Modeling and Verification of Real-Time Systems with the Event Calculus and s(CASP)

Sarat Chandra Varanasi[1](✉), Joaquín Arias[2], Elmer Salazar[1], Fang Li[1], Kinjal Basu[1], and Gopal Gupta[1]

[1] The University of Texas at Dallas, Richardson, USA
{sxv153030,ees101020,fang.li,kinjal.basu,gupta}@utdallas.edu
[2] CETINIA, Universidad Rey Juan Carlos, Madrid, Spain
joaquin.arias@urjc.es

Abstract. Modeling a cyber-physical system's requirement specifications makes it possible to verify its properties w.r.t. the expected behavior. Standard modeling approaches based on automata theory model these systems at the *system architecture level*, as they have to explicitly encode the notion of states and define explicit transitions between these states. Event Calculus encoding using Answer Set Programming (ASP) allows for elegant and succinct modeling of these dynamic systems at the *requirements specification level*, thanks to the near-zero semantics gap between the system's requirement specifications and the Event Calculus encoding. In this work we propose a framework that uses the EARS notation to describe the system requirements, and an Event Calculus reasoner based on s(CASP), a goal-directed Constraint Answer Set Programming reasoner over the rationals/reals, to *directly* model these requirements. We evaluate our proposal by (i) modeling the well-known Train-Gate-Controller system, a railroad crossing problem, using the EARS notation and Event Calculus, (ii) translating the specifications into s(CASP), and (iii) checking safety and liveness of the system.

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 [1,7]. Timed-automata approaches require an explicit notion of state and transitions between states using clock variables [2]. Timed automata have also been modeled as constraint logic programs, where there is little semantic gap between the logic programs and the intended cyber-physical system that is modeled [10]. Techniques based on co-inductive constraint logic programming (Co-CLP) have also been applied in verifying properties of timed-automata [16,17]. 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 (EC) formalism has been used [4] along

J. Cheney and S. Perri (Eds.): PADL 2022, LNCS 13165, pp. 181–190, 2022.
https://doi.org/10.1007/978-3-030-94479-7_12

with powerful reasoning supported in Answer Set Programming. Additionally, the work of Arias et al. [3] extends prior work on Co-CLP to support natural reasoning over hybrid systems, in the language of Event Calculus. Theirs is the first work to use the s(CASP) system to model Event Calculus along with abductive reasoning supported in Answer Set Programming. The s(CASP) system has also been used in knowledge-based methods that analyze faulty requirements in simple avionics software systems modeled with a single automata [11]. In this paper we model more complex cyber physical systems that involve multiple automata (for example, the well-known Train-Gate-Controller system) in Event Calculus. We *start with the requirements specification* written succinctly and concisely in the EARS notation [14] and *model them directly in the Event Calculus*. We use the Event Calculus encoding in Answer Set Programming along with real-time constraints that can be directly run on the s(CASP) system. Thus, the entire system can be modeled in s(CASP). Simulation runs can be executed and safety and liveness of the system automatically checked, with prior knowledge of the physical properties such as train speed, system response time, and the rate at which the gate rotates.

The Event Calculus encoding in s(CASP) is obtained *directly* from the requirements specification written in EARS [14]. The main advantage of this approach is that design decisions do not "creep" into the encoding. This is in contrast to automata theoretic approaches (such as those based on timed automata) where some design decisions have to be made in order to obtain the timed automata encoding (for example, decisions regarding how to split the system into subsystems for each of which an automaton will be designed; what states and transitions these automata will have, etc.). Thus, verification performed at the level of timed automata verifies the requirements as rendered in the design realized in the automata, rather than at the level of requirements specification itself. In our approach, verification of safety and liveness is performed at the level of requirements specification. Thus, we can ensure that requirements are consistent and robust and permit a design that satisfies safety and liveness.

The methods developed in this paper allow us generate simulation runs of the system as well as check the correctness of its requirements specification. These experiments allow the user to refine/correct the system requirement specifications. Errors in specifications are a major source of flaws in software implementations. The later a defect is discovered in requirements specification, the costlier it is to fix. Thus, the ability to faithfully model requirements specification of a system can lead to significant benefits. The main contribution of the paper is the following:

- We show how requirements specification for a cyber-physical system can be directly modeled in the Event Calculus using ASP. This Event Calculus coding of requirement specifications can then be directly executed in s(CASP). The encoding can be used for generating simulation runs and for verification, for example, of safety and liveness properties.
- Use of the Event Calculus, ASP, and s(CASP) for modeling real-time systems has been limited to very simple examples. We present the encoding of a complex system, namely, the canonical Train-Gate-Controller system that has been widely discussed in the literature [2].

Ubiquitous: always active. The `<system name>` shall `<system response>`

State Driven: active as long as the specific state remains true.
WHILE `<precondition(s)>`, the `<system name>` shall `<system response>`

Event Driven: specify how a system must respond when a triggering event occurs.
WHEN `<trigger>`, the `<system name>` shall `<system response>`

Unwanted Behavior: specify the required system response to undesired situations.
IF `<trigger>`, THEN the `<system name>` shall `<system response>`

Complex Behavior: specify requirements for richer system behaviour.
WHILE `<precondition(s)>`, WHEN `<trigger>`,
the `<system name>` shall `<system response>`

Fig. 1. Generic EARS syntax

2 Background

2.1 Easy Approach to Requirement Syntax (EARS)

The Easy Approach to Requirement Syntax (EARS) [13,14] is a pragmatic approach to specifying requirements for cyber-physical systems based on using five structured templates and keywords popular in the avionics industry. Keywords 'WHEN', 'WHILE' and 'IF'-'THEN' are used in these templates and play a major role (see Fig. 1). Studies have shown the use of EARS to reduce requirements errors while improving requirement quality and readability [14]. For cyber-physical real-time systems, response times are important, hence formally budgeting the allocation of time throughout the levels of function & temporal decomposition are primary concerns. An example requirement specification in EARS style is given: **WHEN** the `train_position reaches 10 feet,` the `Train-gate-Controller` **SHALL** `trigger gate_closure within 1 s.`

System response and trigger are typically time constrained events. In Sect. 3 we explain how requirement specifications written in EARS can be directly modeled in Event Calculus.

2.2 Basic Event Calculus (BEC)

Event Calculus (presented at length elsewhere [15]) is a formalism for reasoning about events and change, of which there are several axiomatizations. In this paper we use the Basic Event Calculus (BEC) formulated by [18]. There are three fundamental, mutually related, concepts in EC: *events*, *fluents*, and *time points*. An event is an action or incident that may occur in the world: for instance, a person dropping a glass is an event. A fluent is a time-varying property of the world, such as the altitude of a glass. A time point is an instant in time. Events may happen at a time point; fluents have a truth value at any time point or over an interval, and their truth values are subject to change, upon the occurrence of an event. In addition, fluents may by associated with (continuous) physical quantities that change over time. For example, rolling a ball on the floor can be described by two *fluents*: one *fluent* that states that the ball itself is *rolling*, while another *fluent* captures *movement* of the ball in some metric unit, changing at a certain rate, over time. The *event* of setting the ball to roll *initiates rolling* and also determines the change in position from a starting point. Likewise, the event of stopping the ball *terminates* rolling of the ball and the ball is now stationary in its last position. An EC description consists of a *domain narrative* and a *universal theory*. The domain narrative consists of the causal laws of the domain, the known events, and the fluent properties, and the universal theory is a conjunction of EC axioms that encode, for example, the commonsense laws of inertia. In Sect. 3 we show how EC descriptions can be translated and evaluated using an EC-reasoner implemented using s(CASP) [5].

2.3 Goal-Directed Answer Set Programming

Our framework relies on Answer Set Programming (ASP) [9] to encode system requirements. In particular, we use the goal-directed s(CASP) [3] system. The top-down query-driven execution strategy of s(CASP) has three major advantages w.r.t. traditional ASP system: (a) it does not require to ground the programs; (b) its execution starts with a query and the evaluation only explores the parts of the knowledge base relevant to the query. Hence relying on a strategy not based on grounding, makes s(CASP) scalable for cyber physical domains using dense real-valued time. Additionally, s(CASP) can output the justification tree for issued queries and provide an easily visualizable HTML version of the same tree. The predicates used in the modelling can be mapped to their intended English language meanings to make the justifications more readable. These justifications make it possible to understand the behavior of the cyber-physical system, when its properties hold and when they do not hold.

In Sect. 4 we show that the safety and liveness properties of a CPS can be checked using s(CASP) queries. The direct mapping of EC Axioms in [5] is possible due to s(CASP) capability to support continuous time. To the best of our knowledge, s(CASP) is the only logic programming system that encodes EC with dense time. This is to be distinguished from grounding-based ASP solvers that can only reason over discretized time [12].

3 Modeling and Verifying Cyber Physical Systems in EC

The Train-Gate-Controller (TGC) is a cyber-physical system commonly used to study modeling and verification of properties of such systems [10]. The system consists of a set of sensors and actuators that automatically open and close a railway gate upon detecting the arrival or departure of a train. The system should signal gate closure in a timely fashion.

Let us consider the specifications of the Train-Gate-Controller system described in [2]:[1] *The train signals its* `approach` *and* `exit`*. Events* `in` *and*

[1] We have made some minor (and inconsequential) changes to the Train-Gate-Controller system to simplify the illustration.

`out` signal the entry and exit of the train from the gate area. The train should signal `approach` at least 2 min before entering gate area. This forces the minimum delay between `approach` and `in` to be 2 min. The maximum delay between `approach` and `exit` is 5 min. We make the train's approach more tangible by considering actual movement of the train on a track. We set markers for the entry point and exit point of the gate area. When the trains position hits these points, then correspondingly, the train has entered or exited the gate area. Therefore, the minimum 2 min delay is ignored. Further, we consider if the gate is eventually closed, ignoring the 5 min delay. We assume that the train changes its position uniformly at a rate of 10 units per second. The gate area is at position 10. 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 system should signal the *closing* of the gate before the train is in the gate area. The gate also uniformly changes its angle of inclination when it is in motion. When the *gate angle* becomes 90°, the gate is closed and inclined horizontally.

3.1 Train-Gate-Controller in EARS

In this section we translate the TGC requirements into EARS notation.

R1 **WHEN** the train reaches a position of 10 units, **the** system **shall** signal the train to be in the gate area
R2 **WHEN** the train reaches a position of 5 units, **the** system **shall** signal lowering of the gate
R3 **WHEN** the gate angle reaches vertical angle of 90° from below, **the** system **shall** signal gate closure
R4 **WHEN** the gate angle reaches a vertical angle of 0° from above, **the** system **shall** signal the gate to be open
R5 **WHEN** the train starts leaving the gate area, **the** system **shall** signal raising of the gate
R6 **WHEN** the train reaches a position of 20 units, **the** system **shall** signal the train to be exiting the gate area
R7 **The** system **shall** ensure the gate is closed when the train is passing through the gate area
R8 **The** Gate **shall** be open after train has exited the gate area

3.2 Train-Gate-Requirements in EC Using s(CASP)

We first identify the fluents (sensor triggered) and events (actuator triggered). For clarity in the code below, fluent & event names have been made more descriptive (e.g., `in` has been renamed `train_in`).

```
fluent(passing).      % Train is passing through the gate area
fluent(leaving).      % Train is leaving form the gate area
fluent(position(X)). % Train is at some position X
fluent(gate_angle(A)). % Gate is vertically inclined with an angle A
fluent(opened).       % The gate is completely opened
fluent(closed).       % The gate is completely closed
fluent(lowering).     % The gate is being lowered
fluent(rising).       % The gate is being raised
event(train_in).      % The train enters the gate area
event(signal_lower). % The system signals gate lowered
event(signal_raise). % The system signals gate raised
event(gate_close).    % The gate closes
event(gate_open).     % The gate opens
event(train_exit).    % The train exits the gate area
```

The causal effects of the events in the system follow straightforwardly:

```
initiates(train_in,passing,T).         terminates(signal_lower,opened,T).
initiates(signal_lower,lowering,T).    terminates(gate_close,lowering,T).
initiates(gate_close,closed,T).        terminates(train_exit,passing,T).
initiates(train_exit,leaving,T).       terminates(signal_raise,closed,T).
initiates(raise,rising,T).             terminates(gate_open,rising,T).
initiates(gate_open,opened,T).
```

Next, `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 rises is `R`. We now describe the conditions under which various events happen. The motion of the train itself is modeled as a trajectory. Similarly, the change in inclination of the gate angle is also modeled as a trajectory, depending upon whether `event(signal_lower)` or `event(signal_raise)` happen. If the gate is lowering (rising), then the gate inclination steadily decreases (increases)[2].

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

```
gate_angle_lower(A, T2) :-        gate_angle_rise(A, T2):-
   happens(signal_lower,T),          happens(signal_raise,T),
   angle_lower_rate(L),              angle_rise_rate(R),
   T2 #> T,                          T2 #> T,
   A #= (T2-T1)*L.                   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 system signals `lower_gate` when the train position crosses value = 5. All transitions in the train position, gate angle are resolved at a sampling window of 0.1 time unit. That is, the system

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

can detect changes in continuous quantities at a temporal precision of 0.1 time unit. This is a reasonable assumption to make the system behave realistically. If we used a temporal precision of 0, then the system can detect instantaneous changes in continuous values, which is impossible in a real-world system. We use the `infimum` on the 0.1 s interval, to signify the precise instance when the transition of train position or gate angle crosses a threshold. Note that any arbitrarily small (positive) value can be chosen for this temporal precision.

```
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(signal_lower, T) :-
   holdsAt(position(X1),T1),
   holdsAt(position(X2),T2),
   X1 #< 5, X2 #>= 5,
   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_rise(A2,T2),
   A1 #> 0, A2 #=< 0,
   sampling_window(W),
   T2 #< T1 + W, T2 #> T1,
   infimum(T2, T).
happens(signal_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 also ask whether system is safe, i.e., if when the train is passing through the gate area the gate is open (or rising): `?- holdsAt(passing, T), holdsAt(open, T)`. Similarly, we can check liveness, i.e., if the gate eventually becomes open after becoming closed: `?- 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 train follows the set trajectory when approaching the gate area. As we describe in Sect. 4, if the gate lowers too slowly, it will still be lowering when the train has crossed the gate area. Such scenarios are easily detected in our modeling.

4 Checking Safety and Liveness of Train-Gate-Controller

We present several scenarios using the TGC to reason about the train and the controller behavior and check whether the system satisfies desired properties.

Note that, requirements *R7* and *R8* from EARS spec are safety and liveness checks, respectively.

- Scenario A: (i) train speed is 1 unit per second, (ii) gate angle lower rate is 30 degrees per second and, (iii) gate angle rise rate is 40 degrees per second:
  - The query, `?- happens(train_in,T)` produces binding `T = 11`, i.e., the train enters the gate area at time `11`.
  - The query, `?- happens(train_exit,T)` produces the binding `T = 21`, i.e., the train exits the gate area at time `21`.
  - The query `?- holdsAt(passing,T)` yields the binding `T #> 11` and `T #=< 21`, i.e., it represents the interval $(11, 21]$.
- Scenario B: (i) train speed is 1 unit per second, (ii) gate angle **lower rate is 10** degrees per second and, (iii) gate angle rise rate is 40 degrees per second.
- Scenario C: (i) train speed is 1 unit per second, (ii) gate angle lower rate is 30 degrees per second and, (iii) gate angle **rise rate is 10** degrees per second.

4.1 Safety and Liveness Queries

Let us check what happens when the train passes through the gate area, we can check the *safety* of the system. Thus, we define what it means for the system to be unsafe: the system is in an unsafe state if the gate is either open, lowering, or rising when the train is passing through the gate area:

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

For the scenario A, the query `?- unsafe` yields *no models*, therefore, the system is safe w.r.t. the assumed parameters. However, for the scenario B, the query `?- unsafe` produces a model, meaning that the system is unsafe. Similar to safety, we can check *liveness* of the system, i.e., if the gate after being closed at the time of train passing it becomes opened before a threshold `Th`.

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

In scenario C, the gate will not be open within 30 s, so if we set the liveness threshold to 30 s, the query `?- live` yields *no models*.

Table 1 lists the running times, in seconds, for the above queries to the different scenarios under s(CASP). Running times to check requirements 1 through 6 of TGC are also listed. They are straightforwardly translated into s(CASP) queries. The evaluation is run on a Quad code Intel(R) Core(TM) i7-10510U CPU @ 1.80 GHz with 8-GB RAM. In general, running the discretized versions on the CLINGO ASP system [8] takes a long time at the grounding stage itself due to the huge size of the grounded program. We ran the EC encodings of TGC based on F2LP/Clingo [12]. They produce no results at our timeout value of 40 min. The EC modelling for TGC can be found using this link: https://github.com/sarat-chandra-varanasi/event-calculus-scasp/blob/main/train_example/trajectory/trajectory.lp

Table 1. Run-time (s) comparison of TGC with 3 scenarios under s(CASP).

	Scenario A		Scenario B		Scenario C	
	Answer	Time	Answer	Time	Answer	Time
`?- holdsAt(passing, T).`	(11, 21]	0.241	(11, 21]	0.304	(11, 21]	0.241
`?- unsafe.`	×	2.082	✓	1.834	×	2.156
`?- live.`	✓	0.641	✓	0.607	×	1.972
`?- req1.`	✓	0.245	✓	0.231	✓	0.275
`?- req2.`	✓	0.230	✓	0.240	✓	0.245
`?- req3.`	✓	0.259	✓	0.265	✓	0.245
`?- req4.`	✓	0.461	✓	0.385	✓	0.441
`?- req5.`	✓	0.265	✓	0.300	✓	0.289
`?- req6.`	✓	0.240	✓	0.258	✓	0.249

5 Conclusion and Future Work

We have shown the ease of modeling cyber-physical systems in EC/s(CASP) and verification of their safety and liveness properties. We intend to apply our techniques to the Generalized Railroad crossing problem and industrial examples handled by UPPAAL system [7]. Also, given the EC/s(CASP) description of a cyber-physical system, one should be able to automatically derive the timed-automata implementing the system. For instance, given the railroad crossing system requirements specification, we should be able to synthesize the timed-automata for the various sub-systems, thereby opening doors to generating an implementation directly from requirement specifications that satisfies safety and liveness constraints. This would be a step towards "correct by design" approach to constructing software. In fact, one could go a step further and generate the EC/s(CASP) code directly from requirements specifications written in EARS for cyber-physical systems, and then generate an implementation from that EC/s(CASP) encoding. We leave these explorations for future work.

Acknowledgement. We are grateful to Brendan Hall, Jan Fiedor, and Kevin Driscoll of Honeywell Aerospace for discussions. Authors gratefully acknowledge support from NSF grants IIS 1718945, IIS 1910131, IIP 1916206, from Amazon Corp and US DoD, and MICINN projects RTI2018-095390-B-C33 InEDGEMobility (MCIU/AEI/FEDER, UE). Views expressed are authors' own and not of the funding agencies. We also dedicate this work to the memory of first author's father, Late Prof. Sitaramaiah Varanasi, who was passionate about Number Theory and ever so curious about Theoretical Computer Science.

References

1. Alur, R.: Principles of Cyber-Physical Systems. MIT Press, Cambridge (2015)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)

3. Arias, J., Carro, M., Salazar, E., Marple, K., Gupta, G.: Constraint answer set programming without grounding. In: Theory and Practice of Logic Programming, vol. 18, no. 3–4, pp. 337–354 (2018). https://doi.org/10.1017/S1471068418000285
4. Arias, J., Chen, Z., Carro, M., Gupta, G.: Modeling and reasoning in event calculus using goal-directed constraint answer set programming. In: Gabbrielli, M. (ed.) LOPSTR 2019. LNCS, vol. 12042, pp. 139–155. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45260-5_9
5. Arias, J., Carro, M., Chen, Z., Gupta, G.: Modeling and reasoning in event calculus using goal-directed constraint answer set programming. In: Theory and Practice of Logic Programming, pp. 1–30 (2021). https://doi.org/10.1017/S1471068421000156
6. Bansal, A.: Towards next generation logic programming systems. Ph.D. thesis, Department of Computer Science, University of Texas at Dallas (2007)
7. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_7
8. Gebser, M., et al.: Potassco: the Potsdam answer set solving collection. AI Commun. **24**(2), 107–124 (2011). https://doi.org/10.3233/AIC-2011-0491
9. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: 5th International Conference on Logic Programming, pp. 1070–1080 (1988). http://www.cse.unsw.edu.au/~cs4415/2010/resources/stable.pdf
10. Gupta, G., Pontelli, E.: A constraint-based approach for specification and verification of real-time systems. In: Proceedings Real-Time Systems Symposium, pp. 230–239. IEEE (1997)
11. Hall, B., et al.: Knowledge-assisted reasoning of model-augmented system requirements with event calculus and goal-directed answer set programming. In: Hojjat, H., Kafle, B. (eds.) Proceedings of the 8th Workshop on Horn Clauses for Verification and Synthesis, Virtual, Volume 344 of EPTCS, 28 March 2021, pp. 79–90 (2021). https://doi.org/10.4204/EPTCS.344.6
12. Lee, J., Palla, R.: System F2LP – computing answer sets of first-order formulas. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS (LNAI), vol. 5753, pp. 515–521. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04238-6_51
13. Mavin, A., Wilkinson, P.: Big EARS (the return of "easy approach to requirements engineering"). In: 2010 18th IEEE International Requirements Engineering Conference, pp. 277–282. IEEE (2010). https://doi.org/10.1109/RE.2010.39
14. Mavin, A., et al.: Easy approach to requirements syntax (EARS). In: 2009 17th IEEE International Requirements Engineering Conference, pp. 317–322. IEEE (2009). https://doi.org/10.1109/RE.2009.9
15. Mueller, E.T.: Commonsense Reasoning: An Event Calculus Based Approach. Morgan Kaufmann, Waltham (2014)
16. Saeedloei, N., Gupta, G.: A logic-based modeling and verification of CPS. SIGBED Rev. **8**(2), 31–34 (2011). https://doi.org/10.1145/2000367.2000374
17. Saeedloei, N., Gupta, G.: Timed definite clause ω-grammars. In: Hermenegildo, M.V., Schaub, T. (eds.) Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, Volume 7 of LIPIcs, Edinburgh, Scotland, UK, 16–19 July 2010, pp. 212–221 (2010)
18. Shanahan, M.: The event calculus explained. In: Wooldridge, M.J., Veloso, M. (eds.) Artificial Intelligence Today. LNCS (LNAI), vol. 1600, pp. 409–430. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48317-9_17