Verification of Golog Programs over Description Logic Actions

Franz Baader and Benjamin Zarrieß[*]
{baader,zarriess}@tcs.inf.tu-dresden.de

Theoretical Computer Science, TU Dresden, Germany

Abstract. High-level action programming languages such as Golog have successfully been used to model the behavior of autonomous agents. In addition to a logic-based action formalism for describing the environment and the effects of basic actions, they enable the construction of complex actions using typical programming language constructs. To ensure that the execution of such complex actions leads to the desired behavior of the agent, one needs to specify the required properties in a formal way, and then verify that these requirements are met by any execution of the program. Due to the expressiveness of the action formalism underlying Golog (Situation Calculus), the verification problem for Golog programs is in general undecidable. Action formalisms based on Description Logic (DL) try to achieve decidability of inference problems such as the projection problem by restricting the expressiveness of the underlying base logic. However, until now these formalisms have not been used within Golog programs. In the present paper, we introduce a variant of Golog where basic actions are defined using such a DL-based formalism, and show that the verification problem for such programs is decidable. This improves on our previous work on verifying properties of infinite sequences of DL actions in that it considers (finite and infinite) sequences of DL actions that correspond to (terminating and non-terminating) runs of a Golog program rather than just infinite sequences accepted by a Büchi automaton abstracting the program.

1 Introduction

Action programming languages like Golog [8,11] can be used to control the behavior of autonomous agents and mobile robots. In this setting, the programming language provides the user with typical programming language constructs such as loops and tests. These constructs can be used to build complex actions from atomic ones. The semantics of the atomic actions and of the programming language constructs is formally specified using an appropriate logical calculus (Situation Calculus in the case of Golog). To ensure that a complex action specified in this way actually shows the desired behavior, one needs to specify the required properties in a formal way, and then verify that these requirements are met by any run of the program defining this action. In principle, this verification

[*] Supported by DFG Research Unit FOR 1513, project A1

problem boils down to a deduction problem in the underlying logical calculus, but due to the expressiveness of the situation calculus, the deduction problem is in general undecidable. For instance, the first work that aims at the fully automated verification of (non-terminating) Golog programs [7] specifies properties in an extension of the situation calculus by constructs of the temporal logic CTL*. Similar to CTL* model checking, the approach tries to compute an appropriate fixpoint, but in contrast to the case of model checking the fixpoint iteration need not terminate. If it does terminates, then the proof that the desired property holds is reduced to a deduction problem in the underlying logic (i.e., situation calculus), which is in general not decidable.

In order to overcome the problems caused by an undecidable base logic, action theories based on decidable Description Logics [2] have been proposed in the literature [1,10,4]. The decidability and complexity results obtained in these papers were mainly concerned with the projection problem: given a finite sequence of atomic actions and a (possibly incomplete) description of the initial world, decide whether a certain property is guaranteed to hold after the execution of this sequence. The papers differ w.r.t. what kind of domain constraints (i.e., constraints that are guaranteed to hold in every world) can be used. Whereas [1] restricts the attention to acyclic TBoxes, the other two papers allow for general TBoxes, i.e., finite sets of general concept inclusions (GCIs). In the presence of GCIs, one has to deal with the so-called ramification problem, i.e., the fact that the direct effects of an action may violate the domain constraints, and thus also indirect effects need to be considered. Whereas the authors of [10] deal with this problem by introducing so-called occlusions, the approach described in [4] uses so-called causal relationships to specify indirect effects of actions.

The first attempt to extend the decidability results for projection in [1] to the verification problem can be found in [5]. However, instead of examining the actual execution sequences of a given Golog program, this approach considers infinite sequences of actions that are accepted by a given Büchi automaton $\mathcal{B}$. If $\mathcal{B}$ is an upper approximation of the program, i.e. all possible execution sequences of the program are accepted by $\mathcal{B}$, then any property that holds in all the sequences accepted by $\mathcal{B}$ is also a property that is satisfied by any execution of the program. As logic for specifying properties of infinite sequences of DL actions, the approach uses the temporalized DL $\mathcal{ALC}$-LTL [3], which extends the well-known propositional linear temporal logic (LTL) [12] by allowing for the use of axioms (i.e., TBox and ABox statements) of the basic DL $\mathcal{ALC}$ in place of propositional letters.[1] Recently, other restrictions on action theories based on the situation calculus that guarantee decidability of the verification problem were considered [9]. However, instead of considering execution sequences of programs, this work looks at all possible infinite sequences of actions. For a more detailed discussion of related work see [6].

In the present paper, we improve on the results in [5] in several respects. First, instead of using Büchi automata to approximate programs, we directly consider

[1] More precisely, [5] uses the extension of $\mathcal{ALC}$-LTL to the more expressive DL $\mathcal{ALCO}$, but disallows TBox statements.

Golog programs. Second, we deal with terminating and non-terminating runs of programs in a uniform way. Finally, our approach works not only for the action formalism introduced in [1], but also for the one considered in [4]. Regarding the underlying DL, our result applies to all DLs considered in [1], but for the sake of simplicity we restrict the attention to the DL $\mathcal{ALCO}$.

In the next section, we introduce the relevant notions concerning DL and action languages based on DLs. Then we define syntax and semantics of Golog programs over DL actions and prove some auxiliary results for such programs. In the subsequent section, we define the verification problem and show that it is decidable. Because of space constraints, detailed proofs of our results have to be omitted. They can be found in [6].

2 Preliminaries

Description Logics The DL $\mathcal{ALCO}$ extends the basic DL $\mathcal{ALC}$ by nominals, i.e., singleton concepts. Starting with sets N_C of *concept names*, N_R of *role names*, and N_I of *individual names*, $\mathcal{ALCO}$-concept descriptions (*concepts* for short) are built from concept names using the *constructors* shown in Table 1.

DL	Name	Syntax	Semantics
$\mathcal{ALC}$	top-concept	$\top$	$\Delta^\mathcal{I}$
	negation	$\neg C$	$\Delta^\mathcal{I} \setminus C^\mathcal{I}$
	conjunction	$C \sqcap D$	$C^\mathcal{I} \cap D^\mathcal{I}$
	disjunction	$C \sqcup D$	$C^\mathcal{I} \cup D^\mathcal{I}$
	existential restriction	$\exists r.C$	$\{x \mid \exists y : (x,y) \in r^\mathcal{I} \wedge y \in C^\mathcal{I}\}$
	value restriction	$\forall r.C$	$\{x \mid \forall y : (x,y) \in r^\mathcal{I} \rightarrow y \in C^\mathcal{I}\}$
$\mathcal{O}$	nominal	$\{a\}$	$\{a^\mathcal{I}\}$

Table 1. Syntax and semantics of $\mathcal{ALCO}$

In the following, we often use A, B to denote concept names, r, s for role names, a, b for individual names (*individuals* for short), and C, D for possibly complex concepts.

An *ABox* is a finite set of *concept assertions* $C(a)$ and positive and negated *role assertions* of the form $r(a,b)$ and $\neg r(a,b)$, respectively. Assertions of the form $A(a), \neg A(a), r(a,b), \neg r(a,b)$ for concept names A and role names r are called *literals*.

A *concept definition* is of the form $A \equiv C$ and a *general concept inclusion* (GCI) is of the form $C \sqsubseteq D$. An *acyclic TBox* is a finite set of concept definitions with unique left-hand sides. Additionally, it is required that there are no cyclic dependencies between the definitions. A *general* TBox is a finite set of GCIs.

The semantics of concepts is defined in terms of interpretations. An *interpretation* $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ consists of a non-empty domain $\Delta^\mathcal{I}$ and a mapping $\cdot^\mathcal{I}$, which maps each concept name A to a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, each role name r to a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and each individual a to an element $a^{I} \in \Delta^{I}$. We assume that $a^{\mathcal{I}} \neq b^{\mathcal{I}}$ for any two distinct individuals a, b (*unique name assumption*). The extension of $\cdot^{\mathcal{I}}$ to complex concepts is defined inductively as shown in Table 1.

An interpretation $\mathcal{I}$ satisfies an ABox assertion $C(a)$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$, $r(a,b)$ if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$, and $\neg r(a,b)$ if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \notin r^{\mathcal{I}}$. It is a *model* of an ABox $\mathcal{A}$ (written as $\mathcal{I} \models \mathcal{A}$) if $\mathcal{I}$ satisfies all assertions in $\mathcal{A}$. An interpretation $\mathcal{I}$ satisfies a concept definition $A \equiv C$ if $A^{\mathcal{I}} = C^{\mathcal{I}}$ and a GCI $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. It is a model of a TBox $\mathcal{T}$ (written $\mathcal{I} \models \mathcal{T}$) if it satisfies each definition or GCI, respectively, in $\mathcal{T}$. The ABox $\mathcal{A}$ is *consistent* w.r.t. the TBox $\mathcal{T}$ if there exists a model of $\mathcal{A}$ that is also a model of $\mathcal{T}$. The set of models of $\mathcal{A}$ and $\mathcal{T}$ is denoted by $\mathcal{M}(\mathcal{A})$ and $\mathcal{M}(\mathcal{T})$, respectively. The assertion φ is *entailed* by $\mathcal{A}$ and $\mathcal{T}$ (written as $\mathcal{T}, \mathcal{A} \models \varphi$) if every model of $\mathcal{A}$ and $\mathcal{T}$ (i.e., element of $\mathcal{M}(\mathcal{A}) \cap \mathcal{M}(\mathcal{T})$) satisfies φ.

For $\mathcal{ALCO}$, the consistency and the entailment problem are PSpace-complete w.r.t. an acyclic TBox and ExpTime-complete w.r.t. a general TBox [2].

DL-based Action Formalism Instead of introducing a specific DL-based action formalism, we take a more abstract point of view and describe a whole class of DL-based action formalisms. This class has the formalisms introduced in [1] (without occlusions) and in [4] as instances, but not the one of [10] (since there occlusions are a key ingredient of the formalism). Basically, we abstract from the concrete way the formalism determines the effect of applying an action to a world, and assume that there is an appropriate function that provides us with the effect. Later on, we need to impose additional restrictions in order to obtain our decidability results.

A *DL-based action theory* consists of the following components:

- the *domain constraints*, given as a TBox $\mathcal{T}$;
- an incomplete description of the *initial world* given by an ABox $\mathcal{A}$;
- a finite set Σ of *action names*;
- a finite set of *relevant ABox assertions*, denoted by $\mathcal{D}$.

In the following we use the (possibly indexed) letters α, β to denote action names and φ to denote ABox assertions (or assertions for short). The set of literals contained in $\mathcal{D}$ is denoted by *Lit*. We require that the assertions contained in the description of the initial world are also contained in $\mathcal{D}$, and that $\mathcal{D}$ is closed under negation (modulo elimination of double negation). For the formalism in [1], the elements of $\mathcal{D}$ are the assertions occurring in the initial ABox and in the pre- and post-conditions of action descriptions.

Instead of introducing the syntactic form of action descriptions and then defining the effects of actions by providing these descriptions with an appropriate semantics, we define the semantics of action names directly using an effect function. The literals in $\mathcal{D}$ are used to specify how an action changes the actual world. An interpretation $\mathcal{I}$ completely describes the current state of the world.

The semantics of actions is thus defined by specifying how they transform a given interpretation into a successor interpretation. First, we have to determine whether an action α is *applicable* to an interpretation $\mathcal{I}$. Then, if α is applicable to $\mathcal{I}$, we define the *effects* of α on $\mathcal{I}$ as a set of literals.

Definition 1. *Let Σ be the set of action names, $\mathcal{D}$ the set of relevant assertions with $Lit \subseteq \mathcal{D}$ the set of literals occurring in $\mathcal{D}$, and $\mathcal{T}$ the TBox specifying the domain constraints. An* effect function *$\mathcal{E}$ w.r.t. Σ, $\mathcal{D}$, and $\mathcal{T}$ is a* partial *function $\mathcal{E} : \Sigma \times \mathcal{M}(\mathcal{T}) \to 2^{Lit}$. If $\mathcal{E}$ is defined for a pair $(\alpha, \mathcal{I}) \in \Sigma \times \mathcal{M}(\mathcal{T})$, then we say that α is* applicable *to $\mathcal{I}$. Otherwise, α is* not applicable *to $\mathcal{I}$.*

For the action formalism in [1], $\mathcal{E}(\alpha, \mathcal{I})$ consists of the literals of the conditional post-conditions whose condition is satisfied by $\mathcal{I}$. For the action formalism in [10], in addition to such direct effects, the set $\mathcal{E}(\alpha, \mathcal{I})$ also contains indirect effects generated by the causal relationships.

For every $\alpha \in \Sigma$, the effect function induces a binary relation $\Longrightarrow_{\alpha}^{\mathcal{E}}$ on $\mathcal{M}(\mathcal{T})$:

Definition 2. *Let $\mathcal{E}$ be an effect function w.r.t. Σ, $\mathcal{D}$, and $\mathcal{T}$. Then $\mathcal{I} \Longrightarrow_{\alpha}^{\mathcal{E}} \mathcal{I}'$ if the following conditions are satisfied:*

1. *α is applicable to $\mathcal{I}$,*
2. *there exists no $L \in Lit$ such that $\{L, \neg L\} \subseteq \mathcal{E}(\alpha, \mathcal{I})$,*
3. *$\Delta^{\mathcal{I}} = \Delta^{\mathcal{I}'}$ and $a^{\mathcal{I}} = a^{\mathcal{I}'}$ for all $a \in N_I$,*
4. *$A^{\mathcal{I}'} := (A^{\mathcal{I}} \cup \{a^{\mathcal{I}} \mid A(a) \in \mathcal{E}(\alpha, \mathcal{I})\}) \setminus \{a^{\mathcal{I}} \mid \neg A(a) \in \mathcal{E}(\alpha, \mathcal{I})\}$ for all $A \in N_C$,*
5. *$r^{\mathcal{I}'} := (r^{\mathcal{I}} \cup \{(a^{\mathcal{I}}, b^{\mathcal{I}}) \mid r(a, b) \in \mathcal{E}(\alpha, \mathcal{I})\}) \setminus \{(a^{\mathcal{I}}, b^{\mathcal{I}}) \mid \neg r(a, b) \in \mathcal{E}(\alpha, \mathcal{I})\}$ for all $r \in N_R$.*

If $\mathcal{I} \Longrightarrow_{\alpha}^{\mathcal{E}} \mathcal{I}'$ then we say: α transforms *the model $\mathcal{I}$ into the model $\mathcal{I}'$.*

Note that, for a given action $\alpha \in \Sigma$ and a model $\mathcal{I} \in \mathcal{M}(\mathcal{T})$, there is at most one $\mathcal{I}'$ such that $\mathcal{I} \Longrightarrow_{\alpha}^{\mathcal{E}} \mathcal{I}'$, i.e., the actions that we consider are *deterministic*.[2] There are several possible reasons why such an $\mathcal{I}'$ may not exist at all. First, the action may not be applicable to $\mathcal{I}$. In the formalism of [1], this corresponds to the fact that $\mathcal{I}$ does not satisfy the preconditions of the action. Second, the action may be applicable, but Condition 2 may not be satisfied. In [1], the action is then called inconsistent w.r.t. $\mathcal{I}$. Finally, it may be the case that Conditions 1 and 2 are satisfied, but the interpretation $\mathcal{I}'$ defined by the last three conditions is not a model of $\mathcal{T}$. In [10], actions for which this case occurs are also considered to be inconsistent.

Definition 3. *An action α is called* consistent *if, for every $\mathcal{I} \in \mathcal{M}(\mathcal{T})$ to which α is applicable, there is an $\mathcal{I}' \in \mathcal{M}(\mathcal{T})$ such that $\mathcal{I} \Longrightarrow_{\alpha}^{\mathcal{E}} \mathcal{I}'$.*

In the following, we will only consider action theories for which all actions are consistent.

[2] However, the complex actions that we will build from these deterministic atomic actions in Section 3 can well be non-deterministic due to the non-deterministic nature of the programming constructs used in Golog programs.

3 Golog Programs over DL Actions

In this section we define the syntax and semantics of a fragment of the action programming language Golog [8,11] that uses DL-based action theories as introduced in the previous section. Golog program expressions describe how a complex action is constructed from atomic actions using programming constructs and tests.

Definition 4. *Let Σ be a set of action names, $\alpha \in \Sigma$ and ψ a* test *that is built according to the following grammar:*

$$\psi ::= \varphi \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2$$

where φ is an assertion. A program expression *is defined inductively as follows.*

- *A* single action *$\alpha \in \Sigma$ is a program expression.*
- *The* empty program *$\langle\rangle$ is a program expression.*
- *If δ is a program expression, then the* non-deterministic iteration *of δ, denoted by $(\delta)^*$, is a program expression.*
- *If δ_1 and δ_2 are program expressions, then the* sequence *of δ_1 and δ_2, denoted by $(\delta_1; \delta_2)$, is a program expression.*
- *If ψ is a* test *and δ a program expression, then $\psi?; \delta$ is a program expression.*
- *If δ_1 and δ_2 are program expressions, then the* non-deterministic choice *between δ_1 and δ_2, denoted by $(\delta_1|\delta_2)$, is a program expression.*
- *If δ_1 and δ_2 are program expressions, then the* interleaving *of δ_1 and δ_2, denoted by $(\delta_1\|\delta_2)$, is a program expression.*

A DL-Golog program *$\mathcal{P} = (\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E}, \delta)$ consists of a DL-based action theory $(\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E})$ and a program expression δ such that every action occurring in δ belongs to Σ and every assertion occurring in a test in δ belongs to $\mathcal{D}$.*

The *length* of program expressions, denoted by $|\delta|$, is defined to be the number of symbols (more precisely, the number of action names, tests, and constructors $*$, ;, $|$, $\|$) occurring in δ.

Note that the tests in the program expression can be used to model the control flow of a program whereas preconditions of actions (which in our setting are realized implicitly by the domain of the function $\mathcal{E}$) can be used to ensure that a single action is only applicable if it leads to a successor model. Together with the other constructs, tests can for example be used to express while-loops and if-then-else statements:

$$\begin{aligned} \textbf{while } \psi \textbf{ do } \delta \textbf{ endWhile} &:= (\psi?; \delta)^*; (\neg\psi?; \langle\rangle) \\ \textbf{if } \psi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf} &:= \psi?; \delta_1 \mid \neg\psi?; \delta_2 \end{aligned}$$

Our notion of tests differs from the one in [8] in that tests are not viewed to be actions, and thus a test $\psi?$ alone is not a valid program expressions. In fact, viewing tests as actions creates problems when combined with interleaving since it may happen that a test is successfully executed, but then is followed by an interleaved action from another part of the program, which may in turn destroy the condition checked by the test. For example, consider the program expression obtained by interleaving of the two if-then-else statements **if** $A(a)$ **then** $\alpha_{-B(a)}$ **else** $\langle\rangle$ **endIf** and **if** $B(a)$ **then** $\alpha_{-A(a)}$ **else** $\langle\rangle$ **endIf**:

$$(A(a)?;\alpha_{-B(a)} \mid \neg A(a)?;\langle\rangle) \parallel (B(a)?;\alpha_{-A(a)} \mid \neg B(a)?;\langle\rangle), \qquad (1)$$

where $\alpha_{-B(a)}$ and $\alpha_{-A(a)}$ are actions such that, for all models $\mathcal{I}$, $\mathcal{E}(\alpha_{-A(a)},\mathcal{I}) = \{\neg A(a)\}$ and $\mathcal{E}(\alpha_{-B(a)},\mathcal{I}) = \{\neg B(a)\}$, respectively. Let us also assume that the TBox describing the domain constraints is empty. Starting with a model $\mathcal{I}$ in which a belongs both to A and B, the program (1) should have two possible outcomes, depending on which if-then-else statement is executed first: either a belongs to A and not to B or a belongs to B and not to A. However, if tests are viewed as actions, then one could first successfully execute the test $A(a)?$, then the whole second if-then-else statement, and then the action $\alpha_{-B(a)}$, resulting in the unintended model in which a is contained neither in A nor in B. To avoid such unintended interactions between interleaving and tests, we consider a sequence of tests followed by an action as a unit, called guarded action, which must be executed in one atomic step.

Definition 5. *A* guarded action *is a program expression of the form*

$$\psi_1?;\ldots;\psi_n?;\alpha,$$

where $n \geq 0$, ψ_i for $i = 1,\ldots,n$ are tests, and $\alpha \in \Sigma$.

We will often use the symbol $\mathfrak{a}$ to denote a guarded action.[3].

In order to deal with terminating and non-terminating programs in a uniform way, we introduce an auxiliary action name ϵ, a fresh individual name p, and a fresh concept name *Term* to indicate that the program has terminated. We assume that these symbols do not occur in the input program, with the only exception that $\neg\mathit{Term}(p)$ belongs to the initial ABox. The effect of the action ϵ is predefined such that ϵ is applicable to all models of the domain constraints $\mathcal{T}$, and $\mathcal{E}(\epsilon,\mathcal{I}) = \{Term(p)\}$ holds for all models $\mathcal{I}$ of $\mathcal{T}$.

To define the semantics of DL-Golog programs, we introduce the functions $\mathsf{head}(\cdot)$ and $\mathsf{tail}(\cdot,\cdot)$. Intuitively, $\mathsf{head}(\delta)$ contains those guarded actions that can be executed first when executing the program expression δ. For $\mathfrak{a} \in \mathsf{head}(\delta)$, $\mathsf{tail}(\mathfrak{a},\delta,)$ yields the remainder of the program, i.e., the part that still needs to be executed after $\mathfrak{a}$ has been executed. Due to the non-deterministic nature of Golog programs, $\mathsf{tail}(\mathfrak{a},\delta,)$ is also a set of program expressions rather than a single one.

Definition 6. *The function $\mathsf{head}(\cdot)$ maps a program expression to a set of guarded actions. It is defined by induction on the structure of program expressions:*

$$\mathsf{head}(\langle\rangle) := \{\epsilon\};$$

$$\mathsf{head}(\alpha) := \{\alpha\} \text{ for all } \alpha \in \Sigma;$$

$$\mathsf{head}(\psi?;\delta) := \{\psi?;\mathfrak{a} \mid \mathfrak{a} \in \mathsf{head}(\delta)\};$$

[3] If $n = 0$, then the guarded action is actually an ordinary action, and thus $\mathfrak{a}$ may also denote an ordinary action.

$$\begin{aligned}
\mathit{head}(\delta^*) &:= \{\epsilon\} \cup \mathit{head}(\delta);\\
\mathit{head}(\delta_1;\delta_2) &:= \{\mathfrak{a} \mid \mathfrak{a} = \psi_1?;...;\psi_n?;\alpha \in \mathit{head}(\delta_1) \wedge \alpha \neq \epsilon\} \cup\\
&\qquad \{\psi_1?;\ldots;\psi_n?;\psi_1'?;...;\psi_m'?;\alpha \mid \psi_1?;...;\psi_n?;\epsilon \in \mathit{head}(\delta_1) \wedge\\
&\qquad\qquad \psi_1'?;\ldots;\psi_m'?;\alpha \in \mathit{head}(\delta_2)\};\\
\mathit{head}(\delta_1|\delta_2) &:= \mathit{head}(\delta_1) \cup \mathit{head}(\delta_2);\\
\mathit{head}(\delta_1\|\delta_2) &:= \{\mathfrak{a} \mid \mathfrak{a} = \psi_1?;\ldots;\psi_n?;\alpha \in \mathit{head}(\delta_i) \wedge i \in \{1,2\} \wedge \alpha \neq \epsilon\} \cup\\
&\qquad \{\psi_1?;\ldots;\psi_n?;\psi_1'?;\ldots;\psi_m'?;\alpha \mid \psi_1?;\ldots;\psi_n?;\epsilon \in \mathit{head}(\delta_i) \wedge\\
&\qquad\qquad \psi_1'?;\ldots;\psi_m'?;\alpha \in \mathit{head}(\delta_j) \wedge\\
&\qquad\qquad \{1,2\} = \{i,j\}\}.
\end{aligned}$$

Consider the definition of $\mathsf{head}(\delta_1;\delta_2)$. In this case, we first have to execute the program δ_1. Therefore, the first guarded action to be executed for the sequence is one of the heads of δ_1. However, if $\psi_1?;\ldots;\psi_n?;\epsilon$ is contained in the head of δ_1, then δ_1 can terminate successfully if the tests are satisfied. But in this case the subsequent program δ_2 still needs to be executed. Therefore, we must continue with a head of δ_2. This is achieved by replacing ϵ in $\psi_1?;\ldots;\psi_n?;\epsilon$ with a head of δ_2. Our definition of $\mathsf{head}(\delta_1\|\delta_2)$ can be explained in a similar way, and the other cases should be obvious.

Example 7. As an example, consider the following program expression

$$\delta = (\psi?;\alpha)^*;\big((\neg\psi)?;\langle\rangle\big),$$

which corresponds to a while-loop with condition ψ and body α. It is easy to see that $\mathsf{head}((\psi?;\alpha)^*) = \{\psi?;\alpha,\epsilon\}$. This means that, if we want to execute $(\psi?;\alpha)^*$, then in the first step we can execute α if ψ is satisfied, or we can terminate (indicated by the action ϵ). For the subsequent expression in δ we obtain $\mathsf{head}((\neg\psi)?;\langle\rangle) = \{(\neg\psi)?;\epsilon\}$. To determine $\mathsf{head}(\delta)$, we apply the definition of $\mathsf{head}(\delta_1;\delta_2)$, which yields $\mathsf{head}(\delta) = \{\psi?;\alpha,(\neg\psi)?;\epsilon\}$.

Thus, when starting to execute the while-loop, we can either execute α if ψ is satisfied, or terminate if ψ is not satisfied. Now consider the program expression ρ, which describes the interleaving of two while-loops:

$$\rho = \big((\psi_0?;\alpha_0)^*;(\neg\psi_0?;\langle\rangle)\big) \;\|\; \big((\psi_1?;\alpha_1)^*;(\neg\psi_1?;\langle\rangle)\big)$$

We have

$$\mathsf{head}(\rho) = \{\psi_0?;\alpha_0,\ \psi_1?;\alpha_1,\ \neg\psi_0?;\psi_1?;\alpha_1,\ \neg\psi_1?;\psi_0?;\alpha_0,\ \neg\psi_0?;\neg\psi_1?;\epsilon\}.$$

First, we have the choice to execute α_0 if ψ_0 is satisfied or α_1 if ψ_1 is satisfied. Furthermore, if ψ_0 is not satisfied and ψ_1 is satisfied, then it is possible to terminate the first while-loop since $\neg\psi_0;\epsilon \in \mathsf{head}((\psi_0?;\alpha_0)^*;(\neg\psi_0?;\langle\rangle))$. But we then have to consider the parallel while-loop. Since ψ_1 is satisfied we cannot terminate the whole program, but must continue with the second while-loop. This case is reflected by $\neg\psi_0?;\psi_1?;\alpha_1 \in \mathsf{head}(\rho)$. In case neither ψ_0 nor ψ_1 is satisfied, the program terminates, which explains $\neg\psi_0?;\neg\psi_1?;\epsilon \in \mathsf{head}(\rho)$.

Next, we need to define the program(s) that remain to be executed once a guarded action from the head has been executed.

Definition 8. *The function* $\mathit{tail}(\cdot,\cdot)$ *maps a guarded action and a program expression to a set of program expressions.*

- *If* $\mathfrak{a} \notin \mathit{head}(\delta)$*, then* $\mathit{tail}(\mathfrak{a},\delta) = \emptyset$*.*
- *If* $\mathfrak{a} \in \mathit{head}(\delta)$ *and* $\mathfrak{a} = \psi_1?;...;\psi_n?;\epsilon$*, then* $\mathit{tail}(\mathfrak{a},\delta) = \{\langle\rangle\}$*.*
- *If* $\mathfrak{a} \in \mathit{head}(\delta)$ *and* $\mathfrak{a} = \psi_1?;...;\psi_n?;\alpha$ *for* $\alpha \in \Sigma\setminus\{\epsilon\}$*, then* $\mathit{tail}(\mathfrak{a},\delta)$ *is defined by induction on the combined size of* $\mathfrak{a}$ *and* δ*:*

$$
\begin{aligned}
&\mathit{tail}(\mathfrak{a},\langle\rangle) := \{\langle\rangle\};\\
&\mathit{tail}(\mathfrak{a},\beta) := \{\langle\rangle\} \text{ for } \beta \in \Sigma;^{4}\\
&\mathit{tail}(\mathfrak{a},\delta^*) := \{\delta';(\delta)^* \mid \delta' \in \mathit{tail}(\mathfrak{a},\delta)\};\\
&\mathit{tail}(\mathfrak{a},\psi?;\delta) := \mathit{tail}(\psi_2?;\ldots;\psi_n?;\alpha,\delta);^{5}\\
&\mathit{tail}(\mathfrak{a},\delta_1;\delta_2) := \{\delta';\delta_2 \mid \delta' \in \mathit{tail}(\mathfrak{a},\delta_1)\} \cup\\
&\qquad \{\delta'' \mid \exists j, 0 \le j \le n \text{ s.t. } \psi_1?;...;\psi_j?;\epsilon \in \mathit{head}(\delta_1) \wedge\\
&\qquad\quad \psi_{j+1}?;...;\psi_n?;\alpha \in \mathit{head}(\delta_2) \wedge\\
&\qquad\quad \delta'' \in \mathit{tail}(\psi_{j+1}?;...;\psi_n?;\alpha,\delta_2)\};\\
&\mathit{tail}(\mathfrak{a},\delta_1|\delta_2) := \mathit{tail}(\mathfrak{a},\delta_1) \cup \mathit{tail}(\mathfrak{a},\delta_2);\\
&\mathit{tail}(\mathfrak{a},\delta_1\|\delta_2) := \{\delta'\|\delta_2 \mid \delta' \in \mathit{tail}(\mathfrak{a},\delta_1)\} \cup \{\delta_1\|\delta' \mid \delta' \in \mathit{tail}(\mathfrak{a},\delta_2)\} \cup\\
&\qquad \{\delta'' \mid \exists j, 0 \le j \le n \text{ s.t. } \psi_1?;...;\psi_j?;\epsilon \in \mathit{head}(\delta_i) \wedge\\
&\qquad\quad \psi_{j+1}?;...;\psi_n?;\alpha \in \mathit{head}(\delta_{i'}) \wedge\\
&\qquad\quad \delta'' \in \mathit{tail}(\psi_{j+1}?;...;\psi_n?;\alpha,\delta_{i'}) \wedge \{1,2\} = \{i,i'\}\}.
\end{aligned}
$$

In the definitions of $\mathit{tail}(\mathfrak{a},\delta^*)$*,* $\mathit{tail}(\mathfrak{a},\delta_1;\delta_2)$*, and* $\mathit{tail}(\mathfrak{a},\delta_1\|\delta_2)$*, we omit* δ' *if* $\delta' = \langle\rangle$*.*

An example illustrating the definition of the tail function can be found in [6].

Intuitively, executing a program δ means first executing a guarded action of its head, then a guarded action of the head of its tail, etc. We call a program expression that can be reached by a sequence of such head and tail applications a reachable subprogram.

Definition 9. *Let* δ *be a program expression. The program expression* ρ *is a* reachable subprogram *of* δ *if there is an* $n \ge 0$ *and program expressions* $\delta_0, \delta_1, ...,$ δ_n *such that* $\delta_0 = \delta$*,* $\delta_n = \rho$*, and for all* $i = 0,\ldots,n-1$ *there exists* $\mathfrak{a}_i \in \mathit{head}(\delta_i)$ *such that* $\delta_{i+1} \in \mathit{tail}(\mathfrak{a}_i,\delta_i)$*. We denote the set of all reachable subprograms of* δ *by* $\mathit{sub}(\delta)$*.*

The following lemma is vital for our proof of decidability of the verification problem since it shows that there are only finitely many reachable subprograms of a given program expression.

[4] Note that $\mathfrak{a} \in \mathsf{head}(\beta)$ implies $\mathfrak{a} = \beta$.

[5] Note that $\mathfrak{a} = \psi_1?;...;\psi_n?;\alpha \in \mathsf{head}(\psi?;\delta)$ implies $\psi = \psi_1$.

Lemma 10. *Let δ be a program expression.*

1. *The cardinality of $\mathsf{sub}(\delta)$ is exponentially bounded in the size $|\delta|$ of δ.*
2. *If δ does not contain the interleaving constructor, then the size of $\mathsf{sub}(\delta)$ is polynomially bounded in the size $|\delta|$ of δ.*

A proof of this lemma can be found in [6]. Note that, in the presence of the interleaving operator, the exponential bound is actually reached. In fact, consider the program expression $\delta = \alpha_1 \parallel (\alpha_2 \parallel \ldots (\alpha_{n-1} \parallel \alpha_n)\ldots)$. We claim that $\mathsf{sub}(\delta)$ contains at least 2^n many reachable subprograms. In fact, it is easy to see that for every subset $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ with $i_1 \leq \ldots \leq i_k$ the expression $\alpha_{i_1} \parallel (\alpha_{i_2} \parallel \ldots (\alpha_{i_{k-1}} \parallel \alpha_{i_k})\ldots)$ is a reachable subprogram of δ.

Now we are ready to define the semantics of a DL-Golog program. Similar to the semantics introduced in [7], a program induces an infinite transition system. The states of this transition system are *program configurations*, which are pairs consisting of a program expression and a model of the TBox. To make a transition from one configuration to another, we pick an applicable guarded action from the head of the program expression, and then transform the model and the program expression using the semantics of actions and the tail function.

Definition 11. *Let $\mathfrak{a} = \psi_1?; \ldots; \psi_n?; \alpha$ be a guarded action and $\mathcal{I}$ an interpretation. We say that $\mathfrak{a}$ is* applicable *to $\mathcal{I}$ iff $\mathcal{I} \models \psi_i$ holds for all $i \in \{1, \ldots, n\}$ and the action α is applicable to $\mathcal{I}$.*[6]

A program configuration of the form $(\mathcal{I}, \delta)$ is called a *failing configuration* if no $\mathfrak{a} \in \mathsf{head}(\delta)$ is applicable to $\mathcal{I}$. To indicate such a crash of the program execution, we add another predefined action $\mathfrak{f}$ to Σ. The action $\mathfrak{f}$ is applicable to all models of $\mathcal{T}$, and we set $\mathcal{E}(\mathfrak{f}, \mathcal{I}) := \{\mathit{Fail}(p)\}$ where *Fail* is a fresh concept name. In addition we require that $\neg \mathit{Fail}(p) \in \mathcal{A}$ for the initial ABox $\mathcal{A}$.

Definition 12 (program semantics). *Let $\mathcal{P} = (\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E}, \delta)$ be a DL-Golog program. The* transition system $T_{\mathcal{P}} = (Q, \rightarrow, I)$ *induced by $\mathcal{P}$ consists of the set of* program configurations $Q := \mathcal{M}(\mathcal{T}) \times \mathsf{sub}(\delta)$ *as well as a transition relation $\rightarrow \subseteq Q \times \Sigma \times Q$ and a set of initial configurations $I \subseteq Q$, which are defined as follows:*

- *The* initial configurations *are defined as $I := \{(\mathcal{I}, \delta) \mid \mathcal{I} \in \mathcal{M}(\mathcal{A}) \cap \mathcal{M}(\mathcal{T})\}$.*
- *We have $\big((\mathcal{I}, \rho), \alpha, (\mathcal{I}', \rho')\big) \in \rightarrow$ (written as $(\mathcal{I}, \rho) \xrightarrow{\alpha} (\mathcal{I}', \rho')$) if one of the following conditions is satisfied:*
  1. *There exists $\mathfrak{a} = \psi_1?, \ldots, \psi_n?; \alpha \in \mathsf{head}(\rho)$ such that $\mathfrak{a}$ is applicable to $\mathcal{I}$ and it holds that $\mathcal{I} \Longrightarrow_{\alpha}^{\mathcal{E}} \mathcal{I}'$ and $\rho' \in \mathsf{tail}(\mathfrak{a}, \rho)$.*
  2. *There is no $\mathfrak{a} \in \mathsf{head}(\rho)$ such that $\mathfrak{a}$ is applicable to $\mathcal{I}$ and it holds that $\alpha = \mathfrak{f}$, $\rho = \rho'$ and $\mathcal{I} \Longrightarrow_{\mathfrak{f}}^{\mathcal{E}} \mathcal{I}'$.*

[6] Recall that tests are Boolean combinations of assertions. The notion of satisfaction in an interpretation is extended in the obvious way from assertions to such Boolean combinations.

Since we assume that all actions are consistent, there always exists a successor configuration $\mathcal{I}'$ in Case 1 of the definition of $\stackrel{\alpha}{\to}$. This fact, together with the presence of the predefined actions ϵ and $\mathfrak{f}$, ensures that every configuration in Q has at least one successor configuration. In particular, every initial configuration $(\mathcal{I}_0, \delta_0) \in I$ is the starting point of an infinite path in the transition system $T_\mathcal{P}$:

$$\pi = (\mathcal{I}_0, \delta_0) \stackrel{\alpha_0}{\to} (\mathcal{I}_1, \delta_1) \stackrel{\alpha_1}{\to} (\mathcal{I}_2, \delta_2) \stackrel{\alpha_3}{\to} (\mathcal{I}_3, \delta_3) \stackrel{\alpha_4}{\to} \dots$$

We call such a path a *run* of the DL-Golog program $\mathcal{P} = (\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E}, \delta)$. The *action trace* of the run π is an infinite word $w(\pi)$ over Σ with $w(\pi) = \alpha_0\alpha_1\alpha_2\alpha_3\dots$. The infinite sequence of interpretations $\mathcal{I}_0, \mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3, \dots$ occurring in the configurations in π is denoted by $\mathfrak{I}(\pi)$. By the definition of $\stackrel{\alpha}{\to}$ we have $\mathcal{I}_i \Longrightarrow^{\mathcal{E}}_{\alpha_i} \mathcal{I}_{i+1}$ for all $i \geq 0$.

The introduction of the predefined actions ϵ and $\mathfrak{f}$ allows us to treat non-terminating, terminating, and failing runs of a given program in a uniform way. Let $\Delta := \Sigma \setminus \{\epsilon, \mathfrak{f}\}$ denote the set of proper actions. A run π of a program $\mathcal{P}$ is called

- a *terminating run* if $w(\pi) \in \Delta^* \cdot \{\epsilon\}^\omega$,[7]
- a *failing run* if $w(\pi) \in \Delta^* \cdot \{\mathfrak{f}\}^\omega$,
- a *non-terminating run* if $w(\pi) \in \Delta^\omega$.

It is easy to show that, according to this definition, every run of a program is either terminating, non-terminating, or failing.

4 Verifying Temporal Properties of DL-Golog Programs

First, we need to introduce the temporal logic used to specify properties of runs of a program. As in [5], we use a variant of the temporalized DL $\mathcal{ALC}$-LTL [3], which we call $\mathcal{ALCO}$-LTL since it is based on the more expressive DL $\mathcal{ALCO}$. The syntax is the same as for propositional linear time logic (LTL), but in place of propositions, assertions are used.[8] More precisely, $\mathcal{ALCO}$-LTL formulas are built according to the following grammar:

$$\Phi ::= \varphi \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \mathsf{X}\Phi \mid \Phi_1 \mathsf{U} \Phi_2$$

where φ stands for assertions. As usual, $\Diamond\Phi$ (*eventually*) and $\Box\Phi$ (*always*) are used as abbreviations for $\top(a) \mathsf{U} \Phi$ and $\neg\Diamond\neg\Phi$, respectively. Moreover, $\Phi_1 \to \Phi_2$ abbreviates $\neg\Phi_1 \vee \Phi_2$.

The semantics of $\mathcal{ALCO}$-LTL is based on the notion of an *$\mathcal{ALCO}$-LTL structure*, which is an infinite sequence of interpretations $\mathfrak{I} = (\mathcal{I}_i)_{i=0,1,2,\cdots}$ over a

[7] i.e., a finite sequence of proper actions followed by an infinite sequence using only the action ϵ.

[8] In $\mathcal{ALC}$-LTL, also GCIs can occur in place of propositions, but this is not needed in the context of this paper. The domain constraints constitute a global TBox that must hold at every time point.

common domain Δ such that $a^{\mathcal{I}_i} = a^{\mathcal{I}_j}$ is satisfied for all $a \in N_I$ and all $i, j \in \{0, 1, 2, \dots\}$. Let Φ be an $\mathcal{ALCO}$-LTL formula, $\mathfrak{I}$ an $\mathcal{ALCO}$-LTL structure, and $i \in \{0, 1, 2, \dots\}$ a time point. Validity of Φ in $\mathfrak{I}$ at time i, denoted by $\mathfrak{I}, i \models \Phi$, is defined as follows:

$$\begin{array}{ll}
\mathfrak{I}, i \models \varphi & \text{iff } \mathcal{I}_i \models \varphi, \\
\mathfrak{I}, i \models \neg\Phi & \text{iff } \mathfrak{I}, i \not\models \Phi, \\
\mathfrak{I}, i \models \Phi_1 \wedge \Phi_2 & \text{iff } \mathfrak{I}, i \models \Phi_1 \text{ and } \mathfrak{I}, i \models \Phi_2, \\
\mathfrak{I}, i \models \Phi_1 \vee \Phi_2 & \text{iff } \mathfrak{I}, i \models \Phi_1 \text{ or } \mathfrak{I}, i \models \Phi_2, \\
\mathfrak{I}, i \models \mathsf{X}\Phi & \text{iff } \mathfrak{I}, i+1 \models \Phi, \\
\mathfrak{I}, i \models \Phi_1 \,\mathsf{U}\, \Phi_2 & \text{iff } \exists k \geq i : \mathfrak{I}, k \models \Phi_2 \text{ and } \forall j, i \leq j < k : \mathfrak{I}, j \models \Phi_1
\end{array}$$

We can use $\mathcal{ALCO}$-LTL formulas to specify desired or unwanted properties of (runs of) DL-Golog programs.

Definition 13 (verification problem). *Let $\mathcal{P} = (\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E}, \delta)$ be a DL-Golog program and Φ an $\mathcal{ALCO}$-LTL formula such that Φ contain only assertions from the set $\mathcal{D}$. The formula Φ is* valid *in $\mathcal{P}$, written as $\mathcal{P} \models \Phi$, if for all runs π of $\mathcal{P}$ it holds that $\mathfrak{I}(\pi), 0 \models \Phi$. The formula Φ is* satisfiable *in $\mathcal{P}$ if there exists a run π of $\mathcal{P}$ such that $\mathfrak{I}(\pi), 0 \models \Phi$.*

Using the literals $\mathit{Term}(p)$ and $\mathit{Fail}(p)$, we can encode variants of the verification problem into the formula. For example, we can check whether there is a failing run by testing satisfiability of the formula $\Diamond \mathit{Fail}(p)$. The fact that all runs of the program are terminating corresponds to the validity of the formula $\Diamond \mathit{Term}(p)$. In order to verify that all infinite runs of the program satisfy Φ, we can test validity of the formula $\Box(\neg \mathit{Fail}(p) \wedge \neg \mathit{Term}(p)) \rightarrow \Phi$.

Clearly, Φ is valid in $\mathcal{P}$ iff $\neg\Phi$ is unsatisfiable in $\mathcal{P}$. Therefore, it is sufficient to focus on showing decidability of the satisfiability problem. To obtain decidability, we need to impose additional restrictions on our action theories. Basically, we need to ensure that the effect function is computable and that we can construct a finite abstraction of the infinite transition system $T_{\mathcal{P}}$ that contains enough information on the runs of the program to enable verification based on this abstraction.

In order to obtain this finite abstraction, we use the notion of a static type to partition the infinite set of models of the TBox into finitely many equivalence classes of elements of the same type.

Definition 14. *Given a DL-based action theory $(\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E})$, a* static type *for this theory is a set $\mathfrak{t} \subseteq \mathcal{D}$ such that the ABox $\mathfrak{t}$ is consistent w.r.t. $\mathcal{T}$ and for all $\neg\varphi \in \mathcal{D}$ we have $\varphi \in \mathfrak{t}$ iff $\neg\varphi \notin \mathfrak{t}$. The* static type of a model $\mathcal{I}$ *of $\mathcal{T}$ is defined as $\mathsf{s\text{-}type}(\mathcal{I}) := \{\varphi \in \mathcal{D} \mid \mathcal{I} \models \varphi\}$.*

Obviously, given a model $\mathcal{I}$ of $\mathcal{T}$, the set $\mathsf{s\text{-}type}(\mathcal{I})$ is indeed a static type. Conversely, for every static type $\mathfrak{t}$, there is a model $\mathcal{I}$ of $\mathcal{T}$ such that $\mathfrak{t} = \mathsf{s\text{-}type}(\mathcal{I})$. Intuitively, models of the same static type cannot be distinguished by an assertion occurring in $\mathcal{D}$.

We are now ready to formulate conditions on DL-based action theories that ensure decidability of the satisfiability problem.

Definition 15 (admissibility). *The DL-based action theory* $(\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E})$ *is called* admissible *if all its actions are consistent and the conditions (A1), (A2), (A3) are satisfied for models* $\mathcal{I}$ *and* $\mathcal{J}$ *of* $\mathcal{T}$*:*

(A1) *If* $\textsf{s-type}(\mathcal{I}) = \textsf{s-type}(\mathcal{J})$*, then* α *is applicable to* $\mathcal{I}$ *iff* α *is applicable to* $\mathcal{J}$
(A2) *If* $\textsf{s-type}(\mathcal{I}) = \textsf{s-type}(\mathcal{J})$ *and* α *is applicable to* $\mathcal{I}$*, then* $\mathcal{E}(\alpha, \mathcal{I}) = \mathcal{E}(\alpha, \mathcal{J})$*.*

If (A1) and (A2) are satisfied and t *is a static type, then we define* $\mathcal{E}(\alpha, \mathsf{t}) := \mathcal{E}(\alpha, \mathcal{I})$*, where* $\mathcal{I}$ *is an arbitrary model of* $\mathcal{T}$ *with* $\mathsf{t} = \textsf{s-type}(\mathcal{I})$*.*

(A3) *For a given static type* t*, it is decidable whether* $\mathcal{E}(\alpha, \mathsf{t})$ *is defined. If it is defined, then this set can be effectively computed.*

It is easy to see that the action theories introduced in [1,4] are indeed admissible (if all actions are required to be consistent). For example, in both formalisms applicability of an action α to a model $\mathcal{I}$ depends on whether the preconditions of α are satisfied in $\mathcal{I}$. Since these preconditions are assertions in $\mathcal{D}$, (A1) is obviously satisfied.

Unfortunately, given two models of the same static type and an action that is applicable to these models, the models obtained by applying the action need not have the same static type. This is illustrated by the following example.

Example 16. Assume that $\mathcal{A} = \{B(b), r(a, b), \exists r.B(a)\}$,

$$\mathcal{D} = \{B(b), \neg B(b), r(a, b), \neg r(a, b), \exists r.B(a), \neg\exists r.B(a)\},$$

and $\mathcal{T} = \emptyset$, and consider an action α with the effect function

$$\mathcal{E}(\alpha, \mathcal{I}) = \{\neg B(b)\} \text{ if } \mathcal{I} \models \exists r.B(a).$$

Otherwise, $\mathcal{E}(\alpha, \mathcal{I})$ is undefined. Intuitively, $\exists r.B(a)$ is the precondition of α and $\neg B(b)$ the effect. It is easy to see that this action theory is admissible.

Consider two models $\mathcal{I}_1$ and $\mathcal{I}_2$ of $\mathcal{A}$ over the same domain with

$$\Delta^{\mathcal{I}_1} = \Delta^{\mathcal{I}_2} = \{a, b, d\}, r^{\mathcal{I}_1} = \{(a, b)\}, r^{\mathcal{I}_2} = \{(a, b), (a, d)\}, B^{\mathcal{I}_1} = B^{\mathcal{I}_2} = \{b, d\},$$

such that $a^{\mathcal{I}_i} = a$, $b^{\mathcal{I}_i} = b$ $(i = 1, 2)$ and $d \notin \{a, b\}$. Clearly, we have $\textsf{s-type}(\mathcal{I}_1) = \textsf{s-type}(\mathcal{I}_2)$ and $\mathcal{E}(\alpha, \mathcal{I}_1) = \mathcal{E}(\alpha, \mathcal{I}_2) = \{\neg B(b)\}$. However, for the interpretations $\mathcal{I}_1', \mathcal{I}_2'$ with $\mathcal{I}_1 \Longrightarrow_\alpha^{\{\neg B(b)\}} \mathcal{I}_1'$ and $\mathcal{I}_2 \Longrightarrow_\alpha^{\{\neg B(b)\}} \mathcal{I}_2'$, it holds that $\mathcal{I}_1' \not\models \exists r.B(a)$ and $\mathcal{I}_2' \models \exists r.B(a)$. Therefore, $\mathcal{I}_1'$ and $\mathcal{I}_2'$ do not have the same static type. Also note that α is applicable to $\mathcal{I}_2'$, but not to $\mathcal{I}_1'$.

This example shows that in general the mapping of models to their static types does *not* preserve the transition relation $\Longrightarrow_\alpha^{\mathcal{E}}$ on models for an action α. Therefore, the transition relation cannot be lifted to a transition relation on static types. To overcome this problem, we define an extended notion of types, called dynamic types. This requires the introduction of some more notation. Let $\mathcal{I}$ be an interpretation and $E \subseteq \mathit{Lit}$ a *non-contradictory* set of literals i.e., a set such that there is no L such that $\{L, \neg L\} \subseteq E$. The *updated interpretation* $\mathcal{I}^E$ w.r.t. E is defined as follows:

- $A^{\mathcal{I}^E} := (A^{\mathcal{I}} \cup \{a^{\mathcal{I}} \mid A(a) \in E\}) \setminus \{a^{\mathcal{I}} \mid \neg A(a) \in E\}$ for all $A \in N_C$ and
- $r^{\mathcal{I}^E} := (r^{\mathcal{I}} \cup \{(a^{\mathcal{I}}, b^{\mathcal{I}}) \mid r(a,b) \in E\}) \setminus \{(a^{\mathcal{I}}, b^{\mathcal{I}}) \mid \neg r(a,b) \in E\}$ for all $r \in N_R$.

Note that $\mathcal{I} \Longrightarrow_{\alpha}^{\mathcal{E}} \mathcal{I}'$ implies $\mathcal{I}' = \mathcal{I}^{\mathcal{E}(\alpha,\mathcal{I})}$. Using the notation $\neg E := \{\neg L \mid L \in E\}$ (modulo elimination of double negation), we can reduce iterated update operations to a single one: For two non-contradictory sets of literals E and E' we have

$$(\mathcal{I}^E)^{E'} = \mathcal{I}^{((E \setminus \neg E') \cup E')}. \tag{2}$$

Given a DL-Golog program $\mathcal{P} = (\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E}, \delta)$, we call a pair (φ, E) consisting of an assertion $\varphi \in \mathcal{D}$ and a non-contradictory set of literals $E \subseteq \mathit{Lit}$ a *type element* for $\mathcal{P}$. The set of all type elements for $\mathcal{P}$ is denoted by $\mathit{TE}(\mathcal{P})$.

Definition 17 (dynamic types). *Let $\mathcal{P} = (\mathcal{A}, \mathcal{T}, \Sigma, \mathcal{D}, \mathcal{E}, \delta)$ be a DL-Golog program and $\mathcal{I}$ a model of $\mathcal{T}$. The* dynamic type of $\mathcal{I}$ *is defined as*

$$\textit{d-type}(\mathcal{I}) := \{(\varphi, E) \in \mathit{TE}(\mathcal{P}) \mid \mathcal{I}^E \models \varphi\}.$$

A set $\mathfrak{t} \subseteq \mathit{TE}(\mathcal{P})$ is called a dynamic type *if there is an interpretation $\mathcal{I}$ such that $\mathfrak{t} = \textit{d-type}(\mathcal{I})$.*

Since $(\varphi, \emptyset) \in \mathsf{d\text{-}type}(\mathcal{I})$ iff $\mathcal{I} \models \varphi$ iff $\varphi \in \mathsf{s\text{-}type}(\mathcal{I})$, models with the same dynamic type also have the same static type. Given a dynamic type $\mathfrak{t}$, we denote the corresponding static type by $\mathcal{A}(\mathfrak{t})$, i.e., $\mathcal{A}(\mathfrak{t}) := \{\varphi \mid (\varphi, \emptyset) \in \mathfrak{t}\}$. The next lemma shows that dynamic types have the desired property that the execution of an action transforms models of the same type again into models of the same type. The dynamic type of the successor models can easily be computed using the identity (2).

Lemma 18. *Let $\mathcal{I}, \mathcal{J}$ be models of $\mathcal{T}$ and α an action such that $\mathcal{I} \Longrightarrow_{\alpha}^{\mathcal{E}} \mathcal{I}'$ and $\mathcal{J} \Longrightarrow_{\alpha}^{\mathcal{E}} \mathcal{J}'$.*

1. *If $\textit{d-type}(\mathcal{I}) = \textit{d-type}(\mathcal{J})$, then $\textit{d-type}(\mathcal{I}') = \textit{d-type}(\mathcal{J}')$.*
2. *If $\mathcal{E}(\alpha, \mathcal{I}) = E'$ and $(\varphi, E) \in \mathit{TE}(\mathcal{P})$, then $(\varphi, E) \in \textit{d-type}(\mathcal{I}')$ iff $(\varphi, (E' \setminus \neg E) \cup E) \in \textit{d-type}(\mathcal{I})$.*

Given a subset $\mathfrak{t}$ of $\mathcal{D}$, DL reasoning can obviously be used to decide whether $\mathfrak{t}$ is a static type or not. Given a subset $\mathfrak{t}$ of $\mathit{TE}(\mathcal{P})$, it is also possible to reduce the check whether it is a dynamic type to DL reasoning, but how to do this is less obvious. In [6] we show how to adapt the *reduction approach* developed in [1] for deciding the projection problem to this purpose. The idea is to encode the model $\mathcal{I}$ and the updated interpretations $\mathcal{I}^E$ into one single model of an appropriate TBox and ABox. Basically, for every non-contradictory set of literals E we introduce renamed copies of all concept and role names, which also yield renamed copies $\varphi^{(E)}$ of all assertion $\varphi \in \mathcal{D}$. The reduction TBox $\mathcal{T}_{\text{red}}$ and the reduction ABox $\mathcal{A}_{\text{red}}(\mathfrak{t})$ are then constructed such that the following lemma holds.

Lemma 19. *Let $\mathfrak{t} \subseteq TE(\mathcal{P})$. Then $\mathfrak{t}$ is a dynamic type iff the following two properties are satisfied:*

1. *For all $(\neg\varphi, E) \in TE(\mathcal{P})$ it holds that $(\varphi, E) \in \mathfrak{t}$ iff $(\neg\varphi, E) \notin \mathfrak{t}$.*
2. *There exists a model $\mathcal{J}$ of $\mathcal{A}_{red}(\mathfrak{t})$ and $\mathcal{T}_{red}$ such that $\mathcal{J} \models \varphi^{(E)}$ holds for all $(\varphi, E) \in \mathfrak{t}$.*

Basically, our the decision procedure for the satisfiability of an $\mathcal{ALCO}$-LTL formula Φ in a DL-Golog program $\mathcal{P}$ works as follows. Instead of the transition system $T_{\mathcal{P}}$ we consider the quotient system $\widehat{T}_{\mathcal{P}}$ that is obtained from $T_{\mathcal{P}}$ by replacing models by their dynamic types. Since there are only finitely many dynamic types and reachable subprograms, this quotient transition system is finite. Similar to the well-known construction for propositional LTL, the formula Φ can be translated into a Büchi automaton $\mathcal{B}_{\Phi}$ such that Φ is satisfiable iff $\mathcal{B}_{\Phi}$ accepts a non-empty language. In order to test satisfiability in $\mathcal{P}$, we consider the Büchi automaton obtained by building the product of $\mathcal{B}_{\Phi}$ and $\widehat{T}_{\mathcal{P}}$, and test it for non-emptiness. In principle, this test boils down to a reachability problem (is there a final state that can be reached from an initial state and from itself). To solve this problem, we guess a dynamic type that satisfies the initial ABox (using the decision procedure for dynamic types obtained from Lemma 18) and pair it with an appropriate initial state of the Büchi automaton. Then we make transitions in the product automaton, where the transitions in the component corresponding to $\widehat{T}_{\mathcal{P}}$ are realized using the head and tail function for computing the new program expression and 2. of Lemma 18 for computing the new dynamic type. More details on how this decision procedure works can be found in [6].

Theorem 20. *For DL-Golog programs $\mathcal{P}$ whose underlying action theory is admissible, satisfiability of an $\mathcal{ALCO}$-LTL formula in $\mathcal{P}$ is decidable.*

The exact complexity of this decision procedure depends, of course, also on the complexity of computing the effect function $\mathcal{E}$. For the action formalisms in [1,4], this is the same complexity as reasoning in $\mathcal{ALCO}$ (i.e., ExpTime if the TBox contains GCIs, and PSpace if the TBox is empty or acyclic). In this case, the overall complexity of deciding satisfiability in a DL-Golog program is majorized by the complexity of testing whether a set $\mathfrak{t} \subseteq TE(\mathcal{P})$ is a dynamic type or not. Due to the fact that there are exponentially many type elements, the reduction TBox and ABox are of exponential size, and thus the complexity of this test is 2ExpTime if the TBox contains GCIs, and ExpSpace if the TBox is empty or acyclic.

5 Conclusion

We have shown first results on how to verify temporal properties of Golog programs that use Description Logic actions. The two main contributions of this paper are the following. First, we have defined a semantics for DL-Golog programs that is similar to the semantics of Golog program introduced in [7], but treats non-terminating, terminating, and failing runs of a given program in a uniform way. Second, we have introduced the new notion of a dynamic type, and have shown that it allows us to obtain a finite, semantics-preserving abstraction of the infinite transition system induced by a program. This is the main reason why the verification problem turns out to be decidable.

In our future work, we intend to extend our results both to more expressive action theories (e.g. ones with non-deterministic atomic actions) and to additional program construct. Regarding the temporal logic used to specify properties, we will also look at CTL and CTL* in place of LTL.

References

1. Baader, F., Lutz, C., Milicic, M., Sattler, U., Wolter, F.: Integrating description logics and action formalisms: First results. In: Proc. AAAI'05 (2005)
2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
3. Baader, F., Ghilardi, S., Lutz, C.: LTL over description logic axioms. ACM Trans. Comput. Log. 13(3) (2012)
4. Baader, F., Lippmann, M., Liu, H.: Using causal relationships to deal with the ramification problem in action formalisms based on description logics. In: Proc. LPAR-17. LNCS 6397 (2010)
5. Baader, F., Liu, H., ul Mehdi, A.: Verifying properties of infinite sequences of description logic actions. In: Proc. ECAI'10 (2010)
6. Baader, F., Zarrieß, B.: Verification of golog programs over description logic actions. LTCS-Report 13-08, Chair of Automata Theory, TU Dresden, Dresden, Germany (2013), `http://lat.inf.tu-dresden.de/research/reports.html`
7. Claßen, J., Lakemeyer, G.: A logic for non-terminating golog programs. In: Proc. KR'08 (2008)
8. De Giacomo, G., Lespérance, Y., Levesque, H.J.: Congolog, a concurrent programming language based on the situation calculus. Artif. Intell. 121(1-2), 109–169 (Aug 2000)
9. De Giacomo, G., Lespérance, Y., Patrizi, F.: Bounded situation calculus action theories and decidable verification. In: Proc. KR'12 (2012)
10. H. Liu, C. Lutz, M. Milicic, F. Wolter: Reasoning about actions using description logics with general TBoxes. In: Proc. JELIA'06, Springer LNAI 4160 (2006)
11. Levesque, H.J., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.B.: Golog: A logic programming language for dynamic domains. J. Log. Program. 31(1-3), 59–83 (1997)
12. Pnueli, A.: The temporal logic of programs. In: Proc. FOCS'77 (1977)