Automated synthesis and ranking of secure BPMN orchestrators

V. Ciancia, F. Martinelli, I. Matteucci, M. Petrocchi
Istituto di Informatica e Telematica
National Research Council, 56124 Pisa, Italy
Email: firstname.surname@iit.cnr.it

J. A. Martín, E. Pimentel
E.T.S. Ingeniería Informática,
Universidad de Málaga,
Campus de Teatinos, 29071 Málaga, Spain
Email: jamartin@lcc.uma.es
ernesto@lcc.uma.es

***Abstract*—We describe a formal methodology for the automatic synthesis of a secure orchestrator for a set of BPMN processes. The synthesized orchestrator is able to guarantee that all the processes that are started reach their end, and the resulting orchestrator process is secure, that is, it does not allow discloure of certain secret messages.**

In this work we present an implementation of a forth and back translation from BPMN to crypto-CCS, in such a way to exploit the PaMoChSA tool for synthesizing orchestrators.

Furthermore, we study the problem of ranking orchestrators based on quantitative valuations of a process, and on the temporal evolution of such valuations and their security, as a function of the knowledge of the attacker.

***Keywords*-Synthesis of Functional and Secure Processes, Secure Service Composition, Partial Model Checking, Process Algebras, Business Process Modelling Notation, Quantitative security.**

I. Overview

Mathematical methods in program semantics and security very often need to be validated trough implementation and technology transfer. Traditionally, the task has been hindered by the gap between abstract results and applications. The advent of *software engineering* brought to light the so-called *semi-formal* languages and methods, such as *Unified Modelling Language* (UML) [1] or *Business Process Model and Notation* (BPMN) [2]. These formalisms provide clean syntax to support abstraction in software and system design, and development practice. Semi-formal methods are nowadays part of the standard background of software engineers, and may be used to bridge the mentioned gap, providing a clean path from theoretical results to implementation.

In this paper, we consider the research line of verification and synthesis of secure systems by partial model checking [3]. In particular, we extend the work in [4] by exploiting the PaMoChSA tool for synthesizing secure BPMN orchestrator processes. The workflow we adopt is described in Figure 1. For the first time, we transfer the related know-how from the abstract realm of process calculi, and Crypto-CCS in particular, to a real-world specification language, namely BPMN. Crypto-CCS is a process calculus that features a tractable form of logical deduction, permitting one to encode various cryptographic primitives. In particular, the calculus faithfully models asymmetric cryptography.

In the semi-formal approach, various indended semantics can be assigned to a formalism, depending on the application context. Following this tradition, we provide a security-aware execution semantics to BPMN, incorporating secure communication facilities, by the means of Crypto-CCS. By doing so, we assign a formal semantics to BPMN operators. This is done in Section III. For the purpose, we define BPMN processeses that exchange cryptographic messages. More precisely, we use existing BPMN facilities to include asymmetric cryptography in the modelling language. In this way, existing tools may be used to design cryptography-aware systems. We provide a proof-of-concept implementation, in the form of two XQuery transformations. The first one translates a BPMN process into Crypto-CCS, whose syntax is represented using a custom XML format. The second transformation turns an XML representation of a sequential Crypto-CCS process back into a BPMN process. The translation is made to interoperate with the previously developed tool PaMoChSA, performing synthesis of (sequential) Crypto-CCS orchestrators [4]. The result is a tool that accepts a BPMN collaboration diagram in input, containing a black-box process representing the orchestrator. The black-box in the original collaboration diagram is filled with a process that orchestrates the BPMN processes, driving all components to successful termination. The orchestrator is secure, in the sense that it uses asymmetric cryptography to forbid an attacker to learn an user-specified secret message.

Furthermore, we present a strategy based on C-semirings [5] to evaluate different synthesized orchestrators in such a way to provide a ranking, that drives the user to select the orchestrator that better fits with the requirements. In particular, the ordering relation we propose here depends upon the knowledge of an attacker, so that a system is considered better than another one, if it obtains better valuations for all the possible initial knowledge bases of the attacker.

Throughout the paper, we use a running example, presented in Section II. Then we proceed to explain the

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant no 256980 (NESSoS) and under grant no 257930 (Aniketos).

978-0-7695-5008-4/13 $26.00 © 2013 IEEE
DOI 10.1109/ARES.2013.60

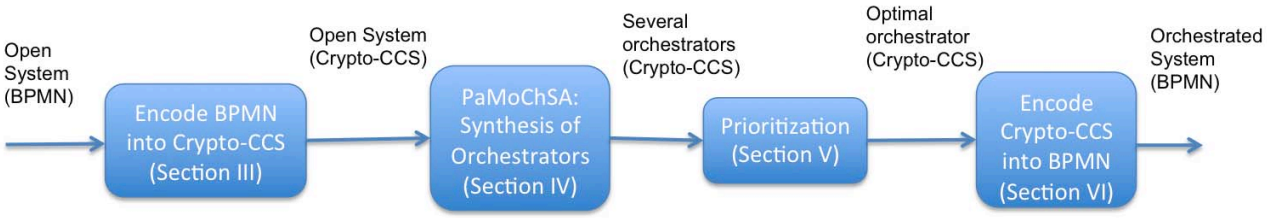

Figure 1. The workflow of the proposed approach

translation from BPMN to Crypto-CCS in Section III. The design of the tool is oriented to its integration into a service repository, or *market place*, where some services may be selected, and a service composition plan may be submitted for execution to the system. Our approach adds to such a framework the ability to automatically design composition plans, in the form of BPMN processes, that orchestrate existing (services and) composition plans. We detail this view in Section IV. Furthermore, in Section V we present our approach based on C-semirings to endow a list of secure orchestrators with an order of preference. This allows the user to chose the optimal one, according to some desired measure, even in the case of multi-criteria optimization. The selected orchestrator is a sequential process that is translated back into a BPMN process and inserted into the original collaboration diagram. This is detailed in Section VI. Finally, Section VII relates our work with the existing literature.

II. Business Process Modelling Notation

The BPMN language [2] provides businesses with the capability of understanding their internal business procedures in a graphical notation, and give organizations the ability to communicate these procedures in a standard manner. The graphical aspects of the notation are organized into five specific kinds of elements: flow objects, data objects, connecting objects, swimlanes, and artifacts[1].

Let us now introduce the running example that we use hereafter in order to show the result presented in the paper.

Example 2.1: Consider an *user* willing to purchase both an airplane ticket and an insurance contract covering the trip. As expected in the service-oriented approach, we suppose that this is achieved by the orchestration of a booking service and an insurance service.

The *booking service* is represented in Figure 2 by a process accepting some cryptographic material from the user, such as an encryption key, and the proof of a successful payment transaction signed with the received key. After such a request, the booking service issues a ticket, which is then sent encrypted to the insurance service for further processing.

[1]In this paper, we focus on a subset of BPMN constructs. In particular we are interesting in the behavioral description of processes, so we consider the BPMN costructs that describe processes.

The *insurance service* receives the encrypted ticket, and returns an insured version of it. The procedure ends correctly once the insured version of the ticket is generated. From a security point of view, it is important to protect the ticket from malicious users willing to discover its content (*e.g.,* pairing the user name with the destination could represent a privacy violation, according to some requirements of the owner; or the ticket could be anonymous, and the attacker could use it in place of the legitimate owner).

In Figure 2 we show the BPMN representation of the open system we want to orchestrate in order to allow the services to communicate to serve the request.

There are some communication issues in this apparently simple procedure. For example, the user might be unable to manage both operations by himself, for some reason that is immaterial here. Furthermore, the two services might not be able to communicate directly, due to firewalls, mismatching policies, or incompatibilities in their interfaces. In this case, one needs to synthesize a third service, called hereafter *orchestrator*, whose aim is to make the system *functional*, *i.e.,* fully satisfying its goal (request and successfully obtain both a ticket and the correspondent insurance). The correct execution of the whole procedure is guaranteed by the issurance and proper reception of the insured ticket.

Each entity of Example 2.1 is represented through a BPMN a *Flow Object*. Flow Objects are the main graphical elements that define the behaviour of a Business Process. There are three types of flow objects:

- *Events*: an Event is something that "happens during the course of a Process. These Events affect the flow of the model and usually have a cause (trigger) or an impact (result). There are three types of Events, based on when they affect the flow: *Start* (the green node in Figure 2), *Intermediate*, and *End* (the red node in Figure 2).
- *Activities*: an activity is a generic term for work that company performs in a Process. An activity can be atomic or non-atomic (compound). *Tasks* are a type of activities. Referring to Figure 2, tasks are represented by the squared boxes between the start and end events. As an example a task of the Insurance service is `Insured_ticket : Insured`.
- *Gateways*: A gateway is used to control the divergence and convergence of *Sequence Flows* in a Process. A

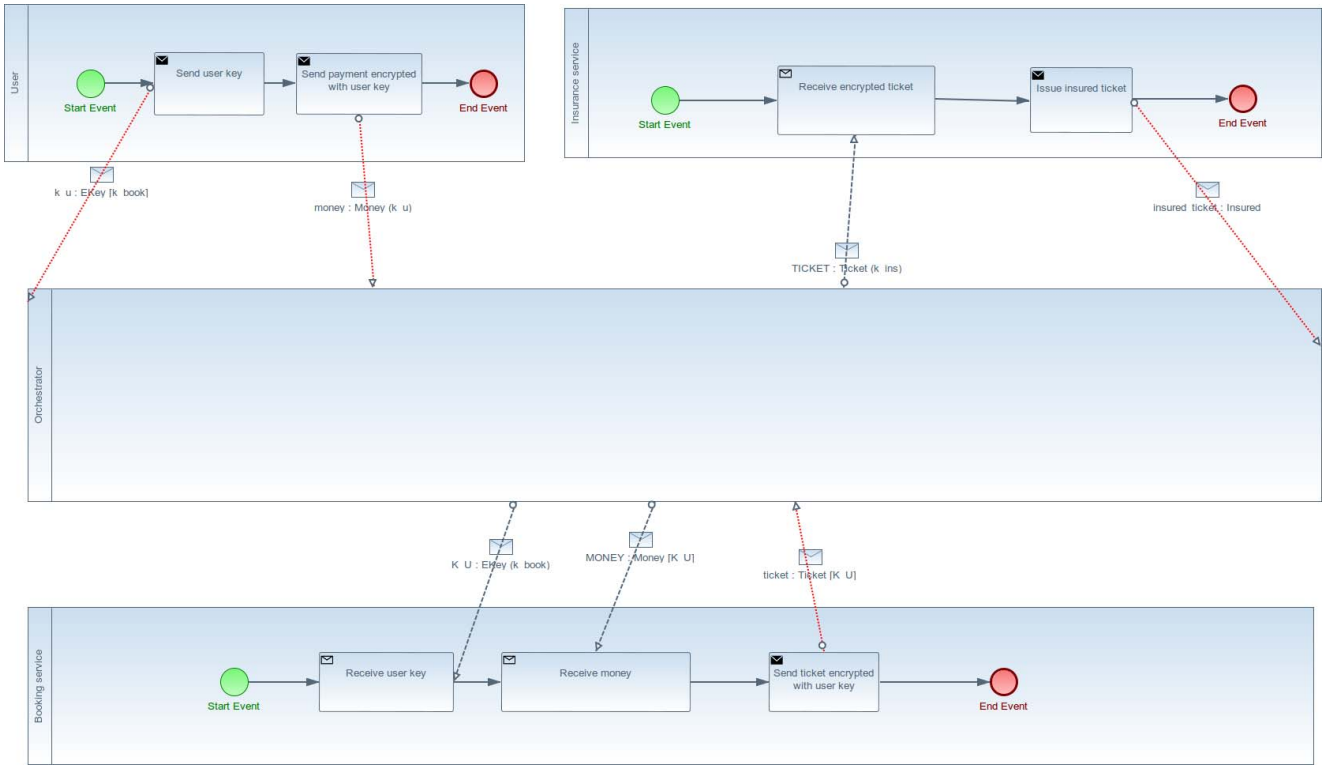

Figure 2. The BPMN specification of a collaboration diagram in which the orchestrator is represented as a black box.

sequence flow is used to show the order that activities will be performed in a process. A gateway will determine branching, forking, merging, and joining of paths. In Figure 3 we show in yellow an example of a parallel gateway and of inclusive gateway.

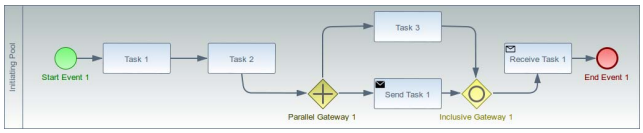

Figure 3. An example of BPMN Pool.

Each entity is represented by a *pool*, see Figure 3. A pool is the graphical representation of a participant in a *collaboration digram*. It also acts as a graphical container for partitioning a set of activities from other pools. A pool may have internal details, in the form of the Process, it is denoted as *white box*, as the processes representing the User, the Booking service and the Insurance service in Figure 2. Or a pool mat have no internal details, i.e., it can be a *black box*, as the pool representing the orchestrator in Figure 2.

A *collaboration diagram* depicts the interactions between two or more business entities. A Collaboration usually contains two or more pools. The Message exchange between the Participants is shown by a *Message Flow* that connects two pools (or the objects within the Pools). A message flow is used to show the flow of Messages between two Participants that are prepared to send and receive them. In BPMN, two separate pools in a collaboration diagram will represent the two participants (e.g., PartnerEntities and/or PartnerRoles). Messages associated with the message flows can also be shown. It is worth noticing the collaboration digram represented in Figure 2 contains crypto-enhanced message names. Cryptography is not a feature of BPMN but we have extended BPMN to support it, as we will show in the next section.

III. From BPMN to Crypto-CCS

BPMN is a semi-formal specification language, whose intended semantics may vary depending upon the entity adopting the language. In this work, we specify a security-aware formal semantics for BPMN collaboration diagrams, using Crypto-CCS. The purpose of the semantics is to synthesize an orchestrator for a given collaboration, using the "synthesis by partial model checking" algorithm available in the most recent version of the tool *PaMoChSA* [4], available at [6]. The input language of PaMoChSA is an XML representation of Crypto-CCS; BPMN is also based on XML. Thus, the implementation of the translation is written in the XML manipulation language Xquery [7]. The BPMN synthesis process takes as input:

- A BPMN collaboration diagram containing exactly one black-box process.
- The *secret* that an attacker (in the Dolev-Yao model) must not be able to derive.
- The *goal*, that is, a success message that the orchestrator must learn, or be able to derive, in order for the orchestration to be meaningful.
- The initial knowledge bases of the attacker and of the orchestrator.

Under such assumptions, the synthesis algorithm produces a new BPMN collaboration diagram where the previous black box is filled with a newly generated process, called the *orchestrator*. The obtained collaboration uses cryptography, as detailed in Section III-B below, in order to guarantee that all the orchestrated services reach termination, the orchestrator learns the goal message, and the attacker does not learn the secret during the process.

A. Crypto-CCS in a nutshell

Here we briefly recall the Crypto-CCS language in order to introduce the reader to the subsequent sections. A detailed description of the language can be found in [3].

Crypto-CCS is a variant of CCS [8], endowed with cryptographic primitives. A model defined in Crypto-CCS consists of a set of sequential agents able to communicate by exchanging messages (*e.g.,* data manipulated by the agents).

$$A \ := \mathbf{0} \mid c!m.A \mid c?x.A \mid [m_1 \cdots m_n \vdash_r x]A; A_1$$

where $m_1, \ldots, m_n, m$ are closed messages or variables, x is a variable and c is an element of the set *Ch* of channels. Informally, the Crypto-CCS semantics used in the sequel is: $\mathbf{0}$ denotes a process that does nothing; $c!m.A$ denotes a message m sent over channel c and then behave as A; $c?x.A$ denotes a message m received over channel c which replaces the variable x and then behave as A; $[m_1 \cdots m_n \vdash_r x]A; A_1$ denotes an inference test that a process may use to check whether message m is derivable from premises $m_1, \ldots, m_n$; the continuations in the positive and negative case are A (where m replaces x), or A_1, respectively. Deduction is the message-manipulating construct of the language, responsible for its expressive power. In particular, it allows to model asymmetric encryption. Let y be a key belonging to an asymmetric pair of keys. We denote by y^{-1} the correspondent complementary key. If y is used for encryption, then y^{-1} is used for decryption, and vice versa. Given a set of messages ϕ, then message $m \in \mathcal{D}(\phi)$ if and only if m can be deduced from the rules modelling public key cryptography.

The control part of the language consists of *compound systems*:

$$S \ := S_1 \parallel S_2 \mid S \backslash L \mid A_\phi$$

Informally, $S_1 \parallel S_2$ denotes the parallel composition of S_1 and S_2, *i.e.* $S_1 \parallel S_2$ performs an action if either S_1 or S_2 does. A synchronization (or *internal*) action, denoted by τ, is observed whenever S_1 and S_2 can perform two complementary send and receive actions over the same channel; $S \backslash L$ prevents actions whose channels belong to the set L, except for synchronization. A_ϕ is a single sequential agent whose knowledge is described by ϕ.

B. The translation from BPMN to Crypto-CCS

In this section we explain the security-aware semantics of BPMN that we propose. We provide semantics only to a basic subset of BPMN, containing the most important sequence flow and communication primitives, and the most widely used types of gateways. The other BPMN gateways can be translated in a similar way, and are not needed to illustrate the synthesis algorithm. Indeed, we provide translation of all the constructs that can be the outcome of the reverse mapping from Crypto-CCS to BPMN.

The XQuery language features recursion and pattern matching on XML trees. This is sufficient to provide an inductive translation, done on the structure of a BPMN process in one direction, and of a Crypto-CCS term in the other direction. The mapping from BPMN to Crypto-CCS is driven by the two dimensions of a BPMN collaboration diagram: flow control and data flow.

Let $\mathcal{B}$ and $\mathcal{C}$ be the set of BPMN and Crypto-CCS processes, respectively. The mapping is a function $\mathcal{T} : \mathcal{B}^+ \rightarrow \mathcal{C}$, where $\mathcal{B}^+$ denotes a non-empty sequence of BPMN processes. Given BPMN processes $P_1, \ldots P_n$ corresponding to the mentioned white-box pools, the resulting Crypto-CCS process is $\mathcal{T}(P_1, \ldots, P_n) = (T(P_1) \parallel \ldots \parallel \mathcal{T}(P_n)) \backslash_{c_1,\ldots,c_n}$ where $c_1, \ldots, c_n$ are fresh channels that are needed to model internal communication of sub-processes, without letting the orchestrator interfere.

One fundamental issue to deal with is how to encode cryptographic primitives in BPMN collaborations. As BPMN does not deal with cryptography explicitly, we encode it in the name of messages. A cryptographically-enhanced BPMN *outgoing* message takes the form $msg : T[key]$ or $msg : T(key)$, where the key may also be omitted. Here, msg is a (lower case, per PaMoChSA syntax) message name, T is its type, and key is a private or public key used for encryption. If the key is omitted, the message is unencrypted. The type T (upper case) may not be omitted, as PaMoChSA is a typed language. Special types are $DKey$ and $EKey$, denoting the public or private part of an encryption key, which may be transmitted. Similarly, an incoming message takes the form $Var : T[key]$ or $Var : T(key)$ where Var (upper case) is a variable. The semantics is that the process continues only if the received message decrypts correctly using the given key. In that case, the decrypted message is bound to Var. When an orchestrator is synthesized, each incoming message $Var : T(key)$ or $Var : T[key]$ is replaced by $Var : T(key) = msg : T(key)$ ($Var : T[key] = msg : T[key]$, respectively) where msg is provided by the synthesis process.

For each BPMN *process* P, a Crypto-CCS process $\mathcal{T}(P)$ is derived, according to the following encoding.

- The BPMN *start event* of P is translated into Crypto-CCS input actions on a special channel having an unique name $Start$, on a channel whose name is derived from the process identifier.
- BPMN *end events* are translated into the Crypto-CCS empty process $\emptyset$, terminating a sequence of actions.
- A BPMN *task* t in the process are translated directly into a CCS action prefix. In particular, each task that is not directly involved in BPMN inter-process communication is represented as a τ (unobservable) action. The rationale is that such tasks are internal actions of the process, therefore no synchronization between those and the orchestrator ought to take place. They are unobservable from an external point of view.
- A task t that participates in inter-process communication must respect three conditions: it has to be either an instance of a BPMN *send task* or of a *receive task*; it has a *message flow* f connecting it to the black-box; f must have an associated BPMN *message reference* m. In this situation, a Crypto-CCS communication is generated. The BPMN *task id* is the channel name; the *message id* of m is used as the message name, in the case of a send action, or variable name, in the case of a receive action. Notice that this encoding adds data-flow semantics to BPMN, in that a received message m can be used as a variable, and then referenced in subsequent send operations, just like in process calculi. This is made implicit in BPMN when using the same message reference in different message flows.
- *Sequence flows* in the process are used to drive, in the obvious way, a Crypto-CCS sequence flow in the generated process; however, particular care has to be put for the case of BPMN *gateways* (see below).
- A *gateway* g is translated into several parallel processes. More specifically, when a sequence flow s going out of a BPMN task t is connected to a gateway, a Crypto-CCS output action for a dummy value on a fresh channel c_i is added to the process of t, with the empty process $\emptyset$ as continuation. This results in several processes, composed in parallel, each one sending a dummy value on a different channel. Similarly, an incoming connection from a gateway is translated into an input action on a fresh channel d_i. Then, a further process P_g is added to the obtained composition, namely the translation of the gateway itself, which is detailed below. All the freshly generated channels c_i go in the restriction at the top level of $\mathcal{T}(P_1, \ldots, P_n)$.
- A gateway g may be either *parallel* or *exclusive*. When g is exclusive, $P_g = c_1?.P' \parallel \ldots c_n?.P'$, where c_i are the fresh channels generated from the tasks having an outgoing sequence flow to g and $P' = d_1!x.\emptyset + \ldots + d_m.\emptyset$, where d_i are the fresh channels coming from tasks having an incoming sequence flow from g. In this way, the exclusive behaviour is replicated in Crypto-CCS, since output on any channel c_i from a task results in just one input on the channels d_j to succeed. When g is parallel, $P_g = c_1?x.c_2?x.\ldots.c_n?x.P'$, where c_i are the channels from the tasks having outgoing flows to g, and $P' = d_1!x.0 \parallel \ldots \parallel d_m!x.0$. In this case, the parallel behaviour is mimicked, as the process waits for input on all channels c_i and then outputs on all channels d_j. Notice that the order of input values does not matter as the dummy value is ignored and the process has to wait for input on all channels.

IV. Synthesis of a BPMN secure orchestrator

In the marketplace, we can find many services described as BPMN processes. In order to satisfy the requirements of a user, some of these services (a finite number of these) has to be selected and combined in order to satisfy functional and security requirements. At synthesis time, all the BPMN processes are composed with the orchestrator in a new collaboration diagram. Hereafter, we consider the compound system as a BPMN process in which the orchestrator is a black box (see Figure 2). Indeed, the service developer design the composition trough a BPMN process in which the selected services are represented by white box pools. We consider each service as a statuful process.

Referring to the example in Section 2.1, the compound system *S* we deal with is specified in Crypto-CCS as three processes, *U*, *BS*, and *InS*, respectively. The User, the Booking service and the Insurance service act in parallel, thus the full system specification is

$$S \doteq U \parallel BS \parallel InS$$

Note that processes cannot directly communicate with each other. Also, whereas the Booking service encrypts the ticket with k_u (see Figure 2), the Insurance service would expect to decrypt the same ticket with decryption key k_{ins}^{-1}. Hence, these processes have to interact with an orchestrator, *O* in order to communicate. The resulting orchestrated system is $S \parallel O$.

The orchestrator we are looking for must not only be *functional*, but also *secure*, since its functionality should obey the security requirements imposed on the composed system.

In this work we exploit the method presented in [4] for automatically generating the description of the process in the black box, i.e., the orchestrator. The PaMoChSA tool takes as input the description of the compound system in Crypto-CCS. Then it returns the orchestrator specified as a Crypto-CCS process. Using the mapping function from Crypto-CCS to BPMN, Section VI, we automatically synthesize a BPMN description of the orchestrator. In this way, we obtain a complete description of the compound system.

A. The PaMoChSA Synthesis Tool

In [4], the author proposed a tool to automatically synthesize functional and secure orchestrator specified in CryptoCCS, based on partial model checking, logic languages and satisfiability. Let m_F be a message that denotes the end of a service execution, ϕ_O be the knowledge of the orchestrator, and ϕ_X be the knowledge of the attacker.

The synthesized orchestrator process is consider functional and secure because it is able to:

- **Functional:** combine several services in such a way that m_F falls into the orchestrator's knowledge ϕ_O. This implies that all services have successfully terminated their execution.
- **Secure:** guarantee that the composite service is secure by checking that the secret message m does not belong to ϕ_X.

The tool PaMoChSA2012 [6] is an extension of the original PaMoChSA tool, the Partial Model Checker Security Analyser, see [9].

It is able to automatically *synthesize* a functional and secure orchestrator starting from the description of services. The algorithm can be more intuitively explained as path-finding in a state graph. In principle, the behaviour of an orchestrator is a tree. However, since the system and the orchestrator are assumed to be deterministic, such a tree has an equivalent description in terms of all its paths.

The PaMoChSA 2012 specification of the example introduced in Section II is the following:

```
<GOAL> insured_ticket : Insured </GOAL>

<ORCH_KNOWLEDGE> k_ins : EKey ;
k_u : DKey; k_u : EKey </ORCH_KNOWLEDGE>

<FORMULA> ticket : Ticket </FORMULA>

<KNOWLEDGE> k_book : EKey; k_ins : EKey;
k_attack : EKey; k_attack : DKey </KNOWLEDGE>

<SPEC>
Parallel

(* User *)
Send(a, Encrypt(k_u: EKey, k_book : EKey)).
Send(a, Encrypt(money : Money, k_u : DKey).0

And

(* Booking service *)
Recv(c,ENC_K_U : Enc(EKey * EKey)).
If Deduce (K_U = Decrypt(ENC_K_U,k_book : DKey))
 Then
 Recv(c,ENC_MONEY : Enc(Money * DKey)).
 If Deduce (MONEY = Decrypt(ENC_MONEY,K_U))
  Then
     Send(c,Encrypt(ticket : Ticket, K_U)).0
 End Deduce
End Deduce

And

(* Insurance service *)
Recv(d, ENC_TICKET : Enc(Ticket * EKey)).
If Deduce (TICKET =
           Decrypt(ENC_TICKET,k_ins : DKey))
   Then
  Send(d,insured_ticket : Insured).0
End Deduce

End Parallel
</SPEC>
```

In our running example, the *functional* goal of the orchestrator is to deliver an insured ticket, while its *secure* goal is to let the ticket a secret not falling into the intruder's knowledge.

The tags in the input file describe the information necessary to synthesize an orchestrator. The tag *formula* introduces the typed message that should not be learned by the intruder (in our case, the ticket). The tag *goal* contains the final "success" message (in this case, the insured ticket). The terms in the *Orch_Knowledge* tag represent the typed messages that the orchestrator knows initially (in our running example, the initial knowledge of the orchestrator contains the public key of the insurance service, k_{ins}, and both the public and private key of the user). Especially since it knows private key, in this example the orchestrator is considered a trusted party that acts on behalf of the user. Indeed, this is not necessarily the case. The terms in the *knowledge* tag represent the typed messages that the intruder knowns at the beginning of the computation, *i.e.,* some public cryptographic keys (k_{book} and k_{ins}), plus a pair of keys (k_{attack}, k^{-1}_{attack}) owned by the intruder. Finally, the *spec* tag introduces the specification of the known part of the system, *i.e.,* the parallel composition of the user, plus the booking and insurance services.

Figure 4 shows the results of the synthesis procedure. The tool correctly synthesizes two secure orchestrators (basically, they represent the same process, up-to a different order of execution). The orchestrator is a process that communicates with the user on channel a, forwarding messages to the booking service on channel b. Then, it receives an encrypted ticket from the booking service on channel b, decrypts the obtained message, encrypts the ticket with the public key k_{ins}, and sends the result to the insurance service over channel c. Thus, the insurance service is able to decrypt the ticket using key (k^{-1}_{ins}) and to successfully issue the insured ticket, therefore fulfilling the functional goal. Synthesized orchestrators are secure, in the sense that their operations do not let a potential intruder learn the secret message "ticket".

```
*** Orchestrator:
Recv(a, Enc[k_book](k_u) : Enc(EKey*EKey)).
Recv(a, Enc[k_u](money) : Enc(Money*DKey)).
Send(c, Enc[k_book](k_u) : Enc(EKey*EKey)).
Send(c, Enc[k_u](money) : Enc(Money*DKey)).
Recv(c, Enc[k_u](ticket) : Enc(Ticket*EKey)).
Send(b, Enc[k_ins](ticket) : Enc(Ticket*EKey)).
Recv(b, insured_ticket : Insured).0

*** is secure.

*** Orchestrator:
Recv(a, Enc[k_book](k_u) : Enc(EKey*EKey)).
Send(c, Enc[k_book](k_u) : Enc(EKey*EKey)).
Recv(a, Enc[k_u](money) : Enc(Money*DKey)).
Send(c, Enc[k_u](money) : Enc(Money*DKey)).
Recv(c, Enc[k_u](ticket) : Enc(Ticket*EKey)).
Send(b, Enc[k_ins](ticket) : Enc(Ticket*EKey)).
Recv(b, insured_ticket : Insured).0

*** is secure.

*** Elapsed time: 0.008
```

Figure 4. Results: two secure orchestrators found.

V. Semiring-based selection of orchestrators

Even though all synthesized orchestrators are inherently secure, it is worth to make some specific considerations on how to order the results of the synthesis process. Clearly, non-functional requirements, such as QoS, could be used to evaluate quantitative aspects on orchestrators. Non functional aspects, including the degree of security, can be formalised using semirings, as done in [10]. Therein, security is a boolean value, determined by the violation of a policy. In this work, we are able to take into account some more context, namely, the presence of an intruder equipped with a specified knowledge base, and capable of deriving facts from it. Therefore, in this section, we provide an ordering of secure orchestrators based on the relationship between the knowledge of a possible attacker and the cost of attacks based on such knowledge. Indeed, PaMoChSA provides us a practical way to account for possible attacks by bulding the state graph in such a way that additional transitions are present, simulating eavesdropping and manipulation of messages by the intruder (with the exception of communications happening on hide channels). Finally, the knowledge of an intruder is always augmented with the messages that are exchanged between the orchestrator and the system, unless they used hidden channel. Rationale is that the intruder can eavesdrop such communications in order to acquire new information.

Moreover, sometimes it could be not possible to satisfy both functional and security requirements. In particular, we can synthesize a functional but not-secure orchestrator. In this case, the designer can decide to prioritize orchestrator with respect to functional aspects by using the same approach based on C-Semirings.

Referring to our example, according to the knowledge of the intruder, it is not always possible to find secure orchestrators. In those cases, the PaMoChSA tool is able to list not only the "only-functional" orchestrator, but also to list all the possible attackers for each orchestrator.

Example 5.1: Let us consider the case in which the initial knowledge of the intruder is augmented with the descritpion key of the user as follows:

```
<KNOWLEDGE> k_book : EKey; k_ins : EKey;
k_attack : EKey; k_attack : DKey;
k_u : Dkey </KNOWLEDGE>
```

In this case, the tool answers that no secure orchestrator can be found. As additional information, the tool outputs the list of non-secure orchestrators and their attacks. In Figure 5 is showed an excerpt of the output.

Using this information, it is possible to order orchestrators according to the number of the attackers with respect to the attacker knowledge. Indeed, if we change the knowledge again, for instance, removing the user decryption key but adding the encryption one, we obtain the result showed in Figure 6.

This is coherent with the intuition. Having the private key of the user allows the intuder to perform more attaks to the system that having the public one.

Hence, in order to prioritize orchestrators, let us consider the semiring $\mathbb{K} = (K, \cap, \cup, K, \emptyset)$, where K is the set of subsets of typed messages; the additive operation is intersection, and the associated partial order is reverse inclusion. This semiring represents the knowledge of an attacker in several different possible worlds. To each such subset ϕ_X, one associates the set of orchestrators that are secure for ϕ_X, that is, there are no attacks starting from that knowledge. Write $\phi_X \vdash O$ whenever O is in this set. Clearly, whenever $\phi_X \subseteq \phi'_X$ and $\phi'_X \vdash O$, we have that $\phi_X \vdash O$, that is, the less the knowledge of the attacker, the easier is for an orchestrator to be secure. Indeed, monotonicity in the other direction does not hold, since all the sets that contain the secret m make the system necessarily not secure.

Let $\sqsubseteq$ be the order associated to a semiring $\mathbb{C}$ that measures the degree of security of the system.

Definition 5.1: Given orchestrators O_1 and O_2, write $O_1 \lesssim_{\phi_X} O_2$ whenever $S \parallel O_1 \sqsubseteq S \parallel O_2$.
Using this definition, we can introduce the following order among orchestrators.

Definition 5.2: The order relation $\leq$ is defined as $O_1 \leq O_2$ whenever, for all ϕ_X, $O_1 \lesssim \phi_X O_2$.

VI. A Mapping from Crypto-CCS to BPMN

Once an orchestrator has been chosen, it is possible to map it again into a BPMN process. The reverse mapping is

```
*** Orchestrator:
Recv(a, Enc[k_book](k_u) : Enc(EKey*EKey)).
Recv(a, Enc[k_u](money) : Enc(Money*DKey)).
Send(c, Enc[k_book](k_u) : Enc(EKey*EKey)).
Send(c, Enc[k_u](money) : Enc(Money*DKey)).
Recv(c, Enc[k_u](ticket) : Enc(Ticket*EKey)).
Send(b, Enc[k_ins](ticket) : Enc(Ticket*EKey)).
Recv(b, insured_ticket : Insured).0

*** has 4810 attacks, listed below.
Eavesdrop(c, Enc[k_u](ticket))

Receive(c, Enc[k_u](ticket) : Enc(Ticket*EKey))

Eavesdrop(c, Enc[k_u](money))
Eavesdrop(c, Enc[k_u](ticket))

Eavesdrop(c, Enc[k_u](money))
Receive(c, Enc[k_u](ticket) : Enc(Ticket*EKey))

Receive(c, Enc[k_u](money) : Enc(Money*DKey))
Send(c, Enc[k_u](money))
Eavesdrop(c, Enc[k_u](ticket))

Receive(c, Enc[k_u](money) : Enc(Money*DKey))
Send(c, Enc[k_u](money))
Receive(c, Enc[k_u](ticket) : Enc(Ticket*EKey))

Eavesdrop(c, Enc[k_book](k_u))
Eavesdrop(c, Enc[k_u](ticket))

Eavesdrop(c, Enc[k_book](k_u))
Receive(c, Enc[k_u](ticket) : Enc(Ticket*EKey))

Eavesdrop(c, Enc[k_book](k_u))
Eavesdrop(c, Enc[k_u](money))
Eavesdrop(c, Enc[k_u](ticket))

Eavesdrop(c, Enc[k_book](k_u))
Eavesdrop(c, Enc[k_u](money))
Receive(c, Enc[k_u](ticket) : Enc(Ticket*EKey))

Eavesdrop(c, Enc[k_book](k_u))
Receive(c, Enc[k_u](money) : Enc(Money*DKey))
Send(c, Enc[k_u](money))
Eavesdrop(c, Enc[k_u](ticket))

.....
```

Figure 5. The result from the orchestration process when the attacker knows the public key of the user.

only needed for sequential BPMN processes whose tasks are only send tasks or receive tasks, as orchestrators synthesized by PaMoChSA always are sequential, communicating processes with no internal actions. The backwards translation is therefore straightforward. The resulting process has a send or receive task for each input or output actions of the orchestrator, connected to the corresponding task in another process by a BPMN message flow

Using the mapping function, we obtain the BPMN specification of the chosen orchestrator. In Figure 7 we show

```
*** Orchestrator:
Recv(a, Enc[k_book](k_u) : Enc(EKey*EKey)).
Recv(a, Enc[k_u](money) : Enc(Money*DKey)).
Send(c, Enc[k_book](k_u) : Enc(EKey*EKey)).
Send(c, Enc[k_u](money) : Enc(Money*DKey)).
Recv(c, Enc[k_u](ticket) : Enc(Ticket*EKey)).
Send(b, Enc[k_ins](ticket) : Enc(Ticket*EKey)).
Recv(b, insured_ticket : Insured).0

*** has 9229 attacks, listed below.
Receive(c, Enc[k_book](k_u) : Enc(EKey*EKey))
Receive(c, Enc[k_u](money) : Enc(Money*DKey))
Send(c, Enc[k_book](k_attack))
Send(c, Enc[k_attack](money))
Eavesdrop(c, Enc[k_attack](ticket))

Receive(c, Enc[k_book](k_u) : Enc(EKey*EKey))
Receive(c, Enc[k_u](money) : Enc(Money*DKey))
Send(c, Enc[k_book](k_attack))
Send(c, Enc[k_attack](money))
Receive(c, Enc[k_attack](ticket) :
 Enc(Ticket*EKey))

Receive(c, Enc[k_book](k_u) : Enc(EKey*EKey))
Send(c, Enc[k_book](k_attack))
Receive(c, Enc[k_u](money) : Enc(Money*DKey))
Send(c, Enc[k_attack](money))
Eavesdrop(c, Enc[k_attack](ticket))

Receive(c, Enc[k_book](k_u) : Enc(EKey*EKey))
Send(c, Enc[k_book](k_attack))
Receive(c, Enc[k_u](money) : Enc(Money*DKey))
Send(c, Enc[k_attack](money))
Receive(c, Enc[k_attack](ticket) :
 Enc(Ticket*EKey))

Send(c, Enc[k_book](k_attack))
Receive(c, Enc[k_book](k_u) : Enc(EKey*EKey))
Receive(c, Enc[k_u](money) : Enc(Money*DKey))
Send(c, Enc[k_attack](money))
Eavesdrop(c, Enc[k_attack](ticket))

Send(c, Enc[k_book](k_attack))
Receive(c, Enc[k_book](k_u) : Enc(EKey*EKey))
Receive(c, Enc[k_u](money) : Enc(Money*DKey))
Send(c, Enc[k_attack](money))
Receive(c, Enc[k_attack](ticket) :
 Enc(Ticket*EKey))

Eavesdrop(a, Enc[k_u](money))
Receive(c, Enc[k_book](k_u) : Enc(EKey*EKey))
Receive(c, Enc[k_u](money) : Enc(Money*DKey))

....
```

Figure 6. The result from the orchestration process when the attacker knows the private key of the user.

the compound system in which the orchestrator black box becomes a white box. The internal description is the one of the first orchestrator obtained by the tool (Figure 4).

Remark 6.1: Converting a Crypto-CCS process to a BPMN process and back results in the identity transformation. The reverse identity holds for sequential processes.

Indeed, the "backwards" translation from Crypto-CCS to BPMN is only defined for sequential processes, since our orchestrators are sequential.

VII. Related work

This paper covers several phases of the synthesis and selection of an orchestrator process: i) inclusion of security requirements in BPMN; ii) exhaustive verification of secrecy properties in the service composition; iii) automatic synthesis of secure orchestrators; and iv) orchestration ranking according to how much information they disclose. Here we recall some related work about almost all these phases. There are many papers that describe possible security extensions of BPMN. Brucker etal. [11] extend the visual notation of BPMN to support role-based access control. Roles and responsibilities are described at design time and later enforced at run time. In comparison, our extension of BPMN is more modest in the sense that we only included cryptographic primitives as part of the activity names, and thus less visual, but similar roles can be defined using different private keys on each activity. Other concerns as the *separation of duties* and *need to know* are covered also by our approach by enforcing the correct termination of all the services involved and by analysing the knowledge gained by each participant.

In the existing literature, we can find several works dealing with the mapping of BPMN into a process algebra languages. For instance, [12] proposes a traslation of BPMN into COWS in order to formally analyse BPMN processes.

In order to facilitate the use of model-checking techniques to business analysts, the authors of [13] created a model-checking plugin for SAP NetWeaver Business Process Management. This plugin support the verification of secrecy properties with a push of a button and the subsequent visualization of possible attack traces. However, since this plugin is intended as a design tool, the designer is left with the task to solve possible flaws in the business process. Our approach, on the other hand, automatizes the generation of secure orchestrators which are guaranteed to preserve the given secrecy properties.

On the other hand, there are some papers proposing compositional approaches to the synthesis of controllers, able to dynamically enhance security, depending on some runtime behaviour of a possible attacker, *e.g.,* [14], [15].

The current work extends the existing research line on the synthesis of secure controller programs [14] with the introduction of cryptographic primitives. Also, it tries to simplify the approach in [16] for the synthesis of deadlock-free orchestrators that are compliant with security adaptation contracts [17]. Compared to [16], this new approach loses the ability to specify fine-grained constrains in the desired orchestration but, on the other hand, there is no need to design and adaptation contract.

Similarly, our approach to synthesis differs from the one in [18], where automatic composition of services under security policies is investigated. Work in [18] uses the AVISPA tool [19] and acts in two stages: first, it derives a protocol allowing composition of some services; then, some desired security properties are implemented. The latter step uses the functionality of AVISPA and, for the former step, the desired composition is turned into a security property, so that AVISPA itself can be used to derive an "attacker" which actually is the orchestrator.

In [20], Li et al. present an approach for securing distributed adaptation. A plan is synthesized and executed, allowing the different parties to apply a set of data transformations in a distributed fashion. In particular, the authors synthesize "security boxes" that wrap services, providing them with the appropriate cryptographic capabilities. Security boxes are pre-designed, but interchangeable at run time. Our approach, in contrast, lacks dynamic planning. However, in our case the orchestrator is synthesized at run time and is able to cryptographically arrange secure service composition.

VIII. Conclusions

In this paper we have exploited and extended the tool presented in [4] in order to automatically synthesize functional and secure BPMN orcehstrators. Indeed, we have extended the tool PaMoChSA2012 with a mapping function that allows to map BPMN processes into Crypto-CCS ones and back. In this way, we also associate cryptographic primitives to BPMN processes as a secret specified by the user. Furthermore, exploiting theory about quantitative security, we are able to provide a method for ranking different orchestrator in order to help the user to select the optimal one for reaching their goal.

References

[1] Introduction to omg's unified modeling language. [Online]. Available: http://www.omg.org/gettingstarted/what_is_uml.htm

[2] Business process model and notation (bpmn). [Online]. Available: http://www.omg.org/spec/BPMN/2.0/PDF/

[3] F. Martinelli, "Analysis of security protocols as open systems," *TCS*, vol. 290, no. 1, pp. 1057–1106, 2003.

[4] V. Ciancia, J. A. Martin, F. Martinelli, I. Matteucci, M. Petrocchi, and E. Pimentel, "A tool for the synthesis of cryptographic orchestrators," in *Model Driven Security Workshop, MDSEC*, ACM, Ed., 2012.

[5] S. Bistarelli, *Semirings for Soft Constraint Solving and Programming*, ser. LNCS. SpringerVerlag, 2004, vol. 2962.

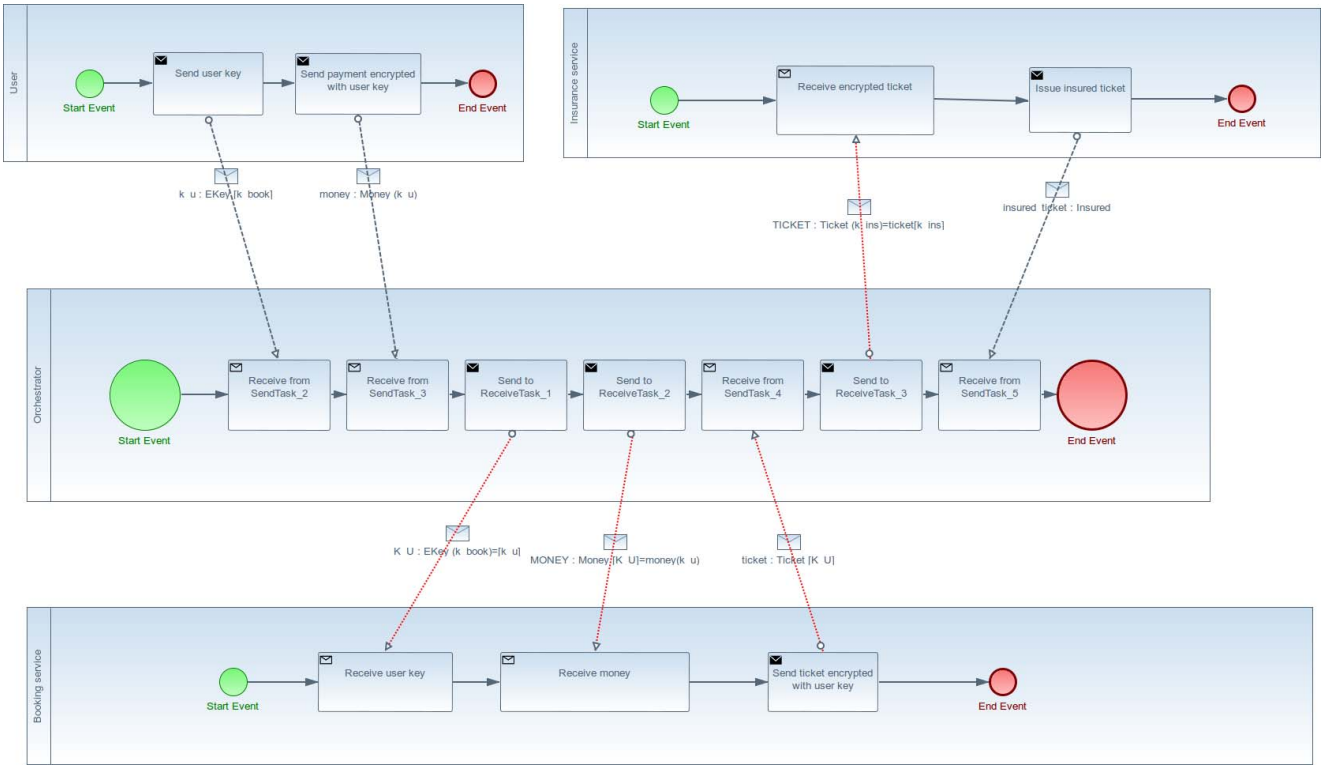

Figure 7. The BPMN specification of the compound system orchestrated by the synthesized orchestrator.

[6] "PaMoChSA 2012," http://www.iit.cnr.it/staff/vincenzo.ciancia/tools.html.

[7] Xquery 3.0: An xml query language. [Online]. Available: www.w3.org/TR/xquery-30/

[8] R. Milner, *Communication and concurrency*. Prentice-Hall, 1989.

[9] F. Martinelli, M. Petrocchi, and A. Vaccarelli, "Automated analysis of some security mechanisms of SCEP," in *ISC*. Springer, 2002, pp. 414–427.

[10] V. Ciancia, F. Martinelli, I. Matteucci, and C. Morisset, "Quantitative evaluation of enforcement strategies," Istituto di Informatica e Telematica - Consiglio Nazionale delle Ricerche, Tech. Rep. TR-04-13, 2013.

[11] A. D. Brucker, I. Hang, G. Lückemeyer, and R. Ruparel, "Securebpmn: modeling and enforcing access control requirements in business processes," in *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, ser. SACMAT '12. New York, NY, USA: ACM, 2012, pp. 123–126. [Online]. Available: http://doi.acm.org/10.1145/2295136.2295160

[12] D. Prandi, P. Quaglia, and N. Zannone, "Formal analysis of bpmn via a translation into cows," in *Proceedings of the 10th international conference on Coordination models and languages*, ser. COORDINATION'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 249–263. [Online]. Available: http://dl.acm.org/citation.cfm?id=1788954.1788970

[13] W. Arsac, L. Compagna, G. Pellegrino, and S. E. Ponta, "Security validation of business processes via model-checking," in *Proceedings of the Third international conference on Engineering secure software and systems*, ser. ESSoS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 29–42. [Online]. Available: http://dl.acm.org/citation.cfm?id=1946341.1946345

[14] F. Martinelli and I. Matteucci, "A framework for automatic generation of security controller," *STVR*, 2010.

[15] O. Maler, A. Pnueli, and J. Sifakis, "On the synthesis of discrete controllers for timed systems," in *STACS*, ser. LNCS, vol. 900. Springer, 2005, pp. 229–242.

[16] J. A. Martín, F. Martinelli, and E. Pimentel, "Synthesis of secure adaptors," *J. Log. Algebr. Program.*, vol. 81, no. 2, pp. 99–126, 2012.

[17] J. A. Martín and E. Pimentel, "Contracts for security adaptation," *J. Log. Algebr. Program.*, vol. 80, no. 3-5, pp. 154–179, 2011.

[18] Y. Chevalier, M. A. Mekki, and M. Rusinowitch, "Automatic composition of services with security policies," in *SERVICES'08 - Part I*. IEEE, 2008, pp. 529–537.

[19] L. Viganò, "Automated security protocol analysis with the AVISPA tool," *ENTCS*, vol. 155, pp. 69–86, 2006.

[20] J. Li, M. Yarvis, and P. Reiher, "Securing distributed adaptation," *Computer Networks*, vol. 38, no. 3, 2002.