Modeling and Verification of Timed Systems with the Event Calculus and s(CASP)

Sarat Chandra Varanasi[1], Brendan Hall[2], Joaquín Arias[3], Elmer Salazar[1], Fang Li[1], Kinjal Basu[1], Kevin Driscoll[2] and Gopal Gupta[1]

[1]*The University of Texas at Dallas, Richardson, USA*
[2]*Honeywell Advanced Technology, Plymouth, USA*
[3]*Universidad Rey Juan Carlos, Madrid, Spain*

Abstract

We model the well-known Train-Gate-Controller (railroad crossing problem) system in Event Calculus using Goal-Direct Answer Programming realized via the s(CASP) system. Our paper illustrates the ease with which such a cyber-physical system's requirements specification is modeled and its properties verified relative to prior assumptions. Event calculus allows for succinct modeling of a dynamic system due to the near-zero semantic gap between the system's requirements specification and their event calculus encoding. This is to be distinguished from automata-theoretic approaches which have to explicitly encode the notion of state and define explicit transitions between states. Further, Event Calculus is naturally expressed in s(CASP) without need for discretization of continuous physical quantities including time. This is due to the goal-direct answer set semantics of s(CASP) combined with constraint solving over reals. Continuous properties require no discretization unlike other approaches to model Event Calculus in SAT-based Answer Set solvers.

1. Introduction

Cyber-physical systems are ever increasing in their prominence in our day-to-day lives. Much research has been published towards modeling and verifying properties of these systems. Primarily timed-automata approaches have been studied and used on industrial scale applications.[1, 2]. Timed-automata approaches require an explicit notion of state and transitions between states using clock variables. However, there have been cyber-physical systems modeled as constraint logic programs, where there is little semantic gap between the logic programs and the intended cyber-physical system that is modeled [3]. Techniques based on coinductive contraint logic programming (Co-CLP) have also been applied in verifying properties of timed-automata [4, 5]. The Co-CLP techniques to study timed systems culminated in the development of Goal-Directed Answer Set Programming [6]. More recently, the well-known Event Calculus formalism has been used [7] along with powerful reasoning supported in Answer Set Programming. More importantly, the work of Arias et. al extends prior work on Co-CLP to support natural reasoning of Cyber-physical systems, in the language of Event Calculus. Theirs is the first work to use the s(CASP) system to model Event Calculus along with abducive reasoning supported in Answer Set Programming. The s(CASP) system has also been used in knowledge-based methods

CEUR Workshop Proceedings (CEUR-WS.org)

that analyze faulty requirements in avionics software systems[8]. In this paper, we model the well-known Train-Gate-Controller system in Event Calculus. We model the entire system in s(CASP) and check the safety and liveness of the system, with prior knowledge of the physical properties such as train speed, the response time of the controller and the rate at which the gate rotates.

2. Background

2.1. Answer Set Programming

Answer Set Programming (ASP) is a declarative knowledge representation and reasoning paradigm widely used in AI and Combinatorial Search Problems [9]. Every rule in an ASP program is of the form: $\{p \leftarrow q_1, q_2, ...q_n, not\ r_1, not\ r_2, ...not\ r_m\}$ where $n \geq 0, m \geq 0$. Intuitively, the generic rule says, infer literal p to be true if the *positive* literals occurring in the rule $\{q_1, q_2, ...q_n\}$ are true and the *negative* literals $\{r_1, r_2, ...r_m\}$ occurring in the same rule are not provable. The *not* operator is the *negation-as-failure* operator which allows us to infer `not s` as true if literal `s` is not provable in a given program. In addition to the above rule form, ASP also allows the literal p to be absent which gives rise to headless rules. Thus, a rule of the form $\{\square \leftarrow q\_1, q\_2, ..., q_n, not\ r_1, not\ r_2, ...not\ r_m\}$ asserts that the conjunction of all the literals in the rule cannot all be simultaneously true. This rule form is also referred as a *constraint* as it constrains the truth value of its literals. The set of literals satisfiable in a given ASP program constitute its answer sets (or stable models). The answers sets are computed according to the stable model semantics [10]. Most of the answer set solvers are model-theoretic and use SAT-solving to find models of a given program[11]. Solvers such as *clingo*, ground the ASP program into literals and perform state-of-the-art SAT-solving to find stable models.

2.2. Goal-Directed Answer Set Programming

Model-theoretic semantics of ASP and their associated solvers are widely used for several industrial applications [11]. However, first order answer set programs with function symbols are infinite and are not finitely groundable. Goal-Directed Answer Set Programming enables computation of answer sets for programs that are not finitely groundable [12, 13]. The s(CASP) system and its predecessor s(ASP) are answer set solvers that compute stable models without grounding. The s(CASP) system does not compute the entire model, rather computes partial models given a query say `?- q(X)`. If the query `q(X)` succeeds, the s(CASP) system produces all the consistent literals that support the `q(X)` in the given ASP program while also producing a justification tree for `q(X)`. Thus s(CASP) while producing partial models, provides a proof-theoretic semantics for ASP. Due to the query driven nature of s(CASP), we can also query `?- not q(X)`. Then, s(CASP) produces bindings for `X` if `q(X)` is not provable in the given ASP program. To achieve this without grounding, s(CASP) uses *constructive negation* and *completion semantics* for the *not* operator [12]. s(CASP) also provides constraint solving over reals which allows us to precisely model physical quantities that are continuous including dense time. Details about the s(CASP) system can be found elsewhere [13].

2.3. Modeling Cyber-Physical Systems with Event Calculus

Event Calculus [14] is a logical formalism used to describe dynamic domains in terms of fluents and events in the domain. Fluents are properties of the system that change over time while events *cause* changes to fluents. Also, the Event Calculus provides axioms to explicitly capture the *frame problem*. We say `holdsAt(F, T)` in EC, if the fluent `F` is true at time `T`. Further, we use `initiates(E, F, T)` to denote that fluent `F` is enabled by event `E` if `E` occurs at time `T`. Also, fluent `F` is captured by commonsense law of inertia. Similar to `initiates/3`, we have `terminates(E, F, T)` and `releases(E, F, T)` to mean, if `E` occurs at time `T`, then `F` ceases to hold and released from commonsense law of inertia respectively. To describe initial states of these dynamic systems, EC uses `initiallyP(F)` to denote that fluent `F` is true initially. Similarly `initiallyN(F)` is used to depict fluent `F` is false to begin with. Finally, EC allows us to express continuously changing fluents using *trajectories*. In EC, `trajectory(F1, T1, F2, T2)` implies that fluent `F2` is true at `T2` if `F1` is initiated by some event at `T1`. These notions, couple with event calculus axioms give rise to powerful modeling capabilities of cyber-physical systems. In this paper we focus on the Basic Event Calculus (BEC) axioms [15]. There are however more general versions of Event Calculus in literature. In the context of cyber-physical systems, fluents correspond to state of the sensors or components while the events represent actuator actions. This allows for a natural and succint mapping of a cyber-physical system description into event calculus. For example, consider the reactor temperature control system with a reactor core and two control rods similar to [5]. The temperature of reactor core should be within a certain threshold θ_{low} to θ_{high}. Initially the temperature of the core is θ_{low}. If rod1 (rod2) is inserted into the core, then temperature of the core decreases at a certain rate r_1 (r_2). This situation is easily expressed in EC as follows:

```
fluent(reactor_core_temperature(T)).
fluent(rod_one_in).
fluent(rod_two_in).
event(insert_rod_one).
event(insert_rod_one).
initiates(insert_rod_one, rod_one_in, T).
initiates(insert_rod_two, rod_two_in T).
initiallyN(rod_one_in). initiallyN(rod_two_in).
initiallyP(reactor_core_temperature(tlow).
trajectory(rod_one_in, T1, reactor_core_temperature(Temp2), T2) :-
           holdsAt(reactor_core_temperature(Temp1),
           Temp2 #= Temp1 - (T2 - T1) * R1.
trajectory(rod_two_in, T1, reactor_core_temperature(Temp2), T2) :-
           holdsAt(reactor_core_temperature(Temp1),
           Temp2 #= Temp1 - (T2 - T1) * R2.
```

Now, inferences about the behavior of the sytem can be made by combining the above rules with the axioms of EC. We show EC axioms in s(CASP) in the following.

2.4. Event Calculus using Goal-directed s(CASP) system

We use the axioms from Basic Event Calculus. We do not show the classical first-order logic encoding of the axioms, rather we directly map the axioms into s(CASP) code as shown below.

```
%% BEC Axiom 1
stoppedIn(T1, F, T2) :-
   T1 #< T, T #< T2,
   terminates(E, F, T),
   happens(E, T).

stoppedIn(T1, F, T2) :-
   T1 #< T, T #< T2,
   releases(E, F, T),
   happens(E, T).

%% BEC Axiom 2
startedIn(T1, F, T2) :-
   T1 #< T, T #< T2,
   initiates(E, F, T),
   happens(E, T).

startedIn(T1, F, T2) :-
   T1 #< T, T #< T2,
   releases(E, F, T),
   happens(E, T).

%% BEC Axiom 3
holdsAt(F2, T2) :-
   initiates(E, F1, T1),
   happens(E, T1),
   trajectory(F1, T1, F2, T2),
   not stoppedIn(T1, F1, T2).

%% BEC Axiom 4
holdsAt(F, T) :-
   0 #< T,
   initiallyP(F),
   not stoppedIn(0, F, T).

%% BEC Axiom 5
-holdsAt(F, T) :-
   0 #< T,
   initiallyN(F),
   not startedIn(0, F, T).

%% BEC Axiom 6
holdsAt(F, T2) :-
   T1 #< T2,
   initiates(E, F, T1),
   happens(E, T1),
   not stoppedIn(T1, F, T2).

%% BEC Axiom 7
-holdsAt(F, T2) :-
    T1 #< T2,
    terminates(E, F, T1),
    happens(E, T1),
    not stoppedIn(T1, F, T2).
```

Axioms 1 and 2 are abstractions. They capture the notions that a fluent F may be stopped (started) in a time interval (t_1, t_2) by some events that initiate (terminate) and occur in the interval (t_1, t_2). Informally, Axioms 6 and 7 say that a fluent `F` starts to hold (not hold) immediately, provided an event `E` that initiates `F` (terminates `F`) occurs, and, the fluent `F` is itself is not stopped (not started) from there on. Axioms 4 and 5 capture the persistence of a fluent `F` that holds (not holds) initially and is not stopped (started) in the time interval $(0, t)$. Here, 0 is used to denote beginning of time. Axiom 3 enables a fluent `F2` to be depend on a fluent `F1` and vary according to some function of `F1` , `T1` and `T2`. The direct mapping of EC Axioms is possible due to s(CASP) capability to support continuous time. Further, the semantics of *not* operator is based upon Clarke's completion semantics for *negation-as-failure* [16, 17]. Due to the completion semantics, dual rules are generated for *not stoppedIn(T1, F, T2)* and *not startedIn(T1, F, T2)*. These dual

rules enable the circumscription of the consequences of events as required by EC semantics. Therefore, to the best of our knowledge, s(CASP) is the only logic programming system that encodes EC with dense time and provides a faithful implementation of circumscription of EC axioms [18]. This is to be distinguished from SAT-based ASP solvers that can only reason over a discretized version of time. Also, other logic programming systems do not implement a robust *negation-as-failure* operator with completion semantics [19]. Rigorous treatment of EC translation into s(CASP) can be found elsewhere [18].

3. Modeling the Train-Gate-Controller system in Event Calculus

The train-gate-controller is a cyber-physical system commonly used to study modeling and verification of properties of the system [3]. The system consists of a train in motion, passing through the gate area. The gate-controller should signal gate closure in a timely fashion. As we present next, s(CASP) allows for succinct modeling of the train's motion, the movement of the gate and the controller behaviour. Due to the availability of constraint solving over reals, continuous properties such as uniform motion are easily expressed.

Assume that the train changes its position uniformly at a rate of 10 units per second. The gate area is at position 40. Once the train reaches the gate area, we consider the train being *in* the gate area. Initially the gate is open and is inclined vertically at an angle of zero degrees. The controller should signal the *closing* of the gate before the train arrives in the gate area. The gate also uniformly changes its angle of inclination when it is in motion. When the *gate angle* becomes 90 degrees, the gate is closed and inclined horizontally. We next provide the fluents and events associated with this cyber-physical system.

```
fluent(passing)  % Train is passing through the gate area
fluent(leaving)  % Train has exited the gate area and is leaving
fluent(position(X)) % Train is some position X
fluent(gate_angle(A)) % Gate is inclined vertically at an angle A
fluent(opened) % The gate is completely opened
fluent(closed) % The gate is completely closed

event(train_in)  % Controller detects that the train is in the gate area
event(signal_lower) % Controller signals gate closure
event(signal_raise) % Controller signals that the gate be raised
event(gate_close) % Signifies the gate is completely closed
event(gate_open) % Signifies the gate is completely open
event(train_exit) % Train has exited the gate area
```

The causal effects of the events in the system are as follows. They should be straightforward to follow.

```
initiates(in,passing,T).
initiates(lower,lowering,T).
initiates(close,closed,T).
initiates(exit,leaving,T).
initiates(raise,rising,T).
initiates(open,opened,T).
```

```
terminates(lower,opened,T).
terminates(close,lowering,T).
terminates(exit,passing,T).
terminates(raise,closed,T).
terminates(open,rising,T).
```

In the following, `train_speed(S)`, `angle_lower_rate(L)`, `angle_rise_rate(R)` denote respectively, that the speed of the train is S, the rate at which the gate lowers is L and the rate at which the gate rises is R. We now describe the conditions under which various events happen. The motion of the train itself is modeled as a trajectory. This is show below:

```
trajectory(started, T1, position(X), T2) :-
       train_speed(S), T2 #> T1, X #= (T2 - T1) * S.
```

Similarly, the change in inclination of the gate angle is also modeled as a trajectory, depending upon whether `event(lower)` or `event(raise)` happen. If the gate is lowering (rising), then the gate inclination steadily decreases (increases)[1].

```
gate_angle_lower(A, T2) :-
        happens(lower, T),
        angle_lower_rate(L),
        T2 #> T,
        A#=(T2-T1)*L.
gate_angle_rise(A, T2):-
        happens(raise, T),
        angle_rise_rate(R),
        T2 #> T,
        A #= 90 - (T2-T1)*R.
```

The events mentioned previously happen when the fluents cross a certain threshold. For example, we consider train to be in the gate area when it has reached a position value = 10. Similarly, the controller signals `lower_gate` when the train position crosses value = 5. Similarly, exit is signalled when the train crosses position = 20. The gate is completely open (close) when its vertical angle decreases (increases) to 0 degrees (90 degrees). Finally, the controller signals gate rising when it detects that the train is leaving the gate area, immediately after passing through the gate area. All transitions in the train position, gate angle are resolved at a sampling window of 0.1 s. That is, the controller can detect changes in continuous quantities at a temporal precision of 0.1 s. This is a reasonable assumption made to make the controller behave in a realistic manner. If we used a temporal precision of 0, then the controller can detect instantaneous changes in continuous values, which is impossible in a real-world system. We use the `infimum` on the 0.1 second interval, to signify the precise instance when the transition of train position or gate angle crosses a threshold.

```
happens(train_in, T) :-
    holdsAt(position(X1), T1),
    holdsAt(position(X2), T2),
    X1 #< 10, X2 #>= 10,
    sampling_window(W),
    T2 #< T1 + W, T2 #> T1,
    infimum(T2, T).
happens(lower_gate, T) :-
        holdsAt(position(X1),
        ↪ T1),
    holdsAt(position(X2), T2),
    X1 #< 5, X2 #>= 5,
```

[1]We treat `gate_angle_lower` and `gate_angle_rise` as derived fluents. They can also be modeled as trajectories

```
    sampling_window(W),
    T2 #< T1 + W, T2 #> T1,
    infimum(T2, T).
happens(gate_close, T) :-
    gate_angle_lower(A1, T1),
    gate_angle_lower(A2, T2),
    A1 #< 90, A2 #>= 90,
    sampling_window(W),
    T2 #< T1 + W, T2 #> T1,
    infimum(T2, T).
happens(gate_open, T) :-
    gate_angle_rise(A1, T1),
    gate_angle_erise(A2, T2),
    A1 #> 0, A2 #=< 0,
    sampling_window(W),
    T2 #< T1 + W, T2 #> T1,
    infimum(T2, T).
happens(gate_raise, T) :-
    holdsAt(passing, T1),
    holdsAt(leaving, T2),
    sampling_window(W),
    T2 #< T1 + W, T2 #> T1,
    infimum(T2, T).
happens(train_exit, T) :-
    holdsAt(position(X1), T1),
    holdsAt(position(X2), T2),
    X1 #< 20, X2 #>= 20,
    sampling_window(W),
    T2 #< T1 + W, T2 #> T1,
    infimum(T2, T).
```

With the above modeling, we query s(CASP) to check various properties relative to the train speed and gate angle rotations. We can ask the question, whether the system is safe. That is, when the train is passing through the gate area, is it possible that the gate is open (or rising). This is expressed by the query `?- holdsAt(passing, T), holdsAt(open, T).` Similarly, we can test the liveness of the system, to check if the gate eventually becomes open after becoming closed, using : `?- holdsAt(closed, T1), holdsAt(open, T2), T2 .>. T1`. Note that, we consider only a single train crossing the gate area. The system is modeled in a way that there is a single track and the trains follow the set trajectory when approaching the gate area. We can also calculate the speed with which the train should approach the gate area, for the controller to respond and the gate to be closed in time. The constraint solving power of s(CASP) (and $CLP(R)$) simplifies the constraints and reduces the speed to a formula in terms of the angle lower and angle rise rate. For example, if we set the train to be moving too fast, then the controller cannot respond in time. In such a case, the gate might still be lowering when the train has crossed the gate area. Such scenarios are easily detected in our modeling.

3.1. Checking Safety and Liveness of Train-Gate-Controller

From the encoding explained above, let us assume that train speed is 1 unit per second, gate angle lower rate is 30 degrees per second and gate angle rise rate is 40 degrees per second. Given these parameters, `?- happens(train_in, T)` produces binding `T = 11`. That is, the train enters the gate area at time 11. Similarly, based on above definitions and assumed speed and angle rotation parameters, the train exits the gate area at time `T = 21`. That is, the query `?- happens(train_exit, T)` produces the binding `T = 21`. Therefore, the query `?- holdsAt(passing, T)` yields the binding `T > 11` and `T =< 21`. Based on these bindings when the train passes through the gate area, we can check the safety of the system. We can define what it means for the system to be unsafe. In our case, the system is in an unsafe state if the gate is either open/lowering/rising when the train is passing through the gate area. That is,

Query	Running Time (s)
`?- holdsAt(passing, T)`	0.0006
`?- unsafe [true]`	1.467
`?- unsafe [no models[`	2.082
`?- live [true]`	0.288
`?- live [no models]`	1.914

Table 1
Running times of sample queries

```
unsafe :- holdsAt(passing, T), holdsAt(rising, T).
unsafe :- holdsAt(passing, T), holdsAt(opened, T).
unsafe :- holdsAt(passing, T), holdsAt(lowering, T).
```

Then, asking s(CASP) the query `?- unsafe` yields *no models*. That is, the system is safe with respect to the assumed parameters. However, if the gate lowers at a slower rate, then it cannot be closed by the time the train is passing. If we lower the gate angle rate to 10 degrees per second instead of 30 degrees per second, the query `?- unsafe` produces a model. Similar to safety, we can check liveness of the sytem. That is, we can check if the gate becomes opened after being closed at the time of train passing. This is easily expressed as:

```
live :-
   holdsAt(passing,T),holdsAt(closed,T),holdsAt(opened,T1),T1 #> T.
```

Similar to safety, if we reduce the rate at which the gate angle rises, say from the original 40 degress per second to 10 degrees per second, the gate will not be open within a maximum time of 30 seconds. We can set 30 seconds within which the gate should be open. With this maximum time limit, the liveness can be checked. Table 1 lists the running times for the above queries in the s(CASP) system. The queries were run on a Quad code Intel(R) Core(TM) i7-10510U CPU @ 1.80Ghz. In general, running the discretized versions on clingo take a long time at the grounding stage itself due to the huge size of the grounded program.

4. Conclusion and Future Work

We have shown the ease of modeling cyber-physical systems requirement specifications in EC/s(CASP) and verification of their safety and liveness. We intend to apply our techniques to the Generalized Railroad crossing problem and industrial examples handled by UPPAAL tool. Also, given the EC/s(CASP) description of a cyber-physical system, one should be able to derive the timed-automata implementing the system. For instance, given the railroad crossing problem, we should be able to synthesize the timed-automata for the controller. We leave this for future work.

References

[1] R. Alur, Principles of cyber-physical systems, MIT press, 2015.

[2] G. Behrmann, A. David, K. G. Larsen, A tutorial on uppaal, in: M. Bernardo, F. Corradini (Eds.), Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures, volume 3185 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 200–236. URL: https://doi.org/10.1007/978-3-540-30080-9_7. doi:10.1007/978-3-540-30080-9_7.

[3] G. Gupta, E. Pontelli, A constraint-based approach for specification and verification of real-time systems, in: Proceedings Real-Time Systems Symposium, IEEE, 1997, pp. 230–239.

[4] N. Saeedloei, G. Gupta, Timed definite clause omega-grammars, in: M. V. Hermenegildo, T. Schaub (Eds.), Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, July 16-19, 2010, Edinburgh, Scotland, UK, volume 7 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010, pp. 212–221. URL: https://doi.org/10.4230/LIPIcs.ICLP.2010.212. doi:10.4230/LIPIcs.ICLP.2010.212.

[5] N. Saeedloei, G. Gupta, A logic-based modeling and verification of CPS, SIGBED Rev. 8 (2011) 31–34. URL: https://doi.org/10.1145/2000367.2000374. doi:10.1145/2000367.2000374.

[6] A. Bansal, Towards next Generation Logic Programming Systems, Ph.D. thesis, USA, 2007. doi:10.5555/1368767.

[7] J. Arias, Z. Chen, M. Carro, G. Gupta, Modeling and reasoning in event calculus using goal-directed constraint answer set programming, in: M. Gabbrielli (Ed.), Logic-Based Program Synthesis and Transformation - 29th International Symposium, LOPSTR 2019, Porto, Portugal, October 8-10, 2019, Revised Selected Papers, volume 12042 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 139–155. URL: https://doi.org/10.1007/978-3-030-45260-5_9. doi:10.1007/978-3-030-45260-5_9.

[8] B. Hall, et al., Knowledge-assisted reasoning of model-augmented system requirements with event calculus and goal-directed answer set programming, in: H. Hojjat, B. Kafle (Eds.), (To Appear) Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, Virtual, 28th March 2021, volume 344 of *EPTCS*, 2021, pp. 79–90. URL: http://dx.doi.org/10.4204/EPTCS.344.6. doi:10.4204/EPTCS.344.6.

[9] E. Erdem, M. Gelfond, N. Leone, Applications of answer set programming, AI Magazine 37 (2016) 53–68.

[10] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming., in: ICLP/SLP, volume 88, 1988, pp. 1070–1080.

[11] M. Gebser, et al., Potassco: The potsdam answer set solving collection, Ai Communications 24 (2011) 107–124. doi:10.3233/AIC-2011-0491.

[12] K. Marple, E. Salazar, G. Gupta, Computing stable models of normal logic programs without grounding, arXiv preprint arXiv:1709.00501 (2017).

[13] J. Arias, M. Carro, E. Salazar, K. Marple, G. Gupta, Constraint answer set programming without grounding, TPLP 18(3-4):337-354 (2018). doi:10.1017/S1471068418000285.

[14] M. Sergot, R. Kowalski, A logic-based calculus of events, New Generation Computing 4 (1986) 67–95. doi:10.1007/BF03037383.

[15] E. T. Mueller, Commonsense reasoning: an event calculus based approach, Morgan Kaufmann, 2014.

[16] J. J. Alferes, L. M. Pereira, T. Swift, Abduction in well-founded semantics and generalized

stable models via tabled dual programs, Theory and Practice of Logic Programming 4 (2004) 383–428.
[17] K. L. Clark, Negation as failure, in: Logic and data bases, Springer, 1978, pp. 293–322.
[18] J. Arias, M. Carro, Z. Chen, G. Gupta, Modeling and reasoning in event calculus using goal-directed constraint answer set programming, CoRR abs/2106.14566 (2021). URL: https://arxiv.org/abs/2106.14566. arXiv:2106.14566.
[19] M. Shanahan, An abductive event calculus planner, The Journal of Logic Programming 44 (2000) 207–240.