A Robust Framework for Securing Composed Web Services

Najah Ben Said, Takoua Abdellatif, Saddek Bensalem, Marius Bozga

A Robust Framework for Securing Composed Web Services *

Najah Ben Said[1], Takoua Abdellatif[3], Saddek Bensalem[1], and Marius Bozga[2]

[1] Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France
[2] CNRS, VERIMAG, F-38000 Grenoble, France
[3] Tunisia Polytechnic School, University of Carthage, Tunis, Tunisia

Abstract. This paper proposes a framework that automatically checks and configures data security in Web Services starting from high level business requirements. We consider *BPEL*-based composed Web Services. *BPEL* processes and initial security parameters are represented as component-based models labeled with security annotations. These models are formal and enable automated analysis and synthesis of security configurations, under the guidance of the service designer. The security property considered is the non-interference. The overall approach is practical since security is defined separately from functional processes and automatically verified. We illustrate its utility to solve intricate security problems using a smart grid application.

Keywords: component-based systems, information flow security, non-interference, dependency flow graph, automated verification.

1 Introduction

With the expansion of Web Services (WS) [1] deployed on the enterprise servers, cloud infrastructures and mobile devices, Web Service composition is currently a widely used technique to build complex Internet and enterprise applications. Orchestration languages like *BPEL* [2] allow rapidly developing composed WS by defining a set of activities binding sophisticated services. Nevertheless, advanced security skills and tools are required to ensure critical information security. Indeed, it is important to track data flow and prevent illicit data access by unauthorized services and networks; this task can be challenging when the service is complex or when the composition is hierarchical (the service is composition of composed services and atomic services). For example, a classical travel organization WS has to keep a client's destination secret as messages are exchanged between different services like travel agency services and the payment service. Each piece of information depending on the destination, like ticket price, can lead to the secret disclosure if it is not protected. WS security standards [3,4]

* The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement ICT-318772 (D-MILS).

provide information flow security solutions for point-to-point inter-service communication but fall short in ensuring end-to-end information flow security in composed services. Furthermore, the *BPEL* language does not state any rules on how to properly apply security mechanisms to services. Generally, developers manually set up their system security configuration parameters which can be tedious and error-prone.

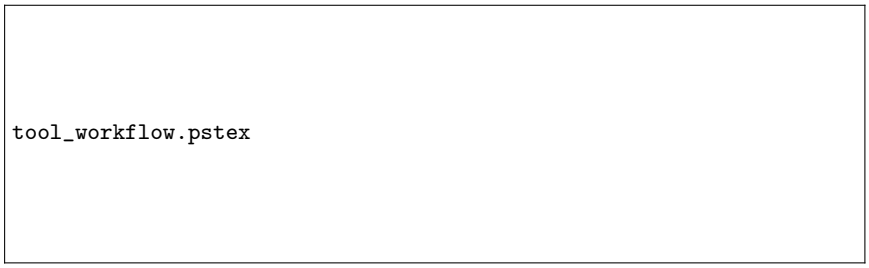

Fig. 1. Information flow analysis overview with component-based model

Information flow control, particularly the non-interference [5] property check, is an alternative, more robust, approach than applying access control for point to point communication. Indeed, it allows tracking information propagation in the entire system and prevents secret or confidential information from being publicly released. Information flow control relies on annotating system data with specific levels of security and uses specific methods for checking non-interference, that is, absence of leakage between different levels. Nonetheless, providing annotations and establishing their correctness are equally difficult, especially for distributed implementations, where only code is available and no higher-level abstractions exist to give a better view and easier way of correction.

In this paper, we propose a robust tool assisting a designer ensuring end-to-end information flow security in WS composition. Figure 1 shows a workflow overview of this tool. The service designer describes in *BPEL* his process and defines partial security constraints in a configuration file. The constraints are expressed as authorization rights, that is, a list of services owners and authorized readers for a subset of critical data. The *BPEL* process and the configuration information are then automatically transformed into a component-based framework. This framework was first adapted to abstract distributed WS orchestration to a component-based model where all Web services are transformed into atomic components communicating through interactions by sending and receiving variables and second, to synthesize security configuration for total system variables with respect to security constraints by considering all implicit and explicit data dependencies in the system. The calculated configuration is optimal that is only data that need to be protected is configured as critical and its security level is minimal. It is indeed, very important to reduce the security processing overhead like cryptography encryption and decryption, signature calculation, certificate verification, etc. In case a total configuration file is generated by the tool, the system information flow is then considered non-interferent with respect to

initially defined configuration. Otherwise, the system is interferent and system designer has to re-define the input initial configuration. As a security advisor, automated configuration synthesis allows designers, developers and administrators to focus on functional constraints and be confident that their secret data is protected and people privacy respected. The proposed framework is based on a formal compositional security model. It is implemented and tested on a smart grid application. This application shows the practical usage of our tool since only basic security skills are required and WS standards are respected.

The paper is structured as follows. Section 2 presents the functional and security aspects of the adopted component-based framework. In Section 3, we present a practical compositional approach to synthesize security configurations in component-based system models. In Section 4, we apply this approach to Web Service orchestration. We consider a transformation from *BPEL* orchestration language to the component-based framework and we present the adopted security annotation model and the tool implementation. In Section 5, we provide a use-case as illustrative example. Finally, Section 6 discusses the related work and Section 7 concludes and presents some lines for future work.

2 Component-Based Model

In the scope of this work, we consider systems composed of atomic components interacting through point-to-point communications. Atomic components are in form of finite automata extended with data. Communication is synchronous and directed between one sender and one receiving component. Regarding security, we consider transitive information flow policies expressed on system variables and we focus on non-interference properties.

The proposed model is general enough to deal with information flow security from a practical point of view for commonly used programming languages and/or modeling frameworks such as *BPEL*. Nevertheless, it should be mentioned that this model is actually a strict subset of the *secureBIP* component model previously introduced in [6,7]. The latter considers additional coordination mechanisms through multiparty interactions as well as different definitions of non-interference. For the sake of readability we recall hereafter the key concepts and we re-formulate the key results which are useful in our precise context, on the restricted subset considered.

2.1 Preliminaries

Let $\mathbb{D} = \{D_j\}_{j\in\mathcal{J}}$ be a universal set of data domains (or data types) including the Boolean domain $\mathbb{D}_{Bool} = \{true, false\}$. Let $\mathbb{E}xpr$ be an universal set of operators, that is, functions of the form $op : \times_{i=1}^{m} D_{j_i} \rightarrow D_{j_0}$, where $m \geq 0$, $D_{j_i} \in \mathbb{D}$ for all $i = 0, m$. We consider typed variables $x : D$ where x denotes the name of the variable and $D \triangleq dom(x) \in \mathbb{D}$ its domain of values. We define expressions e as either (i) constant values $u \in \cup_j D_j$, (ii) variables $x : D$ or (iii) composed expressions $op(e_1, ..., e_m)$ constructed by applying operators op

on sub-expressions $e_1, ..., e_m$ such that, their number and their domains match exactly the domain of *op*. We denote by $use(e)$ the set of variables occurring in expression e and by $\mathbb{E}xpr[X]$ the set of expressions constructed from a set of variables X and operators in $\mathbb{E}xpr$. We denote by $\mathbb{A}sgn[X]$ the set of assignments to variables in X, that is, any subset $\{(x_i, e_i)\}_{i\in I} \subseteq X \times \mathbb{E}xpr[X]$ where $(x_i)_{i\in I}$ are all distinct. An assignment (x, e) is denoted by $x := e$.

Given a set of variables X, we define valuations V of X as functions $V : X \to \cup_{j\in J} D_j$ which assign values to variables, such that moreover, $V(x) \in dom(x)$, for all $x \in X$. We denote by $V[u/x]$ the valuation where variable x has assigned value u and any other variable has assigned the same values as in V. For a subset $Y \subseteq X$, we denote by $V_{|Y}$ the restriction of V to variables in Y.

Given an expression $e \in \mathbb{E}xpr[X]$ and a valuation V on X we denote by $e(V)$ the value obtained by evaluating the expression according to values of X on the valuation V. Moreover, given an assignment $a \in \mathbb{A}sgn[X]$ and a valuation V of X we denote by $a(V)$ the new valuation V' obtained by executing a on V, formally $V'(x) = e(V)$ iff $x := e \in a$ and $V'(x) = V(x)$ otherwise.

2.2 Operational Model

An atomic component B is a tuple (Q, P, X, T) where Q is a set of states, X is a set of local variables, P is a set of ports (or action names) and T is a set of transitions. We distinguish respectively input ports $P^{in} \subseteq P$ and output ports $P^{out} \subseteq P$ and we assume they are disjoint, $P^{in} \cap P^{out} = \emptyset$. Every input or output port $p \in P^{in} \cup P^{out}$ is associated to a unique variable $var(p) \in X$. Every transition $t \in T$ is a tuple (q, p, g, a, q') where $q \triangleq src(t), q' \triangleq dst(t) \in Q$ are respectively the source and the target states, $p \triangleq port(t) \in P$ is a port, $g \triangleq guard(t) \in \mathbb{E}xpr[X]$ is the enabling condition and $a \triangleq asgn(t) \in \mathbb{A}sgn[X]$ is the assignment of t.

In our model, atomic components have exclusive access on their variables. Interactions between components take place only through explicit input/output binary connectors. A connector defines a static communication channel from one output port p^{out} of a sender component B to an input port p^{in} in a receiver component $B' \neq B$. The connector is denoted by the tuple (p^{out}, p^{in}). Intuitively, when communication takes place, the value of $var(p^{out})$ is assigned to $var(p^{in})$.

Figure 2 provides examples of atomic components. The *Producer* component contains two states l_1 and l_2 and one output port *out*. The transition labelled with port *produce* takes place only if the guard $[w \leq 3]$ is true. Then, the variable x is updated by executing the assignment $x := 3x + 1$.

We denote by $\Gamma(B_1, ..., B_n)$ the composition of a set of atomic components $B_i = (Q_i, X_i, P_i, T_i)_{i=1,n}$ through a set of connectors Γ. For the sake of simplicity, we tacitly assume that every input and output port of every B_i is used in exactly one connector in Γ. The operational semantics of a composition is defined as a labelled transition system $(\mathcal{Q}, \mathcal{A}, \to)$ where states correspond to system configurations and transitions to internal steps or communication through connectors. A system configuration $\langle \boldsymbol{q}, \boldsymbol{V} \rangle$ in $\mathcal{Q}$ where $\boldsymbol{q} = (q_1, ..., q_n)$, $\boldsymbol{V} = (V_1, ..., V_n)$ is obtained from component configurations (q_i, V_i) where $q_i \in Q_i$ and V_i is a

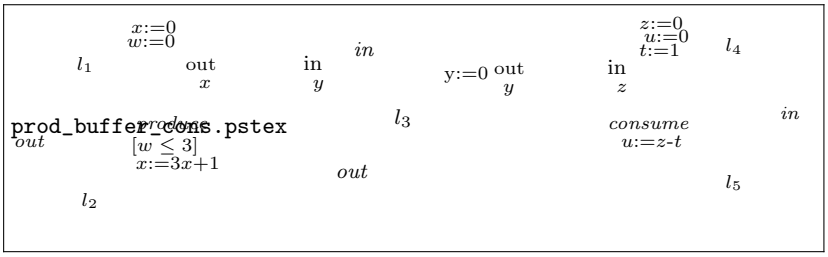

Fig. 2. A *Producer-Buffer-Consumer* example

valuation of X_i, for all $i = 1, n$. The set of labels $\mathcal{A}$ is defined as $\Gamma \cup \{\tau\}$, that is, either communication on connectors or internal action (τ). The set of transitions $\rightarrow \subseteq \mathcal{Q} \times \mathcal{A} \times \mathcal{Q}$ between configurations are defined by the following two rules:

$$\textsc{Inter}\ \frac{(q_i, p_i, g_i, a_i, q_i') \in T_i \quad p_i \notin P_i^{in} \cup P_i^{out} \quad g_i(V_i) = true \quad V_i' = a_i(V_i) \quad \forall k \neq i.\ (q_k', V_k') = (q_k, V_k)}{\langle (q_1, ..., q_n), (V_1, ..., V_n) \rangle \xrightarrow{\tau} \langle (q_1', ..., q_n'), (V_1', ..., V_n') \rangle}$$

$$\textsc{Comm}\ \frac{\begin{array}{c}(p_i^{out}, p_j^{in}) \in \Gamma \quad (q_i, p_i^{out}, g_i, a_i, q_i') \in T_i \quad (q_j, p_j^{in}, g_j, a_j, q_j') \in T_j \\ g_i(V_i) = g_j(V_j) = true \quad u = V_i(var(p_i^{out})) \quad V_i' = a_i(V_i) \\ V_j' = a_j(V_j[u/var(p_j^{in})]) \quad \forall k \neq i, j.\ (q_k', V_k') = (q_k, V_k)\end{array}}{\langle (q_1, ..., q_n), (V_1, ..., V_n) \rangle \xrightarrow{p_i^{out} p_j^{in}} \langle (q_1', ..., q_n'), (V_1', ..., V_n') \rangle}$$

The system evolves either by performing asynchronously an internal step of some component B_i (INTER rule) or by performing a synchronous communication between two components B_i, B_j involving respectively ports p_i^{out}, p_j^{in} related by a connector in Γ (COMM rule). Transitions are executed only if guards are evaluated to *true* in the current configuration. As usual, next configurations are obtained by taking into account variable assignments and communication.

A run ρ of the system $\Gamma(B_1, ..., B_n)$ is a finite sequence $\langle \boldsymbol{q}_0, \boldsymbol{V}_0 \rangle\ \alpha_1\ \langle \boldsymbol{q}_1, \boldsymbol{V}_1 \rangle\ \alpha_2\ ...\ \alpha_\ell\ \langle \boldsymbol{q}_\ell, \boldsymbol{V}_\ell \rangle$ where $\langle \boldsymbol{q}_{k-1}, \boldsymbol{V}_{k-1} \rangle \xrightarrow{\alpha_k} \langle \boldsymbol{q}_k, \boldsymbol{V}_k \rangle$ for all $k = 1, \ell$. The set of all runs starting from a configuration $\langle \boldsymbol{q}_0, \boldsymbol{V}_0 \rangle$ are denoted by $Runs\langle \boldsymbol{q}_0, \boldsymbol{V}_0 \rangle$.

Finally, for a run ρ and a subset of variables $Y \subseteq \cup_{i=1}^{n} X_i$ we denote by $tr(\rho, Y)$ the trace of ρ with respect to Y. Traces represent what is actually observable from a trace by having access to variables in Y. They are inductively defined for runs as follows:

$$tr(\langle \boldsymbol{q}, \boldsymbol{V} \rangle, Y) = \langle \boldsymbol{q}, \boldsymbol{V}_{|Y} \rangle$$

$$tr(\langle \boldsymbol{q}, \boldsymbol{V} \rangle\ \alpha\ \rho', Y) = \begin{cases} tr(\rho', Y) \\ \quad \text{if } \alpha = \tau \text{ and } tr(\rho', Y) \text{ starts with } \langle \boldsymbol{q}, \boldsymbol{V}_{|Y} \rangle \\ \langle \boldsymbol{q}, \boldsymbol{V}_{|Y} \rangle\ \alpha\ tr(\rho', Y) \\ \quad \text{otherwise} \end{cases}$$

where $\boldsymbol{V}_{|Y}$ denotes $(V_{1|Y_1}, ..., V_{n|Y_n})$ for $Y_i = Y \cap X_i$, for all $i = 1, n$. Finally, for a trace $tr(\rho, Y)$ we define the set $Enable(tr(\rho, Y))$ of configurations which are enabling the same trace on alternative runs, formally:

$$Enable(tr(\rho, Y)) = \{\ \langle \boldsymbol{q}_0, \boldsymbol{V}_0 \rangle \mid \exists \rho' \in Runs\langle \boldsymbol{q}_0, \boldsymbol{V}_0 \rangle,\ tr(\rho, Y) = tr(\rho', Y)\ \}$$

2.3 Security Model

We consider transitive information flow policies expressed on system variables and we focus on the non-interference properties. We restrict ourselves to confidentiality and we ensure that no illegal flow of information exists between variables having incompatible security levels.

Formally, we represent security domains as finite lattices $(\mathbb{S}, \leq)$ where $\mathbb{S}$ denotes the security levels and $\leq$ the *flows to* relation. For a level s, we denote by $[-, s]$ (resp. by $[s, -]$) the set of levels allowed to flow into (resp. from) s. Moreover, for any subset $S \subseteq \mathbb{S}$, we denote by $\sqcup S$ (resp. $\sqcap S$) the unique least upper (resp. greatest lower) bound of S according to $\leq$.

Let $\Gamma(B_1, ..., B_n)$ be a system and let $X = \cup_{i=1}^{n} X_i$ (resp. $P = \cup_{i=1}^{n} P_i$) be the set of all components variables (resp. ports). A security annotation on variables is a function $\sigma : X \to \mathbb{S}$ which associates security levels to variables. We denote by $\sigma^{-1} : 2^{\mathbb{S}} \to 2^X$ the pre-image of σ, defined as $\sigma^{-1}(S) = \{x \in X \mid \sigma(x) \in S\}$, for all $S \subseteq \mathbb{S}$. For any s, define $Y_s = \sigma^{-1}([-, s])$, the set of variables having security levels at most s. For a security level s, we denote by $\approx_s$ the indistinguishability relation on configurations at level s defined by $\langle \boldsymbol{q}_1, \boldsymbol{V}_1 \rangle \approx_s \langle \boldsymbol{q}_2, \boldsymbol{V}_2 \rangle$ iff $\boldsymbol{q}_1 = \boldsymbol{q}_2$ and $\boldsymbol{V}_{1|Y_s} = \boldsymbol{V}_{2|Y_s}$. That is, configurations are identical on control states and up to variables with security levels at most s. For a set of configurations $C \subseteq \mathcal{Q}$, we denote by $[\![C]\!]_s = \{c' \in \mathcal{Q} \mid \exists c \in C.\ c' \approx_s c\}$. We are now ready to define the security criterion for an annotated system.

Definition 1. *A security annotation σ is secure for a system $\Gamma(B_1, ..., B_n)$ and initial configurations Init iff* $\forall s \in \mathbb{S}.\ \forall \langle \boldsymbol{q}_0, \boldsymbol{V}_0 \rangle \in Init.\ \forall \rho \in Runs\langle \boldsymbol{q}_0, \boldsymbol{V}_0 \rangle$

$$Enable(tr(\rho, Y_s)) = [\![Enable(tr(\rho, Y_s))]\!]_s$$

Intuitively, the definition states that for any security level s, no additional information is obtained by observing traces with respect to variables Y_s behind the equivalence $\approx_s$. Or, any two indistinguishable initial states enable precisely the same set of traces with respect to Y_s. If this would not be the case for let say, $(\boldsymbol{q}_0, \boldsymbol{V}_0) \approx_s (\boldsymbol{q}_0, \boldsymbol{V}'_0)$, then one could find a run $\rho_0 \in Runs\langle \boldsymbol{q}_0, \boldsymbol{V}_0 \rangle$ such that no run $\rho' \in Runs\langle \boldsymbol{q}_0, \boldsymbol{V}'_0 \rangle$ had the same trace with respect to Y_s. But then, this means $(\boldsymbol{q}_0, \boldsymbol{V}_0) \in Enable(tr(\rho_0, Y_s))$ whereas $(\boldsymbol{q}_0, \boldsymbol{V}'_0) \notin Enable(tr(\rho_0, Y_s))$. and consequently $Enable(tr(\rho, Y_s)) \subsetneq [\![Enable(tr(\rho, Y_s))]\!]_s$.

The following proposition defines static conditions ensuring that a security annotation on variables is secure for a system.

Proposition 1. *A security annotation σ is secure for a system $\Gamma(B_1, ..., B_n)$ with an arbitrary non-empty set of initial configurations Init whenever*

– *all local transitions* t *in components* B_i *are* sequentially consistent
  $\forall (x := e) \in asgn(t).\ \forall y \in use(e) \cup use(guard(t)).\ \ \sigma(y) \leq \sigma(x)$
– *all components* B_i *are* port deterministic *i.e., for all transitions* t_1, t_2
  $src(t_1) = src(t_2)\ \wedge\ port(t_1) = port(t_2)\ \Rightarrow\ guard(t_1) \wedge guard(t_2) \equiv false$

and moreover, there exists a security annotation on ports $\varsigma : P \to \mathbb{S}$ *such that:*

– *ports of all* causal *local transitions* t_1, t_2 *have increasing levels of security*
  $dst(t_1) = src(t_2)\ \Rightarrow\ \varsigma(port(t_1)) \leq \varsigma(port(t_2))$
– *ports of all* conflicting *local transitions* t_1, t_2 *have the same level of security*
  $src(t_1) = src(t_2)\ \Rightarrow\ \varsigma(port(t_1)) = \varsigma(port(t_2))$
– *variables and ports are consistently annotated on all local transitions* t
  $\forall (x := e) \in asgn(t).\ \forall y \in use(guard(t)).\ \sigma(y) \leq \varsigma(port(t)) \leq \sigma(x)$
– *variables and ports are consistently annotated on connectors*
  $\forall (p^{out} p^{in}) \in \Gamma.\ \sigma(var(p^{out})) \leq \varsigma(p^{out}) = \varsigma(p^{in}) \leq \sigma(var(p^{in}))$

Proof. (Sketch) It can be shown that the conditions above imply the *unwinding conditions* of [8] for indistinguishability $\approx_s$ at security level s. In turn, unwinding conditions are guaranteeing non-interference and therefore security as defined in Definition 1. A detailed proof is available in [6,7] for a slightly more general component-based model allowing multiparty interactions between components.

3 Configuration Synthesis

The configuration synthesis problem is defined as follows. Given a partial security annotation of a system, extend it towards a complete annotation which is provable secure according to Proposition 1, or show that no such annotation actually exists. We assume that system components are port deterministic.

We rely on flow dependency graphs as an intermediate artifact for solving this problem. For every component $B_i = (Q_i, X_i, P_i, T_i)$, we define the flow dependency graph $\mathcal{G}_i = (N_i, \hookrightarrow_i)$ where the set of vertices $N_i = X_i \cup P_i$ contains the ports and variables of B_i and edges $\hookrightarrow_i\ \subseteq\ N_i \times N_i$ correspond to flow dependencies required by Proposition 1 and are defined below, for every $x, y \in X_i$, $p, r \in P_i$:

$y \hookrightarrow_i x$	*iff*	$\exists t \in T_i.\ x := e \in asgn(t),\ y \in use(e) \cup use(guard(t))$
$p \hookrightarrow_i x$	*iff*	$\exists t \in T_i.\ x := e \in asgn(t),\ p = port(t)\ \bigvee\ p \in P_i^{in}, x = var(p)$
$y \hookrightarrow_i p$	*iff*	$\exists t \in T_i.\ y \in use(guard(t)), p = port(t)\ \bigvee\ p \in P_i^{out}, y = var(p)$
$p \hookrightarrow_i r$	*iff*	$\exists t, t' \in T_i.\ p = port(t), r = port(t'), (dst(t) = src(t') \vee src(t) = src(t'))$

Using flow dependency graphs, the configuration synthesis problem is formally rephrased as follows:

– Given system $\Gamma(B_1, ..., B_n)$, partial annotation $\sigma_0 : X \to \mathbb{S} \cup \{\bot\}$
– Find complete annotation $\zeta : X \cup P \to \mathbb{S}$ such that
 (C1) (initial annotation) $\forall x \in X.\ \sigma_0(x) \neq \bot\ \Longrightarrow\ \zeta(x) = \sigma_0(x)$
 (C2) (flow preservation) $\forall i = 1, n.\ \forall x, y \in P_i \cup X_i.\ x \hookrightarrow_i y\ \Longrightarrow\ \zeta(x) \leq \zeta(y)$

(C3) (connector consistency) $\forall \gamma = (p^{out} p^{in}) \in \Gamma.\ \zeta(p^{out}) = \zeta(p^{in})$

If a complete annotation ζ exists and satisfies the conditions (C1-C3) above, then the system $\Gamma(B_1, ..., B_n)$ is provable secure for $\sigma = \zeta_{|X}$ and $\varsigma = \zeta_{|P}$, which are respectively the projections of ζ to variables X and ports P. That is, all conditions required by Proposition 1 on annotation of ports and variables within components are captured by dependency graphs $(\mathcal{G}_i)_{i=1,n}$ and satisfied according to (C2). Connectors are consistently annotated according to (C3). Moreover, the initial annotation is preserved by (C1).

An iterative algorithm to compute the complete annotation ζ is depicted as Algorithm 1 below. If the algorithm terminates without detecting inconsistencies, then ζ is the less restrictive annotation satisfying conditions (C1-C3). If an inconsistency is detected, then no solution exists. In this case, the initial annotation is inconsistent with respect to the information flow within the system.

Algorithm 1: Annotation Synthesis

```
1  ζ(n) ← { σ0(n) if n ∈ X, σ0(n) ≠ ⊥ ; ⊓𝕊 otherwise }        ▷ initialization
2  BList ← {B_i}_{i=1,n}                                      ▷ inter-component outer loop
3  while BList ≠ ∅ do
4      choose-and-remove(BList, B_i)
5      nList ← X_i ∪ P_i                                      ▷ intra component inner loop for G_i
6      while nList ≠ ∅ do
7          choose-and-remove(nList, n_i)
8          s_i ← ⊔{ζ(n) | n ↪_i n_i}                          ▷ recompute security level of n_i
9          if ζ(n_i)≤s_i and s_i ≠ ζ(n_i) then
10             if n_i ∈ X_i and σ0(n_i) ≠ ⊥ and σ0(n_i)≤s_i then
11                 stop                                       ▷ inconsistency detected
12             ζ(n_i) ← s_i                                   ▷ update and propagate change within G_i
13             nList ← nList ∪ {n | n_i ↪_i n}
14     foreach p_i ∈ P_i^out ∪ P_i^in do
15         find p_j ∈ P_j^out ∪ P_j^in with (p_i p_j) ∈ Γ or (p_j p_i) ∈ Γ
16         if ζ(p_i) ≠ ζ(p_j) then
17             ζ(p_j) ← ζ(p_i)                                ▷ update and propagate change across connectors
18             BList ← BList ∪ {B_j}
```

Initially, all system variables are either annotated by security levels given from system designer σ_0 if it exist or a default level that correspond to the lowest security level ($\sqcap\mathbb{S}$) in the lattice (line 1). The algorithm visits iteratively all components (lines 2-18). For every component B_i, it propagates forward the current annotation ζ within the flow graph $\mathcal{G}_i$ (lines 3-13). The security level $\zeta(n_i)$ of every node n_i is eventually increased to become more restrictive than

the levels of its predecessors (lines 8-13). An inconsistency is reported if the security level increases for an initially annotated variable (lines 10-11). Any change triggers recomputation of successors nodes of n_i (lines 12-13). Finally, once the annotation within $\mathcal{G}_i$ is computed, any change on security levels on input/output ports is propagated to connected ports (lines 14-18). After this propagation step, any pair of connected ports has again the same security level. As for variables, notice that annotations for connected ports can only increase: any increase due to propagation within a component is immediately propagated to the connected port. The involved components need to be revisited again (line 18). Notice that both while loops are guaranteed to terminate as the number of annotation changes is bounded for every node. That is, the security level can only be increased finitely many times in a bounded lattice $(\mathbb{S}, \leq)$.

Proposition 2. *Algorithm 1 solves the configuration synthesis problem.*

Proof. Initially, the annotation ζ is defined to satisfy initial annotation condition (C1). It equally satisfies connector consistency (C3) but not necessarily flow preservation (C2). The algorithm propagates this annotation along the flow graphs, without changing the initially annotated variables. Intra-component propagation makes flow preservation (C2) hold for the component but may actually destroy (C3). On the contrary, the propagation across connectors re-establish (C3) but may destroy (C2) for connected components. At termination, no annotation changes are possible/needed, hence, the final annotation ζ satisfies both flow preservation condition (C2) but and connector consistency (C3).

As an example, we apply Algorithm 1 to the Producer-Buffer-Consumer presented in Figure 2 with initial annotation $\{x \mapsto M, z \mapsto H\}$, for security levels L(ow), M(edium) and H(igh), such that $L \leq M \leq H$. The three flow dependency graphs and their dependencies through connectors are depicted in Figure 3. For this initial labelling, the algorithm succeeds to generate a complete annotation for variables $\{x \mapsto M, w \mapsto L, y \mapsto M, z \mapsto H, t \mapsto L, u \mapsto H\}$ and all ports are mapped to M. If however we add to the initial configuration a label to the guard variable w, $\{w \mapsto H\}$, Algorithm 1 detects an inconsistency at the *Producer* component and an illicit flow from the w variable to y variable through port *produce* is reported to the user.

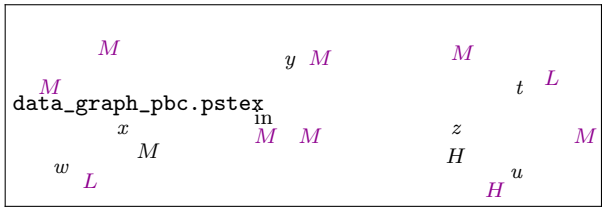

Fig. 3. Dependency graphs of Producer-Buffer-Consumer from Figure 2

4 Application to Web Services

In this section, we apply the synthesis approach to generate secure configurations of WS applications. We consider a composition of *BPEL* and elementary services annotated using the decentralized label security model (DLM) [9]. We briefly present how such WS compositions are represented in the component-based model and we show how the DLM annotations are used in the *BPEL* context.

4.1 The BPEL Composition

BPEL provides structuring mechanisms to compose several WS into a new one. We particularly focus on *BPEL4WS* [10] processes which compose services from activities, that are either (1) *basic* such as receive, reply, invoke, assign, throw, exist, or (2) *structured* such as sequence, if, while, repeatuntil, pick, flow.

The representation of *BPEL* processes in our component model is structural, that is, the structure of the source *BPEL* model is preserved in the target model. More precisely, a process is represented as an atomic component where the behavior encompasses all its basic and structured activities. All process variables are added to the atomic component. Basic activities such as $\langle receive.../\rangle$, $\langle reply.../\rangle$, $\langle invoke.../\rangle$ are translated into specific transitions triggered by respectively *Receive_** and *Reply_** ports. Their corresponding variables are implicitly attached to the above ports. The $\langle assign.../\rangle$ activity is translated as an internal transition that executes the corresponding assignment.

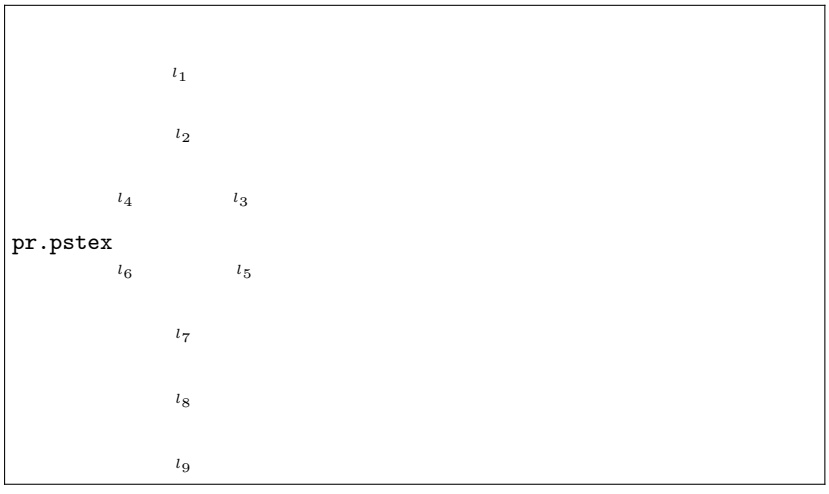

Fig. 4. Atomic component representation of a *BPEL* process

Structured activities define the overall control flow of transitions in the atomic component, in the usual way. In particular, transition guards are extracted from ⟨*if..., while..., repeatuntil.../*⟩ and ⟨*pick.../*⟩ activities. As a restriction, the parallel execution of activities within a ⟨*flow.../*⟩ is not supported. In this case, their execution is made sequential in an arbitrary order.

Finally, we define the connectors and the composition of the atomic components by using the *PartnerLinks* defined for *BPEL* processes. Every ⟨*invoke.../*⟩, ⟨*receive.../*⟩ and ⟨*reply.../*⟩, ⟨*receive.../*⟩ interaction defined over partner links is translated to a connector relating the corresponding components and their respective ports. Let us notice that processes may interact through partner links with external WS, that is, developed in other languages than *BPEL* (such as Java, C, etc). In this case, these WS are represented as atomic components with an implicit behaviour, for arbitrarily sending and receiving data through their connected ports.

Similar translations have been already defined in the literature [11]. As the above translation is structural the resulting model remains comprehensive for the WS designer. The representation relies basically on adapting reusable and composable model components that directly maps processes with limited numbers of execution steps. Despite that, some features in *BPEL* language are not considered such as fault/event handling and scopes. Security errors that can be generated by these aspects are not in the scope of this paper.

4.2 Decentralized Label Model

The decentralized label model (DLM)[9] provides a universal labeling scheme where security labels (or levels) are expressed using set of policies. A confidentiality label L contains (1) an owner set, denoted *O(L)*, that are principals representing the originating sources of the information, and (2) contains for each owner $o \in O(L)$ a set of readers, denoted *R(L,o)*, representing principals to whom the owner o is willing to release the information. The association of an owner o and a set of readers $R(o)$ defines a policy. Principals are ordered using an *acts_for* partial order relation (denoted $\prec$) which is a delegation mechanism that enables a principal to pass his rights to another principal (e.g., $o_1 \prec o_2$ states that o_2 can act for o_1). A security domain is defined over the set of confidentiality labels by using a *flows to* relation defined as follows:

$$L_1 \leq L_2 \equiv \forall o_1 \in O(L_1).\ \forall o_2 \in O(L_2).\ o_1 \prec o_2 \wedge \\ \forall r_1 \in R(L_1, o_1).\ \exists r_2 \in R(L_2, o_2).\ r_1 \prec r_2$$

The intuition behind the *flows to* relation $\leq$ above is that *(1)* the information can only flow from one owner o_1 to either the same or a more powerful owner o_2 where o_2 can act for o_1 and *(2)* the readers allowed by $R(L_2, o)$ must be a subset of the readers allowed by $R(L_1, o)$ where we consider that the readers allowed by a policy include not only the principals explicitly mentioned but also the principals able to act for them.

In our setting for *BPEL* WS composition, the principals used to define the *acts_for* relation and the security domain are obtained from *BPEL* partner links

that correspond to WS URI. That is, principals can be either BPEL processes or atomic WS in some primitive language.

The designer expresses his security policy by tagging *BPEL* variables using DLM labels. The security domain and these annotations are then transposed as such on the component-based model. For example, a confidentiality label L: {*Prosumer*:*SMG*} assigned to the variable *outplan* in the *Prosumer* process presented in Figure 4 (right side) is used directly, as it is, for the *outplan* variable in *Prosumer* atomic component obtained by translation (left side).

4.3 Implementation

The configuration synthesis algorithm described in Section 3 is implemented and available for download at http://www-verimag.imag.fr/∼bensaid/secureBIP/. The user provides the WS composition in *BPEL* and a configuration file (.xml) that contains an *acts_for* relation defining authorities for different processes and the DLM annotations for some process variables. An example of a configuration file is provided in the Appendix. In a first step, the *BPEL* composition is structurally transformed into a component-based model representation in *BIP*[12]. The transformation extends an already existing translation of *BPEL* to BIP developed in [11] to study functional aspects. In a second step, the synthesis tool takes as input the system model (.bip) and the configuration file (.xml), builds the dependency graphs of components and runs the synthesis algorithm to produce the complete configuration.

5 Use-case: Smart Grid Application

To illustrate the use of our framework we consider a simplified model of a smart grid system [13] managed through Internet network using WS. Smart grid systems usually interconect a number of cooperating *prosumers*, (that is, *pro*-ducers and con-*sumers*) of electricity on the same shared infrastructure. In principle, every prosumer is able to produce, store and consume energy within the grid. However, its use of the grid has to be negotiated in advance (e.g., on a daily basis) in order to adapt to external conditions (e.g., weather conditions, day-to-day demands,...) as well as to maintain the behaviour of the grid in some optimal parameters (e.g., no peak consumption). Smart grids are subject to requirements related to safety and security e.g., the power consumption / production of a prosumer must remain secret as it actually may reveal sensitive information.

In our WS model of the smart grid, the system consists of a finite number of prosumer processes, Pr_i, communicating with a *smart grid* process, *SMG*. Initially, each Pr_i sends its consumption and production plan, (P_i, C_i, B_i), for the next day to the grid. Production P_i, consumption C_i and (storage) battery B_i are expressed using energy units (integer) where $0 \preceq P_i \preceq 2$, $-3 \preceq C_i \preceq 0$ and $-1 \preceq B_i \preceq 1$. The *SMG* validates the plans received by checking that the overall energy flow through the grid implied by these plans does not exceed the power line capacity. This check measures the consumption exceed acknowledgment,

ack, compared to a bound, that is, *ack*=0 if the $-1 \preceq \sum_{i=1}^{n}(P_i + C_i + B_i) \preceq 4$, otherwise, it returns the difference between the sum of the plans and the consumption bounds. The *SMG* sends back to each Pr_i an ack_i to negotiate updating its own plan, where $ack = \sum_{i=1}^{n} ack_i$. The negotiation terminates when *ack=0* meaning that the energy flow on the grid does not exceed the line capacity. Figure 5 shows the system overview with two prosumers that exchange queries with the smart grid.

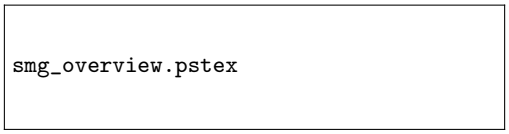

Fig. 5. Smart grid application overview

The information flow security requirements that we emphasize here consist first, on ensuring the confidentiality of energy consumption plan for each Pr_i, (which can reveal sensitive competitive information such as its production capacity) and second, ensuring that no prosumer is able to deduce the consumption plan of any other prosumer by observing the received ack information. For instance, consider two prosumers such that one of them, Pr_1, sends an extreme consumption plan $(0, -3, -1)$ to the *SMG* while the second, Pr_2, sends $(0, -3, 0)$ as a consumption plan. The *SMG* first calculates the acknowledgment message that is *ack*=3 then sends ack_1=1, ack_2=2 messages to respectively Pr_1 and Pr_2. Assume now that Pr_2 sends back a new consumption plan $(1, -2, 1)$ and gets back ack_2=0. By only observing other *ack*1 message sent to Pr_1, the Pr_2 can deduce that the consumption plan of Pr_1 is equal to (0,-3,-1). The translation of Pr_1 process is given in Figure 4 while the translation of the *SMG* process is given in the appendix.

For applying our approach to check system security, the designer introduces initially his partial security policy by tagging intuitively some variables that he considers sensitive in system model with security annotations. He also provides an *acts_for* diagram for all model components where he gives authorities to some of them to act for others. In this system the *SMG* component can only acts for both Pr_1 and Pr_2. To ensure confidentiality of prosumers plan, the system administrator annotates *out_plan*1 with $L_1 = \{Pr_1 : SMG\}$ label and *out_plan*2 with $L_2 = \{Pr_2 : SMG\}$ label. Obviously, $L_1 \not\leq L_2$ and $L_2 \not\leq L_1$ are indicating that both prosumers represent separate security domains that can only communicate with the *SMG* component. Then, the tool automatically generates the dependency graph of the transformed smart grid system. Presented in Figure 6, the dependency graph is build over ports (rectangles) and data variables (circles) locally at each atomic component(big circles), where arrows intra-circles represent dependencies between ports and data in the same atomic

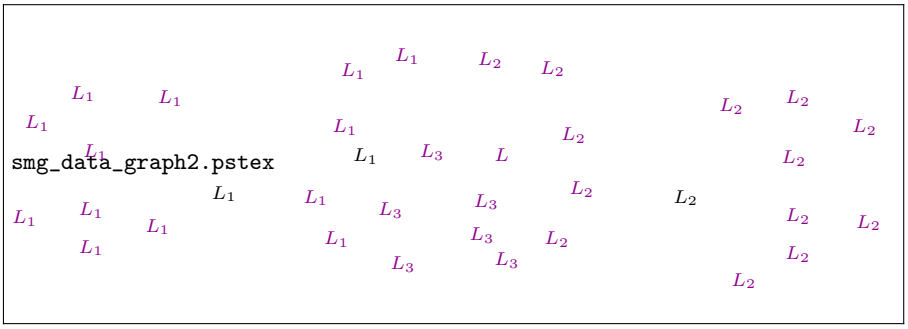

Fig. 6. Generated dependency graphs (fragments).

component while arrow inter-circles represent inter-components dependencies. The application of Algorithm 1 to the system dependency graph detects an illegal information flow in the system and generates an error in the *out_plan* node for both prosumers. Indeed, the label propagation in the system creates at *ack* node of the *SMG* component a new label $L_3 = L_1 \sqcup L_2$. Obviously, label $L_3 = \{Pr_1 : SMG; Pr_2 : SMG\}$ is more restrictive than both labels L_1 and L_2. Since the *ack* node depends on *out_ack*1 in Pr_1 and *out_ack*2 in Pr_2, then it is labelled with L_3 in both prosumers which causes security level inconsistency at *out_plan* nodes. Algorithm 1 generates an inconsistent security level error between both *out_plan* and *ack* nodes. Here, the system designer has to redefine the initial configuration, for instance, by given more privilege to prosumers to act for *SMG* component and enforce variable *ack* to higher security level $L_3 = \{SMG : SMG\}$. In this case, and with the authority that each prosumer gain, flow can go from L_3 to L_1 and L_2.

As an evaluation of the compositional approach performance, table 1 presents some experiments over configuration time t for different variation of the number of prosumer components, n, in the smart grid system for a given number of variables X, ports P and with initial labels number, σ_0. Here we can notice that our configuration synthesis does not introduce an overhead even by increasing the number of system components.

n	**P**	**X**	σ_0	**t**
4	26	24	3	1.82
13	98	87	12	1.94
25	194	171	24	2.01
101	802	703	100	2.82

Table 1. Model size and configuration time (in s) for smart grid application with one initial security label by each prosumer.

6 Related Work

There are many commercial tools like IBMs XML Security Gateway XS40, application servers [14] and Web Service Enhancements for Microsoft .NET

(WSE) [15] that provide GUI to help users configure and verify WS security but the user has to learn about standard WS-Security syntax and options.

In [16], authors propose a high-level GUI for configuring WS security with a business-policy-oriented view. It models the messaging with customers and business partners, lists various threats, and presents best-practice security patterns against these threats. A user can select among proposed generated basic patterns according to the business policies, and then apply them to the messaging model. The result of the pattern application is, afterwards, described in the Web Services Security Policy Language (WS-Security Policy). None of these tools handle the non-interference property like we do. Regarding formal models for non-interference in WS, in [17], authors present WS data flows as extensions of the Petri-net model and in [18], non-interference has been formalized for Petri-nets. Nevertheless, these solutions have some drawbacks which are mainly that data and resource description is manual and can be therefore error prone. Later on, the same authors propose the IFAudit tool that represents data flow as *propagation graphs* generated from workflow's log data. The propagation graphs are analyzed against the security policies.

Distinct security models are proposed in [19] where authors propose a classification of security-aware WS. They list a set of works classified in information flow category. Nevertheless, these works are restricted to verification rather than security configuration synthesis. Here we propose a practical automated verification method for transformed model of composed *BPEL* WS, based on formally proved security conditions. In [20], the authors deal with chained services and security is checked with a notion of back check procedure and pass-on certificate. It is not clear how to apply this solution to WS orchestration workflows and how to handle implicit interference. A recent work extends *BPEL*-orchestra engine [21] to handle IFC security. This work is inspired from SEWSEC framework [22] by adopting the distributed security label to annotate information. Nevertheless, instead of using abstractions like PDG, this annotation is set inside the *BPEL* code. Code annotation requires security skills, does not separate functional and security concerns and induces development overhead.

Finally, our work is related to information flow security in component-based systems. In contrast to [23] where authors verify security in a component-based model by annotating the system ADL (Architecture Description Language) and tracking information flow at intra- and inter-components separately, this work provides a sound model with formal proofs guaranteeing system non-interference. Besides, and compared to our previous work [6] where we adopted a more general component-based model to build secure distributed systems, here we propose a simpler message-based send-receive model suitable to model applications with web-style primitives and communications like BPEL-based composed WS and we propose not only a security verification but also a practical solution for security configuration synthesis.

7 Conclusion and Future Work

In this paper, we propose a component-based approach to assist system designers to analyze and enforce information flow security in WS compositions. We implemented a compositional synthesis algorithm that propagates labels and generates secure system configurations starting from partial configurations.

As future work, we plan to extend this work in several directions. First, we are seeking for less restrictive syntactic conditions for establishing non-interference. In particular, we believe that a finer control flow analysis using for instance dominance analysis [24] can provide finer dependencies amongst variables and ports. Second, we are working on relaxing the non-interference property and introducing declassification mechanisms to our model. Declassification has been studied for sequential interactive programs with inputs and outputs [25], nevertheless, its extension to distributed concurrent component-based models such as Web Services is less understood.

References

1. Walsh, A.: UDDI, SOAP, and WSDL: The Web Services Specification Reference Book. Prentice Hall (2002)
2. Juric, M.B.: Business Process Execution Language for Web Services BPEL and BPEL4WS 2nd Edition. Packt Publishing 2006 (2006)
3. Damiani, E., di Vimercati, S.D.C., Paraboschi, S., Samarati, P.: Securing SOAP e-services. International Journal of Information Security **1**(2) (2002) 100–115
4. Della-Libera, G., Gudgin, M., Hallam-Baker, P., Hondo, M., Granqvist, H., Kaler, C., Maruyama, H., McIntosh, M., Nadalin, A., Nagaratnam, N., Philpott, R., Prafullchandra, H., Shewchuk, J., Walter, D., Zolfonoon, R.: Web services security policy language (WS-SECURITYPOLICY). Technical report (2005)
5. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy. (1982) 11–20
6. Ben Said, N., Abdellatif, T., Bensalem, S., Bozga, M.: Model-driven Information Flow Security for Component-Based Systems. In: From Programs to Systems. The Systems perspective in Computing - ETAPS Workshop, FPS 2014. (2014) 1–20
7. Ben Said, N., Abdellatif, T., Bensalem, S., Bozga, M.: Model-driven information flow security for component-based systems. Technical Report TR-2013-7, VERIMAG `http://www-verimag.imag.fr/TR/TR-2013-7.pdf`.
8. Rushby, J.: Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-2, SRI International (1992)
9. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Transactions on Software Engineering and Methodology **9** (2000)
10. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: BPEL4WS, Business Process Execution Language for Web Services Version 1.1. IBM. (2003)
11. Stachtiari, E., Mentis, A., Katsaros, P.: Rigorous analysis of service composability by embedding WS-BPEL into the BIP component framework. In: 2012 IEEE 19th International Conference on Web Services. (2012) 319–326

12. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based design using the BIP framework. IEEE Software, Special Edition – Software Components beyond Programming – from Routines to Services **28**(3) (2011) 41–48
13. Koss, D., Sellmayr, F., Bauereiss, S., Bytschkow, D., Gupta, P., Schaetz, B.: Establishing a Smart Grid Node Architecture and Demonstrator in an Office Environment Using the SOA Approach. In: First International Workshop on Software Engineering Challenges for the Smart Grid, SE4SG. (2012) 8–14
14. Corporation., I.B.M.: Using BPEL processes in WebSphere Business Integration Server Foundation. IBM, International Technical Support Organization, (2004.)
15. Microsoft Developpement network: `http://msdn.microsoft.com/`
16. Tatsubori, M., Imamura, T., Nakamura, Y.: Best-practice patterns and tool support for configuring secure web services messaging. In: IEEE International Conference on Web Services (ICWS'04). (2004) 244–251
17. Busi, N., Gorrieri, R.: A survey on non-interference with petri nets. In: Lectures on Concurrency and Petri Nets, Advances in Petri Nets. (2003) 328–344
18. Busi, N., Gorrieri, R.: Structural non-interference in elementary and trace nets. Mathematical Structures in Computer Science **19**(6) (2009) 1065–1090
19. Movahednejad, H., Ibrahim, S.B., Sharifi, M., Selamat, H.B., Tabatabaei, S.G.H.: Security-aware web service composition approaches: State-of-the-art. In: iiWAS'2011 - The 13th International Conference on Information Integration and Web-based Applications and Services, ACM (2011) 112–121
20. She, W., Yen, I., Thuraisingham, B.M.: Enhancing security modeling for web services using delegation and pass-on. Int. J. Web Service Res. **7**(1) (2010) 1–21
21. Demongeot, T., Totel, E., Traon, Y.L.: Preventing data leakage in service orchestration. In: 7th International Conference on Information Assurance and Security, IAS 2011. (2011) 122–127
22. Zorgati, H., Abdellatif, T.: Sewsec:a secure web service composer using information flow control. In: CRiSIS 2011 - the Sixth International Conference on Risks and Security of Internet and Systems. (2011) 62–69
23. Abdellatif, T., Sfaxi, L., Robbana, R., Lakhnech, Y.: Automating information flow control in component-based distributed systems. In: 14th International ACM Sigsoft Symposium on Component Based Software Engineering, CBSE 2011, ACM (2011) 73–82
24. Reinhartz-Berger, I., Sturm, A., Clark, T., Cohen, S., Bettin, J., eds.: Domain Engineering, Product Lines, Languages, and Conceptual Models. Springer (2013)
25. Askarov, A., Sabelfeld, A.: Tight enforcement of information-release policies for dynamic languages. In: 22nd IEEE Computer Security Foundations Symposium, CSF 2009. (2009) 43–59

Appendix

Figure 7 shows a transformation of the *SMG* process of the smart grid system given as *BPEL* workflow, into an atomic component. The behavior of the atomic component represents the activities given in the *BPEL* process.

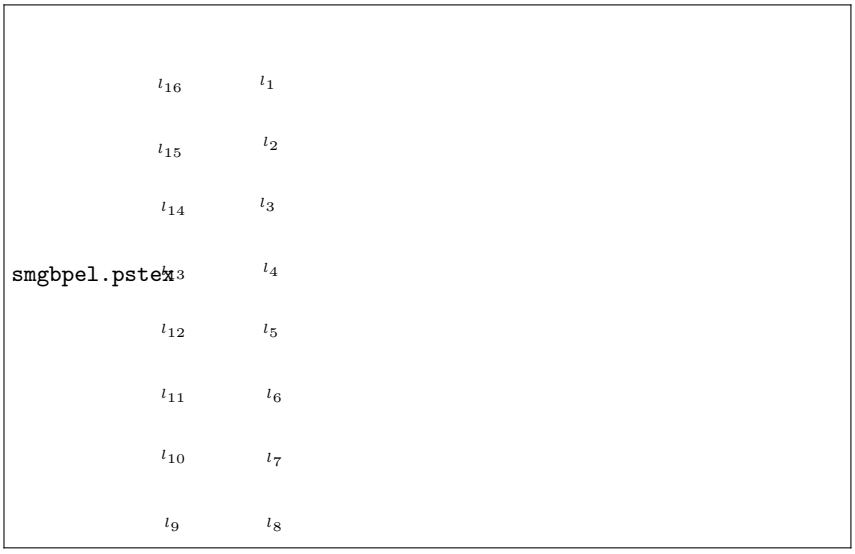

Fig. 7. Translation of the *SMG* component

The designer input configuration file includes an *acts_for* relation as well as some annotated variables. Here we presented an example of a configuration file of the smart grid system. In this xml file we define $\langle authority/\rangle$ to different system components representing the *acts_for* relation. Moreover, we specify by $\langle var\_config/\rangle$ the annotations of variables from different atomic components (processes).

```
<?xml version="1.0"?>
<config>
   <acts_for>
     <authority>SMG: Prosumer1, Prosumer2, Prosumer3</authority>
   </acts_for>
   <var_config>
     <variable var="outplan" process="Prosumer1"
               label="Prosumer1:SMG"></variable>
     <variable var="outplan" process="Prosumer2"
               label="Prosumer2:SMG"></variable>
     <variable var="outplan" process="Prosumer3"
               label="Prosumer3:SMG"></variable>
   </var_config>
</config>
```