Journal of Logic and Computation Advance Access published February 26, 2015

Combining transactions and automatic repairs

ANA SOFIA GOMES and JOSÉ JÚLIO ALFERES, *NOVA-LINCS – Departamento de Informática, Faculdade Ciências e Tecnologias, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal.*
E-mail: sofia.gomes@campus.fct.unl.pt; jja@fct.unl.pt

Abstract

External Transaction Logic ($\mathcal{ETR}$) is an extension of logic programming useful to reason about the behaviour of agents that have to operate in a transactional way, in a two-fold environment: an internal knowledge base defining the agent's internal knowledge and rules of behaviour, and an external world where it executes actions and interacts with other entities. Actions performed by the agent in the external world may fail, e.g. because their preconditions are not met, or because they violate some norm of the external environment. The failure to execute some action should lead, in the internal knowledge base, to its complete rollback, following the standard ACID transaction model used e.g. in databases. Since it is impossible to rollback external actions performed in the outside world, external consistency must be achieved by executing compensating operations (or repairs) that revert the effects of the initial executed actions. In $\mathcal{ETR}$, repairs are stated explicitly in the program. With it, every performed external action is explicitly associated with its corresponding compensation or repair. Such user-defined repairs provide no guarantee to revert the effects of the original action. In this article, we define how $\mathcal{ETR}$ can be extended to automatically calculate compensations in case of failure. For this, we start by explaining how the semantics of Action Languages can be used to model the external domain of $\mathcal{ETR}$, and how we can use it to reason about the reversals of actions.

Keywords: External transaction logic, repair plans, automatic compensations.

1 Introduction and motivation

Intelligent agents in a multi-agent setting must work and reason over a two-fold environment: the outside world where the agent acts (external environment), and which may include other agents; and an internal environment comprising the information about the agent's rules of behaviour, preferences about the outside world, its knowledge and beliefs, intentions, goals, etc. An agent may act on the external environment (by executing external actions), but also on the internal environment (where it executes internal actions). Examples of the latter include insertions and deletions in the agent's own knowledge base, updates on its rules of behaviour or preferences.

When performing actions, agents must take into account the possibility of action failure, and what do to in those situations. This is especially relevant inasmuch as the agent has no control over the behaviour of the external world. External actions may fail because their preconditions are not met at the time of intended execution or, in norm regimentation, because the execution of the action would cause the violation of some norm (e.g. as allowed in [11]), or even due to a totally unknown reason to the agent.

The failure of an action should trigger some clean up or repair plan, to undo whatever effects have been caused by the failed action. This is especially important when the action is part of a plan, in which case it may be necessary to undo the effects of previous actions that have succeeded. When the action to undo is an internal action, the undo should be trivial. In fact, since the agent has full control

doi:10.1093/logcom/exv006

over its own internal environment, actions and updates can be made to follow the standard ACID[1] properties of transactions in databases and, as such, the effects made by internal actions are completely discarded. However, since in general, an agent has no control over the external environment, such transactional properties cannot be guaranteed when undoing external actions.

EXAMPLE 1 (Medical diagnosis)
Consider an agent in a medical scenario, that interacts with patients and executes actions such as giving patients some medications. Moreover, the agent should also store, in an internal Knowledge Base (KB), information about patients and treatments, e.g. treatment specifications and history of successful treatments of patients.

In such a scenario, when a patient arrives with a series of symptoms, the agent needs to reason about what should be the treatment applicable to the given patient, but also execute this treatment by giving some medication, and possibly by updating its internal KB, e.g. with the history of successful treatments of patients.

In case a patient later shows a negative reaction to the medication (e.g. signalled by the failure of an action verifying that everything is alright with the patient), something must be done to counter the possible side-effects of the previously given medication. In other words, in that situation a 'repair plan' must be executed. But this is not enough if, meanwhile, the internal KB has been updated. For instance, suppose that after executing the action of giving the medication, the agent stored in its KB the information that the treatment was successful; in that case, besides giving the medication to counter the side-effects of the previous one, the agent should also roll back whatever updates it made in its internal KB. In other words, actions and updates in the internal KB need to be executed transactionally, so as to guarantee that the history of successful treatments does not contain the medication that showed negative effects, which thereby could lead the agent to apply the same treatment again.

Note that, the distinct nature of each KB requires different ways to deal with failures. Namely, while atomic transactions and rollbacks are possible (and desirable) in the internal KB, the external KB requires a more sophisticate way to counter the side-effects of the failed treatments.

In this example, 'what to do to counter the side-effects of a previous unsuccessful treatment' is a typical case of a repair plan, something that can be found in agent languages such as 2APL [10] (plan-repair rules) and 3APL [12, 21] (plan-revision rules). In these languages, it is possible to state, for each plan, which alternative plan should be performed in case something fails. For example, in the previous example, one could state that if some treatment fails, then one should give the patient some alternative medication to counter the effects of the first medication given in the failed treatment. That is, how to address the failure in the external KB, which cannot be simply rolled back to a previous state.

Moreover, in this particular example, it is reasonable to assume that the plan (and more precisely, the treatment) can only be repaired if the agent's specification explicitly states what are the actions to execute for each failed treatment. In other words, it is reasonable to assume that whoever programmed the agent explicitly included in the program the repair plans for each possible external failure. Explicitly programming such repair plans is possible, e.g. in 2APL, where plan-repair rules explicitly include the actions to execute when a given action or plan fails.

However, if one has some knowledge about the external environment, it should be possible for the agent to automatically infer the repair plan in a given failure situation, thus saving the programmer from that task, and from having to anticipate all possible relevant failures.

[1]Where ACID, as usual, stands for Atomicity, Consistency, Isolation and Durability.

EXAMPLE 2 (Supermarket robot)
Imagine a scenario of a robot in a supermarket that has the task to fill up the supermarket's shelves with products. In its internal KB, the agent keeps information about the products' stocks and prices, but also rules on how products should be placed (e.g. 'premium' products should be placed in the shelves with higher visibility). Externally, the agent needs to perform the task of putting products in a given shelf, something that can be encoded in a blocks-world usual manner. In this case, when some action fails in the context of a plan for e.g. arranging the products in some way, the agent, knowing the effects of the actions in the outside world, should be able to infer what actions to perform to restore the external environment to some consistent configuration, upon which some other alternative plan can be started.

Several solutions exist in the literature addressing the problem of reversing actions. For example, the authors of [13] introduce a solution based on Action Languages [15] that reasons about what actions may revert the effects of other actions. For that, the authors define the notions of reverse action, reverse plan and conditional reversals that undo the effects of a given action or set of actions. These notions may allow the automatic inference of plan repairs.

In this article, we propose a logic programming-like language that tackles all the previously mentioned issues. In particular, the language operates over two-fold environments, with both an internal KB and an external environment; it allows for performing actions both in the internal and the external environment; it deals with failure of actions, having a transactional behaviour in the actions performed in the internal KB, and executing repair plans in the external environment; it allows to automatically infer repair plans when there is knowledge about the effects of actions.

Our solution is based on External Transaction Logic ($\mathcal{ETR}$) [18, 19], an extension of Transaction Logic ($\mathcal{TR}$) [3] for dealing with the execution of external actions. In $\mathcal{ETR}$, if a transaction fails after external actions are executed in the environment, then external consistency is achieved by issuing compensating actions to revert the effects of the initial executed actions, whereas consistency of the internal KB is achieved by completely rolling back the executed internal actions. $\mathcal{ETR}$, as its ancestor $\mathcal{TR}$, is a very general language, that relies on the existence of oracles for querying and updating an internal KB and, in the case of $\mathcal{ETR}$, also for dealing with the external environment. Besides recalling the preliminaries of $\mathcal{ETR}$ (Section 2) and of [13] (Section 4), in this article we:

1. formalize how the external oracle in $\mathcal{ETR}$ can be instantiated using action languages in general, and specifically, with action language $\mathcal{C}$ (Section 3);
2. generalize $\mathcal{ETR}$ to deal with repair plans, rather than simply with compensating actions (Section 5);
3. formalize how to automatically infer repair plans when the external environment is expressed as an action language (Section 5); and
4. elaborate on the properties of these repair plans (Section 5.3) and compare our solution with the related work (Section 6).

This article is a revised extended version of the work in [20] for the Special Issue of the *Journal of Logic and Computation* on Computational Logic in Multi-Agent Systems, with additional discussion, proof of the results and further comparison of the related work.

2 External transaction logic

$\mathcal{ETR}$ [19] is an extension of Transaction Logic [3] to deal with actions performed in an external environment of which an agent has no control. With it, one can specify and reason about changes in

arbitrary KBs that are partitioned into an internal and external component. In this setting, internal changes are required to follow the standard ACID model of transactions, while external changes follow a weaker transaction model based on compensations.

In $\mathcal{ETR}$, formulas are read as transactions, and they are evaluated over sequences of KB states (known as *paths*). As such, a formula (or transaction) ϕ is true over a path π iff the transaction successfully executes over that sequence of states. In other words, in $\mathcal{ETR}$ truth means successful execution of a transaction. The logic itself makes no particular assumption about the representation of states, or on how states change. For that, $\mathcal{ETR}$ requires the existence of *oracles* that are incorporated as a parameter of the theory. To reason about the internal KB, $\mathcal{ETR}$ incorporates a data and transition oracle (respectively $\mathcal{O}^d$ and $\mathcal{O}^t$), where the data oracle abstracts the representation of KB states and how to query them, while the transition oracle abstracts the way the states change and how to update them. Similarly, to reason about the external KB, $\mathcal{ETR}$ includes an external oracle $\mathcal{O}^e$ abstracting the representation of states and how to simultaneously query and update them.

To reason about transactions, $\mathcal{ETR}$ defines the concept of model of a theory, which allows one to prove properties of the theory, independently of the paths chosen. Additionally, to reason about specific execution paths where a transaction succeeds, $\mathcal{ETR}$ also defines the notion of executional entailment. In this case, a transaction is entailed by a theory given an initial state, if there is a path starting in that state on which the transaction succeeds. As such, given a transaction and an initial state, the executional entailment determines the path that the KB should follow to succeed the transaction in an atomic way. For non-deterministic transactions several successful paths exist.

2.1 Syntax and oracles

To deal with internal and external actions, $\mathcal{ETR}$ operates over a KB that includes both an internal and an external component. For that, $\mathcal{ETR}$ formally works over two disjoint propositional languages: $\mathcal{L}_P$ (program language), and $\mathcal{L}_\mathcal{O}$ (oracles primitives language). Propositions in $\mathcal{L}_P$ denote actions and fluents that can be defined in the program. As usual, fluents are propositions that can be evaluated without changing the state and actions are propositions that cause evolution of states. Propositions in $\mathcal{L}_\mathcal{O}$ define the primitive actions and queries to deal with the internal and external KB. Additionally, $\mathcal{L}_\mathcal{O}$ can still be partitioned into $\mathcal{L}_i$ and $\mathcal{L}_a$, where $\mathcal{L}_i$ denotes the primitives that query and change the internal KB, while $\mathcal{L}_a$ defines the external actions primitives that can be executed externally. For convenience, it is assumed that $\mathcal{L}_a$ contains two distinct actions `failop` and `nop`, respectively, defining trivial failure and trivial success in the external domain. Further, $\mathcal{L}_a^*$ is defined as the result of augmenting $\mathcal{L}_a$ with expressions $\mathbf{ext}(a, b)$, called external actions, where $a, b \in \mathcal{L}_a$. Such an expression is used to denote the execution of action a, having action b as compensating action. If b is `nop`, then we simply write $\mathbf{ext}(a)$ or a. Note that, there is no explicit relation between a and b: it is possible to define different compensating actions for the same action in the same program. It is thus the programmer's responsibility to determine which is the correct compensation for action a in a given moment.

In this article, we work with the notion of Herbrand instantiation of the language, where the Herbrand universe $\mathcal{U}$ is the set of all ground first-order terms that can be obtained from the function symbols in the language; the Herbrand base $\mathcal{B}$ is a set of all ground atomic formulas; and a classical Herbrand structure is any subset of $\mathcal{B}$.

To build complex formulas, $\mathcal{ETR}$ employs the common first-order logic connectives $\wedge, \vee, \neg, \leftarrow$ and an additional serial conjunction connective $\otimes$. Informally, the formula $\phi \otimes \psi$ represents the action composed by an execution of ϕ followed by an execution of ψ. Then, $\phi \wedge \psi$ defines the simultaneous

execution of ϕ and ψ; while $\phi \vee \psi$ defines the non-deterministic choice of either executing ϕ, ψ or both simultaneously. Finally, $\phi \leftarrow \psi$ is a *rule* saying that one way to satisfy the execution of ϕ is by executing ψ.

DEFINITION 1 ($\mathcal{ETR}$ atoms, formulas and programs)
An $\mathcal{ETR}$ *atom* is either a proposition in $\mathcal{L}_P$, $\mathcal{L}_i$ or $\mathcal{L}_a^*$ and an $\mathcal{ETR}$ literal is either ϕ or $\neg\phi$ where ϕ is an $\mathcal{ETR}$ atom. An $\mathcal{ETR}$ *formula* is either a literal, or an expression, defined inductively, of the form $\phi \wedge \psi$, $\phi \vee \psi$ or $\phi \otimes \psi$, where ϕ and ψ are $\mathcal{ETR}$ formulas.

An *$\mathcal{ETR}$ program is a set of rules of the form $\phi \leftarrow \psi$ where ϕ is a proposition in $\mathcal{L}_P$ and ψ is an $\mathcal{ETR}$ formula.*

EXAMPLE 3
Recall Example 1 regarding a medical diagnosis. A possible (partial) encoding of it in $\mathcal{ETR}$ can be expressed by the following rules:

$$\begin{aligned}
sick(X) &\leftarrow hasFlu(X)\\
hasFlu(X) &\leftarrow \mathbf{ext}(hasFever(X)) \otimes \mathbf{ext}(hasHeadache(X)) \otimes nonSerious(X)\\
nonSerious(X) &\leftarrow \mathbf{ext}(\neg vomiting(X)) \wedge \ldots \wedge \mathbf{ext}(\neg diarrhea(X))\\
treatment(X,Y) &\leftarrow hasFlu(X) \otimes treatFlu(X,Y) \otimes treatmentHistory(X,Y,Z).ins \otimes\\
&\qquad \mathbf{ext}(goodReaction(X,Y))\\
treatFlu(X,Y) &\leftarrow \mathbf{ext}(giveMeds(X,p_1), giveMeds(X,c_1))\\
treatFlu(X,Y) &\leftarrow \mathbf{ext}(giveMeds(X,p_2), giveMeds(X,c_2))
\end{aligned}$$

The predicate $treatment(X,Y)$ denotes a transaction for treating patient X with treatment Y. Then, one can say, e.g. in the fourth rule, that such a transaction succeeds if a patient X has flu and a medicine Y to treat the flu is given to X (i.e. transaction $treatFlu(X,Y)$ succeeds). Additionally, after a treatment is issued, the medical history of the patient should be updated and the agent needs to check if the patient shows a positive reaction to the treatment in question. In this sense, the formula $treatmentHistory(X,Y,Z).ins \otimes \mathbf{ext}(goodReaction(X,Y))$ denotes the action composed by updating the treatment history of patient X followed by externally asking if the patient X had a good reaction to treatment Y. Moreover, treating a patient with a flu is encoded by the non-deterministic transaction $treatFlu(X,Y)$ (fifth and sixth rules) as the external action of giving patient X the medicine p_1 or the medicine p_2. While the action of asking about the reaction of a patient does not need to be repaired, the same is not true for the action of giving a medication. If a failure occurs, then the agent has to compensate for it. This is, e.g. expressed by the external action $\mathbf{ext}(giveMeds(X,p_1), giveMeds(X,c_1))$ where c_1 cancels the effects of p_1.

Crucial notions in $\mathcal{ETR}$ are that of a *state*, and a sequence of states denoted as *path*. A state in $\mathcal{ETR}$ is a pair (D,E), where D (resp. E) is the internal (resp. external) state identifier taken from a set $\mathcal{D}$ (resp. $\mathcal{E}$). The semantics of states is provided by the three oracles mentioned above: a data oracle $\mathcal{O}^d$ that maps elements of $\mathcal{D}$ into transaction formulas; a transition oracle $\mathcal{O}^t$ that maps a pair of elements from $\mathcal{D}$ into transaction formulas; and an external oracle $\mathcal{O}^e$ that maps a pair of elements from $\mathcal{E}$ into transaction formulas. Intuitively $\mathcal{O}^d(D) \models \varphi$ means that, according to the oracle, φ is true in state D, and $\mathcal{O}^t(D_1,D_2) \models \varphi$ (resp. $\mathcal{O}^e(E_1,E_2) \models \varphi$) that φ is true in the transition of internal (resp. external) states from D_1 to D_2 (resp. E_1 to E_2). In this setting, the behaviour of an internal KB e.g. consisting of a relational database, can be modelled as:

EXAMPLE 4 (Internal relational oracles—[2])
A KB made of a relational database can be modelled by having states represented as sets of ground atomic formulas. The data oracle simply returns all these formulas, i.e. $\mathcal{O}^d(D) = D$. Moreover, for

each predicate p in the KB, the transition oracle defines the primitive formulas $p.ins$ and $p.del$, representing the insertion and deletion of p, respectively. Formally,

$$p.ins \in \mathcal{O}^t(D_1, D_2) \text{ iff } D_2 = D_1 \cup \{p\}$$
$$p.del \in \mathcal{O}^t(D_1, D_2) \text{ iff } D_2 = D_1 \setminus \{p\}$$

Moreover, SQL-style bulk updates can also be defined by $\mathcal{O}^t$ as primitives.

Note that, the oracles' language is completely arbitrary and, in the example, the syntax of primitive updates like $p.ins$ could also be defined as $ins.p$ or $insert(p)$. For other oracles examples, such as based in first-order logic, well-founded semantics or scientific oracles, see [2].

For now, we omit the examples of how the external oracle can be instantiated, and consider it as a 'black box' for the remaining of this section. Afterwards, in Section 3, we illustrate how the external oracle can be defined for a KB based on Action Languages.

$\mathcal{ETR}$ formulas are evaluated on paths of the form: $\langle S_1, {}^{A_1} S_2, {}^{A_2} \ldots, {}^{A_{n-1}} S_n \rangle$ where S_i are states and A_i are annotations specifying the actions executed in each transition of states. For instance, $\langle S_1, {}^{\varphi} S_2 \rangle$ is a path of size 2, and where action φ is said to cause the transition of state S_1 into S_2.

2.2 *Model theory and entailment*

As in most logics, $\mathcal{ETR}$'s theory depends on *interpretations*, and since it reasons about the execution of transactions, an interpretation is a mapping from paths to Herbrand structures. Therefore, if $\phi \in M(\pi)$ then, in interpretation M, path π is said to be a valid execution for the formula ϕ. Moreover, the oracles define elementary primitives for the internal and external KB which all interpretations must model, and thus we only consider as interpretations the mappings that comply with the specified oracles:

DEFINITION 2 (Interpretations)
An interpretation is a mapping M assigning a classical Herbrand structure (or $\top$[2]) to every path. This mapping is subject to the following restrictions, for all states D_i, E_j and every formula φ:

1. $\varphi \in M(\langle (D, E) \rangle)$ iff $\mathcal{O}^d(D) \models \varphi$ for any external state E
2. $\varphi \in M(\langle (D_1, E), {}^{\varphi} (D_2, E) \rangle)$ iff $\mathcal{O}^t(D_1, D_2) \models \varphi$ for any external state E
3. $\varphi \in M(\langle (D, E_1), {}^{\varphi} (D, E_2) \rangle)$ iff $\mathcal{O}^e(E_1, E_2) \models \varphi$ for any internal state D

Satisfaction of $\mathcal{ETR}$ formulas over paths, requires the prior definition of operations on paths. For example, the formula $\phi \otimes \psi$ is true (i.e. it successfully executes) in a path that first executes ϕ up to some point in the middle, and executes ψ from then onwards. To deal with this:

DEFINITION 3 (Path splits)
A *split* of a path

$$\pi = \langle S_1, {}^{A_1} \ldots, {}^{A_{i-1}} S_i, {}^{A_i} \ldots, {}^{A_{k-1}} S_k \rangle$$

of size k (k-path) is any pair of subpaths, π_1 and π_2, such that

$$\pi_1 = \langle S_1, {}^{A_1} \ldots, {}^{A_{i-1}} S_i \rangle \text{ and } \pi_2 = \langle S_i, {}^{A_i} \ldots, {}^{A_{k-1}} S_k \rangle$$

for some i $(1 \leq i \leq k)$. In this case, we write $\pi = \pi_1 \circ \pi_2$.

[2]Similar to $\mathcal{TR}$, for not having to consider partial mappings, besides formulas, interpretation can also return the special symbol $\top$. The interested reader is referred to [3] for details.

Importantly, $\mathcal{ETR}$'s model theory provides three satisfaction relations to evaluate formulas over an interpretation M that are needed to deal with external failures.

Classical satisfaction: equivalent to $\mathcal{TR}$'s satisfaction of formulas but integrating paths with an external component. $M,\pi \models_c \phi$ if ϕ can execute on π without failures.

Partial satisfaction: provides the first ingredient to define failures. $M,\pi \models_p \phi$ if either ϕ succeeds without failures (i.e. if $M,\pi \models_c \phi$) or if it fails because a primitive action in ϕ cannot be executed in a given state.

General satisfaction: corresponds to the real satisfaction of formulas making use of the previous two notions. $M,\pi \models \phi$ if ϕ succeeds classically over path π or; if we can split π into $\pi = \pi_1 \circ \pi_2$ such that ϕ fails and recovers form this failure on π_1 (by rolling back internally and compensating externally) and succeeds on π_2.

The first two satisfaction relations represent the building blocks for defining failures and are *not* used to satisfy formulas directly. As it shall be precisely defined, a formula ϕ is said to fail in a path π if ϕ can be partially satisfied but not classically satisfied (i.e. if $M,\pi \not\models_c \phi$ but $M,\pi \models_p \phi$). If this is the case, then recovery is in order. For that we need to roll back internally and compensate externally. This is encoded in $M,\pi \rightsquigarrow \phi$ meaning that π is a recovery path obtained after failing to execute ϕ and executing actions externally (and will be further explained in Definition 8)

Next we continue to define precisely each of these satisfaction relations. We start with the classical and the partial satisfaction. The former relation is similar to satisfaction in the original $\mathcal{TR}$, and a transaction formula is said to be classically satisfied by an interpretation given a path iff the transaction succeeds in the path without failing any action. Afterwards, a transaction is said to be partially (or partly) satisfied by an interpretation given a path, iff the transaction succeeds in the path up to some point where an action may fail.

DEFINITION 4 (Classical satisfaction)
Let M be an interpretation, π a path and ϕ a formula. If $M(\pi) = \top$ then $M,\pi \models_c \phi$; otherwise:

1. **Base case:** $M,\pi \models_c \phi$ iff ϕ is an atom and $\phi \in M(\pi)$
2. **Negation:** $M,\pi \models_c \neg\phi$ iff it is not the case that $M,\pi \models_c \phi$
3. **'Classical' conjunction:** $M,\pi \models_c \phi \wedge \psi$ iff $M,\pi \models_c \phi$ and $M,\pi \models_c \psi$.
4. **Serial conjunction:** $M,\pi \models_c \phi \otimes \psi$ iff $M,\pi_1 \models_c \phi$ and $M,\pi_2 \models_c \psi$ for some split $\pi_1 \circ \pi_2$ of path π.

DEFINITION 5 (Partial satisfaction)
Let M be an interpretation, π a path and ϕ a formula. If $M(\pi) = \top$ then $M,\pi \models_p \phi$; otherwise:

1. **Base case:** $M,\pi \models_p \phi$ iff ϕ is an atom and one of the following holds:
   (a) $M,\pi \models_c \phi$
   (b) $M,\pi \not\models_c \phi$, $\phi \in \mathcal{L}_i$, $\pi = \langle (D,E) \rangle$, $\neg\exists D_i$ s.t. $M, \langle (D,E), ^{\phi}(D_i,E) \rangle \models_c \phi$
   (c) $M,\pi \not\models_c \phi$, $\phi \in \mathcal{L}_a^*$, $\pi = \langle (D,E) \rangle$, $\neg\exists E_i$ s.t. $M, \langle (D,E), ^{\phi}(D,E_i) \rangle \models_c \phi$
2. **Negation:** $M,\pi \models_p \neg\phi$ iff it is not the case that $M,\pi \models_p \phi$
3. **'Classical' conjunction:** $M,\pi \models_p \phi \wedge \psi$ iff $M,\pi \models_p \phi$ and $M,\pi \models_p \psi$
4. **Serial conjunction:** $M,\pi \models_p \phi \otimes \psi$ iff one of the following holds:
   (a) $M,\pi \models_p \phi$ and $M,\pi \not\models_c \phi$
   (b) $\exists$ split $\pi_1 \circ \pi_2$ of path π s.t. $M,\pi_1 \models_c \phi$ and $M,\pi_2 \models_p \psi$

Based on these definitions, we say that a transaction ϕ fails and can be compensated only if $M,\pi \models_p \phi$ but $M,\pi \not\models_c \phi$ where the last state of π stands for the exact point where ϕ fails.

EXAMPLE 5 (Running example)
Consider and internal KB defined as a relational database, as illustrated in Example 4. Furthermore, consider that the external oracle includes the transition definitions: $\mathcal{O}^e(E_1,E_2)\models a$, (i.e. the external execution of a in state E_1 succeeds, and makes the external world evolve into E_2), $\mathcal{O}^e(E_1,E_4)\models c$, and that for every state E, $\mathcal{O}^e(E_2,E)\not\models b$ (i.e. the execution of b in state E_2 fails).
Finally, besides the oracles, consider the following rules defining transaction t:

$$t \leftarrow p.ins \otimes \mathbf{ext}(a,d) \otimes \mathbf{ext}(b,e)$$
$$t \leftarrow q.ins \otimes \mathbf{ext}(c)$$

In this example, the formula $p.ins \otimes \mathbf{ext}(a,d)$ is classically satisfied by all interpretations in the path $\langle(\{\},E_1),^{p.ins}(\{p\},E_1),^{\mathbf{ext}(a,d)}(\{p\},E_2)\rangle$ while $q.ins \otimes \mathbf{ext}(c)$ is classically satisfied in the path $\langle(\{\},E_1),^{q.ins}(\{q\},E_1),^{\mathbf{ext}(c)}(\{q\},E_4)\rangle$. Moreover, given the external oracle definition, it is easy to check that $\mathbf{ext}(b,e)$ cannot succeed in any path starting in state E_2 (as action b always fails in the state left after the execution of a). The idea of partial satisfaction is to identify the path $\langle(\{\},E_1),^{p.ins}(\{p\},E_1),^{\mathbf{ext}(a,d)}(\{p\},E_2)\rangle$ as one that partly satisfies the complex formula $p.ins \otimes \mathbf{ext}(a,d) \otimes \mathbf{ext}(b,e)$ up to some point, though it eventually fails since the external action $\mathbf{ext}(b,e)$ fails.

When a formula fails in a path after the execution of some external action, we have to say how these actions can be *compensated*. To define this, $\mathcal{ETR}$ specifies some auxiliary operations on paths. To start, one has to collect all actions that have been executed in a path and need to be compensated; and to roll back the internal state:

DEFINITION 6 (Rollback path, and sequence of external actions)
Let π be a k-path of the form $\langle(D_1,E_1),^{A_1}(D_2,E_2),^{A_2}\ldots,^{A_{k-1}}(D_k,E_k)\rangle$. The *rollback* path of π is the path obtained from π by:

1. Replacing all D_is by the initial state D_1;
2. Keeping just the transitions where $A_i \in \mathcal{L}_a^*$.

The sequence of external actions of π, denoted $\texttt{Seq}(\pi)$, is the sequence of actions of the form $\mathbf{ext}(a,b)$ that appear in the transitions of the rollback path of π.

In the former definition, $\texttt{Seq}(\pi)$ only collects the external actions that have the form $\mathbf{ext}(a,b)$. Since this operation aims to compensate the executed actions, then actions without compensations are skipped. As such, to define compensations that always fail, one should use the the primitive `failop` in b. Internal actions are also discarded since the rollback path definition simply rolls back to the initial internal state.

With this, a recovery path is obtained from executing each compensation operation defined in $\texttt{Seq}(\pi)$ in the inverse order.

DEFINITION 7 (Inversion, and recovery path)
Let S be a sequence of actions from $\mathcal{L}_a^*$ of the form $\langle\mathbf{ext}(A_1,A_1^{-1}),\ldots,\mathbf{ext}(A_n,A_n^{-1})\rangle$. Then, the inversion of S is the transaction formula $\texttt{Inv}(S)=A_n^{-1}\otimes\ldots\otimes A_1^{-1}$.

Finally, π_r is a *recovery path* of $\texttt{Seq}(\pi)$ w.r.t. M iff $M,\pi_r \models_c \texttt{Inv}(\texttt{Seq}(\pi))$.

Based on these definitions, we can formalize which paths are said to compensate a formula.

DEFINITION 8 (Compensating path for a transaction)
Let M be an interpretation, π a path and ϕ a formula. $M,\pi \rightsquigarrow \phi$ iff *all* the following hold conditions are true:

1. $\exists \pi_1$ such that $M,\pi_1 \models_p \phi$ and $M,\pi_1 \not\models_c \phi$
2. $\exists \pi_0$ such that π_0 is the rollback path of π_1
3. $\mathtt{Seq}(\pi_1) \neq \emptyset$ and $\exists \pi_r$ such that π_r is a recovery path of $\mathtt{Seq}(\pi_1)$ w.r.t. M
4. π_0 and π_r are a split of π, i.e. $\pi = \pi_0 \circ \pi_r$

The previous definition retrieves paths where a formula ϕ is *not* successfully executed, but where external recovery can still be guaranteed. Consequently, consistency preserving paths are only defined for cases where besides the failure of a primitive action, some external actions with compensations were executed. This is so because the operator $\mathtt{Seq}(\pi_1)$ only collects external actions of the form $\mathbf{ext}(a,b)$. This is as expected: if no external actions were executed on π_1 or if all the external actions executed are not meant to be compensated (e.g. if they are external queries), then $\mathtt{Seq}(\pi_1) = \emptyset$. Intuitively, if this is the case then no compensations are needed, and the formula just fails.

Finally, we are able to formalize what (complex) formulas hold on what paths.

DEFINITION 9 (General satisfaction)
Let M be an interpretation, π a path and ϕ a formula. If $M(\pi) = \top$ then $M,\pi \models \phi$; otherwise:

1. **Base case:** $M,\pi \models \phi$ if ϕ is an atom and $\phi \in M(\pi)$
2. **Negation:** $M,\pi \models \neg\phi$ if it is not the case that $M,\pi \models \phi$
3. **'Classical' conjunction:** $M,\pi \models \phi \wedge \psi$ if $M,\pi \models \phi$ and $M,\pi \models \psi$.
4. **Serial conjunction:** $M,\pi \models \phi \otimes \psi$ if $M,\pi_1 \models \phi$ and $M,\pi_2 \models \psi$ for some split $\pi_1 \circ \pi_2$ of π.
5. **Compensating case:** $M,\pi \models \phi$ if $M,\pi_1 \rightsquigarrow \phi$ and $M,\pi_2 \models \phi$ for some split $\pi_1 \circ \pi_2$ of π
6. For no other M,π and ϕ, $M,\pi \models \phi$.

With this notion of satisfaction, a formula ϕ succeeds if it succeeds classically or, if although an external action failed to be executed, the system can recover from the failure and, ϕ can still succeed in an alternative path (item 9). Obviously, recovery only makes sense when external actions are performed before the failure. Otherwise we can just roll back to the initial state and try to satisfy the formula in an alternative branching.

EXAMPLE 6
Recall Example 5 and assume that $\mathcal{O}^e(E_3,E_4) \models c$ and $\mathcal{O}^e(E_2,E_3) \models d$. Then, the rollback path of $\pi = \langle(\{\},E_1),^{p.ins}(\{p\},E_1),^{\mathbf{ext}(a,d)}(\{p\},E_2)\rangle$ is the path $\langle(\{\},E_1),^{\mathbf{ext}(a,d)}(\{\},E_2)\rangle$ and $\mathtt{Seq}(\pi) = \langle \mathbf{ext}(a,d)\rangle$. Furthermore, the path $\langle(\{\},E_2),^{d}(\{\},E_3)\rangle$ is a recovery path of $\mathtt{Seq}(\pi)$ w.r.t. any interpretation M.

Based on these, the complex formula

$$(p.ins \otimes \mathbf{ext}(a,d) \otimes \mathbf{ext}(b,e)) \vee (q.ins \otimes \mathbf{ext}(c))$$

is satisfied both in the path (without compensations)

$$\langle(\{\},E_1),^{q.ins}(\{q\},E_1),^{\mathbf{ext}(c)}(\{q\},E_4)\rangle$$

but also in the path:

$$\langle(\{\},E_1),^{\mathbf{ext}(a,d)}(\{\},E_2),^{d}(\{\},E_3),^{q.ins}(\{q\},E_3),^{\mathbf{ext}(c)}(\{q\},E_4)\rangle$$

where it uses item 9 above.

We can now define what is a model, and the notion of logical entailment where, as usual, an interpretation models a rule if whenever it satisfies the body of the rule, it also satisfies the head.

DEFINITION 10 (Models and logical entailment)
An interpretation M models a rule $head \leftarrow body$ iff for every path π:

$$\begin{array}{l} \text{If } M,\pi \models body \text{ then } M,\pi \models head \text{ and;} \\ \text{If } M,\pi \models_c body \text{ then } M,\pi \models_c head \text{ and;} \\ \text{If } M,\pi \rightsquigarrow body \text{ then } M,\pi \rightsquigarrow head \end{array}$$

An interpretation M is a model of a program P if it models all its rules. In this case, we write $M \models P$.

We say that formula ϕ logically entails formula ψ, and write $\phi \models \psi$, if every model of ϕ is also a model of ψ.

Based on the definition of models, logical entailment specifies a general consequence relations that takes into account all the possible execution paths of a transaction formula. Hence, this entailment can be used to define general equivalence and implication of formulas, as one can express properties like 'whenever transaction ϕ is executed, ψ is also executed' ($\phi \models \psi$) or 'transaction ϕ is equivalent to transaction ψ' ($\phi \models \psi \wedge \psi \models \phi$).

Logical entailment is very powerful since it considers *all* the paths of execution that satisfy a given formula. However, sometimes one needs a simpler kind of reasoning that is concerned only with a particular execution of a formula. As such, besides logical entailment, $\mathcal{ETR}$ supports another entailment called *executional entailment*. Whereas logical entailment allows one to *reason* about $\mathcal{ETR}$ theories, executional entailment provides a logical account to *execute* $\mathcal{ETR}$ programs.

DEFINITION 11 (Executional entailment)
Let P be a program, ϕ be a formula and $S_1, {}^{A_1}\ldots, {}^{A_{n-1}} S_n$ be a path:

$$P, (S_1, {}^{A_1}\ldots, {}^{A_{n-1}} S_n) \models \phi \tag{1}$$

iff $M, \langle S_1, {}^{A_1}\ldots, {}^{A_{n-1}} S_n\rangle \models \phi$ for every model M of P. We also define

$$P, S_1 - \models \phi \tag{2}$$

to be true (and say that ϕ succeeds in P from the state S_1), if there is a path $S_1, {}^{A_1}\ldots, {}^{A_{n-1}} S_n$ that makes (1) true.

EXAMPLE 7
Recall our Example 5. In the program P defined there, it is the case that $P, (\{\}, E_1) - \models t$, i.e. that transaction t can successfully execute in a path starting in state $(\{\}, E_1)$ Particularly, that statement is true because the execution of transaction t holds in the following paths:

$$\begin{array}{l} P, \langle\ (\{\}, E_1), {}^{\mathbf{ext}(a, a_1 \otimes a_2)}(\{\}, E_2), {}^{a_1}(\{\}, E_3), {}^{a_2}(\{\}, E_4), {}^{q.ins} \\ \quad (\{q\}, E_4), {}^{\mathbf{ext}(c, c_1)}(\{q\}, E_5)\rangle \models t \\ P, \langle\ (\{\}, E_1), {}^{q.ins}(\{q\}, E_1), {}^{\mathbf{ext}(c, c_1)}(\{q\}, E_5)\rangle \models t \end{array}$$

This means that, one way of succeeding transaction t, starting from that initial state, is by externally executing a, followed by executing a_1, by a_2, and then by c, and ending with an internal KB with the fact q. Another way of succeeding, is simply by executing c and ending with an internal KB with the fact q.

3 Defining action languages in $\mathcal{ETR}$

The general $\mathcal{ETR}$ is parametrized by a set of oracles that define the elementary primitives to query and update the internal and external KB. However, to deal with specific problems, these oracles must be defined. As illustrated in Example 5, to deal with simple internal KBs, one can define a so-called relational oracle as shown in Example 4.

In contrast, the external oracle $\mathcal{O}^e$ can be left open if the agent knows nothing about the external environment. This is e.g. the case in Example 5, where we have no information about the meaning of an external state, or the complete transitions of states defined by such oracle. In these situations, whenever we need to evaluate an external action, the oracle is called returning either failure or a subsequent successful state (which can be the same state, e.g. if the external action is simply a query).

However, in some scenarios, the agent may have some knowledge about the behaviour of the external world and use it for reasoning. In this work, we consider the case where such knowledge about the external world exists, and it can be formalized in an Action Language [15]. With this in mind, we start by showing how an $\mathcal{ETR}$'s external oracle can be defined to incorporate such formalisms. Afterwards, in Section 5, we use the external oracle defined in this section to automatically infer repair plans.

Every action language defines a series of *laws* describing actions in the world and their effects. Which laws are possible, as well as the syntax and semantics of each law, depends on the action language in question. Several solutions like STRIPS, languages $\mathcal{A}, \mathcal{B}, \mathcal{C}$ or *PDDL*, have been proposed in the literature, each with different applications in mind. A set of laws of each language is called an action program description. The semantics of each language is determined by a transition system, which can be seen as a labelled directed graph, and is characterized by a transition function $\mathcal{T}$.

DEFINITION 12 (General action language)
Let $\langle \{\texttt{true}, \texttt{false}\}, \mathcal{F}, \mathcal{A}\rangle$ be an action language signature, where $\{\texttt{true}, \texttt{false}\}$ are the set of possible truth values, $\mathcal{F}$ is the set of fluent names and $\mathcal{A}$ is the set of action names in the language.

Let $\langle S, V, R\rangle$ be a transition system where, S is the set of all possible states, V is the evaluation function from $\mathcal{F} \times S$ into $\{\texttt{true}, \texttt{false}\}$, and finally R is the set of possible relations in the system defined as a subset of $S \times \mathcal{A} \times S$.

Then, function $\mathcal{T}(A_p)$ denotes the semantics of an action language, and is a mapping from an action program A_p into a labelled transition system $\langle S, V, R\rangle$, associated with A_p.

Intuitively, relations in R have the form $\langle s, A, s'\rangle$, where the states s' are the possible results of the execution of the action A in the state s. Actions can be non-deterministic, and an action is said to be executable in state s if there is at least one tuple $\langle s, A, s'\rangle$ in R.

Based on these notions, we define $\mathcal{ETR}$'s external language $\mathcal{L}^a$ as the union of the set of possible fluents $\mathcal{F}$, and the set of possible actions, i.e. $\mathcal{L}^a = \mathcal{F} \cup \mathcal{A}$ where as usual, fluents are queries that never change the external state, while actions may cause an external state transition.

Moreover, recall that an $\mathcal{ETR}$ state is a pair (D, E) where D and E are, respectively, the internal and external state component, and that both of these components are abstract and, respectively, defined by the internal and external oracles. As such, equipped with a function $\mathcal{T}(A_p)$ as specified in Definition 12, we can define an Action Language external state as a pair of the form (A_p, s), where A_p is an action program description describing the external domain, and s is a state that belongs to the set S, and which precisely defines the current state of the transition system. Based on this alone, the general external oracle $\mathcal{O}^e$ can be defined as follows.

DEFINITION 13 (General action language external oracle)
Let $\langle\{\texttt{true},\texttt{false}\},\mathcal{F},\mathcal{A}\rangle$ be the signature of an action language. Let A_p be an action language program description, $\mathcal{T}(A_p)$ a function that maps an action program A into a transition system $\langle S,V,R\rangle$ and let s be a state from S.

Then, an external oracle $\mathcal{O}^e$ for a general Action Language is defined as:

$$\mathcal{O}^e((A_p,s),(A_p,s')) \models \mathit{action} \text{ iff } \mathit{action} \in \mathcal{A} \wedge \langle s,\mathit{action},s'\rangle \in R$$
$$\mathcal{O}^e((A_p,s),(A_p,s)) \models \mathit{fluent} \text{ iff } \mathit{fluent} \in \mathcal{F} \wedge V(\mathit{fluent},s) = \texttt{true}$$

The previous definition specifies an $\mathcal{ETR}$ external oracle $\mathcal{O}^e$ for any action language framework, based on a transition system defined as $\langle S,V,R\rangle$. Such specification is still very general and thus, to be more concrete, let us show one instantiation of this, with action language $\mathcal{C}$ [17]. In the context of multi-agent systems, language $\mathcal{C}$ and its extensions like $\mathcal{C}^+$ [16], are traditionally used to represent norms and protocols (e.g. auction, contract formation, negotiation, rules of procedure, communication, etc.) [1, 25].

DEFINITION 14 (Action language $\mathcal{C}$)
Let $\langle\{\texttt{true},\texttt{false}\},\mathcal{F},\mathcal{A}\rangle$ be the signature of an action language.

A *state formula* in action language $\mathcal{C}$ is a propositional combination of names in $\mathcal{F}$, while a *formula* is a propositional combination of names in $\mathcal{F}$ and names in $\mathcal{A}$. A *static law* is a statement of the form: '**caused** F **if** G', such that F and G are state formulas. A *dynamic law* is a statement of the form: '**caused** F **if** G **after** U', such that F and G are state formulas and U is a formula.

An action program description A_p of language $\mathcal{C}$ is a set of static and dynamic laws.

As defined, an action program of language $\mathcal{C}$ is defined by a set of static laws and dynamic laws. In this context, a static law is used to express constraints that hold in all states, whereas a dynamic law defines what changes and what stays the same after the execution of U. An important notion is that the action language $\mathcal{C}$ supports concurrent actions. As a consequence of this, in a transition $\langle s_1,A,s_2\rangle$, A is not read as an individual action but as a subset of $\mathcal{A}$. Intuitively, to execute A from s_1 to s_2 means to execute concurrently the 'elementary actions' represented by the action symbols in A changing the state s_1 into s_2 (where all elementary actions in A are assumed to have the same duration).

Additionally, a state s in action language $\mathcal{C}$ is defined as the interpretation of the set of fluents $\mathcal{F}$ that is closed under the static laws. That is, for every static law '**caused** F **if** G' defined in the action description, and every state s, we say that s must satisfy F if s satisfies G. Based on this, the interpretation function V for a state is simply defined as $V(\mathit{fluent},s) = s(\mathit{fluent})$.

The set of valid relations R is defined based on the notion of *reduct* and causal explanation. These notions can be expressed as follows.

DEFINITION 15 (Causally explained transitions)
Let A_p be an action description defined as a set of static and dynamic laws, and $\langle s_0,A,s_1\rangle$ any transition. The *reduct* of A_p relative to $\langle s_0,A,s_1\rangle$ (denoted as $A_p^{\langle s_0,A,s_1\rangle}$) is the set consisting of:

- F for all static laws in A_p such that s_1 satisfies G
- F for all dynamic laws in A_p such that s_1 satisfies G and $s_0 \cup A$ satisfies U

We say that $\langle s_0,A,s_1\rangle$ is *causally explained* w.r.t. A_p if s_1 is the only state that satisfies the reduct $A_p^{\langle s_0,A,s_1\rangle}$.

Recall that states in action language $\mathcal{C}$ can be seen as interpretations, and thus any state s contains the set of fluents that are true at a given moment w.r.t. the set of static laws (i.e. that are closed under the static laws of A_p). Then, the latter definition states that the execution of A in state s_0 leads to state s_1, if s_1 is completely characterized by the applicable laws given state s_0 and the set of actions A.

Additionally, since the external oracle is defined for elementary actions rather than for sets of actions, we can define the relation R of $\mathcal{T}(A_p)$ as follows: $\langle s_0, a, s_1\rangle \in R$ iff $\langle s_0, A, s_1\rangle$ is causally explained w.r.t. A_p and $a \in A$. Based on these notions, we can now redefine our external oracle for action language $\mathcal{C}$.

DEFINITION 16 (Action language $\mathcal{C}$ external oracle)
Let $\langle\{\texttt{true}, \texttt{false}\}, \mathcal{F}, \mathcal{A}\rangle$ be the signature of an action language. Let A_p be an action language program description, containing a set of static laws and dynamic laws. Let R be the set of tuples $\langle s_0, a, s_1\rangle$ such that $\langle s_0, a, s_1\rangle$ is causally explained w.r.t. A_p and $a \in A$.

Then, an external oracle $\mathcal{O}^e$ for Action Language $\mathcal{C}$ is defined as:

$$\mathcal{O}^e((A_p, s), (A_p, s')) \models \mathit{action} \text{ iff } \mathit{action} \in \mathcal{A} \wedge \langle s, \mathit{action}, s'\rangle \in R$$
$$\mathcal{O}^e((A_p, s), (A_p, s)) \models \mathit{fluent} \text{ iff } \mathit{fluent} \in \mathcal{F} \wedge s(\mathit{fluent}) = \texttt{true}$$

The latter notion of external oracle defines the semantics of an external environment described in action language $\mathcal{C}$, on which the agent may execute actions. By plugging this definition into $\mathcal{ETR}$, one can reason about the possible behaviours of an agent that respect some transactional properties. In other words, one can reason about the paths where the agent can execute its goals transactionally, and where, in case of failure, internal actions are always rolled back, and external actions are compensated. Additionally, $\mathcal{ETR}$'s proof procedure, as defined in [19], can also be used with this external oracle definition to compute these paths.

However, correctness of recovery upon failure is still dependent on the repairs defined by the programmer. To overcome this, in the next section we formalize the concept of action reversals. Namely, we specify what actions can reverse the effects of other actions based on the semantics of the action language $\mathcal{C}$. Then, in Section 5, we extend the previous external oracle definition to incorporate automatic computation of action repairs, based on this concept of action reversals.

4 Reverse actions in action languages

Our automatic inference of repairs for external actions builds upon the work of [13]. Thus, before defining how to automatically infer repair plans in $\mathcal{ETR}$, we briefly overview [13]'s action reverses, adapting it for the action language framework defined above.

We start with the notion of *trajectory* of a sequence of actions. Intuitively, we say that a state s_f is the trajectory of a sequence of actions applied to state s_i if, there is a trace from s_i to s_f by executing the given sequence of actions. Since non-deterministic actions are possible, this trajectory function is a mapping from $S \times \mathcal{A}$ into $\wp(S)$, and the result of $\texttt{traj}(s_0; [a_0 \otimes \ldots \otimes a_{m-1}])$ is the set of possible end states when executing $a_0 \otimes \ldots \otimes a_{m-1}$ in state s_0.

DEFINITION 17 (Trajectory of a sequence of actions)
We define the trajectory of a sequence of actions $a_0 \otimes \ldots \otimes a_{m-1}$ in state s_0 as the set S (denoted as $\texttt{traj}(s_0; [a_0 \otimes \ldots \otimes a_{m-1}]) = S$) iff:

$$\forall s_f \in S \textbf{ then } \exists s'_1, \ldots, s'_m \text{ s.t. } \langle s_0, a_0, s'_1\rangle \in R \wedge \langle s'_i, a_i, s'_{i+1}\rangle \in R \text{ and } s'_m = s_f$$

where $(1 \leq i \leq m-1)$.

We say that s_f is a possible trajectory of $a_0 \otimes \ldots \otimes a_{m-1}$ if it can be reached when applied to s_0 by executing $a_0 \otimes \ldots \otimes a_{m-1}$, i.e. if $s_f \in \texttt{traj}(s_0; [a_0 \otimes \ldots \otimes a_{m-1}])$.

Then, we can define the notion of reverse action. A reverse action states a relation between two singleton actions based on the set of their transition relations. We say that an action a^{-1} is a reverse action of a if whenever we execute a^{-1} after we execute a, we always obtain the (initial) state before the execution of a. This is encoded as follows.

DEFINITION 18 (Reverse action)
Let a, a^{-1} be actions in $\mathcal{A}$. We say that an action a^{-1} reverses a iff

$$\forall s_1, s_2 \text{ if} \langle s_1, a, s_2 \rangle \in R \text{ then } \exists s. \langle s_2, a^{-1}, s \rangle \in R \text{ and } \forall s. \langle s_2, a^{-1}, s \rangle \in R, s = s_1$$

In this case, we write $\texttt{revAct}(a; a^{-1})$.

Besides the notion of reverse action, the authors of [13] also introduce the notion of reverse plan. Since a single action may not be enough to reverse the effects of another action, the notion of reverse is generalized into a sequence of actions, or *plan*. A reverse plan defines what sequences of actions are able to reverse the effects of one action. This is encoded as follows.

DEFINITION 19 (Reverse plan)
Let $a, a_0, \ldots, a_{m-1}$ be actions in $\mathcal{A}$. We say that $a_0 \otimes \ldots \otimes a_{m-1}$ is a plan that reverses action a iff $\forall s_1, s_2$ s.t. $\langle s_1, a, s_2 \rangle \in R$ then $\exists s'$ s.t. $s' \in \texttt{traj}(s_2; [a_0 \otimes \ldots \otimes a_{m-1}])$ and $\forall s'$ s.t. $s' \in \texttt{traj}(s_2; [a_0 \otimes \ldots \otimes a_{m-1}])$ then $s' = s_1$. In this case, we write $\texttt{revPlan}(a; [a_0 \otimes \ldots \otimes a_{m-1}])$.

Intuitively, a reverse plan is a generalization of a reverse action, as every reverse action $\texttt{revAct}(a, a')$ is a reverse plan of size one: $\texttt{revPlan}(a, [a'])$.

The previous definitions state a strong relation between an action and a sequence of actions that holds for *any* state in the set of states defined in the framework. That is, a sequence of actions is a reverse plan of a given action, if the sequence can always be applied after the execution of a and, in all the transitions defined in the set R, the application of this sequence always leads to the state before the execution of a.

However, some states may prevent the existence of a reverse plan. That is, an action may have a reverse plan under some conditions, that do not necessarily hold at every reachable state. Thus, we need a weaker notion of reverse that takes into account the information of the states, e.g. values of some fluents obtained by sensing. By restraining the states where the reverse plan is applied, we might get reverse plans that were not applicable before. This is the idea of conditional reversal plan formalized as follows.

DEFINITION 20 (Conditional reversal plan)
Let $a, a_0, \ldots, a_{m-1}$ be actions in $\mathcal{A}$. We say that $a_0 \otimes \ldots \otimes a_{m-1}$ is a $\phi;\psi$-reverse plan that reverses action a back iff:

$$\begin{aligned} &\forall s_1, s_2 : V(s_2, \phi) = V(s_1, \psi) = \texttt{true} \text{ if } \langle s_1, a, s_2 \rangle \in R \text{ then} \\ &\quad \exists s' \text{ s.t. } s' \in \texttt{traj}(s_2; [a_0 \otimes \ldots \otimes a_{m-1}]) \text{ and} \\ &\quad \forall s' \text{ s.t. } s' \in \texttt{traj}(s_2; [a_0 \otimes \ldots \otimes a_{m-1}]) \Rightarrow s' = s_1 \end{aligned}$$

EXAMPLE 8
Consider the following example adapted from [13] where putting a puppy into water makes the puppy wet, and drying a puppy with a towel makes it dry. The possible states and

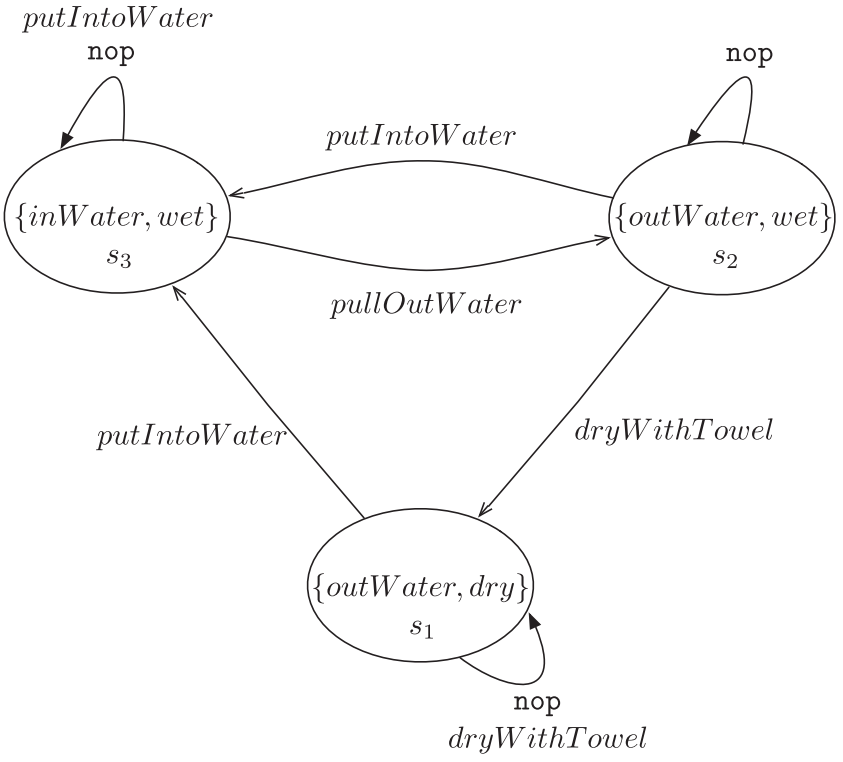

FIGURE 1. Illustration of Example 8.

transitions of this example are illustrated in Figure 1. Based on this representation, we have `revAct`(*putIntoWater*; *pullOutWater*), i.e. *putIntoWater* is a *reverse action* of *pullOutWater*. Furthermore, while a generic reverse plan does not exist, [*pullOutWater* $\otimes$ *dryWithTowel*] is said to be a $\top$; {*dry*} reverse plan that reverts action *putIntoWater*.

5 $\mathcal{ETR}$ with automatic compensations

After defining the reversals of actions for action languages, we can now show how $\mathcal{ETR}$'s external oracle can be instantiated to use these definitions and automatically infer what is the correct repair plan for each action.

However, we do not need such a strong and generic notion of reverse action as the one defined in [13]. In fact, both reverse actions and reverse plans are defined disregarding the initial state where they are being applied. In contrast, when defining compensations or repairs of actions in $\mathcal{ETR}$, we already have information about the specific states where the repairs will be applied. This calls for a weaker notion of reverse action and reverse plan, defined for a pair of states rather than for a given action.

DEFINITION 21 (Situated reverse action)
We say that an action a^{-1} reverses s_2 into s_1 iff $\exists s.\langle s_2, a^{-1}, s\rangle \in R$ and $\forall s.\langle s_2, a^{-1}, s\rangle \in R, s = s_1$. In this case we write `revAct`$(s_1, s_2; a^{-1})$.

The previous definition specifies a reverse action for a pair of states. Intuitively, we say that action a is a reverse action for states s_1 and s_2 iff a can be executed in state s_2 and *all* the transitions that exist in the set of relations R w.r.t. action a applied to state s_2 end in state s_1.

As in [13], instead of only considering singleton actions, we also define the notion of situated reverse plan to specify sequences of actions that are able to reverse the effects of one action. As such, $\texttt{revPlan}(s_1, s_2; [a_0 \otimes \ldots \otimes a_{m-1}])$ states that the sequence of actions $a_0 \otimes \ldots \otimes a_{m-1}$ always restores s_1 when executed in state s_2. For that, the KB may pass through m arbitrary states necessarily ending in s_2.

DEFINITION 22 (Situated reverse plan)
We say that $a_0 \otimes \ldots \otimes a_{m-1}$ is a plan that reverses s_2 back to s_1 iff:

$$\exists s_f \text{ s.t. } s_f \in \texttt{traj}(s_2; [a_0 \otimes \ldots \otimes a_{m-1}]) \text{ and}$$
$$\forall s_f, \text{ if } s_f \in \texttt{traj}(s_2; [a_0 \otimes \ldots \otimes a_{m-1}]) \text{ then } s_f = s_1$$

In this case, we write $\texttt{revPlan}(s_1, s_2; [a_0 \otimes \ldots \otimes a_{m-1}])$

Clearly, several reverse plans may exist restoring s_1 from state s_2. Moreover, some reverse plans may be better than others. For example, imagine that in a state s_i there is an action a_i that always leads us to the same state s_i, i.e. $\langle s_i, a_i, s_i \rangle \in R$. If there is a plan to restore the system back from s_2 to s_1 passing through state s_i, then there are several plans where the only difference is the amount of times we execute the 'dummy' action a_i. For example, in Example 8 it is easy to see that the action `nop` can be incorporated as many times as desired in any reverse plan.

Since recovery is a sensitive operation, to minimize the amount of operations to be executed, we define the notion of shorter reverse plans. In a simple version, a shorter reverse plan $\texttt{revPlan}_s(s_1, s_2; [a_1 \otimes \ldots \otimes a_m])$ is a reverse plan where the number of actions to be executed is minimal (i.e. there is no other $\texttt{revPlan}(s_1, s_2; [a_1 \otimes \ldots \otimes a_n])$ with $n < m$). Note that alternative minimality criteria could be defined, such as where only certain actions are minimized (e.g. `nop` actions), or by extending the action language framework with actions associated with weights or costs and, in the latter case, define minimality w.r.t. the minimal total cost that could be achieved. However, elaborating on these other criteria is outside the scope of this article.

EXAMPLE 9
Recall Example 8 and the states and transitions defined in Figure 1. Here we can conclude that $\texttt{revPlan}_s(s_1, s_3; [\mathit{pullOutWater} \otimes \mathit{dryWithTowel}])$ and that $\texttt{revAct}(s_3, s_2; \mathit{putIntoWater})$.

5.1 *Goal reverse plans*

The previous notions define a reverse action or a reverse plan for a pair of states s_1 and s_2, reverting the system from state s_2 back to state s_1, and imposing that the final state obtained is *exactly* s_1. However, it may happen that, for some pair of states, a reverse plan does not exist. Furthermore, if some information is provided (e.g. by the programmer) about the state that we intend to reach, then we might still achieve a state where this condition holds. This can be useful when the agent has to find repairs to deal with norm violations. For example, it may not be possible to return to the exact state before the violation, but it may be possible to reach a consistent state where the agent complies with all the norms.

This corresponds to the notion of goal reverse plans that we introduce here. Based on a state formula ϕ characterizing the state that we want to reach, then $\texttt{goalRev}(\phi, s_2; [a_0 \otimes \ldots \otimes a_{m-1}])$

says that the sequence $a_0 \otimes \ldots \otimes a_{m-1}$ reverses the system from s_2 into a consistent state s where the state formula ϕ holds:

DEFINITION 23 (Goal reverse plan)
We say that $a_0 \otimes \ldots \otimes a_{m-1}$ is a goal plan that reverses s_2 to a state where ϕ holds iff

$$\exists s' \text{ s.t. } s' \in \texttt{traj}(s_2; [a_0 \otimes \ldots \otimes a_{m-1}]) \text{ and}$$
$$\forall s', \text{ if } s' \in \texttt{traj}(s_2; [a_0 \otimes \ldots \otimes a_{m-1}]) \text{ then } V(\phi, s') = \texttt{true}$$

In this case, we write $\texttt{goalRev}(\phi, s_2; [a_0 \otimes \ldots \otimes a_{m-1}])$.

As before, to preserve efficiency of plans, we define the notion of shorter goal reverse plan. $\texttt{goalPlan}_s(\phi, s_2; [a_1 \otimes \ldots \otimes a_m])$ holds, if the sequence $a_1 \otimes \ldots \otimes a_m$ is a sequence with minimal length that takes s_2 into a state where ϕ is true.

Note that the previous definition can also be seen as a formulation of a classical planning problem. A solution to classical planning problem consists in finding a sequence of actions $\alpha_1, \ldots, \alpha_n$ that when executed in a given initial state S_0, results in a final state S_f in which the intended given goal G holds. Similarly, Definition 23 finds the sequence of actions $a_0 \otimes \ldots \otimes a_{m-1}$ that when executed from state S_2 always achieves a state on which formula (or goal) ϕ is satisfied.

5.2 *External oracle for action languages with automatic compensations*

We can now make it precise as to how and when repairs are calculated in $\mathcal{ETR}$'s semantics, and what changes of the $\mathcal{ETR}$'s language are needed to deal with these automatic repairs.

Moreover, in addition to automatic inferred repairs, we want to give the programmer the option of explicitly defining compensations for external actions. These are useful in scenarios where the agent knows exactly how to repair an action, even when this knowledge is not directly present in the external oracle specification. This is e.g. the case of Example 1 repairs, where the information about which treatments should be chosen for the patient are part of the agent's beliefs and knowledge rather than part of the external world's information.

However, since the agent may need more than one action to repair the effects of a given external action (e.g. in Example 1 it may be necessary to give the patient a series of medications to repair the side-effects of the previously given one), we also extend these explicitly defined compensations to plans.

Consequently, the language of $\mathcal{ETR}$ is augmented so that external actions can appear in a program in three different ways: (i) without any kind of compensation associated, i.e. $\mathbf{ext}(a, \texttt{nop})$, and in this case we write $\mathbf{ext}(a)$ or simply a, where $a \in \mathcal{L}_a$; (ii) with a user-defined repair plan, written $\mathbf{ext}(a, b_1 \otimes \ldots \otimes b_j)$ where $a, b_i \in \mathcal{L}_a$; (iii) with an automatic repair plan, denoted $\mathbf{extA}(a[\phi])$, where $a \in \mathcal{L}_a$, ϕ is an external state formula, and an external state formula is a conjunction of external fluents. Formally:

DEFINITION 24
An $\mathcal{ETR}$ *atom* is either a proposition in $\mathcal{L}_P$, $\mathcal{L}_i$ or an external atom. An external atom is either a proposition in $\mathcal{L}_a$ (where $\mathcal{L}_a = \mathcal{F} \cup \mathcal{A}$), $\mathbf{ext}(a, b_1 \otimes \ldots \otimes b_j)$ or $\mathbf{extA}(a[\phi])$ where $a, b_i \in \mathcal{L}_a$ and ϕ is an external state formula. An $\mathcal{ETR}$ literal is either ϕ or $\neg\phi$ where ϕ is an $\mathcal{ETR}$ atom. An external state formula is a either a literal from $\mathcal{F}$ or an expression $\phi \wedge \psi$ where ϕ and ψ are external state formulas. An $\mathcal{ETR}$ *formula* is either a literal, or an expression, defined inductively, of the form $\phi \wedge \psi$, $\phi \vee \psi$ or $\phi \otimes \psi$, where ϕ and ψ are $\mathcal{ETR}$ formulas. An $\mathcal{ETR}$ *program is a set of rules of the form* $\phi \leftarrow \psi$ *where* ϕ *is a proposition in* $\mathcal{L}_P$ *and* ψ *is an* $\mathcal{ETR}$ *formula.*

Intuitively, **extA**($a[\phi]$) stands for '*execute the external action a, and if something fails, automatically repair the action's effects either leading to the state just before a was executed, or to a state where* ϕ *holds*'. When one wants the repair to restore the system to the very state just before a was executed, one may simply write **extA**(a) (equivalent to **extA**($a[\bot]$)).

EXAMPLE 10
With this modified language one can write, e.g. for the situation described in Example 2, rules like the ones below. Intuitively, these rules say that: to place a product one should decrease the stock and then place the product; one can place a product in a better shelf, or in a normal shelf in case the product is not premium. Moreover, moving a product to a given shelf is an external action that can be automatically repaired based on existing information about the external world. Consequently, **extA**($move(X,w,betterShelf)$) means that, if something fails after the agent has moved X from the warehouse into a better shelf, then a repair plan will be automatically defined for this action by the semantics and the oracle:

$$\begin{aligned} placeProduct(X) &\leftarrow decreaseStock(X) \otimes X > 0 \otimes placeOne(X) \\ decreaseStock(X) &\leftarrow stock(X,S) \otimes stock(X,S).del \otimes stock(X,S-1).ins \\ placeOne(X) &\leftarrow \mathbf{extA}(move(X,w,betterShelf)) \\ placeOne(X) &\leftarrow \neg premium(X) \otimes \mathbf{extA}(move(X,w,normalShelf)) \end{aligned}$$

Note that, the semantics must ensure that the external world is always left consistent by the agent in any possible execution. Particularly, whenever it is not possible to place a non-premium product in the better shelf, a repair plan is executed to put the product back in the warehouse where the agent can try to put the product in the normal shelf; if it is not possible to put the product in either shelf (or to put a premium product in the better shelf), then a repair plan is executed to put the product back in the warehouse, and the stock is rolled back to its previous value (and the transaction fails).

A simplified version of the external environment (oracle) can be described by the following $\mathcal{C}$ program that includes the definition of blocks-world-like actions (where, as usual, we use **inertial** F to stand for **caused** F **if** F **after** F, and α **causes** F **if** G for F **if** $\top$ **after** $\alpha \wedge G$):

caused	$clearBS$	**if** $\neg on(X,betterShelf)$	
caused	$\neg clearBS$	**if** $on(X,betterShelf)$	
caused	$\bot$	**if** $on(X,Y) \wedge on(X,Z) \wedge Y \neq Z$	
$move(X,w,betterShelf)$	**causes** $on(X,betterShelf)$	**if** $on(X,w)$	
$move(X,w,normalShelf)$	**causes** $on(X,normalShelf)$	**if** $on(X,w)$	
$move(X,betterShelf,w)$	**causes** $on(X,w)$	**if** $on(X,betterShelf)$	
$move(X,normalShelf,w)$	**causes** $on(X,w)$	**if** $on(X,normalShelf)$	
$move(X,w,betterShelf)$	**causes** $\bot$	**if** $\neg clearBS$	
inertial $on(X,Y)$			

where we assume that a block can only be moved into a normal or better shelf when moved from the warehouse w; and that a block can only be moved to the better shelf if that shelf is clear (i.e. if $\neg clearBS$ holds).

To illustrate the behaviour of this simple example of rules and external environment, let us show the execution of $P, S \text{--} \models placeProduct(b) \otimes placeProduct(a)$, where S is an initial state described as:

$$S = \langle (premium(a), stock(a,1), stock(b,1)), (on(a,w), on(b,w), clearBS) \rangle$$

Clearly, in this very simple example, the two paths satisfying the actions $placeProduct(b) \otimes placeProduct(a)$ are the ones presente below.. The first one defines the execution where b is inserted

directly in a normal shelf, and no failure occurs (with some obvious abbreviations):

$$
\begin{array}{l}
P, \langle((prem(a), stock(a,1), stock(b,1)), (on(a,w), on(b,w), clearBS)), {}^{stock(b,1).del} \\
((prem(a), stock(a,1)), (on(a,w), on(b,w), clearBS)), {}^{stock(b,0).ins} \\
((prem(a), stock(a,1), stock(b,0)), (on(a,w), on(b,w), clearBS)), {}^{\mathbf{ext}(mv(b,w,nS), mv(b,nS,w))} \\
((prem(a), stock(a,1), stock(b,0)), (on(a,w), on(b,nS), clearBS)), {}^{stock(a,1).del} \\
((prem(a), stock(b,0)), (on(a,w), on(b,nS), clearBS)), {}^{stock(a,0).ins} \\
((prem(a), stock(a,0), stock(b,0)), (on(a,w), on(b,nS), clearBS)), {}^{\mathbf{ext}(mv(a,w,bS), mv(a,bS,w))} \\
((prema), stock(a,1), stock(b,0)), (on(a,bS), on(b,nS), \neg clearBS)\rangle \\
\qquad\qquad \models placeProduct(b) \otimes placeProduct(a)
\end{array}
$$

However, as a valid alternative execution, the agent could have first tried to insert b in a better shelf, discovered that a could no longer be inserted in the better shelf because it is not clear, fail the action, remove b from the better shelf back to the warehouse as a compensation, insert b in the normal shelf, and finally insert a in the better shelf as intended. This execution corresponds to the other path satisfying the executional entailment above:

$$
\begin{array}{l}
P, \langle(prem(a), stock(a,1), stock(b,1)), (on(a,w), on(b,w), clearBS)), {}^{\mathbf{ext}(mv(b,w,bS), mv(b,bS,w))} \\
(prem(a), stock(a,1), stock(b,1)), (on(a,w), on(b,bS), \neg clearBS)), {}^{mv(b,bS,w)} \\
(prem(a), stock(a,1), stock(b,1)), (on(a,w), on(b,w), \neg clearBS), {}^{stock(b,1).del} \\
((prem(a), stock(a,1)), (on(a,w), on(b,w), clearBS)), {}^{stock(b,0).ins} \\
((prem(a), stock(a,1), stock(b,0)), (on(a,w), on(b,w), clearBS)), {}^{\mathbf{ext}(mv(b,w,nS), mv(b,nS,w))} \\
((prem(a), stock(a,1), stock(b,0)), (on(a,w), on(b,nS), clearBS)), {}^{stock(a,1).del} \\
((prem(a), stock(b,0)), (on(a,w), on(b,nS), clearBS)), {}^{stock(a,0).ins} \\
((prem(a), stock(a,0), stock(b,0)), (on(a,w), on(b,nS), clearBS)), {}^{\mathbf{ext}(mv(a,w,bS), mv(a,bS,w))} \\
((prema), stock(a,1), stock(b,0)), (on(a,bS), on(b,nS), \neg clearBS)\rangle \\
\qquad\qquad \models placeProduct(b) \otimes placeProduct(a)
\end{array}
$$

Recall from Definition 6, that the internal changes before the failure are not reflected in the final path of execution, as they are rolled back. Thus, only the external failures (in our case, the ones corresponding to moving b from the warehouse into a better shelf, and then back to the warehouse) appear in this path where a transaction succeeds.

Nota also that in this very simple example the reverse of moving an object from the warehouse to a shelf, is always to move it back in the warehouse, e.g:

$$\texttt{revPlan}(\texttt{(on(a,w), on(b,w), clearBS), (on(a,w), on(b,bS), }\neg\texttt{clearBS)), [move(b,bS,w)])}$$

In contrast to the semantics of the original $\mathcal{ETR}$, which is independent of the defined oracles, the semantics of this new language with actions of the form $\mathbf{extA}(a[\phi])$, can only be defined given specific oracles that allow the inference of repair plans. For example, for external environments described by action languages, an external state is a pair, with the action program A_p representing the external domain and a state of the transition system, and the external oracle $\mathcal{O}^e$ is as follows (where $\mathcal{T}(A_p) = \langle S, V, R\rangle$):

DEFINITION 25 (Action language oracle)
Let f, a be atoms in $\mathcal{L}_a$ s.t. f is a fluent in $\mathcal{F}$ and a is an action in $\mathcal{A}$. Let A_p be an action program description and s_i a state, and (A_p, s_i) be an external state identifier in $\mathcal{ETR}$.

1. $\mathcal{O}^e((A_p, s_1), (A_p, s_1)) \models f$ iff $V(f, s_1) = \texttt{true}$

2. $\mathcal{O}^e((A_p,s_1),(A_p,s_2)) \models a$ iff $\langle s_1,a,s_2\rangle \in R$
3. $\mathcal{O}^e((A_p,s_1),(A_p,s_2)) \models \mathbf{ext}(a, b_1 \otimes \ldots \otimes b_n)$ iff $\langle s_1,a,s_2\rangle \in R$
4. $\mathcal{O}^e((A_p,s_1),(A_p,s_2)) \models \mathbf{ext}(a[\phi], a_0^{-1} \otimes \ldots \otimes a_{m-1}^{-1})$ iff one holds:

   (a) $\langle s_1,a,s_2\rangle \in R \wedge \texttt{revPlan}_s(s_1,s_2;[a_0^{-1} \otimes \ldots \otimes a_{m-1}^{-1}])$; or else

   (b) $\langle s_1,a,s_2\rangle \in R \wedge (\neg\exists a_0^{-1} \otimes \ldots \otimes a_{m-1}^{-1}$ s.t. $\texttt{revPlan}_s(s_1,s_2;[a_0^{-1} \otimes \ldots \otimes a_{m-1}^{-1}])) \wedge \texttt{goalRev}_s(\phi,s_2;[a_0 \otimes \ldots \otimes a_{m-1}])$

Items 25 and 25 above define how the oracle satisfies external actions with compensations. If the agent wants to explicitly define $b_1 \otimes \ldots \otimes b_n$ as the reverse plan for action a, then $\mathbf{ext}(a, b_1 \otimes \ldots \otimes b_n)$ is evaluated solely by what the oracle knows about a, holding in a transition iff a holds in that transition of states.

When the agent wants to infer a repair plan for a, then these repairs are computed based on the notions of reverse plan and goal reverse defined previously. Namely, the formula $\mathbf{ext}(a[\phi], a_0^{-1} \otimes \ldots \otimes a_{m-1}^{-1})$ holds, iff a holds in the transition s_1 into s_2, and $a_0^{-1} \otimes \ldots \otimes a_{m-1}^{-1}$ is a shorter reverse plan to repair s_2 back to s_1 or, if $a_0^{-1} \otimes \ldots \otimes a_{m-1}^{-1}$ is a shorter goal plan to repair s_2 into a state where the state formula ϕ holds.

The order in which we wrote items 4a and 4b is not arbitrary. Goal reverse plans provide an elegant solution to relax the necessary conditions to obtain repairs plans and are especially useful in scenarios where it is not possible to return to the initial state before executing the external action, as e.g. in norms or contracts violations. However, care must be taken when defining the external state formula ϕ of an external action $\mathbf{extA}(a[\phi])$. In fact, if ϕ provides only a very incomplete description of the state that we want to achieve, then we might achieve a state substantially different from the intended one. Particularly, although we constrain the applicability of goal reverse plans to the ones that are shorter, there is no guarantee that the changes of these plans are minimal (w.r.t. the amount of fluents that are different from the previous state). Guaranteeing such a property represents a belief revision problem and is, at this moment, out of the scope of this work.

Finally, compensations can be instantiated by changing the definition of interpretation (Definition 2) which now determines how to deal with automatic repairs. Recall that an external state E is defined as the pair (A_p,s), where A_p is the action description program, and s is a state.

DEFINITION 26 (Interpretations)
An interpretation is a mapping M assigning a classical Herbrand structure (or $\top$) to every path. This mapping is subject to the following restrictions, for all states D_i,E_j and every formula φ, every external atom a and every state formula ψ:

1. $\varphi \in M(\langle(D,E)\rangle)$ iff $\mathcal{O}^d(D) \models \varphi$ for any external state E
2. $\varphi \in M(\langle(D_1,E),^{\varphi}(D_2,E)\rangle)$ iff $\mathcal{O}^t(D_1,D_2) \models \varphi$ for any external state E
3. $\varphi \in M(\langle(D,E_1),^{\varphi}(D,E_2)\rangle)$ iff $\mathcal{O}^e(E_1,E_2) \models \varphi$ for any internal state D
4. $\mathbf{extA}(a[\psi]) \in M(\langle(D,E_1),^{\mathbf{ext}(a[\psi],a_0^{-1}\otimes\ldots\otimes a_{m-1}^{-1})}(D,E_2)\rangle)$ iff $\mathcal{O}^e(E_1,E_2) \models \mathbf{ext}(a[\psi], a_0^{-1} \otimes \ldots \otimes a_{m-1}^{-1})$ for any internal state D

Note that, with automatic repair plans, an external action only appears in the program in the form $\mathbf{extA}(a[\phi])$. With this previous definition, it is the interpretation's responsibility to ask the oracle to instantiate it with the correct repair plan $a_0^{-1} \otimes \ldots \otimes a_{m-1}^{-1}$.

5.3 Properties of repair plans

In this section, we present a series of properties based on the notions of reverse action, reverse plan and conditional reversals of [13]. We start by making precise the relation between the concepts presented here, and the definitions from [13]. Specifically, we state that if a goal reverse plan is not considered, then $a_0^{-1} \otimes \ldots \otimes a_{m-1}^{-1}$ is a valid repair plan iff it is a $\phi;\psi$-conditional plan in [13] where the ψ (respectively ϕ) represents the state formula of the initial (resp. final) state s_1 (resp. s_2).

THEOREM 1 (Relation with [13])
Let F_1 and F_2 be formulas that, respectively, represent completely the states s_1 and s_2, and A_p an action language program.

$$\mathcal{O}^e((A_p,s_1),(A_p,s_2)) \models \mathbf{ext}(a[\bot], a_0^{-1} \otimes \ldots \otimes a_{m-1}^{-1}) \text{ iff}$$
$$a_0^{-1} \otimes \ldots \otimes a_{m-1}^{-1} \text{ is a } F_2;F_1\text{-reverse plan for } a$$

PROOF. $\mathcal{O}^e((A_p,s_1),(A_p,s_2)) \models \mathbf{ext}(a[\bot], a_0^{-1} \otimes \ldots \otimes a_{m-1}^{-1})$ iff item 4a of Definition 25 holds, i.e. if $\langle s_1,a,s_2\rangle \in R \wedge \texttt{revPlan}_s(s_1,s_2;[a_0^{-1} \otimes \ldots \otimes a_{m-1}^{-1}])$. Thus, this means that $\exists s_f$ s.t. $s_f \in \texttt{traj}(s_2;[a_0 \otimes \ldots \otimes a_{m-1}])$ and $\forall s_f$ s.t. $s_f \in \texttt{traj}(s_2;[a_0 \otimes \ldots \otimes a_{m-1}])$ then $s_f = s_1$. Since the latter holds, and F_1 and F_2 represent completely the states s_1 and s_2, then by [13] $a_0^{-1} \otimes \ldots \otimes a_{m-1}^{-1}$ is said to be a $F_2;F_1$-reverse plan for a by Definition 20. ■

We can apply the result on the sufficient condition for the existence of repairs plans from [13] which is based on the notion of involutory actions. An action is said to be involutory if executing the action twice from any state where the action is executable, always results in the starting state, i.e. iff for every s_1,s_2 s.t. $s_2 \in \texttt{traj}(s_1;[a \otimes a])$ then $s_2 = s_1$. An example of an involutory action is a toggle action, as toggling a simple switch twice will always lead the system into the initial state.

LEMMA 1
Let a be an involutory action, and A_p an action language program. For every pair of states s_1,s_2 s.t. $\langle s_1,a,s_2\rangle \in R$ it holds that

$$\mathcal{O}^e((A_p,s_1),(A_p,s_2)) \models \mathbf{ext}(a[\phi],a)$$

for every state formula ϕ.

PROOF. Since a is an involutory action, it holds that $\forall s_1,s_2$, if $s_2 \in \texttt{traj}(s_1;[a])$ then $s_1 \in \texttt{traj}(s_2;[a])$. Thus, $\texttt{revPlan}(s_1,s_2;[a])$ and by case 0a of Definition 25 we know that $\mathcal{O}^e((A_p,s_1),(A_p,s_2)) \models \mathbf{ext}(a[\phi],a)$ ■

Furthermore, we can talk about safety of repairs w.r.t. programs. We say that a program is *repair safe* iff all its external actions have a repair that is guaranteed to succeed.

THEOREM 2 (Repair safety)
Let P be a $\mathcal{ETR}$ program without user-defined repair plans of the form $\mathbf{ext}(a,b_1 \otimes \ldots \otimes b_j)$. If for every $\mathbf{extA}(a[\phi])$ defined in P there is a reverse plan $a_1 \otimes \ldots \otimes a_k$ s.t. $\texttt{revPlan}(a,[a_1 \otimes \ldots \otimes a_k])$ then P is a repair safe program.

PROOF. If P does not have user-defined repair plans, then it means that all repairs are computed using item 25 of Definition 25 and thus this is catered by the `revPlan` which require that the plan can applicable (i.e. the existence of the trajectory) but also that it necessary reaches the intended state. Since all the possible compensations are defined using `revPlan` definition, we never reach a state where the previous calculated compensated is not applicable, and thus the program is safe. ■

Note that, although the conditions for a repair safe program are considerably strong, they allow us to reason about the safeness of a program *before* execution. Obviously, we do not want to restrict only to repair safe programs. However, if an agent is defined by a repair safe program, we know that, whatever happens, the agent will always leave the external world in a consistent state.

We can also define a safe property regarding a particular execution of a transaction.

THEOREM 3 (Repair safe execution)
Let P be a program without user-defined repairs, ϕ be a formula, π be a path, M an interpretation where $M \models P$, and $\mathcal{O}^e$ an external oracle without Item 4b.

If $M, \pi \models_p \phi$, $M, \pi \not\models_c \phi$ and $\texttt{Seq}(\pi) \neq \emptyset$ then $\exists \pi_0, \pi_r$ where π_0 is a rollback path of π, and π_r is a recovery path of π_0 s.t. $\pi' = \pi_0 \circ \pi_r$ and $M, \pi' \rightsquigarrow \phi$

PROOF. This follows from the arguments that prove Theorem 2. If no user-defined repairs exist, then all repairs are computed using item 4a of Definition 25. Thus, definitions of `revPlan` ensure us that all repairs can be applicable and achieve the intended state. Since every execution of a repair is always applicable, then a recovery path exists and so does $M, \pi' \rightsquigarrow \phi$. ■

This result talks about the existence of compensating paths for a given transaction ϕ being executed in a path π. Intuitively, if P does not contain user-defined transactions, the oracle only computes reverse plans, and if π is an execution of ϕ that fails (i.e. $M, \pi \models_p \phi$ but $M, \pi \not\models_c \phi$) after executing some external actions (i.e. if $\texttt{Seq}(\pi) \neq \emptyset$), then there always exists a path where the execution of ϕ is repaired, i.e. there is a path π' where $M, \pi' \rightsquigarrow \phi$ holds.

Note that these theorems only provide guarantees for programs where explicit user-defined repairs and goal reverses are not presented. The problem with user-defined repairs is that it is impossible to predict, before execution, what will be the resulting state of the external world after their execution, or to guarantee any properties about this resulting state. As such, it may be the case that the existence of user-defined repairs jeopardizes the applicability of automatic repair plans. This is as expected: since the user may arbitrarily change the repair of some actions, it may certainly be the case that the specification of the external domain cannot infer any repair plan for other actions in the same sequence. To prevent this, we could exclude user-defined repairs. However, this would make $\mathcal{ETR}$ less expressible, making it impossible to use whenever the agent does not possess enough information about the external world.

Similar to user-defined repairs, the success of goal reverses depend on a state formula ϕ which is specified by the user. As such, in general, without restricting the set of state formulas that can be written, nothing guarantees that in the achieved state the previous reverse plans can be applicable. However, as previously stated, reasoning about state formulas is a hard problem and is out of scope of this work.

6 Related work

Several languages to describe an agent's behaviour partitioned over an internal and external KB have been proposed in the literature. In this context, AgentSpeak/Jason [5], 2APL [10] and 3APL [21] are successful examples of agent programming languages that deal with environments with both internal updates and external actions. Additionally, all these languages have a way to deal with action failures, and define repairs of some form to be executed in case of failure.

AgentSpeak, became a popular multi-agent programming language mainly due to the development of its Jason interpreter. The language identifies the importance of handling plan failures given the intrinsic unpredictable characteristics of dynamic environments. Thus, for that end, it defines a form

of contingency plan to be executed upon a plan execution failure that 'cleans up' the effects of the previous executed actions, before attempting another plan.

Similarly, 2APL and 3APL programming languages rely on the notion of plans to achieve the agent's goals. To deal with the possibility of failure, they also include the notion of plan repair rules which specify how a plan can be repaired upon failure. Similarly to $\mathcal{ETR}$, their semantics first computes the prefix of the plan that caused the failure, matches this prefix to the head of a plan repair rule, and creates a new plan containing the repair plus the postfix of the plan that was not executed because of the failure. The notion of action failure is also very similar to $\mathcal{ETR}$, and when such a failure happens, then the execution of the whole plan is blocked to be compensated/repaired. Nevertheless, repairs need to be explicitly stated by the programmer, and if such a repair does not exist, then nothing will be done, and the failed plan will be tried again, possibly causing some non-termination issues. On the other hand, in $\mathcal{ETR}$ it is possible to automatically infer what should be the repairs of a given action, if there is enough information available about the external world.

It is worth mentioning that in 2APL it is possible to define plans that should not be executed concurrently, and by using repair plans, some relaxed way of atomicity could still be achieved. However, as stated, this ability to recover depends heavily on the programmer, even when the repairs are only in the internal knowledge (i.e. the beliefs of the agent).

Moreover, these languages have an operational semantics that allows one to say what should be the next step of computation if the system is in a particular state, and prove some properties about the agent's behaviour, e.g. about termination of agents execution. However, they do not have a model-theoretic semantics like $\mathcal{ETR}$, allowing us to reason about what formulas (e.g. agents beliefs) are true during the execution.

Self-checking agents [6–8] can do this kind of reasoning by enabling expressing properties (or meta-constraints), by means of a linear temporal logic, defining what constraints should hold during the execution of the agent. Then, in case of a property violation, the agent can try to execute a self-repair to restore an acceptable or desired state of affairs. These properties are stated along with applicability conditions, in reactive-like terms, defining when is the property applicable (the pre-condition), when should the property be abandoned (abandon-condition) and an optional repair to be tried upon property violation.

In $\mathcal{ETR}$, one can also define constraints over the execution of (trans)actions, in several different ways. One possible way to do it is as:

$$p \leftarrow a \wedge const$$

where the constraint *const* should always be true during all steps of execution of transaction p (and where a can be a complex action formula). A violation of *const* during the execution of p, always forces the system to roll back internally and to execute compensations externally to achieve a consistent state. Similarly, we can apply such constraints to be verified before or after some actions. For example, $p \leftarrow const \otimes a$ and $p \leftarrow a \otimes const$, respectively, define the verification of constraint *const* before and after the (possibly complex) action a.

Additionally, pre-conditions and abandon-conditions can also be expressed within the formula *const*, as:

$$const \leftarrow (precondition \otimes constraint) \wedge \neg abandon\_condition$$
$$const \leftarrow \neg precondition$$

which impose the *constraint* to be true unless the precondition is not applicable or the fail-condition holds eventually during the execution. Furthermore, note that such an imposition of constraints as well as the definition of the conditions, regarding individual (internal or external) actions, can also be done directly in the specification of the oracles. Clearly, the meta-constraints defined in [6–8] are

higher level and in some sense, easier to maintain, but they are also less expressible. Moreover, as it happens with the previous solutions, the agent can only attempt to make repairs if they are explicitly stated in the meta-constraint statement, and there is no guarantee that the repair specified by the programmer is correct, as in $\mathcal{ETR}$.

GOLOG [24] is a high-level agent programming language, in which it is possible to prove correctness of a program execution in achieving an agent's goal. An execution of a GOLOG program combines an online planning based on the Situation Calculus, and imperative procedures. Additionally, ACID-like transactions can be achieved in GOLOG [14, 22] by means of non-Markovian theories. However, ensuring transactional properties is not native of the model theory as in $\mathcal{ETR}$, and requires the explicit formulation of all the intended properties (atomicity, rollback, etc.) in second-order logic formulas. In opposition, $\mathcal{ETR}$'s model theory is natively designed to find the paths where the agent can execute correctly, in a transactional way. Also, in GOLOG the meaning of elementary actions is specified in the situation calculus, while $\mathcal{ETR}$'s states and primitives are flexible allowing different semantics to represent the agent's internal knowledge, and the external dynamics of the external world. In fact, while in this article we define the meaning of elementary external actions in Action Language $\mathcal{C}$, nothing precludes us to use a different semantics for representing the dynamics of the external world, like the situation calculus.

Other agent's logic programming-based languages exist (e.g. [9, 23]) but, to our knowledge, none of them deals with transactions nor with repair plans. The closest might be [23], where the authors mention as future work the definition of transactions. However, the model theory of [23] does not consider the possibility of failure, and thus neither the possibility of repairing plans. In contrast, its operational semantics may react to external events defining failure of actions performed externally, but since no tools are provided to model the external environment, the decision about what to do with the failure is based only on internal knowledge (but which has information about external events). Moreover, since there is no strict distinction between actions performed externally and internally, it not clear how the semantics would deal with the different levels of atomicity that the combination between internal and external actions demands.

7 Conclusions and future work

In this article, we have proposed an extension of the $\mathcal{ETR}$ language to deal with external environments described by an action language, and which automatically infers repair plans to be executed when some failure occurs. The obtained language is able to reason and act in a two-fold environment in a transactional way. In an $\mathcal{ETR}$ execution, internal updates are guaranteed to follow the standard transaction ACID model, while consistency of external changes is achieved by executing compensations that repair previously executed actions.

By defining a semantics that automatically infers what should be the correct repairs when something fails in the external world, we ease the programmer's task to anticipate for all the possible failures and write the corresponding repair for it. As such, when enough information is available regarding the external world, $\mathcal{ETR}$ can be used to automatically infer plans to deal with failures, which are guarantee to correctly repair the agent's behaviour. Nevertheless, it is worth noting that this can only be done when such information about the external environment exists. Thus, in contrast, when the agent has no knowledge about the external environment on which he/she performs actions, repair plans can be defined explicitly in the agent's program. Moreover, we have devised a proof procedure for $\mathcal{ETR}$ [19], which readily provides a means for an implementation that is underway.

Additionally, although our solution only deals with serial executions, we also plan to extend $\mathcal{ETR}$ with concurrent capabilities, in a similar way to what was done in Concurrent Transaction Logic [4] for Transaction Logic.

For dealing with the inference of repair plans, we assume an environment described using the action language $\mathcal{C}$, and based the representation of reversals on the work of [13]. An alternative would be to choose another language for representing changes in the external environment like [26]. [26] defines an action language to represent and reason about commitments in multi-agent domains. In it, it is possible to directly encode in the language which actions are reversible and how. Using this language to represent the external world in $\mathcal{ETR}$ could also be done by changing the external oracle definition, similarly to what we have done here. However, we chose the reversals representation from [13] because its generality makes it applicable to a wider family of action languages, like, e.g. the action language $\mathcal{C}$. Since this latter language has several extensions that are already used for norms and protocol representation in multi-agent systems [1, 25], by defining an external oracle using action language $\mathcal{C}$ we provide means to employ such representations together with $\mathcal{ETR}$, extending them with the possibility to describe an agent's behaviour in a transactional way. Furthermore, our version of goal reverse plans can be seen as a contribution to the work of [13]'s as it provides means to relax the conditions for the existence of plans, increasing the possibility of achieving a state with some desirably consistent properties.

Acknowledgements

A.S.G. was supported by FCT grant SFRH/BD/64038/2009. The work was partially supported by project ERRO (PTDC/EIA-CCO/121823/2010).

References

[1] A. Artikis, M. J. Sergot and J. V. Pitt. Specifying norm-governed computational societies. *ACM Transactions on Computational Logic*, **10**, 1–41, 2009.

[2] A. J. Bonner and M. Kifer. Transaction logic programming. In *International Conference in Logic Programming*, pp. 257–279, 1993.

[3] A. J. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-323, University of Toronto, 1995.

[4] A. J. Bonner and M. Kifer. Concurrency and communication in transaction logic. In M. J. Maher, editor, *Joint International Conference and Symposium on Logic Programming*, pp. 142–156. MIT Press, 1996.

[5] R. H. Bordini, M. Wooldridge and J. F. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[6] S. Costantini. Self-checking logical agents. In M. Osorio, C. Zepeda, I. Olmos, J. L. Carballido and R. C. M. Ramírez, editors, *Logic / Languages, Algorithms and New Methods of Reasoning*, vol. 911 of *CEUR Workshop Proceedings*, pp. 3–30. CEUR-WS.org, 2012.

[7] S. Costantini. Self-checking logical agents. In M. L. Gini, O. Shehory, T. Ito and C. M. Jonker, editors, *International Conference on Autonomous Agents and Multiagent Systems*, pp. 1329–1330. IFAAMAS, 2013.

[8] S. Costantini and G. D. Gasperis. Meta-level constraints for complex event processing in logical agents. In *Informal Proceedings of Commonsense 2013, 11th International Symposium on Logical Formalizations of Commonsense Reasoning*, 2013.

[9] S. Costantini and A. Tocchio. About declarative semantics of logic-based agent languages. In M. Baldoni, U. Endriss, A. Omicini and P. Torroni, editors, *Declarative Agent Languages and Technologies*, vol. 3904 of *Lecture Notes in Computer Science*, pp. 106–123. Springer, 2005.
[10] M. Dastani. 2apl: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, **16**, 214–248, 2008.
[11] M. Dastani, J.-J. C. Meyer and D. Grossi. A logic for normative multi-agent programs. *Journal of Logic and Computation*, **23**, 335–354, 2013.
[12] M. Dastani, B. van Riemsdijk, F. Dignum and J.-J. C. Meyer. A programming language for cognitive agents goal directed 3apl. In M. Dastani, J. Dix and A. E. Fallah-Seghrouchni, editors, *Programming Multi-Agent Systems*, vol. 3067 of *Lecture Notes in Computer Science*, pp. 111–130. Springer, 2003.
[13] T. Eiter, E. Erdem and W. Faber. Undoing the effects of action sequences. *Journal of Applied Logic*, **6**, 380–415, 2008.
[14] A. Gabaldon. Non-Markovian control in the situation calculus. *Artificial Intelligence*, **175**, 25–48, 2011.
[15] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions in Artificial Intelligence*, **2**, 193–210, 1998.
[16] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, **153**, 49–104, 2004.
[17] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: preliminary report. In J. Mostow and C. Rich, editors, *Association for the Advancement of Artificial Intelligence / Innovative Applications of Artificial Intelligence Conference*, pp. 623–630. AAAI Press/The MIT Press, 1998.
[18] A. S. Gomes and J. J. Alferes. Transaction logic with external actions. In *Logic Programming and Nonmonotonic reasoning*, pp. 272–277, 2011.
[19] A. S. Gomes and J. J. Alferes. Extending transaction logic with external actions. *Theory and Practice in Logic Programming*, 13(4-5 Online Supplement), 2013.
[20] A. S. Gomes and J. J. Alferes. External transaction logic with automatic compensations. In J. Leite, T. C. Son, P. Torroni, L. van der Torre and S. Woltran, editors, *Computational Logic in Multi-Agent Systems*, vol. 8143 of *Lecture Notes in Computer Science*, pp. 239–255. Springer, 2013.
[21] K. V. Hindriks, F. S. de Boer, W. van der Hoek and J.-J. C. Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, **2**, 357–401, 1999.
[22] I. Kiringa. Simulation of advanced transaction models using golog. In G. Ghelli and G. Grahne, editors, *Database Programming Languages*, vol. 2397 of *Lecture Notes in Computer Science*, pp. 318–341. Springer, 2001.
[23] R. A. Kowalski and F. Sadri. Abductive logic programming agents with destructive databases. *Annals Mathematics and Artifical Intelligence*, **62**, 129–158, 2011.
[24] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin and R. B. Scherl. Golog: a logic programming language for dynamic domains. *Journal of Logic Programming*, **31**, 59–83, 1997.
[25] M. J. Sergot and R. Craven. The deontic component of action language nc+. In L. Goble and J.-J. C. Meyer, editors, *International Conference on Deontic Logic and Normative Systems*, vol. 4048 of *Lecture Notes in Computer Science*, pp. 222–237. Springer, 2006.
[26] T. C. Son, E. Pontelli and C. Sakama. Formalizing commitments using action languages. In C. Sakama, S. Sardiña, W. Vasconcelos and M. Winikoff, editors, *Declarative Agent Languages and Technologies*, vol. 7169 of *Lecture Notes in Computer Science*, pp. 67–83. Springer, 2011.

Received 4 September 2014