Formal analysis of BPMN via a translation into COWS

Davide Prandi[1], Paola Quaglia[2], and Nicola Zannone[3]

[1] Dip. di Medicina Sperimentale e Clinica, Univ. Magna Graecia di Catanzaro, Italy
[2] Dip. di Ing. e Scienza dell'Informazione, Univ. di Trento, Italy
[3] Dep. of Computer Science, Univ. of Toronto, Canada

Abstract. A translation of the Business Process Modeling Notation into the process calculus COWS is presented. The stochastic extension of COWS is then exploited to address quantitative reasoning about the behaviour of business processes. An example of such reasoning is shown by running the PRISM probabilistic model checker on a case study.

1 Introduction

A challenging question for organisations is how to create strong, yet flexible, business processes. Business Process Management is emerging as a means for understanding the activities that organisations can perform to optimise their business processes or to adapt them to new organisational needs. Specifically, it defines the activities to be performed by organisations to manage and, if necessary, to improve their business processes. Business Process Management activities concern the design and capture of existing business processes as well as the analysis of new ones. In this setting, the definition of languages for modelling business processes is a key step in Business Process Management due to the need of describing their structure and behaviour.

Different languages have been proposed in literature to model business processes (e.g., [6, 25, 27]). Among them, the Business Process Modeling Notation (BPMN) [19] is emerging as the de-facto standard modelling notation in industry. BPMN was designed to provide a graphical notation for XML-based business process languages, such as WS-BPEL [18]. Therefore, business analysts can take advantage from the use of BPMN since they can exploit facilities for generating executable WS-BPEL code from BPMN graphical models [20]. Unfortunately, BPMN is informal and leaves room for ambiguity about its semantics [5]. Moreover, it does not allow for formal analysis. These issues are challenging and call for formal frameworks encoding graphical elements into formal specifications.

Process calculi have been proved powerful enough to formalise the activities performed within a business process, to render in a natural way the parallelism and concurrency of interactions among participants as well as to analyse the overall process behaviour. In particular, in this paper we present a translation

This work has been partially sponsored by the project SENSORIA, IST-2005-016004.

of BPMN into the Calculus of Orchestration of Web Services (COWS) [13]. A stochastic extension of COWS [22] is then exploited to obtain semantic models that are quantitatively verified using the PRISM [10] model checker. Such a quantitative reasoning aims to assist system designers in the evaluation and comparison of design alternatives with the intent of driving them in the selection of an appropriate infrastructure supporting the business process.

The choice of COWS is motivated, on one hand, by the fact that, being a foundational calculus, it is based on a very small set of primitives associated with a formal operational semantics that can be exploited for the automated derivation of the behaviour of the specified services. On the other hand, the language is strongly inspired by WS-BPEL, providing, among the rest, macros for fault and compensation handlers. Not least, an on-the-fly model checker for the qualitative verification of COWS specifications is already available [7]. So the proposed translation of BPMN into COWS allows testing business processes against, e.g., responsiveness properties like "after a request, a response is eventually sent to the requesting customer".

The paper is organised as follows. Sec. 2 and Sec. 3 present a brief overview of BPMN and of COWS, respectively. A small business process used as a case study is also illustrated. The translation from BPMN to COWS is then reported in Sec. 4. The following section (Sec. 5) provides some reasoning on the quantitative analysis carried on the case study via PRISM. Sec. 6 discusses related works and presents some final remarks.

2 BPMN

BPMN [19] provides a standard graphical notation for business process modelling. A business process is represented as a Business Process Diagram (BPD), which is composed of a set of partially ordered activities executed by the participants of the process. A BPD is essentially a collection of *pools*, *objects*, *sequence flows*, and *message flows*. Pools represent the participants to the business process. Objects can be *events*, *activities*, or *gateways*. Sequence flows determine the execution order between two objects in the same pool. The behaviour of a process is described by tracking the path(s) of a *token* through a process. A token is an abstract object that traverses the sequence flow passing through the objects of the process. Finally, message flows represent message exchange between two objects in different pools. Fig. 1 presents the core subset of BPMN elements which populate the domain of our translation into COWS.

Events may represent the start of a process (*start event*), the end of a process (*end event*), or something that might happen during the process (*intermediate event*). Intermediate events can also be attached to task. Different types of events are available. Here we consider *none event*, *message event*, and *error event*. None events are used when the modeller does not specify the type and can be start or end events. The meaning of a message event depends on its "position" in the BPD. A start message event represents the fact that a message arrives from a participant and triggers the start of the process; an end message event

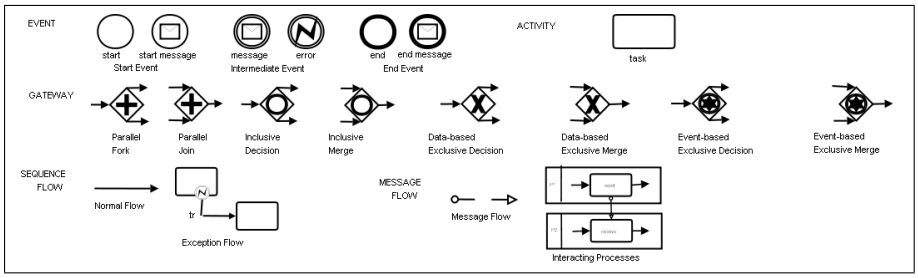


Fig. 1. A core subset of BPMN elements

indicates that a message is sent to a participant at the end of the process; and an intermediate message event indicates that a message arrives from or is sent to a participant during the process execution. An error message is for error handling. If the error is part of a normal flow, it throws an error; if it is attached to the boundary of an activity, it catches the error.

An activity is either a *task* or a *subprocess* (For the sake of simplicity, subprocesses are not dealt with in this paper). A task is an atomic activity. BPMN defines different task types. Here we consider *service tasks*, which provide some service, *receive tasks*, which wait for a message from another participant, *send tasks*, which send a message to another participant, and *none tasks*, which do nothing.

A gateway is a connector used to control sequence flows. Different types of gateways have been defined in BPMN. A *parallel fork gateway* is used to create parallel flows and a *parallel join gateway* is used to synchronise incoming parallel flows. An *exclusive decision gateway* defines a set of alternative paths, each of them is associated with a conditional expression. Only one path can be taken during the execution of the process on the basis of conditions. Conditions may be based either on *data-base entries* or on *external events*. An *exclusive merge gateway* is used as a merge for alternative sequence flows. An *inclusive decision gateway* is a branching point where each alternative is associated with a condition. Differently from exclusive decision, all sequence flows with a true evaluation of the corresponding condition will be traversed. Finally, an *inclusive merge gateway* synchronises all tokens produced upstream.

To illustrate what BPDs are, Fig. 2 shows the diagram for a credit request scenario. This example is an excerpt of a case study analysed in the course of the SENSORIA project. The goal of the scenario is twofold: ensuring the due support to the customer during his credit request application, and reducing the effort of bank employees in preparing an offer. The customer invokes the credit portal. If the authentication process succeeds, he is required to insert his data, the desired credit amount, and security values; otherwise, the process terminates and an error message is produced. The information inserted by the customer is checked against consistency and for validation purposes. In particular, a validation service is invoked. If such a service returns a positive answer, the data are uploaded to

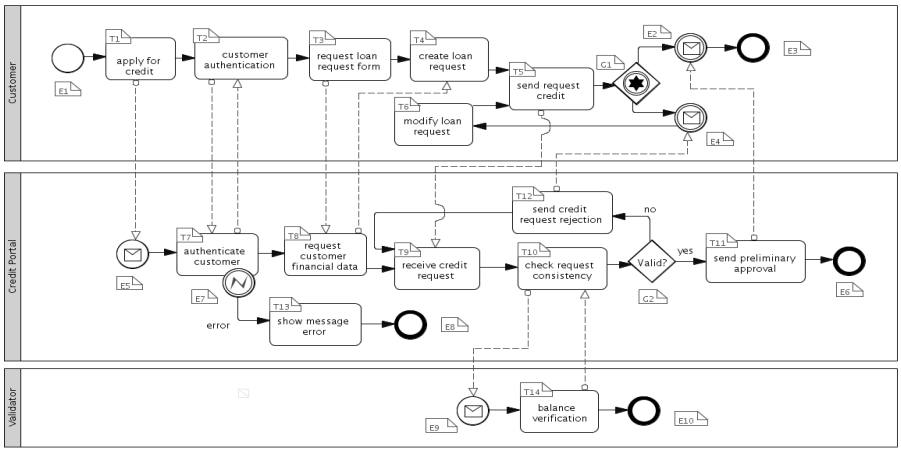


Fig. 2. Credit Request Process in BPMN

the bank and the process goes on. Otherwise, the customer has to update his data.

3 COWS: a short overview

In this section we present COWS [13], a foundational language for SOC that combines elements of well-known process calculi (e.g., the π-calculus [17]) with constructs inspired by WS-BPEL [18]. The computational units of COWS are *services*. They are expressed as structured activities built by combining basic activities by means of a small number of primitive operators. As it is typical for many process calculi, COWS services are given a formal operational semantics in terms of a set of syntax-driven axioms and rules which can be used to describe the dynamics of the system at hand. Specifically, those rules define a *transition relation* $\rightarrow$, with $s \rightarrow s'$ meaning that service s can execute a computation step and transform into service s'. Interested readers are referred to [13] for a detailed description of COWS semantics. Here we just provide an overview of the language and of the interpretation of its constructs.

The syntax of COWS is based on three countable and pairwise disjoint sets: the set of *names*, the set of *variables*, and the set of *killer labels*. The very basic activities in COWS are request and invoke operations which occur at *endpoints*. In [13], endpoints are identified by both a *partner* and an *operation* name. Here, for simplifying the notation, we let endpoints be denoted by single identifiers, and consider a monadic version of the calculus, that is, we suppose that request/invoke interactions can carry one single (vs. many) parameter at a time.

The terms of the COWS language are generated by the following grammar:

$$s ::= u!\,w \mid [\,d\,]s \mid g \mid s \mid s \mid \{\!|s|\!\} \mid \mathbf{kill}(k) \mid * \; s$$
$$g ::= \mathbf{0} \mid p?\,w.\,s \mid g + g$$

Intuitively, service $u!\,w$ performs an *invoke* (sending) activity over endpoint u with parameter w, where w can be either a name or a variable. The actual scope of parameters has to be explicitly defined using the delimitation operator $[\_]\_$. Basically, $[\,d\,]s$ denotes that the scope of d is exactly s, where d can be either a name or a variable or a killer label. Terms of the language can also be generated by guarded commands g. In this case services can either be the empty activity $\mathbf{0}$, or a choice between two guarded commands $(g+g)$, or a request-guarded service $p?\,w.\,s$ that waits for a matching communication over the endpoint p and then proceeds as s after the (possible) instantiation of the input parameter w. In what follows, whenever the parameter of an invoke or request activity is irrelevant, we simply write $u!$ and $p?$ for $u!\,x$ or $[\,x\,]p?\,x$, respectively. Also, we usually omit the trailing '$.\,\mathbf{0}$' from $p?\,w.\,\mathbf{0}$.

Furthermore, services can be described as parallel composition of other services, as, e.g., in $s_1 \mid s_2$. The intended behaviour of service $s_1 \mid s_2$ is given by all the possible communications given rise by matching the invoke (request) actions of s_1 over any endpoint p with the request (invoke) actions of s_2 over p. Interactions can take place only if either the involved parameters coincide or the request parameter is a variable. For instance,

$$p!\,n \mid [\,x\,](p?\,x.\,s) \rightarrow \mathbf{0} \mid s\{{}^{n}\!/\!_{x}\}$$

where x is used for variables and $s\{{}^{n}\!/\!_{x}\}$ represents for the substitution of n for x in service s.

The choice operator $\_ + \_$ models non-determinism. Either of its two arguments can be chosen, and this causes the other be discharged. For instance, service

$$p!\,n \mid ([\,x_1\,](p?\,x_1.\,s_1) + [\,x_2\,](p?\,x_2.\,s_2))$$

can evolve into either $0 \mid s_1\{{}^{n}\!/\!_{x_1}\}$ or $0 \mid s_2\{{}^{n}\!/\!_{x_2}\}$.

The protection primitive, written $\{\!|\_|\!\}$, saves from killer signals sent out by means of the $\mathbf{kill}(\_)$ primitive. The intended behaviour of $\mathbf{kill}(k)$ is to block the activities of all unprotected parallel services in the scope of the killer label k. For example, a kill activity inhibits unprotected communication

$$[\,k\,][\,w\,](p!\,n \mid p?\,w.\,s \mid \mathbf{kill}(k)) \rightarrow [\,k\,][\,w\,](\mathbf{0} \mid \mathbf{0} \mid \mathbf{0})$$

while, if the communication is protected,

$$[\,k\,][\,w\,](\{\!|p!\,n\,|p?\,w.\,s|\!\}\,|\,\mathbf{kill}(k)) \rightarrow [\,k\,][\,w\,](\{\!|p!\,n\,|p?\,w.\,s|\!\}\,|\,\mathbf{0})$$

The replication operator $*\_$ is used to model recursion. Intuitively, $*\,s$ behaves as the parallel composition $*\,s \mid s$, namely applying replication to s means that as many copies of s are spawned as necessary. For instance, the following evolution is possible:

$$\begin{array}{ll} *\,[\,w\,]p?\,w.\,s \mid p!\,n \mid p!\,m & \rightarrow \\ *\,[\,w\,]p?\,w.\,s \mid s\{{}^{n}\!/\!_{w}\} \mid \mathbf{0} \mid p!\,m & \rightarrow \\ *\,[\,w\,]p?\,w.\,s \mid s\{{}^{m}\!/\!_{w}\} \mid s\{{}^{n}\!/\!_{w}\} \mid \mathbf{0} \mid \mathbf{0} & \end{array}$$

The example shows the case when, at the first step, a copy of $[\,w\,]p?\,w.\,s$ communicates with $p!\,n$, and at the second step another copy interacts with $p!\,m$.

COWS allows some higher level imperative and orchestration constructs to be encoded as combinations of the small set of primitives described so far. The encoding of those constructs, written $\langle\!\langle\ \_\ \rangle\!\rangle$, is defined in full detail in [13]. Below we provide the intuition underpinning the constructors used in this work: imperative conditional statements, sequential compositions of services, and fault handlers. The encoding of these constructs is obtained as a combination of communications over reserved endpoints.

Conditional statements can be rendered as follows

$$\langle\!\langle \mathbf{if}\ c\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \rangle\!\rangle \triangleq [\,p\,](p!\,\widehat{c} \mid (p?\,\mathbf{true}.\,\langle\!\langle s_1 \rangle\!\rangle + p?\,\mathbf{false}.\,\langle\!\langle s_2 \rangle\!\rangle))$$

where $\widehat{c}$ stays for the evaluation of the condition c and can either assume the value **true** or the value **false**. If c is evaluated **true**, a communication between $p!\,\widehat{c}$ and $p?\,\mathbf{true}$ takes place enabling service s_1; otherwise, service s_2 is triggered. The scope of endpoint p is delimited to avoid interference with other services.

COWS does not natively support sequential composition of services, here written $s_1;\ s_2$. As in CCS [16], however, this can be encoded by resorting to invoke activities over a special endpoint for termination, say $p_{s_1\_done}$ for service s_1. Intuitively, if the encoding of s_1 is such that each possible execution path has $p_{s_1\_done}!$ as latest action, then a possible encoding for $s_1;\ s_2$ is the following:

$$\langle\!\langle s_1;\ s_2 \rangle\!\rangle \triangleq [\,p_{s_1\_done}\,](\langle\!\langle s_1 \rangle\!\rangle \mid p_{s_1\_done}?\,.\,\langle\!\langle s_2 \rangle\!\rangle)$$

Here notice that, because of killer activities, termination in COWS slightly differs from termination in CCS. A service s is actually terminating if no kill action is enabled when the unprotected $p_{s\_done}!$ can be performed.

Fault handlers can also be expressed in terms of COWS primitives. Here we present an encoding that departs from that proposed in [13] to meet the BPMN informal semantics. The fault generator activity $\mathbf{throw}(\phi)$ is used to rise a fault signal ϕ via the invoke $p_{fault\_\phi}!$ that triggers the execution of the appropriate handler. The scope activity $[s_1 : \mathbf{catch}(\phi)\{s_2\}]$ executes its normal behaviour s_1, until either s_1 terminates or a fault signal ϕ triggers the execution of the handler s_2:

$$\langle\!\langle [s_1 : \mathbf{catch}(\phi)\{s_2\}] \rangle\!\rangle \triangleq$$
$$[\,k\,][\,k'\,][\,p_{s_1\_done}\,](\langle\!\langle s_1 \rangle\!\rangle;\ p_{s_1\_done}! \mid p_{s_1\_done}?\,.\,\mathbf{kill}(k') \mid \langle\!\langle \mathbf{catch}(\phi)\{s_2\} \rangle\!\rangle_k)$$

If s_1 signals its termination through $p_{s_1\_done}!$, the catch activity is killed; otherwise s_2 is enabled and s_1 is killed by the catch:

$$\langle\!\langle \mathbf{catch}(\phi)\{s_2\} \rangle\!\rangle_k \triangleq p_{fault\_\phi}?\,.\,(\mathbf{kill}(k) \mid \{\!|\langle\!\langle s_2 \rangle\!\rangle|\!\})$$

4 From BPMN to COWS

This section provides the intuition underlying the translation of core BPDs into COWS services. Each BPMN object has an interface that is used to connect it

to other objects. This interface is made of request processing waiting for a token from the previous objects (if any), and invoke activities sending the token to the next objects (if any).

The basic idea of the translation is to interpret each object as a distinct COWS service. Services are then assembled using parallel compositions in such a way that COWS terms corresponding to connected objects can communicate to each other. Specifically, request activities correspond to incoming edges of the graph, and invoke activities to outgoing edges. Besides the interface, each object has a kernel, which defines the behaviour of the object itself in terms of the actions that are executed when the object is triggered. Our translation describes the message flow along the objects of the business process at hand. In this way, the transition system of the translating COWS service gives a compact representation of the possible paths of tokens within the business process.

The compositional translation of BPMN objects into COWS services is compactly presented in Tab. 1. As a pre-processing phase, we assume that each object of the diagram is firstly labelled by a name, so that no homonym between nodes and no homonym between edges is given raise. In particular, we adopt both in Tab. 1 and in the forthcoming examples the following easy labelling technique. First we associate distinct names with all the objects in the BPD, then we preliminarily give each edge the same name as the node it points to. If a node, say N, has more than one incoming edge, then the preliminary names of these edges are converted into $N1, N2, \ldots$ after an arbitrary ordering. Preliminary objects are otherwise confirmed.

A *none start event* starts a business process by generating a token, and it is modelled as a service performing the invoke activity $p_X!$, where X is the object pointed by the start event. (Here notice that the name p_X used as endpoint is automatically determined by the labels in the diagram). Symmetrically, a *none end event* E determines the end of the process and it is encoded as a service that consumes all the tokens generated upstream, that is, as a replicated request processing $p_E?$. The replication operator ensures that every incoming token is consumed by a none end event. Indeed, due for instance to the presence of cycles in the diagram, the number of produced tokens is not known a priori. A *message start event* is triggered when receiving a message by means of $p_E?\,w$ and then it activates the process by performing the invoke activity $p_Y!$.

Every time a *none task* T receives a token from the incoming flow (denoted by $p_T?$), a token for the outgoing sequence flow is generated (denoted by $p_Y!$ if the outgoing edge points to Y). A *receive task* T gets a token by means of a $p_{T1}?$ action, then it receives a message w when executing the activity $p_{T2}?\,w$, and it finally produces a token for next object Y. A *send task* T gets a token by means of $p_T?$, and later on it sends a message msg by performing $p_Z!$. When msg is sent, the task is completed [19, p. 65]. The token is thus passed to the next object Y. We do not provide an encoding of service tasks since their kernel depends on the particular service they are supposed to provide. Designers can define them in terms of existing constructs using, for instance, predefined patterns.

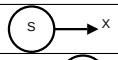
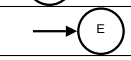
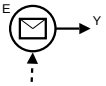
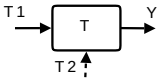
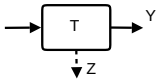
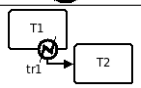
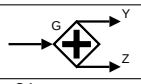
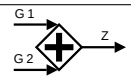
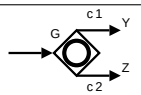
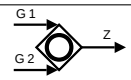
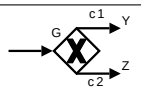
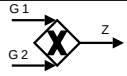
None start event		$p_X!$
None end event		$*\,p_E?\,.\,\mathbf{0}$
Message start event		$*\,[\,w\,]p_E?\,w.\,p_Y!$
None task		$*\,p_T?\,.\,p_Y!$
Receive task		$*\,[\,w\,]p_{T1}?\,.\,p_{T2}?\,w.\,p_Y!$
Send task		$*\,((p_T?\,.\,p_Z!\,msg);\,p_Y!\,)$
Error event (normal flow)		$*\,(p_E?\,.\,\mathbf{throw}(tr1);\,p_X!\,)$
Error event (exception flow)		$[\langle\!\langle T1\rangle\!\rangle : \mathbf{catch}(tr1)\{\langle\!\langle T2\rangle\!\rangle\}]$
Parallel fork gateway		$*\,p_G?\,.\,(p_Y!\,\vert\,p_Z!\,)$
Parallel join gateway		$*\,((p_{G1}?\ \vert\ p_{G2}?\,);\,p_Z!\,)$
Inclusive decision gateway		$*\,p_G?\,.\,(\mathbf{if}\ \widehat{c1}\ \mathbf{then}\,p_Y!\ \vert\ \mathbf{if}\ \widehat{c2}\ \mathbf{then}\,p_Z!\,)$
Inclusive merge gateway		$*\,((p_{G1}?\,+p_{G2}?\,+(p_{G1}?\ \vert\ p_{G2}?\,));\,p_Z!\,)$
Exclusive decision gateway		$*\,p_G?\,.\,(\mathbf{if}\ \widehat{c1}\ \mathbf{then}\,p_Y!\ \mathbf{else\ if}\ \widehat{c2}\ \mathbf{then}\,p_Z!\,)$
Exclusive merge gateway		$(*\,p_{G1}?\,.\,p_Z!\,)\ \vert\ (*\,p_{G2}?\,.\,p_Z!\,)$

Table 1. Translation of core BPMN into COWS

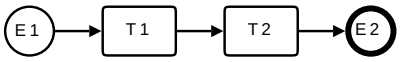


Fig. 3. BPD Example 1

As an easy example, consider the BPMN process in Fig. 3. Its complete translation into COWS is given as:

$$E1 \mid T1 \mid T2 \mid E2 = p_{T1}! \mid *\,p_{T1}?\,.\,p_{T2}! \mid *\,p_{T2}?\,.\,p_{E2}! \mid *\,p_{E2}?$$

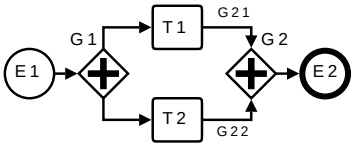


Fig. 4. BPD Example 2

The operational semantics of the calculus prescribes how sequence flows proceed within the service. Specifically:

$$p_{T1}! \mid *p_{T1}?\,.\,p_{T2}! \mid *p_{T2}?\,.\,p_{E2}! \mid *p_{E2}?\,.\,\mathbf{0} \rightarrow \quad (1)$$

$$\mathbf{0} \mid *p_{T1}?\,.\,p_{T2}! \mid p_{T2}! \mid *p_{T2}?\,.\,p_{E2}! \mid *p_{E2}?\,.\,\mathbf{0} \rightarrow \quad (2)$$

$$\mathbf{0} \mid *p_{T1}?\,.\,p_{T2}! \mid \mathbf{0} \mid *p_{T2}?\,.\,p_{E2}! \mid p_{E2}! \mid *p_{E2}?\,.\,\mathbf{0} \rightarrow \quad (3)$$

$$\mathbf{0} \mid *p_{T1}?\,.\,p_{T2}! \mid \mathbf{0} \mid *p_{T2}?\,.\,p_{E2}! \mid \mathbf{0} \mid *p_{E2}?\,.\,\mathbf{0} \mid \mathbf{0} \quad (4)$$

First, a token flows from $E1$ to $T1$ via a communication along the endpoint p_{T1}. Then the token is passed to $T2$, and finally it gets consumed by $E2$. The service highlighted in (4) is stuck as no matching invoke/request activities are enabled.

Error events are used for error handling. An error event in normal flow E receives the token, throws an error *tr1*, and releases the token. When an error event E is attached to the boundary of a task $T1$, $T1$ is executed until either it completes or a trigger *tr1* is caught. In this case, it generates an exception flow by stopping $T1$ and activating $T2$.

A *parallel fork gateway* G repeatedly performs a request processing at p_G, that is, it waits for a token, and then it proceeds as the parallel composition of two invoke activities p_Y and p_Z. Namely, it produces a token for each of the following objects `Y` and `Z`. A *parallel join gateway* first synchronises with its incoming sequence flows. This is represented as the parallel composition of the requests $p_{G1}?$ and $p_{G2}?$. A token is then released to the next object Z via the execution of the invoke activity $p_Z!$. As an example, service S below is the translation of the diagram in Fig. 4.

$$S = E1 \mid G1 \mid T1 \mid T2 \mid G2 \mid E2$$

where:

$$\begin{array}{ll} E1 = p_{G1}! & G1 = *p_{G1}?\,.\,(p_{T1}! \mid p_{T2}!) \\ T1 = *p_{T1}?\,.\,p_{G21}! & G2 = *((p_{G21}? \mid p_{G22}?)\,;\, p_{E2}!) \\ T2 = *p_{T2}?\,.\,p_{G22}! & E2 = *p_{E2}?\,.\,\mathbf{0} \end{array}$$

Fig. 5 describes the behaviour of service S by tracking how the tokens flow along the diagram. Initially, only the communication over the endpoint p_{G1} is enabled. This represents the fact that the token can only pass from the start event $E1$ to the parallel fork gateway $G1$, written $\xrightarrow{E1 \rightarrow G1}$ in Fig. 5. Then, the

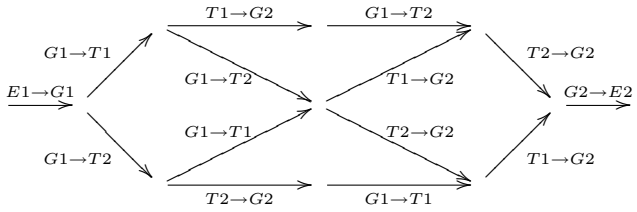


Fig. 5. Behaviour of the process of Fig. 4

service $G1$ generates a parallel flow and two invoke activities are simultaneously enabled: one over the endpoint p_{T1}, and the other over the endpoint p_{T2}. The interleaving semantics of the parallel composition of COWS services allows the derivation of non-trivial paths of the token flow. Consider, for instance, the top level path along the graph in Fig. 5, namely:

$$(E1{\rightarrow}G1)(G1{\rightarrow}T1)(T1{\rightarrow}G2)(G1{\rightarrow}T2)(T2{\rightarrow}G2)(G2{\rightarrow}E2)$$

This path is relative to the case when task $T1$ completes before task $T2$ gets its token. Also, Fig. 5 shows that a token cannot possibly reach the end event $E2$ before both $T1$ and $T2$ terminate their execution.

Inclusive decision gateways are translated as services that can transmit the token to the following objects, under the proviso that the corresponding condition is satisfied. One should notice that "if none of inclusive decision gate condition expressions are evaluated as true, then the process is considered to have an invalid model" [19, p. 78]. However, it is up to the modeller (rather than to the encoding) to ensure that at least one condition is evaluated true during the execution of the business process. To avoid possible problems, BPMN allows modellers to set a *default* gate that is selected if none of the other gates is chosen. In our approach the *default* gate is modelled as a gate whose condition is the negation of the disjunction of the other conditions. For instance, if gates have conditions c_1 and c_2, the *default* condition is $\neg(c_1 \vee c_2)$. *Inclusive merge gateways* synchronise all the tokens produced upstream. This makes the semantics non-local because it requires one to know how many tokens have been produced upstream before deciding whether to immediately release the token or wait. It is a matter of debate how to manage the non-local semantics of this sort of gateways (see for instance [24, 26]). Existing solutions, however, either impose some restrictions on the syntax of BPMN (e.g., avoiding cycles), or define a formal semantics that deviate from the informal one. For pragmatic reasons, we provide a semantics that "includes" the informal one. Since tokens can arrive at either p_{G1}, or p_{G2}, or both, inclusive merge gateways are translated using the choice operator and considering all possible alternatives. During the analysis, when global information are available, the transition system can be refined by cutting out those portions that do not correspond to possible behaviours of the business process. The motivations for our choice are twofold. First, to get com-

positionality we cannot consider global information at translation time. Second, any reasonable implementation of an inclusive merge gateway uses time-outs to stop waiting for tokens. In this way, the gateway might not synchronise even when both tokens arrive. This would introduce unexpected paths in the transition system which could be analysed in our quantitative framework.

Exclusive decision gateways are translated in such a way that the token passes to object Y only if condition $c1$ is true. If this is not the case, and if $c2$ is verified, then the token is sent to Z. Also, *default* gates may be used analogously to the case of inclusive decision gateways. *Exclusive merge gateways* are treated as simple by-passes: each incoming token, no matter where it is coming from, is passed to the next object Z. *Event-based exclusive gateways* (not reported in Tab. 1) are encoded in a similar way. The difference is that the selection of the gate depends on the caught event rather than on the satisfiability of conditions.

5 Quantitative analysis of business processes

Testing and visualising the behaviour of business processes in a timed context before implementing them allows one to correct or optimise the design of the system [4]. For instance, the designer of a new process may have different alternatives to implement the same task. The decision on which alternative should be adopted is usually driven by its cost and performance. System designers thus need tools to compare and evaluate possible design alternatives. In this section, we illustrate a quantitative reasoning on BPDs, relying on the encoding presented in Sec. 4.

The operational semantics of COWS provides a full qualitative description of the behaviour of business processes. Recently, a stochastic extension of COWS was presented [22], where the syntax and semantics of the calculus have been enriched along the lines of Markovian extensions of process calculi [9, 23]. In this way the semantic models associated with BPDs result to be Continuous Time Markov Chains, a popular model for automated verification. In the above mentioned extension, basic actions are associated with a random duration governed by a negative exponential distribution that is characterized by a unique parameter, called *rate*. In this way activities become pairs of the shape (μ,r), where μ represents the basic action, and r is the rate of μ. Once enabled, the probability that a certain activity (μ,r) is performed within a period of time of length t is $1 - e^{-rt}$. Also, the mean value of an exponentially distributed variable with parameter r is $1/r$, and rates to be associated with basic actions can be determined by estimating mean values for the various objects of BPDs. For instance, in our experiments we assumed that, depending on whether encryption is used or not, task T_7 of Fig. 2 (the authenticate costumer task) may require from 30 to 150 units of time. This corresponds to adopting rates varying from $0.0\overline{3}$ to $0.00\overline{6}$. More generally, running the same formula on chains obtained for different rate values of a certain action allows to analyze the sensitivity of the global behaviour to the duration of that particular action.

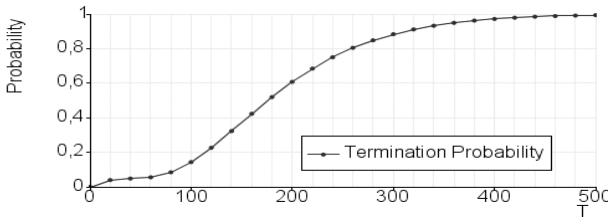


Fig. 6. Termination probability

For testing our approach, we have implemented the semantics of COWS in PRISM [10], a stochastic model checker for the formal modelling and analysis of systems. PRISM supports the automated analysis of a wide range of quantitative properties. The property specification language is based on the Continuous Stochastic Logic [1], a probabilistic extension of the classical temporal logic CTL.

In the remainder of this section, we report an excerpt of the analysis of the credit request process of Sec. 2 against a few properties. The symbol *terminate* is a short-end for any state representing that (*i*) either the costumer receives a preliminary approval for the credit loan request; (*ii*) or the customer fails the authentication procedure. Below we comment on the results obtained by checking our COWS service of the credit request process against the three logic formulae F1, F2, and F3.

- `F1 = P` ≥ 1 `[true U`≤ 240 *terminate*`]`. This formula expresses that *with probability* 1 *the system terminates in at most* 240 *units of time*, namely F1 is *true* if the system reaches one of the relevant states with probability 1 before 240 units of time. F1 is *false* for our model.
- `F2 = P=? [true U`≤ 240 *terminate*`]`. This formula refines the former one by asking *which is the probability that the system terminates in at most* 240 *units of time*. Here the result is *0.75*.
- `F3 = P=? [true U[T,T]` *terminate*`]`. This formula further refines the previous ones by asking *which is the probability that the system terminates at time T*. Fig. 6 shows the PRISM plot resulting with F3 for our COWS model.

Finally, we briefly mention how to use PRISM to evaluate possible implementations of a business process. In a business process some phases can be critical. For instance, authenticating the customer (task T_7 in Fig. 2) is a critical operation in the credit request process. Different authentication software can be used to implement T_7. Each of them provides a different level of encryption, more secure it is more time it requires. Fig. 7 plots the probability that the system terminates at time T when using different authentication software. With the faster software, 30 units of time, the probability of completing the procedure is higher than 0.85. As the authentication software employs more sophisticated encryption algorithms, the time duration increases, e.g., 60, 150, 300, and 600 units of time. It emerges that sophisticated authentication software can drastically affect the performance of the system. Therefore, system designers should make a

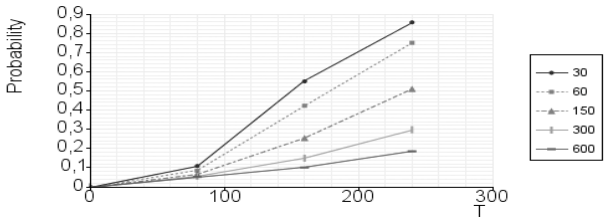


Fig. 7. Cost evaluation

trade off between the security and performance of the system when choosing the authentication software to be adopted. The framework proposed in this paper aims at assisting them in the decision making process.

6 Related works and concluding remarks

Few proposals have already addressed the problem of defining a formal semantics for BPMN. To the best of our knowledge, the first work in this setting is due to Wong et al. [28], who encoded BPMN in CSP. In particular, they used the language and behavioural semantics of CSP as denotational model. Based on it, they provided methods for specifying behavioural properties of business processes as well as for comparing BPMN diagrams. Differently from our work, the encoding is not compositional and so may not scale well to large business processes. Another definition of formal semantics for a subset of BPMN was made by Dijkman et al. [5]. They proposed a formal semantics of BPMN using Petri Nets. Our translation is defined over a super-set of the BPMN fragment considered in [5] where inclusive gateways are not dealt with, and tasks have one incoming sequence flow only, and can either send messages or receive messages, but cannot send and receive simultaneously. The translation presented in this paper relates well to [21], where an encoding of sequence and state diagrams into the π-calculus [17] is proposed. Similarly to our approach, objects of sequence and state diagrams are represented as π-calculus processes and then all such processes are composed by synchronising their interface via parallel composition.

Many interpretations of web services and business process specifications in terms of process calculi have been proposed over the last few years. For instance, Cámara et al. [3] provided an encoding of business process specifications in CCS [16]. Many efforts were also addressed to the enhancement of well-known process calculi with constructs inspired by those of WS-BPEL to better capture specific features of web service systems. Their common goal is to provide a sound mathematical ground to web services definitions as well as to improve their reliability using the analysis tools developed for process calculi. For instance, Meredith et al. [15] used process calculi with type systems [11] to check compatibility between web services. Other works [2, 8] extended the π-calculus with process mobility and with operations for data interaction, so getting a quite rich and

flexible model. Other proposals deal with issues of web transactions such as interruptible processes, failure handlers, and time. This is the case of [12, 14] that respectively present timed and untimed extensions of the π-calculus.

The aim of the present work is to set the basis for the development of a platform for the formal analysis of business processes. To this end, we have presented a semantic foundation for BPMN based on COWS, a calculus equipped with qualitative and quantitative operational semantics. The compositionality of the approach allows us to easily derive COWS specifications from the XML files generated by existing BPMN modelling applications. We have applied the PRISM probabilistic model checker to test and visualise the behaviour of a BPMN model in a timed context. This application intends to assist system designers in the decision making when implementing business processes. Currently, we are extending the platform with analysis facilities tailored to test if, with probability close to 1, the actual implementation of a business process is consistent with its specification.

References

1. A. Aziz, K. Sanwal, V. Singhal, and R.K. Brayton. Model-checking continous-time Markov chains. *ACM TOCL*, 1(1):162–170, 2000.
2. A.L. Brown Jr., C. Laneve, and L.G. Meredith. PiDuce: A Process Calculus with Native XML Datatypes. In *Proc. of EPEW/WS-FM*, LNCS 3670, pages 18–34. Springer, 2005.
3. J. Cámara, C. Canal, J. Cubo, and A. Vallecillo. Formalizing WSBPEL Business Processes Using Process Algebra. *Electr. Notes Theor. Comput. Sci.*, 154(1):159–173, 2006.
4. J. Desel and T. Erwin. Modeling, Simulation and Analysis of Business Processes. In *Business Process Management, Models, Techniques, and Empirical Studies*, pages 129–141. Springer, 2000.
5. R. Dijkman, M. Dumas, and C. Ouyang. Formal Semantics and Automated Analysis of BPMN Process Models. Preprint 7115, Queensland University of Technology, 2007.
6. R. Eshuis and R. Wieringa. Verification support for workflow design with UML activity graphs. In *Proc. of ICSE'02*, pages 166–176. ACM Press, 2002.
7. A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A model checking approach for verifying COWS specifications. In *Proc. of Fundamental Approaches to Software Engineering (FASE'08)*, LNCS. Springer, 2008. To appear.
8. P. Gardner and S. Maffeis. Modelling dynamic web data. *Theor. Comput. Sci.*, 342(1):104–131, 2005.
9. J. Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, 1996.
10. A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *Proc. of TACAS'06*, LNCS 3920, pages 441–444. Springer, 2006.
11. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.

12. C. Laneve and G. Zavattaro. Foundations of Web Transactions. In *Proc. of FoSSaCS'05*, LNCS 3441, pages 282–298. Springer, 2005.
13. A. Lapadula, R. Pugliese, and F. Tiezzi. Calculus for Orchestration of Web Services. In *Proc. of ESOP'07*, LNCS 4421, pages 33–47. Springer, 2007. Full version available at `http://rap.dsi.unifi.it/cows/`.
14. M. Mazzara and I. Lanese. Towards a unifying theory for web services composition. In *Proc. of WS-FM*, LNCS 4184, pages 257–272. Springer, 2006.
15. L.G. Meredith and S. Bjorg. Contracts and types. *Commun. ACM*, 46(10):41–47, 2003.
16. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice hall, 1989.
17. R Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.
18. OASIS. Web Services Business Process Execution Language – Version 2.0. Public Review Draft, 2006.
19. Object Management Group. Business Process Modeling Notation (BPMN) Specification. Final adopted specification, February 2006.
20. C. Ouyang, W. M. P. van der Aalst, M. Dumas, S. Breutel, and A. H.M. ter Hofstede. Translating BPMN to BPEL. BPM Report BPM-06-02, BPMcenter.org, 2006.
21. K. Pokozy-Korenblat and C. Priami. Toward Extracting π-calculus from UML Sequence and State Diagrams. *Electr. Notes Theor. Comput. Sci.*, 101:51–72, 2004.
22. D. Prandi and P. Quaglia. Stochastic COWS. In *Proc. of ICSOC'07*, LNCS 4749, pages 245–256. Springer, 2007.
23. C. Priami. Stochastic π-calculus. *The Computer Journal*, 38(7):578–589, 1995.
24. N. Russell, H.M. Arthur, W.M.P. van der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view. Technical report, BPMcenter.org, 2006.
25. W. M. P. Van der Aalst and M. Pesic. A declarative approach for flexible business processes management. In *Proc. of BPM'06*, LNCS 4103, pages 169–180. Springer, 2006.
26. W.M.P. van der Aalst, J. Desel, and E. Kindler. On the semantics of EPCs: A vicious circle. In *Business Process Management with Event driven Process Chains*, pages 71–79, 2002.
27. J. Vivas, J.A. Montenegro, and J. Lopez. A Formal Business Modelling Approach to Security Engineering with the UML. In *Proc. of ISC'03*, LNCS, pages 381–395. Springer, 2003.
28. P.Y.H. Wong and J. Gibbons. A Process Semantics for BPMN, 2007. Preprint, Oxford. University Computing Laboratory. Available at `http://web.comlab.ox.ac.uk/`.