required of individual activities (i.e., not of *OrchestrationPattern* placeholders), the algorithm terminates. Otherwise, if the security plans include security properties required of activity placeholders that are themselves (sub) workflows, the algorithm attempts to find SCO patterns that match the workflow structure of the sub workflows and could guarantee the security property required of these sub workflows, recursively (see lines 9-11 in INFERRECURSION). This process terminates when a list of security plans includes workflow placeholders matching with no available SCO patterns (see lines 1-3 in INFERRECURSION).

INFERSECURITYPLANS may generate alternative security plans, i.e., *SecPlans* may be a list of alternative security plans $[(R_{11},\ldots,R_{1n}),(R_{21},\ldots,R_{2m}),\ldots,(R_{k1},\ldots,R_{kl})]$. Each of these plans includes a set of ASP properties for the individual services of the input workflow WF of the algorithm. If all the ASPs of a plan are satisfied by the individual services that they refer to, the security property RSP that is required of the original workflow will be also satisfied. This is due to the definition of SCO patterns, according to which for each security plan *SecPlan* it holds that *SecPlan* $\Rightarrow RSP$, and thus: $(R_{11} \wedge \ldots \wedge R_{1n}) \vee (R_{21} \wedge \ldots \wedge R_{2m}) \vee \ldots \vee (R_{k1} \wedge \ldots \wedge R_{kl}) \Rightarrow RSP$. INFERSECURITYPLANS realises a breadth-first inference of all the possible security plans that could guarantee RSP for WF and is used to support both the processes of workflow verification and the process of workflow generation/adaptation. The way in which it is used for each of these purposes is described next.

4.2 Phase 2: Verification of service properties

The process of checking if a given workflow satisfies a required security property is realised by the algorithm VERIFYWORKFLOW that is listed in Table 4.

TABLE 4: WORKFLOW VERIFICATION ALGORITHM

```
Algorithm: VERIFYWORKFLOW(WF, Req, VPlan)
Input:  WF /* workflow */
        Req /*security requirement to verify */
Output: VPlan /*verified security plan for WF */
1. INFERSECURITYPLANS(WF, Req, SecPlans)
2. VPlan := nil
3. For each Plan in SecPlans do
4.     If VERIFYREQUIREMENT(WF, Plan) then
5.           VPlan:= Plan
6.           Exit
7.     Endif
8. EndFor
```

Given a request to check if a workflow *WF* satisfies a given security property expressed by the security requirement *Req*, VERIFYWORKFLOW firstly invokes the algorithm INFERSECURITYPLANS to identify the list of the alternative possible security plans that would guarantee the property expressed by *Req*. These plans are stored in the variable *SecPlans*. Subsequently, it calls the algorithm VERIFYREQUIREMENT to check if the service level security requirements (i.e., $R_{i1},\ldots,R_{im}$) of each specific plan in *SecPlans* is satisfied by the services that they refer to. The algorithm terminates as soon as it finds the first satisfied security plan (see line 6).

The algorithm VERIFYREQUIREMENT is listed in Table 5. As shown in the table, VERIFYREQUIREMENT is invoked with a workflow (*WF*) and an individual security plan that needs to be verified for it (i.e., *($R_{i1} \wedge \ldots \wedge R_{ik}$)*) as inputs, and tries to find if each security requirement R_{ij} in the plan is satisfied. To check this, it tries to find a certificate for the service that is bound to the activity placeholder, which *R_{ij}.subject* refers to, certifying the security property required by R_{ij} (i.e., *R_{ij}.secProperty*). In cases where *R_{ij}.subject* is a sub workflow, the algorithm reports that the security property required of it cannot be verified. This is because INFERSECURITYPLANS has already tried to decompose workflows to individual services and security properties for them that would make a sub workflow satisfy the security property required of it. Hence, having a sub workflow in a security plan means that there was no SCO pattern could be used to decompose it to individual services and properties that would make it satisfy the security property required of it. Similarly, if the subject of a requirement is an unassigned activity placeholder (UA), i.e., a placeholder with no concrete individual service bound to it, the security property required of the placeholder according to the plan cannot be verified and the algorithm reports that the entire workflow cannot be verified.

TABLE 5: REQUIREMENTS VERIFICATION ALGORITHM

```
Algorithm: VERIFYREQUIREMENT(WF, SecPlan): Boolean
Input:    WF /* workflow */
          SecPlan /* plan of security requirements */
Output: true or false /* verification outcome */
1. Plan := SecPlan
2. HoldsSP := true
3. While there are more requirements R in Plan
          and HoldsSP do
4. R:= next requirement in Plan
5. If R.subject is a PartnerLinkActivity placeholder
6.        then
7.        If exists certificate CRT in
             R.subject.certificates such that
             CRT.secProperty = R.secProperty
8.           then HoldsSP:= true
9.           else HoldsSP:= false
10.       EndIf
11. else /* OrchestrationPattern or UnassignedActivity * /
12.       HoldsSP:= false
13. EndIf
14. Remove R from Plan
15. EndWhile
16. Return (HoldsSP)
```

The algorithm VERIFYREQUIREMENT can also be used to verify a security plan against workflow fragments, i.e., parts of a workflow delimited by a control flow activity. In case of BPEL, for example, workflows fragments correspond to *scope* or *control flow* activities (i.e., *sequence*, *flow*, *while*, *forEach*, *repeatUntil*, *if-then-else* or *pick* activity) that may contain multiple service invocations.

4.3 Example

As an example of verifying security requirements consider the case where an SBS designer wishes to check that the *Checkout* process of Fig. 1 preserves the confidentiality of information regarding the credit card and address of the process user. In this case, confidentiality can be expressed through requiring that a low-level user (i.e., a user who should be able to access only public information) should not be able to determine anything about the high level (confidential) information of credit card and customer address (i.e., the PSP property discussed in Sect. 3). *Checkout* can be seen as a sequential workflow consisting of the atomic activity *Payment* and a sub-workflow that follows it (*SubWF*), which itself is a sequential workflow involving two atomic activities: *PlaceOrder* and *WriteReport* (see Fig. 3).

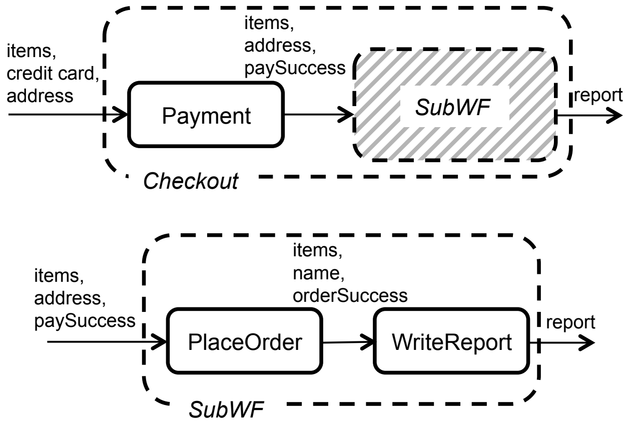

items, credit card, address
items, address, paySuccess
Payment
SubWF
report
Checkout
items, address, paySuccess
items, name, orderSuccess
PlaceOrder
WriteReport
report
SubWF

Fig. 4. The inference process on the sub-workflows of *Checkout* SBS process workflow.

To verify *Checkout*, the algorithm INFERSECPROPERTIES will be called initially. In the first iteration of it, the *PSP on Cascade* SCO pattern will be applied on *Checkout*, returning two security requirements: (1) a requirement for the service *Payment* requiring confidentiality for *credit card*, *address* and *paySuccess*, and (2) a requirement for the sub workflow *SubWF*, requiring confidentiality for *address* and *paySuccess*. In its second iteration, the algorithm applies again the same pattern, but this time on *SubWF*. In the second application of the pattern, INFERSECPROPERTIES creates and adds two security requirements to the ongoing security plan: (3) a requirement for *PlaceOrder* requiring confidentiality for *address*, *paySuccess*, *name* and *orderSuccess* and (4) a requirement for *WriteReport*, requiring confidentiality for *name* and *orderSuccess*. After these steps, the security plan will consist of only individual services and security properties required of them. Subsequently, VERIFYWORKFLOW invokes VERIFYREQUIREMENT to check whether the services bound to the workflow have the required properties. This check is carried out by searching for security certificates that can confirm the required ASP properties for the particular services, i.e., PSP for *address*, *paySuccess* and *orderSuccess* of *PlaceOrder*, and for *name* and *orderSuccess* of *WriteReport.* The process of searching for such certificates is discussed in Sect. 6.

5 GENERATION OF SECURE WORKFLOWS

5.1 Algorithm

When an existing workflow does not satisfy a required RSP security property due to a particular partner service (or a fragment of it), it might be possible to replace the responsible service (or workflow fragment) in order to restore the required RSP. This modification is handled by the algorithm listed in Table 6. This algorithm starts by trying to find appropriate workflows based on a query (Q) expressing structural, behavioural and security requirements for the service or process fragment that should be modified. To do this, initially it tries to identify abstract (i.e., not instantiated) workflows that can provide the requested functionality by searching for appropriate functional workflows in a *repository of reference workflows* (see lines 1-2). This repository contains abstract workflows encoding *reference process models* providing standardised functionalities in different domains. Examples of such reference process models exist for several domains including, for example, financial services (e.g., SWIFT) [32] and electronic data interchange in fields such as manufacturing, logistics and telecommunications (e.g., RosettaNet Partner Interface Processes (PIPs) [29] and IBM Industry Packs [14]). The abstract workflow matching process is based on a structural matching algorithm described in [35].

TABLE 6: SECURE WORKFLOW GENERATION ALGORITHM

```
Algorithm: GenerateSecureWorkflows(Q, ResultSet)
Input:    Q /* query for required placeholder */
Output:   ResultSet /* set of generated secure WFs */
1.  For each known abstract workflow AW do
2.    If StructuralMatching(Q, AW) == true then
3.      RSP := GetSecurityRequirements(Q)
4.      SecPlans := InferSecurityPlans(AW, RSP)
5.      For each security plan SP in SecPlans do
6.        Push (WF,SP) into WST /* WST:stack of WFs */
7.      EndFor
8.    EndIf
9.  EndFor
10. ResultSet := ∅
11. If WST is not empty then
12.   While there are more (WF,SP) pairs in WST do
13.     (W,SP) := get top workflow/ plan pair from WST
14.     A := Get first unassigned activity in W
15.     Services := ServiceDiscovery(A, SP)
16.     If  Services is not empty then
17.       For each service S in Services do
18.         W_S := substitute A with S in W
19.         If another unassigned activity exists in W_S
20.         then Push (W_S,SP) into WST
21.         Else /* all activities of W_S assigned */
22.           ResultSet = ResultSet ∪ {W_S}
23.           Remove (W_S,SP) from WST
24.         EndIf
25.       EndFor
26.     Else /*no services found for unassigned A */
27.       Remove (W,SP) from WS
28.     EndIf
29.   EndWhile
30. EndIf
```

For workflows that match a query Q at an abstract level, GenerateSecureWorkflows creates a list of security plans that could guarantee RSP (see lines 3-4). Each plan in this list specifies either service level security properties (ASPs) for the individual partner services of the matching workflow or for the services of sub workflows that could be used to replace them. Following the identification of the alternative security plans (if any), GenerateSecureWorkflows tries to discover individual services that can be bound to the abstract service in question. This is realised through the call of the algorithm ServiceDiscovery (see line 15). If such services can be identified for all the abstract partner services of an AW, GenerateSecureWorkflows replaces the abstract services in AW with concrete services and/or workflows (see lines 16-25) and returns the instantiated workflow as part of the possible solutions list (see line 30).

5.2 Discovery process

The discovery process is realized by the algorithm ServiceDiscovery, listed in Table 7. This algorithm takes as input the specification of an activity *A* in an abstract workflow and a security plan *SPlan* and finds concrete services (*Servs*) that can be matched with *A* from a structural and a behavioural point of view, whilst also satisfying the security properties specified for *A* in *SecPlan*. The description of A includes the different inputs and outputs of the activity in the abstract workflow. For the activity *Payment* in Fig. 3, for example, the inputs are *items*, *creditCard* and *address* and the outputs are *items*, *address*, *paySuccess*. The activity description in the workflow also specifies the types of these inputs and outputs.

TABLE 7: SERVICE DISCOVERY ALGORITHM

```
Algorithm: ServiceDiscovery(A, SPlan, Servs)
Input:   A /* workflow activity */
         SPlan  /*security plan from A's workflow */
Output:  Servs /*list of Ids of services matching A*/
1.  Servs := {}
2.  ConstructQuery(A, SecPlans, Q)
3.  ConstraintMatching(Q.constraints, Repository, Servs)
4.  StrucuralMatching(Q.structuralPart, Servs, Servs)
5.  BehaviouralMatching(Q.behaviouralPart, Servs, Servs)
```

The first step of the algorithm is to construct a query expressed in *A-SeRDiQueL*, i.e., an XML based query language developed in ASSERT4SOA in [20] (see line 2 of the algorithm). A query in this language has four parts:

(1) A *parameter part* defining generic parameters of the query process (e.g., the matching algorithm that will be used for structural and behavioural matching, the maximum distance threshold for accepting candidate services, whether service composition should be triggered in cases where no single candidate service can match the query).
(2) A *structural part* specifying the interface, i.e., the set of operation signatures of A and the data types of the parameters of these operations.
(3) A *behavioural part* specifying behavioural conditions regarding A that candidate services should match.
(4) A *constraints part* that specifying the security properties and any other constraints, which services that can substitute for A should satisfy. Constraints are specified by logical expressions defining atomic or complex conditions over the contents of service descriptors in

service registries. In the case of security constraints, these conditions refer to service security certificates specified according to the ASSERT4SOA security certificates schema. Every constraint has a *weight* that determines the effect that its satisfaction will have in ranking services in the final answer set of a query, and a *type* that determines whether it is "hard" or "soft". Hard constraints must be satisfied by all services in the answer set of a query. Soft constraints may be violated by services in this set but they affect the ranking of services in this set depending on whether they are satisfied and their weight.

TABLE 8: QUERY FOR PAYMENT (STRUCTURE, BEHAVIOUR)

```
<?xml version="1.0" encoding="utf-8"?>
<tns:StructuralQuery>
<definitions xmlns:tns="http://samples.otn.com" >
<types> <schema>
  <complexType name="Items"> … </complexType>
  <complexType name="CCard"> … </complexType>
  <complexType name="Address"> … </complexType>
   </schema> </types>
<message name="PayReqMes">
  <part name="Items" type="tns:Items" />
  <part name="CreditCard" type="tns:CCard" />
  <part name="Address" type="tns:Address" />
</message> …
<portType name="PaymentService">
  <operation name="payment">
    <input message="tns:PayReqMes"name="PayReq" />
    <output      message="tns:PayResMes"      name="PayRes"/>
  </operation>
  <operation name="CancelPayment">
  <input message="tns:CancelReqM" name="CancelReq" />
    <output    message="tns:CancelResM"    name="CancelRes"/>
</operation> </portType>
</definitions>
</tns:StructuralQuery>
<tnsb:BehaviourQuery>
  <tnsb:LogicalExpression> <tnsb:Condition>
      <tnsb:GuaranteedMember IDREF="payment" />
    </tnsb:Condition></tnsb:Expression>
  <tnsb:LogicalOperator operator="AND" />
    <tnsb:OccursBefore immediate="false"
                        guaranteed="true">
        <tnsb:Member1 IDREF="payment" />
        <tnsb:Member2 IDREF="cancelPayment" />
     </tnsb:OccursBefore> <tnsb:LogicalExpression>
</tnsb:BehaviourQuery>
```

Table 8 shows the structural and behavioural parts of a query for the *Payment* activity in the workflow of Fig. 3, and Table 9 shows the security constraints of the query. In A-SeRDiQueL, the structural part of a query is specified as an abstract service interface specification in WSDL[5]. The behavioural part of the query is specified through constrains regarding the order of execution of the different service operations, whether iterative executions must occur, and the mode of operation execution (i.e., synchronous or asynchronous).

TABLE 9: QUERY FOR PAYMENT (SECURITY CONSTRAINTS)

```
<AssertQuery name="AQ1" type="HARD" assertScope="SINGLE">
<LogicalExpression><Condition relation="SUBSET">
  <Operand1> <AssertOperand facetType="Assert">
    //ASSERTCore/ToC/Assets/Asset
    [@Type='InParameter'or @Type='OutParameter']/Name
    </AssertOperand>
  </Operand1>
  <Operand2>
    <Function name="WSDLLookup"><Arguments>
      <Argument WSDLElementType="message"
                WSDLElementName="creditCard" />
      <Argument WSDLElementType="message"
                WSDLElementName="address" />
      <Argument WSDLElementType=" message"
                WSDLElementName="paySuccess" />
      <Argument WSDLElementType="message"
                WSDLElementName="name" />
      <Argument WSDLElementType="message"
                WSDLElementName="orderSuccess" />
    </Arguments></Function>
  </Operand2>
</Condition>
<LogicalOperator>AND</LogicalOperator>
<LogicalExpression><Condition relation="EQUAL--TO">
  <Operand1> <AssertOperand facetType="Assert">
    //ASSERTCore/SecurityProperty/@PropertyAbstractCategory
    </AssertOperand>
  </Operand1>
  <Operand2> <Constant type="STRING">PSP_C</Constant>
  </Operand2>
</Condition>
</LogicalExpression>
</LogicalExpression>
```

Following the constraint checks, the discovery algorithm carries out the structural matching between a service and a query. This matching is attempted between the operations in the WSDL part of the query and the operations of candidate services. To carry out this matching the algorithm creates graphs representing the query and service operations and the data types of their input and output parameters of the operations. It then matches the graphs by using a variant of the *VF2 algorithm* to detect morphisms between the service and query graphs [35]. The variant allows matches between data type graph edges, whose names have a synonym in WordNet and their origin/destination nodes have matching incoming/outgoing edges. The behavioural matching process checks if the behavioural conditions of a query are satisfied by the behavioural model of a services. This is based on model checking. Further, details of the matching process for non-security querying criteria are beyond the scope of this paper and have been discussed in [35].

5.3 Example

In our *Checkout* process example, let us assume that the service bound to the activity *Payment* does not satisfy the confidentiality property for customer information, i.e., the customer's credit card and address information. To restore this property there are two options: (1) to find an alternative service for *Payment* for which there is a security certificate indicating that the service satisfies the property, or (2) to generate a workflow of services that would satisfy the property. In (2), *Payment* could, for example, be substituted for by a *PayPal* like service, which does not require credit card details. Assuming, however, that the latter service is based on a workflow like the *Express Checkout Purchase* of *PayPal* (www.developer.paypal.com), *Payment* in *Checkout* would need to be replaced by a workflow of three activities: *SetExpressCheckOut*, *ExpressCheckOutPayment* and *PrepareOrder*, as shown in Fig. 5. As in *PayPal*, the former of these activities creates a transaction token that is used to take payment from a customer without passing on his/her credit card details.

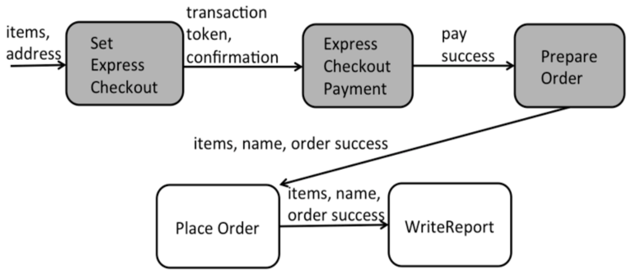
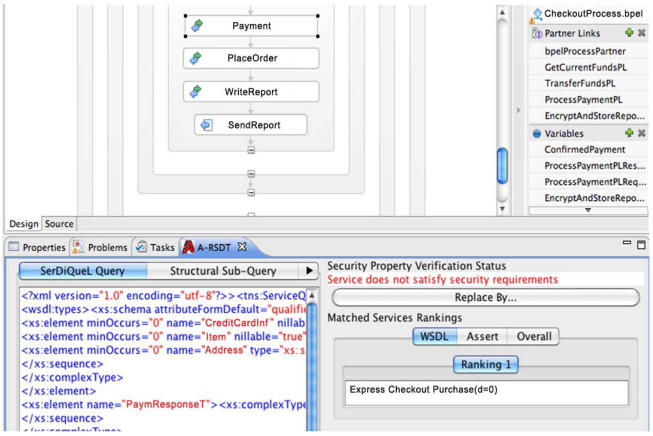


Fig. 5 Modified Checkout process

The change in the *Checkout* workflow shown in Fig. 5 is realised as follows. When the algorithm GENERATESECUREWORKFLOWS is called with a query Q for replacing the service *Payment* in the original *Checkout* process, it retrieves the *PayPal* like payment services workflow shown by the grey activities in Fig. 5. This workflow matches structurally and behaviourally with *Payment* (see line 2 of the algorithm). Subsequently, GENERATESECUREWORKFLOWS infers the alternative security plans for this workflow using the confidentiality property that *Payment* failed to satisfy, and tries to find services that (a) match with the abstract *PayPal* workflow and (b) satisfy the security properties in one security plan.

6 PROTOTYPE

To realise our approach, we have implemented a prototype tool, called *Assurance aware BPEL Designer (A-BPEL Designer)*. *A-BPEL Designer* supports the design and adaptation of SBS workflows in ways that are guaranteed to satisfy given security properties, based on SCO patterns and the algorithms described in the previous sections. *A-BPEL Designer* is based on the BPEL Designer of Eclipse IDE (see *http://www.eclipse.org/bpel/*), which supports the authoring, testing, debugging and deployment of WS-BPEL 2.0 processes. Our tool extends BPEL Designer by implementing the workflow security verification and adaptation capabilities presented in Sect. 4 and 5. To do this, it integrates and makes use of the security certificate based service discovery capabilities of the ASSERT4SOA service discovery engine [20].



```
<?xml version="1.0" encoding="utf-8"?>><tns:ServiceQ
<wsdl:types><xs:schema attributeFormDefault="qualifie
<xs:element minOccurs="0" name="CreditCardInf" nillab
<xs:element minOccurs="0" name="Item" nillable="true"
<xs:element minOccurs="0" name="Address" type="xs:s
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="PaymResponseT"><xs:complexType
</xs:sequence>
```

Security Property Verification Status
Service does not satisfy security requirements

Matched Services Rankings

Express Checkout Purchase(d=0)

Fig. 6 A-BPEL designer

A-BPEL Designer supports the specification of security properties required of BPEL processes and/or specific atomic activities (placeholder) or a group of activities within them. A required security property may also be specified for an asset of a BPEL process partner service (e.g., operation input, operation output). Following this specification, designers can request the verification of the required property for the BPEL process, and the adaptation of the process if the property is not satisfied. Required properties are specified as BPEL process elements annotations and are transformed automatically into A-SerDiQueL queries to enable the verification. Fig. 6 shows the outcome of the verification for the query for the verification of the confidentiality property for the service *Payment* of the *Checkout* process. The *SerDiQuel Query* tab in the figure shows the query generated by *A-BPEL Designer* to verify *Confidentiality*. The tab *Security Property Verification Status* shows that the *Payment* service does not satisfy confidentiality (see status message "Service does not satisfy security requirements"). It also shows alternative services/service compositions that could undertake the functional role of *Payment* in the workflow to satisfy this property, namely the workflow *Express Checkout Purchase* discussed in Sect. 5.

7 EVALUATION

In the following, we present the results of a set of experiments that we conducted to evaluate the performance of framework. The evaluation focused on the time required to verify service workflows, i.e., a key indicator for the scalability of our approach.

7.1 Experimental Setup

To set up the experiments, we used 100 SCO patterns and 100 workflows. The used SCO patterns were generated randomly from a pattern template with three different abstract pattern workflows structures (WF) and five RSP/ASP properties (each WF structure RSP/ASP property have had an equal probability of selection). The abstract WF of the template were: (a) *sequence* WFs, (b) *flow* WFs, and (c) *if-then-else/pick* WFs. From this template, we generated 10 different sets of SBS workflows to be verified. Each of these sets included 10 different workflows of equal size, i.e., they had exactly the same number of activity placeholders/services. The size of workflows in the different sets increased from 10 to 100 services (i.e., *set -1* had 10 workflows with 10 services each, *set-2* had 10 workflows with 20 services each and so on up to *set-10* which had 10 workflows with 100 services each). It should be noted that, although the patterns used in the experiments were synthetic and therefore did not express proven relations between the RSP/ASP properties in them, from a performance evaluation perspective their use of did not compromise the validity of the results.

The evaluation was based on 100 combinations of workflows of 10 different sizes N (N=10, 20, 30, …, 100 services) and sets of SCO patterns of 10 different sizes M (M=10, 20, 30, …, 100 patterns). Each SCO pattern was created as a combination of a WF, one RSP property and as many ASP properties as the activity placeholders in the WF of the pattern.

The verification time recorded for each workflow instance was computed as $VT_S = T_C + T_U$ where (i) T_C is the time required to find a security plan that could verify the RSP property required of it; and (ii) T_U is the time required to confirm that the services bound to the activity placeholders of the given workflow satisfied the ASP properties required of them by the security plan. VT_W was measured up to the point where the first security plan satisfying the RSP property required for a workflow was found or no security plan was found. Also for each individual workflow, VT_W was calculated as the average of 5 different executions of algorithm VERIFYWORKFLOW in order to minimise the possibility of a bias due to interference of background system processes on the machine used for the experiment (e.g., scheduled system jobs, Java garbage collection). The verification time required for a workflow set and a given SCO pattern set (VT_S) was calculated as the average of the VT_W measures across all the 10 different workflows in the workflow set. The tests were executed on an iMac with an Intel Core i3 CPU (3.06 GHz) and 4 GB RAM (DDR3, 1333 MHz) running Mac OS X 10.9.5. During the execution of the experiments no non-system level processes were active on the iMac.

7.2 Results

Fig. 7 shows the average verification execution time (VT_w) for workflow and SCO pattern sets of different sizes upper and lower part of the figure, respectively). The VT_S measures shown in the figure are averages calculated over 250 executions of the algorithm VERIFYWORKFLOW, i.e., 5 executions for 5 different RSP security properties, and for each of the 10 workflows in each workflow set. As the figure indicates even for the most complex case of verifying workflows with 100 services using 100 SCO patterns, the time that it took to verify a workflow did not exceed 300 milliseconds (the exact maximum VT_w was 287.12 milliseconds with a standard deviation of 56.28 milliseconds). Fig. 7 indicates that VT_S increased almost linearly with respect to the workflow size, and did not exceed 150 milliseconds for pattern sets having up to 80 patterns. However, it showed a steeper increase for workflow sets with 90 and 100 services and SCO pattern sets with 90 and 100 patterns. The same effect is also by the graphs showing VT_S with respect to SCO pattern sets of different sizes in the lower part of the figure. For SCO pattern sets with up to 80 patterns, VT_S increased almost linearly with respect to the workflow size and then showed a steeper increase for workflows with more than 50 activities.

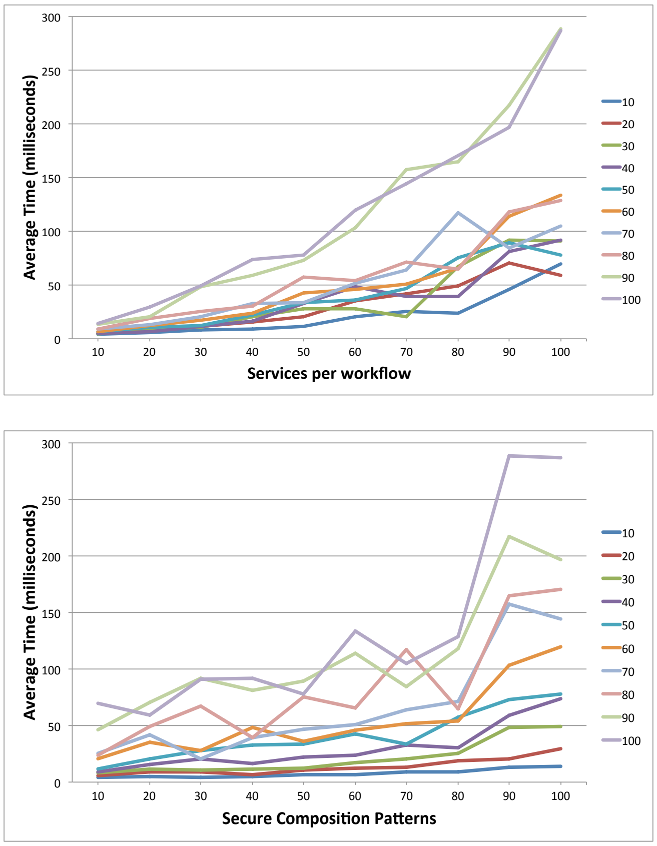


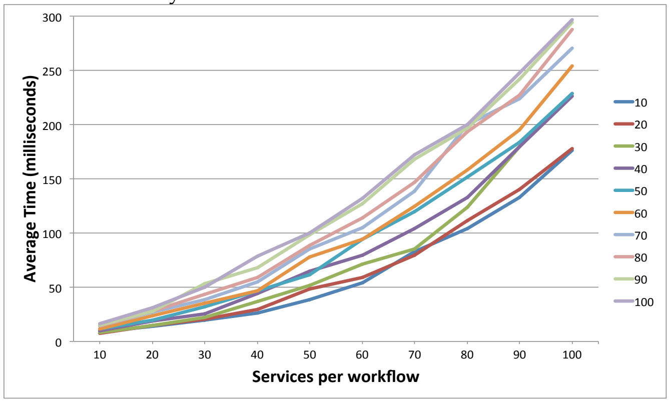
Fig. 7. Average VT_S for different workflow and SCO pattern set sizes

The high variability of VT_S that was observed in these figures was due to fact that the workflow sets included both workflows for which the verification algorithm found a security plan that verified the workflow and stopped once this happened, as well as in cases where no such plan was found and therefore an exhaustive search of all patterns was made.

TABLE 10: SBS WORKFLOWS WITH ONE SECURITY PLAN

		Services Per Workflow									
		10	20	30	40	50	60	70	80	90	100
Secure Composition Patterns	10	14	11	14	11	10	15	11	8	15	18
	20	28	24	20	21	16	27	23	19	23	14
	30	16	23	17	22	22	15	6	24	23	17
	40	16	6	13	11	21	27	14	10	19	17
	50	16	19	7	15	22	13	14	21	21	13
	60	15	15	15	18	22	19	15	16	26	23
	70	19	15	18	23	12	19	18	26	14	15
	80	20	25	22	18	28	18	19	10	22	18
	90	43	31	44	41	34	38	46	40	44	49
	100	41	46	49	46	36	44	40	41	38	48

To confirm the presence of such an effect, we also calculated the average VT_S only for the workflows in each set for which a security plan verifying them was found. Table 9 shows the number of such workflows for the different sizes of workflows and SCO pattern sets. As expected the number of workflows with one verified security plan increased along with the size of the SCO pattern set but was not affected by the size of the workflow.



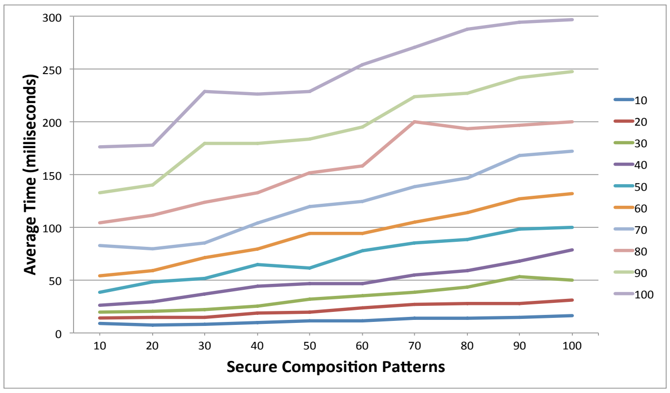


Fig. 8. Average VT_S for workflows and SCO patterns sets of different sizes for workflows with a verified security plan

The average VT_S times computed for workflows with one verified security plan are shown in Fig. 8. The graphs in this figure confirmed the observation regarding the effect of SCO pattern set size and workflow set size on VT_S variability. More specifically, the VT_S variability for workflows with a security plan was lower than for the VT_S variability for workflows. Fig. 8 also shows more consistent trends regarding the effect of workflow and the pattern set size on VT_S than the trends shown in Fig. 7. This variability was caused by the fact that exhaustive search had to be carried workflows with no security plan.

To find a predictive model for the average VT_S time, we fitted alternative trend lines to the data using statistical regression. In all these trend lines, we used the size of the workflow (WFS) and the size of the SCO pattern sets (SCO) as the independent variables and VT_S as the dependent variable.

The regression model with the best fit to the data when considering workflows with and without a verified security plan was the following exponential model:

$$VTs = e^{0.957+0.171*SCO+0.03*WFS} \quad (1)$$

This regression model was able to account for 94% of the variance of VT_S (*F(2,97) = 760.98, overall p < 0.01, SCO p <0.01, WSF p <0.01, R^2 = 0.94*), and was validated for the homoscedasticity of the residual VT_S errors using the Breusch-Pagan homoscedasticity test (p=0.391) [10]. The model is consistent with the worst-case scenario in which INFERSECURITYPLANS algorithm has to search for all possible ways in which it may match the structure of the workflow of an SCO pattern with the SBS workflow to be verified. Due to the prefix-based encoding of pattern and SBS workflows (see line 6 in Table 2), at each recursion cycle of the INFERSECURITYPLANS algorithm the it is possible to perform this match in N steps where N is the number of complex activities of the SBS workflow. However, even if a match is found, there can be non-atomic activities (i.e., sub-workflows) in the matched structure of the SBS workflow that will need to be matched with patterns again and up to the point where no more recursive matching attempts will be possible.

When only workflows with a verified security plan were considered, the best model that was found was the following linear regression model:

$$VTs = 458.49 - 1.916 * SCO + 2.687 * WFS \quad (2)$$

The above regression model was able to account for 92.3% of the variance of VT_S (*F(2,97) = 585.19, overall p < 0.01, SCO p <0.01, WSF p <0.01, R^2 = 0.923*), and was also valid with respect to homoscedasticity based on the Breusch-Pagan test (p=0.43). It should be noted, however, that the linear decrease of VT_S along with the size of the SCO pattern that is indicated by the model is not plausible. Hence, model (2) should not be used as more conclusive evidence than the experimental results shown in Fig. 9 and 10.

Overall, our results are promising although they arise from an initial evaluation. More specifically, the maximum average VT_S time predicted for the verification of an SBS workflow by model (1) is 293.98 milliseconds. This prediction indicates that it is feasible to verify the security of complex SBS workflows (i.e., workflows involving as many as 100 services) with complex SCO pattern sets (i.e., sets involving as many as 100 SCO patterns). We believe that our experiments are realistic in testing the efficacy boundaries of our approach since, although someone may encounter cases with process models involving more than 100 services, it would be unlikely to use SCO pattern sets with significantly more than 100 patterns.

8 RELATED WORK

SBS security has been the focus of several strands of research focusing on: (a) the verification of the security of SBSs, and (b) the design of secure SBSs.

Research approaches focusing on the verification of security properties of during service workflows design, typically rely on the use of formal analysis (i.e., model checking or theorem proving). AutoFocus [6], for example, assumes the specification of SBS systems in UML and the security properties to be verified in CTL [18]. These specifications are transformed into the input language of the SVM model checker, which is used to verify the properties. The properties supported by [6] are authentication and authorisation. Dong et al [8] have used security design patterns to model inter-process communications as UML sequence diagrams. The models resulting from this process are converted into the formal language CCS [21], and verified for security properties using model checking. Brucker et al [3] also use model checking to verify service compositions modelled in BPMN [7] against security properties specified in LTL [18]. Their approach was implemented as of NetWeaver (www.sap.com/uk/community/topic/netweaver.html). Nakajima [22] verified BPEL processes annotated with with security labels, using model checking. This was based on transforming annotated BPEL processes into Promela and using SPIN to detect information leakage. Most of the above approaches support a limited number of properties (e.g., [3], [6], [8]). Also verification based on them is likely to be computationally intractable for large SBS systems (note that [3], [6], [8] and [22] do not provide any experimental performance results).

Bartoletti et al [2] have developed a typed extension of λ-calculus to specify models of service compositions and check the security-related activities in them (e.g., opening a socket connection). [2] provide a formal basis for modelling verifiable service orchestrations but with some limitations (e.g., no support for incremental verification that is required at runtime and potential computational intractability due to formal reasoning). The approach presented in [31] validates access control policies for data transmitted within service compositions, using information flow control rules to define security policy access privileges to be enforced when passing data from one service to another. An experimental validation of this approach has shown that the time required for generating compositions increases exponentially with the composition size (e.g., 30 secs for small sequential compositions of 14 services).

Lelarge et al [15] have used planning techniques to compose workflows that are compliant with lattice-based access control models (e.g. multi-level secure systems) and analysed the complexity of the process for different sets of security constraints (e.g., whilst the composition problem is NP-complete, totally ordered constraints lead to linear time composition).

Albanese et al [1] find compositions of modules (services) that satisfy required security properties using logic programming. In this approach, security properties are defined for service interfaces and can weak, i.e., satisfied in an approximate manner. This is similar to the weak constraints used in workflow generation in our approach. This approach has been shown to be NP-hard.

Salnitri et al [30] also support checks of security policies against BPMN processes. Verification in this approach is not formal; it involves checking security policies expressed as SecBPMN queries against SecBPMN-ml specifications (i.e., security enhanced BPMN models) by searching for the existence of paths in processes that satisfy the policy. An experimental evaluation showed that the verification time increases exponentially with the size of processes and linearly to the number of properties [30].

Significant work there has also been on the generation of secure service workflows. Frankova et al [12], for example, model security requirements in a formal goal oriented requirements language, called *SI*/Secure Tropos*, and – following specific verification checks – they produce a Secure BPEL workflow through an iterative process of refinement.

BPA-Sec4Cloud [17] transforms BPMN processes annotated with security requirements into cloud platform specific services and BPEL processes that comply with the security requirements at execution time. The Sec-MoSC tool [33] supports the addition of required security properties to BPMN processes [7] and the selection of default mechanisms for implementing them. The BPMN processes along with the selected mechanisms are transformed to BPEL and a security engine is used to realise the security mechanisms added to the process. Charfi and Menzini [4] use an aspect-oriented approach to integrate security specifications in BPEL processes, and use them to identify security functionalities of special security services, and integrate them into the process to enforce security.

Overall, our approach might not be as general as approaches that use full formal analysis to support workflow verification and generation. This is because its ability to perform these two tasks depends on the availability of proven SCO patterns for the security property of interest. However, our review of the literature shows that our approach appears to have significantly better performance than those formal analysis approaches for which experimental results have been reported, Furthermore, pur approach does not require as complex security property modelling from SBS designers as formal approaches, since the relevant property and workflow models are available in SCO patterns.

9 CONCLUSION

In this paper, we have described a framework for verifying the security of SBS workflows based on patterns. The verification process is realised by inferring security properties of the individual services of the workflow (ASP) which, when satisfied, would guarantee workflow level security properties (RSP), and checking if the individual services have such properties.

The results of an initial evaluation of our approach were positive: even for workflows with 100 services and large SCO patterns sets (100 patterns) verification was performed in less than 0.3 seconds. This efficiency comes at the expense of completeness in verification. More specifically, our approach is not complete since it will only be able to verify an RSP property if: (a) there is an SCO pattern P that can guarantee RSP; (b) the abstract workflow structure of P, i.e., matches the workflow of interest; and (c) the partner services the workflow of interest that match the activity placeholders of the pattern workflow satisfy the ASP properties required of them by P. Hence, the identification and development of comprehensive sets of SCO patterns is a pre-requisite for the effectiveness and applicability of our approach. Identifying comprehensive pattern sets requires further research focusing not only on finding new patterns but also on establishing the right methodology for doing so and for evaluating the sufficiency of the pattern sets developed using it.

ACKNOWLEDGMENT

The work reported in this paper has been partially funded by the EU F7 project ASSERT4SOA (grant no. 257351) and the H2020 project CYBERSURE (grant no. 734815).

References

[1] Albanese, M., Jajodia, S., and Molinaro, C. (2013). A Logic Framework for Flexible and Security-Aware Service Composition. *IEEE 10th Int. Conf. on Autonomic and Trusted Computing*, pp. 337-346.

[2] Bartoletti, M., Degano, P., and Ferrari, G. L. (2006). Types and effects for secure service orchestration. *In 19th IEEE Computer Security Foundations Workshop.* pp. 57-69.

[3] Brucker, A. D., Compagna, L., and Guilleminot, P. (2014). Compliance Validation of Secure Service Compositions. *In Secure and Trustworthy Service Composition*, pp. 136-149. Springer Int. Pub.

[4] Charfi, A., and Mezini, M. (2005). Using aspects for security engineering of web service compositions. *In Proc. of IEEE Int. Conf. on Web Services,* pp. 59-66.

[5] Chinnici, R., et al. (2007). Web services description language (wsdl) version 2.0 part 1: Core language. *W3C recommendation*.

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

[7] Dijkman, R. M., Dumas, M., & Ouyang, C. (2008). Semantics and analysis of business process models in BPMN. *Information and Software technology*, *50*(12), 1281-1294.

[8] Dong, J., Peng, T., and Zhao, Y. (2010). Automated verification of security pattern compositions. *Information and Software Technology*, 52(3), 274-295. DOI: 10.1016/j.infsof.2009.10.001

[9] Drools. Available from: http://www.drools.org/

[10] Dufour, J. M., et al. (2004). Simulation-based finite-sample tests for heteroscedasticity and ARCH effects. *Journal of Econometrics*, *122*(2), 317-347.

[11] Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern / many object pattern match problem. *Artificial intelligence*, 19(1), 17-37. DOI: 10.1016/0004-3702(82)90020-0

[12] Frankova, G., Massacci, F., and Seguran, M. (2007). From Early Requirements Analysis towards Secure Workflows. *In Proc. of IFIP Joint ITrust and PST Conferences on Privacy, Trust Management and Security,* Springer.

[13] Gürgens, S., Ochsenschläger, P., & Rudolph, C. (2005). On a formal framework for security properties. *Computer Standards & Interfaces*, *27*(5), 457-466.

[14] IBM, IBM Industry Packs. Available from: http://www-01.ibm.com/software/integration/business-process-manager/industry-packs/library/documentation/

[15] Lelarge, M., Liu, Z., and Riabov, A. V. (2006). Automatic composition of secure workflows. *In Proc. of 3rd Int. Conf. on Autonomic and Trusted Computing, pp. 322-331.*

[16] Lin, F. (2008). Situation calculus. *Foundations of Artificial Intelligence*, *3*, 649-669.

[17] Lins, F., et al., (2015). Automation of service-based security-aware business processes in the Cloud. *Computing*, pp.1-24.

[18] Maidi, M. (2000). The common fragment of CTL and LTL. *In Proc. 41st Annual Symposium on Foundations of Computer Science,* pp. 643-652.

[19] Martin, D., et al. (2004). OWL-S: Semantic markup for web services. *W3C member submission*, *22*, 2007-04. Available from: https://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/

[20] Mahbub K. et al., Asserts Aware Service Query Language and Discovery Engine, Deliv. D.2.1, ASSERT4SOA Project, available from: http://www.assert4soa.eu/deliverable/D2.1.pdf

[21] Milner, R. (1989). Communication and concurrency. Prentice-Hall, Inc. ISBN:0-13-115007-3

[22] Nakajima, S. (2004). Model-checking of safety and security aspects in web service flows. In *Int. Conf. on Web Engineering*, pp. 488-501. Springer Berlin Heidelberg.

[23] Pino, L. (2015). *Security Aware Service Composition* (Doctoral dissertation, City University London). Available from: http://openaccess.city.ac.uk/13170/

[24] Pino, L., and Spanoudakis, G. (2012). Constructing secure service compositions with patterns. *In IEEE 8th World Congress on Services*, 2012, pp. 184-191.

[25] Pino, L., and Spanoudakis, G. (2012). Finding secure compositions of software services: Towards a pattern based approach. *In 5th IFIP Int. Conf. on New Technologies, Mobility & Security*.

[26] Pino, L., Mahbub, K., and Spanoudakis, G. (2014). Designing Secure Service Workflows in BPEL. *In Proc. of the Int. Conference on Service-Oriented Computing*, pp. 551-559.

[27] Pino, L., Spanoudakis, G., Gürgens, S., Fuchs, A., and Mahbub K., (2012). ASSERTS aware Service Orchestration Patterns, Deliverable D2.2. ASSERT4SOA Project. Available from: http://www.cspforum.eu/D2.2-revised.pdf

[28] Pino, L., et al., (2015) Generating Secure Service Compositions. In *Communications in Computer and Information Sciences, Vol 512*, (eds) Helfert M., et al., Springer International Pub.

[29] RossettaNet http://www.edibasics.co.uk/edi-resources/document-standards/rosettanet/

[30] Salnitri, M., et al. (2015). Designing secure business processes with secBPMN. *Software & Systems Modelling*, 1-21.

[31] She W., et al., (2013). Security-aware service composition with fine-grained information flow control. *IEEE Transactions on Services Computing*, 6(3), pp.330-343.

[32] Society for Worldwide Interbank Financial Telecommunication (SWIFT). Available from: http://www.swift.com/

[33] Souza, A. R., et al. (2009). Incorporating Security Requirements into Service Composition: From Modelling to Execution. *In Proc. of 7th Int. Joint Conf. on Service-Oriented Computing,* pp. 373-388.

[34] Zakinthinos, A., and Lee, E. S. (1997). A general theory of security properties. *In Proc. of IEEE Symposium on Security and Privacy*, pp. 94-102. IEEE. DOI: 10.1109/SECPRI.1997.601322

[35] Zisman, A., et al, (2013). Proactive and reactive runtime service discovery: a framework and its evaluation. *Software Engineering, IEEE Transactions on*, *39*(7), pp.954-974.

[36] Workflow Management Coalition. Terminology and Glossary (1999), Technical Report Document Number WFMC-TC-1011, Issue 3.0, 1999, Available from: http://www.workflowpatterns.com/documentation/documents/TC-1011_term_glossary_v3.pdf

Luca Pino BSc, MSc, PhD. Luca Pino got his PhD from City, University of London and is now a software engineer at Masabi. His interests span services and SBSs composition, security and automation.

George Spanoudakis. Bsc, Msc, PhD. George Spanoudakis is Professor at City, University of London. His research interests are in software systems security and service oriented systems.

Maria Krotsiani. BSc, MSc, PhD. Maria Krotsiani is a postdoctoral research fellow at City, University of London. Her research interests are in cloud security and continuous security certification.

Khaled Mahbub. B.Eng, M.Eng, PhD. Khaled Mahbub is a Senior Lecturer at Birmingham City University. His research focus on automated software engineering, service based and cloud computing.