First-Order Dynamic Logic for Compensable Processes

Roberto Bruni, Carla Ferreira, Anne Kersten Kauer

First-Order Dynamic Logic for Compensable Processes*

Roberto Bruni[1], Carla Ferreira[2], and Anne Kersten Kauer[3]

[1] Department of Computer Science, University of Pisa, Italy
[2] CITI / Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal,
[3] IMT Institute for Advanced Studies, Lucca, Italy

Abstract. Compensable programs offer a convenient paradigm to deal with long-running transactions, because they offer a structured and modular approach to the composition of distributed transactional activities, like services. The basic idea is that each activity has its own compensation and that the compensable program fixes the order of execution of such activities. The main problem is how to guarantee that if one or even many faults occur then the compensations are properly executed so to reach a consistent configuration of the system. We propose a formal model for such problems based on a concurrent extension of dynamic logic that allows us to distill the hypothesis under which the correctness of compensable programs can be ensured. The main result establishes that if basic activities have a correct compensation we can show the correctness of any compound compensable program. Moreover, we can use dynamic logic to reason about behavioural and transactional properties of programs.

1 Introduction

Recent years have witnessed a massive distribution of data and computation, especially in the case of emerging paradigms like service-oriented computing and cloud computing. This has led to a breakthrough in the design of modern distributed applications, which need to integrate heterogeneous and loosely coupled components or services. When passing from coarse grain design to fine grain implementation, it is often the case that certain groups of activities must be performed in a transactional (all-or-nothing) fashion. However, many such transactions are long-lasting (weeks or even months), which prevents classical lock mechanisms from ACID transactions to be applicable in this setting. Instead, *compensable long-running transactions* seem to offer a more convenient approach, that deals well with distribution and heterogeneity: each activity is assigned a compensation; if the activity succeeds, then the compensation is installed; if the transaction ends successfully, then the installed compensations are discarded; if a fault occurs within the run of the transaction, then the installed compensations are executed (in the reverse order of installation) to compensate the fault and restore the system to a consistent state. While traditional perfect roll-back mechanisms guarantee that the initial state is exactly restored after a fault occurs, in compensable long-running transactions this is not realistic: if a message has been sent it cannot just be held back and

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

maybe another message has to be sent for compensation. For example, late canceling of a booking may require some fees to be paid.

Compensable workflows enjoy an intuitive and expressive graphical representation that has become widely popular in areas such as business process modeling and service orchestration. However, analysis and verification techniques require more work to be done in the area of formal foundation and modeling of programs-with-compensations.

Starting from StAC [7] a number of other formalisms emerged, especially in the area of process calculi, and some of them have been applied as semantic frameworks of widely adopted standard technologies [21, 18, 13]. Such formalisms often employ basic activities abstract in nature [6, 8, 2] (i.e. taken over an alphabet of labels) or are based on message passing approaches for interaction [26, 19, 5, 12]. Other approaches focus only on a basic notion of reversibility [11, 17]. Moreover, although some recent proposals addressed the problem of compensation correctness [25, 9], they miss a well-established logic counterpart, to be used by analysts, designers and programmers to prove some basic consistency properties of faulty systems after a compensation.

In this paper, we improve over existing literature on compensable workflows by proposing a rigorously formalised concurrent programming language with compensations and developing a logical framework based on dynamic logic to investigate program properties. The choice of extending dynamic logic is not incidental: we have been inspired by the interesting literature using deontic logic for error handling [4, 10]. The main novelties with respect to previous approaches is that we study concurrent programs based on compensation pairs. At the semantic level, instead of interpreting programs over pairs of states (initial and final), we take traces and make explicit the presence of possible faults. A more detailed discussion of related work is given in Sections 2.1 and 2.2. The main result establishes some sufficient conditions under which a compensable program is guaranteed to always restore a correct state after a fault.

Structure of the paper. In Section 2 we overview some background on logical formalism for error handling, and more specifically about first-order dynamic logic that we shall extend twice: in Section 3 to deal with concurrent programs and in Section 4 to deal with compensable (concurrent) programs. Our framework is detailed over a toy running example. Some concluding remarks are discussed in Section 5 together with related work and future directions of research.

2 First Order Dynamic Logic

In this section we recap the basic concepts of first order dynamic logic [14]. It was introduced to reason directly about programs, using classical first order predicate logic and modal logics combined with the algebra of regular events. In the second part we shall briefly overview related work on the two main concepts for our extension of dynamic logic, namely deontic formalisms for error handling and concurrency.

Let $\Sigma = \{f, g, \ldots, p, q \ldots\}$ be a finite first-order vocabulary where f, g range over Σ-function symbols and p, q over Σ-predicates. Each element of Σ has a fixed arity, and we denote by Σ_n the subset of symbols with arity $n > 0$. Moreover let $V = \{x_0, x_1, \ldots\}$ be a countable set of variables. Let $Trm(V) = \{t_1, \ldots, t_n, \ldots\}$ be the set of terms over the signature Σ with variables in V.

Definition 1 (Activities). *Let* $x_1, \cdots, x_n \in V$ *and* $t_1, \cdots, t_n \in Trm(V)$*. A basic activity* $a \in Act(V)$ *is a multiple assignment* $x_1, \cdots, x_n := t_1, \cdots, t_n$*.*

As special cases, we write a single assignment as $x := t \in Act(V)$ (with $x \in V$ and $t \in Trm(V)$) and the empty assignment for the inaction $skip$.

Example 1. We take an e-Store as our first example (see Fig. 1 for the complete presentation). Activity $acceptOrder \in Act(V)$ is defined as a multiple assignment to variables $stock$ and $card$ such that $acceptOrder \triangleq stock, card := stock - 1, unknown$. This activity decreases the items in stock by one (the item being sold is no longer available), and resets the current state of the credit card to unknown.

Basic activities can be combined in different ways. While it is possible to consider while programs (with sequential composition, conditional statements and while loops), we rely on the more common approach based on the so-called regular programs.

Definition 2 (Programs). *A program* α *is any term generated by the grammar:*

$$\alpha, \beta \ ::= \ a \ \mid \ \alpha\,;\beta \ \mid \ \alpha \,\Box\, \beta \ \mid \ \alpha^*$$

A program is either: a basic activity $a \in Act(V)$; the sequential composition $\alpha\,;\beta$; the nondeterministic choice $\alpha \,\Box\, \beta$; or the iteration α^* for programs α and β.

To define the semantics of programs we introduce a computational domain.

Definition 3 (Computational Domain). *Let* Σ *be a first-order vocabulary. A first-order structure* $\mathcal{D} = (D, I)$ *is called the domain of computation such that:* D *is a non-empty set, called the carrier, and* I *is a mapping assigning:*

- *to every n-ary function symbol* $f \in \Sigma$ *a function* $f^I : D^n \to D$*;*
- *to every n-ary predicate* $p \in \Sigma$ *a predicate* $p^I : D^n \to Bool$*.*

A state is a function $s : V \to D$ that assigns to each variable an element of D. The set of all states is denoted by $State(V)$. As usual, we denote by $s[x \mapsto v]$ the state s' such that $s'(x) = v$ and $s'(y) = s(y)$ for $y \neq x$. Now we can extend the interpretation to terms in a given state.

Definition 4 (Term Valuation). *The valuation* val *of a term* $t \in Trm(V)$ *in a state* $s \in State(V)$ *is defined by:*

$$val(s, x) \triangleq s(x) \text{ if } x \in V; \quad val(s, f(t_1, \ldots, t_n)) \triangleq f^I(val(s, t_1), \ldots, val(s, t_n)).$$

Basic activities (and thus programs) are interpreted as relations on states. For basic activities this means evaluating the assignments in the current state replacing the old values of the variables.

Definition 5 (Interpretation of Activities). *The valuation* $\rho \in 2^{State(V) \times State(V)}$ *of an activity* $a \in Act(V)$ *is defined by:*

$$\rho(a) \triangleq \{(s, s') \mid s' = s[x_1 \mapsto val(s, t_1), \cdots, x_n \mapsto val(s, t_n)]\}$$

if activity a *is defined by a multiple assignment* $x_1, \cdots, x_n := t_1, \cdots, t_n$*.*

Definition 6 (Interpretation of Programs). *We extend the interpretation ρ of basic activities to programs in the following manner:*

$$\begin{array}{ll} \rho(\alpha\,;\beta) & \triangleq \{(s,r) \mid (s,w) \in \rho(\alpha) \wedge (w,r) \in \rho(\beta)\} \\ \rho(\alpha \,\square\, \beta) & \triangleq \rho(\alpha) \cup \rho(\beta) \\ \rho(\alpha^*) & \triangleq \rho(\alpha)^* \end{array}$$

Sequential composition is defined using the composition of relations. The union is used for nondeterministic choice. The iteration is defined as the choice of executing a program zero or more times.

To reason about program correctness, first order dynamic logic relies on the following syntax for logical formulas.

Definition 7 (Formulas). *The set of formulas $Fml(V)$ is defined by the following grammar:*

$$\begin{array}{rl} \varphi, \psi ::= & p(t_1, \cdots, t_n) \mid \top \mid \bot \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \mid \\ & \forall x.\varphi \mid \exists x.\varphi \mid \langle\alpha\rangle\varphi \mid [\alpha]\varphi \end{array}$$

for $p \in \Sigma_n$ a predicate, $t_1, \cdots t_n \in Trm(V)$, $x \in V$ and $\alpha \in Prog(V)$.

The notion of satisfaction for logic formulas is straightforward for first order operators. The program possibility $\langle\alpha\rangle\varphi$ states that *it is possible that after executing program α, φ is true*. The necessity operator is dual to the possibility. It is defined as $[\alpha]\varphi \triangleq \neg\langle\alpha\rangle\neg\varphi$ stating that *it is necessary that after executing program α, φ is true*.

Definition 8 (Formula Validity). *The satisfiability of a formula φ in a state $s \in State(V)$ of a computational domain $\mathcal{D} = (D, I)$ is defined by:*

$$\begin{array}{lll} s \models p(t_1, \ldots, t_n) & \textit{iff} & p^I(val(s,t_1), \ldots, val(s,t_n)) \\ s \models \top & & \textit{for all } s \in S \\ s \models \neg\varphi & \textit{iff} & \textit{not } s \models \varphi \\ s \models \varphi \wedge \psi & \textit{iff} & s \models \varphi \textit{ and } s \models \psi \\ s \models \forall x.\varphi & \textit{iff} & s[x \mapsto d] \models \varphi \textit{ for all } d \in D \\ s \models \langle\alpha\rangle\varphi & \textit{iff} & \textit{there is a state } r \textit{ such that } (s,r) \in \rho(\alpha) \textit{ and } r \models \varphi \end{array}$$

A formula is valid in a domain $\mathcal{D}$ if it is satisfiable in all states over $\mathcal{D}$, it is valid if it is valid in all domains.

A valid formula for Ex. 1 would be $stock > 0 \rightarrow [acceptOrder]stock \geq 0$ (assuming that the computational domain are the natural numbers).

Below we show how in previous approaches dynamic logic is either extended with deontic formalisms or concurrency. Most of these approaches use propositional dynamic logic [15]. It is more abstract than first order dynamic logic. The interpretation of basic activities is an abstract relation on states, often Kripke frames are used. Thus there is no need for the valuation of terms and the computational domain. However it does not allow for quantification in formulas.

2.1 Deontic Formalisms for Error Handling

In this section we overview previous approaches that combine dynamic logic with deontic logic [27] by introducing operators for permission, obligation and prohibition. While the original deontic logic reasons on predicates, in combination with dynamic logic these operators are applied to actions.

Meyer [20] proposed the use of a violation condition V that describes an undesirable situation, so that the violation condition corresponds to validity of proposition V. The prohibition operator is defined such that $s \models F\alpha$ iff $s \models [\alpha]V$, *i.e.*, it is forbidden to do α in s iff all executions of α terminate in a violation. Obligation and permission are defined based on prohibition. The main problem with Meyer's work is that dependency between permission and prohibition raises paradoxes. Paradoxes of deontic logics are valid logical formulas that go against common sense, *e.g.*, Ross' paradox states if a letter ought *to be sent*, then a letter ought *to be sent or burnt*.

While Meyer focused on permission of states, Meyden [24] defined permission on the possible executions of an action. He extended models for dynamic logic with a relation P on states. An execution of an action α is permitted if every (internal) state transition of α is in P. This implies that if an execution of an action is permitted also each of its subactions must be permitted. This avoids, for example, Ross' paradox. Meyden's definition of violation is however not very different from Meyer's. As shown in [3] there is a correspondence between the two definitions.

In [4] Broersen *et al.* define permission of an action as a proposition over states and actions. Each model contains a valuation function that maps permission propositions over atomic actions to sets of states. Contrary to the previous approaches permission is in fact based on the action itself. For compound actions permission is defined in terms of possible traces and its subactions. Broersen extends this approach in [3] including also concurrency.

Castro and Maibaum [10] refine Broersen's approach. Their definition of violation is very similar, however actions can be combined differently. While previous approaches focused on free choice and sequence for the combination of atomic actions, the authors define the domain of actions as an atomic boolean algebra. Actions can be combined using choice, intersection (*i.e.* concurrency) and a locally restricted form of complement. This allows them to show not only that their logic is sound and complete, but moreover it is decidable and compact.

2.2 Concurrency

There are only a few approaches adding concurrency to dynamic logic. The first, concurrent dynamic logic [23], interpretes programs as a collection of reachability pairs, such that for each initial state it assigns a set of final states. In case of parallel composition the sets of final states are joined, while for choice the reachability pairs are joined. In this approach the formula $\langle\alpha\rangle\varphi$ holds in states s such that a reachability pair (s, U) for the interpretation of α exists and each state $s' \in U$ satisfies φ. In particular, the axiom $\langle\alpha \cap \beta\rangle\varphi \leftrightarrow \langle\alpha\rangle\varphi \wedge \langle\beta\rangle\varphi$ is valid, *i.e.*, actions are independent of each other.

For Broersen [3] this is an undesirable property. He considers only true concurrency, *i.e.* executing actions in parallel has a different effect than interleaving them. The interpretation uses the intersection for concurrency. Moreover he considers an open

concurrency interpretation, which is not applicable to first order dynamic logic (that follows a closed action interpretation).

In [1] the authors define a dynamic logic for CCS where the interpretation of concurrency is based on the labelled transition system of CCS. Thus concurrency is either interpreted as interleaving or the so-called handshake of CCS. This is the first article applying dynamic logic to a process calculus.

As we have seen, concurrency in dynamic logic is often interpreted as a simultaneous execution. This interpretation is not suited for the kind of systems we want to model, where concurrent programs describe activities that can be executed concurrently, or even in parallel, but do not have to happen simultaneously. A possible approach would be to only allow parallel composition of independent processes, *i.e.*, processes that do not interfere with each other. This requirement is quite strong and excludes most long running transactional systems. In the area of transactional concurrency control, extensive work has been done on the correctness criterion for the execution of parallel transactions. Proposed criteria include, linearizability [16], serializability [22], etc. These criteria are more realistic, since they allow some interference between concurrent transactions. Therefore, for our interpretation of concurrency we used a notion of serializability (less restrictive than linearizability), stating that: the only acceptable interleaved executions of activities from different transactions, are those that are equivalent to some sequential execution. Serializability is presented formally in Def. 17.

3 Concurrent Programs

We will consider an extension of first-order dynamic logic. We keep the definitions for the term algebra and variables from Section 2. Our definition of basic activities is extended to take into account a validity formula.

Definition 9 (Basic Activities). *A basic activity $a \in Act(V)$ is a multiple assignment together with a quantifier and program-free formula $E(a) \in Fml(V)$ that specifies the conditions under which activity a leads to an error state.*

Formula $E(a)$ can be seen as a precondition of activity a: if formula $E(a)$ holds on state s, executing a will cause an error. We exploit $E(a)$ to classify state transitions as successful or failed, depending on whether the precondition holds or not on a given state. We could use instead for each activity a an explicit set of error transitions or error states, but those sets (either of error transitions or states) can be obtained from formula $E(a)$. Another point worth discussing is the use of $E(a)$ as a precondition or postcondition for activity a. Using $E(a)$ as a precondition of a ensures erroneous state transitions do not occur, leaving the system in a correct state (the last correct state before the error). Whereas using $E(a)$ as a postcondition of a, the state transition has to occur to determine if $E(a)$ holds. Note that the empty assignment *skip* is always successful. Consider once more activity $acceptOrder$ from Ex. 1. A possible error condition could be $E(acceptOrder) \triangleq stock \leq 0$, that checks if there are any items in stock.

The definition of concurrent programs uses standard dynamic operators, including parallel composition and a test operator for a quantifier-free formula φ.

Definition 10 (Concurrent Programs). *The set $Prog(V)$ of programs is defined by the following grammar:*

$$\alpha, \beta \ ::= \ a \ \mid \ \varphi? \ \mid \ \alpha\,;\beta \ \mid \ \alpha \,\Box\, \beta \ \mid \ \alpha \parallel \beta \ \mid \ \alpha^*$$

Let the computational domain be as in Def. 3, as well as the valuation of terms as in Def. 4. The interpretation of basic activities (and thus programs) differs from the usual interpretation of dynamic logic. First we distinguish between activities that succeed or fail. Second we use traces instead of the relation on states. This is due to the combination of possible failing executions and nondeterminism. We will explain this further when introducing the interpretation of concurrent programs.

Definition 11 (Traces). *A trace τ is defined as a sequence of sets of triples $[\![\ell]\!]$ where $\ell \in \{a, -a, \varphi?\}$ ($a \in Act(V)$ a multiple assignment $x_1, \cdots, x_n := t_1, \cdots, t_n$, $\varphi \in Fml(V)$ a quantifier-free formula) and*

$$\begin{aligned}
[\![a]\!] &\triangleq \{s\, a\, s' \mid s' = s[x_1 \mapsto val(s, t_1), \cdots, x_n \mapsto val(s, t_n)] \wedge s \models \neg E(a)\} \\
[\![-a]\!] &\triangleq \{s\, a\, s \mid s \models E(a)\} \\
[\![\varphi?]\!] &\triangleq \{s\, \varphi?\, s \mid s \models \varphi\}.
\end{aligned}$$

We will use $[\![]\!]$ for the singleton containing the empty (but defined) trace; when combined with another trace it acts like the identity. Moreover we use the notation $[\![\ell.\tau]\!] = [\![\ell]\!][\![\tau]\!]$ for traces with length ≥ 1. Note that if $[\![\ell]\!] = \emptyset$ the trace is not defined. When composing traces there is no restriction whether adjoining states have to match or not. The system is in general considered to be open, *i.e.*, even within one trace between two actions there might be something happening in parallel changing the state. When we build the closure of the system traces that do not match are discarded.

A closed trace is a trace where adjoining states match. We define a predicate $closed$ on traces such that $closed(s\,\ell\,s') = \top$ and $closed(s\,\ell\,s'.\tau) = closed(\tau) \wedge (s' = first(\tau))$. For closed traces we can define functions $first$ and $last$ that return the first and the last state of the trace.

Definition 12 (Interpretation of Basic Activities). *The valuation ρ of an activity $a \in Act(V)$ is defined by:*

$$\rho(a) \triangleq [\![a]\!] \cup [\![-a]\!]$$

With this semantic model an activity a may have "good" (committed) or "bad" (failed) traces, depending on whether the initial state satisfies $E(a)$ or not. As it is clear from the interpretation of basic activities, a precondition violation forbids the execution of an activity. Therefore failed transitions do not cause a state change.

Example 2. Take activity $acceptOrder$ from Ex. 1 and its error formula $E(acceptOrder)$. In this setting, we have that

$$\begin{aligned}
[\![acceptOrder]\!] &\triangleq \{s\ acceptOrder\ s' \mid s' = s[stock \mapsto s(stock) - 1, card \mapsto unknwn] \wedge \\
&\qquad\qquad s \models stock > 0\} \\
[\![-acceptOrder]\!] &\triangleq \{s\ acceptOrder\ s \mid s \models stock \leq 0\}
\end{aligned}$$

As we mentioned for basic activities, we use traces instead of a state relation for the interpretation of programs. To illustrate this decision consider the behaviour of a compensable program. If it is successful the complete program will be executed. If it fails the program is aborted and the installed compensations will be executed. Thus we need to distinguish between successful and failing executions. Hence we need to extend the error formula E. But extending E to programs does not suffice as programs introduce nondeterminism, *i.e.*, a program with choice may both succeed and fail in the same state. Using E for programs would however only tell us that the program might fail, not which execution actually does fail. For a trace we can state whether this execution fails or not.

Definition 13 (Error Formulas of Traces). *We lift error formulas from activities to traces τ by letting $E(\tau)$ being inductively defined as:*

$$E(\llbracket\rrbracket) \triangleq \bot \qquad E(\llbracket a\rrbracket\tau) \triangleq E(\tau) \qquad E(\llbracket\varphi?\rrbracket\tau) \triangleq E(\tau) \qquad E(\llbracket -a\rrbracket\tau) \triangleq \top$$

We exploit error formulas for defining the sequential composition $\circ$ of traces. If the first trace raises an error, the execution is aborted and thus not combined with the second trace. If the first trace succeeds sequential composition is defined as usual, *i.e.* we append the second trace to the first trace.

$$\tau_\alpha \circ \tau_\beta \triangleq \begin{cases} \tau_\alpha & \text{if } E(\tau_\alpha) \\ \tau_\alpha\ \tau_\beta & \text{if } \neg E(\tau_\alpha) \end{cases}$$

Abusing the notation we use the same symbol $\circ$ to compose sets of traces.

Next, to build the trace for parallel composition of two basic programs we would consider the interleaving of any combination of traces:

$$\begin{array}{ll} \llbracket\rrbracket \parallel \tau_2 \triangleq \{\tau_2\} & \llbracket\ell_1\rrbracket\tau_1 \parallel \llbracket\ell_2\rrbracket\tau_2 \triangleq \{\llbracket\ell_1\rrbracket\tau \mid \tau \in (\tau_1 \parallel \llbracket\ell_2\rrbracket\tau_2)\} \\ \tau_1 \parallel \llbracket\rrbracket \triangleq \{\tau_1\} & \qquad\qquad \cup \{\llbracket\ell_2\rrbracket\tau \mid \tau \in (\llbracket\ell_1\rrbracket\tau_1 \parallel \tau_2)\} \end{array}$$

Now we can define the interpretation of concurrent programs:

Definition 14 (Interpretation of Concurrent Programs). *We extend the interpretation ρ from basic activities to concurrent programs in the following manner:*

$$\begin{array}{ll} \rho(\varphi?) & \triangleq \llbracket\varphi?\rrbracket \\ \rho(\alpha\,;\beta) & \triangleq \{\tau_\alpha \circ \tau_\beta \mid \tau_\alpha \in \rho(\alpha) \wedge \tau_\beta \in \rho(\beta)\} \\ \rho(\alpha \,\square\, \beta) & \triangleq \rho(\alpha) \cup \rho(\beta) \\ \rho(\alpha \parallel \beta) & \triangleq \{\tau \mid \tau_\alpha \in \rho(\alpha) \wedge \tau_\beta \in \rho(\beta) \wedge \tau \in \tau_\alpha \parallel \tau_\beta\} \\ \rho(\alpha^*) & \triangleq \llbracket\rrbracket \cup \rho(\alpha) \circ \rho(\alpha^*) \end{array}$$

Test $\varphi?$ is interpreted as the identity trace for the states that satisfy formula φ. The interpretation of sequential programs is the sequential composition of traces. Failed transitions are preserved in the resulting set as executions of α that have reached an erroneous state and cannot evolve. Choice is interpreted as the union of trace sets of programs α and β. The interpretation of parallel composition is the set of all possible interleavings of the traces for both branches. Iteration is defined recursively (by taking the least fixpoint).

$$eStore \triangleq acceptOrder\,;((acceptCard \,\square\, rejectCard\,;throw) \parallel bookCourier)$$

$$\begin{array}{llll}
\mathsf{aO} & \triangleq stock, card := stock - 1, unkown & E(\mathsf{aO}) & \triangleq stock \leq 0 \\
\mathsf{aC} & \triangleq card := accepted & E(\mathsf{aC}) & \triangleq false \\
\mathsf{rC} & \triangleq card := rejected & E(\mathsf{rC}) & \triangleq false \\
\mathsf{bC} & \triangleq courier := booked & E(\mathsf{bC}) & \triangleq card = rejected \\
throw & \triangleq skip & E(throw) & \triangleq true
\end{array}$$

$$\begin{array}{l}
\rho(\mathsf{aO}\,;((\mathsf{aC}\,\square\,\mathsf{rC}\,;throw) \parallel \mathsf{bC})) = \\
[\![\mathsf{aO.aC.bC}]\!] \cup [\![\mathsf{aO.bC.aC}]\!] \cup \\
[\![\mathsf{aO.aC.} - \mathsf{bC}]\!] \cup [\![\mathsf{aO.} - \mathsf{bC.aC}]\!] \cup \\
[\![\mathsf{aO.rC.} - throw.\mathsf{bC}]\!] \cup [\![\mathsf{aO.bC.rC.} - throw]\!] \cup [\![\mathsf{aO.rC.bC.} - throw]\!] \cup \\
[\![\mathsf{aO.rC.} - throw.\, - \mathsf{bC}]\!] \cup [\![\mathsf{aO.} - \mathsf{bC.rC.} - throw]\!] \cup [\![\mathsf{aO.rC.} - \mathsf{bC.} - throw]\!] \cup [\![-\mathsf{aO}]\!]
\end{array}$$

Fig. 1. $eStore$ example.

To build the closure of the system, *i.e.*, a program α, we define the set of all closed traces for α such that $closure(\alpha) \triangleq \{\tau \mid \tau \in \rho(\alpha) \wedge closed(\tau)\}$.

Next we show in an example the application of Definition 14. Take program $eStore$ defined as in Fig. 1. Note that we abbreviate activities using initials, *e.g.*, we write aO for $acceptOrder$. This program describes a simple online shop and it starts with an activity that removes from the stock the ordered items. Since for most orders the credit cards are not rejected, and to decrease the delivery time, the client's card processing and courier booking can be done in parallel. In this example, the activities running in parallel may interfere with each other as $bookCourier$ will fail once the card is rejected. As the order of the execution for the parallel composition is not fixed after rejecting the credit card we issue a throw (defined as the empty assignment that always fails). In the interpretation of program $eStore$ we can first distinguish the traces where $acceptOrder$ succeeds and where it fails. In the latter case no other action is executable. In the successful case the parallel composition is executed where both branches may succeed or fail and we include any possible interleaving. Note that the traces $[\![-\mathsf{aC}]\!]$, $[\![-\mathsf{rC}]\!]$ and $[\![throw]\!]$ are not defined, as their condition is not satisfied by any possible state. Building the closure for these traces we can rule out some possibilities, namely $[\![\mathsf{aO.aC.} - \mathsf{bC}]\!]$, $[\![\mathsf{aO.} - \mathsf{bC.aC}]\!]$, $[\![\mathsf{aO.rC.} - throw.\mathsf{bC}]\!]$ and $[\![\mathsf{aO.rC.bC.} - throw]\!]$ and $[\![\mathsf{aO.} - \mathsf{bC.rC.} - throw]\!]$ would be excluded. A possible closed trace would be

$$\begin{array}{l}
closed(\; s\,\mathsf{aO}\,s'\,.\,s'\mathsf{aC}\,r\,.\,r\mathsf{bC}\,s'' \\
\qquad \mid s' = s[stock \mapsto s(stock) - 1, card \mapsto unkown] \wedge r = s'[card \mapsto accepted] \\
\qquad \wedge s'' = r[courier \mapsto booked] \wedge s \models stock > 0 \wedge r \models \neg card = rejected)
\end{array}$$

For formulas we include two modal operators related to program success where success of a program is interpreted as not reaching an erroneous state. The modal operator success $\mathsf{S}(\alpha)$ states that *every way of executing α is successful*, so program α must never reach an erroneous state. The modal operator weak success $\mathsf{S_W}(\alpha)$ states that *some way of executing α is successful*. The failure modal operator $\mathsf{F}(\alpha)$ is a derived operator, $\mathsf{F}(\alpha) = \neg\mathsf{S_W}(\alpha)$, and states that *every way of executing α fails*. Notice that both weak success and program possibility ensure program termination, while success and program necessity do not.

Definition 15 (Formulas). *The set of formulas $Fml(V)$ is defined by the grammar:*

$$\varphi, \psi ::= p(t_1, \cdots, t_n) \mid \top \mid \bot \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \mid \forall x.\varphi \mid \exists x.\varphi \mid \langle\alpha\rangle\varphi \mid [\alpha]\varphi \mid \mathsf{S}(\alpha) \mid \mathsf{S_W}(\alpha) \mid \mathsf{F}(\alpha)$$

for $p \in \Sigma_n$ a predicate, $t_1, \cdots t_n \in Trm(V)$, $x \in V$ and $\alpha \in Prog(V)$.

Definition 16 (Formula Validity). *The validity of a formula φ in a state $s \in State(V)$ of a computational domain $\mathcal{D} = (D, I)$ is defined by:*

$s \models p(t_1, \ldots, t_n)$	*iff*	$p^I(val(s, t_1), \ldots, val(s, t_n))$
$s \models \top$		*for all* $s \in S$
$s \models \neg\varphi$	*iff*	*not* $s \models \varphi$
$s \models \varphi \wedge \psi$	*iff*	$s \models \varphi$ *and* $s \models \psi$
$s \models \forall x.\varphi$	*iff*	$s[x \mapsto d] \models \varphi$ *for all* $d \in D$
$s \models \langle\alpha\rangle\varphi$	*iff*	$\exists\, \tau \in closure(\alpha)$ *such that* $first(\tau) = s \wedge last(\tau) \models \varphi$
$s \models \mathsf{S}(\alpha)$	*iff*	*for all* $\tau \in closure(\alpha)$, $first(\tau) = s$ *implies* $\neg E(\tau)$
$s \models \mathsf{S_W}(\alpha)$	*iff*	$\exists\, \tau \in closure(\alpha)$ *such that* $first(\tau) = s$ *and* $\neg E(\tau)$
$s \models \mathsf{F}(\alpha)$	*iff*	$s \models \neg\mathsf{S_W}(\alpha)$

The new modal operators for success and failure are defined according to the description given above. Considering Fig. 1 a possible formula would be $\mathsf{F}(acceptOrder)$, that is only satisfiable in some states. However $stock \leq 0 \rightarrow \mathsf{F}(acceptOrder)$ is a valid formula for any state.

As it was discussed in Section 2, we want an interpretation of concurrency where concurrent activities do not have to happen simultaneously. For example, packing the items in a client's order and booking a courier to deliver that same order are independent activities that can be run in parallel. If, contrary to the example just mentioned, parallel activities are not independent then concurrency becomes more complex: as activities may interfere with each other, this may lead to unexpected results. In this work parallel composition is not restricted to independent programs (such that the set of variables updated and read by those programs is disjoint), however without any restriction the logic is too liberal to state any properties. Instead we use a notion of serializability (Def. 17). We will use the abbreviation $\tau \bowtie \tau'$ for traces τ and τ' denoting

$$\tau \bowtie \tau' \triangleq closed(\tau) \wedge closed(\tau') \wedge first(\tau) = first(\tau') \wedge last(\tau) = last(\tau')$$

Definition 17 (Serializable Concurrent Programs). *Processes α and β are serializable if for every trace $\tau \in closure(\alpha \parallel \beta)$ there exist $\nu \in \rho(\alpha)$ and $\mu \in \rho(\beta)$ such that either $\tau \bowtie (\nu\mu)$ or $\tau \bowtie (\mu\nu)$.*

Note that serializability is defined as an implication, as the equivalence cannot be ensured. The fact that activities may be executed independently does not ensure, in the presence of interference, that those activities may be executed concurrently.

We show some useful logical equivalences involving the novel modal operators. We are particularly considering their interplay with parallel composition.

Proposition 1. *Let α, β be two serializable programs. The following are valid formulas in the presented dynamic logic:*

$$\langle\alpha\rangle\langle\beta\rangle\varphi \vee \langle\beta\rangle\langle\alpha\rangle\varphi \leftrightarrow \langle\alpha \parallel \beta\rangle\varphi \qquad \mathsf{S}(\alpha\,;\beta \,\square\, \beta\,;\alpha) \leftrightarrow \mathsf{S}(\alpha \parallel \beta)$$
$$\mathsf{S_W}(\alpha\,;\beta) \vee \mathsf{S_W}(\beta\,;\alpha) \leftrightarrow \mathsf{S_W}(\alpha \parallel \beta) \qquad \mathsf{F}(\alpha\,;\beta) \wedge \mathsf{F}(\beta\,;\alpha) \leftrightarrow \mathsf{F}(\alpha \parallel \beta)$$

4 Compensable Programs

This section defines compensable programs, where the basic building block is the compensation pair. A compensation pair $a \div \overline{a}$ is composed by two activities, such that if the execution of the first activity is successful, compensation $\overline{a}$ is stored. Otherwise, if activity a fails, no compensation is stored. Compensation pairs can be composed using similar operators as for basic programs. The transaction operator $\{\![\cdot]\!\}$ converts a compensable program into a basic program by discarding stored compensations for successful executions and running compensations for failed executions.

Definition 18 (Compensable Programs). *The set $Cmp(V)$ of compensable programs is defined by the following grammar (for $a, \overline{a} \in Act(V)$):*

$$\begin{aligned} \alpha &::= \ldots \mid \{\![\delta]\!\} \\ \delta, \gamma &::= a \div \overline{a} \mid \delta \,;\, \gamma \mid \delta \,\Box\, \gamma \mid \delta \parallel \gamma \mid \delta^* \end{aligned}$$

In order to ensure the overall correctness of a compensable program, the backward program of a compensation pair $a \div \overline{a}$ must satisfy a condition: $\overline{a}$ must successfully revert all forward actions of activity a. In the following we do not require that $\overline{a}$ exactly undoes all assignments of a and thus revert the state to the exact initial state of a. Instead, we require that it performs some compensation actions that lead to a "sufficiently" similar state. The way to determine if two states are similar depends on the system under modeling, so we do not enforce any rigid definition. Still, we propose a concrete notion that may characterize a widely applicable criterion of correctness (but other proposals may be valid as well).

The notion we give is parametric w.r.t. a set of variables whose content we wish to monitor.

Definition 19. *Let X be a set of integer variables. We call $s \backslash_X s'$ the distance over X between two states, i.e., the set of changes occurred over the variables in X, when moving from s to s'. Formally, we define $(s \backslash_X s')(x) = s'(x) - s(x)$ for any $x \in X$ (and let $(s \backslash_X s')(x) = 0$ otherwise).*

For an empty trace the distance is null, the same holds for $s \backslash_X s$.

Definition 20 (Correct Compensation Pair). *Let X be a set of integer variables. Activity $\overline{a} \in Act(V)$ is a correct compensation over X of a iff for all traces $s\, a\, s' \in \rho(a)$ with $s \models \neg E(a)$, then for all $t'\, \overline{a}\, t \in \rho(\overline{a})$ we have $t' \models \neg E(\overline{a})$ and $s \backslash_X s' = t \backslash_X t'$.*

Example 3. For most hotels the cancellation of a booking requires the payment of a cancellation fee. This can be modeled by a compensation pair $bookHotel \div cancelHotel$, where forward activity $bookHotel$ books a room and sets the amount to be paid, while the compensation $cancelHotel$ cancels the reservation and charges the cancellation fee.

$$\begin{aligned} bookHotel &\triangleq rooms, price, fee := rooms - 1, 140\$, 20\$ \\ cancelHotel &\triangleq rooms, price, fee := rooms + 1, fee, 0\$ \end{aligned}$$

In this example $cancelHotel$ does not completely revert all of $bookHotel$ actions, and in fact it is likely that room cancelation imposes some fees. However, if we are only

interested in the consistency of the overall number of available rooms, we can take $X = \{rooms\}$ and consider $s \backslash_X s'$ as a measure of distance. The idea is that a correct compensation should restore as available the rooms that were booked but later canceled.

In Definition 21 each compensable program is interpreted as a set of pairs of traces, where the first element is a trace of the forward program, while the second element is a trace that compensates the actions of its forward program. Compensation pairs are interpreted as the union of two sets. The first set represents the successful traces of forward activity a, paired with the traces of compensation activity $\overline{a}$. The second set represents the failed traces, paired with the empty trace as its compensation. Sequential composition of compensable programs $\delta\,;\gamma$ is interpreted by two sets, one where the successful termination of δ allows the execution of γ, the other where δ fails and therefore γ cannot be executed. As for the compensations of a sequential program, their traces are composed in the reverse order of their forward programs.

The interpretation of a transaction includes two sets: the first set discards compensable traces for all successful traces; while the second set deals with failed traces by composing the correspondent compensation trace with each failed trace. Notice that for this trace composition to be defined, it is necessary to clear any faulty activities in the failed forward trace. Therefore, in defining the interpretation of a transaction $\{\!|\delta|\!\}$, we exploit a function cl that clears failing activities from a run:

$$cl([\![]\!]) \triangleq [\![]\!] \qquad cl([\![\ell]\!]\tau) \triangleq \begin{cases} [\![E(a)?]\!]\ cl(\tau) & \text{if } \ell = -a \\ [\![\ell]\!]\ cl(\tau) & \text{otherwise} \end{cases}$$

where $E(a)?$ is the test for the error formula of activity a. It is always defined in s as activity a only aborts if $s \models E(a)$, thus it is in general like a skip. In fact, if τ is faulty but successfully compensated by $\overline{\tau}$, then we want to exhibit an overall non faulty run, so that we cannot just take $\tau \circ \overline{\tau}$ (if $E(\tau)$ then obviously $E(\tau \circ \overline{\tau})$). The following lemma states that cl does not alter the first nor the last state of a trace:

Lemma 1. *For any τ: $\neg E(cl(\tau))$, $first(\tau) = first(cl(\tau))$ and $last(\tau) = last(cl(\tau))$.*

Definition 21 (Interpretation of Compensable Programs). *Now we can define the interpretation ρ_c of compensable programs in the following manner:*

$$\begin{aligned}
\rho_c(a \div \overline{a}) &\triangleq \{(\tau, \overline{\tau}) \mid \tau \in \rho(a) \wedge \overline{\tau} \in \rho(\overline{a}) \wedge \neg E(\tau)\} \cup \\
&\quad \{(\tau, [\![]\!]) \mid \tau \in \rho(a) \wedge E(\tau)\} \\
\rho_c(\delta \,\square\, \gamma) &\triangleq \rho_c(\delta) \cup \rho_c(\gamma) \\
\rho_c(\delta\,;\gamma) &\triangleq \{(\tau \circ \nu, \overline{\nu} \circ \overline{\tau}) \mid (\tau, \overline{\tau}) \in \rho_c(\delta) \wedge (\nu, \overline{\nu}) \in \rho_c(\gamma) \wedge \neg E(\tau)\} \cup \\
&\quad \{(\tau, \overline{\tau}) \mid (\tau, \overline{\tau}) \in \rho_c(\delta) \wedge E(\tau)\} \\
\rho_c(\delta \parallel \gamma) &\triangleq \{(\tau, \overline{\tau}) \mid (\nu, \overline{\nu}) \in \rho_c(\delta) \wedge (\mu, \overline{\mu}) \in \rho_c(\gamma)\ \wedge \tau \in \nu \parallel \mu\ \wedge\ \overline{\tau} \in \overline{\nu} \parallel \overline{\mu}\} \\
\rho_c(\delta^*) &\triangleq \{([\![]\!], [\![]\!])\}\ \cup\ \rho_c(\delta\,;\delta^*) \\
\rho(\{\!|\delta|\!\}) &\triangleq \{\tau \mid (\tau, \overline{\tau}) \in \rho_c(\delta) \wedge \neg E(\tau)\} \cup \\
&\quad \{cl(\tau) \circ \overline{\tau} \mid (\tau, \overline{\tau}) \in \rho_c(\delta) \wedge E(\tau)\}
\end{aligned}$$

In Fig. 2 we present a compensable program that specifies a hotel booking system. In this example the activity $bookHotel$ updates several variables: it decreases the rooms

$$HotelBooking \triangleq bookHotel \div cancelHotel\,; \\ (acceptBooking \div skip \;\Box\; cancelBooking \div skip\,; throw \div skip))$$

$$\mathsf{bH} \triangleq rooms, status, price, fee := rooms - 1, booked, 140\$, 20\$ \qquad E(\mathsf{bH}) \triangleq rooms \leq 0$$
$$\mathsf{cH} \triangleq rooms, price, fee := rooms + 1, fee, 0\$ \qquad E(\mathsf{cH}) \triangleq false$$
$$\mathsf{aB} \triangleq status := confirmed \qquad E(\mathsf{aB}) \triangleq false$$
$$\mathsf{cB} \triangleq status := cancelled \qquad E(\mathsf{cB}) \triangleq false$$

$$\rho_c(\mathsf{bH} \div \mathsf{cH}\,; (\mathsf{aB} \div skip \;\Box\; \mathsf{cB} \div skip\,; throw \div skip)) \\ = (\llbracket \mathsf{bH}.\mathsf{aB} \rrbracket, \llbracket skip.\mathsf{cH} \rrbracket) \cup (\llbracket \mathsf{bH}.\mathsf{cB}.- throw \rrbracket, \llbracket skip.\mathsf{cH} \rrbracket) \cup (\llbracket -\mathsf{bH} \rrbracket, [\![]\!])$$

Fig. 2. *HotelBooking* example.

$$HotelTransactions \triangleq \{\!\{HotelBooking_1\}\!\} \parallel \{\!\{HotelBooking_2\}\!\}$$

$$\rho(\{\!\{HotelBooking\}\!\}) \;=\; \llbracket \mathsf{bH}.\mathsf{aB} \rrbracket \cup \llbracket \mathsf{bH}.\mathsf{cB}.E(throw)?.skip.\mathsf{cH} \rrbracket \cup \llbracket E(\mathsf{bH})? \rrbracket$$

$$\begin{aligned}
&\rho(HotelTransactions) \\
&= \llbracket \mathsf{bH}_1.\mathsf{aB}_1 \rrbracket \parallel \llbracket \mathsf{bH}_2.\mathsf{aB}_2 \rrbracket \quad \cup \quad \llbracket E(\mathsf{bH}_1)? \rrbracket \parallel \llbracket E(\mathsf{bH}_2)? \rrbracket \\
&\cup \llbracket \mathsf{bH}_1.\mathsf{cB}_1.E(throw_1)?.skip_1.\mathsf{cH}_1 \rrbracket \parallel \llbracket \mathsf{bH}_2.\mathsf{cB}_2.E(throw_2)?.skip_2.\mathsf{cH}_2 \rrbracket \\
&\cup \llbracket \mathsf{bH}_1.\mathsf{aB}_1 \rrbracket \parallel \llbracket \mathsf{bH}_2.\mathsf{cB}_2.E(throw_2)?.skip_2.\mathsf{cH}_2 \rrbracket \\
&\cup \llbracket \mathsf{bH}_1.\mathsf{cB}_1.E(throw_1)?.skip_1.\mathsf{cH}_1 \rrbracket \parallel \llbracket \mathsf{bH}_2.\mathsf{aB}_2 \rrbracket \\
&\cup \llbracket E(\mathsf{bH}_1)? \rrbracket \parallel \llbracket \mathsf{bH}_2.\mathsf{cB}_2.E(throw_2)?.skip_2.\mathsf{cH}_2 \rrbracket \\
&\cup \llbracket \mathsf{bH}_1.\mathsf{cB}_1.E(throw_1)?.skip_1.\mathsf{cH}_1 \rrbracket \parallel \llbracket E(\mathsf{bH}_2)? \rrbracket \\
&\cup \llbracket \mathsf{bH}_1.\mathsf{aB}_1 \rrbracket \parallel \llbracket E(\mathsf{bH}_2)? \rrbracket \quad \cup \quad \llbracket E(\mathsf{bH}_1)? \rrbracket \parallel \llbracket \mathsf{bH}_2.\mathsf{aB}_2 \rrbracket
\end{aligned}$$

Fig. 3. *HotelTransactions* example.

available, sets the booking status to *booked*, while the price and cancellation fee are set to predefined values. Next, there is a choice between confirming or canceling the booking. These last two activities do not have a compensation, represented by defining compensation as *skip*. After *cancelBooking* the process is aborted by executing *throw*. Regarding the interpretation of *HotelBooking*, each forward run is paired with a compensation trace that reverts all its successfully terminated activities. For example, the first pair of traces represents a successful execution, where after booking a room the client accepts that reservation. In this case the stored compensation reverts both the acceptance of the booking and the booking it self. Note that as in Example 3, the compensation activity *cancelHotel* does not revert completely its main activity: it reverts the room booking, charges the cancellation fee, and it leaves the booking status unchanged.

Fig. 3 shows the parallel composition of two *HotelBooking* transactions (the subscripts are used only to distinguish activities from different transactions). The set of all possible interleaving is quite large, even after discarding traces that are not satisfied by any possible state. An example of a trace to be discarded is $\llbracket E(\mathsf{bH}_1)?.\mathsf{bH}_2.\mathsf{aB}_2 \rrbracket$, as $E(\mathsf{bH}_1)$ is true in a state s such that $s \models rooms \leq 0$ and in that case $\llbracket \mathsf{bH}_2.\mathsf{aB}_2 \rrbracket$ could not be executed.

The program of Fig. 3 shows how the interleaved execution of processes may lead to interferences. Consider $\theta_1 = [\![\mathsf{bH}_1.\mathsf{cB}_1.E(throw_1)?.skip_1.\mathsf{cH}_1]\!]$ that describes traces where the booking succeeds and is later canceled by the client, and $\theta_2 = [\![E(\mathsf{bH}_2)?]\!]$ that describes traces where no rooms are available and therefore no activity can be executed. Take the interleaving of traces of θ_1 and θ_2 on an initial state s such that $s \models rooms = 1$. In this setting, θ_1 has to be executed first and consequently activity bH_1 books the last room available. There is no serial execution of θ_1 and θ_2 (these sets of traces cannot be sequentially composed), since after the execution of a trace of θ_1 activity bH_2 should succeed (the last room becomes available again after the execution of cB_1). However, the following interleaved execution is possible $[\![\mathsf{bH}_1.E(\mathsf{bH}_2)?.\mathsf{cB}_1.E(throw_1)?.skip_1.\mathsf{cH}_1]\!]$, because when bH_2 is executed there are no rooms available. This shows that $HotelTransactions$ is not serializable, since some interleaved traces do not correspond to a serial execution.

The aim of compensable programs is that the overall recoverability of a system can be achieved through the definition of local recovery actions. As the system evolves, those local compensation actions are dynamically composed into a program that reverts all actions performed until then. Therefore, it is uttermost important that the dynamically built compensation trace does indeed revert the current state to the initial state. Next, we define the notion of a correct compensable program, where any failed forward trace can be compensated to a state equivalent to the initial state.

Definition 22 (Correct Compensable Program). *Let X be a set of integer variables. A compensable program δ has a correct compensation over X if for all pairs of traces $(\tau, \overline{\tau}) \in \rho_c(\delta)$, if $closed(\tau)$ and $closed(\overline{\tau})$ then $first(\tau)\backslash_X last(\tau) = last(\overline{\tau})\backslash_X first(\overline{\tau})$.*

Serializability is extended from basic programs to compensable programs:

Definition 23 (Serializable compensable parallel programs). *Compensable programs δ and γ are serializable if for all $(\tau, \overline{\tau}) \in \rho_c(\delta \parallel \gamma)$ with $closed(\tau)$ and $closed(\overline{\tau})$ there exist $(\nu, \overline{\nu}) \in \rho_c(\delta)$ and $(\mu, \overline{\mu}) \in \rho_c(\gamma)$ such that the following holds:*
Either $\tau \bowtie (\nu\mu)$ or $\tau \bowtie (\mu\nu)$ and either $\overline{\tau} \bowtie (\overline{\nu}\,\overline{\mu})$ or $\overline{\tau} \bowtie (\overline{\mu}\,\overline{\nu})$.

The following theorem shows the soundness of our language, since it proves that compensation correctness is ensured by construction: the composition of correct compensable programs results in a correct compensable program.

Theorem 1. *Let X be a set of integer variables and δ a serializable compensable program where every compensation pair is correct over X, then δ is correct over X.*

As in general serializability is hard to prove we suggest the simpler definition of apartness. If the set of variables updated and consulted by two programs are disjoint, those programs can be concurrently executed since they do not interfere with each other (a similar approach was taken in [9]). Two compensable programs δ and γ are apart if they do not update or read overlapping variables, then δ and γ can be executed concurrently and their resulting traces can be merged. The final state of a concurrent execution of apart programs can be understood as a join of the resulting states of each program.

Formulas for basic programs can be easily extended to compensable programs as they can be applied to the forward program and stored compensations can be ignored.

The modal operator $\mathsf{C}(\delta)$ states that every failure of δ is compensable. A weak compensable operator $\mathsf{C_W}(\delta)$ states that some failures of δ are compensable.

We extend the notion of closed traces from basic to compensable programs as

$$closure(\delta) \triangleq \{(\tau, \overline{\tau}) \mid (\tau, \overline{\tau}) \in \rho_c(\delta) \wedge closed(\tau) \wedge closed(\overline{\tau})\}.$$

Definition 24 (Formula Validity). *We extend Definition 16 with the new modal operators for compensable programs.*

$s \models \langle\delta\rangle\varphi$	*iff*	$\exists\, (\tau, \overline{\tau}) \in closure(\delta)$ *such that* $first(\tau) = s \wedge last(\tau) \models \varphi$
$s \models \mathsf{S}(\delta)$	*iff*	*for all traces* $(\tau, \overline{\tau}) \in closure(\delta)$ *if* $first(\tau) = s$ *then* $\neg E(\tau)$
$s \models \mathsf{S_W}(\delta)$	*iff*	$\exists (\tau, \overline{\tau}) \in closure(\delta)$ *such that* $first(\tau) = s$ *and* $\neg E(\tau)$
$s \models \mathsf{F}(\delta)$	*iff*	$s \models \neg \mathsf{S_W}(\delta)$
$s \models \mathsf{C}(\delta, X)$	*iff*	*for all traces* $(\tau, \overline{\tau}) \in closure(\delta)$ *if* $E(\tau)$ *and* $first(\tau) = s$ *then* $first(\tau) \backslash_X last(\tau) = last(\overline{\tau}) \backslash_X first(\overline{\tau})$
$s \models \mathsf{C_W}(\delta, X)$	*iff*	$\exists (\tau, \overline{\tau}) \in closure(\delta)$ *such that* $E(\tau)$ *and* $first(\tau) = s$ *and* $first(\tau) \backslash_X last(\tau) = last(\overline{\tau}) \backslash_X first(\overline{\tau})$

Considering Fig. 2 a possible formula is $\langle HotelBooking\rangle status = confirmed$. Moreover we can prove $\mathsf{C}(HotelBooking, \{rooms\})$ for any state.

Proposition 2. *Let δ, γ be two compensable programs that are apart. The following are valid formulas in the presented dynamic logic:*

$$\mathsf{C}(\delta \parallel \gamma, X) \leftrightarrow \mathsf{C}(\delta, X) \wedge \mathsf{C}(\gamma, X) \qquad \mathsf{C_W}(\delta \parallel \gamma, X) \rightarrow \mathsf{C_W}(\delta, X) \vee \mathsf{C_W}(\gamma, X)$$

5 Conclusion

In this paper we have introduced a rigorous language of compensable concurrent programs together with a dynamic logic for reasoning about compensation correctness and verification of behavioral properties of compensable programs. In this sense we go one step further of the approach in [25, 9], where the formulas were mainly concerned with the temporal order of execution of actions in a message-passing calculus with dynamic installation of compensation, by allowing to express properties about the adequacy of the state restored by the compensation after a fault occurred. As detailed in Sections 2.1 and 2.2, our dynamic logic differs from previous proposal for the way in which concurrency is handled and for dealing with compensations. The works presented in [8, 9] study the soundness of a compensating calculus by formalizing a notion of compensation correctness, which we also address in Theorem 1, but do not tackle the subject of verification of behavioral properties for compensable programs. Furthermore, even though compensation correctness is ensured by construction, our logic allows the verification of strong and weak correctness (through a formula) for compensable programs that contain some compensation pairs that are not correct.

Our research programme leaves as ongoing work the development of a suitable computational model and corresponding logic for allowing a quantitative measure of correctness, so that different kinds of compensations can be distinguished (and the best can be selected) depending on their ability to restore a more satisfactory state than the others can do. Moreover, we would like to develop suitable equivalences over states that can reduce the complexity of the analysis, and facilitate the development of automatic reasoning tools.

References

1. M. Benevides and L. Schechter. A propositional dynamic logic for CCS programs. In *WoLLIC'08*, volume 5110 of *LNCS*, pages 83–97. Springer, 2008.
2. M. Bravetti and G. Zavattaro. On the expressive power of process interruption and compensation. *Math. Struct. in Comput. Sci.*, 19(3):565–599, 2009.
3. J. Broersen. *Modal Action Logics for Reasoning about Reactive Systems*. PhD thesis, Faculteit der Exacte Wetenschappen, Vrije Universiteit Amsterdam, 2003.
4. J. Broersen, R. Wieringa, and J.-J. Meyer. A fixed-point characterization of a deontic logic of regular action. *Fundamenta Informaticae*, 48(2-3):107–128, 2001.
5. R. Bruni, H. Melgratti, and U. Montanari. Nested Commits for Mobile Calculi: Extending Join. In *IFIP TCS'04*, pages 563–576. Kluwer Academics, 2004.
6. R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL'05*, pages 209–220. ACM, 2005.
7. M. Butler and C. Ferreira. A Process Compensation Language. In *IFM'00*, volume 1945 of *LNCS*, pages 61–76. Springer, 2000.
8. M. Butler, C. Hoare, and C. Ferreira. A Trace Semantics for Long-Running Transactions. In *25 Years of CSP*, volume 3525 of *LNCS*, pages 133–150. Springer, 2004.
9. L. Caires, C. Ferreira, and H. Vieira. A process calculus analysis of compensations. In *TGC'08*, volume 5474 of *LNCS*, pages 87–103. Springer, 2009.
10. P. Castro and T. Maibaum. Deontic action logic, atomic boolean algebras and fault-tolerance. *Journal of Applied Logic*, 7(4):441–466, 2009.
11. V. Danos and J. Krivine. Reversible Communicating Systems. In *CONCUR'04*, volume 3170 of *LNCS*, pages 292–307. Springer, 2004.
12. E. de Vries, V. Koutavas, and M. Hennessy. Communicating Transactions - (Extended Abstract). In *CONCUR'10*, volume 6269 of *LNCS*, pages 569–583. Springer, 2010.
13. C. Eisentraut and D. Spieler. Fault, compensation and termination in WS-BPEL 2.0 - A comparative analysis. In *WS-FM'08*, volume 5387 of *LNCS*, pages 107–126. Springer, 2009.
14. D. Harel. First order dynamic logic. In *LNCS*, volume 68. Springer, 1979.
15. D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604. MIT Press, 1984.
16. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
17. I. Lanese, C. Mezzina, and J.-B. Stefani. Reversing higher-order pi. In *CONCUR'10*, volume 6269 of *LNCS*, pages 478–493. Springer, 2010.
18. R. Lucchi and M. Mazzara. A pi-calculus based semantics for WS-BPEL. *J. Log. Algebr. Program.*, 70(1):96–118, 2007.
19. M. Mazzara and I. Lanese. Towards a Unifying Theory for Web Services Composition. In *WS-FM*, pages 257–272, 2006.
20. J.-J. Meyer. A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. *Notre Dame Journal of Formal Logic*, 29:109–136, 1988.
21. OASIS. WSBPEL. http://http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, 2007.
22. C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, Oct. 1979.
23. D. Peleg. Concurrent dynamic logic. *J. ACM*, 34:450–479, 1987.
24. R. Van Der Meyden. The dynamic logic of permission. *Journal of Logic and Computation*, 6:465–479, 1996.
25. C. Vaz and C. Ferreira. Towards compensation correctness in interactive systems. In *WS-FM'09*, volume 6194 of *LNCS*, pages 161–177. Springer, 2010.
26. C. Vaz, C. Ferreira, and A. Ravara. Dynamic Recovering of Long Running Transactions. In *TGC'08*, volume 5474 of *LNCS*, pages 201–215. Springer, 2009.
27. G. Von Wright. I. deontic logic. *Mind*, LX(237):1–15, 1951.

A Proof Theorem 1

We use $\Delta_X(\tau)$ to denote $\mathit{first}(\tau)\backslash_X \mathit{last}(\tau)$ and $\nabla_X(\tau)$ for $\mathit{last}(\tau)\backslash_X \mathit{first}(\tau)$. In the following operation $\oplus$ denotes the union of two distances of two states.

Lemma A. *Let $s, s', s'' \in \mathit{State}(V)$. The following equality holds:*

$$s\backslash_X s' \oplus s'\backslash_X s'' = s\backslash_X s''$$

Theorem 1. *Let X be a set of integer variables and δ a serializable compensable program where every compensation pair is correct over X, then δ is correct over X.*

Proof. We proceed by induction on the structure of compensable δ.

1. $\delta = a \div \overline{a}$. For any $(\tau, \overline{\tau}) \in \rho_c(a \div \overline{a})$ with $\neg E(\tau)$ we conclude by applying the hypothesis that the compensation pair $a \div \overline{a}$ is correct over X. For any failed trace $(\tau, \overline{\tau}) \in \rho_c(a \div \overline{a})$ with $E(\tau)$, we have that $\tau = s\, a\, s$ for some state s such that $s \models E(a)$. Furthermore, the compensation trace for a failed basic activity is empty. It is easy to see that $\Delta_X(\tau) = \mathit{null}$, which concludes the proof for this case.
2. $\delta = \delta_1 \,\square\, \delta_2$. By induction on δ_1 and δ_2.
3. $\delta = \delta_1 \,;\, \delta_2$. We need to distinguish two cases, by the definition of $\rho_c(\delta_1 \,;\, \delta_2)$.
   - For any pair $(\tau \circ \nu, \overline{\nu} \circ \overline{\tau}) \in \rho_c(\delta_1 \,;\, \delta_2)$ such that $(\tau, \overline{\tau}) \in \rho_c(\delta_1)$, $(\nu, \overline{\nu}) \in \rho_c(\delta_2)$, and $\neg E(\tau)$, then we want to prove that if $\mathit{closed}(\tau \circ \nu)$ and $\mathit{closed}(\overline{\nu} \circ \overline{\tau})$ then $\Delta_X(\tau \circ \nu) = \nabla_X(\overline{\nu} \circ \overline{\tau})$.
     As we consider closed traces we can conclude that also $\mathit{closed}(\tau)$ and $\mathit{closed}(\nu)$ with $\mathit{last}(\tau) = \mathit{first}(\nu)$ and the same holds for the compensation. Thus we can apply the induction hypothesis getting $\Delta_X(\tau) = \nabla_X(\overline{\tau})$ and $\Delta_X(\nu) = \nabla_X(\overline{\nu})$. We build the union of these two sets, *i.e.*, $\Delta_X(\tau) \oplus \Delta_X(\nu) = \nabla_X(\overline{\nu}) \oplus \nabla_X(\overline{\tau})$. As $\mathit{last}(\tau) = \mathit{first}(\nu)$ and $\mathit{last}(\overline{\nu}) = \mathit{first}(\overline{\tau})$ we can conclude with Lemma A that $\mathit{first}(\tau)\backslash_X \mathit{last}(\nu) = \mathit{last}(\overline{\tau})\backslash_X \mathit{first}(\overline{\nu})$ which is equivalent to $\Delta_X(\tau \circ \nu) = \nabla_X(\overline{\nu} \circ \overline{\tau})$.
   - For any $(\tau, \overline{\tau}) \in \rho_c(\delta_1 \,;\, \delta_2)$ such that $(\tau, \overline{\tau}) \in \rho(\delta_1)$ and $E(\tau)$, then the result follows immediately from the induction hypotheses on δ_1.
4. $\delta = \delta_1 \parallel \delta_2$. With δ_1, δ_2 serializable. For any $(\tau, \overline{\tau}) \in \rho_c(\delta_1 \parallel \delta_2)$ with $(\nu, \overline{\nu}) \in \rho_c(\delta_1)$, $(\mu, \overline{\mu}) \in \rho_c(\delta_2)$ $\tau \in \nu \parallel \mu$, $\overline{\tau} \in \overline{\nu} \parallel \overline{\mu}$, we need to show that if $\mathit{closed}(\tau)$ and $\mathit{closed}(\overline{\tau})$ then $\Delta_X(\tau) = \nabla_X(\overline{\tau})$.
   Note that in general neither $\mathit{closed}(\nu)$ and $\mathit{closed}(\overline{\nu})$ nor $\mathit{closed}(\mu)$ and $\mathit{closed}(\overline{\nu})$ holds, because the interleaving is defined independently of this.
   According to serializability there exist traces $(\nu', \overline{\nu}') \in \rho_c(\delta_1)$ and $(\mu', \overline{\mu}') \in \rho_c(\delta_2)$ with several different possibilities for a sequential representation (though at least one holds). Without loss of generality we assume that $\tau \bowtie (\nu' \mu')$ and $\overline{\tau} \bowtie (\overline{\nu}'\, \overline{\mu}')$. (The other representations can be treated similarly.)
   From $\mathit{closed}(\nu'\, \mu')$ we know that $\mathit{closed}(\nu')$, $\mathit{closed}(\mu')$ and $\mathit{last}(\nu') = \mathit{first}(\mu')$. The same holds for $\mathit{closed}(\overline{\nu}'\, \overline{\mu}')$. Thus we can apply the induction hypothesis. We obtain $\Delta_X(\nu') = \nabla_X(\overline{\nu}')$ and $\Delta_X(\mu') = \nabla_X(\overline{\mu}')$. As for the sequential case we build the union of the two sets $\Delta_X(\nu') \oplus \Delta_X(\mu') = \nabla_X(\overline{\mu}') \oplus \nabla_X(\overline{\nu}')$. From the equivalences and Lemma A we obtain $\mathit{first}(\nu')\backslash_X \mathit{last}(\mu') = \mathit{last}(\overline{\mu}')\backslash_X \mathit{first}(\overline{\nu}')$ which is equivalent to $\Delta_X(\nu'\, \mu') = \nabla_X(\overline{\nu}'\, \overline{\mu}')$. From the equivalences for serializability we can conclude $\Delta_X(\tau) = \nabla_X(\overline{\tau})$.
5. $\delta = \delta_1^*$ (by induction on the depth of recursion).

■