Game-theoretic Simulations with Cognitive Agents

Nausheen Saba Shahid, Dan O'Keeffe, Kostas Stathis
Department of Computer Science, Royal Holloway University of London, UK
Nausheen.Shahid.2017@live.rhul.ac.uk, {daniel.okeeffe, kostas.stathis}@rhul.ac.uk

***Abstract*—We propose a novel knowledge representation framework called COGNISIM that supports game theoretic simulation experiments using cognitive agents. The framework allows an experimenter to evolve a population of such agents with strategies expressed teleo-reactively as logic programs. When agents encounter each other, events take place in the environment, caused either by agent actions or by environment processes. Such events change the environment's internal state, and these changes are then observed by agents that, in turn, decide to take new actions that affect the environment. This loop continues until the terminating conditions of the simulation are met. Using this framework, we show how to repeat experiments from the literature based on Axelrod's tournament. We also evaluate our platform's performance in efficiently supporting large simulations in game theoretic settings.**

***Index Terms*—game-theoretic simulation, cognitive agent simulation, cognitive simulation platform, cooperation**

I. Introduction

We are concerned with the problem of how to represent executable specifications of game-theoretic simulations using cognitive agents. In such simulations agents use pre-programmed models to play in the environment, constantly review these models with what they observe, and reason about strategic decisions that they need to make to maximise their preset objectives. This is a challenging problem, as representing game-theoretic strategies that are transparent and amenable for inspection at any point during or after a simulation, is not currently readily available. Existing platforms e.g. [1], [2], [3] and [4], provide little details on how experimenters develop and inspect the behaviour of players during simulation.

In order to represent the simulation environment, the games it may contain, and their players, our framework advocates the use of existing meta-programming techniques that enable us to manipulate simulation components as program structures that evolve due to simulation events, as it is customary in existing frameworks [5]. Our target framework is based on normal logic programs implemented in Prolog and uses the same data structures to represent programs as well as data. The novelty of our approach is that we combine meta-interpreters [6] to represent simulation components, the Event Calculus (EC) [7] to represent evolution of these components due to events happening in them, and an EC extension to deal with agent interaction within the resulting agent environment [8]. Also, to formulate player strategies, we use a teleo-reactive model of execution [9], but re-interpreted within our framework.

This work was supported by the Royal Holloway, University of London, UK & the Higher Education Commission (HEC), Pakistan.

Our contribution is a practical, symbolic-AI representation framework and a platform identifying the important parts of a simulation. These parts are then used to develop game-theoretic interactions methodologically [10], so that simulation experiments are easier to repeat. We show how to use the framework to perform domain specific simulations, by repeating existing experiments to obtain their original results. We also report on the computational efficiency of our framework and the trade offs of strategic reasoning within our platform.

Our work is founded on a series of well-known techniques, suitably adjusted, and presented in Section II. However, our framework non-trivially extends this background work in Section III. The experimental evaluation of our framework is presented in Section IV. We conclude with Section V, where we also present our plans for further work.

II. Background

We use normal logic programs as components to represent players and the way they interact in a simulation. Logic programming rules are specified as assertions of the form:

```
rule(C, G, [G1,G2,..Gn]),
```

where C is an explicit name for the component, G is the head of the rule, and the list [G1,G2,...,Gn] represents the rule's body. For instance, to express in a player component p1 the fact that *john knows mike* and the rule that *X is a friend of Y if X knows Y and Y is not a defector*, we write:

```
rule(p1, knows(john, mike),[]).
rule(p1, friend(X,Y), [knows(X,Y), not defector(Y)]).
```

X, Y are variables, john, mike are constants, and not in the second rule is *negation-as-failure* [11]. To interpret such rules, we use the meta-interpreter demo(C,G) [12]:

```
demo(C, G) ← rule(C, G, B), demo(C,B).
```

i.e. a conclusion G holds in component C, if there is a rule in C with a head G and body B (of the form [G1,..Gn]), and all elements in the body of the rule hold in C. We also need:

```
demo(C, [G|Gs]) ← demo(C, G), demo(C, Gs).
demo(_, []).
```

to cater for the body of a rule (first clause, where the list represents a conjunction), or an empty body (second clause dealing with facts i.e. the body is the empty list []). The underscore '_' denotes an anonymous variable. We also need an extra definition for negation-as-failure:

```
demo(C, not G) ← not demo(C, G).
```

and primitives for arithmetic and system operations. Now, to answer the query *who is friend of whom* we only need to specify the component's name as ?-demo(p1, friend(X,Y)).

Organising rules in components still requires to address how components change. We use the EC that relies on three basic constructs to represent change: events, fluents (state variables) and time points with a linear time model. Many versions of the EC exist [13], here we use the one in [14], but with multi-valued fluents [15]. Events happen instantaneously and represented in terms of the time point when they happen. An event initiates/terminates the value of a fluent in the state of interest indexed by a time point. Table I summarises the ontology of the domain-independent axioms representing state changes of a component (e.g. the environment or an agent).

TABLE I: Ontology of the Event Calculus used.

Predicate	Description
happens_at(E, T)	Event E happens at time T.
holds_at(F=V, T)	Fluent F has value V at time T.
holds_for(F=V, [Ts,Te])	Fluent F continuously has value V from time Ts to time Te.
broken(F=V, [Ts,Te])	Fluent F has changed value V from time Ts to time Te.
initiates_at(E, F=V, T)	Event E initiates value V for fluent F at time T.
terminates_at(E, F=V, T)	Event E terminates value V for fluent F at time T.

We can express generically when a fluent F holds at a time T in terms of whether that fluent holds for an interval in which T is member of. We can also express what happens at the initial state and how fluents terminate, for more details see [14]. These generic axioms can then be combined with domain specific ones to capture the evolution of a component. Suppose, for instance, that who is a defector is a fluent i.e. it changes depending on whether they cooperate or not (defect) against us. Suppose also that initially i.e. at time 0, $mike$ is assumed not to be a defector, but as he defects at time 5, i.e. he is perceived as defector after that. We can specify this as:

```
happens_at(initially(defector(mike)=false),0).
happens_at(defects(mike),5).

initiates_at(defects(X), defector(X)=true,T) ←
        happens_at(defects(X), T).
```

A generic axiom for terminates_at/2 [15], combined with the domain independent and the domain dependent EC axioms, answer queries such as ?- holds_at(defector(X)=false, 1) and ?- holds_for(defector(X)=true, [5,∞]). To avoid the problem of performing reasoning every time from scratch, our use of the Cached Event Calculus (CEC) [14] maintains a cached internal representation of fluents with a set of Maximally Validity Intervals (MVIs), see Fig.1.

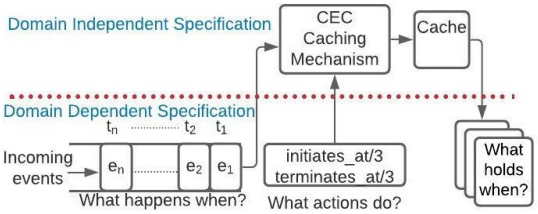


Fig. 1: Cached Event Calculus

An update_at(Event, T) predicate (see in [14], [16]) caches the consequences of events by inferring all relevant consequences using the domain definitions for initiate_at/3 and terminates_at/3, and adding holds_for/2 instances in the cache. These details we omit due to lack of space.

We are now in a position to use the Event Calculus to represent state properties that are affected by events happening within a component C as fluents whose values hold in the component at a specific time, specified as F=V:T. However, we now need to add to the meta-interpreter what we might refer to as a bridge rule:

```
demo(C, F=V:T) ← holds_at(C:F=V, T).
```

This rule requires that some fluents hold within components, specified as C:F=V at the holds_at/2 level, making necessary representational adjustments to event descriptions and their effects accordingly. And, although this tackles the problem of how components evolve, it does not say how components interact. For this we adopt the Ambient Event Calculus (AEC) [8], an EC extension describing how agents interact by attempting events using their actuators (via attempt_at(Event, T) assertions, and allowed to happen only if they are possible (using possible_at(Event, T) assertions). These are then combined as:

```
happens_at(E,T) ← attempt_at(E,T), possible_at(E,T).
```

The environment then notifies to all agents sensors that subscribe to an event (using the AEC notify_at(Event, Sensor, T)), and in this way we simulate interaction. Although the AEC has focused more on how to distribute the GOLEM platform [17], an explicit framework on how the environment evolved during the interaction was missing. In this paper, we show how to explicitly represent this missing aspect and how to use it to build a simulation platform.

III. The COGNISIM Platform

A. The COGNISIM Reference Model

COGNISIM supports an *Experimenter* to configure simulations, which are run via the system's *Display*, see

Fig. 2. An *Experiment Configuration* specifies the players involved in encounters, their strategies and other information such as the rounds to perform. A simulation is managed by the *Simulation Control* module, which creates *Player Agents* using the *Simulation Component Knowledge Base* (SCKB). SCKB defines components such as agent models, strategies, rules for updating an agent's knowledge, and shared rules such as communication protocols. The *Simulation Control* relies upon a *Conductor Agent* to manage the simulation. Simulation events are stored in an *Event History*. For large

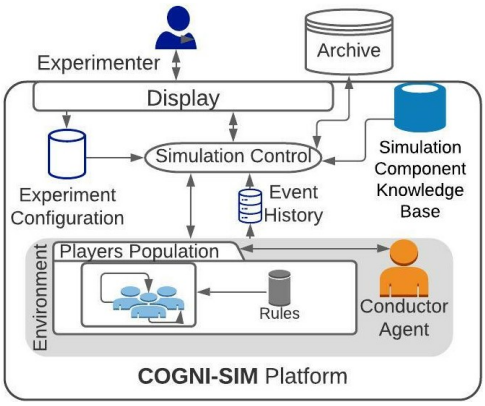


Fig. 2: COGNISIM Reference Model

simulations, the platform relies on an optimisation step, that allows for events to be transferred from the simulation to an external *Archive* for later use. The predicates of the core *Simulation Control* are shown below.

```
simulate_for([Ti,Te])← initialize_at(Ti, [Ts, Te]),
                       evolve_for([Ts,Te]).

evolve_for([Ts, Te])←Ts≤Te
                       consume_at(Es, Ts),
                       display_at(Es, Ts),
                       optimise_at(Ts),
                       evolve_for([Ts+1, Te]).
evolve_for([Ts, Te]) ←Ts>Te, finalize_at([Ts, Te]).
```

The clause of simulate_for/1 takes the initial time Ti, determines the starting and ending times (Ts and Te respectively) of the simulation, initializes the population of agents using initialize_at/2, evolves it and returns the end time Te. The evolution of a tournament is dealt within evolve_for/2, which takes the starting time Ts and ending time Te, consumes all the events Es that take place in the environment, displays the next state to all agents, including the user. If necessary, a state is optimised and then the simulation moves to the next cycle until the end time Te is reached. The evolution of the agent population stops if Te is reached, where the simulation is finalised and the results are saved.

B. Evolving A Tournament

Once a cognitive agent population is initialized, it goes through the process of evolution dealt in evolve_for/2. This starts with a consume_at/2 step, where all the active agents situated in the environment are asked to attempt an action. In COGNISIM agents are conceptualized as entities with a mind that makes decisions and a body that has sensors and actuators as in [18]. A cycle_at/2 predicate defines a control cycle [19] to attempt actions. The following specification represents the definition of consume_at/2.

```
consume_at(Es, T)←
    findall(Ag, active_agent(Ag, T), Ags),
    forall(member(A, Ags),cycle_at(A,T)),
    findall(E, (attempt_at(E, T),possible_at(E,T)), Es),
    forall(member(E, Es), retract(attempt_at(E, T))).
```

Once all agents have gone through one cycle, the consume_at/2 collects all generated events via attempt_at/2, checks if they are possible using possible_at/2, and removes all previous attempts in preparation for the next cycle. Then the display_at/2 takes all generated events, executes them and asserts that they have happened (via happens_at/2) and the environment's state is updated (via update_at/2).

```
display_at( Es, T)← forall(member(E, Es), update_at(E, T)).
```

C. Cognitive Agents

Fig. 3 shows the architecture of an agent in COGNISIM. An agent is a component identified by a unique id that contains the agent's goals and model of the beliefs this agent has about the environment, other agents and itself. The agent also has a strategy triggered by a cycle that is the agent's control interpreter for selecting actions to perform via an action execution module.

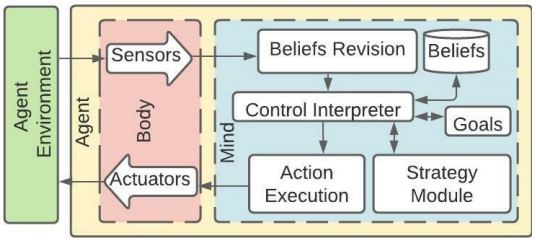


Fig. 3: Agent Architecture

1) The Cycle of Cognitive Agent: An agent evolves with time via cycle_at/2 as follows:

```
cycle_at(Ag, T)← observe_at(Ag, Observations, T),
                 assimilate_at(Ag, Observations, T),
                 decide_at(Ag, Action, T),
                 execute_at(Ag, Action, T).
```

observe_at/2 straightforwardly searches the agent's Ag sensors, to extract the Observations notified to them at time T. Then assimilate_at/3 takes the Observations and caches their consequences one by one, i.e. by updating the knowledge

of the agent by interpreting initiates_at/3 and terminates_at/3 definitions.

```
assimilate_at(Ag, [Obs|RestOfObs], T)←
    update_at(in(Ag, observation(Obs)), T-1),
    assimilate_at(Ag, RestOfObs, T).
assimilate_at(_, [], _).
```

An agent decides an action using the current knowledge and returns it, as shown below.

```
decide_at(Ag, Action, T)← select_at(Ag, Action, T).
```

We assume that a decision, vis. selecting an Action, contributes to a top-level goal G the agent must achieve (or maintain). Such a selection involves using the demo/2 meta-interpreter to interpret the player's beliefs and results in a teleo-reactive behaviour in the sense of [9]. Instantiating a rule at a time T, treats such a rule as an active intention for a persisting goal. Depending on how the state of the environment changes, the agent will adapt its behaviour, as specified by the ordering and conditions in the rules. The link between the agent cycle and its reasoning is shown below:

```
select_at(Ag, Action, T)←
    holds_at(Ag:top_level_goal(G)=true,T),
    demo(Ag, [select(G,Action,T)]).
```

A selected Action is then executed by being directed to a suitable agent actuator, which packages the action to an attempt event. The execution of the attempt event E, is done by asserting it in the simulation's state at time T:

```
execute_at(Ag, Action, T)←
    return_at(Ag, Action, E, T),
    assert(attempt_at(E, T)).
```

D. Performance Optimisation

Simulations in COGNISIM hold large state histories as opposed to only mathematical models of the current state. To improve run-time efficiency, we introduce the concept of *forgetting*, where events and knowledge are removed from the cache and only important events and knowledge are archived externally for later experimental analysis purposes. Forgetting of working memory structures like agent knowledge and notified events is handled in optimise_at/1 that takes the time T to test whether this is a suitable time to optimise. When this is the case, it searches for the working memory for forgettable structures at time T and forgets them. Below we omit the case when it is not the time to optimise (this trivially succeeds).

```
optimise_at(Ts) ← is_time_to_optimise(Ts),
    findall(WMi, forgettable_at(WMi,Ts), WM),
    forall(member(WMi, WM), forget_at(WMi,Ts)).
```

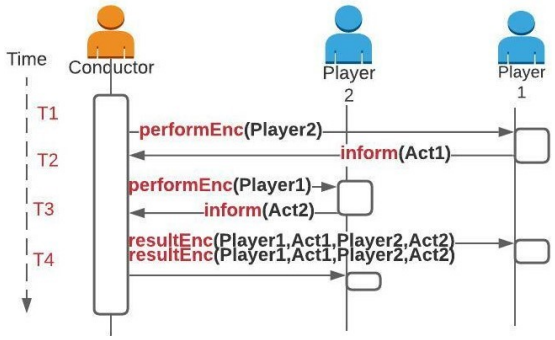


Fig. 4: Flow of an Encounter

An event E is forgettable if it has happened before T (i.e. notified and processed), while a fluent F is forgettable if it holds for [Ti,Tj] and Tj is before T.

```
forgettable_at(event(E,Ti),T) ← happens_at(E,Ti),Ti<T.
forgettable_at(fluent(F,[Ti,Tj]),T)←holds_for(F,[Ti,Tj]),Tj < T.
```

forget_at/2 deals with forgetting from the cache information that is not important. For important information we archive it first and then we forget it. The details we omit, as they are beyond the scope of this work.

IV. Experimental Evaluation

A. Use Case: Axelrod Tournament

We show how to conduct an Axelrod's [20] round-robin tournament, using the Prisoner's Dilemma (PD) game [21]. A single PD game between two players is an encounter, mediated by a conductor agent, which selects players to perform a move, and when they respond, the conductor informs them of the final result (see Fig. 4). The set of encounters where each strategy has played against every other strategy (and itself) is called one round. A tournament consists of a number of rounds. At the end, the player with the maximum payoff wins.

B. Simulation Experiments

An experiment is described with assertions of the form:

```
experiment(Id, Configuration).
```

The Configuration contains statements that instantiate in the specific experiment Id all the necessary structures for the agent environment to start evolving, for example:

```
experiment(exp1, (
  set(payoffs(punish:1,sucker:0,reward:3,tempt:5)),
  output(trace_in('results_exp1.txt')),
  make(1,conductor([players(17),rounds(1,10)])),
  make(1, player([strategy(titfortat)])),.....)).
```

As in [22], above we set a payoff matrix, a file name for storing events and results, a conductor for 17 players and 10 rounds, the player strategies and so on. We describe next how to build the different simulation components.

C. Making Agents

COGNISIM provides a default agent that observes the environment and performs communicative acts. Using the cycle of Section III-C1, the experimenter can extend the default agent with behaviours to create players or a conductor. To make agents following a strategy, the make/2 primitive generates unique agent identifiers, such as ag1, and constructs the initial state with assertions of the form:

```
initially(ag1,[strategy(titfortat)]).
```

Given a strategy, specific rules will be attached to players made to follow that strategy. An example of such specific rules for agent ag1 that plays *titfortat* is given below. This assumes a top-level goal called strategy(titfortat) that triggers the behaviour of ag1 at each cycle:

```
rule(ag1, select(strategy(titfortat), cooperate(ag1,Opp,Enc), T),[
                    in_encounter(Enc)=true:T,
                    opponent_in(Enc)=Opp:T,
                    interacted_before(Opp)=yes:T,
                    last_move(Opp)=cooperate:T]).
```

A rule like this will then be meta-interpreted in our framework.

D. Initialising Population

A population of agents is initialized in COGNISIM using initialize_at/2:

```
initialize_at(Ti, [Ts, Te])← calculate_duration(Ti,Ts,Te),
                findall(P,role_of(P,player),Ps),
                forall(member(P,Ps),initialize_at(player(P), Ti),
                role_of(C,conductor),
                initialize_at(conductor(C), Ti).
```

This way, we initialise all players and the conductor. Initialization of the conductor agent also deals with keeping a record of all the players and their score, initially set to zero.

E. Agent Evolution

1) Agent Initialisation: Definition of initialize_at/2 specific to a player or a conductor agent first performs general initialization for the agent, according to the following rule:

```
initialize_at(agent(Ag), T)← initially(Ag, List),
        forall(member(P,List),update_at(initially(Ag, P),T)).
```

As in [14], we treat initially/2 terms as events that happen to construct the knowledge of the agent. We then use a rule of the form:

```
initiates_at(initially(Ag,P), Ag:P=true, T)←
                happens_at(initially(Ag,P), T).
```

to initialize the agent beliefs.

2) Belief Updates: Agents hold dynamic knowledge (beliefs) which is updated according to observations. This is modeled in the Event Calculus using the domain-specific rules for initiates_at/3 provided by the calculus. Below we show a simplified version of how a fluent last_move(Ag2) holding the last move of Ag2, is updated when Ag2 cooperates with agent Ag1 at time T:

```
initiates_at(in(Ag1, cooperate(Ag2, Ag1)),
        Ag1:last_move(Ag2)=cooperate,T)←
        happens_at(in(Ag1, cooperate(Ag2, Ag1)),T).
```

The remaining axioms of the Event Calculus perform the intended update in the knowledge base of Ag1 with the new value at the next time step.

F. Experiments

1) Experimental Setup: To validate our platform, we repeated the experiments of [22] using 1 conductor agent and a population of 17 players implementing 17 deterministic strategies. We varied the rounds from 1 to 1000 for different experiments, with a round consisting of 153 encounters. All experiments were conducted on an Intel(R) Xeon(R) CPU E5-2683 v4 with 2.10GHz and 98GB of RAM.

2) Results Validity: Fig. 5 shows the ranking and accumulated payoff for every strategy over different rounds. Initially, cooperation strategies score poorly and defection strategies score highly. This reverses as the number of rounds increases and, in the end, HARD TFT, MEM2 and GRADUAL rank as the top three strategies and ALLD, PERDDC and PERCD prove to be the three lowest scoring strategies. This is similar to [22], serving as validation for the correctness of the COGNISIM platform.

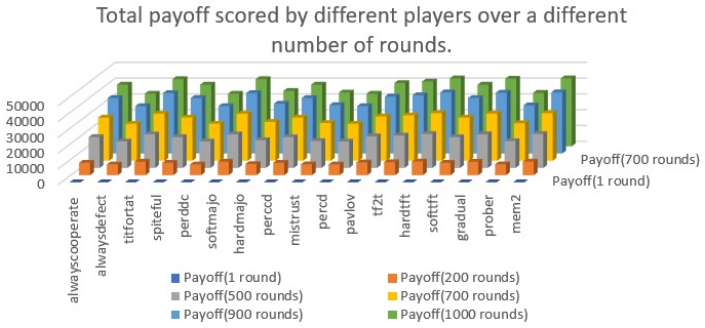
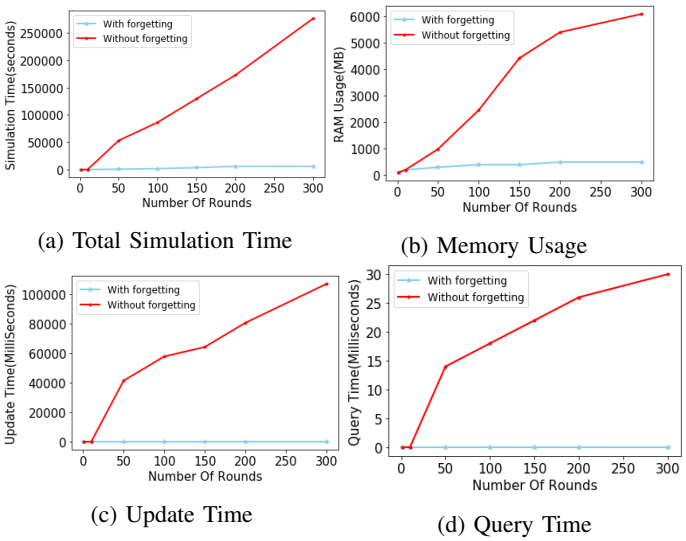


Fig. 5: Strategies Ranking

3) Performance Evaluation: We monitored COGNISIM's total simulation time, update time, query time and memory usage. A round took 612 cycles (∼1683 events were processed) and the execution time and memory usage increased linearly (Fig. 6 (a) and (b)). Also, when using forgetting the simulation run more efficiently in terms of both time and memory than without forgetting; performance deteriorated after 10 rounds. We, therefore, set the optimisation time to 10 rounds.

Fig. 6 (c) and (d) show the update and query times with (✓) and without (×) forgetting. After 300 rounds ∼504,900



(a) Total Simulation Time

(b) Memory Usage

(c) Update Time

(d) Query Time

Fig. 6: Results Of Experiments

events were processed. With forgetting the simulation run time, update time, query time and memory usage were 6058s, 33ms, less than a millisecond and 491.5 MB RAM respectively, clearly suggesting that the underlying event processing is very efficient for large simulations and temporal reasoning. Table II shows the summary of performance evaluation results.

TABLE II: Simulation Results

Rounds	**Total Time(sec)**		**Update Time(ms)**		**Query Time(ms)**		**RAM(MB)**	
	×	✓	×	✓	×	✓	×	✓
1	0.5	0.5	0	0	0	0	98.3	98.3
10	100	100	0	0	0	0	196.6	196.6
50	53210	800	41403	9	14	0	983	295
100	86400	1752	57796	13	18	0	2457.5	393.2
150	129600	3505	64189	20	22	0	4423	393
200	172800	5945	80582	30	26	0	5406.5	491.5
300	276480	6058	106975	33	30	0	6094.6	491.5

V. Conclusions

We have proposed a systematic knowledge representation framework for game-theoretic simulations using cognitive agents. The proposed framework is implemented in COGNISIM intended as a proof-of-concept prototype for our approach. To validate COGNISIM we have shown how to reproduce the results of existing simulation experiments based on a state-of-the-art model from the literature. In the future, we plan to extend the framework with explainable simulations [23] that rely on teleo-reactive strategies for explanation generation, using concepts that humans use to express their explanations [24].

References

[1] M. Grinberg and E. Todorov, "Cognitive agent based simulation platform for modeling large-scale multi-level social interactions with experimental games," *Information Content and Processing*, vol. 3, 2016.

[2] T. Deutsch, T. Zia, R. Lang, and H. Zeilinger, "A simulation platform for cognitive agents," in *2008 6th IEEE International Conference on Industrial Informatics*. IEEE, 2008, pp. 1086–1091.

[3] J. Hopkins, Ö. Kafali, and K. Stathis, "Open game tournaments in STAR-LITE," in *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems, AAMAS'15*, G. Weiss, P. Yolum, R. H. Bordini, and E. Elkind, Eds. ACM, 2015, pp. 1927–1928.

[4] A. Caballero, J. Botía, and A. Gómez-Skarmeta, "Using cognitive agents in social simulations," *Engineering Applications of Artificial Intelligence*, vol. 24, no. 7, pp. 1098–1109, 2011.

[5] A. Borshchev, S. Brailsford, L. Churilov, and B. Dangerfield, "Multi-method modelling: Anylogic," *Discrete-event simulation and system dynamics for management decision making*, pp. 248–279, 2014.

[6] A. Brogi and F. Turini, "Metalogic for knowledge representation," in *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91). Cambridge, MA, USA, April 22-25, 1991*, J. F. Allen, R. Fikes, and E. Sandewall, Eds. Morgan Kaufmann, 1991, pp. 61–69.

[7] R. A. Kowalski and M. J. Sergot, "A logic-based calculus of events," *New Gener. Comput.*, vol. 4, no. 1, pp. 67–95, 1986.

[8] S. Bromuri and K. Stathis, "Distributed agent environments in the ambient event calculus," in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. ACM, 2009, p. 12.

[9] N. Nilsson, "Teleo-reactive programs for agent control," *Journal of artificial intelligence research*, vol. 1, pp. 139–158, 1993.

[10] K. Stathis and M. J. Sergot, "Games as a Metaphor for Interactive Systems," in *People and Computers XI (Proceedings of HCI'96)*, ser. BCS Conference Series, M. A. Sasse, R. Cunningham, and R. L. Winder, Eds. London, UK: Springer-Verlag, August 1996, pp. 19–33.

[11] K. L. Clark, "Negation as failure," in *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. New York: Plemum Press, 1977, pp. 293–322.

[12] K. A. Bowen and R. A. Kowalski, *Amalgamating language and meta-language in logic programming*. Academic Press, 1982, pp. 153–172.

[13] E. T. Mueller, "Event calculus," *Foundations of Artificial Intelligence*, vol. 3, pp. 671–708, 2008.

[14] L. Chittaro and A. Montanari1, "Efficient temporal reasoning in the cached event calculus," *Comput. Intell.*, vol. 12(3), pp. 359–382, 1996.

[15] A. Artikis, M. J. Sergot, and J. V. Pitt, "Specifying norm-governed computational societies," *ACM Trans. Comput. Log.*, vol. 10, no. 1, pp. 1:1–1:42, 2009.

[16] O. Kafali, A. Romero, and K. Stathis, "Agent-oriented activity recognition in the event calculus: An application for diabetic patients," *Computational Intelligence*, vol. 33, no. 4, pp. 899–925, 2017.

[17] S. Bromuri and K. Stathis, "Situating cognitive agents in GOLEM," in *Engineering environment-mediated multi-agent systems*. Springer, 2008, pp. 115–134.

[18] K. Stathis, A. C. Kakas, W. Lu, N. Demetriou, U. Endriss, and A. Bracciali, "PROSOCS: a platform for programming software agents in computational logic," in *Proceedings of the Fourth International Symposium "From Agent Theory to Agent Implementation" (AT2AI-4 – EMCSR'2004 Session M)*, J. Müller and P. Petta, Eds., Vienna, Austria, April "13-16" 2004, pp. 523–528.

[19] A. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni, "Declarative agent control," in *International Workshop on Computational Logic in Multi-Agent Systems*. Springer, 2004, pp. 96–110.

[20] R. Axelrod, "More effective choice in the prisoner's dilemma," *Journal of conflict resolution*, vol. 24, no. 3, pp. 379–403, 1980.

[21] A. Rapoport, A. M. Chammah, and C. J. Orwant, *Prisoner's dilemma: A study in conflict and cooperation*. University of Michigan press, 1965, vol. 165.

[22] P. Mathieu and J.-P. Delahaye, "New winning strategies for the iterated prisoner's dilemma," *Journal of Artificial Societies and Social Simulation*, vol. 20, no. 4, p. 12, 2017.

[23] T. Ahlbrecht and M. Winikoff, "Explaining aggregate behaviour in cognitive agent simulations using explanation," in *International Workshop on Explainable, Transparent Autonomous Agents and Multi-Agent Systems*. Springer, 2019, pp. 129–146.

[24] B. F. Malle, "How the mind explains behavior," *Folk Explanation, Meaning and Social Interaction. Massachusetts: MIT-Press*, 2004.