LTS Semantics for Compensation-based Processes*

Roberto Bruni[1] and Anne Kersten Kauer[2]

[1] Department of Computer Science, University of Pisa, Italy
[2] IMT Institute for Advanced Studies, Lucca, Italy

Abstract. Business processes design is an error-prone task often relying on long-running transactions with compensations. Unambiguous formal semantics and flexible verification tools should be used for early validation of processes. To this aim, we define a small-step semantics for the Sagas calculus according to the so-called "coordinated interruption" policy. We show that it can be tuned via small changes to deal with other compensation policies and discuss possible enhancements.

1 Introduction

Long-running transactions (LRTs) in business processes are composed by services taken off-the-shelf. One important problem is failure recovery, i.e., the ability to bring a faulty process back to a consistent state. Processes may grow large and complex and when a fault occurs the designer has to take several constraints into account: all sibling activities that run unaware of the fault should be stopped and all the activities that were executed before the fault need to be undone in a suitable order. Moreover, in many cases, an action, like a service invocation, cannot simply be undone: it can be an ACID transaction on its own, or it may involve asynchronous messaging (e.g., via SMTP).

A *compensation* is the means of reversing the effects of an activity in case a later fault occurs in the business process. Compensations were introduced in [17] to implement (non-ACID) database LRTs as a sequence of short, ACID sub-transactions $t_1 ... t_n$. Each t_i had an associated activity c_i, its compensation, to be installed when t_i committed, and to be executed if a fault occurred before the whole LRT was committed. Compensations are executed in the reverse order of installation. For example, if t_3 fails, then the observed activities are $t_1 t_2 c_2 c_1$. Service-oriented computing is a particularly favourable setting for the concept of compensation, because services are designed without knowing in advance the context where they will be used. For example, take a process that receives an order form with multiple items and delegates each request to a different supplier. If some request fails while others already succeeded, the process may cancel the successful requests and inform the customer about the failure. Still, certain cancellations may involve fees and others may not be possible at all. Moreover,

* Research supported by the EU Integrated Project 257414 ASCENS and by the Italian MIUR Project IPODS (PRIN 2008).

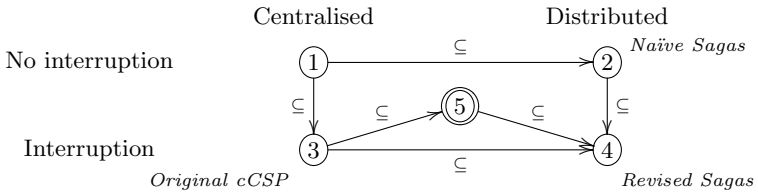


Fig. 1: Compensation policies (arrows stand for trace inclusion)

when the fault occurs one would like to interrupt non-issued requests. Thus, it is natural to demand that a service comes with one (or more) companion service(s) for compensation. An action and its compensation form a *compensation pair*.

Concurrency makes process design an error-prone activity: processes must be assigned with unambiguous semantics and early validated to detect unwanted behaviour and to suppress as many inconsistencies as possible. We focus on the semantics of the Sagas calculus [7]. The core fragment of *parallel* Sagas has been sufficient to characterise different compensation policies for parallel processes. A thorough analysis is presented in [3] by comparing the Sagas calculus with *compensating* CSP [10] (cCSP) along two axes of classification: i) interruption of siblings in case of an abort (*interruption* vs *no interruption*); ii) whether compensations are started at the same time or siblings can start their compensation on their own (*centralised* vs *distributed*). The relation between the four different policies is displayed in Fig. 1. The fifth policy (double lined in Fig. 1) has been formalised in [6] and proved more satisfactory than #1–4, and all semantics #1–5 coincide on the sequential fragment of Sagas. A key contribution in [6] is the definition of a concurrent semantics for policy #5, obtained by encoding Sagas processes in (safe) Petri nets. The Petri net model is more informative than trace semantics, because it accounts for the branching of processes arising from the propagation of interrupts, but the sophisticated mechanism needed for handling interrupts introduces many auxiliary places and transitions that make the Petri net model quite intricate to parse (§ 5.1) and difficult to extend (§ 7).

Our aim is to provide an operational semantics for Sagas whose main requirements are: i) it must follow the small-step style of operational semantics, so to account for the branching caused by the propagation of interrupts; ii) other policies can be implemented without radical redesign; iii) it must be easy to introduce other features, like choice, iteration, and faulty compensations (crashes).

In this paper we propose an LTS semantics that meets all the above requirements. The main result consists of the correspondence theorems with the existing semantics. We started by considering the "optimal" policy #5 and were guided by the correspondence with the Petri net semantics to correct many wrong design choices in our first attempts. The main result is the proof that our LTS semantics matches the Petri nets semantics in [6] up to weak bisimilarity. This gives a way to read markings as (weak bisimilar) terms of a process algebra that describes the run-time status of the process.

$$\begin{array}{llll} (\textsc{act}) & A, B ::= a \mid \mathit{skip} \mid \mathit{throw} & (\textsc{process}) & P, Q ::= X \mid P; Q \mid P|Q \\ (\textsc{step}) & X \quad ::= A \div B & (\textsc{saga}) & S, T ::= A \mid S; T \mid S|T \mid \{\![P]\!\} \end{array}$$

Fig. 2: Core fragment of Sagas

Synopsis. In § 2 we recall the denotational semantics of Sagas. In § 3 we define the LTS semantics for the sequential fragment only, and extend it to the parallel case in § 4. In § 5 we sketch the Petri net semantics from [6] and outline the technique used for proving the main result. In § 6 and § 7 we show the flexibility of our LTS semantics in accommodating other policies and advanced features. Related work, concluding remarks and future work are collected in § 8.

2 Background

The syntax of the parallel Sagas calculus is in Fig. 2. Atomic actions A include generic activities $a \in \mathcal{A}$, the vacuous activity *skip* and the faulty activity *throw*. In a compensation pair $A \div B$, the activity B compensates A. We write *throww* for *throw* $\div$ *skip*. Beside the ordinary sequential and parallel composition, we use $\{\![P]\!\}$ to enclose a compensable process within a saga. Below, we outline the denotational semantics of policies #1–5, while the Petri net semantics for policy #5 is recalled in § 5.1. The Petri net semantics and our novel LTS semantics are parametric to the context of execution that fixes the success or failure of activities. Let $\Omega = \{\boxdot, \boxtimes\}$. A *context* Γ is a function $\Gamma : \mathcal{A} \to \Omega$ that maps a basic activity to $\boxdot$ or $\boxtimes$ depending on whether it commits or aborts, with $\Gamma(\mathit{skip}) = \boxdot$ and $\Gamma(\mathit{throw}) = \boxtimes$. We assume a compensation activity cannot fail. Dealing with faulty compensations is discussed in § 7. The denotational semantics does not use Γ: only *throww* is used for failures.

Notation. A trace for a saga is a string $s\langle\omega\rangle$, where $s \in \mathcal{A}^*$ is said the *observable flow* and $\omega \in \mathcal{R}$ is the *final event*, with $\mathcal{R} = \{\checkmark, !, ?\}$ and $\mathcal{A} \cap \mathcal{R} = \emptyset$ ($\checkmark$ stands for success, ! for fail, and ? for yield to an interrupt). Note that ? appears only in traces of compensable processes. We let ϵ denote the empty observable flow. Slightly abusing the notation, we let $p, q, ...$ range over traces and also observable flows. We denote by $p|||q$ the set of all possible interleavings of the observable flows p and q, with final event $\omega\&\omega'$, where $\&$ is associative and commutative. A trace of a compensable process P is a pair (p, q), where p is the *forward trace* and q is a *compensation trace* for p. We find it convenient to define policy #3 first (see Fig. 3) and then explain the other ones by difference. We use policy numbers as subscripts of the symbol $\triangleq$ when the defining equation may not be valid for other policies (e.g., $\triangleq_{3,4}$ means the definition is valid for policies #3 and #4). Later, we write $[\![\cdot]\!]_i$ to denote the trace semantics w.r.t. policy #i.

Sagas (policies #1–5). For sagas, the most interesting case is the one of $\{\![P]\!\}$: it selects all successful forward traces $s\langle\checkmark\rangle$ of P, and the traces sq, corresponding to failed forward flows $s\langle!\rangle$ followed by their compensations q.

Interruption and centralized compensation (policy #3). When composing compensable traces in series, the forward trace corresponds to the sequential

Traces of sagas

$$S;T \triangleq \{p;q \mid p \in S \wedge q \in T\} \qquad a \triangleq \{a\langle\checkmark\rangle\}$$
$$S|T \triangleq \{r \mid r \in (p||q) \wedge p \in S \wedge q \in T\} \qquad \mathit{skip} \triangleq \{\langle\checkmark\rangle\}$$
$$\{\!|P|\!\} \triangleq \{s\langle\checkmark\rangle \mid (s\langle\checkmark\rangle, q) \in P\} \cup \{sq \mid (s\langle!\rangle, q) \in P\} \qquad \mathit{throw} \triangleq \{\langle!\rangle\}$$

Composition of standard traces

$$\textbf{Sequential} \begin{cases} p\langle\checkmark\rangle; q \triangleq pq \\ p\langle\omega\rangle; q \triangleq p\langle\omega\rangle \text{ when } \omega \neq \checkmark \end{cases}$$

$$\textbf{Parallel} \quad p\langle\omega\rangle||q\langle\omega'\rangle \triangleq \{r\langle\omega\&\omega'\rangle \mid r \in (p|||q)\}, \text{ where } \begin{array}{c|cccccc} \omega & ! & ! & ! & ? & ? & \checkmark \\ \omega' & ! & ? & \checkmark & ? & \checkmark & \checkmark \\ \hline \omega\&\omega' & ! & ! & ! & ? & ? & \checkmark \end{array}$$

$$\text{and} \begin{cases} p|||\epsilon \triangleq \epsilon|||p \triangleq \{p\} \\ a\ p|||b\ q \triangleq \{a\ r \mid r \in (p|||b\ q)\} \cup \{b\ r \mid r \in (a\ p|||q)\} \end{cases}$$

Traces of compensable processes

$$A \div B \triangleq_{3,4} \{(p,q) \mid p \in A \wedge q \in B\} \cup \{(\langle?\rangle, \langle\checkmark\rangle)\}$$
$$P;Q \triangleq \{pp; qq \mid pp \in P \wedge qq \in Q\}$$
$$P|Q \triangleq \{rr \mid rr \in (pp||qq) \wedge pp \in P \wedge qq \in Q\}$$

Composition of compensable traces

$$\textbf{Sequential} \begin{cases} (p\langle\checkmark\rangle, p'); (q, q') \triangleq (pq, q'; p') \\ (p\langle\omega\rangle, p'); (q, q') \triangleq (p\langle\omega\rangle, p') \text{ when } \omega \neq \checkmark \end{cases}$$
$$\textbf{Parallel} \quad (p,p')||(q,q') \triangleq_{1,3} \{(r,r') \mid r \in (p||q) \wedge r' \in (p'||q')\}$$

Fig. 3: Denotational semantics (policy #3)

composition of the original forward traces, while the compensation trace starts by the second compensation followed by the first one. The parallel composition is defined (pairwise) interleaving the forward flows and the backward flows.

No interruption and centralized compensation (policy #1). Policy #1 differs from policy #3 just by ruling out interruption.

$$A \div B \triangleq_{1,2} \{(p,q) \mid p \in A \wedge q \in B\}$$

Interruption and distributed compensation (policy #4). Policy #4 differs from policy #3 only by the following definition of parallel composition of compensable traces. Note that compensations can be triggered by "guessing" that a fault will be issued.

$$(p\langle\checkmark\rangle, p')||(q\langle\checkmark\rangle, q') \triangleq_{2,4} \{(r\langle\checkmark\rangle, r') \mid r \in (p|||q) \wedge r' \in (p'||q')\} \cup \{(r\langle?\rangle, \langle\omega\rangle) \mid r\langle\omega\rangle \in (pp'||qq')\}$$

$$(p\langle\omega\rangle, p')||(q\langle\omega'\rangle, q') \triangleq_{2,4} \{(r\langle\omega\&\omega'\rangle, \langle\omega''\rangle) \mid r\langle\omega''\rangle \in (pp'||qq')\} \quad \text{if } \omega\&\omega' \neq \checkmark$$

No interruption and distributed compensation (policy #2). Policy #2 differs from policy #3 by combining together the two changes above.

Coordinated compensation (policy #5). Policy #5 differs from policy #3 by slightly changing the semantics of compensation pairs, to allow a successfully completed activity to yield, and the semantics of parallel composition, to account for distributed compensation without the guessing mechanism.

$$
\begin{array}{rcl}
A \div B & \triangleq_5 & \{(p,q) \mid p \in A \wedge q \in B\} \cup \{(\langle ? \rangle, \langle \checkmark \rangle)\} \cup \\
& & \{(p\langle ? \rangle, q) \mid p\langle \checkmark \rangle \in A \wedge q \in B\} \\
(p\langle \checkmark \rangle, p')||(q\langle \checkmark \rangle, q') & \triangleq_5 & \{(r\langle \checkmark \rangle, r') \mid r \in (p|||q) \wedge r' \in (p'||q')\} \\
(p\langle \omega \rangle, p')||(q\langle \omega' \rangle, q') & \triangleq_5 & \begin{cases} itp((p\langle \omega \rangle, p'), (q\langle \omega' \rangle, q')) \cup \\ itp((q\langle \omega' \rangle, q'), (p\langle \omega \rangle, p')) & \text{when } \omega, \omega' \neq \checkmark \\ \emptyset & \text{otherwise} \end{cases} \\
itp((p\langle \omega \rangle, p'), (q\langle \omega' \rangle, q')) & \triangleq & \{((p|||q_1)\langle \omega \rangle, (p'||q_2 q')) \mid q = q_1 q_2\}
\end{array}
$$

In summary: in #1 and #2 all sibling processes will finish their execution before compensating; in #3, aborted and interrupted processes cannot start the compensation before all their siblings are ready to compensate; #2 and #4 rely on a "guessing" mechanism for which a process may start its compensation when a sibling will fail in the future; #5 is "optimal" in the sense that distributed compensations can start as soon as needed, but only after an actual fault occurred.

Example 1. Consider the processing of an order in an eStore. First the order is accepted, then, in parallel, the customer's credit card is processed and the order is packed and the courier is booked. If something goes wrong each activity can be compensated, the courier will be cancelled, the order unpacked, for the credit card an error message will be sent and the order can be deleted. Assume that the booking of the courier will always fail and is thus replaced with *throww*.

$$eStore \triangleq \mathsf{aO} \div \overline{\mathsf{aO}}; (\mathsf{pC} \div \overline{\mathsf{pC}} | \mathsf{pO} \div \overline{\mathsf{pO}}; throww)$$

In policies #1 and #2, pC and its compensation will always be executed, while policies #3 and #4 admit e.g. the trace $\mathsf{aO}\ \mathsf{pO}\overline{\mathsf{pO}}\ \overline{\mathsf{aO}}$. Policies #1 and #3 are centralized and no compensation activity can precede a forward activity. Policies #2 and #4 admit the trace $\mathsf{aO}\ \mathsf{pO}\overline{\mathsf{pO}}\ \mathsf{pC}\overline{\mathsf{pC}}\ \overline{\mathsf{aO}}$ (distributed case). They also admit the less realistic trace $\mathsf{aO}\ \mathsf{pC}\overline{\mathsf{pC}}\ \mathsf{pO}\overline{\mathsf{pO}}\ \overline{\mathsf{aO}}$ where the compensation $\overline{\mathsf{pC}}$ is executed before the actual *throww* could have issued a fault. This trace is forbidden in policy #5 (where $\mathsf{aO}\ \mathsf{pO}\overline{\mathsf{pO}}\ \mathsf{pC}\overline{\mathsf{pC}}\ \overline{\mathsf{aO}}$ is still allowed).

3 A Small-Step Semantics for Sequential Sagas

In this section we define a small-step LTS semantics for the sequential fragment of the Sagas calculus. (w.r.t. the syntax in Fig. 2, we ignore parallel composition). To be able to reason on intermediate states in the execution of a process we introduce a runtime syntax.

$$
\begin{array}{lll}
(\text{COMP})\ C ::= A \mid C;C \mid \mathbf{nil} & & (\text{PROCESS})\ P ::= \ldots \mid P\$C \mid [C] \\
& & (\text{SAGA}) \quad S ::= \ldots \mid \mathbf{nil}
\end{array}
$$

First we add a distinct type for compensations. They can either be basic activities A, the sequential composition of compensations $C;C$ or **nil**. With **nil** we denote completion of a compensation, in the sense that, e.g., the compensation $\mathbf{nil};C$ can never execute activities in C. For compensable processes, $P\$C$ denotes a process P running with the already installed compensation C. Compensations

$$\text{(C-ACT)}\ \frac{}{\Gamma \vdash A \xrightarrow{A} \mathbf{nil}} \qquad \text{(C-SEQ1)}\ \frac{\Gamma \vdash C \xrightarrow{\lambda} C' \wedge \neg dn(C')}{\Gamma \vdash C; D \xrightarrow{\lambda} C'; D} \qquad \text{(C-SEQ2)}\ \frac{\Gamma \vdash C \xrightarrow{\lambda} C' \wedge dn(C')}{\Gamma \vdash C; D \xrightarrow{\lambda} D}$$

Fig. 4: LTS for sequential compensations

$$\text{(S-ACT)}\ \frac{A \mapsto_\Gamma \boxdot}{\Gamma \vdash \boxdot, A \div B \xrightarrow{A} \boxdot, [B]} \qquad \text{(SEQ)}\ \frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxdot, P' \wedge \neg dn_{\boxdot}(P')}{\Gamma \vdash \boxdot, P; Q \xrightarrow{\lambda} \boxdot, P'; Q}$$

$$\text{(F-ACT)}\ \frac{A \mapsto_\Gamma \boxtimes}{\Gamma \vdash \boxdot, A \div B \xrightarrow{\tau} \boxtimes, [\mathbf{nil}]} \qquad \text{(S-SEQ)}\ \frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxdot, P' \wedge dn_{\boxdot}(P')}{\Gamma \vdash \boxdot, P; Q \xrightarrow{\lambda} \boxdot, Q\$cmp(P')} \qquad \text{(A-SEQ)}\ \frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxtimes, P'}{\Gamma \vdash \boxdot, P; Q \xrightarrow{\lambda} \boxtimes, P'}$$

$$\text{(STEP)}\ \frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \sigma', P' \wedge \neg dn_{\sigma'}(P')}{\Gamma \vdash \sigma, P\$C \xrightarrow{\lambda} \sigma', P'\$C} \qquad \text{(AS-STEP1)}\ \frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \sigma', P' \wedge dn_{\sigma'}(P') \wedge tocmp(P')}{\Gamma \vdash \sigma, P\$C \xrightarrow{\lambda} \sigma', [cmp(P'); C]}$$

$$\text{(AS-STEP2)}\ \frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \sigma', P' \wedge dn_{\sigma'}(P') \wedge \neg tocmp(P')}{\Gamma \vdash \sigma, P\$C \xrightarrow{\lambda} \sigma', [C]} \qquad \text{(COMP)}\ \frac{\Gamma \vdash C \xrightarrow{\lambda} C'}{\Gamma \vdash \boxtimes, [C] \xrightarrow{\lambda} \boxtimes, [C']}$$

Fig. 5: LTS for sequential compensable processes

of P will be installed on top of C once P is finished. The completion of forward activities is denoted by $[C]$ instead of **nil**, because we need to consider the installed compensation C (informally, $[C]$ can be read as **nil**$\$C$). We also add **nil** for marking the completion of a saga.

The small-step semantics is defined by three LTSs, one for each syntax category. Given the set of compensations $\mathcal{C}$, the set of compensable processes $\mathcal{P}$ and the set of sagas $\mathcal{S}$, we let $\mathcal{S}_C = \mathcal{C}$, $\mathcal{S}_P = \Omega \times \mathcal{P}$, $\mathcal{S}_S = \Omega \times \mathcal{S}$.

Definition 1. *The LTS semantics of (sequential) sagas is the least LTS $(\mathcal{S}, L, \mathcal{T})$ generated by the rules in Fig. 4–6, whose set of states is $\mathcal{S} = \mathcal{S}_C \cup \mathcal{S}_P \cup \mathcal{S}_S$, whose set of labels is $L = \mathcal{A} \cup \{\tau\}$.*

We will write transitions $t \in \mathcal{T}$ as $t : \Gamma \vdash s \xrightarrow{\lambda} s'$ for states s, s', a label $\lambda \in L$ and a context Γ. The component Ω in a state describes whether the process can still commit (it can still move forward) or must abort (a fault was issued that needs to be compensated). Note that states of the LTS for compensations have clearly no Ω component. Sagas initially start executing in a commit state.

The semantics exploits some auxiliary notation. The predicate dn_σ checks the completion of (the forward execution of) a compensable process. The subscript σ stands for $\boxdot$ or $\boxtimes$ and means that the process is either evaluated in a commit or an abort context. The predicate dn_σ is inductively defined as:

$$dn_\sigma([C]) \triangleq \mathtt{tt} \quad dn_\sigma(A \div B) \triangleq \mathtt{ff} \quad dn_\sigma(P\$C) \triangleq dn_\sigma(P; Q) \triangleq dn_\sigma(P)$$

$$\frac{A \mapsto_\Gamma \boxdot}{\Gamma \vdash \boxdot, A \xrightarrow{A} \boxdot, \mathbf{nil}}\ (\textsc{s-act}) \qquad \frac{\Gamma \vdash \sigma, S \xrightarrow{\lambda} \sigma', S' \wedge \neg dn(S')}{\Gamma \vdash \sigma, S; T \xrightarrow{\lambda} \sigma', S'; T}\ (\textsc{sseq})$$

$$\frac{A \mapsto_\Gamma \boxtimes}{\Gamma \vdash \boxdot, A \xrightarrow{\tau} \boxtimes, \mathbf{nil}}\ (\textsc{f-act}) \qquad \frac{\Gamma \vdash \sigma, S \xrightarrow{\lambda} \boxdot, S' \wedge dn(S')}{\Gamma \vdash \sigma, S; T \xrightarrow{\lambda} \boxdot, T}\ (\textsc{s-sseq}) \qquad \frac{\Gamma \vdash \sigma, S \xrightarrow{\lambda} \boxtimes, S' \wedge dn(S')}{\Gamma \vdash \sigma, S; T \xrightarrow{\lambda} \boxtimes, S'}\ (\textsc{a-sseq})$$

$$\frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \sigma', P' \wedge \neg dn_{\sigma'}(P')}{\Gamma \vdash \sigma, \{\!|P|\!\} \xrightarrow{\lambda} \sigma', \{\!|P'|\!\}}\ (\textsc{saga}) \qquad \frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \boxtimes, P' \wedge dn_{\boxtimes}(P') \wedge tocmp(P')}{\Gamma \vdash \sigma, \{\!|P|\!\} \xrightarrow{\lambda} \boxtimes, \{\!|P'|\!\}}\ (\textsc{a-saga1})$$

$$\frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxdot, P' \wedge dn_{\boxdot}(P')}{\Gamma \vdash \boxdot, \{\!|P|\!\} \xrightarrow{\lambda} \boxdot, \mathbf{nil}}\ (\textsc{s-saga}) \qquad \frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \boxtimes, P' \wedge dn_{\boxtimes}(P') \wedge \neg tocmp(P')}{\Gamma \vdash \sigma, \{\!|P|\!\} \xrightarrow{\lambda} \boxdot, \mathbf{nil}}\ (\textsc{a-saga2})$$

Fig. 6: LTS for sequential sagas

Note that for sequential processes dn is independent of the subscript; this will change when introducing parallel composition. Analogously, we define a predicate dn on compensations and sagas, together with a function $cmp(P)$ that extracts the installed compensation from a process P that is "done". When P is done, we use the shorthand $tocmp(P) \triangleq \neg dn(cmp(P))$ (i.e., $tocmp(P)$ holds when there is some compensation to run).

$$dn(\mathbf{nil}) \triangleq \mathtt{tt} \qquad dn(A) \triangleq \mathtt{ff} \qquad dn(C; C') \triangleq dn(C)$$

$$dn(\mathbf{nil}) \triangleq \mathtt{tt} \qquad dn(A) \triangleq dn(\{\!|P|\!\}) \triangleq \mathtt{ff} \qquad dn(S; T) \triangleq dn(S)$$

$$cmp([C]) \triangleq C \qquad cmp(P; Q) \triangleq cmp(P) \qquad cmp(P\$C) \triangleq \begin{cases} C & \text{if } dn(cmp(P)) \\ cmp(P); C & \text{if } \neg dn(cmp(P)) \end{cases}$$

The rules in Fig. 4 handle compensations. As we assume a compensation is always successful, only rule C-ACT is needed for basic activities. Rules C-SEQ1 and C-SEQ2 exploit the "done" predicate to avoid reaching states such as $\mathbf{nil}; C$.

For processes (Fig. 5), a basic activity A of $A \div B$ can either commit or abort, depending on the context: if A commits then B is installed (S-ACT); if A fails, then there is nothing to be compensated (F-ACT). A sequential composition $P; Q$ acts according to how P acts (SEQ and A-SEQ). If P finishes successfully (S-SEQ), then Q will run under the installed compensation $cmp(P')$. The process $P\$C$ acts according to P. When P finishes its compensation is installed on top of C (AS-STEP1). The rule AS-STEP2 ensures that a **nil** is not installed on top of a compensation. Compensations are executed via COMP.

The rules for sagas A and $S; T$ are as expected (Fig. 6). A saga $\{\!|P|\!\}$ can be executed as long as either it is still running forward (SAGA and S-SAGA) or it has already aborted and compensates (SAGA and A-SAGA1). If the saga aborts but is able to compensate, then it reaches a good state (A-SAGA2).

The formal correspondence between the LTS semantics and the denotational semantics of policies #1–5 is an immediate consequence of our main result and will be deferred to § 4 (see Corollary 1).

Example 2. Let $eS \triangleq \mathsf{aO} \div \overline{\mathsf{aO}}; \mathsf{pC} \div \overline{\mathsf{pC}}; \mathsf{pO} \div \overline{\mathsf{pO}}; \mathsf{bC} \div \overline{\mathsf{bC}}$. Assume that the packing of the order fails, and let Γ map pO to $\boxtimes$ and the other actions to $\boxdot$. We have e.g. $\boxdot, \{\!|eS|\!\} \xrightarrow{\mathsf{aO}} \xrightarrow{\mathsf{pC}} \xrightarrow{\tau} \xrightarrow{\overline{\mathsf{pC}}} \xrightarrow{\overline{\mathsf{aO}}} \boxdot, \mathbf{nil}$, because

$$\boxdot, eS \xrightarrow{\mathsf{aO}} \boxdot, (\mathsf{pC} \div \overline{\mathsf{pC}}; \mathsf{pO} \div \overline{\mathsf{pO}}; \mathsf{bC} \div \overline{\mathsf{bC}})\$\overline{\mathsf{aO}} \xrightarrow{\mathsf{pC}}$$
$$\boxdot, ((\mathsf{pO} \div \overline{\mathsf{pO}}; \mathsf{bC} \div \overline{\mathsf{bC}})\$\overline{\mathsf{pC}})\$\overline{\mathsf{aO}} \xrightarrow{\tau} \boxtimes, [\overline{\mathsf{pC}}; \overline{\mathsf{aO}}] \xrightarrow{\overline{\mathsf{pC}}} \boxtimes, [\overline{\mathsf{aO}}] \xrightarrow{\overline{\mathsf{aO}}} \boxtimes, [\mathbf{nil}]$$

4 Extension to Concurrency

In this section we extend the LTS semantics to handle parallel Sagas. First, we extend the runtime syntax as follows:

$$\begin{array}{lll} (\textsc{comp}) & C ::= A \mid C;C \mid \mathbf{nil} \mid C|C \\ (\textsc{process}) & P ::= X \mid P;P \mid P\$C \mid [C] \mid P_\sigma|_{\sigma'}P \\ (\textsc{saga}) & S ::= A \mid S;S \mid \{\!|P|\!\} \mid \mathbf{nil} \mid S_\sigma|_{\sigma'}S \end{array}$$

We add parallel composition to compensations. We use subscripts for the parallel composition of processes or sagas ${}_\sigma|_{\sigma'}$ such that $\sigma, \sigma' \in \Omega$. If a thread is denoted with $\boxdot$, it can still move forward and commit. A thread denoted with $\boxtimes$ either aborted or was interrupted, so it can compensate. If a thread is denoted with a $\boxdot$ then also every parallel composition contained as a subprocess in this thread must have a $\boxdot$. Similarly if the global state is a $\boxdot$, any parallel composition in this state has subscripts $\boxdot$. We consider $P_\boxdot|_\boxdot Q$ part of the normal syntax, not just of the runtime syntax, and usually write just $P|Q$. We sometimes use $\|$ instead of ${}_\sigma|_{\sigma'}$ if the values of σ, σ' are irrelevant.

Definition 2. *The LTS semantics of parallel sagas is the least LTS $(\mathcal{S}, L, \mathcal{T})$ generated by the rules in Fig. 4–6 together with the rules in Fig. 7 (symmetric rules* C-PAR-R*,* PAR-R*,* INT-L *and* SPAR-R *are omitted).*

The semantics exploits some auxiliary notation. First, the binary function $\sqcap : \Omega \times \Omega \to \Omega$ is defined such that $\sigma \sqcap \sigma' = \boxdot$ iff $\sigma = \sigma' = \boxdot$. It is easy to check that $\sqcap$ is associative and commutative. Then, the predicates dn_σ, dn and the function cmp are extended to parallel composition:

$$dn(C|C') \triangleq dn(C) \wedge dn(C') \quad dn_\sigma(P_{\sigma_1}|_{\sigma_2}Q) \triangleq dn_\sigma(P) \wedge dn_\sigma(Q) \wedge (\sigma = \sigma_1 = \sigma_2)$$
$$dn(S_{\sigma_1}|_{\sigma_2}T) \triangleq dn(S) \wedge dn(T) \quad cmp(P \parallel Q) \triangleq cmp(P)|cmp(Q)$$

The process $P_{\sigma_1}|_{\sigma_2}Q$ is done when both P and Q are done and both subscripts are the same and coincide with the global state σ.

The rules C-PAR-L/C-PAR-R define just the ordinary interleaving of compensations. The rules PAR-L/PAR-R are analogous, but the subscript determines the

$$\text{(C-PAR-L)}\ \frac{\Gamma \vdash C \xrightarrow{\lambda} C'}{\Gamma \vdash C|D \xrightarrow{\lambda} C'|D} \qquad \text{(SPAR-L)}\ \frac{\Gamma \vdash \sigma_1, S \xrightarrow{\lambda} \sigma_1', S'}{\Gamma \vdash \sigma, S_{\sigma_1}|_{\sigma_2} T \xrightarrow{\lambda} \sigma_1' \sqcap \sigma_2, S'_{\sigma_1'}|_{\sigma_2} T}$$

$$\text{(PAR-L)}\ \frac{\Gamma \vdash \sigma_1, P \xrightarrow{\lambda} \sigma_1', P'}{\Gamma \vdash \sigma, P_{\sigma_1}|_{\sigma_2} Q \xrightarrow{\lambda} \sigma \sqcap \sigma_1', P'_{\sigma_1'}|_{\sigma_2} Q} \qquad \text{(INT-R)}\ \frac{Q \rightsquigarrow Q'}{\Gamma \vdash \boxtimes, P_\sigma|_\boxdot Q \xrightarrow{\tau} \boxtimes, P_\sigma|_\boxtimes Q'}$$

Fig. 7: LTS rules for parallel Sagas (symmetric rules omitted for brevity)

$$\frac{}{[C] \rightsquigarrow [C]} \quad \frac{}{A \div B \rightsquigarrow [\mathbf{nil}]} \quad \frac{P \rightsquigarrow P' \wedge \neg par(P)}{P; Q \rightsquigarrow P'} \quad \frac{par(P)}{P; Q \rightsquigarrow P} \quad \frac{P \rightsquigarrow P'}{P|Q \rightsquigarrow P'_\boxtimes|_\boxdot Q} \quad \frac{Q \rightsquigarrow Q'}{P|Q \rightsquigarrow P_\boxdot|_\boxtimes Q'}$$

$$\frac{P \rightsquigarrow P' \wedge dn_\boxtimes(P') \wedge tocmp(P')}{P\$C \rightsquigarrow [cmp(P'); C]} \quad \frac{P \rightsquigarrow P' \wedge \neg dn_\boxtimes(P')}{P\$C \rightsquigarrow P'\$C} \quad \frac{P \rightsquigarrow P' \wedge dn_\boxtimes(P') \wedge \neg tocmp(P')}{P\$C \rightsquigarrow [C]}$$

Fig. 8: Predicate $P \rightsquigarrow P'$ for interrupting a process

modality of execution. A thread can move forward when it is in a commit state. If a thread aborts, the failure is annotated also in the global state by taking $\sigma \sqcap \sigma_1'$. A commit thread can still move forward even if the global mode is abort.

The rules INT-L/INT-R use an "extract" predicate $P \rightsquigarrow P'$ to interrupt a commit thread if the global process is in abort mode. In $P \rightsquigarrow P'$, the process P' is a possible result of interrupting P (see Fig. 8). As a special case, note the interrupt of a sequential composition: we distinguish whether P is a parallel composition (predicate $par(P)$ is true) or not. This is motivated by the intention to adhere to the Petri net semantics, where $(P|Q); R$ can be interrupted discarding R but without necessarily interrupting P or Q.

For the parallel composition of sagas (SPAR-L/SPAR-R) we just remark that in the case of fault of one thread we let the other threads execute as much as possible and just record the global effect in the σ component of the state.

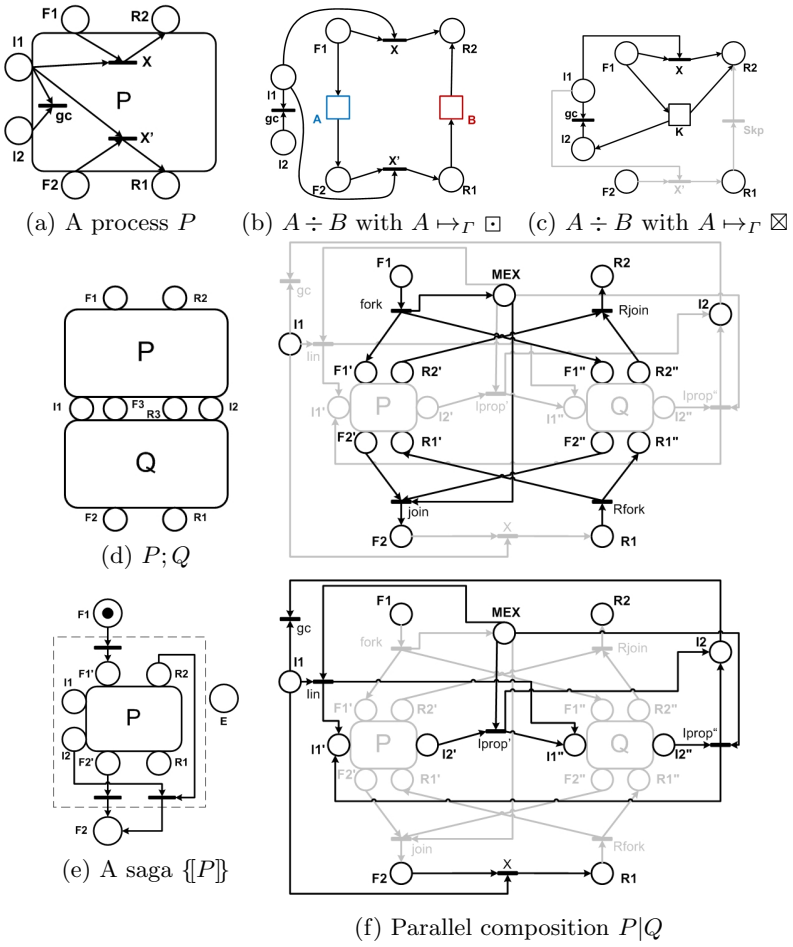
Example 3. Let $eS' \triangleq \mathsf{aO} \div \overline{\mathsf{aO}}; (\mathsf{pC} \div \overline{\mathsf{pC}} | \mathsf{pO} \div \overline{\mathsf{pO}}; \mathsf{bC} \div \overline{\mathsf{bC}})\$\overline{\mathsf{aO}}$, and assume that the processing of the card fails while the other actions are successful.

$$\begin{aligned}
&\boxdot, eS' \xrightarrow{\mathsf{aO}} \boxdot, (\mathsf{pC} \div \overline{\mathsf{pC}} \mid \mathsf{pO} \div \overline{\mathsf{pO}}; \mathsf{bC} \div \overline{\mathsf{bC}})\$\overline{\mathsf{aO}} \xrightarrow{\mathsf{pO}} \\
&\quad \boxdot, (\mathsf{pC} \div \overline{\mathsf{pC}} \mid (\mathsf{bC} \div \overline{\mathsf{bC}})\$\overline{\mathsf{pO}})\$\overline{\mathsf{aO}} \xrightarrow{\tau} \boxtimes, ([\mathbf{nil}]_\boxtimes|_\boxdot (\mathsf{bC} \div \overline{\mathsf{bC}})\$\overline{\mathsf{pO}})\$\overline{\mathsf{aO}} \xrightarrow{\tau} \\
&\quad \boxtimes, [(\mathbf{nil} \mid \overline{\mathsf{pO}}); \overline{\mathsf{aO}}] \xrightarrow{\overline{\mathsf{pO}}} \boxtimes, [\overline{\mathsf{aO}}] \xrightarrow{\overline{\mathsf{aO}}} \boxtimes, [\mathbf{nil}]
\end{aligned}$$

5 Operational Correspondence

In this section we will show a weak bisimilarity between our novel LTS semantics and the Petri net semantics of [6].

5.1 Petri Net Semantics (for Policy #5)



(a) A process P (b) $A \div B$ with $A \mapsto_\Gamma \boxdot$ (c) $A \div B$ with $A \mapsto_\Gamma \boxtimes$

(d) $P;Q$

(e) A saga $\{\![P]\!\}$

(f) Parallel composition $P|Q$

Fig. 9: Encoding of compensable processes as (safe) Petri nets

In [6] Sagas processes are encoded in safe Petri nets by structural induction (see Fig. 9). A saga has just three places to interact with the environment: F_1 starts its flow, F_2 signals successful termination, and E raises a fault. Each compensable process has six places to interact with the environment: a token in F_1 triggers the forward flow, to end in F_2; a token in R_1 starts the compensation, to end in R_2; a token in I_1 indicates the arrival of an interrupt from the outside; a token in I_2 informs the environment that a fault occurred. For sagas, a computation starting in F_1 will lead either to F_2 or to E, while for compensable processes we expect to have the following kinds of computations:

Successful (forward) computation: from marking F_1 the net reaches F_2
Compensating (backward) computation: from R_1 the net reaches R_2.
Aborted computation: from F_1 the net reaches $R_2 + I_2$.
Interrupted computation: from $F_1 + I_1$ the net reaches R_2.

The nets for compensable processes are depicted in Fig. 9. The encoding introduces several auxiliary transitions (thinner and black filled), e.g. to fork and join the control flow, to catch an interrupt and reverse the flow.

Depending on the context, for a successful compensation pair $A \div B$ (Fig. 9b) we have the obvious transitions modelling activities A and B together with auxiliary transitions for handling interruption. The net for a failing compensation pair (Fig. 9c) has a transition K that models the abort of the transaction. The net for the sequential composition $P; Q$ (Fig. 9d) is obtained by merging the forward output place F_3 of P with the forward input place of Q and the backward output place R_3 of Q with the backward input place of P. Moreover, P and Q share also the places for I_1 and I_2.

The encoding of parallel composition $P|Q$ (Fig. 9f) is more complex. We use two subnets for the two processes, with places $F_1', F_2', \ldots$ and $F_1'', F_2'', \ldots$ resp. The upper part of the figure highlights the transitions used in absence of interruptions and the lower part focuses on transitions exploited by interruption.

5.2 Weak Bisimilarity Result

In the following, we write $p \stackrel{\hat{\tau}}{\Rightarrow} q$ if $(p, q) \in (\stackrel{\tau}{\rightarrow})^*$. Moreover, for $\mu \neq \tau$ we write $p \stackrel{\hat{\mu}}{\Rightarrow} q$ if there exists p', q' such that $p' \stackrel{\mu}{\rightarrow} q'$ and $(p, p'), (q, q') \in (\stackrel{\tau}{\rightarrow})^*$.

Definition 3. *Let (S_1, L, T_1) and (S_2, L, T_2) be two LTSs. A relation $\mathbf{R} \subseteq S_1 \times S_2$ is a* weak bisimulation *if whenever $(s_1, s_2) \in \mathbf{R}$, then:*

1. *if $s_1 \stackrel{\mu}{\rightarrow}_1 s_1'$ then there exists s_2' such that $s_2 \stackrel{\hat{\mu}}{\Rightarrow}_2 s_2'$ and $(s_1', s_2') \in \mathbf{R}$; and*
2. *if $s_2 \stackrel{\mu}{\rightarrow}_2 s_2'$ then there exists s_1' such that $s_1 \stackrel{\hat{\mu}}{\Rightarrow}_1 s_1'$ and $(s_1', s_2') \in \mathbf{R}$.*

The largest weak bisimulation is called weak bisimilarity *and denoted by $\approx$.*

We shall let the marking graph of the net N_P play the role of (S_1, L, T_1) and (the fragment of) the LTS reachable from process P play the role of (S_2, L, T_2), so that $\approx$ relates markings of N_P with processes P' reachable from P. More precisely, we assume the only observable actions in the marking graph are those corresponding to activities $a \in \mathcal{A}$; all the other transitions are labelled with τ.

We have seen that the Petri net semantics associates to a compensable process P a corresponding net N_P that exchanges tokens with the context via six places. The places F_1, R_1, I_1 are used to receive tokens in input from the environment, while the places F_2, R_2, I_2 are used to output tokens to the environment. Nets are usually considered up-to isomorphism, therefore the names of their places and transitions are not important, as long as the same structure is maintained. However, to establish the behavioural correspondence between our LTS for P

and the marking graph of the net N_P we need to fix a particular naming of the elements in N_P. Moreover, the same activity can occur many times in a process and every instance corresponds to a different element of the net. One way to eliminate any ambiguity is to annotate processes with the names of the places to be used for building the interface of the corresponding net (before the translation takes place). The proof of the main theorem requires some ingenuity to fix the correspondence between net markings and process terms. Here, we just mention that we write $P@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle$ meaning that process P (and all its sub-processes) has been annotated in such a way that the names of the places in the "public interface" of the net N_P are $F_1, F_2, R_1, R_2, I_1, I_2$.

Theorem 1. *Let N_P be the Petri net associated with the tagged compensable process $P@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle$. Then, $F_1 \approx (\boxdot, P)$.*

As an immediate consequence of the theorem and the main result of [6] the correspondence to the denotational semantics given in § 2 follows.

For any sagas S we let $(\!|S|\!)$ denote the set of *weak traces* in our LTS semantics:

$$(\!|S|\!) \triangleq \{\, a_1...a_n\langle\checkmark\rangle \mid \exists S_1,...,S_n,\sigma_1,...,\sigma_{n-1}.\ \boxdot, S \stackrel{\hat{a_1}}{\Rightarrow} \sigma_1, S_1 \stackrel{\hat{a_2}}{\Rightarrow} \cdots \stackrel{\hat{a_n}}{\Rightarrow} \boxdot, S_n \not\rightarrow \}\cup$$
$$\{\, a_1...a_n\langle!\rangle \mid \exists S_1,...,S_n,\sigma_1,...,\sigma_{n-1}.\ \boxdot, S \stackrel{\hat{a_1}}{\Rightarrow} \sigma_1, S_1 \stackrel{\hat{a_2}}{\Rightarrow} \cdots \stackrel{\hat{a_n}}{\Rightarrow} \boxtimes, S_n \not\rightarrow \}$$

Actually, under the assumption that compensation cannot fail, only successful traces are present in $(\!|S|\!)$ (as well as in $[\![S]\!]_i$ for any $i \in [1,5]$). This is not necessarily the case for the last extension in § 7.

Corollary 1. *For any sagas $S = \{\!|P|\!\}$ we have $[\![S]\!]_5 = (\!|S|\!)$. Moreover, if P is sequential then $[\![S]\!]_i = (\!|S|\!)$ for $i \in [1,5]$.*

6 Dealing with Other Compensation Policies

In this section we show that we can tune the LTS semantics to match and improve other compensation policies discussed in the literature.

Notification and distributed compensation. To remove the possibility to interrupt a sibling process before it ends its execution we just redefine the "extract" predicate by removing most cases, so that the interrupt is possible only when the process is "done".

$$\frac{}{[C] \rightsquigarrow [C]} \qquad \frac{P \rightsquigarrow P'}{P\$C \rightsquigarrow P'\$C} \qquad \frac{P \rightsquigarrow P'}{P|Q \rightsquigarrow P'_{\boxtimes}|_{\boxdot}Q} \qquad \frac{Q \rightsquigarrow Q'}{P|Q \rightsquigarrow P_{\boxdot}|_{\boxtimes}Q'}$$

Now, the rule INT is only applicable if the interrupted process consists of an installed compensation $[C]$. The new extract predicate only changes the subscripts, not the process: since any interrupted thread is "done" we never inhibit sibling forward activities upon a fault.

We call this strategy *Notification and distributed compensation* (policy #6) to emphasize the fact that siblings are notified about the fault, not really interrupted. Since compensations are distributed and the fault is not observable, it

can happen that a notified thread starts compensating even before the sibling that actually aborted. However, contrary to policy #2, a thread cannot guess the presence of faulty siblings, so it is not possible to observe a forward activity of the only faulty thread after a compensation activity of a notified thread. Thus policy #6 defines a variant of policy #2 where unrealistic traces are discarded.

Proposition 1. *Let $(\!|\cdot|\!)_6$ denote the set of weak traces generated by policy #6 above. Then, for any sagas $S = \{\!|P|\!\}$ we have $[\![S]\!]_1 \subseteq (\!|S|\!)_6 \subseteq [\![S]\!]_2$. Moreover, for some P the inclusion is strict, while for sequential processes $[\![P]\!]_1 = (\!|P|\!)_6 = [\![P]\!]_2$.*

Interruption and centralized compensation. To move from distributed to centralized execution we simply strengthen the premise in PAR-L (and PAR-R):

$$\frac{(\sigma_1 = \boxdot \vee dn_{\boxtimes}(P_{\sigma_1}|_{\sigma_2} Q) \vee \neg dn_{\boxtimes}(P)) \qquad \Gamma \vdash \sigma_1, P \xrightarrow{\lambda} \sigma_1', P'}{\Gamma \vdash \sigma, P_{\sigma_1}|_{\sigma_2} Q \xrightarrow{\lambda} \sigma \sqcap \sigma_1', P'_{\sigma_1'}|_{\sigma_2} Q} \text{(PAR-L)}$$

Thus a process can only be executed if it is either moving forward ($\sigma_1 = \boxdot$) or the complete parallel composition finished in a failing case ($dn_{\boxtimes}(P_{\sigma_1}|_{\sigma_2} Q)$) or the thread has not yet finished its execution in a failing case ($\neg dn_{\boxtimes}(P)$).

Proposition 2. *Let $(\!|\cdot|\!)_3$ denote the set of weak traces generated by policy #3 above. Then, for any sagas $S = \{\!|P|\!\}$ we have $[\![S]\!]_3 = (\!|S|\!)_3$.*

No interruption and centralized compensation. By combining the above changes we recover policy #1.

7 Possible Extensions

Choice and iteration. Our first extension adds choice $P + P$ and iteration P^* operators to the syntax for processes The corresponding rules are in Fig. 10. In a process $P + Q$ one option is nondeterministically executed while the alternative is dropped. For iteration, a process P^* either executes a τ and finishes or acts as the sequential composition $P; P^*$. Note that, while it is easy to account for choice and iteration in the denotational semantics, the extension is harder for the Petri net semantics. For example, let us consider the sequential process $(A \div A' + B \div B')^*; \mathit{throww}$. At any iteration, either A or B is executed and thus either A' or B' is installed. When the iteration is closed, the installed compensation may be any arbitrary sequence of A' or B', an information that cannot be recorded in the state of a finite (safe) Petri net.

Failing compensations. One important contribution of [7] was the ability to account for the failure of compensations. Here we discuss how to extend our LTS semantics accordingly.

For compensations, we extend the states with $\Omega = \{\boxdot, \boxtimes\}$, modify the sources / targets from C to $\boxdot, C$ in the rules we have presented, change the rule C-ACT as below and add the rules F-C-SEQ, C-PAR-L and C-PAR-R that record the execution of a faulty compensation in the target of the transition:

$$dn_\sigma(P+Q) \triangleq dn_\sigma(P) \wedge dn_\sigma(Q) \quad dn_\sigma(P^*) \triangleq dn_\sigma(P) \quad \frac{}{P+Q \rightsquigarrow [\mathbf{nil}]} \quad \frac{}{P^* \rightsquigarrow [\mathbf{nil}]}$$

$$\text{(CHOICE-L)}\ \frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \sigma, P'}{\Gamma \vdash \boxdot, P+Q \xrightarrow{\lambda} \sigma, P'} \qquad \text{(E-ITER)}\ \frac{}{\Gamma \vdash \boxdot, P^* \xrightarrow{\tau} \boxdot, [\mathbf{nil}]} \qquad \text{(S-ITER)}\ \frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxdot, P' \wedge \neg dn(P')}{\Gamma \vdash \boxdot, P^* \xrightarrow{\lambda} \boxdot, P'; P^*}$$

$$\text{(CHOICE-R)}\ \frac{\Gamma \vdash \boxdot, Q \xrightarrow{\lambda} \sigma, Q'}{\Gamma \vdash \boxdot, P+Q \xrightarrow{\lambda} \sigma, Q'} \qquad \text{(A-ITER)}\ \frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxtimes, P'}{\Gamma \vdash \boxdot, P^* \xrightarrow{\lambda} \boxtimes, P'} \qquad \text{(S-ITER2)}\ \frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxdot, P' \wedge dn(P')}{\Gamma \vdash \boxdot, P^* \xrightarrow{\lambda} \boxdot, P^*\$cmp(P')}$$

Fig. 10: LTS for choice and iteration

$$\text{(C-ACT)}\ \frac{A \mapsto_\Gamma \sigma}{\Gamma \vdash \boxdot, A \xrightarrow{A} \sigma, \mathbf{nil}} \qquad \text{(F-C-SEQ)}\ \frac{\Gamma \vdash \boxdot, C \xrightarrow{\lambda} \boxtimes, C'}{\Gamma \vdash \boxdot, C; D \xrightarrow{\lambda} \boxtimes, C'} \qquad \text{(C-PAR-L)}\ \frac{\Gamma \vdash \boxdot, C \xrightarrow{\lambda} \sigma, C'}{\Gamma \vdash \boxdot, C|D \xrightarrow{\lambda} \sigma, C'|D}$$

For compensable processes, we extend the state in LTS to $\Omega^{\circledast} = \{\boxdot, \boxtimes, \circledast\}$, where the symbol $\circledast$ denotes the fault of a compensation, i.e., a non recoverable crash. As a matter of notation for meta-variables, we let $\sigma, \ldots \in \Omega$ and $\delta \in \{\boxtimes, \circledast\}$. When executing a compensation $[C]$, we must take into account the possibility of a crash (COMP-1 and COMP-2). Moreover, if we generate a crash, previously installed local compensations will not be executed (C-STEP):

$$\text{(COMP-1)}\ \frac{\Gamma \vdash \boxdot, C \xrightarrow{\lambda} \boxdot, C'}{\Gamma \vdash \delta, [C] \xrightarrow{\lambda} \delta, [C']} \qquad \text{(COMP-2)}\ \frac{\Gamma \vdash \boxdot, C \xrightarrow{\lambda} \boxtimes, C'}{\Gamma \vdash \delta, [C] \xrightarrow{\lambda} \circledast, [C']} \qquad \text{(C-STEP)}\ \frac{\Gamma \vdash \boxtimes, P \xrightarrow{\lambda} \circledast, P'}{\Gamma \vdash \boxtimes, P\$C \xrightarrow{\lambda} \circledast, P'}$$

(Note that in the premises of rules COMP-1 and COMP-2 we intentionally put $\boxdot$ in the source of the transition, because the LTS for compensations has only such states as sources of transitions.) The other rules for sequential Sagas stay as before. For parallel composition we redefine the predicate dn such that

$$dn_\boxdot(P_\boxdot|_\boxdot Q) \triangleq dn_\boxdot(P) \wedge dn_\boxdot(Q) \quad dn_\delta(P_{\delta_1}|_{\delta_2} Q) \triangleq dn_\delta(P) \wedge dn_\delta(Q)$$

where $\delta, \delta_1, \delta_2 \in \{\boxtimes, \circledast\}$. The rules PAR-L/PAR-R are as before however for any meta-variable we allow also $\circledast$ as a possible value, i.e., $\sigma, \sigma_1, \sigma_2, \sigma_1', \sigma_2' \in \Omega^{\circledast}$. Thus we have to extend the operation $\sqcap$ such that $\circledast \sqcap \sigma = \circledast$. The rules INT-L/INT-R are also applicable in a global $\circledast$ state.

The rules guarantee that in case of a crash parallel branches can execute their compensations as far as possible, only previously installed compensation (i.e., before the parallel composition) are not reachable anymore.

8 Concluding Remarks

We presented an LTS semantics for the Sagas calculus. Using a weak bisimulation we investigated the correspondence with previously defined Petri net and denotational semantics. Moreover, with small changes we can deploy a different policy for the execution of concurrent compensable processes. We have shown suitable semantics extensions enriching first the syntax and then the LTS itself.

This work is a first step towards a flexible tool for specifying and verifying LRTs. While previous semantic definitions for Sagas gave a formal model for LRTs, the LTS semantics is more suitable for custom property verification, like model-checking. The LTS has been implemented in Maude, a language based on rewriting logic and including tools like an inductive theorem prover or an LTL model-checker (http://maude.cs.uiuc.edu). In the end we would like to integrate the specified extensions as well as the option to choose which compensation policy should be used together with a high-level dynamic logic for validation and verification. Some promising steps in this direction are described in [5].

Related work. One of the first attempt to a process algebraic formalization of LRTs is StAC [9], from which both Sagas [7] and cCSP [10] later originated. A small-step semantics for cCSP was defined in [11]. It relies on the centralized compensation policy, but is otherwise similar to our approach. Using a synchronizing step at the end of the forward flow the success or failure of the transaction is published, in case of a failure the compensations are executed as normal saga processes (outside the transaction scope). In our approach the information about a failure is kept in the state and compensations are executed inside the saga.

Compensation for a simple class of nets, called workflow nets, has been studied in [1]. It is simpler than the net semantics of [6] as it does not account for interruption after a fault, but it is less elegant because a compensated run may end with some remaining tokens. A *dynamic* policy for compensation is defined in [20], where the compensation of a concurrent process depends on the order of the interleaving of the forward actions, i.e. there is a unique compensation stack that is updated by each action. The above studies have been applied to provide formal support to standard technologies for web services [12, 25, 15, 2] and to develop provably correct engines for transactional workflows [4, 18, 22, 19].

Finally, we mention other approaches that focus on the interaction between processes. Notable examples are: webπ [23] and dcπ [26] that extend the π-calculus, cJoin [8] that extends the Join calculus, CommTrans [14] for CCS and the reversible process calculi in [13, 24, 21], where the special case of perfect roll-back is investigated. We refer to [16] for some conceptual comparisons.

References

1. Acu, B., Reisig, W.: Compensation in workflow nets. In: ICATPN'06. LNCS, vol. 4024, pp. 65–83 (2006)
2. Bocchi, L., Guanciale, R., Strollo, D., Tuosto, E.: BPMN modelling of services with dynamically reconfigurable transactions. In: ICSOC'10. LNCS, vol. 6470, pp. 396–410 (2010)

3. Bruni, R., Butler, M.J., Ferreira, C., Hoare, C., Melgratti, H.C., Montanari, U.: Comparing two approaches to compensable flow composition. In: CONCUR'05. LNCS, vol. 3653, pp. 383–397 (2005)
4. Bruni, R., Ferrari, G.L., Melgratti, H.C., Montanari, U., Strollo, D., Tuosto, E.: From theory to practice in transactional composition of web services. In: EPEW/WS-FM'05. LNCS, vol. 3670, pp. 272–286 (2005)
5. Bruni, R., Ferreira, C., Kauer, A.K.: First-order dynamic logic for compensable processes. In: COORDINATION'12. LNCS, vol. 7274, pp. 104–121 (2012)
6. Bruni, R., Kersten, A., Lanese, I., Spagnolo, G.: A new strategy for distributed compensations with interruption in LRT. In: WADT'10. LNCS, vol. 7137, pp. 42–60 (2012)
7. Bruni, R., Melgratti, H., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: POPL'05. pp. 209–220. ACM (2005)
8. Bruni, R., Melgratti, H.C., Montanari, U.: Nested Commits for Mobile Calculi: Extending Join. In: IFIP-TCS'04. pp. 563–576 (2004)
9. Butler, M.J., Ferreira, C.: A process compensation language. In: IFM'00. LNCS, vol. 1945, pp. 61–76 (2000)
10. Butler, M.J., Hoare, C., Ferreira, C.: A trace semantics for long-running transactions. In: 25 Years of CSP. LNCS, vol. 3525, pp. 133–150 (2004)
11. Butler, M.J., Ripon, S.: Executable Semantics for Compensating CSP. In: EPEW/WS-FM'05. LNCS, vol. 3670, pp. 243–256 (2005)
12. Chen, Z., Liu, Z., Wang, J.: Failure-divergence refinement of compensating communicating processes. In: FM'11. LNCS, vol. 6664, pp. 262–277 (2011)
13. Danos, V., Krivine, J., Sobocinski, P.: General reversibility. Electr. Notes Theor. Comput. Sci. 175(3), 75–86 (2007)
14. de Vries, E., Koutavas, V., Hennessy, M.: Communicating transactions. In: CONCUR'10. LNCS, vol. 6269, pp. 569–583 (2010)
15. Eisentraut, C., Spieler, D.: Fault, compensation and termination in WS-BPEL 2.0 - a comparative analysis. In: WS-FM'08. LNCS, vol. 5387, pp. 107–126 (2008)
16. Ferreira, C., Lanese, I., Ravara, A., Vieira, H.T., Zavattaro, G.: Advanced mechanisms for service combination and transactions. In: Results of the SENSORIA Project, LNCS, vol. 6582, pp. 302–325. Springer (2011)
17. Garcia-Molina, H., Salem, K.: Sagas. In: SIGMOD. pp. 249–259. ACM Press (1987)
18. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: Dynamic error handling in service oriented applications. Fundam. Inform. 95(1), 73–102 (2009)
19. Johnsen, E.B., Lanese, I., Zavattaro, G.: Fault in the future. In: COORDINATION'11. LNCS, vol. 6721, pp. 1–15 (2011)
20. Lanese, I.: Static vs dynamic SAGAs. In: ICE'10. EPTCS, vol. 38, pp. 51–65 (2010)
21. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.B.: Controlling reversibility in higher-order pi. In: CONCUR'11. LNCS, vol. 6901, pp. 297–311 (2011)
22. Lanese, I., Zavattaro, G.: Programming sagas in SOCK. In: SEFM'09. pp. 189–198. IEEE Computer Society (2009)
23. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: FoSSaCS'05. LNCS, vol. 3441, pp. 282–298 (2005)
24. Phillips, I.C.C., Ulidowski, I.: Reversing algebraic process calculi. J. Log. Algebr. Program. 73(1-2), 70–96 (2007)
25. Qiu, Z., Wang, S., Pu, G., Zhao, X.: Semantics of BPEL4WS-like fault and compensation handling. In: FM'05. LNCS, vol. 3582, pp. 350–365 (2005)
26. Vaz, C., Ferreira, C., Ravara, A.: Dynamic recovering of long running transactions. In: TGC'08. LNCS, vol. 5474, pp. 201–215 (2008)