*TPLP* **26** (1-2): 302–319, 2026. © The Author(s), 2026. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.

doi:10.1017/S147106842610060X First published online 15 July 2026

# *Event Calculus Meets Hybrid ASP**

ONDŘEJ VAŠÍČEK
*Brno University of Technology Faculty of Information Technology, Czech Republic*
(*e-mail:* ivasicek@fit.vut.cz)

JOAQUÍN ARIAS
*Universidad Rey Juan Carlos, Spain*
(*e-mail:* joaquin.arias@urjc.es)

JAN FIEDOR
*Brno University of Technology Faculty of Information Technology, Czech Republic*
*Honeywell International Inc., Czech Republic*
(*e-mail:* ifiedor@fit.vut.cz)

GOPAL GUPTA
*Computer Science Dept., The University of Texas at Dallas, USA*
(*e-mail:* gupta@utdallas.edu)

BOHUSLAV KŘENA
*Brno University of Technology Faculty of Information Technology, Czech Republic*
(*e-mail:* krena@fit.vut.cz)

JAKUB NĚMEC
*Faculty of Informatics, Masaryk University, Czech Republic*
(*e-mail:* jaknem@mail.muni.cz)

JAVIER ROMERO
*Institute of Computer Science, University of Potsdam, Germany*
(*e-mail:* javier.romero.davila@uni-potsdam.de)

TOMÁŠ VOJNAR
*Brno University of Technology Faculty of Information Technology, Czech Republic*
*Faculty of Informatics, Masaryk University, Czech Republic*
(*e-mail:* vojnar@fi.muni.cz)

*submitted 30 May 2026 UTC; revised 30 May 2026 UTC; accepted 1 June 2026 UTC*

* The work was supported by: project 23-06506S of the Czech Science Foundation; Horizon Europe project VASSAL (101160022); project COSASS (PID2021-123673OB-C32) funded by MCIN/AEI/10.13039/501100011033; project EVASAI (PID2024-158227NB-C32) funded by MICIU/AEI/10.13039/501100011033 and FEDER, EU; a research gift from Nexco Corp; and grant DFG SCHA 550/15, Germany.

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

**Abstract**

Event Calculus (EC) implemented in Answer Set Programming (ASP) has proven suitable for specifying requirements on safety-critical systems thanks to its elegant representation of both discrete and continuous changes and its semantic closeness to semi-formal natural language. However, continuous changes and the size of value domains of time and system properties (fluents) pose significant challenges. Grounding-based ASP solvers, for example, clingo, which implement Discrete EC (DEC), lead to combinatorial explosion in program size and inaccurate representation. The grounding-free s(CASP) does not discretize but struggles with non-termination due to its top-down execution. This paper introduces Hybrid EC, an extended axiomatization of DEC, that tackles the challenges via functional fluents and a mapping of time to abstract steps. We implement it using clingcon and clingo-lpx (Hybrid ASP systems over integers and rationals, respectively) where the value (dense) domains of fluents and time are represented as linear constraints and evaluated by external solvers, while ensuring termination whenever solutions exist. We validate both implementations on a number of examples and observe that they are unaffected by the size of the domains and that handling rationals does not impact scalability. Most importantly, the ability of clingo-lpx to handle dense domains enables accurate modeling of continuous change.

*KEYWORDS:* event calculus, answer set programming, hybrid ASP, cyber-physical systems

## 1 Introduction and motivation

Modern cyber-physical systems (CPSs) are rapidly growing in size and complexity, making rigorous verification increasingly costly. Using high-level, near-natural specifications with early verification can help mitigate this, but such requirements must first be translated into an appropriate formal language for verification.

Event Calculus (EC) has recently been advocated as a suitable formalism for this purpose, for example by Varanasi *et al.* (2022), Murugesan *et al.* (2024), and Vašíček *et al.* (2024). EC is a logical formalism for reasoning about events and change and is well suited for the given purpose due to its small semantic gap with semi-formal natural language requirements. In particular, Vašíček *et al.* (2024) illustrates this by analyzing a real-life safety-critical system with an open-source high-level specification, which led to the discovery of a number of inconsistencies and a safety violation in the specification.

However, while EC allows natural modeling of CPS specifications, the resulting models typically use large or even dense value domains and describe continuous behavior. Analyzing such models is a significant challenge for current state-of-the-art general-purpose reasoning tools for EC, which are based on *Answer Set Programming* (ASP). Grounding-based ASP solvers, for example clingo by Gebser *et al.* (2019), require discretization of dense domains, which can lead to inaccurate representation of the system. This can be avoided on a case-by-case basis by finding the right discretization (e.g., increasing the precision from seconds to milliseconds), but then the value and time domains can easily become large, making the solvers run out of resources due to an explosion in the size of the grounded program. The grounding-free ASP solver s(CASP) by Arias *et al.* (2018) does not discretize but faces non-termination issues.

To address these issues, we introduce *Hybrid EC*, an extension of the Discrete EC (DEC) axiomatization of Mueller (2014). Hybrid EC introduces the so-called

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

*functional fluents* to represent various system properties. The values of such fluents are represented using constraints. Moreover, Hybrid EC replaces the integer-time semantics of DEC with *abstract step indices* representing the sequence of system evolution. These steps are subsequently mapped to time through constraints.

We then utilize clingcon and clingo-lpx, Hybrid ASP solvers supporting linear constraints over integers and rationals, to implement the proposed Hybrid EC through the `HEC` axioms. Execution under Hybrid ASP avoids the grounding-size explosion with respect to time and value domains. More importantly, the `HEC` implementation enables accurate modeling of and reasoning about continuous change since clingo-lpx supports the representation of dense domains using rationals. This allows reasoning without discretization and eliminates inaccuracies due to discretized representations.

We have implemented and analyzed several examples from the literature to demonstrate the modeling capabilities and performance of our approach using clingcon and clingo-lpx. We compare the results with those obtained from the plain ASP solver clingo. Our hybrid approach scales independently of the size of the time and value domains, while the plain ASP approach does not. Furthermore, the use of rationals enables accurate modeling of continuous change, which was not possible previously, without affecting scalability compared to reasoning over integers.

## 2 Background

This section provides a brief overview of EC, a formalism for reasoning about events and change, initially proposed by Kowalski and Sergot (1986) and Shanahan (1997), and Hybrid ASP, an extension of ASP that incorporates linear constraints.

### *2.1 Event Calculus*

Several versions of EC have been proposed, see survey by Mueller (2008), among which the most relevant are the `EC` and the `DEC` by Mueller (2004) formulated for non-monotonic reasoning using first order logic and circumscription. EC was later reformulated by Lee and Palla (2012) in the theory of stable models and translated into ASP. Throughout the paper, we use the ASP encoding of `DEC` in code listings and as the base of our hybrid formulation of EC.

There are three basic concepts in EC: *events* representing actions that may occur in the world; *fluents* modeling time-varying properties of the world; and *timepoints* representing instants of time. Events may happen at a timepoint. Properties represented by fluents change discretely via events and stay constant between occurrences of the events due to inertia. When released from inertia, they change continuously via *(anti-)trajectories* or are non-deterministic. Predicates used in `EC` and `DEC` are listed in Table 1. A comprehensive introduction to EC can be found in Mueller (2014).

EC specifications of problem instances consist of a universal theory, a domain description, and a narrative. The theory is a conjunction of EC axioms, the domain description consists of the causal laws of the domain, and the narrative provides observations of event occurrences and about fluents. A basic example encoded using `DEC` follows:

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

Table 1. *Predicates used in* `EC` *and* `DEC`*. Based on Table 17.4 from Mueller (2008)*

| Predicate | Meaning (`E` *event*, `F` *fluent*, `T` *timepoint*) |
|---|---|
| `holdsAt(F,T)` | `F` is true at `T` |
| `happens(E,T)` | `E` occurs at `T` |
| `releasedAt(F,T)` | `F` is released from inertia at `T` |
| `initiates(E,F,T)` | if `E` occurs at `T`, then `F` is true and not released from inertia after `T` |
| `terminates(E,F,T)` | if `E` occurs at `T`, then `F` is false and not released from inertia after `T` |
| `releases(E,F,T)` | if `E` occurs at `T`, then `F` is released from inertia after `T` |
| `trajectory(F1,T1,F2,T2)` | if `F1` is initiated by an event at `T1`, then `F2` is true at `T1+T2` |
| `antiTrajectory(F1,T1,F2,T2)` | if `F1` is terminated by an event at `T1`, then `F2` is true at `T1+T2` |
| `startedIn(T1,F,T2)` | `F` is initiated by an event in the interval `(T1,T2)` *(auxiliary pred.)* |
| `stoppedIn(T1,F,T2)` | `F` is terminated by an event in the interval `(T1,T2)` *(auxiliary pred.)* |

```
time(0..100).     dom(0..10).
fluent(val(V)) :- dom(V).
initiates(inc,val(V+1),T) :- holdsAt(val(V),T).
terminates(inc,val(V),T)  :- holdsAt(val(V),T).
```

(a) Domain description

```
holdsAt(val(0), 0).       % initially zero
:- holdsAt(val(V), 0), V != 0, dom(V).
happens(inc,   10).       % increment to 1
happens(inc,   20).       % increment to 2
```

(b) Narrative

```
holdsAt(F,T+1) :-     % event effect
  happens(E,T), initiates(E,F,T),
  time(T+1).
holdsAt(F,T+1) :-     % inertia
  holdsAt(F,T), not releasedAt(F,T+1),
  not terminated1(F,T), time(T+1).
terminated1(F,T) :-   % auxiliary
  happens(E,T), terminates(E,F,T).
releasedAt(F,T+1) :- ...
  % similar to holdsAt using releases
```

(c) Relevant `DEC` axioms

Fig. 1. `DEC` encoding of the counter example.

*Example 1* (*Counter* [1])
Consider the example from Mueller (2008) that models a counter that can be incremented or reset. Figure 1(a) models a fragment (omitting the reset) of the rules of the domain, defining the fluent `val(V)`, the `inc` event, and its conditional effects on the fluent. Then, Figure 1(b) models the narrative of initial observations and event occurrences. For this narrative, `val(1)` would hold at all timepoints in the interval `(10,20]`. Finally, Figure 1(c) models the relevant `DEC` axioms defining when a fluent holds.

### *2.2 Hybrid ASP*

We rely on a basic acquaintance with ASP. The syntax of our logic programs follows the one of clingo by Gebser *et al.* (2015b); its semantics are detailed by Gebser *et al.* (2015a).

[1] We use "" to link to our repository https://github.com/ovasicek/hybrid-ec DOI 10.5281/zenodo.19217875. The left links to a traditional ASP version and the right to the Hybrid ASP version.

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

The Hybrid ASP systems clingcon and clingo-lpx[2] extend the input language of clingo by *theory atoms* representing *linear constraints* that have the form

$$\texttt{\&sum\{}\ \alpha_1\texttt{*}t_1\texttt{;}\ \ldots\texttt{;}\ \alpha_k\texttt{*}t_k\ \texttt{\}}\ \circ\ \beta$$

where $k \geq 0$, the $\alpha_i$'s and $\beta$ are numeral terms, the $t_i$'s are symbolic terms, and $\circ \in \{<, <=, =, >=, >, !=\}$. Such a theory atom represents the linear constraint

$$\alpha_1 \cdot t_1 + \ldots + \alpha_k \cdot t_k \circ \beta$$

where $t_1$, ..., $t_k$ serve as variables, and the $\alpha_i$'s and $\beta$ stand for integer constants. For convenience, $\alpha_i * t_i$ may be written as $\alpha_i$ or $t_i$, and $\beta$ may also be of the form $\alpha * t$, $\alpha$ or $t$.

In clingcon by Banbara *et al.* (2017), stable models are extended by an assignment from symbolic terms to *integers*. Theory atoms are evaluated with respect to that assignment, and symbolic terms are left open: they are not required to be founded and can therefore take any integer value without being justified by a rule. The semantics of clingo-lpx are the same as for clingcon, except that the assignment maps symbolic terms to *rationals*.[3]

These hybrid systems do not ground symbolic terms over their possible values but instead rely on dedicated algorithms for linear constraints. This reduces grounding size and often improves solving performance. Additionally, clingo-lpx extends the expressiveness of clingo by supporting reasoning about rational numbers.

*Example 2* (*Falling Object*)
Consider the example by Mueller (2014) of an object falling at constant speed. The following program represents the height of the object at two time steps (without EC):

```
1 &sum{ 0 } < time(1).
2 falling :- &sum{ (height,0) } > 0.
3 &sum{ (height,0); -2*time(1) }  = (height,1) :-     falling.
4 &sum{ (height,0) }              = (height,1) :- not falling.
```

The term `time(1)` denotes the time passed at step `1`, while `(height,0)` and `(height,1)` denote the height of the object at steps `0` and `1`, respectively. The first rule requires the `time` at step `1` to be greater than `0`. The `height` at step `0` is left open. If it is above `0`, then the second rule makes `falling` true, and consequently the third rule sets the `height` at `1` to be the initial `height` minus `2` times the time passed at step `1`. Otherwise, the fourth rule makes the `height` at `1` persist from step `0`. This program has one stable model for every value of `(height,0)` and every value of `time(1)` greater than `0`. If `(height,0)` is above `0`, then `falling` belongs to the model and the value of `(height,1)` is given by the expression `(height,0)-2*time(1)`. Otherwise, `falling` is not in the model and `(height,1)` has the same value as `(height,0)`.

## 3 Related work

Automated reasoning techniques for the EC have evolved over time. An early approach is the Prolog-based EC planner by Shanahan (2000). This was later followed by the

[2] Available at https://potassco.org/clingcon and https://github.com/potassco/clingo-lpx.
[3] The current clingo-lpx implementation does not allow theory atoms in the body, but these can be simulated by theory atoms in the head and additional regular atoms.

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

SAT-based DEC reasoner by Mueller (2014). More recently, Lee and Palla (2012) reformulated the EC in terms of ASP and provided the automated transformation tools EC2ASP and F2LP. This enabled the use of state-of-the-art ASP solvers, such as clingo by Gebser *et al.* (2019), which offer superior performance and expressiveness compared to the earlier approaches. All these approaches considered only discrete time and discrete values of fluents. This restriction is inherent to SAT- and ASP-based methods, as the ground phase requires finite domains and this precludes the representation of infinite/dense domains. Moreover, grounding can lead to the well-known grounding explosion as the size of the domain increases.

Alternative approaches for reasoning about EC, for example by Arias *et al.* (2022) and Vašíček *et al.* (2024, 2025), use s(CASP), a grounding-free, query-driven ASP solver by Arias *et al.* (2018) which supports linear constraints over rationals. To the best of our knowledge, these are the only ones for fully-automated, general-purpose, logical reasoning about EC that reasons considering continuous change while avoiding the grounding explosion and offering multiple reasoning modes, including abduction. However, as an academic prototype, s(CASP) lacks the optimized performance of clingo and, further, can suffer from non-termination problems due to its query-driven, Prolog-style execution.

There are other systems that extend ASP with linear constraints, for example those based on translation-based approaches, such as EZSMT or ASPMT2SMT, and constraint propagation approaches, such as DLVHEX[CP] or EZCSP, which each have their own advantages and disadvantages—for a comprehensive survey see the work by Lierler (2023).

Finally, there exists a wide range of specialized applications of EC, which often use their own specialized tool for the reasoning. Such tools typically offer much better performance but only for a particular class of problems or for specific reasoning tasks. For example, RTEC by Artikis *et al.* (2015) supports reasoning in real-time and is specialized in monitoring streams of events to detect composite events.

## 4 Hybrid EC

The main goal of our work is to reason about EC with continuous changes and, thus, with dense value domains of fluents and time. Traditional ASP solvers, such as clingo, fully ground the program. On the other hand, Hybrid ASP allows one to keep only part of the model grounded, while other parts can be represented by constraints, in such a way that they are not grounded, and, thus, we can use these constraints to represent dense and/or infinite domains. In this section, we present Hybrid EC, an extended axiomatization of `DEC`, a formulation introduced by Mueller (2014) that has been proven to be equivalent to the `EC` formulation when restricted to integers. The core aspect of `DEC` is that it operates *timepoint by timepoint* instead of over *spans of timepoints*, that is always referencing two consecutive timepoints `T` and `T+1`. Hybrid EC further introduces two core ideas:

1. The definition of *functional fluents* that can hold dense values since these values (and the relations among them) can be represented by constraints and, thus, can be grounding-free (Section 4.1).
2. A *hybrid representation of time* where significant timepoints are mapped to discrete steps of the system and only those are grounded, while the actual time is handled by the constraint solver and is, thus, grounding-free and can be dense (Section 4.2).

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

In addition, we present HEC, an implementation of Hybrid EC in Hybrid ASP, that builds on the ASP-based encoding of DEC proposed by Lee and Palla (2012) and utilizes the two above-mentioned ideas. The implementation incrementally executes a Hybrid ASP solver, either clingcon or clingo-lpx (Section 5), gradually increasing the number of steps in the hybrid representation of time until a model is found (Section 4.6). It utilizes the Hybrid ASP encoding of our HEC axioms (Section 4.3) along with additional rules for handling observations defined in a narrative (Section 4.4) and events triggered via continuous change or time (Section 4.5). A problem instance then needs to provide a general encoding of the domain description and a particular narrative.

### *4.1 Functional fluents*

Fluents in EC represent time-varying properties of the world: for example `falling(apple)` represents that an apple is falling, and `height(apple,Value)` represents the height of the apple within a given range of numerical values. Standard EC supports only relational fluents, that is, fluents are either true or false at any given timepoint, which is represented as `holdsAt(Fluent,Time)` or its negation. In DEC, fluents are ground atoms, for example `height(apple,1), ..., height(apple,10)`[4] that, in principle, can hold at the same time. The *situation calculus* by Lee and Palla (2012) supports functional fluents, which can take different values, for example `height(apple)=Value`. According to Mueller (2014), functional fluents can be represented by relational fluents in EC, for example, `f(X)=Y` can be encoded as `f(X,Y)`, provided that for any `X` and `T` there is exactly one `Y` such that `holdsAt(f(X,Y),T)`.

We introduce functional fluents in EC using the language of Hybrid ASP. The value of a functional fluent `F` at a step `S` is given by the symbolic term `(F,S)`. A value `Val` can be assigned to this term using the theory atom `&sum(F,S)=Val`. Recall that in clingcon and clingo-lpx, the values of symbolic terms are left open. This corresponds closely to the relational representation of DEC by Lee and Palla (2012), where fluent values are left open using choice rules of the form `{holdsAt(F,T)}`.

### *4.2 Hybrid representation of time*

DEC uses integer time steps, and all predicates are grounded to all timepoints. However, all changes in EC are caused by event occurrences, and fluents either maintain a constant value or their value changes via a trajectory between occurrences. Therefore, one can focus the reasoning only on the significant timepoints, that is, those with event occurrences, seeing them as states. We propose to treat DEC time steps as *abstract steps of the system* instead of integer increments of time. Each step corresponds to the occurrence of an event, representing the state of the system at the time of occurrence and the state transition initiated by it. The number of steps must be equal to the number of event occurrences at distinct timepoints plus one step for an implicit final state that allows us to observe the effects of the last event. Each step then has a timepoint associated with it via constraints. This means that a variable amount of time can pass between each pair of subsequent steps and that the grounding size is determined by the number of steps

[4] Each ground instance of `height(apple, Value)` is treated as a separate relational fluent.

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

```
1 &sum{(time, 0) } >= 0.                                  % step 0 is at time 0
2 &sum{(time, laststep)} = maxtime.                       % last step is at maxtime
3 &sum{(time, S1+1) } > (time,S1) :- step(S1), step(S1+1). % ordering of consecutive steps
4 % T maps to S when an event is observed at T and the constraint (time,S) = T is satisfied
5 mapEventObs(T,S) :- obs(happens,E,T), step(S), &sum{ (time,S) } = T.
6                  :- obs(happens,E,T), not mapEventObs(T,_). % all observs. must be mapped
7     happens(E,S) :- obs(happens,E,T),     mapEventObs(T,S). % version of happens for steps
```

Fig. 2. Mapping of (dense) time values to discrete steps.

instead of the size of the time domain. A limitation of this approach is that, currently, we must specify the number of steps to be used when executing the reasoning. For now, assume that we guess the right number of steps, we explain an incremental execution approach in Section 4.6.

Figure 2 shows the encoding of this hybrid representation, where the time at step `S` is denoted by the symbolic term `(time,S)`. Lines 1–3 define the time of the first and last steps and ensure that the time associated with consecutive steps is strictly increasing. Within this representation, an atom `holdsAt(F,S)` expresses that a relational fluent `F` holds at step `S`. For `S=0`, this means that `F` holds in the time interval `[0,(time,0)]`, while for `S>0`, `F` holds in `((time,S-1),(time,S)]`. Note that `(time,0)` may be greater than zero. Fluents that undergo continuous change have a specific value at each step `S`, but their value changes between steps according to the corresponding trajectory.

Figure 2 also shows how observations of event occurrences, which come as facts in the narrative of a problem instance, are mapped to time steps. An observation of an event `E` at a timepoint `T` is represented by an atom `obs(happens,E,T)` (see Section 4.4 for more details on observations). Lines 5–6 map timepoint `T` to some step `S`, and line 7 derives the atom `happens(E,S)`, which is used by the axioms introduced in the next section.

### *4.3 Modeling the axioms of hybrid EC*

This section explains how we extend the `DEC` axioms as implemented by Lee and Palla (2012) to Hybrid EC. The extension builds on the previously defined functional fluents and the hybrid representation of time as discrete steps. This proposal provides a formal foundation for representing dense domains of fluents and time within a constraint-based framework, enabling a systematic encoding of axioms that support reasoning under Hybrid EC. The resulting axioms, referred to as `HEC N` (where `N` denotes the id of the corresponding `DEC` axiom), are summarized below:[5]

- `HEC 1` redefines `stoppedIn` as in `DEC` only for relational fluents.
- `HEC 2` extends `startedIn` for both types of fluents.
- `HEC 3-4` capture the transition from gradual change of relational fluents to continuous change of functional fluents via `trajectory`/`antiTrajectory`.
- `HEC 5` extends positive inertia for values of functional fluents and `holdsAt` of relational fluents.

[5] We include the original `DEC` axioms in Appendix A.7 and the full `HEC` axioms in Appendix A.8. Appendix A is available online as *supplementary material* at the TPLP archive.

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

```
1  %% HEC 5: transformed DEC 5 to steps and added a clause for functional fluents
2  holdsAt(F,S+1) :-  holdsAt(F,S), not terminated1(F,S), not releasedAt(F,S+1), step(S+1).
3  &sum{(F,S)} = (F,S+1) :-
4    ffluent(F), step(S), not initiated1(F,S), not releasedAt(F,S+1), step(S+1).
5  %% HEC 9: transformed DEC 9 to steps and added a clause for functional fluents
6  holdsAt(F,S+1) :- happens(E,S), initiates(E,F,S), step(S+1).
7  &sum{C*V : @member(LE) = (C,V)} = (F,S+1) :- happens(E,S), initiates(E,F,LE,S), step(S+1).
8  %% Auxiliary
9  terminated1(F,S) :- happens(E,S), terminates(E,F,S).
10 initiated1(F,S) :- happens(E,S), initiates(E,F,_,S).                    % extra arg. '_'
```

Fig. 3. Axioms of `HEC 5` and `HEC 9` representing event effects and inertia.

`HEC 6` redefines negative inertia of `holdsAt` as in `DEC` only for relational fluents.
`HEC 7-8` redefines (the absence of) `releasedAt` capturing when any type of fluent is released or "un-released" from inertia by an event.
`HEC 9` extends positive effects of events via `initiates` for both types of fluents.
`HEC 10` redefines negative effects of events via `terminates` only for relational fluents.
`HEC 11-12` extends inertia for (the absence of) `releasedAt` for both types of fluents.

For brevity, we only focus on the implementation details of two most important sets of axioms: (1) In Section 4.3.1, we explain how to capture the effects of events and inertia on functional fluents. (2) In Section 4.3.2, we explain the transition from gradual change of relational fluents to continuous change of functional fluents.

### *4.3.1 Axioms for event effects and inertia*

In `DEC`, a relational fluent starts to hold after being *initiated* by an event, ceases to hold when *terminated* by an event, or persists by *inertia* unless it is *released*. This behavior is encoded by Lee and Palla (2012) using axioms `DEC 5-6` for inertia and axioms `DEC 9-10` for event effects (`DEC 5` and `DEC 9` were shown in Figure 1c). In `HEC`, these axioms must be adapted to handle both relational and functional fluents (see Figure 3). For relational fluents, this adaptation is straightforward, as we simply replace predicate `time` by `step` (lines 2 and 6 of Figure 3).

For functional fluents, the rule in lines 3–4 of `HEC 5` represents inertia. The theory atom in the head assigns to fluent `F` at step `S+1` the value of `F` at step `S` unless the fluent is initiated or released from inertia. Line 7 of `HEC 9` represents the effect of initiating fluent `F` whenever event `E` happens at step `S`. The corresponding theory atom assigns to fluent `F` at step `S+1` the value of the linear expression `LE`, given by a new predicate `initiates/4`. A linear expression `c1*v1+...+cN*vN` is encoded as a tuple `((c1,v1),...,(cN,vN))`,[6] which is unpacked by the Python external function `@member`, yielding the set of theory elements `{(c1*v1);...;(cN*vN)}`. Rules for `HEC 6` or `HEC 10` about the termination of fluents are not needed, since functional fluents have exactly one value at every time step, and initiating a new value breaks the inertia of the previous value by deriving `initiated1` (line 10).

[6] A tuple with a single element is encoded with a comma at the end, for example, `((c1,v1),)`.

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

```
holdsAt(F2,T1+T2) :-
 trajectory(F1,T1,F2,T2),
 0<T2, happens(E,T1), initiates(E,F1,T1),
 not stoppedIn(T1,F1,T1+T2), time(T1+T2).
```

(a) Original axiom DEC 3

```
&sum{C*V: @member(LE)=(C,V); R*(time,S1+S2);
      -R*(time,S1)} = (F2,S1+S2) :-
 trajectory(F1,S1,F2,(LE,R),S2),
 0<S2, happens(E,S1), initiates(E,F1,S1),
 not stoppedIn(S1,F1,S1+S2), step(S1+S2).
```

(b) Replacement axiom HEC 3

Fig. 4. Axioms for representing gradual (DEC) and continuous (HEC) change.

*Example 3* (*Counter, Example 1 cont.* ↗ ↗)
To demonstrate how to encode the effects of events on functional fluents we transform the DEC encoding of the domain description of the counter example from Figure 1 into HEC as follows (see the full encoding in Appendix A.4):

```
ffluent(val).                    % argument V moved into constraints and no domain needed
initiates(inc,val,((1,(val,S)),(1,1)),S) :- step(S).    % LE: (val,S+1) = 1*(val,S) + 1*1
```

### *4.3.2 Axioms for continuous change*

Fluents in EC can undergo continuous change via trajectories and anti-trajectories. DEC can only model gradual change over discrete steps. In contrast, Hybrid EC supports reasoning about dense domains and can faithfully model continuous change, although it is limited to monotonic linear changes.

Figure 4(a) shows axiom DEC 3 for gradual change in DEC using trajectories. Figure 4(b) shows the corresponding rule HEC 3 for continuous change, which only applies to functional fluents as there are no trajectories for relational fluents in HEC. The representation of anti-trajectories in DEC 4 and HEC 4 is analogous, and we do not discuss it further. In HEC 3, the value of a functional fluent F2 at step S1+S2 is defined by a new predicate `trajectory(F1,S1,F2,(LE,R),S2)` if an event E initiated the relational fluent F1 at step S1, and it has not been stopped since. The new atom over the predicate `trajectory` specifies the value of F2 using the tuple `(LE,R)`. The linear expression LE provides the value of F2 at the beginning of the trajectory, and R gives the changing rate during the trajectory.[7] The new atom further refers to step S1 and *duration in steps* S2 instead of timepoints. This is a key difference because the value assigned by the trajectory to F2 cannot be derived directly from the step-based duration, as the result would vary if additional steps are inserted (e.g., due to an unrelated event occurring during the trajectory). Therefore, the theory atom in the head represents the linear expression LE as in HEC 9 (unpacked by the function `@member`) but further adds `R*(time,S1+S2)` and `-R*(time,S1)` to encode the change `R*((time,S1+S2)-(time,S1))`, which refers to the actual time difference between the concerned steps.

*Example 4* (*Falling Object, Example 2 cont.* ↗ ↗)
To show the transition from gradual to continuous change, we encode the trajectory of the falling object both in DEC and in HEC as follows: (see the full encoding in Appendix A.5)

[7] R is separated from LE so that it can be used in the head of HEC 3 and for observations in Section 4.4 .

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

```
1 trajectory(falling(O),T1,height(O,H2),T2) :- holdsAt(height(O,H),T1), H2 = H-2*T2, %% DEC
2   time(T2), time(T1+T2).
3 trajectory(falling(O),S1,height(O),(((1,(height(O),S1)),),-2),S2) :-          %% HEC
4   object(O), step(S1), step(S2), step(S1+S2).
```

The trajectory starts when the fluent `falling(O)` is initiated by a `drop(O)` event, which also releases `height(O)` from inertia, and ends when `falling(O)` is terminated.

### *4.4 Modeling narratives in hybrid EC*

A narrative of a problem instance consists of observations of event occurrences and fluent values. In `HEC`, observations are represented by facts of the form `obs(O1,O2,T)` where `T` denotes a timepoint: `obs(happens,E,T)` for event occurrences, `obs(holdsAt,F,T)` and `obs(notHoldsAt,F,T)` for values of relational fluents, `obs(F,Val,T)` for values of functional fluents, and `obs(releasedAt,F,T)` and `obs(notReleasedAt,F,T)` for fluents released from inertia. We have already discussed event observations in Section 4.2. In this section, we focus on the representation of the other (fluent) observations.

The main task is to map observation timepoints to the steps used in `HEC`. Recall that fluents do not change between steps unless they are under the effect of a trajectory. As discussed in Section 4.2, the value of a fluent at step `S` represents its value over the time interval `((time,S-1),(time,S)]`. Hence, any observation at a timepoint within `((time,S-1),(time,S)]` can be checked at step `S`. Based on this, the encoding maps timepoints to their corresponding steps and checks the observations on those steps. The mapping is defined in lines 2–5 of Figure 5 using predicate `mapObs(T,S)`. The first rule covers the initial step, while the second rules handles the case where `T` falls within `((time,S-1),(time,S)]` for `S >0`. The constraint in line 7 uses this predicate to encode the observation of a relational fluent `F` holding at timepoint `T`. It forbids cases where `T` is mapped to step `S` but `F` does not hold at `S`. Other observations of relational fluents are encoded analogously.[8]

For functional fluents, the auxiliary predicate `in_ttory(F,S,R)`, defined in lines 12–13, identifies the case where fluent `F` is under a trajectory at step `S`, and extracts the rate of change `R`. If no trajectory applies to `F` at step `S`, the situation is analogous to relational fluents, and line 9 checks the observed value of `F` at the corresponding step `S`. Otherwise, line 10 computes the value of `F` at the time `T` of the observation as `(F,S)-R*((time,S)-T))`, and checks it against the observed value `V`.

*Example 5* (*Falling Object, Example 4 cont.* ↗ ↗)
Assume step `0` is mapped to time `10`, step `1` is mapped to time `20`, object `obj` has height `20` at step `0`, and height `0` at step `1`. That is, `(time,0)=10`, `(time,1)=20`, `(height(obj),0)=20`, and `(height(obj),1)=0`. Observation `obs(holdsAt,falling(obj),15)` holds because timepoint `15` is mapped to step `1`, and `falling(obj)` is true at that step. Indeed, `obj` is always falling during the time between steps `0` and `1`. Observation `obs(height(obj),10,15)` also holds. The height at step `1` is `0`, but the height at time `15` is `20-2*(20-15)=10` due to the trajectory, which is consistent with the observation.

[8] Alternatively, we could introduce new steps for observations, as for event occurrences. However, since the grounding size of `HEC` depends on the number of steps, we chose not to introduce additional steps.

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

```
% observation at T applies to a step with the closest larger time
mapObs(T,S) :- obs(O,_,T), O != happens, S = 0, &sum{( (1,(time,S )) )} >= T.
mapObs(T,S) :- obs(O,_,T), O != happens, step(S), S > 0,
                  &sum{( (1,(time,S-1)) )} < T, &sum{( (1,(time,S )) )} >= T.
:- obs(O,_,T), O != happens, not mapObs(T, _). % every non-event must have a step
% Relational fluents' observations
:- obs(holdsAt,F,T), fluent(F), mapObs(T,S), not holdsAt(F,S). % ... other obs similar ...
% Functional fluents' observations
&sum{(F,S)} = V :- obs(F,V,T), ffluent(F), mapObs(T,S), not in_ttory(F,S,_).
&sum{(F,S); -R*(time,S); R*T} = V :- obs(F,V,T), ffluent(F), mapObs(T,S), in_ttory(F,S,R).
% F2 is in a trajectory at step S1+S2 with rate R
in_ttory(F2,S1+S2,R) :- happens(E,S1), initiates(E,F1,S1), 0<S2,
  trajectory(F1,S1,F2,(LE,R),S2), not stoppedIn(S1,F1,S1+S2), step(S1+S2).
```

Fig. 5. Non-event observations.

```
:- no_jump(F,S,V), &sum{ (F,S-1) } < V, &sum{ (F,S) } > V. % cant step over V going up
:- no_jump(F,S,V), &sum{ (F,S-1) } > V, &sum{ (F,S) } < V. % cant step over V going down

obs(happens(drop(obj)), 10).
happens(hitGround(O), S) :- holdsAt(falling(O), S), &sum{ (height(O), S) } = 0.
no_jump(height(O), S, 0) :- holdsAt(falling(O), S). % no jump over height 0 while falling
```

Fig. 6. The `no_jump` constraint and a triggered event for the falling object.

### *4.5 Modeling triggered events in hybrid EC*

Special considerations must be made for *triggered events*, that is, events whose occurrences are not given as observations in the narrative but rather implied by the rules of the domain description. For such events, there will be no mapping from observations to steps (Section 2). We handle triggered events by giving the reasoning extra *free-floating steps* that will not be mapped to an event occurrence observation. The event is triggered at one of the free-floating steps which is then assigned a time according to the evaluation of linear constraints defined in the trigger rule. However, additional constraints are needed in order to ensure that the step will not be placed at a different timepoint or that the step is not taken up by another triggered event that occurs later.

The above is a common problem in hybrid systems and can be addressed using a constraint that prevents steps from jumping over particular values of interest when dealing with monotonic changes, as was discussed by Shin and Davis (2005) for SAT-based planning. In our case, this is relevant for events triggered based on continuous change of fluents or triggered after some delay. We define a predicate `no_jump(Fluent, Step, Value)` that introduces such a constraint for typical scenarios, as shown in Figure 6 (lines 1–2). The body of the rule that implies `no_jump` should mirror the body of the trigger rule.

The implicit last step (discussed in Section 4.2) is crucial to force the reasoning to check that the `no_jump` constraints are not violated after the last event occurrence (second to last step), which would signal the need for more steps.

*Example 6* (*Falling Object, Example 5 cont.*)
Consider, the falling object that is dropped from an initial height of 20 at time 10 and hits the ground when its height reaches zero. The observation of dropping the object

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

```
1 :- step(S), not significant_step(S).   % all steps must be significant
2 significant_step(S) :- S=laststep.     % last step is significant
3 significant_step(S) :- happens(E,S).   % step is significant if an event happens
```

Fig. 7. The significant step constraint.

and the triggered event of hitting the ground are encoded on lines 4–5 of Figure 6. This problem would need 3 steps in HEC: drop, hit the ground, and the last step. Reaching height zero is defined by the falling trajectory which in this case implies a trigger time of 20. Line 6 shows the necessary `no_jump` constraint saying that there must not exist two consecutive steps such that the height is less than zero at the first one and greater than zero at the second one, that is, one of the steps must have height exactly zero. Note that the object must be falling at the second step and not necessarily at the first step, as the value of a fluent at step `S` represents the time period `((time,S-1),(time, S)]`.

### *4.6 Incremental solving*

Mapping time to steps (Section 4.2) requires at least as many steps as there are observations of event occurrences at distinct timepoints. However, it might be hard to know how many triggered events will occur in a particular problem instance. We have designed the axioms in such a way that the reasoning will only produce models for the right number of steps to cover all the events that occur from time zero to the time of the implicit last step, that is, there will be no models for not enough steps and for too many steps.

This is achieved by a combination of three techniques: (1) the implicit last step, (2) the `no_jump` constraints, and (3) a significant step constraint. The `no_jump` constraints and the implicit last step (Section 4.2) ensure that no models exist when too few steps are provided. To avoid excessive floating steps when too many steps are given, we introduce the constraint encoded in Figure 7 requiring all steps to be significant. A step is considered *significant* if it involves an event occurrence or if it is the last step.

Our implementation uses an incremental execution approach to find the right number of steps automatically. A limitation of this approach is that it currently is not possible to automatically identify that there is no model for any number of steps. Instead, the script currently explores up to a given maximum number of steps as defined by the user.

## 5 Accurately representing continuous change

The use of integer time and values of fluents in DEC means that it cannot model continuous change. Instead, it models gradual change which can lead to inaccurate representation.

*Example 7* (*Falling Object, Example 6 cont.*)
Consider the falling object. If it is dropped from height 20 at time 10 and falls via a constant speed of 2 units, then it will hit the ground at time 20 with height 0. However, if it was dropped from height 21, then its height would be 1 at time 20 and –1 at time 21, that is, there would be no integer time at which the height is equal to zero.

This problem can be mitigated on a case-by-case basis by approximation (e.g., triggering before or after stepping over height zero), which leads to inaccurate representation,

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

or by increasing the precision of the discretization (e.g., multiplying the time and height by a factor of 10), which can quickly lead to very large value domains. Although a large value domain is no longer a problem for `HEC` with clingcon, it can be difficult or impossible to identify how fine the discretization needs to be for some problem instances.

Since `HEC` reasons about values of functional fluents and time using theory atoms, we can switch to reasoning using dense values instead of integers, which would be impossible without the hybrid approach. We can do this by simply replacing clingcon with clingo-lpx, which reasons about theory atoms in the domain of rational numbers (see Section 2.2).

However, one must consider the challenges that come with reasoning in dense time, such as *Zeno behavior* and various *Zeno-like phenomena*, whose causes in EC were studied by Vašíček *et al.* (2025). For example, preventing the *repeated triggering of events*, a basic modeling technique for ensuring that an event is only triggered once after its trigger condition starts to hold, becomes a fundamental problem because one may end up reasoning about the smallest timepoint `T2` such that `T2 > T1` for some given timepoint `T1`, with no such `T2` existing in a dense domain. These challenges can be addressed using the techniques proposed by Vašíček *et al.* (2025). Although a detailed discussion of this in the context of `HEC` is outside the scope of this paper and left for future work, we provide encodings of four such problems (listed in Appendix A.1).

## 6 Experimental evaluation

The key features to evaluate for Hybrid EC are: (1) its expressive power to encode and reason about continuous change in EC, and (2) improvements in time and memory scalability, particularly, with respect to the size of the time and fluent value domains.

### *6.1 Expressiveness*

We used `HEC` to encode benchmark problems from the works by Mueller (2004) and Lee and Palla (2012), additional problems from the works by Mueller (2008, 2014), relevant problems from the work by Vašíček *et al.* (2025), and some modifications to highlight the benefits of using `HEC` under rationals. The implementations for clingo, clingcon, and clingo-lpx of over 30 problems are available at https://github.com/ovasicek/hybrid-ec and archived at DOI 10.5281/zenodo.19217875; for brevity, Table 2 lists only a subset (the full table is available in Appendix A.1). The problems considered illustrate the modeling and reasoning capabilities of the compared approaches. They are grouped by their most notable features, including the reasoning mode they use, with two examples showcasing abductive reasoning. Classification of features and reasoning follows the one used by Mueller (2004, 2014). The table confirms that `HEC` with both clingcon and clingo-lpx can reason about all the examples that `DEC` with clingo handles. In addition, the use of rationals in clingo-lpx allows us to accurately reason about problems that require continuous change, that is, dense domains for time and fluent values, which is not possible with clingo and neither with clingcon.

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

Table 2. *Comparing the expressiveness of* `DEC` *and* `HEC`

| Reasoning type and notable features | Problem | `DEC` clingo | `HEC` clingcon | `HEC` clingo-lpx |
|---|---|---|---|---|
| D: state constraint | DeadOrAlive | ✓ | ✓ | ✓ |
| | Happy | ✓ | ✓ | ✓ |
| | StuffyRoom | ✓ | ✓ | ✓ |
| D: conditional effect axiom | Telephone | ✓ | ✓ | ✓ |
| | Yale | ✓ | ✓ | ✓ |
| D: effect constraint | WalkingTurkey | ✓ | ✓ | ✓ |
| M: non-determinism (release from inertia/determining fluent) | RussianTurkey | ✓ | ✓ | ✓ |
| | ChessBoard | ✓ | ✓ | ✓ |
| | CoinToss | ✓ | ✓ | ✓ |
| D: concurrent event | SuperMarket | ✓ | ✓ | ✓ |
| A: abducible event occurrence | StolenCar | ✓ | ✓ | ✓ |
| A: abd. fluent, disjunctive event trigger | BusRide | ✓ | ✓ | ✓ |
| D: compound event | Commuter | ✓ | ✓ | ✓ |
| D: causal constraint | ThielscherCircuit | ✓ | ✓ | ✓ |
| D: gradual change (integer), event trigger, release from inertia | FallingObject | ✓ | ✓ | ✓ |
| | KitchenSink | ✓ | ✓ | ✓ |
| | HotAirBalloon | ✓ | ✓ | ✓ |
| D: continuous change (rational), event trigger, release from inertia | FallingObject | × | × | ✓ |
| | KitchenSink | × | × | ✓ |
| | HotAirBalloon | × | × | ✓ |

Reasoning types: D = deduction, M = model finding, A = abduction.

### *6.2 Performance*

We now present two experiments performed on an AMD EPYC 9124 CPU (base clock of 3 GHz), measuring execution times and memory usage averaged over 10 runs with a timeout of 300 s while finding all models.[9]

*a) Domain size:* Figure 8a shows the execution time scaling for clingo, clingcon, and clingo-lpx with the domain size (time and values of fluents) using problems from the work by Lee and Palla (2012) and Example 1 the counter (full encoding is available in Appendix A.4). For plain ASP (clingo), we highlight problems that feature fluents with numerical values using purple lines since for these, the domain size increases not only in time but also in values of the fluents. The two blue lines that reach a timeout correspond to the StolenCar and the ThielscherCircuit (detailed Figures are available in Appendix A.3). The results show that the domain size indeed does not affect the Hybrid

[9] For the StolenCar problem, we search up to 10 models because abducing events leads to many models.

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

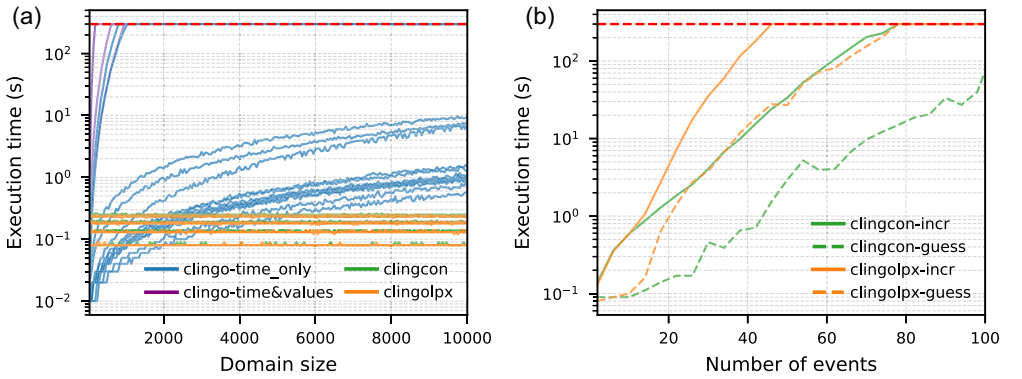



Fig. 8. (a) Scaling with the size of the domain, (b) Scaling with the number of events.

ASP execution at all, while plain ASP does not scale. This is a great advantage of using the hybrid approach, since plain ASP is fundamentally not equipped to handle large domains which greatly limits its usability in reasoning about requirements specifications of CPSs. The memory usage scaling looks very similar to the execution time scaling with the hybrid approach using up to 30 MB, and plain ASP using up to 50 GB for the curves that reach the timeout and up to 1 GB for the others (see Figure 1 of Appendix A.2).

*b) Number of events:* To measure the impact of including more events, we use Example 2 the falling object (full encoding is available in Appendix A.5), increasing the number of times the object is dropped. Figure 8b shows the execution time scaling with the number of events, which is particularly relevant for evaluating the scalability of Hybrid ASP solvers, as an increasing number of events raises both the grounding size (number of steps in `HEC`) and the complexity of the reasoning. For clingcon and clingo-lpx, we report (i) the cost of an incremental evaluation to determine the correct number of steps to model the problem and (ii) a single evaluation using the correct (guessed) number of steps. The resulting measurements show that clingo-lpx is slower than clingcon but overall scales similarly, meaning that the introduction of rationals does not significantly impact scaling. In addition, the incremental execution approach shows scaling similar to that of guessing the right number of steps. The memory usage scaling again looks similar to the execution time scaling with the maximum reaching around 300 MB. A notable difference is that clingo-lpx only uses more memory than clingcon for up to around 30 events and uses less from then on (see Figure 2 of Appendix A.2).

Clearly, unlike the domain size, the number of events does have a significant impact on the hybrid approach, which is complementary to the results of clingo (not shown in the graph not to clutter it too much), on which the number of events has almost no impact since the domain size remains unchanged. However, we note that, in the context of high-level requirements specifications of CPSs, it is often much more important to be able to handle large domains and accurately represent continuous behavior rather than to represent large numbers of events. Indeed, many errors can be detected within relatively short narratives corresponding to use cases described as a part of the specification as shown, for example in the work by Vašíček *et al.* (2024), while restricting the domain or discretization can lead to missing the error. Finally, we note that, compared to s(CASP),

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

the hybrid approach provides a significant speedup since s(CASP) reaches the timeout on the problem considered in Figure 8 at already eight events.

## 7 Conclusion

We have presented Hybrid EC, an encoding of the EC for Hybrid ASP solvers based on `DEC`, and its implementation in `HEC` axioms. The resulting framework combines ASP with linear constraints (over integers and/or rationals) that effectively avoids the grounding bottleneck when the domain of time and fluent values increases, as well as the problem of inaccurate modeling due to discretization when reasoning considering continuous changes (requiring dense domains). We have shown both of these advantages on a number of problems from the literature and provide all their encodings as modeling guides at https://github.com/ovasicek/hybrid-ec DOI 10.5281/zenodo.19217875.

This work opens several lines for future research, for example: using `HEC` to analyze real-life CPS specifications as done with s(CASP) for the PCA pump specification in the work by Vašíček *et al.* (2024), hopefully avoiding some of the scalability/termination barriers hit with s(CASP); a deeper comparison of `HEC` for clingo-lpx against s(CASP) and other Hybrid ASP solvers; utilization of clingo's multi-shot solving for more efficiency; improving the syntax of `HEC` (especially nested tuples); and finally, application of `HEC` on planning problems in the general-purpose definition language PDDL.

## Supplementary material

To view supplementary material for this article, please visit https://doi.org/10.1017/S147106842610060X

## References

Arias, J., Carro, M., Chen, Z. and Gupta, G. 2022. Modeling and reasoning in event calculus using goal-directed constraint answer set programming. *Theory and Practice of Logic Programming* 22, 1, 51–80.

Arias, J., Carro, M., Salazar, E., Marple, K. and Gupta, G. 2018. Constraint answer set programming without grounding. *Theory and Practice of Logic Programming* 18, 337–354.

Artikis, A., Sergot, M. J. and Paliouras, G. 2015. An event calculus for event recognition. *IEEE Transactions on Knowledge and Data Engineering* 27, 4, 895–908.

Banbara, M., Kaufmann, B., Ostrowski, M. and Schaub, T. 2017. Clingcon: The next generation. *Theory and Practice of Logic Programming* 17, 4, 408–461.

Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V. and Schaub, T. 2015a. Abstract Gringo. *Theory and Practice of Logic Programming* 15, 449–463.

Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T. and Thiele, S. 2015b. *Potassco User Guide*, 2nd ed. University of Potsdam.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming* 19, 1, 27–82.

Kowalski, R. and Sergot, M. 1986. A logic-based calculus of events. *New Generation Computing* 4, 67–95.

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press

Lee, J. and Palla, R. 2012. Reformulating the situation calculus and the event calculus in the general theory of stable models and in answer set programming. *Journal of Artificial Intelligence Research* 43, 571–620.

Lierler, Y. 2023. Constraint answer set programming: Integrational and translational (or SMT-based) approaches. *Theory and Practice of Logic Programming* 23, 1, 195–225.

Mueller, E. T. 2004. Event calculus reasoning through satisfiability. *Journal of Logic and Computation* 14, 5, 703–730.

Mueller, E. T. 2008. Chapter 17 event calculus. In *Handbook of Knowledge Representation*, Vol. 3 of Foundations of Artificial Intelligence, Elsevier, 671–708.

Mueller, E. T. 2014. *Commonsense Reasoning: An Event Calculus Based Approach*. Morgan Kaufmann.

Murugesan, A., Wong, I. H., Arias, J., Stroud, R. J., Varadarajan, S., Salazar, E., Gupta, G., Bloomfield, R. and Rushby, J. 2024. Automating semantic analysis of system assurance cases using goal-directed ASP. *Theory and Practice of Logic Programming* 24, 4, 805–824.

Shanahan, M. 1997. *Solving the Frame Problem - A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press.

Shanahan, M. 2000. An abductive event calculus planner. *The Journal of Logic Programming* 44, 207–240.

Shin, J. and Davis, E. 2005. Processes and continuous change in a SAT-based planner. *Artificial Intelligence* 166, 194–253.

Vašíček, O., Arias, J., Fiedor, J., Gupta, G., Hall, B., Křena, B., Larson, B., Varanasi, S. C. and Vojnar, T. 2024. Early validation of high-level system requirements with event calculus and answer set programming. *Theory and Practice of Logic Programming* 24, 4, 844–862.

Vašíček, O., Arias, J., Fiedor, J., Gupta, G., Hall, B., Křena, B., Larson, B. and Vojnar, T. 2025. On Zeno-like behaviors in the event calculus with goal-directed answer set programming. In *Proc. of ICLP'25 2025*. https://arxiv.org/abs/2601.03852, https://dblp.org/rec/journals/corr/abs-2601-03852.html?view=bibtex

Varanasi, S. C., Arias, J., Salazar, E., Li, F., Basu, K. and Gupta, G. 2022. Modeling and verification of real-time systems with the event calculus and s(CASP). In *Proc. of PADL'22*, Springer International Publishing, Cham, 181–190.

https://doi.org/10.1017/S147106842610060X Published online by Cambridge University Press