An Extended cCSP with Stable Failures Semantics

Zhenbang Chen[1], Zhiming Liu[2]

[1] National Laboratory for Parallel and Distributed Processing,
National University of Defense Technology, Changsha, China
[2] International Institute for Software Technology
United Nations University, Macao SAR, China
`zbchen@nudt.edu.cn`, `Z.Liu@iist.unu.edu`

Abstract. Compensating CSP (cCSP) is an extension to CSP for modeling long-running transactions. It can be used to specify programs of service orchestration written in a programming language like WS-BPEL. So far, only an operational semantics and a trace semantics are given to cCSP. In this paper, we extend cCSP with more operators and define for it a stable failures semantics in order to reason about non-determinism and deadlock. We give some important algebraic laws for the new operators. These laws can be justified and understood from the stable failures semantics. A case study is given to demonstrate the extended cCSP.

1 Introduction

Long-Running Transactions (LRT) are attracting increasing research attention recently because of their importance in Service-Oriented Computing (SOC) [10]. A transaction in SOC usually lasts for a long period of time, and involves interactions with different organizations. The notion of atomic transaction is too strict for this scenario due to some requirements such as isolation [10]. LRT are therefore introduced to cope with this problem by using compensation to recover from a failure to ensure the required atomicity and consistency.

Industrial service composition languages, such as WS-BPEL [1] and XLANG [14] are now designed and implemented for programming LRT in service orchestration. For specification and verification of LRT, formalisms have been being proposed and they include StAC [4], Sagas [3], cCSP [6], *etc.* Formalisms can provide formal semantics to an industrial language and serve as the foundation for the understanding of LRT and the development of tool support to verification and analysis.

Compensating CSP (cCSP) extends the process calculus of Communicating Sequential Process (CSP) [13] with mechanisms of interruption and recovery from exceptions for describing LRT. The recovery mechanism in cCSP is the same as the backward recovery proposed in Sagas [9]. There are two types of processes in cCSP, and they are called respectively *standard processes* and *compensable processes*. A standard process is a subset of a CSP process extended with exception handling and transaction block. A compensable process specifies the behavior of the recovery when an exception occurs. A trace semantics is presented in [4] and an operational semantics is in [7], and the consistency between them is studied in [12]. However, without non-deterministic (internal) choice and hiding, cCSP is not expressive enough for relating specifications at different levels of abstraction. It is important to note that abstraction (via hiding) is the main source of non-determinism and non-determinism can causes deadlocks when composing processes.

Internal choice and hiding are motivated in the definition of StAC [4, 5], a formal notation for LRT that supports synchronized parallel composition, internal and external choices, hiding, and programmable compensation. A serious drawback of StAC compared to cCSP is that StAC does not support compositional reasoning. A thorough comparative study between Sagas [3] and cCSP is presented in [2] and shows two equivalent subsets of them. The paper also compares the policies of the interruption and compensation in Sagas and cCSP, and finds that the revised Sagas [3] is more expressive than cCSP [6]. There is an attempt to extend cCSP [11], but only with synchronized parallel composition.

In this paper, we extend cCSP by bringing back the CSP operators of hiding, internal choice for non-determinism, and synchronized parallel composition for general composition. Accordingly to characterize non-determinism and deadlock, we define a stable failures semantics for the extended language. We show most algebraic laws in the trace semantics of the original cCSP still hold in the stable failures semantics. Also, we show that a few laws that were claimed to hold for the original trace semantics do not hold there, but they hold for the semantics we define in this paper. We study the laws for the newly introduced operators. Due to the page limit, the proofs of the laws are omitted, but they can be found in a technical report [8].

The rest of this paper is organized as follows. Section 2 gives a brief introduction to the syntax and semantics of the original cCSP. Section 3 presents the extended cCSP including the syntax, semantics and laws. Section 4 gives a case study to demonstrate the extended cCSP. Section 5 concludes the paper and reviews the related work.

2 Compensating CSP

The syntax of cCSP [6] is as follows, where P and PP represent a *standard process* and a *compensable process*, respectively.

$$\begin{aligned} P &::= A \mid P \,;\, P \mid P \Box P \mid P \parallel P \mid \mathit{SKIP} \mid \mathit{THROW} \mid \mathit{YIELD} \mid P \rhd P \mid [PP] \\ PP &::= P \div P \mid PP \,;\, PP \mid PP \Box PP \mid PP \parallel PP \mid \mathit{SKIPP} \mid \mathit{THROWW} \mid \mathit{YIELDD} \end{aligned}$$

Process A denotes the process that terminates successfully after performing event A. There are three operators on both the standard processes and the compensable processes: sequential composition (;), deterministic choice ($\Box$) and parallel composition ($\parallel$). *SKIP* is the process that immediately terminates successfully. *THROW* indicates the occurrence of an exception, and the process will be interrupted. *YIELD* can terminate successfully or yield to an interrupt from environment to result in an interruption. $P \rhd Q$ executes process Q after an exception is thrown from P, otherwise it behaves like P. $[PP]$ is a transaction block specifying a long-running transaction, in which a compensable process is defined to specify the transaction.

A compensable process is constructed from *compensation pairs* of the form $P \div Q$, where the execution of process Q can compensate the effects after executing P. *SKIPP* immediately terminates successfully without the need to be compensated. *THROWW* throws an exception and *YIELDD* either terminates successfully or yields to an interrupt. We use $\mathcal{P}$ and $\mathcal{PP}$ to denote the set of standard processes and the set of compensable processes, respectively.

2.1 Basic notations

Let Σ be the set of all the *normal events* that all processes can perform, called *alphabet* of processes, and Σ^* be the set of the finite traces over Σ. In cCSP, three more events $\checkmark$, ! and ? not in Σ are used. Event $\checkmark$, called the *success terminal event*, represents that the process terminates successfully. Event !, called the *exception terminal event*, represents that the trace terminates with an occurrence of an exception. Event ?, called the *yield terminal event*, represents that the execution terminates by yielding to an interrupt from environment. We use $\Omega = \{\checkmark, !, ?\}$ to denote the set of the terminal events, and define $\Sigma^{\Omega} = \Sigma \cup \Omega$. In addition, we use $s\hat{\ }t$ to represent the *concatenation* of traces s and t, and define

- $\Sigma^*_O = \{s\hat{\ }\langle\omega\rangle \mid s \in \Sigma^* \wedge \omega \in O\}$: for an $O \subseteq \Omega$.

Let $\Sigma^{*O} = \Sigma^* \cup \Sigma^*_O$, and we call traces in Σ^*_Ω *terminating traces* and traces in $\Sigma^*_{\{\checkmark\}}$ *successfully terminating traces.*

2.2 Trace semantics

In contract to the CSP convention [13], the trace set of a standard process in cCSP is not prefix closed. The *trace semantic function* $\mathcal{T} : \mathcal{P} \to \mathbb{P}(\Sigma^*_\Omega)$ assigns

Atomic process For all $A \in \Sigma$, $\mathcal{T}(A) = \{\langle A, \checkmark\rangle\}$

Sequential composition

$$p \ ;\ q = \begin{cases} p_1\hat{\ }q & p = p_1\hat{\ }\langle\checkmark\rangle \\ p & p = p_1\hat{\ }\langle\omega\rangle \wedge \omega \neq \checkmark \end{cases} \qquad \mathcal{T}(P \ ;\ Q) = \{p \ ;\ q \mid p \in \mathcal{T}(P) \wedge q \in \mathcal{T}(Q)\}$$

Choice $\mathcal{T}(P \Box Q) = \mathcal{T}(P) \cup \mathcal{T}(Q)$

Parallel composition

$$p_1\hat{\ }\langle\omega_1\rangle \parallel q_1\hat{\ }\langle\omega_2\rangle = \{r\hat{\ }\langle\omega_1 \& \omega_2\rangle \mid r \in (p_1 \mid\mid\mid q_1)\}$$
$$\mathcal{T}(P \parallel Q) = \{r \mid r \in (p \parallel q) \wedge p \in \mathcal{T}(P) \wedge q \in \mathcal{T}(Q)\}$$

where

$$\begin{array}{r|cccccc} \omega_1 & \checkmark & \checkmark & \checkmark & ! & ! & ? \\ \omega_1 & \checkmark & ? & ! & ! & ? & ? \\ \hline \omega_1 \& \omega_2 & \checkmark & ? & ! & ! & ! & ? \end{array}$$

Exception handling

$$p \rhd q = \begin{cases} p_1\hat{\ }q & p = p_1\hat{\ }\langle !\rangle \\ p & p = p_1\hat{\ }\langle\omega\rangle \wedge \omega \neq ! \end{cases} \qquad \mathcal{T}(P \rhd Q) = \{p \rhd q \mid p \in \mathcal{T}(P) \wedge q \in \mathcal{T}(Q)\}$$

Basic processes $\mathcal{T}(\mathit{SKIP}) = \{\langle\checkmark\rangle\}$, $\mathcal{T}(\mathit{THROW}) = \{\langle !\rangle\}$, $\mathcal{T}(\mathit{YIELD}) = \{\langle ?\rangle, \langle\checkmark\rangle\}$

Fig. 1: The semantics of standard process

each process P a set $\mathcal{T}(P)$ of terminating traces. Fig. 1 shows the definition of $\mathcal{T}$, where p and q are terminating traces, and $p_1 \,|||\, q_1$ represents the set of interleavings of traces p_1 and q_1, whose formal definition can be refereed to Section 2.3 in [13]. The processes in a parallel composition only synchronize on the terminal events, performing other events in an interleaving manner. An exception occurs in the composition if any sub-process throws an exception, and the composition terminates successfully only if both sub-processes do.

A compensable process is defined by a set of pairs of traces, called the *forward trace* and *compensation trace*, respectively. It is necessary to note that the semantics of the sequential composition conforms to the semantics of the classical Sagas [9], and compensation actions are executed in the reverse order of their corresponding forward actions. For example, the forward behavior of $A_1 \div B_1 \ ; \ A_2 \div B_2$ will perform A_1 followed by A_2, but the compensation behavior will perform B_1 after B_2 in case of an exception occurred later. Fig. 2 defines the *trace semantic function* $\mathcal{T}_c : \mathcal{PP} \to \mathbb{P}(\Sigma_\Omega^* \times \Sigma_\Omega^*)$ of the compensable processes.

Compensation pair

$$p \div q = \begin{cases} (p, q) & p = p_1 \,\hat{}\, \langle \checkmark \rangle \\ (p, \checkmark) & p = p_1 \,\hat{}\, \langle \omega \rangle \wedge \omega \neq \checkmark \end{cases}$$

$$\mathcal{T}_c(P \div Q) = \{p \div q \mid p \in \mathcal{T}(P) \wedge q \in \mathcal{T}(Q)\} \cup \{(\langle ? \rangle, \langle \checkmark \rangle)\}$$

Compensable sequential composition

$$(p, p') \ ; \ (q, q') = \begin{cases} (p_1 \,\hat{}\, q, q' \ ; \ p') & p = p_1 \,\hat{}\, \langle \checkmark \rangle \\ (p, p') & p = p_1 \,\hat{}\, \langle \omega \rangle \wedge \omega \neq \checkmark \end{cases}$$

$$\mathcal{T}_c(PP \ ; \ QQ) = \{(p, p') \ ; \ (q, q') \mid (p, p') \in \mathcal{T}_c(PP) \wedge (q, q') \in \mathcal{T}_c(QQ)\}$$

Compensable choice $\mathcal{T}_c(PP \Box QQ) = \mathcal{T}_c(PP) \cup \mathcal{T}_c(QQ)$

Compensable parallel composition

$$(p, p') \parallel (q, q') = \{(r, r') \mid r \in (p \parallel q) \wedge r' \in (q \parallel q')\}$$

$$\mathcal{T}_c(PP \parallel QQ) = \{rr \mid rr \in (pp \parallel qq) \wedge pp \in \mathcal{T}_c(PP) \wedge qq \in \mathcal{T}_c(QQ)\}$$

Compensable basic processes

$SKIP = SKIP \div SKIP$, $THROWW = THROW \div SKIP$, $YIELDD = YIELD \div SKIP$

Fig. 2: The semantics of compensable process

To allow a compensable process PP to implicitly yield to an interrupt from the environment at the beginning, in the definition of a compensation pair $P \div Q$ in Fig. 2, the trace pair $(\langle ? \rangle, \langle \checkmark \rangle)$ is included. On the other hand, *YIELD* can be used in any place in a process if one would like to explicitly specify a yield to an interrupt at that place. The semantics of a transaction block $[PP]$ is defined below.

$$\mathcal{T}([PP]) = \{p \,\hat{}\, p' \mid (p \,\hat{}\, \langle ! \rangle, p') \in \mathcal{T}_c(PP)\} \cup \{p \,\hat{}\, \langle \checkmark \rangle \mid (p \,\hat{}\, \langle \checkmark \rangle, p') \in \mathcal{T}_c(PP)\}$$

It says that after an exception occurs, the compensation trace will be executed to recover from the failure. Otherwise, the compensation trace is not executed.

Discussion In paper [6] some laws are given for the trace semantics. However, our careful investigation finds that some of them do not actually hold there. The first is $PP; SKIPP = PP$. For example, according to the semantic definition in Fig. 2, the semantics of the process $A \div B$ is $\{(\langle A, \checkmark\rangle, \langle B, \checkmark\rangle), (\langle ?\rangle, \langle \checkmark\rangle)\}$, but the semantics of $A \div B \ ; \ \mathit{SKIPP}$ is $\{(\langle A, \checkmark\rangle, \langle B, \checkmark\rangle), (\langle ?\rangle, \langle \checkmark\rangle), (\langle A, ?\rangle, \langle B, \checkmark\rangle)\}$. The law is not valid because of the extra trace pair $(\langle ?\rangle, \langle \checkmark\rangle)$ added to a compensation pair in the semantic definition.The laws $[P \div Q]{=}P$ and $[P \div Q; \mathit{THROWW}]{=}P; Q$ do not hold either when P does not terminate with the yield terminal event ?. It is because the transaction block will remove the exception terminal event ! of the forward trace. Intuitively, we expect these laws to hold. Indeed, we will see later they become valid in the stable failures semantics in this paper.

3 Extended cCSP and its stable failures semantics

We extend cCSP with operators of internal and external choices, hiding, renaming and generalized parallel composition for both the standard and compensable processes. The syntax of the extended cCSP is defined as follows, where $A \in \Sigma$, $X \subseteq \Sigma$, and $R \subseteq \Sigma \times \Sigma$.

$$\begin{aligned} P ::= {} & A \mid P \ ; \ P \mid P \sqcap P \mid P \Box P \mid P \underset{X}{\parallel} P \mid \mathit{SKIP} \mid \mathit{THROW} \mid \mathit{YIELD} \mid \\ & \mathit{STOP} \mid P \setminus X \mid P[\![R]\!] \mid P \rhd P \mid [PP] \\ PP ::= {} & P \div P \mid PP \ ; \ PP \mid PP \sqcap PP \mid PP \Box PP \mid PP \underset{X}{\parallel} PP \mid \mathit{SKIPP} \mid \\ & \mathit{THROWW} \mid \mathit{YIELDD} \mid PP \setminus X \mid PP[\![R]\!] \end{aligned}$$

$P \sqcap Q$ and $P \Box Q$ represent internal and external choices, respectively. In the generalized parallel composition $P \underset{X}{\parallel} Q$, processes P and Q synchronize on the events in X, as well as on the terminal events in Ω. $P \setminus X$ is the process with the events in X being restricted from happening during the execution of P, and $P[\![R]\!]$ the process obtained from P by renaming its events according to the renaming relation R.

We extend the compensable processes similarly by introducing the same operators. The internal and external choices in the compensable processes are made during the execution of the forward behaviors of the sub-processes. PP and QQ in $PP \underset{X}{\parallel} QQ$ synchronize on the events in X between both the forward behaviors and the compensation behaviors of the two sub-processes.

3.1 Semantics of standard process

The semantics of a standard process is slightly different from the stable failures semantics of a CSP process in [13], due to the two new terminal events ! and ?. The stable failures model of a standard process P is a pair (T, F), where $T \subseteq \Sigma^{*\Omega}$ is the *trace set* and $F \subseteq \Sigma^{*\Omega} \times \mathbb{P}(\Sigma^{\Omega})$ is the *stable failure set*. The domain of the pairs of traces and failues should satisfy the following axioms.

$$T \text{ is non-empty and prefix closed} \tag{1}$$

$$(s, X) \in F \Rightarrow s \in T \tag{2}$$

$$(s, X) \in F \wedge Y \subseteq X \Rightarrow (s, Y) \in F \tag{3}$$

$$(s, X) \in F \wedge \forall a \in Y \bullet s\,\hat{}\,\langle a\rangle \notin T \Rightarrow (s, X \cup Y) \in F \tag{4}$$

$$s\,\hat{}\,\langle \omega\rangle \in T \Rightarrow (s, \Sigma^{\Omega} \setminus \{\omega\}) \in F, \ \mathit{where} \ \omega \in \Omega \tag{5}$$

$$s\,\hat{}\,\langle \omega\rangle \in T \Rightarrow (s\,\hat{}\,\langle \omega\rangle, X) \in F, \ \mathit{where} \ \omega \in \Omega \wedge X \subseteq \Sigma^{\Omega} \tag{6}$$

In what follows we define the *trace set function* $\mathcal{T_S} : \mathcal{P} \to \mathbb{P}(\Sigma^{*\Omega})$ and the *stable failure set function* $\mathcal{F_S} : \mathcal{P} \to \mathbb{P}(\Sigma^{*\Omega} \times \mathbb{P}(\Sigma^{\Omega}))$ for the standard processes in the extended cCSP.

Atomic and basic processes Process A can perform event A and terminate successfully. The semantic functions are as follows.

$$\begin{aligned}
&\mathcal{T_S}(A) = \{\langle\rangle, \langle A\rangle, \langle A, \checkmark\rangle\}\\
&\mathcal{F_S}(A) = \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\Omega} \wedge A \notin X\} \cup \{(\langle A\rangle, X) \mid X \subseteq \Sigma^{\Omega} \wedge \checkmark \notin X\} \cup\\
&\qquad\qquad \{(\langle A, \checkmark\rangle, X) \mid X \subseteq \Sigma^{\Omega}\}
\end{aligned}$$

The trace and failure sets of processes *SKIP*, *THROW*, *YIELD* and *STOP* are defined below.

$$\begin{aligned}
&\mathcal{T_S}(\mathit{SKIP}) &&= \{\langle\rangle, \langle\checkmark\rangle\} \qquad \mathcal{T_S}(\mathit{THROW}) = \{\langle\rangle, \langle !\rangle\}\\
&\mathcal{T_S}(\mathit{YIELD}) &&= \{\langle\rangle, \langle\checkmark\rangle, \langle ?\rangle\} \qquad \mathcal{T_S}(\mathit{STOP}) = \{\langle\rangle\}\\
&\mathcal{F_S}(\mathit{SKIP}) &&= \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\Omega} \wedge \checkmark \notin X\} \cup \{(\langle\checkmark\rangle, X) \mid X \subseteq \Sigma^{\Omega}\}\\
&\mathcal{F_S}(\mathit{THROW}) &&= \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\Omega} \wedge\, ! \notin X\} \cup \{(\langle !\rangle, X) \mid X \subseteq \Sigma^{\Omega}\}\\
&\mathcal{F_S}(\mathit{YIELD}) &&= \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\Omega} \wedge\, ? \notin X\} \cup \{(\langle ?\rangle, X) \mid X \subseteq \Sigma^{\Omega}\} \cup\\
&&&\quad\ \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\Omega} \wedge \checkmark \notin X\} \cup \{(\langle\checkmark\rangle, X) \mid X \subseteq \Sigma^{\Omega}\}\\
&\mathcal{F_S}(\mathit{STOP}) &&= \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\Omega}\}
\end{aligned}$$

Internal choice $P \sqcap Q$ can refuse an event set after performing a trace s if P or Q can refuse the event set after s. The semantic of internal choice is same as that in [13], i.e. the traces and failures of an internal choice are the unions of the traces and failures of its sub-processes, respectively.

$$\mathcal{T_S}(P \sqcap Q) = \mathcal{T_S}(P) \cup \mathcal{T_S}(Q) \qquad \mathcal{F_S}(P \sqcap Q) = \mathcal{F_S}(P) \cup \mathcal{F_S}(Q)$$

It is straightforward to see that $\mathit{YIELD} \sqcap \mathit{SKIP} = \mathit{YIELD}$.

External choice External choice is different from internal choice on the empty trace ($\langle\rangle$), at which $P \Box Q$ can refuse an event set only if both P and Q refuse it. The failure set of external choice needs to take the terminal events ? and ! into account to make axiom (1) on page 5 hold.

$$\begin{aligned}
&\mathcal{T_S}(P \Box Q) = \mathcal{T_S}(P) \cup \mathcal{T_S}(Q)\\
&\mathcal{F_S}(P \Box Q) = \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathcal{F_S}(P) \cap \mathcal{F_S}(Q)\} \cup\\
&\qquad\qquad \{(s, X) \mid (s, X) \in \mathcal{F_S}(P) \cup \mathcal{F_S}(Q) \wedge s \neq \langle\rangle\} \cup\\
&\qquad\qquad \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\Omega} \setminus \{\omega\} \wedge \langle\omega\rangle \in \mathcal{T_S}(P) \cup \mathcal{T_S}(Q) \wedge \omega \in \Omega\}
\end{aligned}$$

The internal and external choices are indistinguishable on the basic processes.

$$\mathit{SKIP} \Box \mathit{YIELD} = \mathit{YIELD} \quad \mathit{SKIP} \Box \mathit{THROW} = \mathit{SKIP} \sqcap \mathit{THROW}$$
$$\mathit{YIELD} \Box \mathit{THROW} = \mathit{YIELD} \sqcap \mathit{THROW}$$

Sequential composition The definition of sequential composition is different from the classic CSP [13] due to the terminal events ! and ?.

$$\begin{aligned}
&\mathcal{T_S}(P\ ;\ Q) = \{s \mid s \in \mathcal{T_S}(P) \cap \Sigma^{*\{!,?\}}\} \cup \{s\hat{\ }t \mid s\hat{\ }\langle\checkmark\rangle \in \mathcal{T_S}(P) \wedge t \in \mathcal{T_S}(Q)\}\\
&\mathcal{F_S}(P\ ;\ Q) = \{(s, X) \mid s \in \Sigma^{*\{!,?\}} \wedge (s, X \cup \{\checkmark\}) \in \mathcal{F_S}(P)\} \cup\\
&\qquad\qquad \{(s\hat{\ }t, X) \mid s\hat{\ }\langle\checkmark\rangle \in \mathcal{T_S}(P) \wedge (t, X) \in \mathcal{F_S}(Q)\}
\end{aligned}$$

However, the following two laws in [6] still hold here.

$$\mathit{THROW}\ ;\ P = \mathit{THROW} \qquad \mathit{YIELD}\ ;\ \mathit{YIELD} = \mathit{YIELD}$$

The first law ensures the *exception-stop* semantics that is adopted in many modern languages.

Parallel composition The parallel composition has to take care of the synchronization of the terminal events. We use $s \underset{X}{\parallel} t$ to represent the trace set of the synchronization between two traces s and t on X. As well as on the terminal events, s and t need to synchronize on the events in X. The definition of $s \underset{X}{\parallel} t$ can be referred to that in classical CSP [13] except the synchronization between terminal events, which uses the definition in cCSP (cf. $\omega_1 \& \omega_2$ in Fig. 1).

To define the semantics of $P \underset{X}{\parallel} Q$, we first define its trace set, and then its failure set. The trace set of $P \underset{X}{\parallel} Q$ is as follows based on the cases defined above.

$$\mathcal{T_S}(P \underset{X}{\parallel} Q) = \{u \mid \exists s \in \mathcal{T_S}(P), t \in \mathcal{T_S}(Q) \bullet u \in s \underset{X}{\parallel} t\} \tag{7}$$

$P \underset{X}{\parallel} Q$ can refuse an event in $X \cup \Omega$ if either P or Q can. However, because both P and Q can perform the events outside $X \cup \Omega$ independently, $P \underset{X}{\parallel} Q$ refuses an event outside $X \cup \Omega$ only if both P and Q refuse it. For a failure (s, Y) in P and (t, Z) in Q, the following set is their synchronized failure set under the classical CSP definition.

$$(s, Y) \oplus (t, Z) = \{(u, Y \cup Z) \mid Y \setminus (X \cup \Omega) = Z \setminus (X \cup \Omega) \wedge u \in s \underset{X}{\parallel} t\} \tag{8}$$

However, this definition has to be modified for the extended cCSP, to take into account the different cases of synchronization on the terminal events in Ω.

- If P or Q cannot perform a terminal event after executing s or t, then $P \underset{X}{\parallel} Q$ cannot terminate because P and Q need to synchronize on the terminal events. We can use the definition (8) for this case. For example, if Σ is $\{A, B\}$, consider processes A and $B\ ;\ \mathit{THROW}$. We have the failure $(\langle\rangle, \{B, \checkmark, !, ?\})$ of A and the failure $(\langle B\rangle, \{B, \checkmark, ?\})$ of $B\ ;\ \mathit{THROW}$, and thus the failure $(\langle B\rangle, \{B, \checkmark, !, ?\})$ of $A \underset{\{\}}{\parallel} (B\ ;\ \mathit{THROW})$.
- If both P and Q can terminate, the synchronized terminal event should be removed from the refusal set of the synchronized failure. For example, if Σ is $\{A\}$, consider processes A and $A; \mathit{THROW}$. A has the failure $(\langle A\rangle, \{A, !, ?\})$, and $A\ ;\ \mathit{THROW}$ has the failure $(\langle A\rangle, \{A, \checkmark, ?\})$. We can see that $\checkmark$ is the terminal event A can perform, and $!$ is the terminal event $A\ ;\ \mathit{THROW}$ can perform. The synchronization of these two terminal events is $!$, which should not be contained in the refusal set of the synchronized failure in $A \underset{\{A\}}{\parallel} (A; \mathit{THROW})$, i.e. $(\langle A\rangle, \{A, \checkmark, ?\})$. If we use the definition (8), the synchronized failure set will contain $(\langle A\rangle, \{A, \checkmark, ?, !\})$, which indicates $A \underset{\{A\}}{\parallel} (A; \mathit{THROW})$ will deadlock after executing $\langle A\rangle$.

The synchronized failure set of two failures is defined as follows.

$$(s, Y) \oplus (t, Z) = \begin{cases} \{(u, Y \cup Z) \mid Y \setminus (X \cup \Omega) = Z \setminus (X \cup \Omega) \wedge u \in s \underset{X}{\parallel} t\} \\ \qquad \textbf{if } (s, Y \cup \Omega) \in \mathcal{F_S}(P) \vee (t, Z \cup \Omega) \in \mathcal{F_S}(Q) \\ \{(u, (Y \cup Z) \setminus \Theta) \mid Y \setminus (X \cup \Omega) = Z \setminus (X \cup \Omega) \wedge u \in s \underset{X}{\parallel} t \wedge \\ \qquad \Theta = rf(\omega_1, \omega_2)\} \\ \qquad \textbf{otherwise} \end{cases} \tag{9}$$

where ω_1 is the terminal event that P can engage in after performing s, i.e. $\forall (s, Y_1) \in \mathcal{F_S}(P) \bullet Y \subseteq Y_1 \Rightarrow (\omega_1 \in \Omega \wedge \omega_1 \notin Y_1)$, ω_2 is the terminal event that Q can engage in after performing t, and the function rf synchronizing the terminal events is defined as follows, in which $\omega_1 \& \omega_2$ is defined in Fig. 1.

$$rf(\omega_1, \omega_2) = \begin{cases} \{\omega_1 \& \omega_2\} & \omega_1 \in \Omega \wedge \omega_2 \in \Omega \\ \{\omega_1\} & \omega_1 \in \Omega \wedge \omega_2 = \epsilon \\ \{\omega_2\} & \omega_2 \in \Omega \wedge \omega_1 = \epsilon \\ \{\} & \omega_1 = \epsilon \wedge \omega_2 = \epsilon \end{cases} \tag{10}$$

If P or Q can terminate with different terminal events after executing a trace, ω_1 or ω_2 may not exist for some failures, e.g. $(\langle\rangle, \{?\})$ in the failure set of the process $\mathit{SKIP} \sqcap \mathit{THROW}$. If ω_1 or ω_2 does not exist, we use ϵ to represent it. Now the failure set of $P \underset{X}{\parallel} Q$ is defined below.

$$\begin{aligned}\mathcal{F_S}(P \underset{X}{\parallel} Q) = \{(u, E) \mid (u, E) \in (s, Y) \oplus (t, Z) \wedge \\ \exists s, t \bullet (s, Y) \in \mathcal{F_S}(P) \wedge (t, Z) \in \mathcal{F_S}(Q)\}\end{aligned}$$

For example, the trace and failure sets of the process $A \underset{\{A\}}{\parallel} (A; \mathit{THROW})$ in the last example are $\{\langle\rangle, \langle A\rangle, \langle A, !\rangle\}$ and $\{(\langle\rangle, X) \mid X \subseteq \Omega\} \cup \{(\langle A\rangle, X) \mid X \subseteq \{A, \checkmark, ?\}\}$ $\cup \{(\langle A, !\rangle, X) \mid X \subseteq \Sigma^{\Omega}\}$, respectively.

The following laws for parallel composition reflect the termination policies in a parallel composition.

$$\begin{aligned} \mathit{YIELD} \underset{X}{\parallel} \mathit{SKIP} &= \mathit{YIELD} & \mathit{THROW} \underset{X}{\parallel} \mathit{SKIP} &= \mathit{THROW} \\ \mathit{THROW} \underset{X}{\parallel} \mathit{YIELD} &= \mathit{THROW} & \mathit{THROW} \underset{X}{\parallel} \mathit{THROW} &= \mathit{THROW} \end{aligned}$$

If P does not terminate with an yield terminal event, i.e. $\forall s \in \mathcal{T_S}(P) \bullet s \notin \Sigma^*_{\{?\}}$, the parallel composition $\parallel$ without synchronization agrees with the composition $\underset{\{\}}{\parallel}$ and it enjoys the following laws.

$$\begin{aligned} \mathit{THROW} \parallel P &= P \ ; \ \mathit{THROW} \\ \mathit{THROW} \parallel (\mathit{YIELD} \ ; \ P) &= \mathit{THROW} \sqcap (P \ ; \ \mathit{THROW}) \end{aligned}$$

The last law says that a process can be interrupted by an interrupt from the environment, but the interruption does not have priority over other events.

Exception handling $P \rhd Q$ behaves similarly to $P; Q$, but Q starts to execute only after an exception is thrown in P.

$$\begin{aligned} \mathcal{T_S}(P \rhd Q) &= \{s \mid s \in \mathcal{T_S}(P) \cap \Sigma^{*\{\checkmark, ?\}}\} \cup \{s\,\hat{}\,t \mid s\,\hat{}\langle !\rangle \in \mathcal{T_S}(P) \wedge t \in \mathcal{T_S}(Q)\} \\ \mathcal{F_S}(P \ ; \ Q) &= \{(s, X) \mid s \in \Sigma^{*\{\checkmark, ?\}} \wedge (s, X \cup \{!\}) \in \mathcal{F_S}(P)\} \cup \\ &\quad \{(s\,\hat{}\,t, X) \mid s\,\hat{}\langle !\rangle \in \mathcal{T_S}(P) \wedge (t, X) \in \mathcal{F_S}(Q)\} \end{aligned}$$

Laws for exception handling:

$$\begin{aligned} P \rhd \mathit{THROW} &= P & P \rhd (Q \sqcap R) &= (P \rhd Q) \sqcap (P \rhd R) \\ \mathit{THROW} \rhd P &= P & (P \sqcap Q) \rhd R &= (P \rhd R) \sqcap (Q \rhd R) \\ \mathit{SKIP} \rhd P &= \mathit{SKIP} & P \rhd (Q \Box R) &= (P \rhd Q) \Box (P \rhd R) \\ \mathit{YIELD} \rhd P &= \mathit{YIELD} & P \rhd (Q \rhd R) &= (P \rhd Q) \rhd R \\ \mathit{STOP} \rhd P &= \mathit{STOP} \end{aligned}$$

The terminal events do not affect hiding and renaming operators. Thus, their definitions remain the same as those given in the classical CSP.

3.2 Semantics of compensable process

The semantics of a compensable process PP is to be defined as a triple (T, F, C), where T and F are the trace and failure sets of the *forward behavior*, and $C \subseteq \Sigma_\Omega^* \times \mathbb{P}(\Sigma^{*\Omega}) \times \mathbb{P}(\Sigma^{*\Omega} \times \mathbb{P}(\Sigma^\Omega))$ defines the *compensation behavior.* The reason for the separation of forward and compensation behaviors is the compensation behavior needs to be recorded during the execution of the forward behavior. An element in C is (s, T^c, F^c), which shows that the behavior defined by the trace set T^c and the failure set F^c can compensate the effects caused by executing the terminating trace s from the forward behavior. Therefore, both the forward behavior (T, F), denoted by PP_f, and the compensation behavior (T^c, F^c) of each element in C, denoted by PP_c, satisfy the axioms of the semantics of the standard processes given in Section 3.1. We can thus overload the semantic functions $\mathcal{T}_\mathcal{S}$ and $\mathcal{F}_\mathcal{S}$ and the operators on standard processes and apply them to PP_f and PP_c. For examples, $\mathcal{F}_\mathcal{S}(PP_f) = F$ and $\mathcal{F}_\mathcal{S}(PP_c) = F^c$, and later when we define the semantics of PP ; QQ, we will use the notations $\mathcal{T}_\mathcal{S}(PP_f \ ; \ QQ_f)$ and $\mathcal{F}_\mathcal{S}(PP_f \ ; \ QQ_f)$ as if PP_f and QQ_f are standard processes. In addition, (T, F, C) is required to satisfy the following axiom.

$$\forall (s, T^c, F^c) \in C \bullet s \in \Sigma_\Omega^* \cap \{s \mid (s, X) \in F\} \tag{11}$$

It means the trace s of each element in C is a stable terminating trace in the forward behavior.

We define the triple (T, F, C) for a PP by three semantic functions: the *forward trace set* function $\mathcal{T}^c : \mathcal{PP} \to \mathbb{P}(\Sigma^{*\Omega})$, the *forward failure set* function $\mathcal{F}^c : \mathcal{PP} \to \mathbb{P}(\Sigma^{*\Omega} \times \mathbb{P}(\Sigma^\Omega))$, and the *compensation behavior set* function $\mathcal{C} : \mathcal{PP} \to \mathbb{P}(\Sigma_\Omega^* \times \mathbb{P}(\Sigma^{*\Omega}) \times \mathbb{P}(\Sigma^{*\Omega} \times \mathbb{P}(\Sigma^\Omega)))$.

Compensation pair If the forward behavior terminates successfully, the behavior of Q is recorded such that it can be executed to compensate the effect of P when triggered by an exception later. Otherwise, Q will not be executed. The semantics of a compensation pair $P \div Q$ attaches the successfully terminating trace in the forward behavior, i.e. $s \in T \cap \Sigma_{\{\checkmark\}}^*$, with the trace and failure sets of Q, and the others terminating traces (the traces in $\Sigma_{\{!,?\}}^*$) with those of $SKIP$.

$$\begin{aligned}
\mathcal{T}^c(P \div Q) &= \mathcal{T}_\mathcal{S}(P) \quad \mathcal{F}^c(P \div Q) = \mathcal{F}_\mathcal{S}(P) \\
\mathcal{C}(P \div Q) &= \{(s, T^c, F^c) \mid \exists s \in T \cap \Sigma_\Omega^* \bullet \\
&\qquad (s = t\hat{\ }\langle \checkmark \rangle \wedge T^c = \mathcal{T}_\mathcal{S}(Q) \wedge F^c = \mathcal{F}_\mathcal{S}(Q)) \vee \\
&\qquad (s \in \Sigma_{\{!,?\}}^* \wedge T^c = \mathcal{T}_\mathcal{S}(SKIP) \wedge F^c = \mathcal{F}_\mathcal{S}(SKIP))\}
\end{aligned}$$

The compensation behavior set of the compensable processes $STOP \div P$ is empty. In the following, we use $STOPP$ to denote the compensable processes whose forward behaviors are $STOP$. The following two laws hold for compensation pairs.

$$Q_1 = Q_2 \Rightarrow P \div Q_1 = P \div Q_2 \qquad P_1 = P_2 \Rightarrow P_1 \div Q = P_2 \div Q$$

The definitions of the basic compensable processes are the same as those in Section 2.2.

Transaction block The semantics of a transaction block $[PP]$ can be defined in terms of the semantics of the compensable process PP in the block.

$$\begin{aligned}\mathcal{T}_{\mathcal{S}}([PP]) = {} & (\mathcal{T}^c(PP) \setminus \Sigma^*_{\{!\}}) \cup \\ & \{s_1 \mid \exists (s, T^c, D^c) \in \mathcal{C}(PP) \bullet s = t\,\hat{}\,\langle ! \rangle \wedge s_2 \in T^c \wedge s_1 = t\,\hat{}\,s_2\} \\ \mathcal{F}_{\mathcal{S}}([PP]) = {} & \{(s, X) \mid s \in \Sigma^* \wedge (s, X \cup \{!\}) \in \mathcal{F}^c(PP)\} \cup \\ & \{(s_1, X_1) \mid \exists (s, T^c, F^c) \in \mathcal{C}(PP) \bullet (s \in \Sigma^*_{\{\checkmark, ?\}} \wedge s_1 = s \wedge X_1 \subseteq \Sigma^{\Omega}) \vee \\ & \qquad (s = t\,\hat{}\,\langle ! \rangle \wedge (s_2, X_2) \in F^c \wedge s_1 = t\,\hat{}\,s_2 \wedge X_1 = X_2)\}\end{aligned}$$

The compensation behavior of PP will be executed to recover from a failure occurred in the forward behavior. The trace set of $[PP]$ contains the traces of the forward behavior of PP and the traces of compensation behavior. The failure set $\mathcal{F}_{\mathcal{S}}([PP])$ contains the failures of the forward behavior that do not terminate with an exception terminal event. It also includes the failures that extend the exception terminating traces of the forward behavior with the failures of the compensation behavior. Different from the original cCSP, we keep the yield interruption behavior in the semantics of transaction block. The following laws hold.

$$\begin{aligned} [SKIP \div P] &= SKIP & [STOPP] &= STOP \\ [THROW \div P] &= SKIP & [P \div Q] &= P \rhd SKIP \\ [YIELD \div P] &= YIELD & PP_1 = PP_2 &\Rightarrow [PP_1] = [PP_2] \end{aligned}$$

The law $[P \div Q] = P \rhd SKIP$ fixes the problem of the original cCSP pointed out in Section 2.2, i.e. $[P \div Q] = P$ under the assumption that P does not terminate with the yield terminal event.

Sequential composition In a sequential composition $PP;QQ$, the forward behavior PP_f and the forward behavior QQ_f are composed first, and the compensation behavior PP_c and the compensation behavior QQ_c are composed in the reverse direction, just like the model of Sagas [9].

$$\mathcal{T}^c(PP\ ;\ QQ) = \mathcal{T}_{\mathcal{S}}(PP_f\ ;\ QQ_f) \quad \mathcal{F}^c(PP\ ;\ QQ) = \mathcal{F}_{\mathcal{S}}(PP_f\ ;\ QQ_f)$$

$$\begin{aligned}\mathcal{C}(PP\ ;\ QQ) = {} & \{(s, T^c, F^c) \mid \exists (s_1, PP_c) \in \mathcal{C}(PP), (s_2, QQ_c) \in \mathcal{C}(QQ) \bullet \\ & (s_1 = t\,\hat{}\,\langle \checkmark \rangle \wedge s = t\,\hat{}\,s_2 \wedge T^c = \mathcal{T}_{\mathcal{S}}(QQ_c; PP_c) \wedge F^c = \mathcal{F}_{\mathcal{S}}(QQ_c; PP_c)) \vee \\ & (s_1 \neq t\,\hat{}\,\langle \checkmark \rangle \wedge s = s_1 \wedge T^c = \mathcal{T}_{\mathcal{S}}(PP_c) \wedge F^c = \mathcal{F}_{\mathcal{S}}(PP_c))\ \}\end{aligned}$$

For example, the compensation behavior of $A_1 \div B_1; A_2 \div B_2$ is $\{(\langle A_1, A_2, \checkmark \rangle, \mathcal{T}_{\mathcal{S}}(B_2; B_1), \mathcal{F}_{\mathcal{S}}(B_2; B_1))\}$, and that of $A_1 \div B_1; YIELDD$ contains two elements: $(\langle A_1, \checkmark \rangle, \mathcal{T}_{\mathcal{S}}(B_1), \mathcal{F}_{\mathcal{S}}(B_1))$ and $(\langle A_1, ? \rangle, \mathcal{T}_{\mathcal{S}}(B_1), \mathcal{F}_{\mathcal{S}}(B_1))$.

Sequential composition satisfies the following laws. In the last two laws, we assume the standard processes P, P_1 and P_2 do not terminate with an exception terminal event.

$$\begin{aligned} SKIPP\ ;\ PP &= PP & YIELDD\ ;\ YIELDD &= YIELDD \\ PP\ ;\ SKIPP &= PP & [P \div Q; THROWW] &= P; Q \\ THROWW\ ;\ PP &= THROWW & [P_1 \div Q_1; P_2 \div Q_2; THROWW] &= P_1; P_2; Q_2; Q_1 \end{aligned}$$

The second law fixes the right unit problem of the original trace model pointed out in Section 2.2. The fifth law fixes another problem pointed out there and ensures that $[P \div Q; THROWW] = P; Q$ provided that P does not terminate with ?. The last two laws are also valid in the case that the standard processes terminate with ?, and they relax the assumption in the original cCSP that requires all the standard processes terminate successfully.

Internal choice The semantics of internal choice $PP \sqcap QQ$ is as follows.

$$\mathcal{T}^c(PP \sqcap QQ) = \mathcal{T}_\mathcal{S}(PP_f \sqcap QQ_f) \quad \mathcal{F}^c(PP \sqcap QQ) = \mathcal{F}_\mathcal{S}(PP_f \sqcap QQ_f)$$
$$\mathcal{C}(PP \sqcap QQ) = \mathcal{C}(PP) \cup \mathcal{C}(QQ)$$

For example, the compensation behavior set of $A \div B_1 \sqcap A \div B_2$ is $\{(\langle A, \checkmark\rangle, \mathcal{T}_\mathcal{S}(B_1), \mathcal{F}_\mathcal{S}(B_1)), (\langle A, \checkmark\rangle, \mathcal{T}_\mathcal{S}(B_2), \mathcal{F}_\mathcal{S}(B_2))\}$. We have the following laws hold for internal choice.

$$\begin{array}{rl} PP \sqcap PP = PP & PP \sqcap (QQ \sqcap RR) = (PP \sqcap QQ) \sqcap RR \\ PP \sqcap QQ = QQ \sqcap PP & PP \ ; \ (QQ \sqcap RR) = (PP \ ; \ QQ) \sqcap (PP \ ; \ RR) \\ [PP \sqcap QQ] = [PP] \sqcap [QQ] & (QQ \sqcap RR) \ ; \ PP = (QQ \ ; \ PP) \sqcap (RR \ ; \ PP) \end{array}$$

External choice As in the case of the internal choice, the external choice is made during the forward behavior, but it is the environment to make the choice.

$$\mathcal{T}^c(PP \Box QQ) = \mathcal{T}_\mathcal{S}(PP_f \Box QQ_f) \quad \mathcal{F}^c(PP \Box QQ) = \mathcal{F}_\mathcal{S}(PP_f \Box QQ_f)$$
$$\mathcal{C}(PP \Box QQ) = \mathcal{C}(PP) \cup \mathcal{C}(QQ)$$

For example, $\mathcal{C}(STOPP \Box A \div B)$ equals to $\mathcal{C}(STOPP \sqcap A \div B)$, and they are equal to $\{(\langle A, \checkmark\rangle, \mathcal{T}_\mathcal{S}(B), \mathcal{F}_\mathcal{S}(B))\}$. But their forward failures sets are different: $\mathcal{F}^c(STOPP \Box A \div B)$ is $\mathcal{F}_\mathcal{S}(A)$, and $\mathcal{F}^c(STOPP \sqcap A \div B)$ is $\mathcal{F}_\mathcal{S}(A) \cup \mathcal{F}_\mathcal{S}(STOP)$. The following laws hold for external choice.

$$\begin{array}{rl} PP \Box PP = PP & [PP \Box QQ] = [PP] \Box [QQ] \\ PP \Box QQ = QQ \Box PP & PP \Box (QQ \Box RR) = (PP \Box QQ) \Box RR \\ STOPP \Box PP = PP & PP \Box (QQ \sqcap RR) = (PP \Box QQ) \sqcap (PP \Box RR) \end{array}$$

Notice that external choice distributes over internal choice. From the laws for internal and external choices, we can see that a transaction block process of a choice between compensable processes equals to a choice between the transaction block processes of the compensable processes.

Parallel composition In a generalized parallel composition $PP \underset{X}{\parallel} QQ$, the forward behaviors of PP and QQ synchronize on X, so do their compensation behaviors:

$$\mathcal{T}^c(PP \underset{X}{\parallel} QQ) = \mathcal{T}_\mathcal{S}(PP_f \underset{X}{\parallel} QQ_f) \quad \mathcal{F}^c(PP \underset{X}{\parallel} QQ) = \mathcal{F}_\mathcal{S}(PP_f \underset{X}{\parallel} QQ_f)$$
$$\mathcal{C}(PP \underset{X}{\parallel} QQ) = \{(s, T^c, F^c) \mid \exists (s_1, PP_c) \in \mathcal{C}(PP), (s_2, QQ_c) \in \mathcal{C}(QQ) \bullet$$
$$s \in (s_1 \underset{X}{\parallel} s_2) \wedge T^c = \mathcal{T}_\mathcal{S}(PP_c \underset{X}{\parallel} QQ_c) \wedge F^c = \mathcal{F}_\mathcal{S}(PP_c \underset{X}{\parallel} QQ_c)\}$$

Here are two examples. First, $[(A \div B_1 \underset{\{A\}}{\parallel} A \div B_2); THROWW] = A; B_1 \parallel B_2$ shows the synchronization between the forward behaviors, and then $A_1 \div B_1 \underset{\{A_1, A_2\}}{\parallel} A_2 \div B_2$ $= STOPP$ demonstrates the case of a deadlock in the forward behavior. The following laws for parallel composition hold.

$$PP \underset{X}{\parallel} QQ = QQ \underset{X}{\parallel} PP$$
$$PP \underset{X}{\parallel} (QQ \underset{X}{\parallel} RR) = (PP \underset{X}{\parallel} QQ) \underset{X}{\parallel} RR$$
$$PP \underset{X}{\parallel} (QQ \sqcap RR) = (PP \underset{X}{\parallel} QQ) \sqcap (PP \underset{X}{\parallel} RR)$$

Furthermore, parallel composition and sequential composition are related by the following laws, where all the standard processes are assumed to terminate successfully.

$$[(P_1 \div Q_1 \underset{X}{\parallel} P_2 \div Q_2) \ ; \ \mathit{THROWW}] = (P_1 \underset{X}{\parallel} P_2); (Q_1 \underset{X}{\parallel} Q_2)$$
$$[(P_1 \div Q_1 \ ; \ P_2 \div Q_2) \parallel \mathit{THROWW}] = P_1 \ ; \ P_2 \ ; \ Q_2 \ ; \ Q_1$$
$$[(P_1 \div Q_1 \ ; \ \mathit{YIELDD} \ ; \ P_2 \div Q_2) \parallel \mathit{THROWW}] = (P_1 \ ; \ Q_1) \sqcap (P_1 \ ; \ P_2 \ ; \ Q_2 \ ; \ Q_1)$$
$$[(\mathit{YIELDD} \ ; \ P_1 \div Q_1 \ ; \ \mathit{YIELDD} \ ; \ P_2 \div Q_2) \parallel \mathit{THROWW}] = \mathit{SKIP} \sqcap (P_1 \ ; \ Q_1) \sqcap (P_1 \ ; \ P_2 \ ; \ Q_2 \ ; \ Q_1)$$
$$[P_1 \div Q_1 \parallel P_2 \div Q_2 \parallel \mathit{THROWW}] = (P_1 \parallel P_2) \ ; \ (Q_1 \parallel Q_2)$$
$$[(\mathit{YIELDD} \ ; \ P_1 \div Q_1) \parallel (\mathit{YIELDD} \ ; \ P_2 \div Q_2) \parallel \mathit{THROWW}] = \mathit{SKIP} \sqcap (P_1 \ ; \ Q_1) \sqcap (P_2 \ ; \ Q_2) \sqcap ((P_1 \parallel P_2) \ ; \ (Q_1 \parallel Q_2))$$

The 3rd and 4th laws say that a compensable process in a parallel composition can be interrupted by *YIELDD*, meaning that the process yields to an interrupt from the environment. A compensable process will not be interrupted if no *YIELDD* is used (the 2nd and 5th laws). This is one of the differences from the original cCSP, where a compensable process can implicitly yield to an interrupt from the environment (cf. Section 2.2). We believe that it is more reasonable to let the designer to specify where a compensable process can yield to an interruption from the environment.

Hiding and renaming Hiding and renaming are defined by the standard hiding and renaming on the forward behavior and the compensation behavior.

$$\mathcal{T}^c(PP \setminus X) = \mathcal{T}_{\mathcal{S}}(PP_f \setminus X) \quad \mathcal{F}^c(PP \setminus X) = \mathcal{F}_{\mathcal{S}}(PP_f \setminus X)$$
$$\mathcal{C}(PP \setminus X) = \{(s, T^c, F^c) \mid \exists (s_1, PP_c) \in \mathcal{C}(PP) \bullet s = s_1 \setminus X \wedge T^c = \mathcal{T}_{\mathcal{S}}(PP_c \setminus X) \wedge F^c = \mathcal{F}_{\mathcal{S}}(PP_c \setminus X)\}$$

The renaming semantics is as follows.

$$\mathcal{T}^c(PP[\![R]\!]) = \mathcal{T}_{\mathcal{S}}(PP_f[\![R]\!]) \quad \mathcal{F}^c(PP[\![R]\!]) = \mathcal{F}_{\mathcal{S}}(PP_f[\![R]\!])$$
$$\mathcal{C}(PP[\![R]\!]) = \{(s, T^c, F^c) \mid \exists (s_1, PP_c) \in \mathcal{C}(PP) \bullet s_1 \ R \ s \wedge T^c = \mathcal{T}_{\mathcal{S}}(PP_c[\![R]\!]) \wedge F^c = \mathcal{F}_{\mathcal{S}}(PP_c[\![R]\!])\}$$

Hiding and renaming satisfy the following laws. In particular, both are distributive among internal choice, but the hiding operator is not distributive among external choice, e.g. $((A; A_1) \div B \Box (A; A_2) \div B) \setminus \{A\}$ is equal to $A_1 \div B \sqcap A_2 \div B$.

$$(PP \setminus X) \setminus Y = (PP \setminus Y) \setminus X \qquad (PP \sqcap QQ)[\![R]\!] = PP[\![R]\!] \sqcap QQ[\![R]\!]$$
$$(PP \setminus X) \setminus Y = PP \setminus (X \cup Y) \qquad (PP \Box QQ)[\![R]\!] = PP[\![R]\!] \Box QQ[\![R]\!]$$
$$(PP \sqcap QQ) \setminus X = (PP \setminus X) \sqcap (QQ \setminus X) \qquad (PP[\![R]\!])[\![R']\!] = PP[\![R \circ R']\!]$$
$$PP \setminus \{\} = PP$$

4 Case study

This section will give a case study to demonstrate the extended cCSP. It is a business process of an online shop. The system is composed by four parties: a shop, a supplier, a shipper and a bank. The business behavior of each party is compensable, and will be specified by a compensable process.

After receiving a client request (`ReceiveRequest`), the shop contacts its supplier to ask (`SupplierRequest`) whether there exist enough goods. If the storage is not enough (`NotEnough`), the whole process will result in an exception.

Otherwise, the shop will make an order (`Order`) of the goods. The shop then contacts the bank for authorizing the credit card of client (`CreditCheck`). If the credit card is valid, the shop processes payment for the client (`Payment`) and informs the supplier (`NotifySupplier`) that the payment is made. For the sake of efficiency, after receiving the notification, the supplier contacts the bank for checking the payment (`PaymentCheck`) and requests the shipper to ship the goods (`ShipRequest`) to client concurrently. If the credit card authorization or payment checking fails (`NotValid, NotPValid`), the whole process will result in an exception. After receiving the shipping request, the shipper schedules a shipping plan (`Schedule`), delivers the goods (`Deliver`) to the client of the shop, and notifies the supplier (`ShipResult`) about the shipping result. The alphabet Σ of the system is given as follows.

$$\begin{aligned}\Sigma = \{&\texttt{ReceiveRequest, ApologyMail, SupplierRequest, Enough, NotEnough,}\\ &\texttt{Order, UndoOrder, CreditCheck, Valid, NotValid, Payment, Refund,}\\ &\texttt{NotifySupplier, PaymentCheck, PValid, NotPValid, ShipRequest,}\\ &\texttt{NotifyShopShip, ShipResult, Schedule, Deliver, ShipBack}\}\end{aligned}$$

The processes `Shop`, `Supplier`, `Shipper` and `Bank` are specified as follows.

$$\begin{aligned}\texttt{Shop} = \;&\texttt{ReceiveRequest} \div \texttt{ApologyMail}\ ;\\ &(\texttt{SupplierRequest}\ ;\ (\texttt{Enough} \,\Box\, (\texttt{NotEnough}\ ;\ \texttt{THROW}))) \div \texttt{SKIP}\ ;\\ &\texttt{Order} \div \texttt{SKIP}\ ;\ (\texttt{CreditCheck}\ ;\ (\texttt{Valid} \,\Box\, (\texttt{NotValid}\ ;\ \texttt{THROW}))) \div \texttt{SKIP}\ ;\\ &\texttt{Payment} \div \texttt{Refund}\ ;\ \texttt{NotifySupplier} \div \texttt{SKIP}\ ;\ \texttt{NotifyShopShip} \div \texttt{SKIP}\end{aligned}$$

$$\begin{aligned}\texttt{Supplier} = \;&(\texttt{SupplierRequest}\ ;\ (\texttt{Enough} \,\sqcap\, (\texttt{NotEnough}\ ;\ \texttt{THROW}))) \div \texttt{SKIP}\ ;\\ &\texttt{Order} \div \texttt{UndoOrder}\ ;\ \texttt{NotifySupplier} \div \texttt{SKIP}\ ;\\ &((\texttt{PaymentCheck}\ ;\ (\texttt{PValid} \,\Box\, (\texttt{NotPValid}\ ;\ \texttt{THROW}))) \div \texttt{SKIP} \parallel\\ &\ (\texttt{ShipRequest} \div \texttt{SKIP}\ ;\ \texttt{NotifyShopShip} \div \texttt{SKIP}))\ ;\ \texttt{ShipResult} \div \texttt{SKIP}\end{aligned}$$

$$\begin{aligned}\texttt{Shipper} = \;&\texttt{ShipRequst} \div \texttt{SKIP}\ ;\ \texttt{YIELDD}\ ;\ \texttt{Schedule} \div \texttt{SKIP}\ ;\\ &\texttt{YIELDD}\ ;\ \texttt{Deliver} \div \texttt{ShipBack}\ ;\ \texttt{ShipResult} \div \texttt{SKIP}\end{aligned}$$

$$\begin{aligned}\texttt{Bank} = \;&((\texttt{CreditCheck}\ ;\ (\texttt{Valid} \sqcap (\texttt{NotValid}; \texttt{THROW}))) \div \texttt{SKIP}\ ;\ \texttt{Payment} \div \texttt{Refund}) \parallel\\ &(\texttt{PaymentCheck}\ ;\ (\texttt{PValid} \sqcap (\texttt{NotPValid}\ ;\ \texttt{THROW}))) \div \texttt{SKIP}\end{aligned}$$

The global process (`DetailGlobalProcess`) is a transaction block of the synchronized parallel composition of the four compensable processes. If the compensable parallel process in the global process results in an exception, a recovery should be taken, e.g. the credit card will be refunded and the shipper will ship back the delivered goods. We can get a more abstract process (`AbstractGlobalProcess`) by hiding some synchronized events.

$$\texttt{DetailGlobalProcess} = [\ \texttt{Shop} \underset{X}{\parallel} \texttt{Supplier} \underset{X}{\parallel} \texttt{Shipper} \underset{X}{\parallel} \texttt{Bank}\ ]$$

$$\begin{aligned}X = \{&\texttt{SupplierRequest, Enough, NotEnough, Order, CreditCheck, Valid,}\\ &\texttt{NotValid, Payment, Refund, NotifySupplier, PaymentCheck, PValid,}\\ &\texttt{NotPValid, ShipRequest, NotifyShopShip, ShipResult}\}\end{aligned}$$

$$\texttt{AbstractGlobalProcess} = [\ \texttt{DetailGloablProcess} \setminus X_1\ ]$$

$X_1 = (X \cup \{\texttt{Schedule}\}) \setminus \{\texttt{Payment, Refund, Order}\}$

Based on the preceding semantic definitions and laws, the abstract process can be reduced to the following process.

$[\ \texttt{ReceiveRequest} \div \texttt{ApologyMail}\ ;\ \texttt{SKIPP} \sqcap \texttt{THROWW}\ ;\ \texttt{Order} \div \texttt{UndoOrder}\ ;$
$\ \texttt{SKIPP} \sqcap \texttt{THROWW}\ ;\ \texttt{Payment} \div \texttt{Refund}\ ;$
$\ (\texttt{SKIPP} \sqcap \texttt{THROWW}) \parallel (\texttt{YIELDD}\ ;\ \texttt{Deliver} \div \texttt{ShipBack})\ ]$

It provides an abstract choreography view of the system. According to the semantics, we can get the following results. The proofs of results are omitted due to the space limit and are reported in [8].

- The global process will not deadlock. If we add the event `ReceiveRequest` to the synchronization set X, a deadlock will happen at the beginning. The reason is: besides `Shop`, no other process can execute the event `ReceiveRequest` at the beginning.
- If an exception occurs, `ApologyMail` is the last event performed in the recovery. From the abstract process, we can see that there are four different cases of recovery: 1) if the storage is not enough, then only `ApologyMail` will be preformed; 2) if the credit card authorization fails, then the trace $\langle$`UndoOrder, ApologyMail`$\rangle$ will be performed; 3) if the shipper yields to the exception from the supplier, and the goods delivery will be canceled, then $\langle$`Refund, UndoOrder, ApologyMail`$\rangle$ will be performed; 4) if the payment checking fails after the goods delivery, then the execution sequence of the recovery is $\langle$`ShipBack, Refund, UndoOrder, ApologyMail`$\rangle$.

5 Conclusions and future work

LRT are important in SOC. It is important that LRT can be formally specified and verified with tool support. The extension of CSP into cCSP is one of the useful attempts in this direction. However, cCSP does not provide the facilities for defining internal choice, hiding and synchronization. This together with the only available trace semantics (and the operational semantics) severely limits the expressive power of the language that it does not specify and reason about non-determinism and dead-lock. In this paper, we have extended cCSP with internal choice, hiding and synchronization in order to be able to specify behavior of LRT at different levels of abstractions. Accordingly, we have provided a stable failures semantics for the extended notation for reasoning about non-determinism and deadlock.

Along with the semantic definitions of the operators, we present the important algebraic laws of the operators. As a by-product of the investigation of the algebraic laws, we have discovered some laws which were claimed to hold but actually do not hold for the trace semantics of cCSP (cf. Section 2.2). In addition, we have proved those laws do hold for our stable failures semantics.

The separation of the forward behavior and compensation behavior in the semantic definition of the compensable processes allows us to understand and analysis the two kinds of computations individually. We can refine and reason about the two parts separately, but the price we are paying is it makes the fixed point theory not clear when recursion is introduced.

From the perspective of theory, future work includes the study of recursion and divergence of LRT, and thus to develop of a full theory of failure-divergence of LRT and their refinements.

Acknowledgments The work of the first author was supported by the National Basic Research Program of China (973) under Grant No. 2005CB321802 and the NSFC projects under Grant Nos. 60725206, 60970035, 60803042, and the second author acknowledges the support from Macau Science and Technology Development Fund through the project GAVES.

References

1. A. Alves, A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, Sterling, D. König, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web services business process execution language version 2.0. OASIS Committee Draft, May 2006.
2. R. Bruni, M. J. Butler, C. Ferreira, C. A. R. Hoare, H. C. Melgratti, and U. Montanari. Comparing two approaches to compensable flow composition. In M. Abadi and L. de Alfaro, editors, *CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2005.
3. R. Bruni, H. C. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In J. Palsberg and M. Abadi, editors, *POPL*, pages 209–220. ACM, 2005.
4. M. J. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In R. D. Nicola, G. L. Ferrari, and G. Meredith, editors, *COORDINATION*, volume 2949 of *Lecture Notes in Computer Science*, pages 87–104. Springer, 2004.
5. M. J. Butler, C. Ferreira, and M. Y. Ng. Precise modelling of compensating business transactions and its application to BPEL. *J. UCS*, 11(5):712–743, 2005.
6. M. J. Butler, C. A. R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In A. E. Abdallah, C. B. Jones, and J. W. Sanders, editors, *25 Years Communicating Sequential Processes*, volume 3525 of *Lecture Notes in Computer Science*, pages 133–150. Springer, 2004.
7. M. J. Butler and S. Ripon. Executable semantics for compensating CSP. In M. Bravetti, L. Kloul, and G. Zavattaro, editors, *EPEW/WS-FM*, volume 3670 of *Lecture Notes in Computer Science*, pages 243–256. Springer, 2005.
8. Z. Chen and Z. Liu. An extended cCSP with stable failures semantics. Technical Report 431, UNU-IIST, 2010.
9. H. Garcia-Molina and K. Salem. SAGAS. In U. Dayal and I. L. Traiger, editors, *SIGMOD Conference*, pages 249–259. ACM Press, 1987.
10. M. C. Little. Transactions and web services. *Commun. ACM*, 46(10):49–54, 2003.
11. S. Ripon. *Extending and Relating Semantic Models of Compensating CSP*. PhD thesis, University of Southampton, 2008.
12. S. Ripon and M. J. Butler. PVS embedding of cCSP semantic models and their relationship. *Electr. Notes Theor. Comput. Sci.*, 250(2):103–118, 2009.
13. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
14. S. Thatte. XLANG web services for business process design, 2001.