An Executable Ontology for Social Simulation

Samuel Hill[1] and Ian Horswill[1]

[1] *Northwestern University, 2233 Tech Drive, Evanston, IL, 60208, USA*

Abstract

Social simulations often need many of the same things – characters, locations, interactions, and relationships to name a few. These regular components of simulators are present in games ranging from major titles like *The Sims* [1]–[3], indie classics like *Dwarf Fortress* [4], and experimental interactive experiences such as *Bad News* (or at least in the simulator *Bad News* is build off – *Talk of the Town*) [5], [6]. Not only do these components need reimplementation in each sim, but they are also often difficult for anyone but original developers to modify [7].

In this paper we discuss how social simulators can be declaratively authored from an ontology-based representation while still maintaining playable framerates. We argue that an ontology for these entities allows for the expressive and flexible creation of social simulations. We present a language and ontology, Socialog, which enables one to declaratively author a simulation with these ontological statements while maintaining very good performance.

Keywords

Social simulation, declarative programming, ontology[1]

1. Introduction

The use of declarative programming in game development can provide several benefits such as least-commitment design (leaving decisions about implementation as open as possible), abstraction, and modularity (by way of abstraction). Least-commitment design is a principle that is helpful when handling major design changes that inevitably occur during development. Abstraction and modularity are commonly lauded principles in software development and academia and industry more generally [8], [9]. And rightfully so: it allows flexible and varied usage of the underlying components.

Unfortunately, the major hurdle for declarative programming is often performance - an extra sticky point in game development where milliseconds matter. We believe there is a "sweet spot" when developing social simulations where declarative programming can still be performant. Using a bottom-up logic programming language [10] we can build social simulators almost entirely declaratively. However, while this is a high-level abstraction over the normal implementation of social sims, this bottom-up evaluation language only operates at the level of predicates.

While predicates allow for the description of logical relations and properties, ideally one would be able to make ontological statements about the social simulation where the descriptions are of what stuff exists, how things relate, and when events can happen. Socialog is a language that allows one to build a simulation with these ontological statements while maintaining performance by compiling down to TED.

Figure 1 shows a fragment of *Voix de la Ville* – the example simulation built in Socialog – that uses these highly abstracted ontological statements to tell the simulation how locations both come to and cease to exist. This can be translated to say "There exist *places* that are locations with some attributes like category and position. A place starts existing when a location (building) was just created, and that place ceases to exist probabilistically after 40 years (if the location is not a permanent location like a Cemetery or City Hall)." This fragment governs locations – and as such changes to the logic of when new locations start, or end can be implemented with minor changes to the code.

2. Related work

Many AI-based games have used symbolic rules for character control. *MKULTRA* [11] was written primarily in Prolog, save for the graphics and UI code. *City of Gangsters* [12] used another top-down logic programming language, albeit with an exotic implementation. Many other games have used some kind of rule engine. *The Sims 3* [1] used a rule-based system to script the interactions between situations, personality traits, and actions available to a given character [13]. *Façade* [14], [15] was implemented primarily in a reactive planning language, *ABL* [16], however its internal working memory included a

AIIDE Workshop on Experimental Artificial Intelligence in Games, October 08, 2023, University of Utah, Utah, USA

samuelhill2022@u.northwestern.edu (S. Hill);
ian@northwestern.edu (I. Horswill)

```
var Place = Exists("Place", location, locType.Indexed, locCategory.Indexed,
                   position.Indexed, businessStatus.Indexed, founding)
    .StartWhen(CreatedLocation)
    .StartCauses(Add(Place.Attributes[location, locType, locCategory, position,
                 InBusiness]).If(CreatedLocation, LocationInformation))
    .EndWhen(Place.Attributes, !In(locType, PermanentLocationTypes),
             Place, Time.YearsOld[founding, age], age > 40, PerYear(0.1f))
    .EndCauses(Set(Place.Attributes, location, businessStatus, OutOfBusiness));
```

Figure 1: Locations in *Voix de la Ville*.

forward-chaining production system. Several other systems have used forward-chaining rule-based systems such as *Comme Il Faut* [17], [18], the social simulation engine (implemented in JavaScript) upon which *Prom Week* [19] was built, and the *Ensemble Engine* [20], *CiF's* successor.

Several game development frameworks and social simulation middleware have used symbolic rules, particular for interactive narrative. One of the earliest and most influential such systems is Nelson's *Inform 7* language [21], [22], which allows designers to build interact narrative systems, particularly simulationist systems, using declarative statements. The *Versu* simulationist narrative system [23] used a custom logic programming language, *Praxis*, which was based on an exotic modal logic called eremic logic (aka exclusion logic) [24]. More recently, the *Lume* system [25] made extensive use of Prolog's definite clause grammars [26], [27] for text generation. Lapeyrade has also used Prolog for better character decision making [28].

To our knowledge, bottom-up logic programming has not previously been used to implement social simulations. However, Datalog has been used for story-sifting [29], the process of searching the output of a social simulator for interesting narrative content. Bottom-up logic programming, and Datalog in particular, has received the most attention in the database community [30]–[33] where its appeal came partly from the ability to compile it into relational algebra operators for efficient execution on classical database architectures, and because it can be extended to recursive rules using a fixed-point evaluation algorithm. This allows it to compute transitive closure (e.g., reachability in a graph), which standard relational algebra cannot. This is where most of the original research on the language and its implementation was done. More recently, it has seen extensive use for the semantic web [34].

Games involving large-scale social simulation are relatively rare. The best known is *Dwarf Fortress* [35], which supports real-time simulator of small hundreds of characters. Achieving this level of performance requires implementation in C++ and significant programmer effort to optimize cache locality and minimize the number of pointer indirections. *RimWorld* is a very similar game that also involves social simulation for the purpose of storytelling [36]. In the research literature, the best-known system is Ryan's *Talk of the Town* [37], which was used in the award-winning game *Bad News* [5]. TotT was a batch simulation of the growth of a small American town over the course of 140 years, ending with population around 400 people using a time-varying level of detail. It was implemented in Python and required many minutes to simulate a city. More recently, Johnson-Bey has developed *Neighborly* [38], a more modular and modifiable implementation based on an entity-component-system architecture [39]. With the possible exception of *RimWorld*, these systems run the simulation in a single thread.

Kismet [7] is a rapid-prototyping system for social simulations intended for casual users. It used answer-set programming (a type of logic programming) internally. However, its focus was on allowing casual users to build social simulations, rather than on trying to maximize performance.

In the process of deciding on an ontological hierarchy we followed the guidelines of the Common Core Ontology (CCO) for creating ontologies [40], [41]. We also used the popular ontologies *DOLCE* [42], the *Basic Formal Ontology* (*BFO*) [43], and *Cyc* [44], [45] as comparison points for the creation of our ontology. For certain concepts like events and time there were additional ontological considerations.

Time is an inherent component of any simulation, governing the level of detail (LOD) for events and actions [46]. *Kismet* allows for the temporal "cycle" to

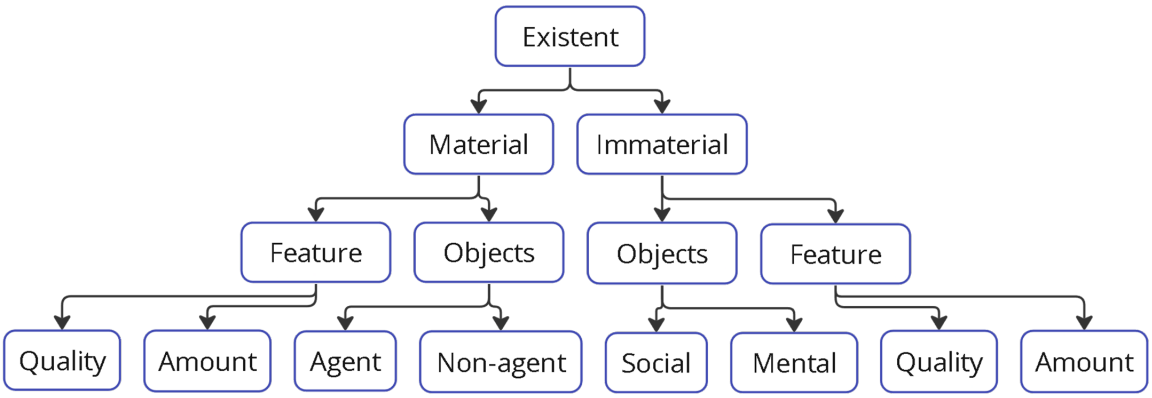

Figure 2: Simplified ontology based on *DOLCE*, *Cyc*, and *BFO*.

be authored [7], while the *Web Ontology Language* (*OWL*) from the W3C has a time ontology [47] that has been extended for non-Gregorian calendar applications [48] which helped to address the possibility for alteration of temporal granularity. Events are another important consideration as there ontological categorization is a hotly debated subject matter in philosophy, linguistics, and cognitive science [49]. Inspiration for the event structure was largely taken from Davidson's logical formalism for actions and causal relation [50], [51].

3. The Ontology

There are a number of best practices in the construction of ontologies, such as adopting a "realism-based" approach or having at a multi-tiered architecture, that helped to guide our construction of this executable ontology [40], [41]. A realism-based approach means we are modeling the entities in the world that our data refers to rather than directly modeling the data elements and their respective relationships like in traditional database design [52], [53]. This technique helps to shift concerns away from implementation and toward declarability. A tiered architecture supports modularity with a common breakdown of levels including an upper ontology that defines generic types of entities, a mid-level that uses the upper ontology concepts to define structures that are common to many domains, and lastly a domain-level that defines domain specific concepts [40], [41]. We have defined a simple upper ontology to guide our designs, built a mid-level ontology as a declarative programming language, and we have an example domain-level system in the research game *Voix de la Ville*.

3.1. High-level ontology

General AI ontologies include very abstract high-level concepts, including some kind of top-level concept of which all other concepts are instances, For example, in *DOLCE* everything maps to a Particular [42], in *Cyc* everything maps to a Thing [44], [45], and in *BFO* everything maps to a continuant [43]. The ontology in Figure 2 is still far more detailed than is needed for the creation of a simulation but reflects the type of taxonomy one would find in a high-level ontology as it pertains to the concept of existence (here called existent, also known as a continuant or endurant).

Since most of the distinctions in Figure 2 are irrelevant to our needs, we do not have an explicit ontology at this level. Socialog is a mid-level ontology. However, it does assume a top-level ontology something like Figure 3.

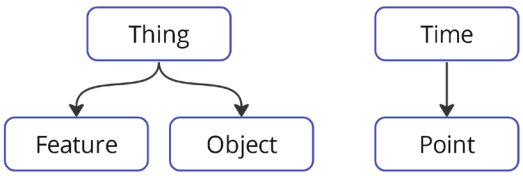

Figure 3: High-level ontology for Socialog.

3.2. Mid-level ontology

The mid-level ontology for our system represents a majority of Socialog's capabilities, though some functionality in Socialog isn't included in this ontology section (such as table operators). Additionally, Effects – used in Events – are used to describe state evolution in the system and are not directly related to the high-level ontology.

3.2.1. Event

Events occur and can cause Effects. Events are parameterized by participants, the Things involved in the event. Events are represented by an occurrence predicate, parameterized by the participants, which is true when that event with those participants is occurring in the current simulation step. A chronicle also stores a record of the occurrence of every event.

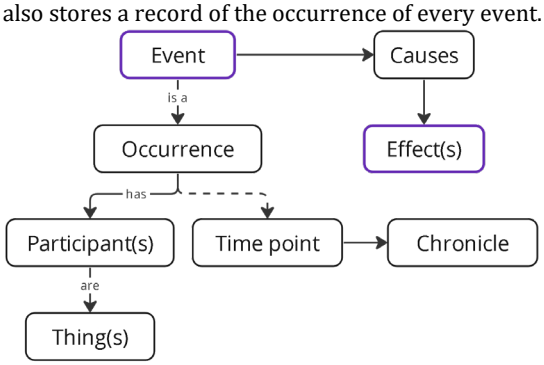

Figure 4: Event ontology – component of the mid-level ontology.

3.2.2. Existent

Existents are objects that have a temporal duration, i.e., a start and an end. Existents can also have features attributed to them.

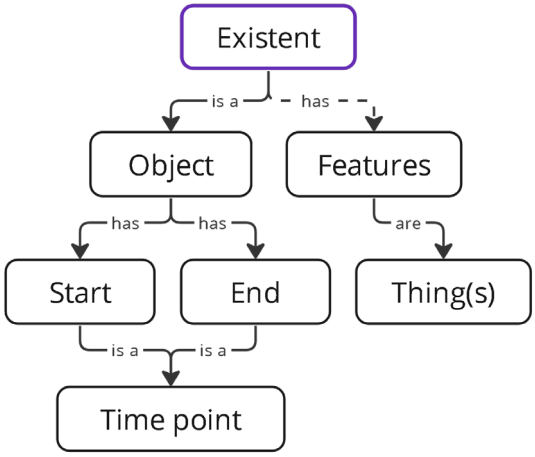

Figure 5: Existent ontology – component of the mid-level ontology.

3.2.3. Relationship

Relationships are pairings that have a value (e.g., strength) attributed to them. The flavor of relationships that uses a numeric strength value we

call Affinities and the flavor using a Boolean state value are called Relationships. Not included in Figure 6 is the ability of Relationships (the bool variety) to be chronicled much like an Existent. Relationships, naturally, have a start and an end but unlike regular existents these relationship states can come into and out of being repeatedly, meaning (potentially) multiple start and end times for a single relationship.

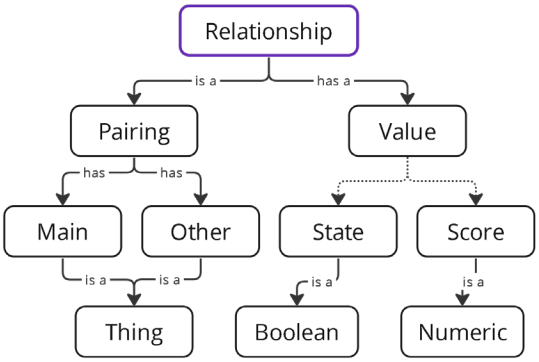

Figure 6: Relationship ontology – component of the mid-level ontology.

4. Voix de la Ville

We are using this ontology to build a simulator called Voix de la Ville (shown in Figure 7). The simulation is a proof-of-concept partial reimplementation of Talk of the Town [5] that also takes inspiration from Dwarf Fortress' depth of interactions and relationships [4].

Simulation proceeds, as all simulations do, by ticks. Following TotT we have two ticks per day, but like Kismet [7] or the OWL-Time ontology [47], [48] the relationship between ticks and clock time can be changed by the designer by editing the definition of Time. Each tick of the clock everyone in town goes to work or school if scheduled or obligated. Everyone else chooses an activity to partake in and subsequently selects a location nearest to their house that can accommodate the chosen activity.

Occasionally, individuals who are romantically interested in each other will go out on a date. While at a location, everyone can choose someone else at the same location to interact with – preferring individuals most similar to themselves in addition to friends and romantic partners. These interactions can range from neutral chatting to positive empathizing to negative dueling, and each of the interactions affects the relationship between its participants.

An individual can have both a platonic and a romantic affinity to another individual, with various associated relationships like friendship, enemy, or romantic partners. Additionally, romantic partners that are also good friends can get married which is an exclusive relationship (although there is still a chance for cheating). The cycle of moving about the town and interacting is how socialization happens and drives the formation of our social networks.

There are 44 location types across 10 categories, each with a schedule of days and times of day they are open. There are 62 vocations that apply to the business locations and each business has an ideal number of positions for each vocation. Every person has an aptitude score for each vocation that is used to assign the best person for an open shift of some job. Some locations are not businesses like houses and as such do not employ anyone. Houses are an extra special case as they also are a type of accommodation – people live in houses (in addition to apartments and inns). Each house has an occupancy and as they fill up members of the household can leave – prioritizing keeping nuclear family units intact – for newly constructed homes. Also, when people die, they are buried in the town cemetery.

New people can be created when a couple goes on a very successful date. After conception approximately nine months pass and the mother gives birth to the baby. This is the major way in which a town progresses. However, drifters can also come settle in

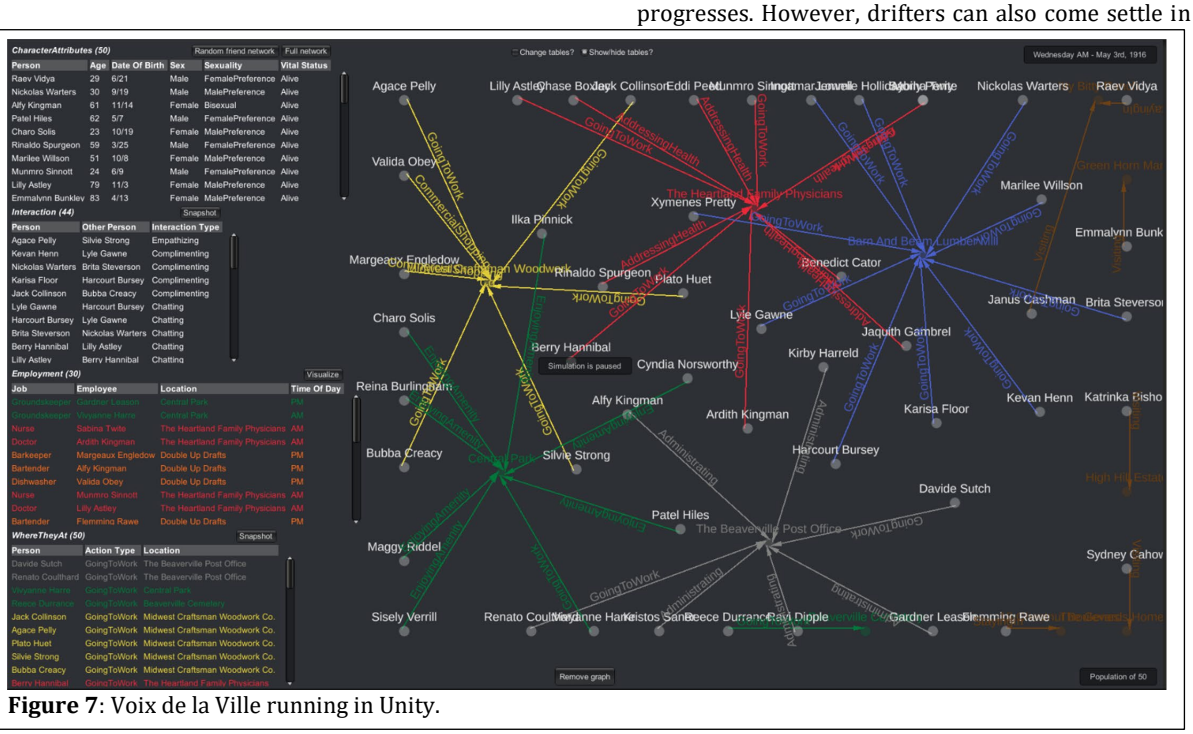

Figure 7: Voix de la Ville running in Unity.

```
var ForeclosedUpon = Event("ForeclosedUpon", occupant)
    .OccursWhen(Home[occupant, location], Place.End[location])
    .Causes(Set(Home, occupant, location, newLocation)
                .If(RandomElement(UnderOccupied, newLocation)));
```

Figure 8: Foreclosure in Voix de la Ville.

town (more come to town when there are ample jobs to fill with a new business). This cycle of birth and death is also applicable to location; new locations are created as the population grows due to population thresholds, density, or various specific cases (Doctors office only opens when you have a really good Doctor in town). Locations can also go out of business, and this forces people to either move to a new home or to find a new job.

The core simulation code – written in Socialog – is under 500 lines with comments. This does not include the Unity/GUI specific code, the TED and Simulog libraries, and ignores some custom value types and utils (mainly declaring enumeration types with some time structs/classes).

5. Socialog

Simulog is implemented as a layer on top of TED [10], a high-performance, bottom-up logic programming language embedded in C#. Like Datalog, TED stores the extensions of predicates in tables that can be queried like databases; in what follows we will use the terms table and predicate largely interchangeably. Socialog internally maintains tables of events, existents, and relationships that are akin to a relational database. As such, the diagrams of Socialog's implementation will include a sort of table notation to imply this underlying structure.

Much of the ontology surrounds the concepts of events and existents. An event is a kind of thing that can happen at a point in time, such as birth or death. An existent is a kind of thing that exists during some interval of time, such as a person. The start and end of an existent are events. For example, the start of a person is their birth.

5.1. Event

Events are both a standalone predicate that allows one to say what should be occurring on a given tick as well as the underlying mechanism that all other Socialog predicates use to express when something should happen. Events can declare Effects, which are additions or modifications to other predicates.

Figure 8 shows foreclosure and can be translated to say "There is an event called foreclosed upon that an occupant (person) participates in when the home they live at is ending (will no longer exist). When foreclosed upon, move the occupant of that place to another home that isn't currently overcrowded." "Place.End" inside Figure 8 is referring to the End event for the Place Existent, while "Home" is a standard Predicate that simply tracks who lives where.

Figure 9 outlines the Event architecture in Socialog, where each event that is defined has an internally maintained table with columns for each participant. A Chronicled Event is derived from its respective event, storing every occurrence of the event ever instead of just the occurrences in the current tick. The chronicle is created automatically when the simulation includes rules that refer to the history of the event (by saying, e.g., "ForeclosedUpon(who).At(time)").

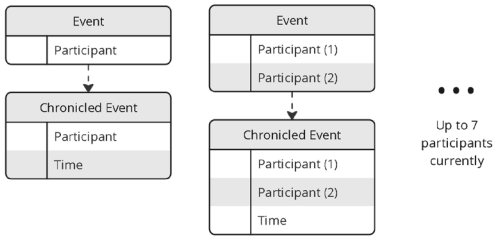

Figure 9: Event architecture.

5.2. Existent

Existents are things that exist with some temporal extent, not instantaneous events but also not universals with no durative quality. Characters and Places are good examples of existents (Figure 1 shows the Place existent), things with a clear start and end.

Existents are made up of several tables with three different events, the table of existents (with a Boolean exists column for quick access of those that do exist as well as those that did), and an optional table of attributes for the existent. Figure 10 outlines this architecture – the ellipses under the attributes table indicate that there can be a variable number of features and, like the limitation described in Figure 9, there can be up to 7 features in the attributes table.

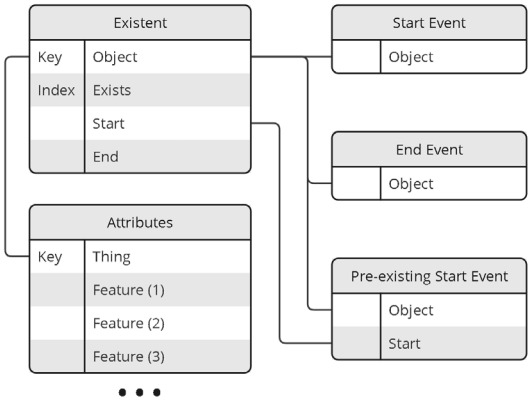

Figure 10: Existent architecture.

Figure 11 shows the event for procreation, the embryo existent, and the connection between the embryo and character existents. This can be translated to say "Embryos are a thing that exist (with some temporal extent), and they have the features of a mother, father, sex, and date of conception. Embryos start when Procreation happens – Procreation happens when a woman and man successfully procreate, and this assigns a sex and name to the child.

```
var Procreation = Event("Procreation", woman, man, sex, child)
    .OccursWhen(SuccessfulProcreation[woman, man], RandomSex[sex],
                RandomFirstName[sex, firstName],
                NewPerson[firstName, Surname[man], child]);
var Embryo = Exists("Embryo", child, woman.Indexed, man, sex, conception.Indexed)
    .StartWhen(Procreation)
    .StartCauses(Add(Embryo.Attributes[child, woman, man, sex,
                     Time.CurrentDate]).If(Procreation))
    .EndWhen(Embryo[child], Embryo.Attributes,
             Time.NineMonthsPast[conception], Prob[0.8f])
    .EndCauses(Add(Parent[parent, child])
                   .If(Embryo.Attributes[child, parent, __, __, __]),
               Add(Parent[parent, child])
                   .If(Embryo.Attributes[child, __, parent, __, __]));
Character.StartWhen(Embryo.End[person]);
```

Figure 11: Procreation in Voix de la Ville.

An embryo ends after 9 months have passed with an 80% probability per tick to simulate labor. When an embryo ends, the child is added to the Parent table as a child of each parent and the embryo becomes a person in the Character existent."

Of note here is the ability to rewrite the "Character.StartWhen(Embryo.End[person]);" line as "Embryo.EndCauses(Character.Start[person]);". Separating out these effects into relevant groupings is yet another example of the modularity found in making these sorts of ontological statements. Effects are critical to Socialog's expressive ability, allowing predicates to be modified by code omnidirectionally – instead of setting a column or adding a row in the code for a given predicate, we can do the same thing from any of the predicates that are referencing it.

5.3. Affinity

Affinity is a type of relation between two things where the relationship has an associated score, or affinity. The Affinity architecture is rather simple, as shown in Figure 12, with a base table storing all affinities and a change event that indicates when an affinity needs to be updated and by what amount.

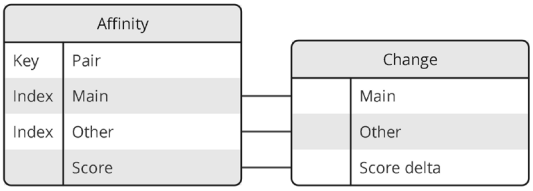

Figure 12: Affinity architecture.

Figure 13 shows the Spark and Charge affinities which translate to platonic and romantic affinity and are lifted from TotT. While this code could be made even shorter by storing the interaction types and associated deltas in a dictionary or table, the readability of the update when functions would be compromised.

Although this example implies a use case of person-to-person affinity, this architecture is flexible enough to let one establish an affinity between any two classes of things. For example, spiritual or institutional affinity as is found in Dwarf Fortresses Deity, Object of Worship, or Force relationships. Another example would be the needs system found in The Sims where each need could be an affinity that scores action urgency.

5.4. Relationships

Relationships come in several varieties, but the most basic is depicted in Figure 15. A relationship can come into and out of being, and the way in which a

```
var Charge = Affinity("Charge", pairing, person, otherPerson, charge).Decay(0.8f)
    .UpdateWhen(InteractionOfType(Empathizing), charge == 900)
    .UpdateWhen(InteractionOfType(Assisting), charge == 800)
    .UpdateWhen(InteractionOfType(Complimenting), charge == 300)
    .UpdateWhen(InteractionOfType(Chatting), charge == 80)
    .UpdateWhen(InteractionOfType(Insulting), charge == -250)
    .UpdateWhen(InteractionOfType(Arguing), charge == -750)
    .UpdateWhen(InteractionOfType(Fighting), charge == -900)
    .UpdateWhen(InteractionOfType(Dueling), charge == -1200);
var Spark = Affinity("Spark", pairing, person, otherPerson, spark).Decay(0.1f)
    .UpdateWhen(InteractionOfType(Procreating), spark == 1500)
    .UpdateWhen(InteractionOfType(Snogging), spark == 1200)
    .UpdateWhen(InteractionOfType(Courting), spark == 900)
    .UpdateWhen(InteractionOfType(Flirting), spark == 150)
    .UpdateWhen(InteractionOfType(Negging), spark == -120)
    .UpdateWhen(InteractionOfType(Insulting), IsRomantic, spark == -750);
```

Figure 13: Affinity in Voix de la Ville.

```
Friend = Charge.Relationship(nameof(Friend), state, 5000, 4000);
Enemy = Charge.Relationship(nameof(Enemy), state, -6000, -3000);
Romantic = Spark.Relationship(nameof(Romantic), state, 7000, 6000);

var Lover = ExclusiveRelationship("Lover", symmetricPair, person,
                                  otherPerson, state)
    .StartWhen(Friend[person, otherPerson], Friend[otherPerson, person],
               Romantic[person, otherPerson], Romantic[otherPerson, person])
    .EndWhen(Character.End[person], Character[otherPerson]);
```

Figure 14: Relationships in Voix de la Ville.

relationship starts or ends is decided by a start and end event. While this information is not stored in the Relationship table, a Relationship Chronicle can be made in much the same way as an Event Chronicle and this Chronicle tracks relationship starts and ends (even if they start and end multiple times).

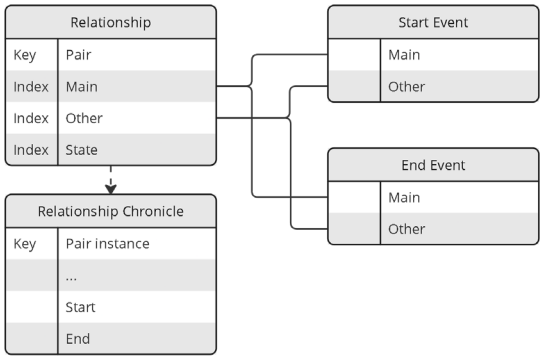

Figure 15: Basic relationship architecture.

In addition to this basic relationship type, there is a symmetric variant – e.g., where A relating to B is the same relationship as B relating to A – that uses the same structure as a relationship but internally maintains pair symmetry. One variant on the symmetric relationship is the exclusive relationship that, in addition to using the same basic structure and maintaining symmetry, also maintains exclusivity (a given individual can maintain the relationship with at most one other individual). By using the same structure for all relationship types, switching the logic of a relationship from the normal asymmetric type to a symmetric type is as easy as changing the constructor function name and/or pairing variable.

5.4.1. Affinity Relationship

Another special case relationship is the affinity relationship. Affinity is already tracking some value for a pairing and a common way that we would create relationships was by using thresholds for the start and end values in some affinity table. To both simplify this process and ensure a performant implementation we added affinity relationships. While structured much the same as other relationship variants, there is no start/end when function as this is handled by the affinity start and end.

Figure 14 shows three affinity relationships and an exclusive relationship that is derived from the other relations. The affinity relationship lines translate to "there is a relationship named *Name* that starts at the affinity score *value 1* and ends at the affinity score *value 2*." In other words, this relationship starts when you cross one threshold and ends when you cross another. The lines for the Lover relation say, "there is an exclusive relationship called lover that starts when two people consider each other to be both friends and romantic partners and ends when one of the two dies." It would be easy enough to make Lover non-exclusive (use Relationship not ExclusiveRelationship), but Lover acts as our analog for marriage currently.

6. Visualization

While there is not any user intervention in the simulation yet, the visualization tools we have built are not only helpful in debugging but also entertaining to watch. There are currently two major visualizers in Voix de la Ville – Table viewers and Graph viewers. Predicates' underlying implementation is as tabular data so a basic table viewer running during the simulation allows for inspection of the contents of the entire sim in neat tabular forms. While table viewers are a critical component of our debug/visualization capabilities, the point of Socialog is to build social sims and thus we naturally form social networks. A graph visualizer can be used for anything from family trees and friend networks to the dataflow diagram of how predicates connect to one another. Both display techniques are shown in Figure 7.

Tilemaps from Unity are also used to display locations. Although this isn't as useful for debugging – only really showing how the town is physically laid out – you can hover over each tile and see some basic information including who is currently at the location.

7. Performance

In addition to the expressivity and modularity demonstrated by Socialog, the performance is acceptable, if not competitive with hand crafted code. The data shown in figures 16 through 18 were collected in one 15-minute run of the simulation. The simulation was running on a desktop PC with an i7-7700k CPU clocked at 4.6GHz (4.8 boost) and 32GB of 1,500 MHz RAM. While running inside of Unity, the data collected only measured the execution time of the Socialog code not any graphical computations (this is why the GPU information is not relevant).

Figure 16 shows the overall performance of Socialog in Voix de la Ville as a function of population. On the low end of the population spectrum – 50 characters – we have 15-25ms execution times per tick while the high end – 450 characters – takes roughly 600ms per tick. This reflects a sum of the per-entity updates, which grows linearly with the population, and

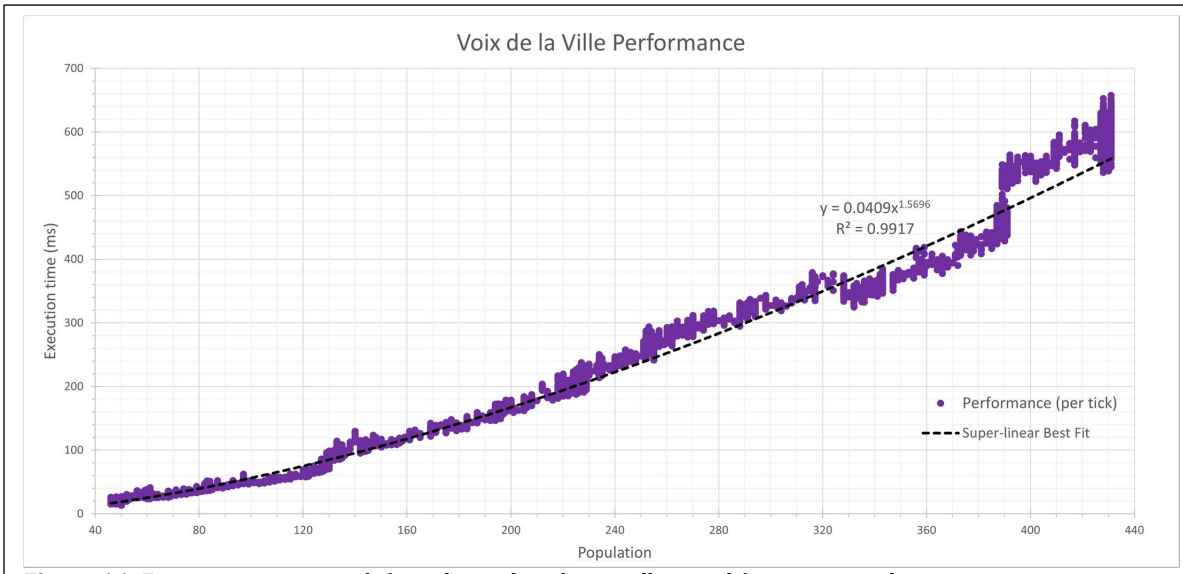

Figure 16: Execution time per tick (socialog code only, in milliseconds) against population.

the per-character-pair updates, such as affinities, which necessarily grow quadratically with the population. The *per capita* performance data is shown in Figure 17. On the low end of the population spectrum, we have 0.4ms execution per character and by the time we are at the high end that has only increased to 1.4ms.

Figure 18 shows the total run that collected this data. Over a period of 37 in-simulation years the population grew (shown by the green points) and with it the execution time (shown by the purple points). Clearly visible are some plateaus in population followed by sharp growth (this is caused by new locations attracting "drifters" to fill the jobs). Regardless of this inconsistent growth, the performance data heavily correlates with population. Additionally of note, the population can be capped to help maintain a higher performance – a technique present in Dwarf Fortress with a common high end of roughly 200 dwarves [54].

While the performance of Socialog is already acceptable for numerous use cases, there are two major bits of apologia that can address potential improvements. One is that the technique that drives Socialog, bottom-up logic programming, is amenable to parallelization and that is not something we have implemented yet. Second is that we have not done an optimization pass over TED, Socialog, and Voix de la Ville to ensure that we are eking as much performance out of these systems as possible.

8. Future Work

Although we handle the quadratic nature of interaction reasonably well in terms of performance, there are doubtless better abstractions for addressing this problem. Similarly, there are modules that have yet to be built out for concepts such as ownership and part/whole relations. More patterns are likely to emerge in the development of Voix de la Ville, and these will inform future ontological categories and components.

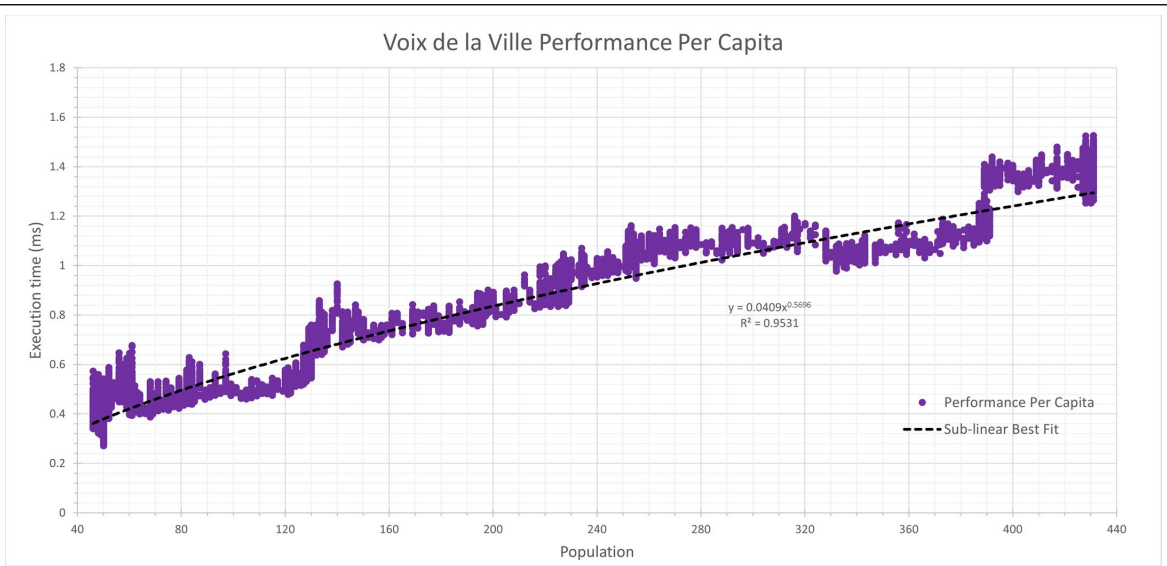

Figure 17: Execution time *per capita* (socialog execution time divided by population) against population.

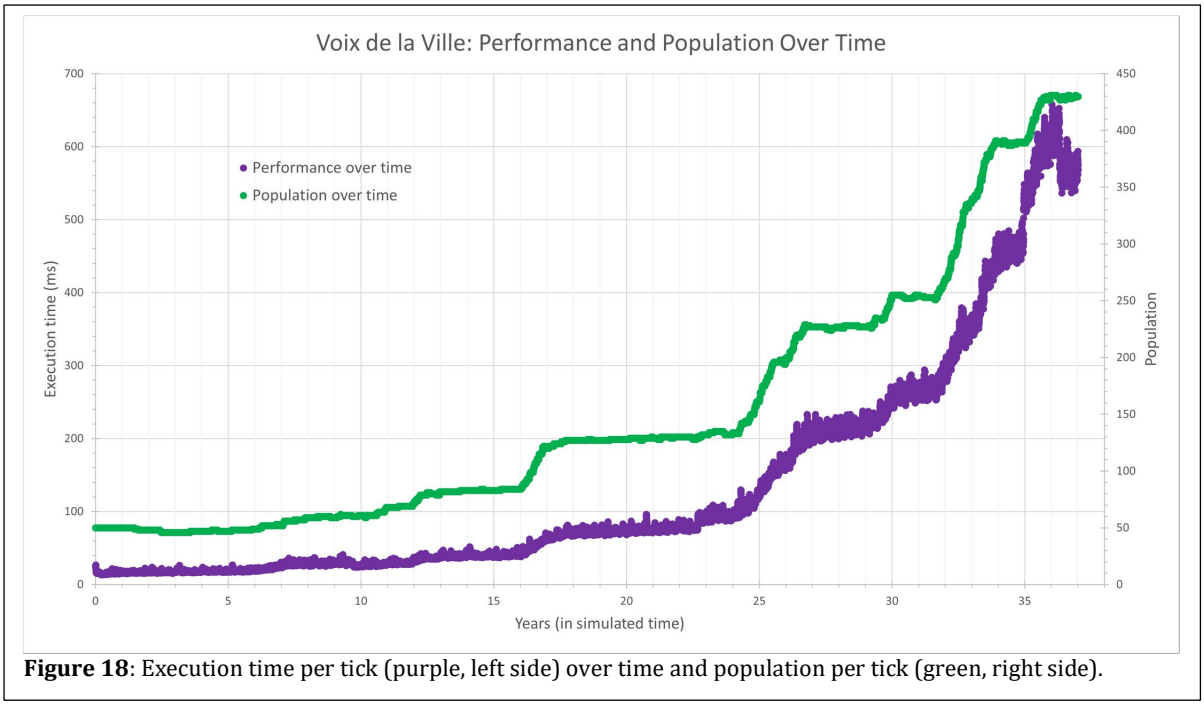

Figure 18: Execution time per tick (purple, left side) over time and population per tick (green, right side).

While a full-fledged STRIPs planner is likely out of the picture for character control (for performance reasons), some form of action planning that goes beyond single-tick action section is also worth investigating. Level of Detail (LOD) support, such as in Talk of the Town, would also be useful. This would allow the system to simulate some parts or eras of the world more coarsely than others.

9. Conclusion

Social simulation is an important emerging area of gameplay, but building these simulators is difficult. Socialog shows that it is possible to author these sims at a remarkably high level while still maintaining acceptable performance.

References

[1] "The Sims 3." Maxis, 2009.

[2] D. Nutt and D. Railton, "The Sims: Real Life as Genre," *Inf. Commun. Soc.*, vol. 6, no. 4, pp. 577–592, Dec. 2003, doi: 10.1080/1369118032000163268.

[3] S. Johnson-Bey, M. J. Nelson, and M. Mateas, "Exploring the Design Space of Social Physics Engines in Games," in *Interactive Storytelling*, M. Vosmeer and L. Holloway-Attaway, Eds., in Lecture Notes in Computer Science, vol. 13762. Cham: Springer International Publishing, 2022, pp. 559–576. doi: 10.1007/978-3-031-22298-6_36.

[4] "DF2014:Relationship - Dwarf Fortress Wiki." https://www.dwarffortresswiki.org/index.php/DF2014:Relationship (accessed Jul. 17, 2023).

[5] B. Samuel, J. Ryan, A. Summerville, M. Mateas, and N. Wardrip-Fruin, "Bad News: An Experiment in Computationally Assisted Performance," Nov. 2016. doi: 10.1007/978-3-319-48279-8_10.

[6] B. Samuel, A. Summerville, J. Ryan, and L. England, "A Quantified Analysis of Bad News for Story Sifting Interfaces," in *Interactive Storytelling*, A. Mitchell and M. Vosmeer, Eds., in Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 142–156. doi: 10.1007/978-3-030-92300-6_13.

[7] A. Summerville and B. Samuel, "Kismet: A Small Social Simulation Language".

[8] S. Brusoni *et al.*, "The power of modularity today: 20 years of 'Design Rules,'" *Ind. Corp. Change*, vol. 32, no. 1, pp. 1–10, Feb. 2023, doi: 10.1093/icc/dtac054.

[9] M. Sonego, M. E. S. Echeveste, and H. Galvan Debarba, “The role of modularity in sustainable design: A systematic review,” *J. Clean. Prod.*, vol. 176, pp. 196–209, Mar. 2018, doi: 10.1016/j.jclepro.2017.12.106.

[10] I. Horswill and S. Hill, “Fast, Declarative, Character Simulation Using Bottom-Up Logic Programming,” presented at the AIIDE Workshop on Experimental Artificial Intelligence in Games, University of Utah, Utah, USA, Oct. 2023.

[11] I. Horswill, “Postmortem: MKULTRA, An Experimental AI-Based Game,” *Proc. AAAI Conf. Artif. Intell. Interact. Digit. Entertain.*, vol. 14, no. 1, Art. no. 1, Sep. 2018, doi: 10.1609/aiide.v14i1.13027.

[12] “City of Gangsters.” SomaSim, Chicago, 2021.

[13] R. Evans, “AI challenges in Sims 3,” *Artif. Intell. Interact. Digit. Entertain.*, 2009.

[14] M. Mateas and A. Stern, “Façade.” 2005.

[15] M. Mateas and A. Stern, “Façade: An Experiment in Building a Fully-Realized Interactive Drama”.

[16] M. Mateas and A. Stern, “A behavior language for story-based believable agents,” *IEEE Intell. Syst.*, vol. 17, no. 4, pp. 39–47, Jul. 2002, doi: 10.1109/MIS.2002.1024751.

[17] J. McCoy, M. Treanor, B. Samuel, N. Wardrip-Fruin, and M. Mateas, “Comme il Faut: A System for Authoring Playable Social Models,” *Proc. AAAI Conf. Artif. Intell. Interact. Digit. Entertain.*, vol. 7, no. 1, pp. 158–163, Oct. 2011, doi: 10.1609/aiide.v7i1.12454.

[18] J. McCoy, M. Treanor, B. Samuel, B. Tearse, M. Mateas, and N. Wardrip-Fruin, “Comme il Faut 2: a fully realized model for socially-oriented gameplay,” in *Proceedings of the Intelligent Narrative Technologies III Workshop*, Monterey California: ACM, Jun. 2010, pp. 1–8. doi: 10.1145/1822309.1822319.

[19] J. McCoy, M. Treanor, B. Samuel, A. A. Reed, N. Wardrip-Fruin, and M. Mateas, “Prom week,” in *Proceedings of the International Conference on the Foundations of Digital Games*, in FDG ’12. New York, NY, USA: Association for Computing Machinery, May 2012, pp. 235–237. doi: 10.1145/2282338.2282384.

[20] B. Samuel, A. A. Reed, P. Maddaloni, M. Mateas, and N. Wardrip-Fruin, “The Ensemble Engine: Next-Generation Social Physics”.

[21] G. Nelson, “Inform 7.” 2006.

[22] G. Nelson, “NATURAL LANGUAGE, SEMANTIC ANALYSIS AND INTERACTIVE FICTION.”

[23] R. Evans and E. Short, “Versu—A Simulationist Storytelling System,” *IEEE Trans. Comput. Intell. AI Games*, vol. 6, no. 2, pp. 113–130, Jun. 2014, doi: 10.1109/TCIAIG.2013.2287297.

[24] R. Evans, “Introducing Exclusion Logic as a Deontic Logic,” in *Deontic Logic in Computer Science*, G. Governatori and G. Sartor, Eds., in Lecture Notes in Computer Science, vol. 6181. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 179–195. doi: 10.1007/978-3-642-14183-6_14.

[25] S. Mason, C. Stagg, and N. Wardrip-Fruin, “Lume: a system for procedural story generation,” in *Proceedings of the 14th International Conference on the Foundations of Digital Games*, San Luis Obispo California USA: ACM, Aug. 2019, pp. 1–9. doi: 10.1145/3337722.3337759.

[26] F. C. N. Pereira and D. H. D. Warren, “Definite clause grammars for language analysis—A survey of the formalism and a comparison with augmented transition networks,” *Artif. Intell.*, vol. 13, no. 3, pp. 231–278, May 1980, doi: 10.1016/0004-3702(80)90003-X.

[27] S. P. Harrison, “Review of Prolog and Natural Language Analysis,” *Language*, vol. 64, no. 3, pp. 627–631, 1988, doi: 10.2307/414538.

[28] S. Lapeyrade, “Reasoning with Ontologies for Non-player Character’s Decision-Making in Games,” *Proc. AAAI Conf. Artif. Intell. Interact. Digit. Entertain.*, vol. 18, no. 1, Art. no. 1, Oct. 2022, doi: 10.1609/aiide.v18i1.21980.

[29] M. Kreminski, M. Dickinson, and N. Wardrip-Fruin, “Felt: A Simple Story Sifter,” in *Interactive Storytelling*, R. E. Cardona-Rivera, A. Sullivan, and R. M. Young, Eds., in Lecture Notes in Computer Science, vol. 11869. Cham: Springer International Publishing, 2019, pp. 267–281. doi: 10.1007/978-3-030-33894-7_27.

[30] S. Ceri, G. Gottlob, and L. Tanca, “What you always wanted to know about Datalog (and never dared to ask),” *IEEE Trans. Knowl. Data Eng.*, vol. 1, no. 1, pp. 146–166, Mar. 1989, doi: 10.1109/69.43410.

[31] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases*, 1st ed. in Addison Wesley. Pearson, 1994. Accessed: Jul. 16, 2023. [Online]. Available: http://webdam.inria.fr/Alice/pdfs/all.pdf

[32] S. Ceri, G. Gottlob, and L. Tanca, *Logic Programming and Databases*. in Surveys in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990. doi: 10.1007/978-3-642-83952-8.

[33] J. D. Ullman, *Principles of database and knowledge-base systems*. in Principles of computer science series. Rockville, Md: Computer Science Press, 1988.

[34] G. Gottlob, G. Orsi, A. Pieris, and M. Šimkus, “Datalog and Its Extensions for Semantic Web Databases,” in *Reasoning Web. Semantic Technologies for Advanced Query Answering: 8th International Summer School 2012, Vienna, Austria, September 3-8, 2012. Proceedings*, T. Eiter and T. Krennwallner, Eds., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 54–77. doi: 10.1007/978-3-642-33158-9_2.

[35] T. Adams and Z. Adams, “Slaves to Armok: God of Blood Chapter II: Dwarf Fortress.” Bay 12 Games, 2006.

[36] T. Sylvester, “RimWorld.” Ludeon Studios, Oct. 2018.

[37] J. Ryan, “Curating Simulated Storyworlds,” Univercity of California Santa Cruz, 2018.

[38] S. Johnson-Bey, M. J. Nelson, and M. Mateas, “Neighborly: A Sandbox for Simulation-based Emergent Narrative,” in *2022 IEEE Conference on Games (CoG)*, Aug. 2022, pp. 425–432. doi: 10.1109/CoG51982.2022.9893631.

[39] “Entity Systems are the future of MMOG development – Part 1 – T-machine.org,” Jul. 31, 2013. https://new.t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/ (accessed Jul. 21, 2023).

[40] R. Rudnicki, “An Overview of the Common Core Ontologies,” White Paper, Feb. 2019. [Online]. Available: https://www.nist.gov/system/files/documents/2021/10/14/nist-ai-rfi-cubrc_inc_004.pdf

[41] R. Rudnicki, B. Smith, T. Malyuta, and W. Mandrick, “Best Practices of Ontology Development,” White Paper, Oct. 2016. [Online]. Available: https://www.nist.gov/system/files/documents/2021/10/14/nist-ai-rfi-cubrc_inc_002.pdf

[42] S. Borgo *et al.*, “DOLCE: A descriptive ontology for linguistic and cognitive engineering,” *Appl. Ontol.*, vol. 17, no. 1, pp. 45–69, Jan. 2022, doi: 10.3233/AO-210259.

[43] J. N. Otte, J. Beverley, and A. Ruttenberg, “Basic Formal Ontology: Case Studies”.
[44] D. Lenat, M. Prakash, and M. Shepherd, “CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlenecks”.
[45] D. Ramachandran, P. Reagan, and K. Goolsbey, “First-orderized ResearchCyc: Expressivity and efficiency in a common-sense ontology,” Jan. 2005.
[46] J. F. Allen, “Towards a general theory of action and time,” *Artif. Intell.*, vol. 23, no. 2, pp. 123–154, Jul. 1984, doi: 10.1016/0004-3702(84)90008-0.
[47] “Time Ontology in OWL.” Nov. 15, 2022. [Online]. Available: https://www.w3.org/TR/owl-time/
[48] S. J. D. Cox, “Time ontology extended for non-Gregorian calendar applications,” *Semantic Web*, vol. 7, no. 2, pp. 201–209, Feb. 2016, doi: 10.3233/SW-150187.
[49] R. Casati and A. Varzi, “Events,” *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2023. Accessed: Jul. 15, 2023. [Online]. Available: https://plato.stanford.edu/archives/fall2023/entries/events/
[50] D. Davidson, “The Logical Form of Action Sentences,” in *Essays on Actions and Events*, 1st ed.Oxford University PressOxford, 2001, pp. 105–148. doi: 10.1093/0199246270.003.0006.
[51] D. Davidson, “Causal Relations,” *J. Philos.*, vol. 64, no. 21, pp. 691–703, 1967, doi: 10.2307/2023853.
[52] B. Smith, “New Desiderata for Biomedical Terminologies,” *J. Biomed. Inform. - JBI*, Jan. 2008.
[53] B. Smith and W. Ceusters, “Ontological realism: A methodology for coordinated evolution of scientific ontologies,” *Appl. Ontol.*, vol. 5, no. 3–4, pp. 139–188, Nov. 2010, doi: 10.3233/AO-2010-0079.
[54] “What is your favourite population cap?” http://www.bay12forums.com/smf/index.php?topic=167690.0 (accessed Jul. 24, 2023).