On Zeno-like Behaviors in the Event Calculus with Goal-directed Answer Set Programming*

Ondřej Vašíček[1] Joaquin Arias[2] Jan Fiedor[1,6] Gopal Gupta[3]
Brendan Hall[4] Bohuslav Křena[1] Brian Larson[5] Tomáš Vojnar[1,7]

[1]Brno University of Technology, CZ [2]CETINIA, Universidad Rey Juan Carlos, Spain [3]UT Dallas, USA
[4]Ardent Innovation Labs, USA [5]Multitude Corp., USA [6]Honeywell International s.r.o., CZ [7]Masaryk University, CZ

It has been argued that Event Calculus (EC) is suitable for modeling high-level specifications of safety-critical cyber-physical systems. The primary advantage lies in the rather small semantic gap between EC models and requirements expressed in a semi-formal natural language. Moreover, its use of continuous time and variables avoids imprecision that stems from discretization. In the past, we have shown that a goal-directed ASP system can be used for implementing these EC models. However, precise representation of time as an infinitesimally divisible continuous quantity leads to Zeno-like behaviors and to non-termination in such a system. In this work, we model a number of well-known example problems from the literature to systematically study various natural EC modeling patterns that yield these Zeno-like behaviors, and propose ways to deal with them. Moreover, we also propose a technique to automatically detect all such cases.

1 Introduction and Motivation

Requirements engineering is a wide field of research due to its various applications in critical sectors, such as medical or aerospace, where it is necessary to verify the correct operation of cyber-physical systems (CPS). Their development starts with the specification of *system requirements*, which is by many considered the most crucial part of the development process. The importance is clearly illustrated by the results of the AVSI SAVI project [19] which show that 70% of errors are introduced during the system specification and design phases, yet most of them are detected later, during the testing of system's concrete implementation. The main reason the errors are not detected earlier is that the system specification describes a whole set of feasible solutions from which one concrete solution is eventually chosen to be implemented. Such a single solution is much easier to verify compared to a potentially huge set of solutions to validate in the early phases, mainly because we can make assumptions, e.g., that the CPS operates in discrete time. When reasoning about all feasible solutions, we need to represent their real-world behavior as closely as possible which includes modeling the system behavior in continuous time and space. However, this introduces issues such as Zeno behavior where, theoretically, an infinite number of events can occur in a finite interval—an unrealistic behavior which makes modeling difficult.

As argued in [18], a promising way to reason about system requirements is to transform them into *Event Calculus* (EC) [12] and use its semantics for the verification and validation. EC is particularly suitable since its semantics follows how a human would think of the requirements, which makes the semantic gap between EC and the requirements near-zero. This allows EC to faithfully reason about the behavior defined by the requirements without being tainted by design or implementation decisions, such

*The work was supported by the project 23-06506S of the Czech Science Foundation and the FIT BUT internal project FIT-S-23-8151; grant VAE (TED2021-131295B-C33) funded by MCIN/AEI/10.13039/501100011033 and by the "European Union NextGenerationEU/PRTR", grant COSASS (PID2021-123673OB-C32) funded by MCIN/AEI/10.13039/501100011033, and by "ERDF A way of making Europe"; by US NSF Grants IIS 1910131 and grants from industry through the UT Dallas Center for Applied AI and Machine Learning.

Submitted to:
ICLP 2025

as “designing" states, transitions, or decomposition into sub-systems as is the case, e.g., with automata-based approaches. The capabilities of this approach have already been shown on a train gate controller system [17] and in our prior work [18], where we managed to discover a number of inconsistencies and a violation of a safety property in the specification of a safety-critical medical device. Both of the above works use *s(CASP)*—a goal-directed, grounding-free reasoner for Answer Set Programming (ASP) with constraints [4]. The grounding-free nature and support for constraint solving allow s(CASP) to reason about *continuous time* and *continuous quantities*, which are both crucial for accurately and realistically representing CPS behavior. The goal-directed nature further allows s(CASP) to provide *explainable results*, which is crucial when using results of automated reasoning as evidence for certification, e.g., when building assurance cases [13]. However, similar to Prolog, s(CASP) can suffer from *non-termination*, especially when reasoning in continuous time, which both of the above works struggled with.

In this work, we identify and systematically classify a number of general *Zeno-like behaviors* which cause non-termination in goal-directed reasoning under continuous time. We have implemented all relevant examples from the literature, including fairly complex ones like the water tanks [2] and real safety-critical systems [18]. We show how to address the identified classes of Zeno-like behaviors so that the non-termination issues are avoided. More precisely, by *Zeno-like* behavior, we mean behavior that makes the reasoning *explore* an infinite number of events with *infinitesimal* time intervals in between, but an infinite number of events do not *actually occur*, making it only similar to Zeno behavior. It is caused by common EC modeling patterns, e.g., preventing repeated triggering, some of which are specific to goal-directed reasoning, e.g., terminating specific trajectories. We have identified a unifying feature of all these behaviors, a *Zeno-descending chain of events* (Section 3), and propose a technique for detecting it during reasoning (Section 3.2). We introduce each of the general Zeno-like behaviors using examples from the literature or their variations (Section 4 and 5) and we propose solutions for each of them. Then, we discuss the water tanks example which combines all the behaviors as sub-problems and, in addition, exhibits real Zeno behavior (Section 6). This example illustrates all the prior solutions combined and solving the Zeno-like sub-problems leads to the real Zeno behavior being dealt with as well.

2 Background & Related Work

Zeno’s paradoxes of motion were formulated by the ancient philosopher Zeno of Elea: his Racetrack paradox (or dichotomy of motion) says that a runner will never reach the finish line because they must run half of the distance first, then half of what is remaining, etc. Zeno executions or Zeno runs are a well-known problem in timed/hybrid systems and automata. A common definition of Zeno behavior is that an infinite number of events or state transitions occur in a finite time interval. Zeno behaviors do not occur in real physical systems and are a byproduct of abstractions. According to [20], many works simply impose non-Zeno assumptions, such as [7]. A way to deal with Zeno behaviors is to remove them by modifying the system using techniques such as regularization [9]. Further research proposes techniques for detecting Zeno behavior [6, 14], or sufficient conditions for the presence (or lack of) Zeno behavior [20, 10, 8]. However, as far as we know, a systematic exploration of Zeno behaviors in EC has not been conducted yet. For example, in Section 4, we show that the bank account example [12] fundamentally has no solution in continuous time leading to Zeno-like behavior in goal-directed reasoning.

2.1 s(CASP) and Event Calculus

The s(CASP) system [4] extends the expressiveness of ASP systems, based on the stable model semantics [5], by including predicates, constraints among non-ground variables, uninterpreted functions, and, most importantly, a top-down, query-driven execution strategy. This makes it possible to return answers with non-ground variables and to compute partial models by returning only the necessary fragment of

```
1 fluent(light_on).
2 event(turn_light_on).   event(turn_light_off).
3 initiates(turn_light_on,  light_on, T).
4 terminates(turn_light_off, light_on, T).
5 % automatically created can_* rules
6 can_initiates(turn_light_on,  light_on, T).
7 can_terminates(turn_light_off, light_on, T).
```

(a) Domain model (ex1/model.pl).

```
1 initiallyN(light_on).
2 happens(turn_light_on,  10).  happens(turn_light_off, 20).
3 ?- holdsAt(light_on,    T).  % expected: T #> 10,T #=< 20
```

(b) Narrative and a query with the expected answer.

```
1 holdsAt(Fluent, T2) :-
2   T1 #> 0,   T1 #< T2,
3   can_initiates(Event, Fluent, T1),
4   happens(Event, T1),
5   initiates(Event, Fluent, T1),
6   not_stoppedIn(T1, Fluent, T2).
7
8 stoppedIn(T1, Fluent, T2) :-
9   T1 #< T,   T #< T2,
10  can_terminates(Event, Fluent),
11  happens(Event, T),
12  terminates(Event, Fluent, T).
13 not_stoppedIn(T1, Fluent, T) :- ...
```

(c) EC axioms at axioms/bec_scasp-small.pl.

Figure 1: s(CASP) encoding of Example 1 (light on/off).

a stable model to support the answer, which can include the full proof tree making them fully explainable. In s(CASP), thanks to its constructive negation, `not p(X)` can return bindings for `X` for which the call `p(X)` would have failed. Thanks to the interface of s(CASP) with constraint solvers, sound non-monotonic reasoning with constraints is possible. Like other ASP implementations and unlike Prolog, s(CASP) handles non-stratified negation and returns the corresponding (partial) stable models, e.g., for `p :- not q. q :- not p.`, under the stable model semantics there are two possible models for this *even loop*, with either `p` or `q` being true.

The Event Calculus (EC) is a formalism for reasoning about events and change [16, 12] of which there are several versions. We use the Basic Event Calculus (BEC) [15, 11]. There are three basic concepts in EC: (i) an *event* is an action that may occur in the world, (ii) a *fluent* is a time-varying property of the world, (iii) a *timepoint* is an instant of time. Events may happen at a timepoint. Fluents have a truth value at any timepoint and may have quantities associated with them. They change discretely via events or continuously via *trajectories*. An EC formulation of an example consists of a universal theory, a domain model, and a narrative. The theory is a conjunction of EC axioms, the domain model consists of the causal laws of the domain, and the narrative provides observations of event occurrences and properties of fluents. Consider the following example from [12]:

Example 1 (Light on/off): A model of a light that can be turned on and off. First, we define the domain model (Fig. 1a) where: (i) the state of the light is represented by a fluent (line 1), (ii) turning it on/off is represented by events (line 2), (iii) event effects are stated at lines 3–4, and s(CASP) generates lines 6–7 (discussed in Section 2.1.1). Second, we define a narrative (Fig. 1b). The light is initially off (line 1) and it is turned on/off at times 10/20 (line 2). We query timepoints at which the light is on (line 3) and the answer is between 10 and 20. The code is available at ex1/model.pl.[1]

The relevant EC axioms for this example are shown in Fig. 1c and implemented in axioms/bec_scasp-small.pl. The predicate `holdsAt(F, T2)` is provable for time `T2` if an event *happened* at some earlier time `T1` whose effect was to *initiate* that fluent at `T1` and the fluent was *not stopped* between `T1` and `T2`. A fluent is stopped in a time interval if some event occurs within the interval that terminates it. The full 7 BEC axioms, used in further examples, are implemented in axioms/bec_scasp.pl. Implementation of the predicate `not_stoppedIn/3` is discussed in Section 2.1.1.

2.1.1 Modeling EC Axioms in s(CASP)

Two key factors contribute to the s(CASP)'s ability to model EC in continuous time: the preservation of non-ground variables during the execution and the integration with constraint solvers. Using the trans-

[1] All files are available at https://github.com/ovasicek/iclp25-scasp-zeno/tree/master/examples DOI 10.5281/zenodo.15737913.

lation rules, introduced in [3], one can translate the BEC axioms into s(CASP) programs that follow the logic programming convention. However, special considerations must be made to avoid non-termination.

First, it is crucial to consider the relative order of `happens/2` and `initiates/3` (same for `terminates/3` and `releases/3`) in the axiom for `holdsAt/2` shown in Fig. 1c. When proving `happens/2` first, the reasoning might go through an expensive (or non-terminating) proof of an event happening only to then realize that it has no effects on the fluent. However, proving `initiates/3` first can lead to the reasoning trying to prove the effect of an infinite number of events without considering whether they happen[2]. In [18], we proposed to define predicates `can_initiates/3`, `can_terminates/3`, `can_releases/3`, and `can_trajectory/4` via preprocessing that is done by s(CASP) automatically when using the `--ec` command-line option. It defines a fact `can_initiates(E,F,T)` for each fact and/or rule `initiates(E,F,T):- some_body` (and similarly for others) without duplicates. Custom rules can be defined manually. This has proven effective in avoiding non-termination while pruning the search space.

Second, s(CASP) implements the `not` keyword by generating *dual rules*. The generation is uninformed of the semantics of the predicates and, thus, generates more rules than needed including ones that may explore unrealistic combinations of values of continuous time arguments. We use a custom implementation of the predicates `not_stoppedIn(T1, F, T2)` and `not_startedIn/3` in order to avoid non-termination. The custom implementation (in axioms/bec_scasp.pl, line 169) leverages the semantics of the predicates to be more efficient and prevent non-termination.

3 Zeno-Like Behavior in s(CASP): Zeno-Descending Chain of Events

In our experiments, the main source of non-termination are *triggered events*. An event is triggered if its occurrences are implied by rules of the domain model. In continuous time, such trigger rules can lead to an infinite number of occurrences and, thus, to an infinite number of changes to reason about.

We have encountered many examples that share the same pattern in their reasoning trees while non-terminating. The reasoning gets stuck in a loop through event trigger rules, where each iteration has a `happens(Event, TimeN)` predicate with a new time variable which is strictly less than the one from the previous iteration but otherwise both are constrained to the same interval. The tree thus contains a chain: `happens(e,T1)`, `T2 < T1`, `happens(e,T2)`, ..., `TN < ... < T2 < T1`, `happens(e,TN)`. We call such a sequence a *Zeno-descending chain of events* (or Zeno chain). A crucial aspect is that there is no time delta between the `T`'s, and so the distance between the timepoints can be infinitesimal. Reasoning with a Zeno chain results in *exploring* an infinite number of event occurrences in a finite time interval, which matches the definition of Zeno behavior. However, as we show throughout this paper, a Zeno chain can be encountered by goal-directed reasoning even in examples in which only a finite number of events occur. Therefore, not all Zeno chains represent Zeno behavior but rather only *Zeno-like* behavior.

In Section 3.2, we propose a technique for automatic detection of a Zeno chain, which allows us to alert users to its presence and to halt the non-terminating reasoning.

3.1 Example(s) of a Zeno-Descending Chain of Events

Example 2 (Simplified bank account ↗): A *simplified version* of the bank account problem [12] reasons about a single bank account and money withdrawal from it. If the balance is below the minimum of 1000, then a service fee of 10 will be charged via a triggered event. In this version, the service fee can only be charged once *per narrative* (instead of once per month). Fig. 2 shows the s(CASP) encoding of this example and an example narrative with a few queries and their expected answers. The code is available at ex2/model.pl.

[2] Such non-termination is shown in ex1/logs/non_term_summary.log using a modified version of the light example.

```
fluent(balance(M)).   fluent(noServiceFeeYet).
event(withdraw(M)).   event(serviceFee).

initiates(withdraw(X), balance(NewB), T) :-
  NewB #= OldB - X,
  holdsAt(balance(OldB), T).
terminates(withdraw(_), balance(OldB), T) :-
  holdsAt(balance(OldB), T).

terminates(serviceFee, noServiceFeeYet, T).
initiates(serviceFee, balance(NewB), T) :-
  NewB #= OldB - 10,   % service fee is 10
  holdsAt(balance(OldB), T).
terminates(serviceFee, balance(OldB), T) :-
  holdsAt(balance(OldB), T).
```

(a) Domain model (ex2/model.pl).

```
happens(serviceFee, T) :-
  holdsAt(noServiceFeeYet, T),
  B #< 1000,   % min balance is 1000
  holdsAt(balance(B), T).
```

(b) Problematic event trigger rule.

```
initiallyP(balance(10000)).
initiallyP(noServiceFeeYet).
happens(withdraw(8000), 10).
happens(withdraw(1500), 20).

?- holdsAt(balance(X),  5).  % 10000
?- holdsAt(balance(X),  15). % 2000
?- holdsAt(balance(X),  25). % 490
?- happens(serviceFee,  T).  % smallest T > 20
```

(c) Narrative and queries with expected answers.

Figure 2: s(CASP) encoding of Example 2 (simplified bank account).

```
1  happens(serviceFee, {T1~[T1 > 0, T1 =< 100]})                    % T1
2    holdsAt(noServiceFeeYet, {T1~[T1 > 0, T1 =< 100]})
3      initiallyP(noServiceFeeYet)
4      not_stoppedIn(0,noServiceFeeYet, {T1~[T1 > 0, T1 =< 100]})
5        {T2~[T2 > 0, T2 < 100]} #< {T1~[T1 > 0, T1 =< 100]}       % T2 < T1
6        happens(serviceFee, {T2~[T2 > 0, T2 < 100]})              % T2
7          holdsAt(noServiceFeeYet, {T2~[T2 > 0, T2 < 100]})
8            initiallyP(noServiceFeeYet)
9            not_stoppedIn(0,noServiceFeeYet, {T2~[T2 > 0, T2 < 100]})
10               {T3~[T3 > 0, T3 < 100]} #< {T2~[T2 > 0, T2 < 100]}  % T3 < T2
11               happens(serviceFee, {T3~[T3 > 0, T3 < 100]})        % T3
```

Figure 3: Zeno-descending chain of events in a simplified trace of Ex. 2 (full in ex2/logs/non_term.log)

All queries from Fig. 2c non-terminate due to the trigger rule of `serviceFee` (Fig. 2b). A simplified snapshot of the non-terminating reasoning tree is shown in Fig. 3. The Zeno chain starts at line 6 and continues at line 11. The reasoning enters a Zeno chain because to prove that the fee happened at some `T2`, it must consider that it could have already happened earlier at `T3`, but then it must consider an even earlier `T4`, etc. Details and solutions are discussed in Section 4.

In the following sections, we present and describe solutions to a selected class of problems that lead to non-termination through a Zeno chain. Each class of problems is introduced with an example from the literature or a variation thereof. Table 1 summarizes all the examples, alongside further unnumbered examples (available at the repository under the folder named other), and for each example shows which combination of problems (Zeno-like or Zeno behavior) is present.

3.2 Automated Detection of Zeno-Descending Chains of Events

We propose a simple yet effective technique to detect Zeno chains, made available as a feature of s(CASP) from version 0.25.06.25 at https://gitlab.software.imdea.org/ciao-lang/sCASP. It can be enabled using the `--zeno_halt` flag. When a Zeno chain is detected, s(CASP) stops and issues a warning to inform the user (avoiding non-termination). This technique, works as follows: when a predicate `happens(E,T)` is expanded, s(CASP) will check the current derivation looking for predicates `happens(E,T1)` and `happens(E,T2)`, such as: (i) all three have the same event `E`, the variables `T1` and `T2` are constrained to the same interval (`low < Tx < high`), (ii) `happens(E,T1)` was expanded earlier than `happens(E,T2)`, and (iii) `T2` is strictly smaller than `T1`, i.e., there is a constraint `T2 < T1`. E.g., the Zeno chain shown in Fig. 3 between lines 6 and 11 would be detected (see the full detection trace in ex2/logs/zeno_halt.log).

We validated the ability of this technique to (quickly) detect Zeno-like behaviors with each of the examples listed in Table 1 (their detection traces are available in the repository).

Table 1: Summary of examples and their problem composition.

		Repeated trigger	Trajectory related problems			Zeno Behaviors	
			Self-ending	Circular	With inequal.	Trivial	Paradoxes
Light on/off	Ex. 1						
Simpl. bank account	Ex. 2	*					
Bank account		*					
Light trigger off		*					
Fading light	Ex. 3		*				
Train gate			*				
Falling object			*				
Filling vessel			*				
Pulsing light	Ex. 4		*	*			
No-loss water tanks			*	*			
No-loss bouncing ball			*	*			
Simpl. water tanks	Ex. 5	*	*		*		
Blinking light	Ex. A1	*				*	
Water tanks	Ex. 6	*	*	*	*		*
Bouncing ball			*	*			*

4 Preventing Repeated Triggering of Events

A common concept in EC is preventing events from triggering repeatedly [12]. However, reasoning in continuous time introduces issues. Consider Ex. 2 from the previous section. A service fee should be charged if two conditions are met: (i) the balance is below some minimum value and (ii) the fee has not been charged yet. When money is withdrawn at time `T1`, the new balance will be provable via `holdsAt(ballance(X),T2)` for any `T2` such that `T2 > T1` (recall the axiom from Fig. 1c). Should the withdrawal make the balance go below the minimum, Condition (i) would hold for all `T2`'s (a continuous interval). Without Condition (ii), the fee would be charged infinitely many times. With Condition (ii), the first occurrence *prevents any subsequent* ones. However, there is a problem when we consider continuous time—it is impossible to find the first occurrence due to the non-inclusive lower bound, in our case `T2` such that `T2 > T1`. As a consequence, it is inherently impossible to answer the query from Fig 2c considering continuous time, causing non-termination in s(CASP) due to its attempt to find the smallest `T2` (recall Fig. 3). To solve this problem we propose two solutions:

Solution 1 (Add Epsilon): An epsilon in time is introduced and the fee is charged after a delay (see Fig. 4a). This is achieved using a new predicate `holdsAt(F, T, EPS, E)` (see its encoding in Fig. 4b). The third argument makes the predicate provable only at time `T = T1 + EPS` where `T1` is the timepoint of occurrence of the event `E`, specified by the fourth argument, that initiates the fluent `F`. The fourth argument is not strictly needed but helps avoid a slowdown from exploring irrelevant events. This solution ensures that the trigger rule proves `happens(withdraw(X), T1)` first and only then triggers the `serviceFee` at some `T2 = T1 + EPS`. Since `withdraw(X)` is not a triggered event, there only is a finite number of `T1`'s to explore. Hence, non-termination through this rule is no longer possible, and the query terminates (see ex2/logs/fix-holdsAt4.log). Introducing an epsilon here is not like discretizing since `T1` and `T2` can still take any value in the continuous domain (and there is no impact on reasoning time).

Solution 2 (Remodel Behavior): We remodel the example with continuous time in mind to charge the fee at the same time as the withdrawal and combine the effects of the two events (see Fig. 4c). The fee is charged at the originally lower bound instead of infinitely close to it, and the query terminates (see ex2/logs/fix-remodel.log). In this example, the effect of the `serviceFee` is to initiate a new balance by combining both the amount subtracted by the fee and the event which caused it. The `withdraw(X)` event has no effect if the fee is charged at the same time. This solution is closest to the original behavior, but it is only suitable for examples where the effects of the events can be combined.

```
happens(serviceFee, T2) :-
  EPS #= 1/1000000,   B #< 1000,
  holdsAt(balance(B), T2, EPS, withdraw(_)),
  holdsAt(noServiceFeeYet, T2).
```

(a) Solution 1: Add Epsilon (ex2/fix-holdsAt4.pl).

```
holdsAt(Fluent, T2, Dur, Event) :-
  T2 #= T1 + Dur, ..., happens(Event, T1),
  initiates(Event, Fluent, T1), ...
```

(b) New axiom (axioms/bec_scasp.pl, line 118).

```
happens(serviceFee, T) :- happens(withdraw(Amount), T),
  holdsAt(noServiceFeeYet, T),  NewB #< 1000,
  NewB #= OldB - Amount, holdsAt(balance(OldB), T).
initiates(withdraw(X), balance(NewB), T) :-
  not_happens(serviceFee, T), ...
terminates(withdraw(_), balance(OldB), T) :-
  not_happens(serviceFee, T), ...
initiates(serviceFee, balance(NewB), T) :-
  happens(withdraw(X), T),
  NewB #= (OldB - X) - 10, holdsAt(balance(OldB), T).
```

(c) Solution 2: Remodel Behavior (ex2/fix-remodel.pl).

Figure 4: Two solutions to the problem in Example 2 (simplified bank account).

Using the above solutions, we modeled additional examples that feature the same problem: (i) the **full bank account** models multiple accounts and transactions and adds a monthly reset for the service fee (encoding in other/1-bank_account). (ii) the **light trigger off**, a version of Ex. 1 where the light automatically turns off when it is on (encoding in other/2-light_trigger_off).

5 Trajectory-Related Problems

EC represents continuous change via *trajectories*. A trajectory, defined using `trajectory(F1,T1,F2,T2):-body`, starts when its *control fluent* `F1` is initiated at `T1` by an event and stays active until the fluent is terminated. While the trajectory is active, it is used to determine the value of the *continuous fluent* `F2` at any `T2`. Below, we discuss three non-termination problems with trajectories that we have encountered.

5.1 Self-Ending Trajectory Problem

Autotermination is a common modeling pattern where a trajectory ends via an event triggered based on the continuous value defined by the trajectory. We call such trajectories *self-ending*. Modeling these in s(CASP) can lead to non-termination. Consider the following example:

Example 3 (Fading light, cont. Ex. 1): When the light is turned on, its brightness gradually fades in from 0 until it reaches full brightness of 10. The continuous change of the fluent `brightness(X)` is modeled by a trajectory controlled by the fluent `fading_in` and ends automatically via a triggered event `fade_in_end`. The event triggers when full brightness is reached and initiates `brightness(10)` as the new discrete value of the fluent. Fig. 5 shows the new parts of the s(CASP) encoding (compared to Ex. 1) and an example narrative with a few queries. The code is available at ex3/model.pl.

All queries in Fig. 5c non-terminate, looping through the trigger rule (Fig. 5a, line 13) and the trajectory axiom (Fig. 5b, lines 1-4). The Zeno chain looks like this: `happens(fade_in_end,T1)`, `holdsAt(brightness(10),T1)`, `T2<T1`, `can_initiates(fade_in_end,brightness(10),T2)`, `happens(fade_in_end,T2)` (see ex3/logs/non_term_summary.log). This chain can be detected automatically (see ex3/logs/zeno_halt.log). The problem is that the effect of the event is considered as a way to prove its own trigger. This is a limitation of goal-directed reasoning as the example does not inherently contain any Zeno(-like) behavior. We propose one solution:

Solution 1 (Restrict HoldsAt): When reasoning about ending a trajectory, we assume that it is active; otherwise we would not need to reason about its end. We restrict that `holdsAt(brightness(10),T)` should only be proven via the trajectory controlled by the fluent `fading_in` (see Fig. 6a) using a new predicate `holdsAt(F,T,CF)` (Fig. 6b), which limits the choice of control fluents when proving the value of a fluent. his prevents the original loop, and the query terminates (see ex3/logs/fix-holdsAt3.log).

```
event(fade_in_end).   fluent(light_on).
fluent(brightness(X)).   fluent(fading_in).
initiates(turn_light_on, fading_in, T).
releases(turn_light_on, brightness(X), T).
terminates(fade_in_end, fading_in, T).
initiates(fade_in_end, brightness(10), T).
terminates(fade_in_end, brightness(X), T):- X #≠ 10.

trajectory(fading_in, T1, brightness(NewB), T2) :-
  NewB #= OldB + ((T2-T1) * 1),
  holdsAt(brightness(OldB), T1).

happens(fade_in_end, T) :- holdsAt(brightness(10), T).
```

(a) Domain model additions (ex3/model.pl).

```
holdsAt(Fluent2, T2):-  T1 #> 0,  T1 #< T2,
  can_trajectory(Fluent1, T1, Fluent2, T2),
  can_initiates(Event, Fluent1, T1),
  happens(Event, T1),
  initiates(Event, Fluent1, T1),
  trajectory(Fluent1, T1, Fluent2, T2),
  not_stoppedIn(T1, Fluent1, T2).
```

(b) Trajectory axiom (axioms/bec_scasp.pl).

```
initiallyP(brightness(0)).
happens(turn_light_on,    10).
?- holdsAt(brightness(X), 25). % 10
?- happens(fade_in_end,   T).  % 20
```

(c) Narrative and some queries.

Figure 5: s(CASP) encoding of Example 3 (fading light), extension of Example 1 (light on/off)

```
happens(fade_in_end, T) :-
  holdsAt(brightness(10), T, fading_in).
```

(a) Solution: Restrict HoldsAt (ex3/fix-holdsAt3.pl).

```
holdsAt(Fluent2, T2, Fluent1) :- ...,
  trajectory(Fluent1, T1, Fluent2, T2), ...
```

(b) New axiom (axioms/bec_scasp.pl, line 51).

Figure 6: Solution for the problem in Example 3 (fading light).

The above problem is present in all the trajectory-related examples in the rest of this paper. The solution from this section is applied in all of them by default. In addition, we have modeled the following examples that feature the same problem: (i) the **filling vessel** [16] which stops filling and starts spilling once it fills up to its maximum level (encoding in other/5-filling_vessel/). (ii) The **train gate** controller [1] models a railway crossing with trajectories for lowering/raising the crossing gates (encoding in other/3-train_gate/). (iii) The **falling object** example [12] models an object that is dropped from some height and falls to the ground (encoding in other/4-falling_object/).

5.2 Circular Trajectory Problem

This section introduces a problem of *circular dependency* between trajectories—the end of one starts the other and vice versa. The following example does not inherently contain any Zeno(-like) behavior; however, it is a problem for goal-directed, top-down reasoning in continuous time.

Example 4 (Pulsing light, cont. Ex. 3): The light fades in and out repeatedly. As soon as the light reaches maximum brightness, it will start fading out back to zero. Once zero is reached, the light will start fading in again, repeating the process. Fig. 7 shows the modification of the s(CASP) encoding (compared to Ex. 3) and a narrative with a few queries. The code is available at ex4/model.pl.

All the queries from Fig. 7b non-terminate. To prove the effect of a trajectory at some `T2`, we first need to find its start time `T1` (see the axiom in Fig. 5b). However, due to the circularity, the reasoning will proceed as follows: (i) try to prove the effect of trajectory `A` at `TA2`, (ii) need to prove its start time `TA1` such that `TA1 < TA2`, (iii) trajectory `A` can start via the end of an earlier trajectory `B`, (iv) need to prove the effect of trajectory `B` at `TB2` such that `TB2 = TA1`, which is Step (i) with `TB2 < TA2`. The loop contains the trajectory axiom (Fig. 5b lines 1–4) and the trigger rules of the trajectory start/end events (Fig. 7a lines 6–7, Fig. 6a lines 1-2). The Zeno chain looks as follows: `happens(fade_out_end,T1)`, `holdsAt(brightness(0),T1)`, `T2<T1`, `happens(fade_in_end,T2)`, `holdsAt(brightness(10),T2)`, `happens(fade_out_end,T3)` (see ex4/logs/non_term_summary.log); and it can be automatically detected (see ex4/logs/zeno_halt.log). The problem is that the reasoning never reaches the body of the `trajectory/4` predicate which means that there is no knowledge of the potential duration of the trajectory, i.e., the reasoning is exploring an infinite number of infinitely short trajectories. We propose two solutions for this problem:

```
fluent(fading_out).          event(fade_out_end).
terminates(fade_out_end, fading_out, T).
initiates(fade_out_end, fading_in, T).
trajectory(fading_out, T1, brightness(NewB), T2) :-
  NewB #= OldB - (T2-T1), holdsAt(brightness(OldB), T1).
happens(fade_out_end, T) :-
  holdsAt(brightness(0), T, fading_out).
```

(a) Encoding additions (ex4/model.pl).

```
initiallyP(brightness(0)).
happens(turn_light_on,   10).

?- happens(fade_in_end,  T). % 20
                             % and 40
?- happens(fade_out_end, T). % 30
```

(b) Narrative and some queries.

Figure 7: s(CASP) encoding of Example 4 (pulsing light), extension of Example 3 (fading light).

```
happens(fade_in_end, T) :-   Dur #= 10,
  holdsAt(brightness(10), T, fading_in, Dur).
happens(fade_out_end, T) :-   Dur #= 10,
  holdsAt(brightness(0), T, fading_out, Dur).
```

(a) Solution 1: Add Duration (ex4/fix-holdsAt4.pl).

```
holdsAt(Fluent2, T2, Fluent1, Duration) :-
  T2 #= T1 + Duration, ...
  trajectory(Fluent1, T1, Fluent2, T2), ...
```

(b) New axiom (axioms/bec_scasp.pl, line 65).

```
incr_event(fade_in_end).
incr_event(fade_out_end).

can_initiates(fade_in_end, fading_out, T) :-
  incr_happens(fade_in_end, T).

can_initiates(fade_out_end, fading_in, T) :-
  incr_happens(fade_out_end, T).
```

(c) Solution 2: Incremental Reasoning (ex4/fix-incr.pl).

Figure 8: Two solutions for the problem in Example 4 (pulsing light).

Solution 1 (Add Duration): Introduces duration information into the event trigger rules (see Fig. 8a) using a new predicate `holdsAt(F1,T,F2,Duration)` (see Fig. 8b) which is based on the `holdsAt/3` predicate from Ex. 3 and extended with a duration similarly to the `holdsAt/4` predicate from Ex. 2. It specifies a trajectory control fluent `F2` and a `Duration` for which the fluent must hold. This introduces the duration of the trajectories into the loop and the query terminates (see ex4/logs/fix-holdsAt4.log). This approach is suitable for cases where the trajectories always have the same duration; for cases with variable duration, we could specify a minimum duration or use the next solution.

Solution 2 (Incremental Reasoning): Uses forward reasoning for trajectories and accurately deals with trajectories with variable duration. The reasoning starts with the *first trajectory*. Once we prove its end, we can start reasoning whether it caused the start of some second trajectory and so on. This is achieved by taking the trajectory ending events (Fig. 8c, lines 1-2) and decoupling all their effects, other than terminating the trajectory, using `incr_happens(E, T)` (Fig. 8c, lines 4-8). The reasoning repeatedly executes queries to see if any of the events happen. If an occurrence of `E1` is found at `T1`, then a fact `incr_happens(E1, T1)` is added to the knowledge base. The next increment will then query events at times greater than `T1`. If multiple occurrences are found in an increment, only the earliest ones are saved as they could impact the occurrence of the later ones in the next increment. This process is repeated until no more occurrences are found or until a time limit is reached. The original query is then executed (see ex4/logs/fix-incr.log) with the knowledge base of occurrences of all incremental events. This approach is implemented in s(CASP) and can be enabled via the `--incremental` cmd-line option.

Using the above, we have modeled two additional examples: (i) The **no-loss bouncing ball** is a simplified bouncing ball that bounces up to the original height on every bounce (encoding in other/7-no_loss_bouncing_ball). (ii) The **no-loss water tanks** models two water tanks which are being slowly drained and one at a time is being refilled by a water input arm, while only considering narratives where the combined draining rate is the same as the input rate (encoding in other/6-no_loss_water_tanks).

5.3 Self-Ending Trajectory with End Condition Based on Inequality

In the previous trajectory examples, we considered trajectories whose end events were triggered based on equality, e.g., the light stops fading in when its brightness is *equal* to 10. In the case of strictly monotonic

```
fluent(water_left(X)).
fluent(left_filling).   fluent(left_draining).
event(switch_left).   event(switch_right).

initiates(switch_left, left_filling, T).
terminates(switch_left, left_draining, T).
terminates(switch_right, left_filling, T).
initiates(switch_right, left_draining, T).

trajectory(left_filling, T1, water_left(NewW), T2) :-
  NewW #= OldW + ((T2-T1) * 10), % rate 30 - 20
  holdsAt(water_left(OldW), T1).
trajectory(left_draining, T1, water_left(NewW), T2) :-
  NewW #= OldW - ((T2-T1) * 20), % out rate 20
  holdsAt(water_left(OldW), T1).

happens(switch_left, T):- W #=< 50, % target lvl 50
  holdsAt(water_left(W), T, left_draining).
```

(a) Partial domain model (full at ex5/model.pl).

```
initiallyP(water_left(100)).
happens(start(right),     10).
happens(switch_right,     16.25).

?- happens(switch_left,   T). % 12.5, 18.125
?- holdsAt(water_left(X), 12.5).  % 50
?- holdsAt(water_left(X), 16.25). % 87.5
?- holdsAt(water_left(X), 18.125).% 50
?- holdsAt(water_left(X), 19.5).  % 50.625
```

(b) Working narrative (ex5/model-nar1.pl).

```
initiallyP(water_left(0)).
happens(start(left),      10).
happens(switch_right,     13).

?- holdsAt(water_left(X), 13).% 30 (term.)
?- happens(switch_left, T).   % smallest T>13
?- holdsAt(water_left(X), 15).% 50
```

(c) Non-term. narrative (ex5/model-nar2.pl).

Figure 9: s(CASP) encoding of Example 5 (simplified water tanks).

```
happens(switch_left, T) :- W #= 50,  % = target lvl
  holdsAt(water_left(W), T, left_draining).
happens(switch_left, T) :-   Dur #= 1,
  W #< 50,   % < target lvl
  holdsAt(water_left(W), T, left_draining, Dur).
```

(a) Solution 1: Add Duration (ex5/fix-split_holdsAt4.pl).

```
terminates(switch_right, left_filling, T) :-
  X #> 50, % > target level 50
  holdsAt(water_left(X), T, left_filling).
initiates(switch_right, left_draining, T) :-
  X #> 50, % > target level 50
  holdsAt(water_left(X), T, left_filling).
```

(b) Solution 2: Remodel Beh. (ex5/fix-split_no_start.pl).

Figure 10: Two solutions for the problem in Example 5 (simplified water tanks).

trajectories, the brightness will be exactly 10 only at one timepoint, and so the end event can only occur once. However, sometimes an end condition with inequality may be called for. In such cases, we can face a manifestation of the repeated trigger problem from Ex. 2 where an event was defined to occur infinitely close to some non-inclusive bound. Consider the following example:

Example 5 (Simplified water tanks ↗): A simplified version of the well-known example of two leaking water tanks [2], where the tanks share one input arm that switches back and forth between them in an attempt to keep their water levels above a target level. Our simplified version only models the left tank, and any switches to the right tank come only as facts in the narrative. There is a trajectory for draining the left tank via a draining rate (20 units) and a second trajectory for filling it via a combination of an input rate (30 units) and the draining rate. A triggered event `switch_left` causes the input arm to switch to the left tank. It is triggered while the tank is draining if the current water level is *equal to or below* the target level (50 units) —the draining trajectory has an end condition with inequality. Fig. 9 shows most of the s(CASP) encoding of this example and two example narratives with a few queries. The code is available at ex5/model.pl.

Queries for the first narrative (Fig. 9b) all terminate.[3] However, the second and third query in the second narrative (Fig. 9c) do not. In the first narrative, the arm starts in the right tank meaning that the tank is initially draining, from its starting level 100, drops to the target level 50 at time 12.5, and a `switch_left` is triggered. A `switch_right` happens at time 16.25, when the tank is already filled up to 87.5 (above the target level). In the second narrative, the tank starts empty and is filling. A `switch_right` happens at time 13 while the left tank is only at level 30—still below the target level. The filling trajectory ends at time 13, but a `switch_left` is triggered as soon as possible at some `T > 13`. However, this is a problem

[3] However, a more complex impl. of `not_stoppedIn/3` is needed (in axioms/bec_scasp-interval_not.pl, line 194).

because an infinitely short draining trajectory is created. The reasoning non-terminates due to its attempt to find an event infinitely close to a non-inclusive lower bound (see ex5/logs/non_term_summary.log and detection in ex5/logs/zeno_halt.log). Such an event cannot be found in continuous time, hence, it is impossible to answer the query. This is a manifestation of the repeated trigger problem from Ex. 2 in the context of a trajectory. We propose two solutions, similar to the ones from Ex. 2. Both solutions split the reasoning into two situations: (a) the trajectory starts when its end condition is not met yet (1st narrative) and (b) the trajectory starts when its end condition already is met (2nd narrative). In Case (a), the trajectory reaches the equality first and ends before the inequality becomes applicable , therefore, the inequality is not needed. Similarly, in Case (b), the equality is never reached (it has been already exceeded) and hence not needed.

Solution 1 (Add Duration): Uses an equality-based trigger condition for Case (a) and introduces a minimum duration for the trajectory in Case (b) using the `holdsAt/4` predicate from Ex. 4 (see Fig. 10a). The fixed duration prevents the occurrence of an infinitely short trajectory, removing the non-termination problem (see ex5/logs/fix-split_holdsAt4.log). The duration is inherently required by this example and does not lead to the inaccuracies that would arise if we discretized.

Solution 2 (Remodel Behavior): Also uses an equality-based trigger condition for Case (a) but a different approach for Case (b). An infinitely short trajectory only makes an infinitely small difference, thus, should not start at all. This can be achieved by limiting all the effects of the `switch_right` event only to Case (a) (see Fig. 10b) and the query terminates (see ex5/logs/fix-split_no_start.log).

6 True Zeno Behavior

We say the prior examples contain *Zeno-like* behaviors as they do not fully meet the definition of Zeno behavior—the reasoning was *considering* an infinite number of events in finite time, but they never *needed to happen*. This section presents a true Zeno behavior in the **water tanks** [2] where an infinite number of events occurs due to delays between them getting shorter and shorter. In addition, we modeled two more examples: (i) the **bouncing ball** [9] which looses energy with each bounce. (ii) The **blinking light** which blinks with an infinitesimal delay, showing trivial Zeno behavior (detailed in A).

Example 6 (Water tanks, cont. Ex. 5): The full version of the water tanks [2], where the tanks and the arm switches are modeled via triggered events based on reasoning about both water levels. The Zeno behavior [2, 20, 9] manifests when the input flow rate is higher than the output flow rates of each tank but lower than their combined output rates—the arm is able to fill each tank, but overall the amount of water in the system is decreasing. The arm switches instantaneously and, thus, will keep both tanks from going below the target level by switching for shorter and shorter durations as they both drop closer to their target levels. Fig. 11 sketches s(CASP) encoding (full code in ex6/model.pl).

Queries shown in Fig. 11b non-terminate (see ex6/logs/non_term_summary.log) due to a combination of the problems that we have presented so far: (a) real Zeno behavior, (b) the repeated trigger problem from Ex. 2 manifested in a trajectory, (c) the self-ending trajectory problem from Ex. 3, (d) the circular trajectory problem from Ex. 4, and (e) the inequality end condition problem from Ex. 5. All the sub-problems, except (a), cause non-termination through a Zeno chain and, thus, can be detected (ex6/logs/zeno_halt.log). Like the two prior examples, the initial encoding already solves Problem (c) via the `holdsAt/3` predicate. We further propose three solutions:

Partial Solution 1 (Incremental Reasoning): This solution addresses Problem (d), which is due to reasoning limitations, and terminates up to Problem (a). We solve Problem (d) via incremental reasoning (see Fig. 12a). The incremental events are the events for switching the arm left and right. Due to the

```
fluent(right_filling). % renamed fluent(left_draining)
fluent(water_right(X)).

trajectory(right_filling, T1, water_right(NewW), T2) :-
  ... % similar to trajectory for water_left
trajectory(right_filling, T1, water_right(NewW), T2) :-
  ... % similar to trajectory for water_left

happens(switch_right, T) :-
  CurrW #=< 50,   % target level
  holdsAt(water_right(CurrW), T, left_filling).
```

(a) Domain model additions (ex6/model.pl).

```
initiallyP(water_left(100)).
initiallyP(water_right(100)).
happens(start(right),      10).
?- T #=< 19.5,
   happens(switch_left, T). % 12.5, 18.125
?- T #=< 19.5,
   happens(switch_right, T).% 16.25, 19.0625
?- holdsAt(water_left(X),  12.5).  % 50
?- holdsAt(water_right(X), 16.25). % 50
?- holdsAt(water_right(X), 19.5).  % 54.375
?- holdsAt(water_left(X),  19.5).  % 50.625
```

(b) Narrative and some queries.

Figure 11: s(CASP) encoding of Example 6 (water tanks), full version of Example 5.

```
incr_event(switch_left).   incr_event(switch_right).
can_initiates(switch_left, left_filling, T) :-
  incr_happens(switch_left, T).
can_initiates(switch_right, right_filling, T) :-
  incr_happens(switch_right, T).
?- !incr_max_time(19.5), ...
```

(a) Partial Solution 1: Incremental (ex6/partfix-incr.pl).

```
happens(switch_right, T) :-    MinD #= 1,
  D #>= MinD,    W #=< 50,   % target level 50
  holdsAt(water_right(W), T, left_filling, D).
% same for happens(switch_left, T) :- ...
```

(b) Solution 2: Add Duration(ex6/fix-holdsAt4.pl).

```
% same as Solution 1
incr_event(...).
can_initiates(...) :- incr_happens(...).

% similar to Sol 2., but split trigger rules
happens(switch_right, T) :-   MinD #= 1,
  D #>= MinD,   W #= 50,   % target level 50
  holdsAt(water_right(W), T, left_filling, D).
happens(switch_right, T) :-   D #= 1,
  W #< 50,   % target level 50
  holdsAt(water_right(W), T, left_filling, D).
% same for happens(switch_left, T) :- ...
```

(c) Solution 3: Combined (ex6/fix-incr_holdsAt4.pl).

Figure 12: Three solutions for the problem in Example 6 (water tanks).

Zeno behavior, typical narratives will not run into Problems (e) and (b) as the tanks never go below the target level (cf. ex6/logs/partfix-incr.log).

Solution 2 (Add Duration): This solution addresses the Zeno-like behavior of Problems (b) and (e), and the Zeno behavior of Problem (a). Problem (b) manifests as part of Problem (e), and so we only have to focus on the latter. We solve Problem (e) by defining a *minimum* duration for the trajectories using the `holdsAt/4` predicate (see Fig. 12b). This introduces a delay between the switch events which, additionally, solves Problem (a)—highlighting the similarity of Zeno and Zeno-like behaviors. Moreover, solving Problem (e) via a minimum duration is also an alternative solution to Problem (d), meaning that incremental reasoning is no longer necessary for termination (see ex6/logs/fix-holdsAt4.log). However, a smaller value of the minimum negatively impacts reasoning time as it allows more iterations of the loop.

Solution 3 (Combined): This solution combines Solutions 1 and 2 (see Fig. 12c and ex6/logs/fix-incr_holdsAt4.log), while Solution 1 is only a partial solution, the combination of both solutions leads to better performance (from 30 min to 10 seconds).

Note that other Zeno behaviors may require still other solutions. For example, a minimum duration for the trajectory of a **bouncing ball** results in it falling through the floor. In this case, a suitable solution is to make it stop bouncing when the last bounce was shorter than a minimum duration or to introduce a flat loss of velocity on each bounce (modeled in other/8-bouncing_ball).

7 Conclusion and Summary

We have implemented a novel technique, made available as a feature of s(CASP), that automatically detects Zeno-like behaviors based on the presence of a Zeno-descending chain of events. We have further identified and characterized a set of non-termination problems, caused by Zeno-like behaviors, that can be encountered in common Event Calculus modeling tasks in continuous time and have proposed four

Table 2: Summary of the proposed solutions and their applicability to the presented problems.

		Repeated trigger	Trajectory related problems			Zeno Behaviors	
			Self-ending	Circular	With inequal.	Trivial	Paradoxes
Automated Detection	(Sect. 3.2)	✓	✓	✓	✓	✓	
Restrict HoldsAt	(Ex. 3, 4, 5, 6)		✓				
Incremental Reasoning	(Ex. 4, 6)			✓			
Add Epsilon / Duration	(Ex. 2, 4, 5, 6, A1)	✓		✓	✓	✓	✓
Remodel Behavior	(Ex. 2, 5)	✓			✓		✓

types of solutions that can be used to solve them without discretizing. We believe that this set of problems (and solutions) is enough to cover a wide range of real use cases, such as a safety-critical system [18].

As shown in Table 2, the detection technique works for all classes of problems, except for nontrivial Zeno behaviors because those feature a descending chain of events with delays between them getting shorter and shorter (instead of infinitesimally small). Additionally, it summarizes the proposed solutions, in which examples they have been applied, and which classes of problems are solved by each of them. The **Restrict HoldsAt** solution uses external knowledge of the example to guide EC reasoning by restricting the search rule to chose a specific clause of the predicate `holdsAt`. **Incremental Reasoning** addresses circularity via forward reasoning which is particularly suitable for EC due to its need to reconstruct history. The **Add Epsilon / Duration** solution introduces a time epsilon into the loop, replacing infinitesimal time steps without causing inaccurate behaviors (as would be the case if we discretized). The **Remodel Behavior** solution tunes the model of the example for reasoning in continuous time to avoid infinitesimal time steps while preserving the intended behavior.

References

[1] Rajeev Alur & David L. Dill (1994): *A Theory of Timed Automata*. *TCS* 126, pp. 183–235, doi:10.1016/0304-3975(94)90010-8.

[2] Rajeev Alur & Thomas A. Henzinger (1997): *Modularity for Timed and Hybrid Systems*. In: *CONCUR*, *LNCS* 1243, Springer, pp. 74–88, doi:10.1007/3-540-63141-0_6.

[3] Joaquín Arias, Manuel Carro, Zhuo Chen & Gopal Gupta (2022): *Modeling and Reasoning in Event Calculus using Goal-Directed Constraint Answer Set Programming*. *TPLP* 22(1), pp. 51–80, doi:10.1017/S1471068421000156.

[4] Joaquín Arias, Manuel Carro, Elmer Salazar, Kyle Marple & Gopal Gupta (2018): *Constraint Answer Set Programming without Grounding*. *TPLP* 18(3-4), pp. 337–354, doi:10.1017/S1471068418000285.

[5] Michael Gelfond & Vladimir Lifschitz: *The Stable Model Semantics for Logic Programming*. In: *ICLP*, pp. 1070–1080, doi:10.2307/2275201.

[6] Rodolfo Gómez & Howard Bowman (2007): *Efficient Detection of Zeno Runs in Timed Automata*. In: *FORMATS*, *LNCS* 4763, Springer, pp. 195–210, doi:10.1007/978-3-540-75454-1_15.

[7] Thomas A. Henzinger (1996): *The Theory of Hybrid Automata*. In: *LICS*, IEEE CS, pp. 278–292, doi:10.1109/LICS.1996.561342.

[8] Frédéric Herbreteau & B Srivathsan (2013): *Coarse abstractions make Zeno behaviours difficult to detect*. *Logical Methods in Computer Science* Volume 9, Issue 1:6, doi:10.2168/LMCS-9(1:6)2013.

[9] Karl Henrik Johansson, Magnus Egerstedt, John Lygeros & Shankar Sastry (1999): *On the regularization of Zeno hybrid automata*. *Systems & control letters* 38(3), pp. 141–150, doi:10.1016/S0167-6911(99)00059-6.

[10] Andrew G. Lamperski & Aaron D. Ames (2013): *Lyapunov Theory for Zeno Stability*. *IEEE TAC* 58, pp. 100–112, doi:10.1109/TAC.2012.2208292.

[11] Erik T. Mueller (2008): *Chapter 17 Event Calculus*. In: *Handbook of Knowledge Representation, Foundations of Artificial Intelligence* 3, Elsevier, pp. 671–708, doi:10.1016/S1574-6526(07)03017-9.

[12] Erik T. Mueller (2014): *Commonsense Reasoning: An Event Calculus Based Approach*. Morgan Kaufmann, doi:10.1016/B978-0-12-801416-5.00002-4.

[13] Anitha Murugesan, Isaac Hong Wong, Joaquín Arias & et. al. (2024): *Automating Semantic Analysis of System Assurance Cases Using Goal-Directed ASP*. *TPLP* 24(4), pp. 805–824, doi:10.1017/S1471068424000425.

[14] Jonas Rinast & Sibylle Schupp (2012): *Static Detection of Zeno Runs in UPPAAL Networks Based on Synchronization Matrices and Two Data-Variable Heuristics*. In: *FORMATS*, Springer, Berlin, Heidelberg, pp. 220–235.

[15] Murray Shanahan (1996): *Robotics and the Common Sense Informatic Situation*. In: *ECAI*, Wiley, pp. 684–688.

[16] Murray Shanahan (1997): *Solving the frame problem: a mathematical investigation of the common sense law of inertia*. MIT Press.

[17] Sarat Chandra Varanasi, Joaquín Arias, Elmer Salazar & et. al. (2022): *Modeling and Verification of Real-Time Systems with the Event Calculus and s(CASP)*. In: *PADL*, Springer International, doi:10.1007/978-3-030-94479-7_12.

[18] Ondrej Vasicek, Joaquin Arias, Jan Fiedor & et. al. (2024): *Early Validation of High-Level System Requirements with Event Calculus and Answer Set Programming*. *TPLP* 24(4), pp. 844–862, doi:10.1017/S1471068424000280.

[19] Don Ward (2012): *AVSI's System Architecture Virtual Integration Program: Moving SAVI to the Launch Pad*. In: *15th Annual NDIA Systems Engineering Conference*. Available at https://ndia.dtic.mil/wp-content/uploads/2012/system/ttrack914871.pdf.

[20] Jun Zhang, Karl Henrik Johansson & et. al. (2001): *Zeno hybrid systems*. *IJRNC* 11(5), pp. 435–451, doi:10.1002/rnc.592.

A Trivial Zeno Behavior: Zero Delay Circular Events

This appendix presents an example that did not quite fit into the paper. The example shows trivial Zeno behavior but it is a modeling pattern that is easily encountered in EC. When reasoning in continuous time, it is easy to introduce event trigger rules which will cause an infinite number of events to occur in a finite time interval. Events have no duration and their effects are instantaneous, unless we explicitly introduce some duration or delay. Consider the following example:

Example A1 (Blinking light, cont. Ex. 1): Both the events for turning the light on and off were changed into triggered events. The trigger rules define that the light should be turned off when it is on and vice versa. However, the encoding does not define a frequency for the blinking. The triggered events are defined to turn the light off/on *as soon as* it is on/off. Fig. 13 shows the encoding additions for this example (compared to Ex. 1 and an example narrative with a few queries. The code is available at apx1/model.pl.

All the queries in Fig. 13b will non-terminate and, furthermore, their expected answers are unknown. There are two problems: (i) two instances of the repeated trigger problem (from Ex. 2) one for each triggered event and (ii) trivial zeno behavior since the light is defined to blink infinitely quickly. Problem (ii) means that while the light is blinking we are unable to conclude whether it is on or off. The problem manifests as a circular dependency between the two triggered events without any delay in between (see apx1/logs/non_term_summary.log) and it can be automatically detected (see apx1/logs/zeno_halt.log).

We present our solution in Fig. 14. To address Problem (i) we utilize a simplified version of the first solution from Ex. 2. The second one is not suitable here because the turn on/off events have opposite effects. Solving Problem (i) leads to Problem (ii) being solved as well.

The solution (Fig. 14a) adds a delay to the event trigger rules using a new predicate `holdsAt/3` (Fig 14b), which is the same as the `holdsAt/4` introduced in Ex. 2 except without the fourth argument, which is not needed here (there is only one event to consider for each rule).

To address Problem (ii), the Zeno behavior, we need to define some blinking frequency for the light. Notice that the solution to Problem (i) already introduced a delay between the events and, thus, it solves this problem as well. This shows the similarity between Zeno behaviors and the Zeno-like behaviors that we have presented in this paper.

```
happens(turn_light_off, T) :-
  holdsAt(light_on, T).
happens(turn_light_on, T) :-
  not_holdsAt(light_on, T).
```

(a) Domain model additions (apx1/model.pl).

```
happens(turn_light_on,     10).
?-     holdsAt(light_on,   15). % ? (unknown)
?- happens(turn_light_off, T). % ? (unknown)
?- happens(turn_light_on,  T). % 10 and ? (unknown)
```

(b) Narrative and queries with unknown answers.

Figure 13: s(CASP) encoding of Example A1 (blinking light), based on Example 1 (light on/off).

```
happens(turn_light_off, T) :- holdsAt(light_on, T, 1).
happens(turn_light_on, T) :- not_holdsAt(light_on, T, 1).
```

(a) Solution (apx1/fix-holdsAt3.pl).

```
holdsAt(Fluent, T2, Dur) :- T2 #= T1 + Dur,
  ..., happens(Event, T1), ...
```

(b) New axiom (axioms/bec_scasp.pl, line 80).

Figure 14: Solution for the problem in Example A1 (blinking light).