Verifying Compensating Transactions*

Michael Emmi and Rupak Majumdar

UC Los Angeles
mje,rupak@cs.ucla.edu

Abstract. We study the safety verification problem for business-process orchestration languages with respect to regular properties. Business transactions involve long-running distributed interactions between multiple partners which must appear as a single atomic action. This illusion of atomicity is maintained through programmer-specified *compensation actions* that get run to undo previous actions when certain parts of the transaction fail to finish. Programming languages for business process orchestration provide constructs for declaring compensation actions, which are co-ordinated by the run time system to provide the desired transactional semantics. The safety verification problem for business processes asks, given a program with programmer specified compensation actions and a regular language specifying "good" behaviors of the system, whether all observable action sequences produced by the program are contained in the set of good behaviors.
We show that the usual trace-based semantics for business process languages leads to an undecidable verification problem, but a tree-based semantics gives an algorithm that runs in time exponential in the size of the business process. Our constructions translate programs with compensations to tree automata with one memory.

1 Introduction

Long-running business processes involve hierarchies of interactive activities between possibly distributed partners whose execution must appear logically atomic to the environment. The long-running and interactive nature of business processes make traditional checkpointing and rollback mechanisms that guarantee transactional semantics [13] difficult or impossible to implement. For example, the long-running nature makes the performance penalties associated with locking unacceptable, and the interactive nature makes rollback impossible since some parts of the transaction (e.g., communications with external agents) are inherently impossible to undo automatically. Business processes therefore implement a weaker notion of atomicity based on *compensations*, programmer-specified actions that must be executed to semantically "undo" the effects of certain actions that cannot be undone automatically, should parts of the transaction fail to complete. A long-running transaction is structured as *sagas* [12],

* This research was sponsored in part by the grants NSF-CCF-0427202, NSF-CNS-0541606, and NSF-CCF-0546170.

a sequence of several smaller sub-transactions, each with an associated compensation. If one of the sub-transactions in the sequence aborts, the compensations associated with all committed subtransactions are executed in reverse order.

Flow composition or orchestration languages, such as WSCL [17], WSFL [18], BPML [2], and BPEL4WS [1], provide primitives for programming long-running transactions, including programmer-specified compensations and structured control flows. For example, BPEL4WS provides the `compensate` construct that can be used by the programmer to specify actions that must be taken if later actions fail. The formal semantics for these languages (or their core features) are given as extensions to process algebras with compensations (e.g., compensating CSP or cCSP) [6, 5] or transaction algebras with compensation primitives (called the *sagas calculus*) [4]. One central issue is to develop automatic static analysis techniques to increase confidence in the correctness of complex business processes implemented in these languages [14]. For example, in a business process implementing an e-commerce application, it may be desirable to verify that no product is shipped before a credit check is performed, or that the user's account is credited if it is found later that the order cannot be fulfilled. In this paper, we present model checking algorithms for the automatic verification of *temporal safety properties* of flow composition languages with compensations. We take the automata theoretic approach and specify safety properties as *regular sets* of traces of observable actions. Then, the verification problem can be formulated as a *language containment* question: check that any trace that can be produced by the execution of a saga also belongs to the set of "good" behaviors prescribed by the specification.

Our starting point is the sagas calculus [4], although our results generalize to most other languages with similar core features. We show that the safety verification problem for programs in the sagas calculus and safety properties encoded as finite word automata is undecidable in the usual trace-based semantics [6, 3]. On the other hand, perhaps surprisingly, the verification problem becomes decidable (in time exponential in the size of the sagas program) if we associate a *tree semantics* with the execution. The tree semantics exposes more structure on the sequence of observable actions by making the sequential or parallel operator at each intermediate step observable. For the tree semantics, we consider safety properties encoded as regular tree languages, rather than word languages. The key hurdle is that the tree language of a sagas program is not regular: this is intuitively clear since first, the compensations are dynamically pushed on to a (possibly unbounded) stack, and second, the actions on the execution path up to an abort are related to the compensation actions thereby requiring comparisons on sibling subtrees.

Our main technical tool consists of tree automata with one memory [7], that generalize finite tree automata by allowing a memory element which is built up as the automaton walks a tree bottom-up and which can be compared across children. Specifically, we show that the tree language of any program in the sagas language is accepted by a tree automaton with one memory. Tree automata with one memory generalize pushdown automata over words and tree automata with equality tests [8]. However, their emptiness problem is decidable [7], and they are closed under intersection with finite tree automata. Our construction, together with the above properties of tree automata with one memory, provides a decision procedure for the safety verification problem. While automatic model checking techniques for web services and business process applications have been proposed before [10, 11, 9], to the best of our knowledge, we provide the first automatic handling of compensation stacks.

2 Sagas with Compensation

A *saga* is a high-level description of the interaction between components for web services. The building blocks of sagas are atomic actions, which execute without communication from other services. In addition, to each atomic action is attached a (possibly null) *compensation*, which is executed if the action succeeds but a later action in the saga does not complete successfully. Sagas are then built from (compensated) atomic actions, sequential and parallel composition, nondeterministic choice, and nesting. The execution order for compensating actions is determined by interpreting each sequential composition operator in reverse.

More formally, given an alphabet Σ of atomic actions containing a special *null action* 0, and a set of variable names X, the set of *transaction terms* $\mathbb{T}^{\Sigma,X}$ *over* Σ *and* X is the smallest set which includes the *atomic terms* $a \div b$, for $a, b \in \Sigma$, the variables $x \in X$, and is closed under binary operators for sequential composition (;), parallel composition (||), and nondeterministic choice ($\oplus$), and the unary saga-nesting operator $\{\![\cdot]\!\}$. The binary expression $a \div b$ attaches to a the compensating action b. The operators $\|$ and $\oplus$ are commutative and associative, while the sequential operator ; is defined here to be left-associative. We refer to terms of the form $\{\![t]\!\}$ as *transactions*, and use $\mathbb{T}^{\Sigma,X}_{\{\![\cdot]\!\}}$ to denote the set of transactions. For an atomic action $a \in \Sigma$, we abbreviate $a \div 0$ with a.

A saga is given as a tuple $\mathcal{S} = \langle \Sigma, X, s_0, T \rangle$, where $T : X \to \mathbb{T}^{\Sigma,X}_{\{\![\cdot]\!\}}$ maps variables to transactions, and $s_0 \in X$ determines *the top-level transaction.* We frequently abuse the notation and write $x = \{\![t]\!\}$ in place of $T(x) = \{\![t]\!\}$, and as Σ, X, and T are usually understood from the context, we often refer to a saga by its transaction variable s_0. We refer to any term of the form $\{\![t]\!\} \neq s_0$ as a *nested saga* or *subtransaction.*

Example 1. The sagas calculus is capable of expressing realistic long-running business transactions. Suppose AcceptOrder, Restock, FulfilledOK, CreditCheck, CreditOK, BookCourier, CancelCourier and PackOrder are atomic actions with the obvious meanings, and consider the saga

$$
\begin{aligned}
&Main = \{\![\ (\textsc{AcceptOrder} \div \textsc{Restock}); FulfillOrder; \textsc{FulfilledOK}\ ]\!\} \\
&FulfillOrder = \{\![\ WarehousePackaging \parallel \textsc{CreditCheck}; \textsc{CreditOK}\ ]\!\} \\
&WarehousePackaging = \\
&\qquad\qquad \{\![\ (\textsc{BookCourier} \div \textsc{CancelCourier}) \| \textsc{PackOrder}\ ]\!\}.
\end{aligned}
$$

$$\textsc{(null)}\quad \langle 0,\beta\rangle \xrightarrow{0} \langle \boxdot,\beta\rangle$$

$$\textsc{(atom-s)}\quad \langle a \div b,\beta\rangle \xrightarrow{a} \langle \boxdot, b;\beta\rangle$$

$$\textsc{(atom-f)}\quad \frac{\langle \beta,0\rangle \xrightarrow{\alpha} \langle \boxdot,0\rangle}{\langle a \div b,\beta\rangle \xrightarrow{\alpha} \langle \boxtimes,0\rangle}$$

$$\textsc{(atom-a)}\quad \frac{\langle \beta,0\rangle \xrightarrow{\alpha} \langle \boxtimes,0\rangle}{\langle a \div b,\beta\rangle \xrightarrow{\alpha} \langle \boxast,0\rangle}$$

$$\textsc{(seq-s)}\quad \frac{\langle P,\beta\rangle \xrightarrow{\alpha} \langle \boxdot,\beta''\rangle \quad \langle Q,\beta''\rangle \xrightarrow{\alpha'} \langle \Box,\beta'\rangle}{\langle P;Q,\beta\rangle \xrightarrow{\alpha;\alpha'} \langle \Box,\beta'\rangle}$$

$$\textsc{(seq-fa)}\quad \frac{\langle P,\beta\rangle \xrightarrow{\alpha} \langle \Box,0\rangle \quad \Box \in \{\boxtimes,\boxast,\overline{\boxtimes},\overline{\boxast}\}}{\langle P;Q,\beta\rangle \xrightarrow{\alpha} \langle \Box,0\rangle}$$

$$\textsc{(par-s)}\quad \frac{\langle P,0\rangle \xrightarrow{\alpha} \langle \boxdot,\beta'\rangle \quad \langle Q,0\rangle \xrightarrow{\alpha'} \langle \boxdot,\beta''\rangle}{\langle P\|Q,\beta\rangle \xrightarrow{\alpha\|\alpha'} \langle \boxdot,\beta'\|\beta'';\beta\rangle}$$

$$\textsc{(par-f)}\quad \frac{\begin{array}{ll}\langle P,0\rangle \xrightarrow{\alpha} \langle \Box_P,0\rangle & \Box_P,\Box_Q \in \{\boxtimes,\overline{\boxtimes}\}\\ \langle Q,0\rangle \xrightarrow{\alpha'} \langle \Box_Q,0\rangle & \langle \beta,0\rangle \xrightarrow{\alpha''} \langle \Box_\beta,0\rangle\end{array}}{\langle P\|Q,\beta\rangle \xrightarrow{(\alpha\|\alpha');\alpha''} \langle \Box_P \wedge \Box_Q \wedge \mathsf{force}(\Box_\beta),0\rangle}$$

$$\textsc{(par-a)}\quad \frac{\begin{array}{ll}\langle P,0\rangle \xrightarrow{\alpha} \langle \Box_P,0\rangle & \Box_P \in \{\boxast,\overline{\boxast}\}\\ \langle Q,0\rangle \xrightarrow{\alpha'} \langle \Box_Q,0\rangle & \Box_Q \in \{\boxtimes,\boxast,\overline{\boxtimes},\overline{\boxast}\}\end{array}}{\langle P\|Q,\beta\rangle \xrightarrow{\alpha\|\alpha'} \langle \Box_P \wedge \Box_Q,0\rangle}$$

$$\textsc{(nondet)}\quad \frac{\langle P,\beta\rangle \xrightarrow{\alpha} \langle \Box,\beta'\rangle}{\langle P \oplus Q,\beta\rangle \xrightarrow{\alpha} \langle \Box,\beta'\rangle}$$

$$\textsc{(var)}\quad \frac{\langle T(x),\beta\rangle \xrightarrow{\alpha} \langle \Box,\beta'\rangle}{\langle x,\beta\rangle \xrightarrow{\alpha} \langle \Box,\beta'\rangle}$$

$$\textsc{(saga)}\quad \frac{\langle P,0\rangle \xrightarrow{\alpha} \langle \Box,\beta\rangle}{\{\![P]\!\} \xrightarrow{\alpha} \Box}$$

$$\textsc{(sub-s)}\quad \frac{\langle P,0\rangle \xrightarrow{\alpha} \langle \boxdot,\beta'\rangle}{\langle \{\![P]\!\},\beta\rangle \xrightarrow{\alpha} \langle \boxdot,\beta';\beta\rangle}$$

$$\textsc{(sub-f)}\quad \frac{\langle P,0\rangle \xrightarrow{\alpha} \langle \boxtimes,0\rangle}{\langle \{\![P]\!\},\beta\rangle \xrightarrow{\alpha} \langle \boxdot,\beta\rangle}$$

$$\textsc{(sub-a)}\quad \frac{\langle P,0\rangle \xrightarrow{\alpha} \langle \boxast,0\rangle}{\langle \{\![P]\!\},\beta\rangle \xrightarrow{\alpha} \langle \boxast,0\rangle}$$

$$\textsc{(sub-forced-f)}\quad \frac{\langle P,0\rangle \xrightarrow{\alpha} \langle \overline{\boxtimes},0\rangle \quad \langle \beta,0\rangle \xrightarrow{\alpha'} \langle \Box,0\rangle}{\langle \{\![P]\!\},\beta\rangle \xrightarrow{\alpha;\alpha'} \langle \mathsf{force}(\Box),0\rangle}$$

$$\textsc{(sub-forced-a)}\quad \frac{\langle P,0\rangle \xrightarrow{\alpha} \langle \overline{\boxast},0\rangle}{\langle \{\![P]\!\},\beta\rangle \xrightarrow{\alpha} \langle \overline{\boxast},0\rangle}$$

$$\textsc{(forced)}\quad \frac{\langle \beta,0\rangle \xrightarrow{\alpha} \langle \Box,0\rangle}{\langle P,\beta\rangle \xrightarrow{\alpha} \langle \mathsf{force}(\Box),0\rangle}$$

Table 1. The formal semantics for a saga $\mathcal{S} = \langle \Sigma, X, s_0, T\rangle$. The symbols P and Q range over transaction terms, a and b range over the atomic actions of Σ, x ranges over the variables of X, $\alpha, \alpha', \alpha''$ range over observations, β, β', β'' range over compensation stacks, and $\Box, \Box_P, \Box_Q$ range over outcomes.

The saga *Main* encodes a long running business transaction where an order is deemed a success upon the success of order placement, credit check, courier booking, and packaging. If some action were to fail during the transaction, then compensations would be run for the previously completed actions. For example, if the packaging were to fail after the credit check and courier booking had completed, then the courier booking would be canceled, and the order restocked.

The operational semantics of sagas are shown in Table 1. To reduce the number of rules, we define them up to structural congruence implied by the associativity of ;, $\|$ and $\oplus$, the commutativity of $\|$ and $\oplus$, as well as the identities $0;P \equiv P;0 \equiv P$ and $P\|0 \equiv 0\|P \equiv P$, for transaction terms P. The execution of a saga leads to an *outcome* which is either success, failure, or abortion, represented by the boxed symbols $\boxdot$, $\boxtimes$ and $\boxast$ respectively. The semantics is given for a fixed set of variables X and mapping $T : X \to \mathbb{T}^{\Sigma,X}_{\{\![\cdot]\!\}}$ from variables to sub-transactions.

An *observation* is a term constructed from atomic actions and the sequential and parallel composition operators. The semantics of sagas is given by the

$\wedge$	$\boxdot$	$\boxtimes$	$\boxast$	$\overline{\boxtimes}$	$\overline{\boxast}$
$\boxdot$	$\boxdot$				
$\boxtimes$	—	$\boxtimes$			
$\boxast$	—	$\boxast$	$\boxast$		
$\overline{\boxtimes}$	—	$\boxtimes$	$\boxast$	$\overline{\boxtimes}$	
$\overline{\boxast}$	—	$\boxast$	$\boxast$	$\overline{\boxast}$	$\overline{\boxast}$

Table 2. The composition operation $\wedge$.

rule (SAGA), whose consequent $\{\![P]\!\} \xrightarrow{\alpha} \Box$ specifies that the execution of transaction $\{\![P]\!\}$ results in outcome $\Box$, emitting the observation α. The semantics relation uses an auxiliary relation $\langle t, \beta \rangle \xrightarrow{\alpha} \langle \Box, \beta' \rangle$ which dictates that the execution of term t results in the outcome $\Box$, while the initial compensations β are destructively replaced by the compensations β'. The observation α in these relations describes the flow of control while t is executed.

The special symbols $\overline{\boxtimes}$ and $\overline{\boxast}$ are the forced failure and abortion outcomes, and result from failure or abortion in a parallel thread of execution. When a thread encounters failure, the entire transaction must subsequently fail. When each thread can complete its compensations, the resulting outcome is $\overline{\boxtimes}$; otherwise $\overline{\boxast}$ results. The (associative and commutative) binary operator $\wedge$ over the set $\{\boxdot, \boxtimes, \boxast, \overline{\boxtimes}, \overline{\boxast}\}$ determines the outcome of two branches executing in parallel. Its definition is given in Table 2 (since $\wedge$ is commutative, only half the table is displayed). The auxiliary function $\mathsf{force} : \{\boxdot, \boxtimes, \boxast, \overline{\boxtimes}, \overline{\boxast}\} \to \{\overline{\boxtimes}, \overline{\boxast}\}$ is given by $\mathsf{force}(\boxdot) = \overline{\boxtimes}$, and $\mathsf{force}(\Box) = \overline{\boxast}$, for $\Box \in \{\boxtimes, \boxast, \overline{\boxtimes}, \overline{\boxast}\}$.

We briefly describe the operational semantics given in Table 1, for a more detailed discussion, see [4]. The rule (NULL) says that the null process never fails. The rules (ATOM-S), (ATOM-F), and (ATOM-A) deal with atomic action execution. If action a succeeds, rule (ATOM-S) installs the compensation b on the compensation stack. If a fails (rules (ATOM-F) and (ATOM-A)) when the currently installed compensation is β, then β should be executed. If all compensating actions of β execute successfully (as in (ATOM-F)), then the outcome for the term $a \div b$ is $\boxtimes$; if some compensating action of β fails (as in (ATOM-A)), then the outcome for the term $a \div b$ is $\boxast$.

The rules (SEQ-S) and (SEQ-FA) execute the sequential composition of two terms. The rule (PAR-S) declares the order in which compensations from parallel branches are executed. When the terms P and Q result in the compensations β' and β'', then the term $P||Q$ results in the compensation $\beta'||\beta''; \beta$, where β is the compensation for actions before $P||Q$. The associated rules (PAR-F) and (PAR-A) deal with failure on parallel branches and failed compensation after failure on parallel branches respectively. Rule (VAR) executes the term bound to a variable by T, rule (NONDET) executes one branch of a nondeterministic choice, and (FORCED) allows a thread to fail due to the failed execution of another. The remaining rules specify the semantics of nested sagas.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle a \div b, 0\rangle \xrightarrow{a} \langle \boxdot, b\rangle_1} \quad \dfrac{\dfrac{}{\langle b \div 0, 0\rangle \xrightarrow{b} \langle \boxdot, 0\rangle_1}}{\langle c \div d, b\rangle \xrightarrow{b} \langle \boxtimes, 0\rangle_2}}{\langle a \div b; c \div d, 0\rangle \xrightarrow{a;b} \langle \boxtimes, 0\rangle_3} \quad \dfrac{\dfrac{}{\langle e \div 0, 0\rangle \xrightarrow{e} \langle \boxdot, 0\rangle_1} \quad \dfrac{\dfrac{}{\langle 0, 0\rangle \xrightarrow{0} \langle \boxdot, 0\rangle_6}}{\langle 0, 0\rangle \xrightarrow{0} \langle \overline{\boxtimes}, 0\rangle_5}}{\langle e \div 0, 0\rangle \xrightarrow{e} \langle \overline{\boxtimes}, 0\rangle_3}}{\langle a \div b; c \div d \| e \div 0, 0\rangle \xrightarrow{a;b\|e} \langle \boxtimes, 0\rangle_{4,*}}}{\{\![a \div b; c \div d \| e \div 0]\!\} \xrightarrow{a;b\|e} \boxtimes_7}$$

Fig. 1. An execution of saga s_0 from example 2. The corresponding rules from table 1 are (1) ATOM-S, (2) ATOM-F, (3) SEQ-S, (4) PAR-F, (5) FORCED, (6) NULL, and (7) SAGA. An additional application of (NULL) is omitted in $(*)$.

Example 2. Consider the saga $\langle\{a, b, c, d, e, 0\}, \{s_0\}, s_0, T\rangle$ with $T(s_0) = \{\![a \div b; c \div d \| e \div 0]\!\}$. That is, the action a occurs before c, while e occurs in parallel. If the action c were to fail, then the completed actions a, and possibly e, are to execute their compensations. Since e has a null compensation, only b would be executed. Figure 1 shows an execution of the saga, where c fails after a and e have both executed, and a's compensation b is run successfully.

3 Trace Semantics

From the operational semantics of a saga and a fixed environment, we define a *trace language*, containing all sequences of observations that may be generated from an execution. The function trace defines this language by induction over the structure of observations, as generated by the execution of a saga, as

$$\begin{aligned} \mathsf{trace}(a) &= \{a\} \\ \mathsf{trace}(t_1; t_2) &= \mathsf{trace}(t_1) \circ \mathsf{trace}(t_2) \\ \mathsf{trace}(t_1 \| t_2) &= \mathsf{trace}(t_1) \otimes \mathsf{trace}(t_2) \end{aligned}$$

where $\circ$ and $\otimes$ denote the concatenation and interleaving composition of languages: $L_1 \circ L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1, w_2 \in L_2\}$ and $L_1 \otimes L_2 = \{x_1 y_1 \ldots x_k y_k \mid x_1 \ldots x_k \in L_1, y_1 \ldots y_k \in L_2\}$.

Let $\mathcal{S} = (\Sigma, X, s_0, T)$ be a saga. The *trace language* $L_W(s)$ of a variable $s \in X$ is defined as $\{\mathsf{trace}(\alpha) \mid \exists \boxdot . \{\![T(s)]\!\} \xrightarrow{\alpha} \boxdot\}$. The trace language $L_W(\mathcal{S})$ of a saga $\mathcal{S}$ is the language $L_W(s_0)$. Clearly, the language $L_W(\mathcal{S})$ may not be regular.

Unfortunately, the trace language of sagas is unsuitable for verification since, as Theorem 1 shows, language inclusion in a regular set is undecidable. While we state the theorem for sagas, a similar theorem also holds for other compensable flow composition languages such as cCSP [6].

Theorem 1. *The language inclusion problem $L_W(\mathcal{S}) \subseteq L_R$, for an input saga $\mathcal{S} = \langle \Sigma, X, s_0, T\rangle$ and regular language $L_R \subseteq \Sigma^*$, is undecidable.*

Proof. We proceed by reduction from the halting problem of 2-counter machines, similarly to the proof for process algebras in [15]. Let $\mathcal{M}$ be a 2-counter machine with n numbered instructions: $\langle 1 : \mathbf{ins}_1 \rangle \ldots \langle n-1 : \mathbf{ins}_{n-1} \rangle \langle n : \mathbf{halt} \rangle$ where each $\mathbf{ins}_k$ for $k \in \{1, \ldots, n-1\}$ is either $c_j = c_j + 1;$ **goto** ℓ, or $c_j = c_j - 1;$ **goto** ℓ, or **if** $c_j = 0$ **then goto** ℓ **else goto** ℓ', for $j \in \{1, 2\}$. Furthermore, let $\Sigma = \{zero_k, inc_k, dec_k \mid k \in \{1,2\}\} \cup \{0\}$, where $zero_k$, inc_k, and dec_k stand for zero-assertion, increment, and decrement actions, respectively, where we associate (in the obvious way) traces of $\mathcal{M}$ with Σ-sequences. We construct a saga $\mathcal{S}$, whose language is the Σ-sequences corresponding to traces of $\mathcal{M}$, irrespective of $\mathcal{M}$'s control location, and a finite state automaton $\mathcal{A}$, whose language is the Σ-sequences corresponding to traces of $\mathcal{M}$, irrespective of $\mathcal{M}$'s counter values. With these constructions, the intersection $L_W(\mathcal{A}) \cap L(\mathcal{S})$ is the language of Σ-sequences corresponding to traces of $\mathcal{M}$.

We define $\mathcal{S}$ to be the saga $\langle \Sigma, X, s_0, T \rangle$, where $X = \{C_k, Z_k \mid k \in \{1,2\}\} \cup \{s_0\}$ are the variables of $\mathcal{S}$, s_0 is the top-level transaction, and T, mapping variables to transaction terms, is given by

$$\left.\begin{array}{l} Z_k \mapsto \{[(zero_k; Z_k) \oplus (inc_k; C_k; Z_k)]\} \\ C_k \mapsto \{[(inc_k \div dec_k; C_k; C_k)]\} \end{array}\right\} \text{ for } k \in \{1,2\}$$

and $s_0 \mapsto \{[(C_1^{m_1}; Z_1) || (C_2^{m_2}; Z_2)]\}$. Intuitively, the transaction term $T(C_k)$ defines a state of $\mathcal{M}$ which attempts to decrease the value of counter k by one, the transaction term $T(Z_k)$ defines a state which holds the value of counter k at 0, and the term $T(s_0)$ defines a state in which the counter values start at m_1 and m_2 respectively. Finite traces of $\mathcal{S}$ exist, since any action may fail, and if C_k ever fails its compensating action of dec_k, then the entire transaction s_0 aborts. Notice that traces of $\mathcal{S}$ correspond to runs of a "stateless" $\mathcal{M}$, where every step could execute any instruction.

The finite state machine $\mathcal{A}$ over alphabet Σ has states $\{1, 2, \ldots, n+1\}$, one for each instruction, and a sink state $n+1$. The transitions are given by the instructions of $\mathcal{M}$ as follows. If instruction i is an increment (resp., decrement) of counter c_j followed by a move to ℓ, then $\mathcal{A}$ has a transition $\langle i, inc_j, \ell \rangle$ (resp., $\langle i, dec_j, \ell \rangle$). If i moves to ℓ when $c_j = 0$, and ℓ' otherwise, then $\mathcal{A}$ has the transitions $\langle i, zero_j, \ell \rangle$ and $\langle i, 0, \ell' \rangle$. Each state also has a self loop on the action 0. The automaton is completed by adding a transition $\langle k, \sigma, n+1 \rangle$ for each state k in which σ is otherwise not enabled (note that this construction induces a self loop $\langle n+1, \sigma, n+1 \rangle$ for all $\sigma \in \Sigma$). Every state of $\mathcal{A}$ is accepting, however since n has no enabled actions, the language of $\mathcal{A}$ is not universal. In particular, $\mathcal{A}$'s language does not include Σ-sequences whose (proper) prefixes correspond to halting computations of $\mathcal{M}$. Notice that the traces of $\mathcal{S}$ correspond to runs of a "memoryless" $\mathcal{M}$, where every step ignores the values of the counters.

It only remains to check that $L_W(\mathcal{S}) \not\subseteq L(\mathcal{A})$ if and only if $\mathcal{M}$ has a halting computation. First suppose that $L_W(\mathcal{S}) \not\subseteq L(\mathcal{A})$, and let $w \in \mathrm{L}_W(\mathcal{S}) \setminus L(\mathcal{A})$. Since w is not accepted by $\mathcal{A}$, a prefix of w corresponds to a halting trace of $\mathcal{M}$ (recall that $\mathcal{A}$ must have moved to state n) consistent with $\mathcal{M}$'s control. Since w is accepted by $\mathcal{S}$, w also corresponds to a trace of $\mathcal{M}$ consistent with $\mathcal{M}$'s counter values. Thus $\mathcal{M}$ has a halting computation. On the other hand, if $\mathcal{M}$ has a halting computation then $\mathcal{A}$ rejects a Σ-sequence with a prefix corresponding to a halting trace of $\mathcal{M}$, which is a trace of $\mathcal{S}$ since the counter values are necessarily consistent; thus $L_W(\mathcal{S}) \nsubseteq L(\mathcal{A})$. Thus, the language inclusion problem $L_W(\mathcal{S}) \subseteq L(\mathcal{A})$ is undecidable. $\square$

4 Tree Semantics

In this section we give an alternative interpretation to the set of observations given by a saga. Instead of interpreting executions as flattened sequences of actions, we interpret them as trees where the actions become leaves, and the composition operators become internal nodes. We then give an automata-theoretic classification of a sagas by building tree automata which recognize the set of trees representing valid executions.

4.1 Yield Language of a Saga

Trees. A (ranked) alphabet is a tuple $\langle \mathcal{F}, \mathsf{ar} \rangle$ where $\mathcal{F}$ is a finite alphabet, and ar is a map, called *arity*, from $\mathcal{F}$ to $\mathbb{N}$. The set of symbols from $\mathcal{F}$ of arity k (i.e., $\{f \in \mathcal{F} \mid \mathsf{ar}(f) = k\}$) is denoted $\mathcal{F}_k$. The set of symbols of arity zero are called constants; arity one symbols (resp. two, k) are called unary (resp. binary, k-ary) symbols. In what follows, we assume $\mathcal{F}$ has at least one constant, i.e., $\mathcal{F}_0 \neq \emptyset$. For ease of notation we write $\mathcal{F}$, omitting ar, by assuming that the arity information is encoded into each symbol in $\mathcal{F}$.

A finite ordered tree t over a ranked alphabet $\mathcal{F}$ is a mapping from a prefix-closed set $\mathrm{dom}(t) \subseteq \mathbb{N}^*$ to $\mathcal{F}$, such that (1) each leaf is mapped to a constant: for all $p \in \mathrm{dom}(t)$, we have $t(p) \in \mathcal{F}_0$ iff $\{j \mid p \cdot j \in \mathrm{dom}(t)\} = \emptyset$; and (2) each internal node mapped to symbol $f \in \mathcal{F}_k$ has exactly k children numbered $1, \ldots, k$: for all $p \in \mathrm{dom}(t)$, if $t(p) \in \mathcal{F}_k$ and $k \geq 1$, then $\{j \mid p \cdot j \in \mathrm{dom}(t)\} = \{1, \ldots, k\}$. The set of all trees over alphabet $\mathcal{F}$ is denoted $\mathsf{Trees}(\mathcal{F})$. A set of trees is a *tree language*.

Yield Language. Fix the saga $\mathcal{S} = \langle \Sigma, X, s_0, T \rangle$, and let $\mathcal{F}$ be a ranked alphabet consisting of a constant symbol for each atomic action of Σ, as well as the binary symbols $\sigma_;$ and $\sigma_\|$.

Given an observation α from an execution of $\mathcal{S}$, the set $\mathsf{yield}(\alpha)$ of *yield trees* over the tree alphabet $\mathcal{F}$ is defined inductively as

$$\begin{aligned}
\mathsf{yield}(a) &= \{a\} \\
\mathsf{yield}(t_1; t_2) &= \{\sigma_;(t_1', t_2') \mid t_1' \in \mathsf{yield}(t_1), t_2' \in \mathsf{yield}(t_2)\} \\
\mathsf{yield}(t_1 \| t_2) &= \{\sigma_\|(t_1', t_2') \mid t_1' \in \mathsf{yield}(t_1), t_2' \in \mathsf{yield}(t_2)\}
\end{aligned}$$

where $\sigma(t_1, t_2)$ denotes the tree with a σ-labeled root whose left and right children are the roots of the trees t_1 and t_2 respectively, and a is an atomic action of Σ. Informally, a yield tree considers the term α as a finite ordered tree over the

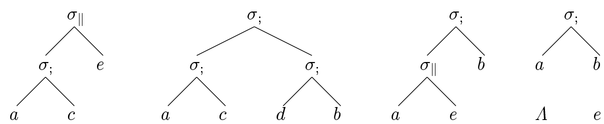

Fig. 2. The yield language of $\{\![a \div b; c \div d \| e \div 0]\!\}$.

alphabet of atomic actions and the sequential and parallel compositions. The *yield language* of $\mathcal{S}$, denoted $L(\mathcal{S})$, is the set

$$L(\mathcal{S}) = \bigcup_{\{\![T(s_0)]\!\} \xrightarrow{\alpha} \square} \mathsf{yield}(\alpha).$$

Example 3. The yield language of the saga $s_0 = \{\![a \div b; c \div d \| e \div 0]\!\}$ from example 2, consisting of six trees, is shown in Figure 2.

4.2 Tree Automata with One Memory

Tree Automata. A *finite tree automaton* over $\mathcal{F}$ is a tuple $\mathcal{A} = \langle Q, \mathcal{F}, Q_f, \Delta \rangle$ where Q is a finite set of states, $\mathcal{F}$ is a finite alphabet, $Q_f \subseteq Q$ is a set of final states, and Δ is a set of transitions (i.e., a relation) of the form $f(q_1, \ldots, q_k) \to q$ for $q, q_1, \ldots, q_k \in Q$, $f \in \mathcal{F}_k$.

A *run* of $\mathcal{A}$ on a tree t is a labeling $r : \mathrm{dom}(t) \to Q$ such that $t(\ell) \to r(\ell) \in \Delta$ for each leaf ℓ, and $t(n)(r(n \cdot 1), \ldots, r(n \cdot k)) \to r(n) \in \Delta$ for each internal node n. A run r is *accepting* if $r(\Lambda) \in Q_f$, and we say that a tree t is *accepted* by $\mathcal{A}$ if there exists an accepting run of $\mathcal{A}$ on t. The *language* of $\mathcal{A}$, denoted $L(\mathcal{A})$ is the set of trees which are accepted by $\mathcal{A}$. A tree language L is *regular* if there exists a finite tree automaton $\mathcal{A}$ such that $L = L(\mathcal{A})$.

Example 4. Regular tree languages can specify many interesting properties of sagas. For example, the property "all b-actions occur (sequentially) after all a-actions," over the actions $\{a, b, c\}$, is specified by the tree automaton $\mathcal{A} = \langle \{q_a, q_b, q_c, q_f\}, \{a, b, c, \sigma_;, \sigma_\|\}, \{q_a, q_b, q_c, q_f\}, \Delta \rangle$, where Δ contains the transitions:

$$\begin{array}{cccc}
 & a \to q_a \quad b \to q_b \quad c \to q_c & & \\
 & \sigma_;(q_a, q_b) \to q_f \quad \sigma_;(q_a, q_f) \to q_f \quad \sigma_;(q_f, q_b) \to q_f & & \\
\sigma_\|(q_c, q) \to q & \sigma_\|(q, q_c) \to q & \sigma_\|(q_a, q_a) \to q_a & \sigma_\|(q_b, q_b) \to q_b \\
\sigma_;(q_c, q) \to q & \sigma_;(q, q_c) \to q & \sigma_;(q_a, q_a) \to q_a & \sigma_;(q_b, q_b) \to q_b
\end{array}$$

for $q \in \{q_a, q_b, q_c, q_f\}$. Given a tree $t \in L(\mathcal{A})$ where both a and b occur (as leaves) in t, let r be an accepting run of $\mathcal{A}$ on t. The transitions of $\mathcal{A}$ ensure that there is some internal node n such that every ancestor of n is labeled with q_f, and no descendant of n is labeled with q_f. This path of q_f-labeled nodes in r divides t: no a's (b's, resp.) can occur in a right-subtree (left-subtree, resp.) of a q_f-labeled node. On the other hand, every such tree is accepted by $\mathcal{A}$.

Unfortunately, as Example 5 shows, the yield language of a saga may be non-regular, and hence we must expand the expressive power of finite tree automata to model the yield language of sagas. The extended model we consider allows a tree automaton to use an arbitrarily large memory.

Example 5. Consider the simple saga $\mathcal{S} = \langle \{a, b\}, \{s\}, s, s \mapsto \{\![s \| a \div b]\!\}\rangle$ for which any finite run must reach a failure, resulting in an observation of a's followed sequentially by b's. The yield language $L(\mathcal{S})$ consists of the set of binary trees where the root node is a sequential composition, and each subtree's internal nodes are parallel compositions. Every leaf of the left subtree is labeled with a, while every leaf of the right subtree is labeled with b, and there are at least as many a's as b's. This tree language is not regular.

Tree Automata with One Memory. A more powerful family of tree automata can be obtained by extending the finite tree automata with a tree-structured memory for which equality of the memory built from subtrees can be enforced. Given a ranked memory alphabet Γ, define Φ_Γ as the smallest set of composition-closed functions over $\mathsf{Trees}(\Gamma)$ where (1) if $f \in \Gamma_n$ then the *constructor function* $\lambda x_1, \ldots, x_n.f(x_1, \ldots, x_n)$ is in Φ_Γ; (2) if $n \in \mathbb{N}$ and $0 < i \leq n$, then the *projection function* $\lambda x_1, \ldots, x_n.x_i$ is in Φ_Γ; (3) if $f \in \Gamma_n$ and $0 < i \leq n$, then the *pattern matching (partial) function* that associates each term $f(t_1, \ldots, t_n)$ with t_i, written $\lambda f(x_1, \ldots, x_n).x_i$, is in Φ_Γ. A *tree automaton with one memory* (TAWOM) [7] $\mathcal{A} = \langle \mathcal{F}, \Gamma, Q, Q_f, \Delta\rangle$ consists of an input alphabet $\mathcal{F}$, an alphabet Γ of memory symbols, a set Q of states, a set $Q_f \subseteq Q$ of final states, and a transition relation Δ. The transition relation Δ is given as a set of transitions of the form

$$f(q_1, \ldots, q_n) \xrightarrow[F]{c} q$$

where $q_1, \ldots, q_n, q \in Q$, $f \in \mathcal{F}_n$, $c \subseteq \{1, \ldots, n\}^2$ defines an equivalence relation of index m on $\{1, \ldots, n\}$, and $\lambda x_1, \ldots, x_m.F(x_1, \ldots, x_m)$ is a function from Φ_Γ. We often denote the function $\lambda \boldsymbol{x}.F(\boldsymbol{x}) \in \Phi_\Gamma$ simply as F, and the composition of functions $F, G \in \Phi_\Gamma$ (when F and G are naturally composable) as $F \cdot G$.

A *configuration* of $\mathcal{A}$ is a pair $\langle q, \gamma\rangle$ of a state $q \in Q$ and memory term $\gamma \in \mathsf{Trees}(\Gamma)$. Intuitively, a TAWOM constructs a configuration in a bottom-up manner, computing the new memory state from the memory states of each child. The transitions also check for equality between the children's memory states, based on the given equivalence relation.

A *run* of $\mathcal{A}$ is a labeling $r : \mathrm{dom}(t) \to Q \times \mathsf{Trees}(\Gamma)$ such that for each leaf ℓ,

$$t(\ell) \xrightarrow[r(\ell)_2]{} r(\ell)_1 \in \Delta$$

(where we use subscripts for tuple indexing), and for each internal node n with $t(n)$ of arity k, there exists $F \in \Phi_\Gamma$ such that

$$t(n)(r(n \cdot 1)_1, \ldots, r(n \cdot k)_1) \xrightarrow[F]{c} r(n)_1 \in \Delta$$
$$\text{and} \quad F(r(n \cdot 1)_2, \ldots, r(n \cdot m)_2) = r(n)_2,$$

where c is of index m, and $r(n \cdot i)_2 = r(n \cdot j)_2$ when $i \equiv_c j$, for $i, j \in \{1, \dots, k\}$. A run r is *accepting* if $r(\Lambda) \in Q_f$, and the *language* of $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of trees on which there exist accepting runs of $\mathcal{A}$.

Example 6. TAWOM can encode pushdown automata [7]: a transition $(q, \alpha \cdot \gamma) \xrightarrow{a} (q', \beta \cdot \gamma)$ from the state q and stack $\alpha \cdot \gamma$ on letter a to the state q' and stack $\beta \cdot \gamma$ can be written (considering letters as unary symbols) as $a(q) \xrightarrow[\lambda x.\beta\alpha^{-1}x]{} q'$.

Example 7. The yield language of the saga $\{\![s \| a \div b]\!\}$ from Example 5 is accepted by the automaton $\mathcal{A} = \langle \{a, b, \sigma_{\|}, \sigma_{;}\}, \{\gamma_0, \gamma\}, \{q_a, q_b, q_f\}, \{q_a, q_f\}, \Delta \rangle$ where Δ contains the following transitions:

$$\sigma_{\|}(q_a, q_a) \xrightarrow[\lambda x_1, x_2.\gamma(x_1, x_2)]{\top} q_a \qquad \sigma_{\|}(q_b, q_b) \xrightarrow[\lambda x_1, x_2.\gamma(x_1, x_2)]{\top} q_b$$

$$\sigma_{\|}(q_a, q_a) \xrightarrow[\lambda x_1, x_2.x_1]{\top} q_a \qquad \sigma_{\|}(q_a, q_a) \xrightarrow[\lambda x_1, x_2.x_2]{\top} q_a$$

$$\sigma_{;}(q_a, q_b) \xrightarrow{1=2} q_f \qquad a \xrightarrow[\gamma_0]{\top} q_a \qquad b \xrightarrow[\gamma_0]{\top} q_b$$

Note that the equivalence relation $\{\langle 1,1 \rangle, \langle 1,2 \rangle, \langle 2,1 \rangle, \langle 2,2 \rangle\}$ is here denoted by $1 = 2$, and $\top$ denotes the identity relation. An accepting run of $\mathcal{A}$ can successfully match the right subtree's memory state with the left subtree's memory state, as $\sigma_{;}$ is consumed, only when there are least as many a's as b's. The frequency of the memory symbol γ_0 is the exact number of b's, and a lower-bound of the number of a's.

Lemma 1. *1. The class TAWOM is closed under intersection with finite tree automata.*
2. The emptiness problem for TAWOM is decidable in time exponential in the size of the automaton.
3. For a TAWOM $\mathcal{A}$ and a finite tree automaton $\mathcal{B}$, $L(\mathcal{A}) \subseteq L(\mathcal{B})$ is decidable in time exponential in the size of $\mathcal{A}$ and doubly exponential in the size of $\mathcal{B}$.

Proof. The first result is by a product construction. The second result is from [7], and the third is immediate from the complementation of finite tree automata [8], the product construction, and part (2). □

4.3 Verification in the Tree Semantics

We now give an algorithm for the automata-theoretic verification of regular tree specifications. Our main technical construction is a TAWOM that accepts the yield language of a saga.

For a saga $\mathcal{S} = \langle \Sigma, X, s_0, T \rangle$, define the *reachable* terms of $\mathbb{T}^{\Sigma,X}$ from s_0, denoted $\mathsf{Reach}(\mathbb{T}^{\Sigma,X}, s_0)$, to be the smallest set which includes s_0 and is closed under $T(\cdot)$, and the inverses of $\{\![\cdot]\!\}$, ;, $\|$, and $\oplus$.

Theorem 2. *For every saga $\mathcal{S}$ there exists a tree automaton with one memory $\mathcal{A}$ such that $L(\mathcal{S}) = L(\mathcal{A})$.*

In our construction, the transitions of $\mathcal{A}$ encode the semantics of sagas as given in section 2. For clarity, we deal only with the transitions generating yield-trees of successfully compensated computations (including computations that need not compensate). The transitions for failed compensations, which finish with the abort outcome, are similar; the main technical difference is that the TAWOM projection functions become necessary. Because of this, and the hand-coded nature of the previous example, the automaton of example 7 does not match up exactly with the one constructed by our theorem, which is a much larger automaton.

Proof. Fix the saga $\mathcal{S} = (\Sigma, X, s_0, T)$. We define the tree automaton with one memory $\mathcal{A} = (\mathcal{F}, \Gamma, Q, Q_f, \Delta)$ in what follows. The states of $\mathcal{A}$ are the reachable terms of $\mathcal{S}$ combined with outcomes:

$$Q = \left\{ \langle t, \square \rangle, \langle t, \square \rangle^c \;\middle|\; \begin{array}{c} t \in \mathsf{Reach}(\mathbb{T}^{\Sigma,X}, s_0) \\ \square \in \{\boxdot, \boxtimes, \boxast\} \end{array} \right\},$$

while the final states are $Q_f = \{s_0\} \times \{\boxdot, \boxtimes, \boxast\}$, the input alphabet is $\mathcal{F} = \Sigma \cup \{\sigma_;, \sigma_\|\}$, and the memory alphabet is $\Gamma = \{\gamma_t \mid t \in \mathsf{Reach}(\mathbb{T}^{\Sigma,X}, s_0)\}$, where the arity of γ_t is the arity of the top level operator in t (e.g., $\gamma_{t_1;t_2} \in \Gamma_2$). The states of $\mathcal{A}$ encode the outcomes for executions of particular terms, and the superscript c of a state $\langle t, \square \rangle^c$ is used for the outcome of a compensation for the term t. The trees $\mathsf{Trees}(\mathcal{F})$ encode execution observations of $\mathcal{S}$, which can be decoded by in-order traversal, and the trees $\mathsf{Trees}(\Gamma)$ encode compensation stacks.

In what follows we present a definition schema for the transition relation, and define Δ to be the smallest relation satisfying our schema. By convention we denote an arbitrary equivalence relation with e, a state of $\mathcal{A}$ with q, and a function from Φ_Γ with φ. The function flip is defined as $\lambda x_1, x_2.x_2, x_1$. Throughout our schema, we enforce the following properties:

(P_1) For every closest $\langle \{[t]\}, \square \rangle$-labeled ancestor n of a $\langle t, \boxtimes \rangle$-state in an accepting run t, $t(n \cdot 1)$ is a compensation tree of $t(n \cdot 0)$. This property ensures that when a subtransaction $\{[t]\}$ fails with the observation tree $t(n \cdot 0)$, the proper compensating actions are observed in $t(n \cdot 1)$.

(P_2) A term which does not complete any action does not appear in an accepting run. This property is a technical convenience; without this, the automaton we defined would necessarily accept trees with 0-labeled leaves.

The definition schema is as follows.

Atomic actions. The atomic actions generate the leaves of memory and observation trees. For reachable atomic terms $a \div b$,

$$a \xrightarrow[\gamma_{a \div b}]{\top} \langle a \div b, \boxdot \rangle \quad \text{and} \quad b \xrightarrow[\gamma_{a \div b}]{\top} \langle a \div b, \boxdot \rangle^c$$

handle the execution of a, and allow for the compensation b to execute. This takes care of the rule (ATOM-S) from table 1, while the rule (ATOM-F) is taken care of by (P_1).

Sequential composition. The memory trees at $\sigma_;$-labeled nodes must be reversed when constructing the compensation's memory tree; thus we apply the flip function. For reachable sequential terms $t_1; t_2$ and outcomes $\square \in \{\boxdot, \boxtimes\}$,

$$\sigma_;(\langle t_1, \boxdot\rangle, \langle t_2, \square\rangle) \xrightarrow[\gamma_{t_1;t_2}]{\top} \langle t_1; t_2, \square\rangle$$
$$\text{and} \quad \sigma_;(\langle t_1, \boxdot\rangle^c, \langle t_2, \square\rangle^c) \xrightarrow[\gamma_{t_1;t_2}\cdot\mathsf{flip}]{\top} \langle t_1; t_2, \square\rangle^c$$

partially handle rule (SEQ-S), where t_1 completes successfully but t_2 may fail after performing at least one action, whereas the transitions generated by

$$\text{if } q \xrightarrow[\varphi]{e} \langle t_1, \square\rangle \text{ then } q \xrightarrow[\varphi]{e} \langle t_1; t_2, \boxtimes\rangle \quad \text{and} \quad \text{if } q \xrightarrow[\varphi]{e} \langle t_1, \boxdot\rangle^c \text{ then } q \xrightarrow[\varphi]{e} \langle t_1; t_2, \boxdot\rangle^c$$

handle the cases where t_1 completes but t_2 fails before completing any action, or t_1 fails after completing at least one action, as in rule (SEQ-FA). Note that the case where t_1 fails before completing any action corresponds to an empty $\mathcal{F}$-subtree for $t_1; t_2$, and is taken care of by (P_2).

Parallel composition. With parallel threads, we get away without needing the $\overline{\boxtimes}$ outcome by the invoking property (P_2). For reachable parallel terms $t_1 \| t_2$ and outcomes $\square_1, \square_2 \in \{\boxdot, \boxtimes\}$,

$$\sigma_{\|}(\langle t_1, \square_1\rangle, \langle t_2, \square_2\rangle) \xrightarrow[\gamma_{t_1\|t_2}]{\top} \langle t_1 \| t_2, \square_1 \wedge \square_2\rangle$$
$$\text{and} \quad \sigma_{\|}(\langle t_1, \boxdot\rangle^c, \langle t_2, \boxdot\rangle^c) \xrightarrow[\gamma_{t_1\|t_2}]{\top} \langle t_1 \| t_2, \boxdot\rangle^c$$

handle rule (PAR-S) where t_1 and t_2 may complete successfully, and partially rule (PAR-F) where t_1 or t_2 fail after completing at least one action, whereas the transitions generated by

$$\text{if } q \xrightarrow[\varphi]{e} \langle t_i, \square_i\rangle \text{ then } q \xrightarrow[\varphi]{e} \langle t_1 \| t_2, \boxtimes\rangle \quad \text{and} \quad \text{if } q \xrightarrow[\varphi]{e} \langle t_i, \boxdot\rangle^c \text{ then } q \xrightarrow[\varphi]{e} \langle t_1 \| t_2, \boxdot\rangle^c,$$

for $i \in \{1, 2\}$, handle the other cases of (PAR-F) where one parallel branch fails before completing any action. Again the case where both branches fail before completing any action is taken care of by (P_2).

Nondeterministic choice. The rule (NONDET) of table 1 is taken care of by closing our transitions over nondeterministic terms. For the reachable terms $t_1 \oplus t_2$, $i \in \{1, 2\}$, and outcomes $\square \in \{\boxdot, \boxtimes\}$, the transitions generated by

$$\text{if } q \xrightarrow[\varphi]{e} \langle t_i, \square\rangle \text{ then } q \xrightarrow[\varphi]{e} \langle t_1 \oplus t_2, \square\rangle$$
$$\text{and} \quad \text{if } q \xrightarrow[\varphi]{e} \langle t_i, \boxdot\rangle^c \text{ then } q \xrightarrow[\varphi]{e} \langle t_1 \oplus t_2, \boxdot\rangle^c$$

coincide exactly with (NONDET).

Subtransactions. For the reachable subtransaction terms $\{[t]\}$, the transitions generated by

$$\text{if } q \xrightarrow[\varphi]{e} \langle t, \boxdot \rangle \text{ then } q \xrightarrow[\varphi]{e} \langle \{[t]\}, \boxdot \rangle \quad \text{and} \quad \text{if } q \xrightarrow[\varphi]{e} \langle t, \boxdot \rangle^c \text{ then } q \xrightarrow[\varphi]{e} \langle \{[t]\}, \boxdot \rangle^c$$

take care of successful completion, as in rules (SUB-S) and (SAGA), while compensated completions of (SUB-F) and (SAGA) are handled by

$$\sigma_;(\langle t, \boxtimes \rangle, \langle t, \boxdot \rangle^c) \xrightarrow[\gamma_{\{[t]\}}]{1=2} \langle \{[t]\}, \boxdot \rangle \quad \text{and} \quad 0 \xrightarrow[\gamma_{\{[t]\}}]{\top} \langle \{[t]\}, \boxdot \rangle^c.$$

The first group of transitions here helps ensure (P_1) by enforcing identical memory-trees between a partially-completed subtransaction and its compensation. The second group is used to match a completed (but locally failed, and thus already compensated for) subtransaction with a null compensation. Note that while technically we do not want to consider leaves labeled with 0, it is possible to replace the previous set of transitions with a more complicated set which introduces the memory symbol $\gamma_{\{[t]\}}$ arbitrarily at any place in the tree.

For the reachable variables $x \in X$ and outcomes $\square \in \{\boxdot, \boxtimes\}$, the transitions generated by

$$\text{if } q \xrightarrow[\varphi]{e} \langle T(x), \square \rangle \text{ then } q \xrightarrow[\varphi]{e} \langle x, \square \rangle \quad \text{and} \quad \text{if } q \xrightarrow[\varphi]{e} \langle T(x), \boxdot \rangle^c \text{ then } q \xrightarrow[\varphi]{e} \langle x, \boxdot \rangle^c$$

allow named subtransactions, as in (VAR).

It is not difficult to check that properties (P_1) and (P_2) are preserved in our schema, and that the language of $\mathcal{A}$ is the yield language of $\mathcal{S}$. □

From this construction, and Lemma 1, we get the main result.

Corollary 1. [**Saga Verification**] *For every saga $\mathcal{S}$ and regular tree language specification (given as a nondeterministic finite tree automaton B), the verification problem $L(\mathcal{S}) \subseteq L(B)$ can be decided in time exponential in the size of $\mathcal{S}$ and doubly exponential in the size of B.*

This provides an exponential time algorithm in the size of the structure. On the other side, the problem is PSPACE-hard in both the structure and the specification, by reduction from term reachability of process algebras [16] and universality of word automata respectively.

5 Conclusions and Future Work

We have presented a first step towards automatic verification of business processes with compensations. While this paper provides a complexity-theoretic upper bound on the complexity of model checking, engineering effort is needed before we can obtain a practical tool for business process verification. Among other things, this means that our algorithms must be extended to model dataflow as well as deal with programming language features that are absent in the abstract formulation.

References

1. BPEL Specification v 1.1. `http://www.ibm.com/developerworks/library/ws-bpel`.
2. Business Process Modeling Language BPML. `http://www.bpmi.org/`.
3. R. Bruni, M.J. Butler, C. Ferreira, C.A.R. Hoare, H.C. Melgratti, and U. Montanari. Comparing two approaches to compensable flow composition. In *CONCUR 05*, LNCS 3653, pages 383–397. Springer, 2005.
4. R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL 05*, pages 209–220, 2005. ACM.
5. M. Butler and C. Ferreira. An operational semantics for StAC, a language for modeling business transactions. In *Proc. Co-ordination 04*, LNCS 2429, pages 87–104. Springer, 2004.
6. M. Butler, C.A.R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *Proc. 25 years of CSP*, LNCS 3525, pages 133–150. Springer, 2005.
7. H. Comon, V. Cortier, and J. Mitchell. Tree automata with one memory, set constraints, and ping-pong protocols. In *ICALP 01*, pages 682–693, 2001. Springer.
8. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997.
9. A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web services. In *PODS 04*, pages 71–82. ACM, 2004.
10. X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and verification of reactive electronic services. In *CIAA 03*, LNCS 2759, pages 188–200. Springer, 2003.
11. X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *WWW 04*, pages 621–630. ACM, 2004.
12. H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD 87*, pages 249–259. ACM, 1987.
13. J. Gray and A. Reuter. *Transaction processing: Concepts and techniques*. Morgan Kaufmann, 1993.
14. R. Hull, M. Benedikt, V. Christophides, and J. Su. E-services: a look behind the curtain. In *PODS 03*, pages 1–14. ACM, 2003.
15. A. Kučera and R. Mayr. Simulation preorder over simple process algebras. *Info. and Comp.*, 173:184–198, 2002.
16. R. Mayr. Decidability of model checking with the temporal logic EF. *TCS*, 256, 31–62, 2001.
17. Web Services Conversation Language WSCL 1.0. `http://www.w3.org/TR/wscl10/`.
18. WSFL Specification v 1.0. `http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`.