On Correlation Sets and Correlation Exceptions in ActiveBPEL

Conference Paper · June 2011

DOI: 10.1007/978-3-642-30065-3_13

2 authors:

Hernán Melgratti
University of Buenos Aires
42 PUBLICATIONS **605** CITATIONS
SEE PROFILE

Christian Roldan
University of Buenos Aires
2 PUBLICATIONS **1** CITATION
SEE PROFILE

On correlation sets and correlation exceptions in ActiveBPEL [⋆]

Hernán Melgratti[1,2] and Christian Roldán[1]

[1] Departamento de Computación, FCEyN, Universidad de Buenos Aires.
[2] CONICET.

Abstract. Correlation sets are a programming primitive that allows instance identification in orchestration languages. A correlation set is a set of properties (i.e., values carried on by messages) that are used to associate each received message with a process instance: every time a service receives a message, it explores its content and determines a service instance that should handle the received message. Based on a concrete implementation, this paper proposes a formal model for correlation sets accounting for correlation exceptions. We also investigate different type systems aimed at ensuring that orchestrators are free from some kind of correlation exceptions.

1 Introduction

Service instances are a key concept when dealing with service composition. Typically, a service may have several instances that concurrently interact with different partners. Each service provides a template definition used to create process instances (all instances interact by using the same operations). Instances are created when the service receives a message that matches one of the start activities of its definition. For instance, a service that handles purchase orders creates a new instance any time it receives a new purchase order. All subsequent messages directed to the newly created instance should precisely identify the target instance. For example, when the client sends the payment details, the corresponding message should arrive to the correct instance (i.e., the one created when the client placed the purchase order). Orchestration languages (like the standard BPEL [4]) provide different alternatives to facilitate instance identification: they may rely on external mechanisms like dynamic endpoint references (as defined by WS-ADDRESSING [3]) or may use built-in primitives, like correlation sets. The main idea behind correlations sets is that messages carry on the identification of the instance they are targeted for, i.e., any service defines a set of properties that it will use to identify instances. Then, any instance is associated to a particular assignment of values to those selected properties. Consequently, any time a service receives a message, it compares the values of the received message against the values associated to any of its instances. The incoming message is routed to the matching instance.

Few approaches appeared in the literature have proposed a formal account for correlation sets, namely core correlation calculus [13], SOCK [6], COWS [9], and BLITE [10].

⋆ Research supported by ANPCyT Project BID-PICT-2008-00319, and UBACyT 20020090300122.

Each of these approaches proposes process calculi enriched with correlation primitives. Nevertheless, none of them includes a definition for correlation exceptions as defined in BPEL. In this paper, we aim at studying the relationship between correlation sets and correlation exceptions. We start by proposing a process calculus with a correlation mechanism, called Corr. Although Corr shares similarities with both SOCK, COWS, and BLITE, it is very different in scope. Basically, Corr is not aimed at providing a formal account for a complete orchestration language, consequently, several features that are usually present in composition languages, like scopes, compensations, fault and termination handlers, state, two-way operations, are not included in the calculus. For the sake of simplicity, we have preferred to focus on a minimal language exhibiting correlations and correlation exceptions and to leave orthogonal features outside of the model.

It has been shown in [10] that different implementations of BPEL exhibit discrepancies on the implementation of different primitives. In this paper, we follow the interpretation made by ActiveBPEL for setting the semantics of the correlation mechanism. The choices made during the design of Corr have been based on the runs of toy examples of BPEL orchestrators (such examples can be found at `http://www.di.unipi.it/~melgratt/activebpel`). Then, we use Corr to reason about exceptions originated by the correlation mechanism. We also propose a type system that singles out services that are free from different kind of correlation exceptions. The main result of this paper shows that well-typed services are free from correlation exceptions.

2 Correlation Language

We assume the countable sets of operation names $\mathcal{O}$ ranged over by $o, o_1, \ldots$; service names $\mathcal{S}$ ranged over by $s, s', \ldots$; data variables $\mathcal{V}$ range over by $x, y, \ldots$, data constants $\mathcal{A}$ ranged over by $a, a', \ldots$. We write v for either a data variable or constant, i.e., $v \in \mathcal{A} \cup \mathcal{V}$.

A correlation set C is a finite set of data variables, i.e., $C \subset \mathcal{V}$, and a correlation instance is a partial function $c : \mathcal{V} \to \mathcal{A} \cup \{\bot\}$. For any correlation set C, we denote with $C_\bot$ the uninitialized correlation instance, i.e., $dom(C_\bot) = C$ and $C_\bot(x) = \bot$ for all $x \in C$. We will say that two correlation instances c_1 and c_2 do not *collide* if and only if $\forall x \in dom(c_1) \cap dom(c_2).(c_1(x) \neq \bot \wedge c_2(x) \neq \bot \Rightarrow c_1(x) \neq c_2(x))$. We will explicitly write correlation instances as sets of pairs, for instance $c = \{x \mapsto \bot, y \mapsto b\}$. We will also use correlation instances as substitutions. When a correlation instance is applied to a term, we only substitute the variables that are mapped to values different from $\bot$. For instance, when $c = \{x \mapsto \bot, y \mapsto b\}$, $(\overline{o}\langle x, y\rangle; P)c = (\overline{o}\langle x, y\rangle; P)[b/y] = \overline{o}\langle x, b\rangle; P[b/y]$. Let c_1 and c_2 be correlation instances, we define the update operator $\_[\_]$ such that $dom(c_1[c_2]) = dom(c_1)$ and

$$c_1[c_2](x) = \begin{cases} c_1(x) & \text{if } x \notin dom(c_2) \vee (x \in dom(c_2) \wedge (c_1(x) = c_2(x) \vee c_2(x) = \bot)) \\ c_2(x) & \text{if } c_1(x) = \bot \wedge x \in dom(c_2) \\ \textit{undefined} & \text{Otherwise} \end{cases}$$

Definition 2.1 (Corr). *The syntax of flows, service instances and systems is given by the following grammar*

$$\begin{array}{lll}
(\text{FLOW}) & P ::= 0 \mid \sum_i o_i(\vec{x_i});P_i \mid \overline{o}\langle\vec{v}\rangle \mid P|P \mid P;P \mid \textbf{if } v = v' \textbf{ then } P \textbf{ else } P \mid \textbf{rec}_X\ P \mid X \\
(\text{INST}) & I ::= 0 \mid c \triangleright [P] \mid I|I \\
(\text{SYS}) & N ::= 0 \mid s^O_C\{P,I,M\} \mid N \| N \\
(\text{MSG}) & M ::= \emptyset \mid \overline{o}\langle\vec{v}\rangle|M
\end{array}$$

Flows are processes of value-passing CCS [11] with guarded choices, conditional if-then-else, and without any form of restriction. We include the operator ";" for sequential composition because we are not able to encode this primitive with the remaining ones (we do not have restriction). We only consider closed guarded recursive terms. The set of instances of a service can be either the empty set denoted by 0, the singleton containing the instance $c \triangleright [P]$ or the union of instances $I_1|I_2$. The instance $c \triangleright [P]$ denotes a service instance whose correlation variables have been initialized as described by c and its execution state is described by P. The simplest system is a service $s^O_C\{P,I,M\}$, where s is the name of the service, O is the set of ports or operations it provides, C is the set of correlated variables that it uses, P is the definition of the service, I are the active instances and M is the bag of all the received messages that are still pending.

We will refer to the set of input and output operations of a flow P (respectively, instances I and systems N), denoted $in(P)$ and $out(P)$ (respectively $in(I)$ and $out(I)$, and $in(N)$ and $out(N)$) as the sets of operation names that are subjects of the input and output prefixes occurring in P. We will write $subj(M)$ for the set of all operation names that appear as subjects of messages in M.

We remark that any name $x \in C$ acts as a binder in $s^O_C\{P,I,M\}$. For instance, all occurrences of x in $N = s^O_{\{x\}}\{o(x,y).P, \{x \mapsto a\} \triangleright [Q], M\}$ are bound to the correlation variable x. Note that x in $o(x,y).P$ is also bound to correlation name and cannot be α-renamed without renaming the correlation variable, i.e., $N \equiv_\alpha s^O_{\{z\}}\{o(z,y).P[z/x], \{z \mapsto a\} \triangleright [Q[z/x]], M\}$ for any fresh z. Contrastingly, $N \not\equiv_\alpha s^O_{\{x\}}\{o(z,y).P[z/x], \{x \mapsto a\} \triangleright [Q], M\}$. In what follows we consider only systems in which any two different input prefixes of a flow only share correlation variables (this constraint is analogous to Barendregt's hygiene convention).

We will restrict our attention to systems satisfying a well-formedness condition defined below. We first introduce some auxiliary notions.

Definition 2.2 **($C \blacktriangleright I$).** *A set of service instances I is correlated by a correlation set C, written $C \blacktriangleright I$, iff*

$$C \blacktriangleright 0 \qquad \frac{dom(c) = C}{C \blacktriangleright c \triangleright [P]} \qquad \frac{C \blacktriangleright I_1 \quad C \blacktriangleright I_2}{C \blacktriangleright I_1|I_2}$$

Definition 2.3 **(Input-blocked).** *We say a flow P is input blocked iff one of the following conditions holds*

$$\begin{array}{l}
P = \sum_i o_i(\vec{x_i});P_i \\
P = P_1|P_2 \textit{ with } P_1 \textit{ and } P_2 \textit{ input blocked} \\
P = P_1;P_2 \textit{ with } P_1 \textit{ input blocked}
\end{array}$$

The well-formedness condition is formally stated by the next definition.

Definition 2.4 (Well-formedness). *A flow P is well-formed if $in(P) \cap out(P) = \emptyset$. An instance $c \triangleright [P]$ is well-formed iff P is well-formed. A system $s_C^O\{P,I,M\}$ is well-formed iff all the following conditions hold:*

1. *The service definition P is well-formed and input-blocked;*
2. *All instances in I are well-formed;*
3. *Instances I are correlated by C, i.e., $C \blacktriangleright I$;*
4. *The input operations appearing in service instances I and service definition P are declared as operations provided by the service, i.e., $in(I) \cup in(P) \subseteq O$.*
5. *The bag of pending messages consists of messages for the operations provided by the service, i.e., $subj(M) \subseteq O$.*

A system $N \| N'$ is well-formed iff there are not input conflicts among different services, i.e., $in(N) \cap in(N') = \emptyset$.

The well-formedness condition states assumptions underlying business process models. Well-formed flows use input and output operations for communicating with third parties and not to establish intra service synchronization (i.e., $in(P) \cap out(P) = \emptyset$). This is a standard assumption implicit in most orchestration languages because each operation is associated to a particular partner link, which is different from the service itself. Our model does not include partner links explicitly but we require a consistent usage of operations by imposing a well-formed condition over systems. Orchestration languages provide particular primitives for internal synchronization, like links. (For simplicity's sake we do not include links in our model since this primitive is somehow orthogonal to correlation and exceptions). For services, we require all definitions to be input blocked (condition 1), which relates to the fact that activities should causally depend on start activities. Conditions 2, 3 and 4 stand for a relaxed form of a condition requiring instances to actually describe partial executions of service definitions.

2.1 Operational Semantics

In order to describe the dynamics of a system, we will consider an extended form of flows, which denotes the fact that an exception has been thrown. We add two additional forms of processes to account for the two different correlation exceptions defined by BPEL, namely, *ambiguous receive*(†) and *conflicting receive*(‡).

$$(\text{FLOW}) \quad P ::= \ldots \mid \dagger \mid \ddagger$$

The operational semantics of Corr is defined by a labeled transition system over well-formed terms, up-to the structural congruence defined below.

Definition 2.5 (Structural Congruence). *The structural congruence is the smallest congruence over the extended form of systems such that $|$, $+$, $\|$ are associative, commutative and have 0 as identity, $;$ is associative and have 0 as identity and the following axioms hold (P below does not contain † nor ‡).*

$$0;P \equiv P \qquad \dagger;P \equiv \dagger \qquad \ddagger;P \equiv \ddagger \qquad \dagger \mid P \equiv \dagger \qquad \ddagger \mid P \equiv \ddagger$$

Parallel composition of messages $|$ is associative, commutative and have $\emptyset$ as identity.

The labeled transition system considers the following actions, whose meaning is standard.

$$\alpha ::= o(\vec{v}) \mid \overline{o}\langle\vec{v}\rangle \mid \tau$$

We usually refer to $o(\vec{v})$ as a receive action instead of as an input and $\overline{o}\langle\vec{v}\rangle$ as an invoke instead of as an output.

The semantics of Corr is given by a Labeled Transition System (LTS) defined by structural induction on the syntax of the process, following Plotkin's Structural Operational Semantics (SOS) scheme [12]. Transitions for systems and instances are labeled by actions α as usual, while transitions for flows are labeled by pairs α, c, where α is an action and c is a correlation instance. A transition $P \xrightarrow{\alpha,c} P'$ denotes that P becomes P' by performing α and assigning variables as described by c. The semantics for flows is defined by using the auxiliary reduction relation $P \xmapsto{\alpha,c} P'$. The meaning of labels is analogous to $\xrightarrow{\alpha,c}$. The main difference between $\xmapsto{\alpha,c}$ and $\xrightarrow{\alpha,c}$ is that the latter accounts for correlation exceptions while the former does not (as explained below).

Definition 2.6. *The label transition system for Corr flows, instances and systems is defined by the rules in Figure 1.*

Rule (IN) is standard except for the fact that the label contains also the partial function $\vec{x}_i \mapsto \vec{v}$ recording the assignment of received values (this information is used by other rules to ensure that the use of correlated variables is consistent). Note that the function is set to $\emptyset$ in rule (OUT) because no variable is instantiated when a process performs an invoke. Rules (PAR), (SEQ), (REC), (THEN), (ELSE) are standard. Rule (NO-EXCP) states that a flow P can perform an action α without throwing any exception only when P can perform the action α (first premise) and there is no way (i.e., any other computation) for P to raise a correlation exception by performing the same action α (second and third premises). Rule (AMB-REC-EXCP) states that a flow that concurrently activates two receive operations for handling the same input action raises the ambiguous-receive exception. Differently, rule (CONF-REC-EXCP) states that a flow P raises the conflicting-receive exception after performing an action α if the residual of P after α enables two different input actions that are indistinguishable. There are two main differences between (AMB-REC-EXCP) and (CONF-REC-EXCP). Firstly, ambiguous-receive exception is raised when a flow attempts to perform an input action that can be handled in different ways while the conflicting-receive exception is raised when a flow performs an action α (note that it can be any action) and the residual enables at least two input actions for handling the same request. Secondly, the rules differ also on the conditions imposed over the instantiation of received variables. Rule (CONF-REC-EXCP) requires the same usage of received variables on conflicting inputs (i.e., last two premises require the same instantiation c_1) while (AMB-REC-EXCP) allows for different instantiation (note that premises use different correlations c_1 and c_2). These rules are aligned with BPEL specification statement that reads: "*If a business process instance simultaneously enables two or more IMAs [inbound message activities] for the same partnerLink, portType, operation but different correlationSet(s), and the correlations of multiple of these activities match an incoming request message, then the bpel:ambiguousReceive standard fault MUST be thrown by all IMAs whose correlation set(s) match the incoming message*". This makes clear that ambiguous-receive is raised when considering

FLOW

(IN)

$$\textstyle\sum_i o_i(\vec{x}_i);P_i \xmapsto{o_i(\vec{v}),\vec{x}_i\mapsto\vec{v}} P_i\{\vec{x}_i/\vec{v}\}$$

(OUT)

$$\overline{o_i}\langle\vec{a}\rangle \xmapsto{\overline{o_i}\langle\vec{a}\rangle,\emptyset} 0$$

(PAR)

$$\frac{P_1 \xmapsto{\alpha,c} P_1'}{P_1|P_2 \xmapsto{\alpha,c} P_1'|P_2}$$

(SEQ)

$$\frac{P_1 \xmapsto{\alpha,c} P_1'}{P_1;P_2 \xmapsto{\alpha,c} P_1';(P_2c)}$$

(REC)

$$\frac{P[^{\mathbf{rec}_X\ P}/_X] \xmapsto{\alpha,c} P'}{\mathbf{rec}_X\ P \xmapsto{\alpha,c} P'}$$

(THEN)

$$\mathbf{if}\ a=a\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 \xmapsto{\tau,\emptyset} P_1$$

(ELSE)

$$\frac{a \neq b}{\mathbf{if}\ a=b\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 \xmapsto{\tau,\emptyset} P_2}$$

(NO-EXCP)

$$\frac{P \xmapsto{\alpha,c} P' \quad P \not\xmapsto{\alpha,c_1} \dagger \quad P \not\xmapsto{\alpha,c_1} \ddagger}{P \xrightarrow{\alpha,c} P'}$$

(AMB-REC-EXCP)

$$\frac{P_1 \xmapsto{o(\vec{v}),c_1} P_1' \qquad P_2 \xmapsto{o(\vec{v}),c_2} P_2' \qquad c_1 \neq c_2}{P_1|P_2 \xrightarrow{o(\vec{v}),\emptyset} \dagger}$$

(CONF-REC-EXCP)

$$\frac{P \xmapsto{\alpha,c} P' \quad P' \equiv P_1|P_2 \quad P_1 \xmapsto{o(\vec{v}),c_1} P_1' \qquad P_2 \xmapsto{o(\vec{v}),c_1} P_2'}{P \xrightarrow{\alpha,c} \ddagger}$$

INSTANCES

(CORR)

$$\frac{P \xrightarrow{\alpha,c'} P' \qquad c[c']\ \text{defined}}{c \triangleright [P] \xrightarrow{\alpha} c[c'] \triangleright [P']}$$

(I-PAR)

$$\frac{I_1 \xrightarrow{\alpha} I_1'}{I_1|I_2 \xrightarrow{\alpha} I_1'|I_2}$$

SYSTEMS

(SVC-IN)

$$\frac{o \in O}{s_C^O\{P,I,M\} \xrightarrow{o(\vec{v})} s_C^O\{P,I,\overline{o}\langle\vec{v}\rangle|M\}}$$

(NEW)

$$\frac{C_\perp \triangleright [P] \xrightarrow{o(\vec{v})} c \triangleright [P']}{s_C^O\{P,I,\overline{o}\langle\vec{v}\rangle|M\} \xrightarrow{\tau} s_C^O\{P,I \mid c \triangleright [P'],M\}}$$

(DISPATCH)

$$\frac{I \xrightarrow{o(\vec{v})} I'}{s_C^O\{P,I,\overline{o}\langle\vec{v}\rangle|M\} \xrightarrow{\tau} s_C^O\{P,I',M\}}$$

(SVC-NON-IN)

$$\frac{I \xrightarrow{\alpha} I' \qquad \alpha \neq o(\vec{v})}{s_C^O\{P,I,M\} \xrightarrow{\alpha} s_C^O\{P,I',M\}}$$

(S-PAR)

$$\frac{N_1 \xrightarrow{\alpha} N_1'}{N_1\|N_2 \xrightarrow{\alpha} N_1'\|N_2}$$

(COMM)

$$\frac{N_1 \xrightarrow{o(\vec{v})} N_1' \quad N_2 \xrightarrow{\overline{o}\langle\vec{v}\rangle} N_2'}{N_1\|N_2 \xrightarrow{\tau} N_1'\|N_2'}$$

Fig. 1. Labeled Transition System for Corr.

different correlation sets (i.e., variables). For conflicting-receive, BPEL specification says: "*if two or more receive actions for the same partnerLink, portType, operation and correlationSet(s) are simultaneously enabled during execution, then the standard fault bpel:conflictingReceive MUST be thrown*". Consequently, conflicting-receive is thrown when considering the same correlation set. Moreover, ActiveBPEL implements this requirement as referring to IMAs of the same instance (as for ambiguous-receive) and does not impose restrictions over simultaneous enabling on different instances.

We also remark that our calculus does not provide exception handling and hence we make the whole instance to raise the exception. Models accounting for exception handling should keep track of the places in which exceptions are raised.

Rule (CORR) states that a service instance $c \triangleright [P]$ can perform an action only when the instantiation of variables induced by the execution of such action (i.e., the assignments described by c') is consistent with the correlation values of the instance (condition $c[c']$ defined). In this case, both P evolves to P' and the correlation is updated to $c[c']$. Since $c' = \emptyset$ when α is either τ or an invoke action, the correlation update has effect only for receive actions. Rule (I-PAR) deals with the behaviour of a set of multiple instances. Note that instances are independent from each other, which corresponds to the fact that ActiveBPEL does not impose correlation constraints between different instances.

We remark that services interact asynchronously and received messages are kept in a bag of received messages (rule (SCV-IN)) and then they are used either for creating a new instance (rule (NEW)) or in a received action of an existing instance (rule (DISPATCH)). Differently from other approaches such as COWS, rule (NEW) have no side conditions ensuring that it has less priority than (DISPATCH). This accounts for the fact that ActiveBPEL may create new instances even when some other instance may handle the received message. This behaviour appears related to an implementation aspect that introduces delay in the registration of ready inputs. For this reason, ActiveBPEL may create a new instance when an input action of an instance has not completed its registration. The semantics of Corr abstracts away from timing issues and specifies this kind of behaviours by introducing non-determinism for handling received messages. Rule (SCV-NON-IN) lifts non-input actions (i.e., outputs or silent moves) of an instance to the service level. Rules (PAR) and (COM) are standard. We only remark here that communication is possible only between two different services.

Notation We will write $\Rightarrow$ to denote the relation $\Rightarrow = \bigcup_{\alpha} \xrightarrow{\alpha}$. By abusing notation, we also write $\Rightarrow$ for $\Rightarrow = \bigcup_{\alpha,c} \xrightarrow{\alpha,c}$. As usual we write $\Rightarrow^n$ for the sequential composition of n steps of $\Rightarrow$ and $\Rightarrow^*$ for the reflexive and transitive closure of $\Rightarrow$.

We use the following examples to illustrate the main features of Corr.

Example 2.1 (Simple Correlation). Consider the following system built-up from two different services:

$$\begin{aligned} N = \quad & s^{\{o_1,o_2\}}_{\{x\}}\{o_1(x,y);o_2(x,z);\overline{o}\langle y,z\rangle,0,\emptyset\} \\ & \parallel s'^{O}_{C}\{Q, c \triangleright [\overline{o_1}\langle a,b\rangle;\overline{o_1}\langle d,e\rangle;\overline{o_2}\langle d,f\rangle;\overline{o_2}\langle a,c\rangle], M\} \end{aligned}$$

Service s provides two operations, namely o_1 and o_2, it has no active instances and uses x as the only correlation variable. The two receive actions contained in the definition of s (i.e., o_1 and o_2) use the same correlation property x (x is the first parameter in both input prefixes). Service s' has only one active instance with correlation c —the particular values are uninteresting because all actions are outputs. The instance of s' is sending the request $\overline{o_1}\langle a,b\rangle$. Since o_1 is an operation provided by s, this request will be added to the message bag of s, as shown below.

$$\begin{aligned} N \xrightarrow{\tau}\quad & s^{\{o_1,o_2\}}_{\{x\}}\{o_1(x,y);o_2(x,z);\overline{o}\langle y,z\rangle, 0, \overline{o_1}\langle a,b\rangle\} \\ & \parallel s'^{O}_{C}\{Q, c \triangleright [\overline{o_1}\langle d,e\rangle;\overline{o_2}\langle d,f\rangle;\overline{o_2}\langle a,c\rangle], M\} \end{aligned}$$

At this point, s may consume the message $\overline{o_1}\langle a,b\rangle$ in its message bag to create a new instance (by using rule DISPATCH), as shown below.

$$\begin{aligned} \xrightarrow{\tau}\quad & s^{\{o_1,o_2\}}_{\{x\}}\{o_1(x,y);o_2(x,z);\overline{o}\langle y,z\rangle, \{x\mapsto a\}\triangleright[o_2(x,z);\overline{o}\langle b,z\rangle], \emptyset\} \\ & \parallel s'^{O}_{C}\{Q, c \triangleright [\overline{o_1}\langle d,e\rangle;\overline{o_2}\langle d,f\rangle;\overline{o_2}\langle a,c\rangle], M\} \end{aligned}$$

After two reduction steps s will activate a new instance to handle the request $\overline{o_1}\langle d,e\rangle$. Note that $\{x\mapsto a\}\triangleright[o_2(x,z);\overline{o}\langle b,z\rangle]$ is not able to perform $o_1(d,e)$ and hence the creation of a new instance is the only possibility. Then, the system evolves as follows.

$$\begin{aligned} \xrightarrow{\tau}\xrightarrow{\tau}\quad & s^{\{o_1,o_2\}}_{\{x\}}\{o_1(x,y);o_2(x,z);\overline{o}\langle y,z\rangle,\ \{x\mapsto a\}\triangleright[o_2(x,z);\overline{o}\langle b,z\rangle] \mid \\ & \qquad\qquad \{x\mapsto d\}\triangleright[o_2(x,z);\overline{o}\langle e,z\rangle],\quad \emptyset\} \\ & \parallel s'^{O}_{C}\{Q, c \triangleright [\overline{o_2}\langle d,f\rangle;\overline{o_2}\langle a,c\rangle], M\} \end{aligned}$$

After two communication steps the system reduces to

$$\begin{aligned} \xrightarrow{\tau}\xrightarrow{\tau}\quad & s^{\{o_1,o_2\}}_{\{x\}}\{o_1(x,y);o_2(x,z);\overline{o}\langle y,z\rangle, \{x\mapsto a\}\triangleright[o_2(x,z);\overline{o}\langle b,z\rangle] \mid \\ & \qquad\qquad \{x\mapsto d\}\triangleright[o_2(x,z);\overline{o}\langle e,z\rangle],\ \overline{o_2}\langle d,f\rangle \mid \overline{o_2}\langle a,c\rangle\} \\ & \parallel s'^{O}_{C}\{Q, c \triangleright [0], M\} \end{aligned}$$

Now, the message $\overline{o_2}\langle d,f\rangle$ will be handled by the instance correlated by $\{x\mapsto d\}$, and the message $\overline{o_2}\langle a,c\rangle$ by the instance correlated by $\{x\mapsto a\}$, as below

$$\begin{aligned} \xrightarrow{\tau}\quad & s^{\{o_1,o_2\}}_{\{x\}}\{o_1(x,y);o_2(x,z);\overline{o}\langle y,z\rangle,\ \{x\mapsto a\}\triangleright[o_2(x,z);\overline{o}\langle b,z\rangle] \mid \\ & \qquad\qquad \{x\mapsto d\}\triangleright[\overline{o}\langle e,f\rangle],\quad \overline{o_2}\langle a,c\rangle\} \\ & \parallel s'^{O}_{C}\{Q, c \triangleright [0], M\} \\ \xrightarrow{\tau}\quad & s^{\{o_1,o_2\}}_{\{x\}}\{o_1(x,y);o_2(x,z);\overline{o}\langle y,z\rangle,\ \{x\mapsto a\}\triangleright[\overline{o}\langle b,c\rangle] \mid \\ & \qquad\qquad \{x\mapsto d\}\triangleright[\overline{o}\langle e,f\rangle],\quad \emptyset\} \\ & \parallel s'^{O}_{C}\{Q, c \triangleright [0], M\} \end{aligned}$$

Example 2.2 (Multiple correlations). Orchestration languages provide the possibility of defining multiple correlation sets. This is especially useful for defining services that interact with different partners. These scenarios usually require the usage of one correlation value for each partner. Multiple correlation sets are a built-in feature of Corr as shown by the following example.

$$\begin{aligned} N = \;& s_{\{x,y\}}^{\{o_1,o_2\}}\{P, \{x \mapsto a, y \mapsto b\} \triangleright [(o_1(x,z) \mid o_2(y,w))], \emptyset\} \\ & \parallel s_{1}{}_{C_1}^{O_1}\{P_1, c_1 \triangleright [\overline{o_1}\langle a,d\rangle; R_1], M_1\} \\ & \parallel s_{2}{}_{C_2}^{O_2}\{P_2, c_2 \triangleright [\overline{o_2}\langle b,e\rangle; R_2], M_2\} \end{aligned}$$

The instances of the service s can be identified by using indistinctly x, y or a combination of both of them. In particular, the communication between the instance of s_1 and the instance of s takes place by using operation o_1 and the correlation set x, while the communication with the instance of s_2 will take place over operation o_2 in combination with correlation set y.

Example 2.3 (Colliding instances). In Corr (as in ActiveBPEL), some or all correlation values of two different instances may coincide, i.e., there is not a unique association of correlation values and instances. For example, the following system

$$N = s_{\{x\}}^{\{o_1,o_2\}}\{o_1(x); o_2(x), 0, \emptyset\} \parallel s_{1}{}_{C_1}^{O_1}\{P_1, c_1 \triangleright [\overline{o_1}\langle a\rangle; \overline{o_1}\langle a\rangle], M\}$$

may reduce after two communication steps as below

$$N \xrightarrow{\tau}\xrightarrow{\tau} s_{\{x\}}^{\{o_1,o_2\}}\{o_1(x); o_2(x), 0, \overline{o_1}\langle a\rangle | \overline{o_1}\langle a\rangle\} \parallel s_{1}{}_{C_1}^{O_1}\{P_1, c_1 \triangleright [0], M\}$$

At this time, s may create a new instance with correlation $\{x \mapsto a\}$, as below

$$\xrightarrow{\tau} s_{\{x\}}^{\{o_1,o_2\}}\{o_1(x); o_2(x), \{x \mapsto a\} \triangleright [o_2(x)], \overline{o_1}\langle a\rangle\} \parallel s_{1}{}_{C_1}^{O_1}\{P_1, c_1 \triangleright [0], M\}$$

In the state above, service s has an instance associated with the correlation $\{x \mapsto a\}$ and an available message $\overline{o_1}\langle a\rangle$. Note that the message cannot be handled by the only instance of s, but s may create a new instance because its definition starts with receive o_1. Hence, the system reduces to

$$\xrightarrow{\tau} s_{\{x\}}^{\{o_1,o_2\}}\{o_1(x); o_2(x), \{x \mapsto a\} \triangleright [o_2(x)] | \{x \mapsto a\} \triangleright [o_2(x)], \emptyset\} \parallel s_{1}{}_{C_1}^{O_1}\{P_1, c_1 \triangleright [0], M\}$$

Now s contains two instances with exactly the same correlation values. Assume that s receives a message $\overline{o_2}\langle a\rangle$. Then it evolves as follows

$$\xrightarrow{o_2(a)} s_{\{x\}}^{\{o_1,o_2\}}\{o_1(x); o_2(x), \{x \mapsto a\} \triangleright [o_2(x)] | \{x \mapsto a\} \triangleright [o_2(x)], \overline{o_2}\langle a\rangle\} \parallel \ldots$$

We remark here that the available message $\overline{o_2}\langle a\rangle$ is non-deterministically dispatched to one of the existing instances. It should be noted that correlation mechanism in ActiveBPEL does not ensures univocal identification of a session. Hence, clients of a service cannot rely only on correlation values to identify a particular instance of a service.

Example 2.4 (Exception due to ambiguous receive). Consider the following system

$$N = s_{\{x,y\}}^{\{o_1,o_2\}}\{o_1(x,y); (o_2(x) | o_2(y)), 0, \emptyset\}$$

Then, the following computation is allowed

$$\begin{array}{ll}\xrightarrow{o_1(a,a)} & s^{\{o_1,o_2\}}_{\{x,y\}}\{o_1(x,y);(o_2(x)|o_2(y)),0,\overline{o_1}\langle a,a\rangle\}\\ \xrightarrow{o_2(a)} & s^{\{o_1,o_2\}}_{\{x,y\}}\{o_1(x,y);(o_2(x)|o_2(y)),0,\overline{o_2}\langle a\rangle|\overline{o_1}\langle a,a\rangle\}\\ \xrightarrow{\tau} & s^{\{o_1,o_2\}}_{\{x,y\}}\{o_1(x,y);(o_2(x)|o_2(y)),\{x\mapsto a,y\mapsto a\}\triangleright[o_2(x)\mid o_2(y)],\overline{o_2}\langle a\rangle\}\end{array}$$

At this time, the service s may dispatch the message $\overline{o_2}\langle a\rangle$ to its unique instance, which will raise an exception. Note that the flow of the instance will reduce as follows

$$\text{(AMB-REC-EXCP)}\ \frac{o_2(x)\xmapsto{o_2(a),\{x\mapsto a\}}0 \qquad o_2(y)\xmapsto{o_2(a),\{y\mapsto a\}}0 \qquad \{x\mapsto a\}\neq\{y\mapsto a\}}{o_2(x)|o_2(y)\xrightarrow{o_2(a),\emptyset}\dagger}$$

and hence, the complete system will evolve as shown below

$$\xrightarrow{\tau}\ s^{\{o_1,o_2\}}_{\{x,y\}}\{o_1(x,y);(o_2(x)|o_2(y)),\{x\mapsto a,y\mapsto a\}\triangleright[\dagger],\emptyset\}$$

Example 2.5 (Exception due to conflicting receive). The simplest system exhibiting a conflicting receive can be written as follows

$$N=s^{\{o_1,o_2\}}_{\{x\}}\{o_1(x);(o_2(x)|o_2(x)),0,\emptyset\}$$

Then, N can reduce as follows

$$\xrightarrow{o_1(a)}\ s^{\{o_1,o_2\}}_{\{x,y\}}\{o_1(x);(o_2(x)|o_2(x)),0,\overline{o_1}\langle a\rangle\}$$

Then, the exception is raised when the service attempts to create a new instances for the message $\overline{o_1}\langle a\rangle$. In fact,

$$\text{(CONF-REC-EXCP)}\ \frac{o_1(x);(o_2(x)|o_2(x))\xmapsto{o_1(a),x\mapsto a}o_2(x)|o_2(x) \quad o_2(x)\xmapsto{o_1(a),x\mapsto a}0 \qquad o_2(x)\xmapsto{o_1(a),x\mapsto a}0}{P\xrightarrow{o_1(a),x\mapsto a}\ddagger}$$

Example 2.6 (No exception for undetermined receiver). An invoke that neither matches an existing instance nor creates a new instance will remain blocked instead of raising an exception. For example,

$$\begin{array}{lll}N & = & s^{\{o_1,o_2\}}_{\{x,y\}}\{o_1(x);o_2(x),\{x\mapsto a\}\triangleright[0],\emptyset\}\parallel s1^{O_1}_{C_1}\{P_1,c\triangleright[\overline{o_2}\langle a\rangle],M\}\\ \xrightarrow{\tau} & & s^{\{o_1,o_2\}}_{\{x,y\}}\{o_1(x);o_2(x),\{x\mapsto a\}\triangleright[0],\overline{o_2}\langle a\rangle\}\parallel s1^{O_1}_{C_1}\{P_1,c\triangleright[0],M\}\end{array}$$

Note that N is blocked because the unique instance of s has terminated and the message $\overline{o_2}\langle a\rangle$ cannot create a new instance of the service definition (operation o_2 is not a start activity of the service definition). Although BPEL does not prescribe the behaviour of implementations, there are some engine implementations (like WEBSPHERE) that choose to raise ad hoc exceptions in these cases. For the sake of simplicity, we prefer to keep operational semantics simple and do not include such behaviour. Our choice reflects also the behaviour of ActiveBPEL.

(ZERO)
$$\Gamma \vdash 0 : \emptyset$$

(INPUT)
$$\frac{\Gamma \vdash P : \mathcal{T}}{\Gamma \vdash o(\vec{x});P : \mathcal{T} \oplus \{o \mapsto \vec{x}\}}$$

(SUM)
$$\frac{\Gamma \vdash o_1(\vec{x_1});P_1 : \mathcal{T}_1 \qquad \ldots \qquad \Gamma \vdash o_n(\vec{x_n});P_n : \mathcal{T}_n}{\Gamma \vdash \Sigma_i o_i(\vec{x_i});P_i : \bigoplus_i \mathcal{T}_i}$$

(OUTPUT)
$$\Gamma \vdash \overline{o}\langle \vec{v} \rangle : \emptyset$$

(PAR)
$$\frac{\Gamma \vdash P_1 : \mathcal{T}_1 \qquad \Gamma \vdash P_2 : \mathcal{T}_2 \qquad \mathcal{T}_1 \owedge \mathcal{T}_2}{\Gamma \vdash P_1 | P_2 : \mathcal{T}_1 \oplus \mathcal{T}_2}$$

(SEQ)
$$\frac{\Gamma \vdash P_1 : \mathcal{T}_1 \qquad \Gamma \vdash P_2 : \mathcal{T}_2}{\Gamma \vdash P_1;P_2 : \mathcal{T}_1 \oplus \mathcal{T}_2}$$

(REC)
$$\frac{X \mapsto \mathcal{T}, \Gamma \vdash P : \mathcal{T}}{X \mapsto \mathcal{T}, \Gamma \vdash \mathbf{rec}_X\ P : \mathcal{T}}$$

(X)
$$X \mapsto \mathcal{T}, \Gamma \vdash X : \mathcal{T}$$

(IF)
$$\frac{\Gamma \vdash P_1 : \mathcal{T}_1 \qquad \Gamma \vdash P_2 : \mathcal{T}_2}{\Gamma \vdash \mathbf{if}\ v = v'\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 : \mathcal{T}_1 \oplus \mathcal{T}_2}$$

Fig. 2. Typing rules

3 Correlation exceptions

As illustrated in the examples above, there are situations in which a system may raise an exception. These exceptions are due to the fact that the flow concurrently activates several input actions over the same operation. In cases in which the concurrent actions use exactly the same variables in the same position, then the raised exception is a conflicting-receive. Otherwise, the exception is ambiguous-receive.

Definition 3.1. *A service $s^O_C\{P,0,M\}$ is free from ambiguous-receive exception iff $\forall I$ such that $s^O_C\{P,0,\emptyset\} \Rightarrow^* s^O_C\{P,I,M\}$, then $I \not\equiv c \triangleright [\dagger]|I'$. Similarly, it is free from conflicting-receive when $I \not\equiv c \triangleright [\ddagger]|I'$. We say that the service is free from correlation exceptions when it is free from ambiguous- and conflicting-receive exceptions.*

Following section introduces a simple type system characterizing those services that are free from correlation exceptions.

3.1 Type system for correlation-exception-free services

We will consider the following type judgements for flows: $\Gamma \vdash P : \mathcal{T}$. The type $\mathcal{T}$ assigned to a flow is a partial function from operation names to a set of tuples of variables, i.e., $\mathcal{T} : O \to \mathcal{P}_f(\mathcal{V}^*)$. Basically, the type of a flow P associates any input name occurring in P with a set containing the formal parameters of all its occurrences in P. Moreover, Γ is a partial function from process variables to types, i.e., Γ assigns a type to any process variable in P. For example, $\mathcal{T}(o) = \{\langle x,y\rangle, \langle z,x\rangle\}$ means that all input actions for the operation o have parameters $\langle x,y\rangle$ or $\langle z,x\rangle$. With abuse of notation, we use use $\mathcal{T}$ also to denote the obvious total function defined such that $\mathcal{T}(o) = \emptyset$ when o is not in the domain of the corresponding partial function. We define also type composition as follows $(\mathcal{T}_1 \oplus \mathcal{T}_2)(o) = \mathcal{T}_1(o) \cup \mathcal{T}_2(o)$.

Typing rules are shown in Figure 2. The main idea behind typing rules is that of collecting all tuples used as formal parameters of input actions. Note that rule (INPUT) adds a tuple corresponding to the formal parameters of the input prefix to the type of the continuation of the process. Differently, an output prefix has no effect over the type of a flow (see rule OUTPUT). Rule (PAR) takes into account the compatibility of the types assigned to parallel branches. Note that any input action that takes place in one branch may be concurrently enabled with an input action occurring in the other branch. Hence, type compatibility states sufficient conditions for avoiding exceptions. Our type system is parametric with respect to the definition of the compatibility operation $\circledS$. We will actually consider the following three alternative definitions for compatibility:

$$\mathcal{T}_1 \circledS_c \mathcal{T}_2 = \forall o. \mathcal{T}_1(o) \cap \mathcal{T}_2(o) = \emptyset$$
$$\mathcal{T}_1 \circledS_a \mathcal{T}_2 = \forall o. \#(\mathcal{T}_1(o) \cup \mathcal{T}_2(o)) > 1 \Rightarrow (\mathcal{T}_1(o) = \emptyset \vee \mathcal{T}_2(o) = \emptyset)$$
$$\mathcal{T}_1 \circledS_e \mathcal{T}_2 = \forall o. \mathcal{T}_1(o) = \emptyset \vee \mathcal{T}_2(o) = \emptyset$$

Definition for $\circledS_c$ requires that input actions for a particular operation taking place in different branches use different parameters (actually, we require different tuples, which implies that there is at least one formal parameter that differs). Differently, $\circledS_a$ states that concurrently enabled input actions for a particular operation must use exactly the same parameters. Finally, $\circledS_e$ forbids the concurrent enabling of two or more input actions for the same operation. We will show that each of these definitions can be associated with different notions of correlation exception freeness. Remaining rules are straightforward (note that they do not check compatibility since they do not introduce concurrent enabling of input prefixes).

The following result states that the type of a flow captures the actual parameters of all ready input actions of a flow.

Proposition 3.1. *Let P be a flow. If $\Gamma \vdash P : \mathcal{T}$ and $P \xmapsto{o(v_1,\dots,v_n),c'} P'$, then there exists $\langle y_1,\dots,y_n\rangle \in \mathcal{T}(o)$ s.t. $dom(c') = \langle y_1,\dots,y_n\rangle$.*

Proof (Sketch). The proof follows by induction on derivation $P \xmapsto{o(v_1,\dots,v_n),c'} P'$. When last applied rule is (IN) $P = \Sigma_i o_i(\vec{x}_i); P_i$ and $P' = P_i$ for some i. By typing rules, we know that $\mathcal{T}(o) = \bigoplus_i \mathcal{T}_i(o)$ where $\Gamma \vdash o_i(\vec{x}_i); P_i : \mathcal{T}_i$. Moreover, $\Gamma \vdash o_i(\vec{x}_i); P_i : \mathcal{T}'_i \oplus \{o_i \mapsto \vec{x}_i\}$. Clearly, $\vec{x}_i \in \mathcal{T}(o)$ because $\vec{x}_i \in \mathcal{T}_i(o)$. Other cases follows using inductive hypothesis.

Proposition 3.2. *Let P be a flow. If $\Gamma \vdash P : \mathcal{T}$ and $P \xrightarrow{o(v_1,\dots,v_n),c'} P'$, then there exists $\langle y_1,\dots,y_n\rangle \in \mathcal{T}(o)$ s.t. $dom(c') = \langle y_1,\dots,y_n\rangle$.*

Proof. By analysis of the applied rule for $P \xrightarrow{o(v_1,\dots,v_n),c'} P'$ and Proposition 3.2.

Proposition 3.3 (Subject reduction for $\xmapsto{\alpha,c}$). *Let P be flow. If $\Gamma \vdash P : \mathcal{T}$ and $P \xmapsto{\alpha,c'} P'$, then there exists $\mathcal{T}'$ such that (i) $\Gamma \vdash P' : \mathcal{T}'$ and (ii) $\forall o. \mathcal{T}'(o) \subseteq \mathcal{T}(o)$.*

Proof (Sketch). It follows by induction on the derivation $P \xmapsto{\alpha,c'} P'$. Interesting cases are rules (PAR) and (REC). For (PAR), $P = Q_1 | Q_2$ and $P' = Q'_1 | Q_2$ with $Q_1 \xmapsto{\alpha,c'} Q'_1$. Since P is well-typed, $\Gamma \vdash Q_1 : \mathcal{T}_1$, $\Gamma \vdash Q_2 : \mathcal{T}_2$, $\mathcal{T}_1 \circledS \mathcal{T}_2$ and $\mathcal{T} = \mathcal{T}_1 \oplus \mathcal{T}_2$. Then, by inductive hypothesis we know that there exists $\mathcal{T}'_1$ such that $\Gamma \vdash Q'_1 : \mathcal{T}'_1$ and $\mathcal{T}'_1(o) \subseteq \mathcal{T}_1(o)$ for all o. It is easy to check that for any compatibility operator $\circledS_c$, $\circledS_a$ or $\circledS_e$ the following statement holds: $\mathcal{T}_1 \circledS \mathcal{T}_2$ and $\mathcal{T}'_1(o) \subseteq \mathcal{T}_1(o)$ for all o implies $\mathcal{T}'_1 \circledS \mathcal{T}_2$, hence $\Gamma \vdash Q'_1 | Q_2 : \mathcal{T}'_1 \oplus \mathcal{T}_2$. Moreover, $\mathcal{T}'_1(o) \subseteq \mathcal{T}_1(o)$ implies that $\mathcal{T}'_1(o) \cup \mathcal{T}_2(o) \subseteq \mathcal{T}_1(o) \cup \mathcal{T}_2(o)$, and hence $\mathcal{T}'(o) \subseteq \mathcal{T}(o)$. For (REC) we rely on an auxiliary property stating that $\Gamma \vdash \mathbf{rec}_X\ Q : \mathcal{T}$ implies $\Gamma \vdash Q[^{\mathbf{rec}_X\ Q}/_X] : \mathcal{T}$ and inductive hypothesis.

Following result states that the type of a flow captures all formal parameters of the input operations that a process may execute.

Lemma 3.1. *Let P be a flow. If $\Gamma \vdash P : \mathfrak{T}$ and $c \triangleright [P] \Rightarrow^* c_n \triangleright [P_n] \xrightarrow{o(v_1,\ldots,v_n),c'} c[c'] \triangleright [P']$, then there exists $\langle y_1,\ldots,y_m\rangle \in \mathfrak{T}(o)$ s.t. $dom(c') = \langle y_1,\ldots,y_n\rangle$.*

Proof. The proof follows by induction on the length of the derivation $\Rightarrow^*$.

- **n=0**. This case follows immediately by Proposition 3.2.
- **n=k+1**. $c \triangleright [P] \xrightarrow{\alpha,c_0} c'' \triangleright [P''] \Rightarrow^k c_{k+1} \triangleright [P_{k+1}] \xrightarrow{o(v_1,\ldots,v_n),c_{k+1}} c' \triangleright [P']$. We proceed by case analysis on the structure of P. When $P = \Sigma_i o_i(\vec{x_i});P_i$, $c \triangleright [P] \xrightarrow{o_i(\vec{v}),\vec{x_i}\mapsto\vec{v}} c[\vec{x_i} \mapsto \vec{v}] \triangleright [P_i']$. By Proposition 3.3 we know that there exists $\mathfrak{T}_1$ such that $\Gamma \vdash P_i' : \mathfrak{T}_1$ and $\forall o.\mathfrak{T}_1(o) \subseteq \mathfrak{T}(o)$. By inductive hypothesis (applied over $c'' \triangleright [P''] \Rightarrow^k c_{k+1} \triangleright [P_{k+1}] \xrightarrow{o(v_1,\ldots,v_n),c_{k+1}} c' \triangleright [P']$) we know that there exist $\langle y_1,\ldots,y_m\rangle \in \mathfrak{T}_1(o)$ s.t. $dom(c') = \langle y_1,\ldots,y_n\rangle$. Since, $\forall o.\mathfrak{T}_1(o) \subseteq \mathfrak{T}(o)$ we conclude that $\langle y_1,\ldots,y_m\rangle \in \mathfrak{T}_1(o)$. Remaining cases follow analogously.

Lemma 3.2 (Subject reduction). *Let P be a flow and c a correlation instance. If $\Gamma \vdash P : \mathfrak{T}$ and $P \xrightarrow{\alpha,c'} P'$, then one of the following holds*

1. *there exists $\mathfrak{T}'$ such that $\Gamma \vdash P' : \mathfrak{T}'$ and $\forall o.\mathfrak{T}'(o) \subseteq \mathfrak{T}(o)$,*
2. *if $P' = \dagger$ then compatibility operator is $\circledS_c$, or*
3. *if $P' = \ddagger$ then compatibility operator is $\circledS_a$.*

Proof (Sketch). The proof follows by analysis of the rule applied for $P \xrightarrow{\alpha,c'} P'$. For rule (NO-EXCP) we show that 1 holds by using Proposition 3.2. For rule (AMB-REC-EXCP) there are two cases. (i) When compatibility is $\circledS_c$ then $P = \dagger$ and 2 holds. (ii) For $\circledS_e$ and $\circledS_a$ we show by contradiction that this rule cannot be applied. Since $P = Q_1 | Q_2$ is well-typed, $\Gamma \vdash Q_1 : \mathfrak{T}_1$, $\Gamma \vdash Q_2 : \mathfrak{T}_2$ and $\mathfrak{T}_1 \circledS \mathfrak{T}_2$. Moreover, $Q_1 \xrightarrow{o(\vec{v}),c_1} Q_1$ and $Q_2 \xrightarrow{o(\vec{v}),c_2} Q_2$ with $c_1 \neq c_2$. For Proposition 3.2, $dom(c_1) \in \mathfrak{T}_1$ and $dom(c_2) \in \mathfrak{T}_2$. It is easy to check that this implies that neither $\mathfrak{T}_1 \circledS_a \mathfrak{T}_2$ nor $\mathfrak{T}_1 \circledS_e \mathfrak{T}_2$ holds, which contradicts the fact that P is well-typed. For rule (AMB-REC-EXCP) we proceed as in the previous case.

Next theorem states the main result of the paper, saying that services with well-typed definitions do are free from correlation exceptions.

Theorem 3.1. *Let P be a flow s.t. $\Gamma \vdash P : \mathfrak{T}$, then the well-formed service $s_C^O\{P, 0, \emptyset\}$ is*

1. *free from ambiguous-receive exception if type compatibility is taken as $\circledS_a$.*
2. *free from conflict-receive exception if type compatibility is taken as $\circledS_c$.*
3. *free from correlation exception if type compatibility is taken as $\circledS_e$.*

Proof. 1. We prove by induction on the length of the derivation $\Rightarrow^n$ that $s_C^O\{P,0,\emptyset\} \Rightarrow^n s_C^O\{P,I,M\}$ implies (i) $I \not\equiv c \triangleright [\dagger] | I'$ and $I \equiv c \triangleright [Q] | I'$ implies Q is well-typed. Case **n=0** is immediate since $I \equiv 0$. For **n=k+1**, $s_C^O\{P,0,\emptyset\} \Rightarrow^k s_C^O\{P,I_k,M_k\} \Rightarrow s_C^O\{P,I,M\}$. By inductive hypothesis, $I_k \not\equiv c \triangleright [\dagger] | I_k'$. We show that exception cannot be raised in the last step by case analysis on the reduction $s_C^O\{P,I_k,M_k\} \Rightarrow s_C^O\{P,I,M\}$.

- $s_C^O\{P,I_k,M_k\} \xrightarrow{\tau} s_C^O\{P,I,M\}$. There are three possibilities. For rule (DISPATCH) it should be $I' \equiv c \triangleright [Q] | I_2 \xrightarrow{o_1(v_1),c'} I = c' \triangleright [Q'] | I_2$ with $Q \xrightarrow{o_1(v_1),c'} \dagger$, but this contradicts Lemma 3.2. Hence, $Q' \neq \dagger$ and Q is well-typed. Rule (NEW) follows analogously. Rule (SVC-NON-IN) follows immediately since τ actions in flow do not introduce $\dagger$.
- $s_C^O\{P,I_k,M_k\} \xrightarrow{\bar{o}\langle v\rangle} s_C^O\{P,I,M\}$ and $s_C^O\{P,I_k,M_k\} \xrightarrow{o(v)} s_C^O\{P,I,M\}$. These cases follow by noticing that these reductions do not introduce exceptions.

Cases 2. and 3. follow analogously.

4 Related works and concluding remarks

This paper introduces Corr, which is a process calculus with correlation primitives. The formal definition of the correlation mechanism exhibited by Corr has been greatly inspired by SOCK [6]. However, we do not include two-way operations, state manipulation and assignment to keep the language simple. We are convinced that the proposed approach smoothly extends to a calculus containing those features (this is left as a future work). We remark that the semantics of SOCK blocks computations that generate correlation instances that collide, while Corr does not impose restrictions among different instances. Differently from SOCK, Corr has a mechanism that automatically raises exceptions when instances activate receive actions that may handle the same request non-deterministically. Although some extensions of SOCK (like [5]) provide primitives for exception handling, exceptions in those approaches are thrown by the execution of a particular primitive but not as a consequence of some correlation violation.

BLITE [9] is a process calculus aimed at explaining most of BPEL features. As a consequence, it contains several primitives that are not included in Corr. With respect to the subset of BLITE that corresponds to Corr we remark that: (i) a BLITE service chooses the receiver of a message by using “the most specific instance principle”, i.e., if several instances can handle an incoming message, then the message is directed to the instance associated with the correlation instance that have more initialized values matching the incoming message. If there are several ones, then BLITE chooses non-deterministically one of them. On the contrary, Corr does not have control over different instances and raises an exception when non-determinism is internal to an instance.

Corr does not model explicitly partner links (as done in BLITE and SOCK) because the correlation mechanism is usually used in combination with static endpoints. Consequently, if partner links cannot change dynamically, we see no reason for including them into the model (the study of correlation mechanism in combination with dynamic endpoint identification is out of the scope of this paper).

COWS [9] relies on a pattern matching mechanism to deal with correlation sets. In this sense, COWS describes correlations at a lower level of abstraction — although it has been shown in [10] that it is expressive enough for encoding the correlation mechanism of BLITE.

As already mentioned, the distinctive feature of Corr when compared against previous proposals like SOCK, COWS, and BLITE is that Corr accounts for exceptions raised as a consequence of incorrect usage of correlation sets. None of the previous proposals accounts for the interaction between correlations and exceptions.

We also mention that a completely different approach for structuring interactions among service instances is related to the concept of sessions [2,8,1,7]. Sessions are a more abstract way of thinking about service interaction, which facilitates the analysis of the interaction between instances.

Acknowledgements The authors thank the anonymous reviewers for their valuable comments and suggestions.

References

1. Eduardo Bonelli and Adriana Compagnoni. Multisession session types for a distributed calculus. In *Proc. of Trustworthy Global Computing 2007*, volume 4912 of *Lecture Notes in Computer Science*, pages 240–256. Springer, 2007.
2. M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos, and G. Zavattaro. SCC: a service centered calculus. In *Proc. of WS-FM 2006*, volume 4184 of *Lect. Notes in Comput. Sci.*, pages 38–57. Springer, 2006.
3. Web services addressing (ws-addressing). Available at `http://www.w3.org/Submission/ws-addressing/`, August 2004.
4. Web services Business Process Execution L anguage (BPEL). version 2.0. Available at `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`, April 2007.
5. C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro. On the interplay between fault handling and request-response service invocations. In *Proceedings of 8th International Conference on Application of Concurrency to System Design (ACSD 2008)*, pages 190–198. IEEE, 2008.
6. C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A calculus for service oriented computing. In *Service-Oriented Computing - ICSOC 2006*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2006.
7. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *Proc. of ESOP'98*, volume 1381 of *Lect. Notes in Comput. Sci.*, pages 22–138. Springer, 1998.
8. I. Lanese, V.T. Vasconcelos, F. Martins, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *Proc. of SEFM'07*, pages 305–314. IEEE Computer Society Press, 2007.
9. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *Proc. of ESOP'07*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 33–47. Springer, 2007.
10. A. Lapadula, R. Pugliese, and F. Tiezzi. A formal account of ws-bpel. In *COORDINATION*, volume 5052 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 2008.
11. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lect. Notes in Comput. Sci.* Springer, 1980.
12. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981.
13. M. Viroli. A core calculus for correlation in orchestration languages. *J. Log. Algebr. Program.*, 70(1):74–95, 2007.