Modeling and Reasoning in Event Calculus Using Goal-Directed Constraint Answer Set Programming[⋆]

Joaquín Arias[1,2], Zhuo Chen[3], Manuel Carro[1,2], and Gopal Gupta[3]

[1] IMDEA Software Institute
[2] Universidad Politécnica de Madrid
joaquin.arias@imdea.org, manuel.carro@{imdea.org,upm.es}
[3] University of Texas at Dallas
{zhuo.chen,gupta}@utdallas.edu

Abstract. Automated commonsense reasoning is essential for building human-like AI systems featuring, for example, explainable AI. Event Calculus (EC) is a family of formalisms that model commonsense reasoning with a sound, logical basis. Previous attempts to mechanize reasoning using EC faced difficulties in the treatment of the continuous change in dense domains (e.g., time and other physical quantities), constraints among variables, default negation, and the uniform application of different inference methods, among others. We propose the use of s(CASP), a query-driven, top-down execution model for Predicate Answer Set Programming with Constraints, to model and reason using EC. We show how EC scenarios can be naturally and directly encoded in s(CASP) and how its expressiveness makes it possible to perform deductive and abductive reasoning tasks in domains featuring, for example, constraints involving dense time and fluents.

Keywords: ASP · goal-directed · Event Calculus · Constraints

1 Introduction

The ability to model continuous characteristics of the world is essential for Commonsense Reasoning (**CR**) in many domains that require dealing with continuous change: time, the height of a falling object, the gas level of a car, the water level in a sink, etc.

Event Calculus (**EC**) is a formalism based on many-sorted predicate logic [13, 23] that can represent continuous change and capture the commonsense law of inertia, whose modeling is a pervasive problem in CR. In EC, time-dependent properties and events are seen as objects and reasoning is performed on the truth values of properties and the occurrences of events at a point in time.

[⋆] Work partially supported by EIT Digital (https://eitdigital.eu), MINECO project TIN2015-67522-C3-1-R (TRACES), Comunidad de Madrid project S2018/TCS-4339 BLOQUES-CM co-funded by EIE Funds of the European Union, and US NSF Grant IIS 1718945.

Answer Set Programming (**ASP**) is a logic programming paradigm that was initially proposed to realize non-monotonic reasoning [17]. ASP has also been used to model the Event Calculus [15, 16]. Classical implementations of ASP are limited to variables ranging over discrete and bound domains and use mechanisms such as *grounding* and SAT solving to find out models (called *answer sets*) of ASP programs. However, non-monotonic reasoning often needs variables ranging over dense domains (e.g., those involving time or physical quantities) to faithfully represent the properties of these domains.

This paper presents an approach to modeling Event Calculus using s(CASP) [1] as the underlying reasoning infrastructure. The s(CASP) system is an implementation of Constraint Answer Set Programming over first-order predicates which combines ASP and constraints. It features predicates, constraints among non-ground variables, uninterpreted functions, and, most importantly, a top-down, query-driven execution strategy. These features make it possible to return answers with non-ground variables (possibly including constraints among them) and to compute partial models by returning only the fragment of a stable model that is necessary to answer a given query. Thanks to its interface with constraint solvers, sound non-monotonic reasoning with constraints is possible.

Our approach based on s(CASP) achieves more conciseness and expressiveness than other related approaches because dense domains can be faithfully modeled as continuous quantities, while in other proposals such domains [20, 16] had to be discretized, therefore losing precision or even soundness. Additionally, in our approach the amalgamation of ASP and constraints and its realization in s(CASP) is considerably more natural: under s(CASP), answer set programs are executed in a goal-directed manner so constraints encountered along the way are collected and solved dynamically as execution proceeds — this is very similar to the way in which Prolog has been extended with constraints. Implementations of other ASP systems featuring constraints are considerably more complex.

2 Related Work

Previous work translated *discrete* EC into ASP [15, 16] by reformulating the EC models as first-order stable models and translating the (almost universal) formulas of EC into a logic program that preserves stable models. Given a finite domain, EC2ASP (and its evolution, F2LP) compile (discrete) Event Calculus formulas into ASP programs [15, 16]. This translation scheme relies on two facts: second order circumscription and first order stable model semantics coincide on canonical formulas, and almost-universal formulas can be transformed into a logic program while the stable models are preserved. As a result, computing models of Event Calculus descriptions can be done by computing the stable models of an appropriately generated program under the stable model semantics.

Clearly, approaches featuring discrete domains cannot faithfully handle continuous quantities such as time. In addition, because of their reliance on SAT solvers to find the stable models, they can only handle *safe* programs. In contrast, the s(CASP) system, because of its direct support for predicates with arbitrary terms, constructive negation, and the novel *forall mechanism* [18, 1], program safety is not a requirement. Thus, s(CASP) can model Event Calculus axioms much more directly and elegantly.

Action languages [10] were developed by Gelfond and Lifschitz to model the elements of natural language that are used to describe the effects of actions. They have been implemented using answer set programming to perform planning and diagnosis [8]. There have been extensions of action languages to accommodate time: for example, the action language C+ has been extended by Lee and Meng to accommodate continuous time [14].

The approaches mentioned above assume discrete quantities and do not support reasoning about continuous time or change. As long as SAT-based ASP systems are used to model Event Calculus, continuous fluents cannot be straightforwardly expressed since they require unbounded, dense domains for the variables. The work closest to incorporating continuous time makes use of SMT solvers. In this approach, constraints are incorporated into ASP and the grounded theory is executed using an SMT solver [14]. However, this approach has not been directly applied to modeling the Event Calculus. The closest tool chain is ASPMT2SMT [3] that uses *gringo* to partially ground the ASPMT theories and generate constraints that are processed by *Z3*. However, regular, discrete ASP variables are at the heart of the model, and these are grounded and used to generate the constraints. Therefore, if these discrete variables approximate continuous variables in the model, the constraints generated will only approximate the conditions of the original problem and therefore their solutions will also be an approximation (or a subset) of the solutions for the real problem. In other words, the initial discretization done for the ASP variables will be propagated via the generated constraints to the final solutions that will, in the best case, be a discretized version of the actual solutions. As an example, if time is discretized, the solutions to the model will suffer from this discretization.

Other ASP-based approaches to deal with planning in continuous domains include, for example, PDDL+ [6], which was developed to allow reasoning with continuous time-dependent effects. It models temporal behavior in terms of the initiation and termination of processes, which in turn act upon the numeric components of states. Processes are initiated and terminated instantaneously by actions or exogenous events. Continuous changes are made by concurrent processes. In PDDL+, reasoning is monotonic and thus the degree of elaboration tolerance is low. There are implementations of PDDL+ using constraint answer set programming (CASP) [2] though these have not been applied to modeling the Event Calculus and requires the use of discrete variables to model some quantities, e.g., time.

EC can be written as a logic program, but it cannot be executed directly by Prolog [25], as it lacks some necessary features, such as constructive negation, deduction of negative literals, and (to some extent) detection of infinite failure [21]. A common approach is to write a metainterpreter specific to the EC variant at hand. This can be as complex as writing a (specialized) theorem prover or, more often, a specialized interpreter whose correctness is difficult to ascertain (see the code at [4]). Therefore, some Prolog implementations of EC do not completely formalize the calculus or implement a reduced version. In our case, we leverage on the capabilities of s(CASP) to provide constructive, sound negation, plus negative rule heads, and loop detection [1].

3 Background

Answer Set Programming is a logic programming and modelling language that evaluates normal logic programs under the stable model semantics [9]. s(ASP) [18] is a top-down, goal-driven ASP system that can evaluate ASP programs with function symbols (*functors*) **without** *grounding* them either before or during execution. Grounding is a procedure that substitutes program variables with the possible values from their domain. For most classical ASP solvers, grounding is a necessary pre-processing phase. Grounding, however, requires program variables to be restricted to take values in a finite domain. As a result, ASP solvers cannot be used to model continuous time or change.

3.1 s(CASP)

s(CASP) [1] extends s(ASP) similarly to how CLP extends Prolog. s(CASP) adds constraints to s(ASP); these constraints are kept and used both during execution and in the answer. Constraints have historically proved to be effective in improving both expressiveness and efficiency in logic programming, as constraints can succinctly express properties of a solution and reduce the search space. As a result, s(CASP) is more expressive and faster than s(ASP), while retaining the capability of executing non-ground predicate answer set programs. A s(CASP) program is a set of clauses of the following form:

$$\texttt{a :- c}_a\texttt{, b}_1\texttt{, ..., b}_m\texttt{, not b}_{m+1}\texttt{, ..., not b}_n\texttt{.}$$

where `a` and $\texttt{b}_1$, ..., $\texttt{b}_n$ are atoms and `not` corresponds to *default* negation. The difference w.r.t. an ASP program is $\texttt{c}_a$, a simple constraint or a conjunction of constraints.

In s(CASP), and unlike Prolog's negation as failure, `not p(X)` can return bindings for `X` on success. This is possible thanks to the use of constructive negation [18] and coinduction [11]. This highlights two differences w.r.t. Prolog: first, s(CASP) resolves negated atoms $\texttt{not b}_i$ against *dual rules* of the program [1, 18], which makes it possible for a non-ground call `not p(X)` to return the bindings for `X` for which the positive call `p(X)` would have failed, therefore supporting constructive negation. Second, the dual program is **not** interpreted under SLD semantics in order to handle the different kind of loops that can appear in s(CASP) [1, 18].

The execution of a s(CASP) program starts with a *query* of the form $\texttt{?- c}_q\texttt{, l}_1\texttt{,..., l}_m$, where $\texttt{l}_i$ are (negated) literals and $\texttt{c}_q$ is a conjunction of constraint(s). The answers to the query are partial stable models where each partial model is a subset of a stable model [9] including only the literals necessary to support the query (see [1] for details). Additionally, for each partial stable

Predicate	**Meaning**
$InitiallyN(f)$	fluent f is false at time 0
$InitiallyP(f)$	fluent f is true at time 0
$Happens(e, t)$	event e occurs at time t
$Initiates(e, f, t)$	if e happens at time t, f is true and not released from the commonsense law of inertia after t
$Terminates(e, f, t)$	if e occurs at time t, f is false and not released from the commonsense law of inertia after t
$Releases(e, f, t)$	if e occurs at time t, f is released from the commonsense law of inertia after t
$Trajectory(f_1, t_1, f_2, t_2)$	if f_1 is initiated by an event that occurs at t_1, then f_2 is true at t_2
$StoppedIn(t_1, f, t_2)$	f is stopped between t_1 and t_2
$StartedIn(t_1, f, t_2)$	f is started between t_1 and t_2
$HoldsAt(f, t)$	fluent f is true at time t

Fig. 1. Basic event calculus (BEC) predicates
(e = event, f, f_1, f_2 = fluents, t, t_1, t_2 = timepoints)

model s(CASP) returns on backtracking the justification tree and the bindings for the free variables of the query that correspond to the most general unifier (*mgu*) of a successful top-down derivation consistent with this stable model.

3.2 Event Calculus

EC is a formalism for reasoning about events and change [23], of which there are several axiomatizations. There are three basic, mutually related, concepts in EC: *events*, *fluents*, and *time points* (see Fig. 1). An event is an action or incident that may occur in the world: for instance, a person dropping a glass is an event. A fluent is a time-varying property of the world, such as the altitude of a glass. A time point is an instant of time. Events may happen at a time point; fluents have a truth value at any time point or over an interval, and these truth values are subject to change upon an occurrence of an event. In addition, fluents may have (continuous) quantities associated with them when they are true. For example, the event of dropping a glass initiates the fluent that captures that the glass is falling, and perhaps its height above the ground, and the event of holding a glass terminates the fluent that the glass is falling. An EC description consists of a universal theory and a domain narrative. The universal theory is a conjunction of EC axioms and the domain narrative consists of the causal laws of the domain and the known events and fluent properties.

Circumscription [19] is applied to EC domain narratives to minimize the extension of predicates and has two effects: the only events that happen are those defined and the only effects of events are those defined.

The original EC (OEC) was introduced by Kowalski and Sergot in 1986 [13]. OEC has sorts for event occurrences, fluents, and time periods. In this paper

BEC1. $StoppedIn(t_1, f, t_2) \equiv$
$$\exists e, t \;\; (\, Happens(e, t) \wedge t_1 < t < t_2 \wedge (\, Terminates(e, f, t) \vee Releases(e, f, t) \,)\,)$$

BEC2. $StartedIn(t_1, f, t_2) \equiv$
$$\exists e, t \;\; (\, Happens(e, t) \wedge t_1 < t < t_2 \wedge (\, Initiates(e, f, t) \vee Releases(e, f, t) \,)\,)$$

BEC3. $HoldsAt(f_2, t_2) \leftarrow$
$$Happens(e, t_1) \wedge Initiates(e, f_1, t_1) \wedge Trajectory(f_1, t_1, f_2, t_2) \wedge \neg StoppedIn(t_1, f_1, t_2)$$

BEC4. $HoldsAt(f, t) \leftarrow \quad InitiallyP(f) \wedge \neg StoppedIn(0, f, t)$

BEC5. $\neg HoldsAt(f, t) \leftarrow \quad InitiallyN(f) \wedge \neg StartedIn(0, f, t)$

BEC6. $HoldsAt(f, t_2) \leftarrow$
$$Happens(e, t_1) \wedge Initiates(e, f, t_1) \wedge t_1 < t_2 \wedge \neg StoppedIn(t_1, f, t_2)$$

BEC7. $\neg HoldsAt(f, t_2) \leftarrow$
$$Happens(e, t_1) \wedge Terminates(e, f, t_1) \wedge t_1 < t_2 \wedge \neg StartedIn(t_1, f, t_2)$$

Fig. 2. Formalization of BEC axioms [23].

we use the Basic Event Calculus (BEC) formulated by Shanahan [21]. Fig. 2 summarizes the seven BEC axioms. An explanation of these axioms follows:

– **Axiom BEC1**. A fluent f is stopped between time points t_1 and t_2 iff it is terminated or released by some event e that occurs after t_1 and before t_2.
– **Axiom BEC2**. A fluent f is started between time points t_1 and t_2 iff it is initiated or released by some event e that occurs after t_1 and before t_2.
– **Axiom BEC3**. A fluent f_2 is true at time t_2 if a fluent f_1 initiated at t_1 does not finish before t_2 and it makes fluent f_2 be true.[4]
– **Axiom BEC4**. A fluent f is true at time t if it is true at time 0 and is not stopped on or before t.
– **Axiom BEC5**. A fluent f is false at time t if it is false at time 0 and it is not started on or before t.
– **Axiom BEC6**. A fluent f is true at time t_2 if it is initiated at some earlier time t_1 and it is not stopped before t_2.
– **Axiom BEC7**. A fluent f is false at time t_2 if it is terminated at some earlier time t_1 and it is not started on or before t_2.

4 From Event Calculus to s(CASP)

4.1 Modeling EC with s(CASP)

Two key factors contribute to s(CASP)'s ability to model Event Calculus: the preservation of non-ground variables during the execution and the integration with constraint solvers.

[4] For implementation convenience, and without loss of expressiveness, we assume that argument t_2 in $Trajectory(f_1, t_1, f_2, t_2)$ is not a time difference w.r.t. t_1, but an absolute time after t_1.

```
%% BEC1
stoppedIn(T1,F,T2) :-
   T1 #< T, T #< T2,
   terminates(E,F,T),
   happens(E,T).
stoppedIn(T1,F,T2) :-
   T1 #< T, T #< T2,
   releases(E,F,T),
   happens(E,T).
%% BEC2
startedIn(T1,F,T2) :-
   T1 #< T, T #< T2,
   initiates(E,F,T),
   happens(E,T).
startedIn(T1,F,T2) :-
   T1 #< T, T #< T2,
   releases(E,F,T),
   happens(E,T).
%% BEC3
holdsAt(F2,T2) :-
   initiates(E,F1,T1),
   happens(E,T1),
   trajectory(F1,T1,F2,T2),
   not stoppedIn(T1,F1,T2).
%% BEC4
holdsAt(F,T) :-
   0 #< T, initiallyP(F),
   not stoppedIn(0,F,T).
%% BEC5
-holdsAt(F,T) :-
   0 #< T, initiallyN(F),
   not startedIn(0,F,T).
%% BEC6
holdsAt(F,T) :-
   T1 #< T, initiates(E,F,T1),
   happens(E,T1),
   not stoppedIn(T1,F,T).
%% BEC7
-holdsAt(F,T) :-
   T1 #< T, terminates(E,F,T1),
   happens(E,T1),
   not startedIn(T1,F,T).
```

Fig. 3. Basic Event Calculus (BCE) modeled in s(CASP)

Treatment of variables in s(CASP): Thanks to the usage of non-ground variables, s(CASP) is able to directly model Event Calculus axioms that would otherwise require "unsafe" rules. In classical ASP, a rule is safe when every variable that appears in its head or in a negated literal in its body also appears in a positive literal in the body of the rule, and it is unsafe otherwise. Safety guarantees that every variable can be grounded. For example, BEC4 is unsafe since parameter t, that appears in the head, does not appear in a positive literal in the body (i.e., it only appears in $\neg StoppedIn(0, f, t)$). A SAT-based ASP solver such as *clingo* [7] will not be able to directly process unsafe rules like this. However, the top-down strategy of the execution of s(CASP) makes it possible to keep logical variables both during execution and in answer sets and therefore free (logical) variables can be handled in heads and in negated literals.

Integration with constraint solvers: The s(CASP) system has a generic interface to enable plugging in constraint solvers. s(CASP) currently uses Holzbaur's CLP(Q) linear constraints solver [12], that supports the constraints $<, >, =, \neq, \leq, \geq$. As has been shown, all definitions and axioms in EC involve inequality comparisons over time points. The ability of s(CASP) to make use of constraint solvers makes it ideal to model continuous time in EC.

4.2 Translating BEC into s(CASP)

Our translation of EC description into s(CASP) program is similar to that of the systems EC2ASP and F2LP [15, 16], but we differ in some key aspects that improve performance and are relevant for expressiveness: *the treatment of rules with negated heads, the possibility of generating unsafe rules,* and *the use of constraints over rationals*. We describe below, with the help of a running example, the translation that turns logic statements (as found in BEC) into a s(CASP) program. The code corresponding to the translations of the axioms of BEC in Fig. 2 can be found in Fig. 3. s(CASP) code follows the syntactical conventions of logic programming: constants (including function names) and predicate symbols start with a lowercase letter and variables start with an uppercase letter. In addition, constraints are written as constraints in s(CASP), (e.g., `#<`) to make it clear that they do not correspond to Prolog's arithmetic comparisons:

- **Atoms and Constants:** Their names are preserved. *Uniqueness of Names* [24] is assumed by default (and enforced) in logic programming.
- **Constraints:** Predicates that represent constraints (e.g., on time) are directly translated to their counterparts in s(CASP). E.g., $t_1 < t_2$ becomes `T1#<T2`, where `#<` is in our examples handled by a linear constraint solver. The translation (and s(CASP) itself) is parametric on the constraint domain.
- **Definitions:** Axiomatizations of EC use definitions of the form $D(x) \equiv \exists y B(x,y)$, where $B(x,y)$ is a conjunction of (negated) atoms, disjunctions of atoms, and constraints (e.g., BEC1). The use of definitions makes it easier to build conceptual blocks out of basic predicates. However, for performance reasons we treat them as if they were written as $\forall x(D(x) \leftarrow \exists y B(x,y))$, following [16]. Intuitively, if we ignore the truth value of D in the (partial) models that s(CASP) generates, the models returned using implication and/or equivalence are the same, and the literal D can be ignored because it would anyway have disappeared had it been expanded. Additionally, s(CASP) internally performs Clark's completion [5] to the s(CASP) program, and therefore, we can assume that s(CASP) rules expresses all possible ways in which heads can be true.
- **Rules with Positive Heads:** A rule (e.g., BEC6)

  $$\forall x(H(x) \leftarrow \exists y(A(y) \wedge \neg B(x,y) \wedge x < y))$$

  where $x < y$ is a constraint, is translated into

  ```
  h(X) :- X #< Y, a(Y), not b(X,Y).
  ```

  Since s(CASP) performs left-to-right evaluation, placing constraints earlier in the rule is in general better, as constraint solvers are deterministic and constraining variables as soon as possible helps reduce the size of the search tree.
- **Rules with Negated Heads:** BEC rules 5 and 7 infer negated heads $\neg HoldsAt(f,t)$ while rule 4 and 6 infer positive heads $HoldsAt(f,t)$, i.e., they have the scheme

  $$\forall x(H(x) \leftarrow \exists y A(x,y)) \quad \wedge \quad \forall x(\neg H(x) \leftarrow \exists y B(x,y))$$

The standard approach to translate rules with negated heads is to convert them into global constraints [15]:

```
:- b(X,Y), h(X).
```

Our approach is to define instead a rule for the literal `-h(X)` that captures the explicit evidence that `h(X)` is false:

```
-h(X) :- b(X,Y).
```

which makes it possible to call `-h(X)` in a top-down execution. This construct was termed *classical* negation in [18] and behaves as a regular predicate, except that the s(CASP) compiler, to ensure that `-h(X)` and `h(X)` cannot be simultaneously true, automatically adds the global constraint `:- -h(X),h(X)`. Therefore, s(CASP) can detect an inconsistency (and returns an empty model) if both $HoldsAt(f,t)$ and $\neg HoldsAt(f,t)$ can be simultaneously derived from an EC narrative. Since circumscription is not applied to the BEC theory, not being able to derive $HoldsAt(f,t)$ does not immediately determine that its negation is true. We will see how this is connected with the translation of the narrative.

– **Rules with Disjunctive Bodies:** A rule (e.g., BEC1)

$$\forall x[H(x) \leftarrow \exists y(\ (A(x,y) \vee B(x,y)) \wedge C(x,y)\ )]$$

is translated into two separate clauses:

```
h(X) :- a(X,Y), c(X,Y).
h(X) :- b(X,Y), c(X,Y).
```

4.3 Translation of the narrative

The definition of a given scenario (its *narrative* part) states the basic actions and effects using the predicates in Fig. 1. EC assumes circumscription of the predicates defined in the *narrative*: the events (resp., effects) known to occur are the only events (resp., effects) that occur. Note that this is automatic in s(CASP), since it produces the Clark's completion of s(CASP) programs when generating the dual program. In addition, global constraints can restrict the admissible states of the system.

Every basic BEC predicate $P(x)$ (where P can stand for an event occurrence, an effect of an event on a fluent, etc.) is translated into a s(CASP) rule $P(x) \leftarrow \gamma$, where γ states **all** the cases where $P(x)$ is true. In many cases, these are *facts*, but in other cases γ captures the conditions for $P(x)$ to hold.

Let us consider example 14 in [23], which reasons about turning a light switch on and off. Fig. 4 shows the encoding of this example under s(CASP).

– **Events:** The description below (translated in lines 1-3 of Fig. 4):

$$\begin{aligned} Happens(e,t) \quad &\equiv (e = TurnOn \wedge (t = 2 \vee t = 6)) \ \vee \\ &\quad\ (e = TurnOff \wedge t = 4) \end{aligned}$$

states that the $TurnOn$ event will happen exclusively at time $t = 2$ or $t = 6$ and that $TurnOff$ will happen at $t = 4$.

```
1  happens(turn_on, 2).
2  happens(turn_off, 4).
3  happens(turn_on, 6).
4
5  initiates(turn_on, on, T).
6  terminates(turn_off, on, T).
7  trajectory(on, T1, red, T2) :-
8        T1 #< T2, T2 #< T1 + 1.
9  trajectory(on, T1, green, T2) :-
10       T2 #>= T1 + 1.
11 releases(turn_on, red, T).
12 releases(turn_on, green, T).
```

Fig. 4. Narrative of the light scenario modeled in s(CASP)

- **Event effects:** When the event $TurnOn$ happens, the light is put in *on* status; similarly, when the event $TurnOff$ happens, the *on* status of the light is terminated. In both cases, this can happen at any time t (lines 5 and 6 in Fig. 4)
- **Release from Inertia:** When turned on, the light emits red light within the first second, and then green light is emitted. $Trajectory$ expresses how this change depends on the time elapsed since an event occurrence. The $Trajectory$ formula has the shape $P(x) \leftarrow \gamma$, as we need to state the (time) conditions for the fluent to become activated (see lines 7-10 in Fig. 4). $Releases$ states that the color of the light is released from the commonsense law of inertia. After a fluent is released, its truth value is not determined by BEC and it can change. Thus, there may be models in which the fluent is true, and models in which the fluent may be false. Releasing a fluent (see lines 11 and 12 in Fig. 4) frees it up so that other axioms in the domain description can be used to determine its truth value, thus allowing us to represent continuous change of the fluent.
- **State Constraint:** State constraints usually contain $HoldsAt(f, t)$ or $\neg HoldsAt(f, t)$ and represent restrictions on the models. In our running example, a light cannot be red and green at the same time: $\forall t. \neg(HoldsAt(Red, t) \wedge HoldsAt(Green, t))$. This is translated as `:- holdsAt(red,T),holdsAt(green,T)`. Adding this constraint to the program in Fig. 4 does not change its models. However, if we change line 8 stating that the light is red for 2 second (i.e., `T2#<T1+2`), the state constraint is violated and therefore there are no valid models.
- **A Note on using $\neg HoldsAt(f, t)$ in γ:** The basic BEC predicates may depend on what the BEC theory can deduce, e.g., γ may depend on $HoldsAt(f, t)$ or $\neg HoldsAt(f, t)$ (see Fig. 5). $HoldsAt(f, t)$ can be invoked directly, but $\neg HoldsAt(f, t)$ ought to be called using classical negation, e.g., `-holdsAt(F,T)`. The reason is that since EC does not apply circumscription to its axioms, we can deduce only the truth (or falsehood) of a predicate when we have direct evidence of either of them — i.e., what the positive (`holdsAt(F,T)`) and negative (`-holdsAt(F,T)`) heads provide.

4.4 Continuous Change: A Complete Encoding

We consider now an example from [24]: a water tap fills a vessel, whose water level is subject to continuous change. When the level reaches the bucket rim, it

```
1  #include bec_theory.
2
3  max_level(10):- not max_level(16).
4  max_level(16):- not max_level(10).
5
6  initiallyP(level(0)).
7  happens(overflow,T).
8  happens(tapOn,5).
9
10 initiates(tapOn,filling,T).
11 terminates(tapOff,filling,T).
12 initiates(overflow,spilling,T):-
13    max_level(Max),
14    holdsAt(level(Max), T).
15 releases(tapOn,level(0),T):-
16    happens(tapOn,T).
17
18 trajectory(filling,T1,level(X2),T2):-
19    T1 #< T2, X2 #= X + T2-T1,
20    max_level(Max), X2 #=< Max,
21    holdsAt(level(X),T1).
22 trajectory(filling,T1,level(overflow),T2):-
23    T1 #< T2, X2 #= X + T2-T1,
24    max_level(Max), X2 #> Max,
25    holdsAt(level(X),T1).
26 trajectory(spilling,T1,leak(X),T2):-
27    holdsAt(filling, T2),
28    T1 #< T2, X #= T2-T1.
```

Fig. 5. Encoding of an Event Calculus narrative with continuous change

starts spilling. We will present the main ideas behind its encoding (Fig. 5) and will show some queries we can ask about its state and behavior.

- **Continuous Change:** The fluent *Level*(x) represents that the water is at level x in the vessel. The first *Trajectory* formula (lines 18-21) determines the time-dependent value of the *Level*(x) fluent,[5] which is active as long as the *Filling* fluent is true and the rim of the vessel is not reached. Additionally, the second *Trajectory* formula (lines 22-25) allows us to capture the fact that the water reached the rim of the vessel and overflowed.
- **Triggered Fluent:** The fluent *Spilling* is triggered (lines 12-14) when the water level reaches the rim of the vessel. As a consequence, the *Trajectory* formula in lines 26-28 starts the fluent *Leak*(x) and captures the amount of water leaked while the fluent *Spilling* holds.
- **Different Worlds:** The clauses in lines 3-4 force the vessel capacity to be either 10 or 16, i.e., they create two possible worlds/models: {`max_level(10), not max_level(16), ...`} and {`max_level(16), not max_level(10), ...`}. The same mechanism can be used to state whether an event happens or not. For this, a keyword `#abducible` is provided as a shortcut in s(CASP). We will use it in the *Abduction* subsection later on.

5 Examples and Evaluation

The benchmarks used in this section are available at `http://bit.ly/EventCalculus`. They were run on a MacOS 10.14.3 laptop with an Intel Core i5 at 2GHz.

[5] For simplicity the amount of water filled/leaked correspond directly to how long the water has been pouring in / spilling from the vessel.

Deduction: Deduction determines whether a state of the world is possible given a theory (in our case, BEC) and an initial narrative. We can perform deduction in BEC for the previous examples through queries to the corresponding s(CASP) program. For the lights scenario (Fig. 4):

`?- holdsAt(on,3)` succeeds: it deduces that the light is on at time 3.
`?- -holdsAt(on,5)` succeeds: the light is not on at time 5.
`?- holdsAt(F,3)` is true in one stable model containing `holdsAt(green,3)` and `holdsAt(on,3)`, meaning that at time 3, the light is on and green.

In the water level scenario (Fig. 5) we can make queries involving time and the water level:

`?- holdsAt(level(H),15/2)` is true when `H=5/2`.
`?- holdsAt(level(5/2),T)` is true when `T=15/2`.

Note that, as explained with more detail in the *Evaluation* subsection below, s(CASP) can operate and answer correctly queries involving rationals without having to modify the original program to introduce domains for the relevant variables or to *scale* the constants to convert rationals into integers.

Abduction: Abductive reasoning tries to determine a sequence of events/actions that reaches a final state. In the case of ASP, actions are naturally captured as the set of atoms that are true in a model which includes the initial and final states and are consistent with BEC. For the water scenario, (Fig. 5), let us assume we want to reach water level 14 at time 19. The query `?- holdsAt(level(14),19)` will return a single model with a vessel size of 16 and the rest of the atoms in the model capture what must (not) happen to reach this state.

More interesting abductive tasks can be performed: adding the line `#abducible happens(tapOff,U)` to the program, we specify that it is possible (but not necessary) for the tap to close at some time `U`. As we mentioned in Section 4.4, this directive is translated into code that creates different worlds/models. The query `?- holdsAt(spilling,T)` determines if the water may overspill and under which conditions. s(CASP) returns two models:

- One containing `T>15, holdsAt(spilling,T), happens(tapOn,5), 5<U<15, not happens(TapOff,U), max_level(10)` meaning that the water will spill at `T=15` if the vessel has a capacity of 10, the tap is open at `T=5`, and it is **not** closed between times 5 and 15.
- Another similar model, with the water spilling at `T=21` in a vessel with capacity of 16 and where the tap was not closed before `U=21`.

Note that s(CASP) determined the truth value of *Happens* and, more importantly, performed constraint solving to infer the time ranges during which some events ought (and ought not) to take place, represented by the negated atoms in the models inferred by constructive negation. Since all relevant atoms have a time parameter, they actually represent a *timed plan.* Due to the expressiveness of constraints, this plan contains information on time points when events

Table 1. Run time (ms) comparison for the light scenario.

Queries	s(CASP)	F2LP+clingo
`?-holdsAt(red,6.9).`	216	**73**
`?-holdsAt(red,6.99).`	**217**	8,798
`?-holdsAt(red,6.999).`	**213**	>5 min.

must (not) happen and also on time *windows* (sometimes in relation with other events) during which events must (not) take place. Note that it would be impossible to (finitely) represent this interval with ground atoms, as it corresponds to an infinite number of points.

Evaluation: Comparing directly our implementation of EC in s(CASP) with implementations in other systems is not easy: most previous systems implemented Discrete Event Calculus (DEC) and they do not support continuous quantities. One of them is F2LP [16], an ASP-based system that according to [15] outperforms *DEC reasoner* [22], reported by [15] as the more efficient SAT-based implementation. F2LP is a tool that executes DEC by turning first order formulas under the stable model semantics into a logic program w.o. constraints that is evaluated using an ASP solver.

We compare the light scenario in Fig. 4 running under s(CASP) with the F2LP translation under *clingo 5.1.1*, the current version of the state-of-the-art ASP system. Since the directive `#domain` is no longer available in *clingo*, we had to adapt the translation of F2LP adding `timestep(1..10)` and `timestep/1` to make the clauses safe (the code is also available in Appendix A). While under s(CASP) we can reason about time points in an unbounded continuous domain, the previous encoding of F2LP will make time belong to the integers from 1 to 10. Therefore, since the light is red for $t > 2, t < 3$ and for $t > 6, t < 7$, there are no integer time points from 1 to 10 when the emitted light is red. I.e., for the query `?- holdsAt(red,T)` the execution under *clingo* fails and the execution under s(CASP) returns the constraint `T#>2,T#<3` and `T#>6,T#<7`.

In order to find at what time point the light red is on under *clingo*, we had to modify the program generated by F2LP to refine the `timestep` domain with `timestep(1..10*P):-precision(P)`, where the new predicate `precision(P)` makes it possible to have a finer grain for the possible values of `timestep` by increasing the value of `P`. E.g, for `P=10` it is possible to check if the light is red at time $t = 6.9$ by querying `?- holdsAt(red,69)`, for `P=100` it is possible to check for $t = 6.99$ by querying `?- holdsAt(red,699)`, and so on. This modification, also available in Appendix B, obfuscates the resulting encoding (note that for more complex narratives it could be harder or even infeasible) and also impact negatively its performance. Table 1 shows that additional precision in the F2LP encoding (to handle each of the queries) increases the execution run-time of *clingo* by orders of magnitude. On the other hand, under s(CASP) we do not have to adapt the encoding/queries and the performance does not change.

6 Conclusions

We showed how Event Calculus can be modeled in s(CASP), a goal-directed implementation of constraint answer set programming with predicates, with much fewer limitations than other approaches. s(CASP) can capture the notion of continuous time (and, in general, fluents) in Event Calculus thanks to its grounding-free top-down evaluation strategy. It can also represent complex models and answer queries in a flexible manner thanks to the use of constraints.

The main contribution of the paper is to show how Event Calculus can be directly modeled using s(CASP), and ASP system that seamlessly supports constraints. The modeling of the Event Calculus using s(CASP) is more elegant and faithful to the original axioms compared to other approaches such as F2LP, where time has to be discretized. While other approaches such as ASPMT do support continuous domains, their reliance on SMT solvers makes their implementation really complex as associations among variables are lost during grounding. The use of s(CASP) brings other advantages: for example, the justification for the answers to a query is obtained for free, since in a query-driven system, the justification is merely the trace of the proof. Likewise, explanations for observations via abduction are also generated for free, thanks to the goal-directed, top-down execution of s(CASP).

To the best of the authors' knowledge, our approach is the only one that faithfully models continuous-time Event Calculus under the stable model semantics. All other approaches discretize time and thus do not model EC in a sound manner. Our approach supports both deduction and abduction with little or no additional effort.

The work reported in this paper can be seen as the first serious application of s(CASP) [1]. It illustrates the advantages that goal-directed ASP systems have over grounding and SAT solver-based ones for certain applications. Our future work includes applying the s(CASP) system to solving planning problems where a generated plan must obey real-time constraints.

References

1. Arias, J., Carro, M., Salazar, E., Marple, K., Gupta, G.: Constraint Answer Set Programming without Grounding. Theory and Practice of Logic Programming **18**(3-4), 337–354 (2018)
2. Balduccini, M., Magazzeni, D., Maratea, M.: PDDL+ planning via constraint answer set programming. In: 9th Workshop on Answer Set Programming and Other Computing Paradigms (October 2016)
3. Bartholomew, M., Lee, J.: System aspmt2smt: Computing ASPMT theories by SMT solvers. In: 14th European Conference on Logics in Artificial Intelligence. LNCS, vol. 8761, pp. 529–542. Springer (2014)
4. Chittaro, L., Montanari, A.: Efficient Temporal Reasoning in the Cached Event Calculus. Computational Intelligence **12**, 359–382 (1996)
5. Clark, K.L.: Negation as Failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Springer (1978)

6. Fox, M., Long, D.: PDDL+: Modeling continuous time dependent effects. In: Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space. vol. 4, p. 34 (2002)
7. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Clingo = ASP + Control: Preliminary Report. arXiv preprint arXiv:1405.3694 (2014)
8. Gelfond, M., Kahl, Y.: Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach. Cambridge University Press (2014)
9. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: 5th International Conference on Logic Programming. pp. 1070–1080 (1988)
10. Gelfond, M., Lifschitz, V.: Representing Action and Change by Logic Programs. The Journal of Logic Programming **17**(2-4), 301–321 (1993)
11. Gupta, G., Bansal, A., Min, R., Simon, L., Mallya, A.: Coinductive Logic Programming and its Applications. In: 23rd Int'l. Conference on Logic Programming. pp. 27–44. Springer (2007)
12. Holzbaur, C.: OFAI CLP(Q,R) Manual, Edition 1.3.3. Tech. Rep. TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna (1995)
13. Kowalski, R., Sergot, M.: A Logic-Based Calculus of Events. In: Foundations of knowledge base management, pp. 23–55. Springer (1989)
14. Lee, J., Meng, Y.: Answer set programming modulo theories and reasoning about continuous changes. In: IJCAI 2013. pp. 990–996 (2013)
15. Lee, J., Palla, R.: Reformulating the Situation Calculus and the Event Calculus in the General Theory of Stable Models and in Answer Set Programming. J. of Artif. Intell. Research **43**, 571–620 (2012)
16. Lee, J., Palla, R.: F2LP: Computing Answer Sets of First Order Formulas. http://reasoning.eas.asu.edu/f2lp/ (February 2019), accessed on February, 2019
17. Lifschitz, V.: What Is Answer Set Programming? In: 23rd National Conference on Artificial Intelligence. vol. 3, pp. 1594–1597. AAAI Press (2008)
18. Marple, K., Salazar, E., Gupta, G.: Computing Stable Models of Normal Logic Programs Without Grounding. CoRR **eprint arXiv:1709.00501** (2017)
19. McCarthy, J.: Circumscription - A Form of Non-Monotonic Reasoning. Artificial Intelligence **13**(1-2), 27–39 (1980)
20. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating Answer Set Programming and Constraint Logic Programming. Annals of Mathematics and Artificial Intelligence **53**(1-4), 251–287 (2008)
21. Mueller, E.T.: Chapter 17: Event calculus. In: Handbook of Knowledge Representation, Foundations of AI, vol. 3, pp. 671 – 708. Elsevier (2008)
22. Mueller, E.T.: Discrete event calculus reasoner documentation. Software documentation, IBM Thomas J. Watson Research Center, PO Box **704** (2008), http://decreasoner.sourceforge.net/, accessed on February, 2019
23. Mueller, E.T.: Commonsense reasoning: an event calculus based approach. Morgan Kaufmann (2014)
24. Shanahan, M.: The Event Calculus Explained. In: Artificial Intelligence Today, pp. 409–430. Springer (1999)
25. Shanahan, M.: An Abductive Event Calculus Planner. The Journal of Logic Programming **44**(1-3), 207–240 (2000)

A F2LP encoding of light scenario

```
timestep(0..10).

% If a light is turned on, it will be on:
initiates(turn_on,on,T) :- timestep(T).

% If a light is turned on, whether it is red or green will be released
% from the commonsense law of inertia:
releases(turn_on,red,T) :- timestep(T).
releases(turn_on,green,T) :- timestep(T).

% If a light is turned off, it will not be on
terminates(turn_off,on,T) :- timestep(T).

% After a light is turned on, it will emit red for up to 1 second
% and green after at least 1 second
trajectory(on, T1, red, T2) :-
                     timestep(T1), timestep(T2),
                     T1 < T2, T2 < T1 + 1.
trajectory(on, T1, green, T2) :-
                     timestep(T1), timestep(T2),
                     T2 >= T1 + 1.

initiallyN(on).

%% Actions
happens(turn_on,2).
happens(turn_off,4).
happens(turn_on,6).

%% Query
:- not query.
query :- holdsAt(red,_).
```

B Adapted F2LP translation of light scenario with increased precision

```
timestep(0..10*P) :- precision(P).

% If a light is turned on, it will be on:
initiates(turn_on,on,T) :- timestep(T).

% If a light is turned on, whether it is red or green will be released
% from the commonsense law of inertia:
releases(turn_on,red,T) :- timestep(T).
releases(turn_on,green,T) :- timestep(T).

% If a light is turned off, it will not be on
```

```
terminates(turn_off,on,T) :- timestep(T).

% After a light is turned on, it will emit red for up to 1 second
% and green after at least 1 second
trajectory(on, T1, red, T2) :-
                     timestep(T1), timestep(T2), precision(P),
                     T1 < T2, T2 < T1  +  (1*P).
trajectory(on, T1, green, T2) :-
                     timestep(T1), timestep(T2), precision(P),
                     T2 >= T1  +  (1*P).

initiallyN(on).

%% Actions
happens(turn_on,2*P) :- precision(P).
happens(turn_off,4*P) :- precision(P).
happens(turn_on,6*P) :- precision(P).

%% Query
:- not query.

precision(10).
query :- holdsAt(red,69).
```