An Operational Semantics for Model Checking Long Running Transactions

Hengbiao Yu[1,2], Zhenbang Chen[1,2]*, and Ji Wang[1,2]

[1] College of Computer, National University of Defense Technology, Changsha, China
[2] Science and Technology on Parallel and Distributed Processing Laboratory, Changsha, China
fishzqyhb@gmail.com,{zbchen,wj}@nudt.edu.cn

Abstract. Compensating CSP (cCSP) is an extension to CSP for modeling long running transactions (LRTs). In our work, we extended the original cCSP with the ability of modeling non-determinism, deadlock and livelock. Until now, there is only a failure-divergence semantics for the extended cCSP, and there is no model checking or animating tool for it. In this paper, we present an operational semantics for model checking the extended cCSP. We prove that the general problem of model checking the extended cCSP with respect to regular properties is undecidable. Using the operational semantics, we have implemented an animator and a prototype model checker for the extended cCSP based on the platform Process Analysis Toolkit (PAT). In addition, a case study is given to demonstrate the tool.

Keywords: Long Running Transactions, Extended cCSP, Model Checking, Operational Semantics.

1 Introduction

Service-Oriented Computing (SOC) [18] provides a new computing paradigm for distributed computing. In SOC, services are supposed to be loosely coupled, widely located and provided by different organizations. Usually, a task in SOC is accomplished by a coordination of different services, and the coordination is often carried out by a third party. How to ensure the consistency of a task in case of a service failure is not an easy job. The classical ACID transaction model [12] is not realistic for this scenario, because the third party cannot isolate the resources of service providers. There are other consistency mechanisms for distributed systems, such as two-phrase protocol [15], but they are not appropriate for SOC either, especially on Internet.

Recently, long-running transaction (LRT) [12] models, such as SAGA [11], are used as a mainstream approach [19] for ensuring the consistency of service

* Corresponding author.

coordinations. An LRT in SOC usually involves the interactions with multiple services, and needs a certain period to complete. The most important notion in LRTs is *compensation*, which is used for state recovery in case of a failure. Until now, many industrial service composition languages, such as WS-BPEL [1] and XLANG [22], have used LRT models and provided compensation-based LRT programming facilities.

In order to ensure the correctness of the LRTs in SOC, some formal languages have been proposed to specify and verify LRTs, including StAC [4], SAGAs calculi [3], cCSP [5], *etc.* These formalisms can be used not only as the theoretical foundations for the rigorous design of LRTs, but also as the basis for the application of verification techniques, such as model checking and theorem proving, to improve the reliability of service coordinations. Compensating CSP (cCSP) extends CSP [20] with *backward recovery* mechanism [11]. In cCSP, there are two kinds of processes and they are *standard* processes and *compensable* processes. Standard processes are basically CSP processes extended with exception handling processes and transaction block processes. A compensable process specifies the recovery behaviour of an LRT.

The original cCSP in [5] is given with a trace-based denotational semantics and later with an operational semantics in [6]. In [7], cCSP is extended to support non-determinism and deadlock modeling enabled by a stable-failures semantics. Later, cCSP is further extended in [8][9] with recursion and we call this version the extended cCSP. A failure-divergence (FD) semantics and a refinement calculus are presented in [8][9]. Thus, the extended cCSP has the fully expressive power, as that of CSP for modeling standard communicating processes, for describing all features of LRTs, including compensation, backward recovery, non-determinism, deadlock, livelock, *etc.* However, until now, there is no operational semantics for this cCSP. In the result of it, no model checker or animator exists for the extended cCSP, which brings difficulties to the LRT modeling and verification using the extended cCSP.

To this end, we define an operational semantics in this paper for the extended cCSP. It follows the ideas in the definitions of the operational semantics [6] for the original cCSP. Based on the operational semantics, we have implemented an animator and a prototype model checker for the extended cCSP. Furthermore, the derivation of the FD semantics [9] from the operational semantics is given. The model checking problem of the extended cCSP with respect to regular properties is studied, and we prove that the problem is undecidable in general. In addition, we have carried out a case study to justify our tool. To the best of our knowledge, our tool is the first one supporting LTL modeling checking and refinement checking for LRTs.

The rest of this paper is organized as follows. We give a brief introduction to the extended cCSP in Section 2. Section 3 presents the operational semantics and the derivation of FD semantics from the operational semantics. Section 4 studies the model checking problem with respect to regular properties and introduces our prototype tool for the extended cCSP. Section 5 demonstrates the tool by a case study. Section 6 discusses and compares the related work. Finally, Section 7 draws conclusions and points out future work.

2 Extended Compensating CSP

The extended cCSP [9] extends cCSP by distinguishing internal and external choices, and incorporating the operators of synchronized parallel composition, hiding, renaming and recursion. Assuming Σ is a *finite* set of *normal events* that the processes of the extended cCSP can perform, the syntax of the extended cCSP is presented in the follows, where $a \in \Sigma$, $X \subseteq \Sigma$ is a *finite* event subset, and $\rho \subseteq \Sigma \times \Sigma$ is a renaming relation.

$$\begin{aligned}
P &::= a \mid P;P \mid P \sqcap P \mid P \Box P \mid P \underset{X}{\parallel} P \mid P \setminus X \mid P[\![\rho]\!] \mid P \rhd P \mid [PP] \mid \textbf{skip} \mid \\
&\quad\;\; \textbf{throw} \mid \textbf{yield} \mid \mu\, p.F(p) \\
PP &::= P \div P \mid PP;PP \mid PP \sqcap PP \mid PP \Box PP \mid PP \underset{X}{\parallel} PP \mid PP \boxtimes PP \mid PP \setminus X \mid \\
&\quad\;\; PP[\![\rho]\!] \mid \textbf{skipp} \mid \textbf{throww} \mid \textbf{yieldd} \mid \mu\, pp.FF(pp)
\end{aligned}$$

The extended cCSP has two kinds of processes: the standard processes ranged over by P, and the compensable processes ranged over by PP. Process a terminates successfully after performing event a. The sequential composition $P;Q$ executes P first, then executes Q if P terminates successfully; the choice between P and Q executes either P or Q, the selection in external choice $P \Box Q$ depends on which one is first to be able to start, while in internal choice $P \sqcap Q$, it is nondeterministic; the process P and Q in $P \underset{X}{\parallel} Q$ will synchronize on any event in X; $P \setminus X$ represents that any event in X in the execution of P will be invisible; $P[\![\rho]\!]$ renames the events in the execution of P, according to the binary relation ρ; an exception handling process $P \rhd Q$ will execute Q when P throws an exception, otherwise it behaves like P; transaction block $[PP]$ provides a way to convert a compensable process to a standard process; there are three primitive standard processes: **skip** immediately terminates successfully, **yield** will yield to an interrupt or terminate successfully, and **throw** represents an exception happens and the process will be interrupted; $\mu\, p.F(p)$ represents a recursive process whose behavior is defined by the function $F(p)$.

The operators on compensable processes are similar to those in standard processes. A compensable process is composed by compensation pairs of the form $P \div Q$, in which P is the forward process, and Q is the compensation process that can compensate the effects caused by P. The meanings of most composition operators are similar to those of the operators in standard processes. In the choice process, forward processes make the choices of the compositions. The synchronized parallel composition $(\underset{X}{\parallel})$, hiding($\setminus$), and renaming ($[\![\rho]\!]$) affect both the forward processes and the compensation processes. $PP \boxtimes QQ$ represents a speculative choice that executes PP and QQ in parallel until one of them succeeds, then the other one will be compensated. Similar to standard processes, there are three primitive compensable processes: **skipp** is a special compensation pair that both the forward process and the compensation process are skip, **throww** will immediately throw an exception, and **yieldd** will yield to an interruption or terminate successfully.

3 Operational Semantics

3.1 Basic Notations

Let $\Omega=\{\checkmark,!,?\}$ be disjoint with Σ. Events in Ω are called *terminals* and they indicate different terminating scenarios: "$\checkmark$" indicates that the process terminates successfully, "!" means that the process terminates with an occurrence of an exception, and "?" represents that the process terminates by yielding to an interruption from environment. In addition, we introduce a special event $\tau \notin \Sigma \cup \Omega$, called *invisible* event, and τ is invisible to the outside of any process. We use A^{τ} as the shorthand for $A \cup \{\tau\}$. Thus, the processes of the extended cCSP can perform the events in Γ^{τ}, where $\Gamma = \Sigma \cup \Omega$.

Let A^* denote the set of *finite* sequences of the elements in a set A of symbols, $s{\cdot}t$ the *concatenation* of sequences s and t, $s \setminus a$ the resulting sequences after removing each occurrence of a from s, and $T_1{\cdot}T_2$ the set of concatenated sequences of the sequence sets T_1 and T_2. In particular, for a *non-empty* set B, let $A^{\bigstar}_B{=}A^*{\cdot}B$ denote the set of the sequences of A terminated with an element in B, and let $A^{\circledast}_B{=}A^* \cup A^{\bigstar}_B$. We use $A^{\bigstar}$ and $A^{\circledast}$ as the shorthands of $A^{\bigstar}_{\Omega}$ and $A^{\circledast}_{\Omega}$. In particular, Σ^* is the set of *interaction traces* of the extended cCSP processes, and $\Sigma^{\bigstar}$ is the set of *terminated traces* of the extended cCSP. We use ε to denote the empty trace. For the sake of brevity, in the following, we use a to represent the event, the process that performs a and then successfully terminates, or the trace of the single event a, depending on its context.

There are rules for the processes in the extended cCSP to follow to synchronize on different terminals. We order the three terminals such that $! \prec ? \prec \checkmark$, define $\omega_1 \parallel \omega_2 = \omega_1$ if $\omega_1 \preceq \omega_2$, and $\omega_1 \parallel \omega_2 = \omega_2 \parallel \omega_1$. Therefore, the synchronization of any terminal with an exception will result in an exception, and a composite process terminates successfully *iff* both parties do.

3.2 Semantics of Standard Processes

In this subsection, we give the *small step* operational semantics for standard processes. The semantics decribes how a standard process evolves to another process by performing an event. We introduce a special standard process 0, which represents the *null process* that cannot evolve anymore. For a standard process P, if P can preform a terminal event $\omega \in \Omega$, then P evolves to 0 and finishes, *i.e.*, $P \xrightarrow{\omega} 0$. The hiding and renaming operators do not have any effect on the null process, *i.e.*, $0 \setminus X = 0$ and $0[\![\rho]\!] = 0$. If P is not the null process 0, it will continue to evolve except when a deadlock happens.

In the following semantic definitions, we use process P' to represent the new process of P after performing an event. The definitions for atomic and basic processes, sequential composition and exception handling are basically the same as those in [6].

Atomic and Basic Processes. The atomic event process a performs the event a and evolves to the process **skip**. The primitive processes **skip** and **throw** will perform the terminal events ✓ and ! and become the null process, respectively. For defining **yield**, we introduce **interp** that represents a process that is interrupted and terminates. The process **yield** will continue or be interrupted in a non-deterministic style.

$$a \xrightarrow{a} \mathbf{skip} \quad \mathbf{skip} \xrightarrow{\checkmark} 0 \quad \mathbf{throw} \xrightarrow{!} 0 \quad \mathbf{interp} \xrightarrow{?} 0 \quad \mathbf{yield} \xrightarrow{\tau} \mathbf{skip} \quad \mathbf{yield} \xrightarrow{\tau} \mathbf{interp}$$

Choices. If P and Q are standard processes, the internal choice $P \sqcap Q$ behaves either like P or Q, the selection depends on the actual execution of the process and is non-deterministic. We use τ to make this choice. For an external choice $P \Box Q$, if any sub-process can perform an event, the composite process can perform the event.

$$P \sqcap Q \xrightarrow{\tau} P \qquad P \sqcap Q \xrightarrow{\tau} Q \qquad \frac{P \xrightarrow{e} P'}{P \Box Q \xrightarrow{e} P'}\ (e \in \Gamma^{\tau}) \qquad \frac{Q \xrightarrow{e} Q'}{P \Box Q \xrightarrow{e} Q'}\ (e \in \Gamma^{\tau})$$

Sequential Composition. In a sequential composition $P; Q$, if P terminates successfully, Q can continue to evolve; otherwise, the whole process evolves as same as P.

$$\frac{P \xrightarrow{e} P'}{P; Q \xrightarrow{e} P'; Q}\ (e \in \Sigma^{\tau}) \qquad \frac{P \xrightarrow{\checkmark} 0 \wedge Q \xrightarrow{e} Q'}{P; Q \xrightarrow{e} Q'}\ (e \in \Gamma^{\tau}) \qquad \frac{P \xrightarrow{\omega} 0}{P; Q \xrightarrow{\omega} 0}\ (\omega \in \{!, ?\})$$

Exception Handling. The definition of exception handling operator is similar to that of sequential composition, except that Q is enabled if P terminates with an exception.

$$\frac{P \xrightarrow{e} P'}{P \rhd Q \xrightarrow{e} P' \rhd Q}\ (e \in \Sigma^{\tau}) \qquad \frac{P \xrightarrow{!} 0 \wedge Q \xrightarrow{e} Q'}{P \rhd Q \xrightarrow{e} Q'}\ (e \in \Gamma^{\tau}) \qquad \frac{P \xrightarrow{\omega} 0}{P \rhd Q \xrightarrow{\omega} 0}\ (\omega \in \{?, \checkmark\})$$

Parallel Composition. $P \underset{X}{\parallel} Q$ represents the synchronized parallel composition of the processes P and Q. If P or Q performs a non-terminal event that is not in X, the process can perform it independently.

$$\frac{P \xrightarrow{e} P'}{P \underset{X}{\parallel} Q \xrightarrow{e} P' \underset{X}{\parallel} Q}\ (e \in \Sigma^{\tau} \setminus X) \qquad \frac{Q \xrightarrow{e} Q'}{P \underset{X}{\parallel} Q \xrightarrow{e} P \underset{X}{\parallel} Q'}\ (e \in \Sigma^{\tau} \setminus X)$$

For any event in the synchronization event set X, both P and Q need to synchronize on it. In addition, besides the events in X, the sub-processes also need to synchronize on the terminal events.

$$\frac{P \xrightarrow{e} P' \land Q \xrightarrow{e} Q'}{P \underset{X}{\parallel} Q \xrightarrow{e} P' \underset{X}{\parallel} Q'} \ (e \in X) \qquad \frac{P \xrightarrow{\omega_1} 0 \land Q \xrightarrow{\omega_2} 0}{P \underset{X}{\parallel} Q \xrightarrow{\omega_1 \| \omega_2} 0} \ (\omega_1, \omega_2 \in \Omega)$$

Hiding. The hiding operator makes any event in the hiding event set X invisible. Any event not in X, except τ, performed by the hidden process is still visible.

$$\frac{P \xrightarrow{e} P'}{P \setminus X \xrightarrow{\tau} P' \setminus X} \ (e \in X) \qquad \frac{P \xrightarrow{e} P'}{P \setminus X \xrightarrow{e} P' \setminus X} \ (e \in \Gamma^{\tau} \setminus X)$$

Renaming. In a renaming process $P[\![\rho]\!]$, $\rho \subseteq \Sigma \times \Sigma$ is a binary relation, each element (e_1, e_2) of which means the outside of the renaming process will observe e_2 if the process P performs e_1.

$$\frac{P \xrightarrow{e_1} P' \land (e_1, e_2) \in \rho}{P[\![\rho]\!] \xrightarrow{e_2} P'[\![\rho]\!]} \qquad \frac{P \xrightarrow{e} P' \land \nexists e_1 \in \Sigma \bullet (e, e_1) \in \rho}{P[\![\rho]\!] \xrightarrow{e} P'[\![\rho]\!]} \ (e \in \Gamma^{\tau})$$

Recursion. A recursive process $\mu\, p.F(p)$ can replicate itself at each p in $F(p)$. In addition, if $F(p)$ can evolve to a new process $F(p)'$ by performing an event, then the recursive process $\mu\, p.F(p)$ can also perform the event. In the following, $F[a/b]$ means that the free variable b will be substituted by the expression a in the function F.

$$\mu\, p.F(p) \xrightarrow{\tau} F(p)[\mu\, p.F(p)/p] \qquad \frac{F(p) \xrightarrow{e} F(p)'}{\mu\, p.F(p) \xrightarrow{e} F(p)'[\mu\, p.F(p)/p]} \ (e \in \Gamma^{\tau})$$

3.3 Semantics of Compensable Processes

A compensable process also evolves by performing events. The special part is that a compensable process can perform a terminal event $\omega \in \Omega$ to evolve to a standard process, *i.e.*, $PP \xrightarrow{\omega} P$, and the standard process is used for compensation. Because a compensable process is composed by compensation pairs, we define the semantics of a compensation pair first.

Compensation pair. For a compensation pair $P \div Q$, if the forward process P can perform an event, then $P \div Q$ can perform the event. If the forward process can terminate successfully, the compensation pair evolves to the compensation process; otherwise, the compensation process is **skip**, which means that there is no need to compensate the forward process.

$$\frac{P \xrightarrow{e} P'}{P \div Q \xrightarrow{e} P' \div Q} \ (e \in \Sigma^{\tau}) \qquad \frac{P \xrightarrow{\checkmark} 0}{P \div Q \xrightarrow{\checkmark} Q} \qquad \frac{P \xrightarrow{\omega} 0}{P \div Q \xrightarrow{\omega} \mathbf{skip}} \ (\omega \in \{!, ?\})$$

For the basic processes **skipp**, **throww** and **yieldd**, their forward processes are **skip**, **throw** and **yield**, respectively; their compensation processes are all **skip**.

We need to record and compose the compensation process when executing the forward process of a compensable process. Therefore, we introduce *nested configuration* to define the semantics of composite compensable processes. A nested configuration is $\langle C, P\rangle$, where C is a nested configuration *or* a compensable process PP, and P is a standard process. C is current forward behavior that is not terminated, and P is the current compensation process. For the sake of brevity, in the following of this paper, we use PP or QQ to denote both compensable processes and nested configurations. In addition, same as the semantic definitions of standard processes, we also use PP' or QQ' to denote the process or configuration after evolving.

Nested configuration. If the forward behavior can perform a non-terminal event, then the nested configuration can also be transferred by performing the event; otherwise, if the forward behavior can evolve to a standard process, the nested configuration composes the resulting processes with the current compensation process in a reversed order, which implements the backward recovery mechanism in LRTs.

$$\frac{QQ \xrightarrow{e} QQ'}{\langle QQ, P\rangle \xrightarrow{e} \langle QQ', P\rangle}\ (e \in \Sigma^{\tau}) \qquad \frac{QQ \xrightarrow{\omega} Q}{\langle QQ, P\rangle \xrightarrow{\omega} Q; P}\ (\omega \in \Omega)$$

It is necessary to point out that the evolving of a nested configuration will be determined by its innermost nested configuration. For example, $a \div b \xrightarrow{a} \mathbf{skip} \div b$ will make the nested configuration $\langle\langle a \div b, c\rangle, d\rangle$ evolve to $\langle\langle \mathbf{skip} \div b, c\rangle, d\rangle$ after performing the event a, by applying the above first rule *twice*.

Sequential composition. A sequential composition $PP; QQ$ will perform a non-terminal event if PP can perform it; if the forward process of PP terminates non-successfully, QQ will be disregarded and the whole process will evolve to the compensation process of PP.

$$\frac{PP \xrightarrow{e} PP'}{PP; QQ \xrightarrow{e} PP'; QQ}\ (e \in \Sigma^{\tau}) \qquad \frac{PP \xrightarrow{\omega} P}{PP; QQ \xrightarrow{\omega} P}\ (\omega \in \{!, ?\})$$

If the forward process of PP terminates successfully, the compensation process will be recorded by a newly created nested configuration. If the second process can evolve to its compensation process, the compensation processes of PP and QQ will be composed in the reversed order to satisfy the requirement of backward recovery.

$$\frac{PP \xrightarrow{\checkmark} P \wedge QQ \xrightarrow{e} QQ'}{PP; QQ \xrightarrow{e} \langle QQ', P\rangle}\ (e \in \Sigma^{\tau}) \qquad \frac{PP \xrightarrow{\checkmark} P \wedge QQ \xrightarrow{\omega} Q}{PP; QQ \xrightarrow{\omega} Q; P}\ (\omega \in \Omega)$$

Internal and external choices. For an internal choice between PP and QQ, the selection is nondeterministic, and the definition is similar to that of the internal choice of standard processes.

$PP \sqcap QQ \xrightarrow{\tau} PP \qquad PP \sqcap QQ \xrightarrow{\tau} QQ$

An external choice can perform any event that can be performed by any of its sub-processes, and can evolve to the standard process that any of the sub-processes can evolve to.

$$\frac{PP \xrightarrow{e} PP'}{PP \Box QQ \xrightarrow{e} PP'} \ (e \in \Sigma^\tau) \qquad \frac{QQ \xrightarrow{e} QQ'}{PP \Box QQ \xrightarrow{e} QQ'} \ (e \in \Sigma^\tau)$$

$$\frac{PP \xrightarrow{\omega} P}{PP \Box QQ \xrightarrow{\omega} P} \ (\omega \in \Omega) \qquad \frac{QQ \xrightarrow{\omega} Q}{PP \Box QQ \xrightarrow{\omega} Q} \ (\omega \in \Omega)$$

Parallel composition. Any sub-process of $PP \underset{X}{\parallel} QQ$ can perform a non-terminal event independently if the event is not in the synchronization set X.

$$\frac{PP \xrightarrow{e} PP'}{PP \underset{X}{\parallel} QQ \xrightarrow{e} PP' \underset{X}{\parallel} QQ} \ (e \in \Sigma^\tau \backslash X) \qquad \frac{QQ \xrightarrow{e} QQ'}{PP \underset{X}{\parallel} QQ \xrightarrow{e} PP \underset{X}{\parallel} QQ'} \ (e \in \Sigma^\tau \backslash X)$$

On the other hand, PP and QQ need to synchronize on any event in X and the terminal events, and the synchronization affects both the forward behavior and the compensation behavior.

$$\frac{PP \xrightarrow{e} PP' \wedge QQ \xrightarrow{e} QQ'}{PP \underset{X}{\parallel} QQ \xrightarrow{e} PP' \underset{X}{\parallel} QQ'} \ (e \in X) \qquad \frac{PP \xrightarrow{\omega_1} P \wedge QQ \xrightarrow{\omega_2} Q}{PP \underset{X}{\parallel} QQ \xrightarrow{\omega_1 \| \omega_2} P \underset{X}{\parallel} Q} \ (\omega_1, \omega_2 \in \Omega)$$

Hiding and renaming. Hiding and renaming operators affect both the forward and compensation processes of a compensable process. For the forward processes, the rules are as follows, which are similar to those of standard processes.

$$\frac{PP \xrightarrow{e} PP'}{PP \setminus X \xrightarrow{\tau} PP' \setminus X} \ (e \in X) \qquad \frac{PP \xrightarrow{e} PP'}{PP \setminus X \xrightarrow{e} PP' \setminus X} \ (e \in \Sigma^\tau \setminus X)$$

$$\frac{PP \xrightarrow{e_1} PP' \wedge (e_1, e_2) \in \rho}{PP[\![\rho]\!] \xrightarrow{e_2} PP'[\![\rho]\!]} \qquad \frac{PP \xrightarrow{e} PP' \wedge \nexists e_1 \in \Sigma \bullet \in (e, e_1) \in \rho}{PP[\![\rho]\!] \xrightarrow{e} PP'[\![\rho]\!]} \ (e \in \Sigma^\tau)$$

If a compensable process can evolve to its compensation process, the events performed by the compensation process also need to be hidden or renamed.

$$\frac{PP \xrightarrow{\omega} P}{PP \setminus X \xrightarrow{\omega} P \setminus X} \ (\omega \in \Omega) \qquad \frac{PP \xrightarrow{\omega} P}{PP[\![\rho]\!] \xrightarrow{\omega} P[\![\rho]\!]} \ (\omega \in \Omega)$$

Speculative choice. $PP \boxtimes QQ$ is the speculative choice between two compensable processes, it behaves like that PP and QQ run in parallel until one of them succeeds, then the other one will be compensated. Thus, we give the rules for the parallelism in forward behavior first.

$$\frac{PP \xrightarrow{e} PP'}{PP \boxtimes QQ \xrightarrow{e} PP' \boxtimes QQ} \ (e \in \Sigma^\tau) \qquad \frac{QQ \xrightarrow{e} QQ'}{PP \boxtimes QQ \xrightarrow{e} PP \boxtimes QQ'} \ (e \in \Sigma^\tau)$$

If one sub-process succeeds in the forward process, the other one will be compensated by executing its compensation process.

$$\frac{PP \xrightarrow{\checkmark} P \wedge QQ \xrightarrow{\omega} Q}{PP \boxtimes QQ \xrightarrow{\checkmark} \langle Q \div \mathbf{skip}, P\rangle} \ (\omega \in \Omega) \quad \frac{PP \xrightarrow{\omega} P \wedge QQ \xrightarrow{\checkmark} Q}{PP \boxtimes QQ \xrightarrow{\checkmark} \langle P \div \mathbf{skip}, Q\rangle} \ (\omega \in \Omega)$$

However, when all the attempts fail, the speculative choice fails, and their compensations will be accumulated in parallel.

$$\frac{PP \xrightarrow{\omega_1} P \wedge QQ \xrightarrow{\omega_2} Q}{PP \boxtimes QQ \xrightarrow{\omega_1 \| \omega_2} P \parallel Q} \quad (\omega_1, \omega_2 \in \{!, ?\})$$

Recursion. Like standard recursive processes, we also need to unfold recursive compensable process. The evolving of $FF(pp)$ can also make $\mu\ pp.FF(pp)$ evolve. The different part is: when $FF(pp)$ can evolve to the compensation standard process, the recursive process can also evolve to the standard process.

$$\mu\ pp.FF(pp) \xrightarrow{\tau} FF(pp)[\mu\ pp.FF(pp)/pp] \quad \frac{FF(pp) \xrightarrow{\omega} P}{\mu\ pp.FF(pp) \xrightarrow{\omega} P} \quad (\omega \in \Omega)$$

$$\frac{FF(pp) \xrightarrow{e} FF(pp)'}{\mu\ pp.FF(pp) \xrightarrow{e} FF(pp)'[\mu\ pp.FF(pp)/pp]} \quad (e \in \Sigma^\tau)$$

Transaction Block. Based on the semantics of compensable processes, we can define the transaction block in standard processes. If the forward behavior terminates successfully, the compensation will be disregarded, *i.e.*, the LRT is successful; otherwise, if the forward behavior terminates with an exception, the compensation process will be executed.

$$\frac{PP \xrightarrow{e} PP'}{[PP] \xrightarrow{e} [PP']} \ (e \in \Sigma) \quad \frac{PP \xrightarrow{\checkmark} P}{[PP] \xrightarrow{\checkmark} 0} \quad \frac{PP \xrightarrow{!} P \wedge P \xrightarrow{e} P'}{[PP] \xrightarrow{e} P'} \ (e \in \Gamma^\tau)$$

3.4 Correspondence with Failure-Divergence Semantics

The operational semantics defines a transition system for each process in the extended cCSP. We present in [9] an FD semantics for the extended cCSP. Based on the method in [20], we can derive the FD semantics of a standard process or a compensable process from its operational semantics. First, we define the trace transition. For a trace s in $\Sigma^\circledast$, $P \xrightarrow{s} P'$ means there exists $s_1 \in (\Sigma^\tau)^\circledast$ such that $s_1 \setminus \tau = s$, and the process P can perform each event of s_1 in sequence to evolve to P'. In addition, $P \xrightarrow{\varepsilon} P$ is valid for any process P.

The FD semantics of a standard process P is $(\mathcal{F}(P), \mathcal{D}(P))$, where $\mathcal{F}(P) \subseteq \Sigma^\circledast \times \mathbb{P}(\Gamma)$ is the failure set each element (s, X) of which means P refuses to perform any event in X after executing the trace s, and $\mathcal{D}(P)$ is the divergence trace set after executing each element of which P diverges. Based on the trace transition definition, for a standard process P, we can derive its FD semantics as follows, where $D(P)$ and $F(P)$ are the derived divergence set and failure set of P, respectively.

$$\begin{aligned} D(P) = \;& \{s{\cdot}t \mid s \in \Sigma^* \wedge t \in \Sigma^{\circledast} \wedge \exists P' \in \mathcal{P} \bullet P \xrightarrow{s} P' \wedge div(P')\} \\ F(P) = \;& \{(s, X) \mid \exists P' \in \mathcal{P} \bullet P \xrightarrow{s} P' \wedge X \subseteq ref(P') \wedge \Omega \subseteq ref(P')\} \\ & \cup \{(s, X) \mid \exists P' \in \mathcal{P} \bullet P \xrightarrow{s\cdot\omega} P' \wedge X \subseteq \Gamma \setminus \{\omega\} \wedge s{\cdot}\omega \in \Sigma^{\bigstar}\} \\ & \cup \{(s, X) \mid \exists P' \in \mathcal{P} \bullet P \xrightarrow{s} P' \wedge X \subseteq \Gamma \wedge s \in \Sigma^{\bigstar}\} \\ & \cup \{(s, X) \mid s \in D(P) \wedge X \subseteq \Gamma\} \end{aligned}$$

where $div(P')$ means there exists $P_0 = P', P_1, P_2, ...$, such that $P_i \xrightarrow{\tau} P_{i+1}$ for each $i \in \mathbb{N}$, and $ref(P) = \{a \mid a \in \Gamma \wedge \nexists P' \in \mathcal{P} \bullet P \xrightarrow{a} P'\}$ is the set of events P refuses to perform. Following theorem ensures that the correspondence relation between the operational semantics in this paper and the denotational semantics in [9] holds for standard processes.

Theorem 1. *For a standard process P, the derived model $(F(P), D(P))$ according to the operational semantics is equal to the FD model $(\mathcal{F}(P), \mathcal{D}(P))$ of P.*

Proof. The basic idea of proof is to induce the structures of standard processes. First, we prove that the correspondence relation holds for atomic and basic processes. For the atomic event process a, according to the above definition and the operational semantics of a, $D(a)$ is $\{\}$, and $F(a)$ is as follows:

$$F(a) = \{(\varepsilon, X) \mid X \subseteq \Gamma \setminus \{a\}\} \cup \{(a, X) \mid X \subseteq \Gamma \setminus \{\checkmark\}\} \cup \{(a\checkmark, X) \mid X \subseteq \Gamma\}$$

According to the definition in [9], the divergence set of a is also $\{\}$, and $\mathcal{F}(a)$ is equal to $F(a)$. Thus, the correspondence relation holds for the atomic event process a. The proofs for the basic processes including **skip**, **throw**, **interp** and **yield** are similar.

For a composite process $P \oplus Q$, where P and Q are standard processes and $\oplus$ is a composition operator, we need to prove that the correspondence relation holds for $P \oplus Q$ with the assumption that the relation holds for both P and Q. For the sake of space, we only show the proof for sequential composition, and the other operators can be proved in a same manner.

For a sequential process $P \,;\, Q$, we need to prove the following two equations: $D(P \,;\, Q) = \mathcal{D}(P \,;\, Q)$ and $F(P \,;\, Q) = \mathcal{F}(P \,;\, Q)$. For the first equation $D(P \,;\, Q) = \mathcal{D}(P \,;\, Q)$, if $s \in D(P \,;\, Q)$, according to the preceding derivation method, there exists s_1 and t_1 such that $s = s_1{\cdot}t_1$, where $s_1 \in \Sigma^*$, $t_1 \in \Sigma^{\circledast}$, $\exists P' \in \mathcal{P} \bullet P;Q \xrightarrow{s_1} P'$ and $div(P')$. According to the operational semantics of $P \,;\, Q$ in Section 3.2, there are following two cases for $P \,;\, Q \xrightarrow{s_1} P'$:

- $P \xrightarrow{s_1} P'$ and $s_1 \in \Sigma^*$, which means P diverges after executing s_1. Therefore, $s_1 \in D(P)$, which implies $s_1 \in \mathcal{D}(P)$ by the assumption. According to the definition in [9], *i.e.*, $\mathcal{D}(P \,;\, Q) = \mathcal{D}(P) \cup \{s{\cdot}t \mid s\checkmark \in \mathit{traces}(P) \wedge t \in \mathcal{D}(Q)\}$, where $\mathit{traces}(P)$ is the set of traces that P can execute, $s_1 \in \mathcal{D}(P \,;\, Q)$ is valid.
- $\exists s_2\checkmark \in \Sigma^{\bigstar}, s_3 \in \Sigma^* \bullet P \xrightarrow{s_2\checkmark} 0 \wedge Q \xrightarrow{s_3} P' \wedge s_1 = s_2{\cdot}s_3$, which means P terminates successfully without a divergence, and Q diverges after executing s_3.

Thus, same with the first case, we can have $s_3 \in \mathcal{D}(Q)$. In addition, because $s_2\checkmark \in \mathit{traces}(P)$ is valid, we can have $s_1 \in \mathcal{D}(P \;;\; Q)$ according to the definition of $\mathcal{D}(P \;;\; Q)$.

In total, we can have $s_1 \in \mathcal{D}(P \;;\; Q)$, which implies $s \in \mathcal{D}(P \;;\; Q)$, because a divergence set is suffix closed. Thus, we have proved $D(P \;;\; Q) \subseteq \mathcal{D}(P \;;\; Q)$. On the other side, if $s \in \mathcal{D}(P \;;\; Q)$, there are also two cases:

– $s \in \mathcal{D}(P)$, which implies $s \in D(P)$. According to the derivation method, the following result is valid: $D(P) \subseteq D(P \;;\; Q)$, so we can have $s \in D(P \;;\; Q)$.
– $\exists s_1\checkmark \in \mathit{traces}(P), s_2 \in \mathcal{D}(Q) \bullet s = s_1 \cdot s_2$. Thus, $s_2 \in D(Q)$, which means there exist Q', s_3 and s_4 such that $Q \xrightarrow{s_3} Q'$, Q' diverges and $s_2 = s_3 \cdot s_4$. According to the operational semantics of $P \;;\; Q$, we can have that $P \;;\; Q$ diverges after executing $s_1 \cdot s_3$. Therefore, $s \in D(P \;;\; Q)$.

Hence, we can have $s \in D(P \;;\; Q)$ that implies $\mathcal{D}(P \;;\; Q) \subseteq D(P \;;\; Q)$. In the result of the above proofs, $D(P \;;\; Q) = \mathcal{D}(P \;;\; Q)$ is proved. In a same way, we can prove the equality between failure sets, *i.e.*, $F(P \;;\; Q) = \mathcal{F}(P \;;\; Q)$. In total, we have proved that the correspondence relation holds for sequential compositions. □

It is necessary to note that Theorem 1 does not consider the transaction block processes, which need the derivation method for compensable processes to prove the correspondence relation. The FD semantics of a compensable process PP is defined to be $(\mathcal{F}_f(PP), \mathcal{D}_f(PP), \mathcal{F}_c(PP), \mathcal{D}_c(PP))$, where $\mathcal{F}_f(PP)$ and $\mathcal{D}_f(PP)$ are the failure and divergence sets of the forward behavior of PP, $\mathcal{F}_c(PP)$ and $\mathcal{D}_c(PP)$ are the compensation failure and compensation divergence sets of PP. An element (s, s_1, X) in $\mathcal{F}_c(PP)$ and an element (s, s_1) in $\mathcal{D}_c(PP)$ records a standard failure and a divergence of the compensation behavior for the forward terminated trace s. Based on the derivation of standard processes, for a compensable process PP, its failure divergence semantics can be derived as follows, where $D_f(PP)$ and $F_f(PP)$ are the derived forward divergence and failure sets, $D_c(PP)$ and $F_c(PP)$ are the derived compensation divergence and failure sets. Furthermore, the definitions of trace transition, *div* and *ref* are extended for compensable processes and nested configurations.

$$\begin{aligned}
D_f(PP) = \; & \{s \cdot t \mid s \in \Sigma^* \wedge t \in \Sigma^{\circledast} \wedge \exists PP' \in \mathcal{PP} \bullet PP \overset{s}{\rightarrow} PP' \wedge div(PP')\} \\
F_f(PP) = \; & \{(s, X) \mid \exists PP' \in \mathcal{PP} \bullet PP \overset{s}{\rightarrow} PP' \wedge X \subseteq ref(PP') \wedge \Omega \subseteq ref(PP')\} \\
& \cup \{(s, X) \mid \exists P \in \mathcal{P} \bullet PP \xrightarrow{s \cdot \omega} P \wedge X \subseteq \Gamma \setminus \{\omega\} \wedge s \cdot \omega \in \Sigma^{\bigstar}\} \\
& \cup \{(s, X) \mid \exists P \in \mathcal{P} \bullet PP \overset{s}{\rightarrow} P \wedge X \subseteq \Gamma \wedge s \in \Sigma^{\bigstar}\} \\
& \cup \{(s, X) \mid s \in D_f(PP) \wedge X \subseteq \Gamma\} \\
D_c(PP) = \; & \{(s, t) \mid \exists P \in \mathcal{P} \bullet PP \overset{s}{\rightarrow} P \wedge t \in D(P)\} \\
F_c(PP) = \; & \{(s, t, X) \mid \exists P \in \mathcal{P} \bullet PP \overset{s}{\rightarrow} P \wedge (t, X) \in F(P)\}
\end{aligned}$$

where $PP \overset{s}{\rightarrow} P$ means there exists $s_1 \in (\Sigma^{\tau})^{\circledast}$ such that $s_1 \setminus \tau = s$, and the compensable process PP can perform each event of s_1 in sequence to evolve to its compensation process P. Same as standard processes, $PP \overset{\varepsilon}{\rightarrow} PP$ is valid for any compensable process.

Theorem 2. *For a compensable process PP, the derived model $(F_f(PP), D_f(PP), F_c(PP), D_c(PP))$ according to the operational semantics is equal to the FD model $(\mathcal{F}_f(PP), \mathcal{D}_f(PP), \mathcal{F}_c(PP), \mathcal{D}_c(PP))$ of PP.*

Proof. The way of proving the correspondence relation for compensable processes is basically the same as that for standard processes. We first prove that the relation holds for compensation pair. For a compensation pair $P \div Q$, according to the preceding derivation method and the operational semantics of $P \div Q$, $D_f(P \div Q)$ is equal to $D(P)$. In [9], the forward divergence set of $P \div Q$ is defined to be $\mathcal{D}(P)$. Based on the assumption, *i.e.*, $D(P) = \mathcal{D}(P)$, we can have $D_f(P \div Q) = \mathcal{D}_f(P \div Q)$. Similarly, we can prove $F_f(P \div Q) = \mathcal{F}_f(P \div Q)$ based on the fact that $F_f(P \div Q)$ is equal to $F(P)$. For the compensation divergence set, according to the operational semantics, there are two cases that a compensation pair can evolve to a standard process: $P \div Q \xrightarrow{s\checkmark} Q$, which means the forward process P terminates successfully; $P \div Q \xrightarrow{s\omega} \textbf{skip}$ and $\omega \in \Omega \setminus \{\checkmark\}$, which mean P terminates non-successfully. Thus, the derived compensation divergence set, *i.e.*, $D_c(P \div Q)$, is $\{(s\checkmark, t) \mid P{\div}Q \xrightarrow{s\checkmark} Q \wedge t \in D(Q)\} \cup \{(s\omega, t) \mid P{\div}Q \xrightarrow{s\omega} \textbf{skip} \wedge t \in D(\textbf{skip})\}$, the first part is equal to $(\mathit{trace}_t(P) \cap \Sigma^*_{\{\checkmark\}}) \times \mathcal{D}(Q)$, where $\mathit{trace}_t(P)$ is the terminated traces that P can execute, and the second part is equal to the following set: $(\mathit{trace}_t(P) \cap \Sigma^*_{\{?,!\}}) \times \mathcal{D}(\textbf{skip})$. According to the definition of $P \div Q$ in [9], we can conclude that $D_c(P \div Q)$ is equal to $\mathcal{D}_c(P \div Q)$. The equality between $F_c(P \div Q)$ and $\mathcal{F}_c(P \div Q)$ can be proved in a same way. In total, the correspondence relation holds for any compensation pair process.

Based on the proof for compensation pairs, we can induce on the structures of composite compensable process or nested configurations to prove the validity of the correspondence relation. The basic way is to discuss each case of the operational semantics of each operator to prove that each element in the derived model also belongs to the FD model; on the other hand, each case in the FD semantic definition of each operator in [9] can also be discussed to prove that each element in the FD model also belongs to the derived model. The way of proving is basically the same as that for composite standard processes in the proof of Theorem 1. The detailed proof is omitted due to the space limit. □

Based on Theorem 2, we can immediately get that the corresponding relation holds for transaction block processes.

4 Model Checking and Implementation

With respect to the operational semantics, we can derive the transition system of an LRT specified by the extended cCSP. Thus, we can use model checking techniques to check whether some critical properties hold for the LRT. In the following of this section, we first study the model checking problem of the extended cCSP, then we present the implementation of an animator and a prototype model checker for the extended cCSP.

4.1 Regular Property Model Checking of Extended cCSP

Given a standard process P and a finite state machine (FSM) R for a regular property M, the model checking problem of $P \models M$ is basically to check whether $L(T(P)) \subseteq L(R)$ holds, where $L(T_S)$ denotes the language accepted by the transition system T_S, and $T(P)$ is the transition system of P generated with respect to the operational semantics in Section 3. We have proved that this problem is undecidable in general in the following theorem.

Theorem 3. *Given a standard process P of the extended cCSP and an FSM R, the language inclusion problem $L(T(P)) \subseteq L(R)$ is undecidable.*

Proof. Our proof is inspired by the reduction method in [13][10]. The problem is reduced to the halting problem of Minsky 2-counter machine that is known to be undecidable [17]. The basic idea of the reduction is to construct a standard process P and an FSM R for a 2-counter machine C. P models the behavior of C with respect the memory constraint but without regard to the control constraint. R models the control behavior of C but disregards the memory constraint of C, and accepts all the *non-halted* traces of C. Therefore, $L(T(P)) \subseteq L(R)$ *iff* C does not halt can be proved. Thus, we can conclude the problem is undecidable in general. The details of the prove is provided in the Appendix. □

4.2 Animator and Prototype Model Checker

We have implemented our operational semantics of the extended cCSP on Process Analysis Toolkit (PAT) [21], which is a platform to develop the tools for modeling, simulation and model checking of different types of systems. PAT separates a model checker into different parts, such as modeling, animating and verification, and encapsulates each part as a module to be easily extended. The different parts are connected by the transition system of the system that needs modeling and verification. Users can quickly develop a prototype model checker and an animator for their own language by implementing the syntax and the operational semantics.

Based the operational semantics, we have built an animator and a prototype model checker on PAT for the extended cCSP. Our tool can verify the critical properties of the models in the extended cCSP, including deadlock-free, reachability, linear temporal logic (LTL) properties, *etc.* In addition, the tool also supports refinement checking for standard processes, in which a transaction block process specifies an LRT. Although the model checking problem is proved to be undecidable in general, our model checker works well for finite models. Besides, users can use the animator to play complex models specified in the extended cCSP. To the best of our knowledge, our tool is the first one that supports both LTL model checking and refinement checking for LRT models. The prototype tool and some case study examples are available at `http://rcos.iist.unu.edu/~zbchen/LRT.html`.

5 Case Study

We use the case study of an online travel agency to demonstrate the tool. Usually, for a travel agency providing Web Services for booking air tickets, reserving hotel rooms and renting cars, it will use the services provided by its business partners, such as Airlines, Hotels, Car Rental Centers and Banks. The main business process is as follows. After receiving a request, the Agency carries out the air ticket booking, hotel reservation, car rental process and payment in parallel. Usually, the Agency has to repeatedly request the car rental service at the destination, until getting a car. If all the steps succeed, the Agency replies to the client with the booking information and waits for the confirmation. Then, after being confirmed, the Agency sends all the details of the booking and payment to the client. If one exception occurs in any of the above steps, such as the client cancels the request, the whole process fails and needs to be recovered by compensating the successful steps before, *e.g.*, canceling the air ticket, and the Agency sends an apology letter to the client.

5.1 Specification

The whole process involves five parties, each of which is specified by a compensable process. The processes AGENCY, AIR, CAR, HOTEL and BANK are as follows.

Travel Agency. After receiving a request, the Travel Agency will reserve the hotel room, book the flight ticket, arrange a car and do the payment for the client (RESPAY). Then, if the client agrees to what the Agency reserves (CFM), the Agency will complete the process by sending the final result to the client.

$$\begin{aligned}
\textsc{Agency} &= (\texttt{reqTravel} \div \texttt{letter})\ ;\ \textsc{ResPay}\ ;\ \textsc{Cfm}\ ;\ \texttt{result} \div \texttt{skip}\\
\textsc{ResPay} &= (\texttt{reqHotel};(\texttt{okRoom}\Box(\texttt{noRoom};\texttt{throw})))\div\texttt{cancelHotel}\|\\
&\quad\ (\texttt{bookAir};(\texttt{okAir}\Box(\texttt{noAir};\texttt{throw})))\div\texttt{cancelAir}\|\\
&\quad\ \mu\, pp.(\texttt{reqCar}\div\texttt{skip};(((\texttt{noCar}\div\texttt{skip});pp)\Box(\texttt{hasCar}\div\texttt{cancelCar})))\|\\
&\quad\ (\ (\texttt{checkCredit}\ ;\ (\texttt{valid}\ \Box\ (\texttt{inValid};\texttt{throw})))\div\texttt{skip};\\
&\quad\ \ \ (\texttt{payment}\div\texttt{refund});(\texttt{pValid}\ \Box\ (\texttt{pInValid};\texttt{throw}))\div\texttt{skip}\ );\\
\textsc{Cfm} &= (\texttt{sendConfirm}\ ;\ (\texttt{agree}\ \Box\ (\texttt{disAgree};\texttt{throw})))\div\texttt{skip}
\end{aligned}$$

The remaining parties. The hotel service responds each request with whether there is a room. The airline company receives requests and then responses whether there are rest tickets. The car rental service is usually requested multiple times to rent a car. The bank service first checks the credit card, and then finalizes the payment.

$$\begin{array}{ll}
\textsc{Hotel} & = \texttt{reqHotel}\div\texttt{skip};(\texttt{okRoom}\div\texttt{cancelHotel}\sqcap(\texttt{noRoom}\div\texttt{skip};\texttt{throww})) \\
\textsc{Air} & = \texttt{bookAir}\div\texttt{skip};(\texttt{okAir}\div\texttt{cancelAir}\sqcap(\texttt{noAir}\div\texttt{skip};\texttt{throww})) \\
\textsc{Car} & = \mu\ pp.(\texttt{reqCar}\div\texttt{skip};((\texttt{noCar}\div\texttt{skip});pp)\sqcap(\texttt{hasCar}\div\texttt{cancelCar}))) \\
\textsc{Bank} & = (\texttt{checkCredit};(\texttt{valid}\sqcap(\texttt{inValid};\texttt{throw})))\div\texttt{skip}\ ; \\
 & \quad (\texttt{payment}\div\texttt{refund});(\texttt{pValid}\sqcap(\texttt{pInValid};\texttt{throw}))\div\texttt{skip}
\end{array}$$

The global business process (GBP) is the transaction block of the synchronized parallel composition of the above five processes.

$$\begin{array}{ll}
\text{GBP} & = [((((\textsc{Agency} \underset{X_1}{\parallel} \textsc{Hotel}) \underset{X_2}{\parallel} \textsc{Air}) \underset{X_3}{\parallel} \textsc{Car}) \underset{X_4}{\parallel} \textsc{Bank})], \\
X_1 & = \{\texttt{reqHotel}, \texttt{okRoom}, \texttt{noRoom}, \texttt{cancelHotel}\} \\
X_2 & = \{\texttt{bookAir}, \texttt{okAir}, \texttt{noAir}, \texttt{cancelAir}\} \\
X_3 & = \{\texttt{reqCar}, \texttt{noCar}, \texttt{hasCar}, \texttt{cancelCar}\} \\
X_4 & = \{\texttt{checkCredit},\texttt{valid},\texttt{inValid},\texttt{payment},\texttt{refund},\texttt{pValid},\texttt{pInValid}\}
\end{array}$$

5.2 Verification Results

Using our tool, we have verified the process GBP with respect to different types of properties. Table 1 lists the verification results. All the experiments were conducted on a laptop with 2G memory and one 2.67GHz Intel i5 CPU, and the timeout threshold for verification is set to be two minutes.

Table 1: Verification results of the case study

	Property	Result	Time	Memory (MB)
1	GBP $\models \Box(!\ \texttt{hasCar}\ \mathbf{U}\ \texttt{okAir})$	False	18.43(ms)	28.33
2	GBP $\models \Box(!\ \texttt{noCar}\ \mathbf{U}\ \texttt{sendConfirm})$	False	14.38(ms)	28.33
3	GBP $\models (!\ \texttt{letter}\ \mathbf{U}\ \texttt{cancelCar}) \vee (\Box\ !\ \texttt{letter})$	True	34.06(s)	118.13
4	GBP $\models \texttt{cancelAir}\ \mathbf{R}\ !\ \texttt{letter}$	False	6.66(ms)	8.84
5	GBP $\models \texttt{agree}\ \mathbf{R}\ !\ \texttt{result}$	True	32.78(s)	12.71
6	GBP **reaches** hasCar	True	29.14(ms)	8.87
7	[PCar] **refines**$\langle$**FD**$\rangle$ [Car]	True	18.24(ms)	23.59
8	GBP **deadlockfree**	True	15.36(s)	12.86
9	GBP **divergencefree**	True	66.41(s)	9.85
10	[Car $\setminus \{\texttt{reqCar}, \texttt{noCar}\}$] **divergencefree**	False	9.16(ms)	8.58

In Table 1, all of the ten properties can be successfully verified. The first five properties are more business related, and they are explained as follows: 1) the first property is an LTL property, which means getting the airline ticket will happen, and the successful car renting cannot happen before getting the airline ticket, but this property is not valid, because car renting and airline ticket booking are carried out in parallel or airline ticket booking may fail; 2) the second property means asking the client for confirmation will happen and no available car cannot happen before asking the client to confirm the booking, which is not valid either, because it may need to request the car service multiple times to get a car or the process before the confirmation may fail; 3) the third property is about compensation behaviour, which means that sending apology letter cannot happen before cancelling the car in case that the global process fails, and the validity of the property implies that the business process satisfies the backward recovery requirement; 4) the fourth property is also about compensation behaviour, which requires sending letter cannot happen before cancelling the airline ticket, but the property is not valid, because the failure of the global process may be caused by the unavailability of the airline tickets; 5) the fifth property is also an LTL property, which means the final booking result delivery cannot happen before the client agreement, and the property is satisfied by the global process.

The rest properties are more related to reachability, refinement or concurrent features. The sixth property is a reachability property, which means the `hasCar` event is reachable by the global process GBP. The seventh property is an FD refinement property, and $\text{PC}{\small\text{AR}} = \texttt{reqCar} \div \texttt{skip}; \texttt{hasCar} \div \texttt{cancelCar}$, which is the model for a perfect car rental service, and $[\text{PC}{\small\text{AR}}]$ refines $[\text{C}{\small\text{AR}}]$ with respect to the FD refinement in [9]. The eighth property means the global process is deadlock free, which is also valid and can be verified successfully. The divergence free of the global process is specified by the ninth property, and the property is verified successfully to be true. The last property is also a divergence-free property, which means the transaction block of the CAR process by hiding the `reqCar` and `noCar` is divergence-free, but the process can diverge at the beginning.

6 Related Work

Formal modeling and verification of LRTs attracts much attention recently. There are many formalisms proposed for LRT modeling. Most of the formalisms are extensions to process algebra. These existing work differs in the expressiveness and the supported policies of LRTs. Many of them have an operational semantics, including StAC [4], the original cCSP [6], SAGAs calculi [3], π-calculus extended formalisms [2][14], *etc.* StAC is provided with a small-step operational semantics in [4], which supports a more flexible indexed compensation mechanism, but without considering the verification problem. SAGAs calculi in [3] have a big-step operational semantics, and support more flexible interruption and compensation policies in modeling LRTs, but do not support deadlock or livelock modeling. Most of these work lacks of tool support for modeling and verification, which brings difficulties to the application of the theories. In [23], an asynchronous polyadic π-calculus $\mathtt{dc}\pi$++ is proposed, and the operational semantics is implemented in Prolog and connected with the ProB model checker for LTL model checking, but no animating or refinement checking is supported by their tool.

For the operational semantics in [6] for the original cCSP, we find the small-step operational semantics in [6] is not complete, especially for compensable processes. Actually, more transition rules of the construct $\langle QQ, P\rangle$ used in the semantic interpretation of compensable processes are needed. For example, for the compensable process $a_1 \div b_1; (\mathbf{skip} \div b_2; a_3 \div b_3)$, with respect to the operational semantics in [6], it evolves to $\mathbf{skip} \div b_1; (\mathbf{skip} \div b_2; a_3 \div b_3)$ after performing a_1. However, the resulting process cannot evolve anymore. The reason is $\mathbf{skip} \div b_2; a_3 \div b_3$ can evolve to $\langle \mathbf{skip} \div b_3, b_2\rangle$ after performing a_3, but there is no rule in [6] for sequential composition taking into account this situation, *i.e.*, the second subprocess evolving from a compensable process to a construct. In this paper, we solve this problem by using nested configurations and taking into account nested configurations in the transition rules (*c.f.* Section 3.3). In addition, our operational semantics supports the verification of more properties, such as deadlock-free and divergence-free.

7 Conclusion and Future Work

LRTs are widely used in SOC to improve the consistency in service coordinations. This paper presents an operational semantics for modeling and verifying the LRTs specified in the extended cCSP. The semantics provides a theoretical foundation for implementing and model checking the extended cCSP. Furthermore, the derivation of the FD semantics from the operational semantics is provided, which serves as the basis for justifying the correctness of the FD semantics. Based on the operational semantics, we have implemented an animator and a prototype model checker, which supports verifying different kinds of properties, such as deadlock-free, LTL properties and refinement. In addition, we have investigated the model checking problem of the extended cCSP with respect to regular properties, and proved the problem to be undecidable.

The future work lies in several aspects. For some case studies, especially for the models within which many parallel compositions exist, using our tool will generate a huge state space when carrying out some analysis tasks, such as deadlock free analysis. Therefore, we will try to use the algebraic laws in [9] to improve the efficiency of the verification. Because the model checking problem with respect to regular properties is undecidable, more efficient model checking algorithms especially for the extended cCSP are needed, and tree-automata based model checking [16] may be a direction. In addition, more applications for using the extended cCSP and the tool are appreciated.

Acknowledgments. This research is supported in part by grants from the National NSFC projects (Nos. 61103013 and 61120106006), the National 973 project 2014CB340703, and the Specialized Research Fund for the Doctoral Program of Higher Education 20114307120015. The authors would like to thank the anonymous reviewers for their suggestions that help to improve the paper. Furthermore, we would like to appreciate the help from Yang Liu and Manchun Zheng in PAT group during our tool development.

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. L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In *Proc. FMOODS 2003, LNCS 2884*, pages 124–138. Springer, 2003.
3. R. Bruni, H. C. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *Proc. POPL 2005*, pages 209–220. ACM Press, 2005.
4. M. J. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *Proc. COORDINATION 2004, LNCS 2949*, pages 87–104. Springer, 2004.
5. M. J. Butler, C. A. R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes, LNCS 3525*, pages 133–150. Springer, 2004.
6. M. J. Butler and S. Ripon. Executable semantics for compensating CSP. In *Proc. WS-FM 2005, LNCS 3670*, pages 243–256. Springer, 2005.
7. Z. Chen and Z. Liu. An extended cCSP with stable failures semantics. In *Proc. ICTAC, LNCS 6255*, pages 121–136. Springer, 2010.
8. Z. Chen, Z. Liu, and J. Wang. Failure-divergence refinement of compensating communicating processes. In *Proc. FM 2011, LNCS 6664*, pages 262–277, 2011.
9. Z. Chen, Z. Liu, and J. Wang. Failure-divergence semantics and refinement of long running transactions. *Theor. Comput. Sci.*, 455:31–65, 2012.
10. M. Emmi and R. Majumdar. Verifying compensating transactions. In *Proc. VMCAI 2007, LNCS 4349*, pages 29–43, 2007.
11. H. Garcia-Molina and K. Salem. SAGAS. In *Proc. SIGMOD 1987*, pages 249–259. ACM Press, 1987.
12. J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, 1993.
13. A. Kucera and R. Mayr. Simulation preorder over simple process algebras. *Inf. Comput.*, 173(2):184–198, 2002.
14. C. Laneve and G. Zavattaro. Foundations of web transactions. In *Proc. FoSSaCS 2005, LNCS 3441*, pages 282–298, 2005.
15. M. C. Little. Transactions and web services. *Commun. ACM*, 46(10):49–54, 2003.
16. D. Lugiez and P. Schnoebelen. The regular viewpoint on pa-processes. *Theor. Comput. Sci.*, 274(1-2):89–115, 2002.
17. M. L. Minsky. *Computation: Finite and infinite machines*. Prentice-Hall, Englewood Cliffs, New Jersey, 1967.

18. M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *IEEE Computer*, 40(11):38–45, 2007.
19. G. Ramalingam and K. Vaswani. Fault tolerance via idempotence. In *Proc. POPL 2013*, pages 249–262. ACM Press, 2013.
20. A. W. Roscoe. *The theory and practice of concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
21. J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards flexible verification under fairness. In *Proc. CAV 2009, LNCS 5643*, pages 709–714, 2009.
22. S. Thatte. XLANG web services for business process design, 2001.
23. C. Vaz and C. Ferreira. On the analysis of compensation correctness. *J. Log. Algebr. Program.*, 81(5):585–605, 2012.

Appendix

Proof of Theorem 3

Theorem 3. *Given a standard process P of the extended cCSP and an FSM R, the language inclusion problem $L(T(P)) \subseteq L(R)$ is undecidable.*

Proof. The problem can be reduced to the halting problem of Minsky 2-counter machine that is known to be undecidable [17]. The basic idea of the reduction is to construct a standard process P and an FSM R for a 2-counter machine M. P models the behavior of M with respect the memory constraint but without regard to the control constraint of M. R models the control behavior of M but disregards the memory constraint, and accepts all the traces of M that are not halted. Therefore, $L(T(P)) \subseteq L(R)$ *iff* M does not halt, which implies the problem is undecidable in general.

Hence, we need to give how to construct the process and the FSM. Let M be a 2-counter machine with n numbered instructions:

$$\langle 1 : ins_1 \rangle \ \langle 2 : ins_2 \rangle \ ...... \ \langle n-1 : ins_{n-1} \rangle \ \langle n : halt \rangle$$

where $ins_i \in \{(c_k{:=}c_k + 1; \mathbf{goto}\ j), (\mathbf{if}\ (c_k = 0)\ \mathbf{goto}\ j; (c_k{:=}c_k - 1; \mathbf{goto}\ l))\}$, $k \in \{1, 2\}$, $1 \leq j, l \leq n$ and $1 \leq i \leq n-1$.

The construction of the extended cCSP process P is as follows, where $k \in \{1, 2\}$.

$$M_k = (inc_k \div [M_k] \ ; \ M_k) \Box ((dec_k; \mathbf{throw}) \div \mathbf{skip})$$
$$C_k = [M_k]$$
$$Z_k = [(zero_k \div \mathbf{skip} \ ; \ Z_k \div \mathbf{skip}) \Box (inc_k \div Z_k \ ; \ (C_k \div \mathbf{skip} \ ; \ \mathbf{throww}))]$$
$$P = (C_1^{m_1} \ ; \ Z_1) \parallel (C_2^{m_2} \ ; \ Z_2)$$

In the above construction, inc_k, dec_k and $zero_k$ represent the events of increasing, decreasing and zeroing the k-th counter, respectively. C_1 and C_2 are the processes that try to increase or decrease the first counter and the second counter, respectively. After executing each terminated trace of C_k, the corresponding counter is decreased by one. During the execution of C_k, the memory constraint of the corresponding counter is preserved, *i.e.*, the counter is no less than 1 if it is set to 2 at the beginning of executing C_k. Z_k is the process that tries to keep the content of the kth counter to be zero. Thus, the evolving of the processes C_k and Z_k is consistent with the memory constraint of M. The process P models the 2-counter machine that the initial values of first counter and second counter are m_1 and m_2, respectively, where $C_k^{m_k}$ is the sequential composition of m_k copies of C_k.

According to the instructions of M, we can construct the FSM R=$(\Sigma, S, s_0, \delta, F)$, where Σ is the alphabet set and $\Sigma = \{inc_k, dec_k, zero_k \mid k \in \{1, 2\}\}$, S is the state set and $S = \{s_i \mid 1 \leq i \leq n\} \cup \{s_{n+1}\}$, s_0 is the initial state, *i.e.*, s_1 in S, $\delta : S \times \Sigma \to S$ is the transition function, F is the final state set and $F = S \setminus \{s_n\}$. For each construction, we can add the transitions as follows, where $k \in \{1, 2\}$.

$$\langle i : \ c_k := c_k + 1; \textbf{goto } j\rangle \Longrightarrow \delta(s_i, inc_k) = s_j$$
$$\langle i : \textbf{if } c_k = 0 \textbf{ goto } j; \ c_k := c_k - 1; \textbf{goto } l\rangle \Longrightarrow \delta(s_i, zero_k) = s_j, \delta(s_i, dec_k) = s_l$$

Then, we complete R by adding a self-transition to s_{n+1} for each element in Σ, *i.e.*, $\forall a \in \Sigma \bullet \delta(s_{n+1}, a) = s_{n+1}$. Finally, for each state s except s_n, if there does not exist a transition for an element a in Σ ($\nexists s_1 \in S \bullet \delta(s, a) = s_1$), we add a transition from s to s_{n+1} with the label as a to δ, *i.e.*, $\delta(s, a) = s_{n+1}$. Thus, the FSM R is constructed according to the control constraints of M, and R accepts the trace any prefix of which is not a halted trace of M, because there is no transition from s_n to s_{n+1}, and s_n is not a final state.

According to the above constructions, we next prove "$L(T(P)) \subseteq L(R)$ *iff* M does not halt". Instead of proving it directly, we prove "$L(T(P)) \nsubseteq L(R)$ *iff* M halts".

- If M halts, then there exists a trace s produced by a halted execution of M, which is accepted by $T(P)$, because P satisfies the memory constraints. However, according to the construction of R, s is not accepted by R. Thus, $L(T(P)) \nsubseteq L(R)$.
- If $L(T(P)) \nsubseteq L(R)$, then there exists a trace s accepted by $T(P)$, but s is not accepted by R. According to the construction of R, there must exist s_t that is a prefix of s and s_t can reach the state s_n in the state set S of R, *i.e.*, M can reach the last instruction (*halt*) via s_t. Also, because s is accepted by $T(P)$, the execution of s_t satisfies the memory constraint. Thus, M has a halting execution, *i.e.*, M halts.

In total, we can have $L(T(P)) \subseteq L(R)$ *iff* M does not halt. Therefore, according to the undecidable result of 2-counter machine [17], the problem $L(T(P)) \subseteq L(R)$ is undecidable in general. □