Proceedings of the Twenty-First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2025)

Declarative Specification of Temporal Entities for Large-Scale Character Simulation

Samuel Hill, Ian Horswill
Northwestern University
samuelhill2022@u.northwestern.edu, ian@northwestern.edu

Abstract

Large-scale character simulations for games such as Dwarf Fortress and Bad News involve both large numbers of interacting sub-simulations, and very large numbers of entities being simulated. This combination of software complexity and real-time performance constraints make them difficult to develop.

One recurring issue in these systems is the simulation of temporal entities: entities that come into and out of existence. The system must typically track the set of temporal entities that exist in the current simulation step, their states, the events of their creation and destruction, and, when needed, the set of all temporal entities that have ever existed.

In this paper, we present Simulog, a very high-level declarative language for large scale character simulation based on ideas from the formal ontology literature. Simulog allows developers to declaratively specify temporal entities, and their relationships. It compiles these specifications to logic programs that are then compiled to C# for multithreaded execution, providing very high performance.

Introduction

There is an emerging interest in city-scale character simulation. Commercial games with such city-scale character simulations include *The Sims 4* (Maxis 2014), which has a maximum recommended character limit of 200, the colony simulator *Dwarf Fortress* (Adams and Adams 2006), which frequently runs in the small hundreds of characters, and the business simulator *City of Gangers* (SomaSim 2021), which defaults to approximately 1000 characters. In the research literature, Ryan's *Talk of the Town* (Ryan 2018), (aka TotT) used in the award-winning game *Bad News* (Samuel et al. 2016), simulates the growth of a small American town over the course of 140 years, ending with a population around 400 people while using a time-varying level of detail. Although these "cities" are more like small towns than true cities, they nonetheless provide a sense of immersion in a "real" world with a life of its own beyond the player's actions. These unique aesthetics make them very interesting for game design.

Unfortunately, such large-scale character simulations are very difficult to build. First, they involve large numbers of game subsystems that interact in unpredictable manners, making them difficult to debug and even more difficult to balance. Second, the sheer number of simulated entities makes it very difficult to achieve real-time performance. TotT doesn't run in real time. *Dwarf Fortress* does, however it is optimized down to the level of arranging data structure layout to minimize cache misses. It's not surprising, therefore, that its author considers it only 50% complete after 23 years of development. As a result, deployed large-scale character simulations are rare.

There has been some work in recent years to make LCCSs easier to build. *Neighborly* (Johnson-Bey et al. 2022) was a more modular and modifiable implementation of TotT that could be used as a toolkit for building TotT variants. Other approaches involved the use of more declarative languages. *Kismet* (Summerville and Samuel 2021) allowed casual users to build small-scale social simulations that compile into Answer-set Prolog (Gebser et al. 2012). While not practical for large-scale simulations, it provided a language that was accessible to non-programmers. *City of Gangsters* used a custom implementation of top-down logic programming in what is probably the largest-scale simulation in a commercial game. Although only used for social reasoning, it demonstrated that a declarative language could be performant at scale in a commercial game. More recently, Horswill and Hill describe the use of bottom-up logic programming for constructing full simulators (Horswill and Hill 2024) with execution speed comparable to equivalent native C# code.

One issue with logic programming is that it is atemporal in a certain sense. It describes problems in terms of objects (terms) and predicates (relations between them), but it has no notion of time, change, causality, creation, or destruction

Listing 1: Example existent `BehaviorTreeExampleSim.cs`

```
1  Plant = Exists("Plant", plant).StartWhen(NeedNewPlant, MakeNewPlant[plant])
2                                .EndWhen(Harvested);
```

per se. These are core issues in games and similar simulations.

These notions can be represented using predicates, of course. For example, fluent properties can be represented by predicates with an extra time parameter. Objects that can be created and destroyed can be represented by an existence predicate that holds when a given object exists at a given timepoint. And rules can be added stating the conditions under which objects do and do not exist. However, implementing such design patterns is a repetitive, error-prone process and requires the programmer to commit to a particular representation of temporal entities that would be difficult to change after the fact.

It would be preferrable to have a declarative language with a native understanding of time and temporal entities, one that was able to take over the housekeeping duties of tracking what objects exist, when they are created and deleted, and so on. However, such declarative languages are difficult to find. The closest approximations probably come from the formal ontology literature (Lenat et al. 1985; Ramachandran et al. 2005; Otte et al. 2021; Borgo et al. 2022). Upper ontologies typically build in types for objects that persist through time and the rules that regulate them.

In this paper, we describe *Simulog* – a declarative, domain-specific language for simulating temporal entities using lessons from the formal ontology literature. Simulog is built on top of a logic programming language (TED), adding to it first-class notions of time, events, object creation, and so on. It allows declarative statements such as "characters exist in time" and "businesses are destroyed by bankruptcy."

Simulog is fast: its constructs compile into TED, which in turn compiles to parallel C# code, which compiles of course to native code. It is compatible with Unity and other game engines based on the Common Language Runtime (Miller and Ragsdale 2003).

Listing 1 shows a fragment of a simulation built in Simulog. It says, "*Plants* exist in time, a new *plant* starts existing on each tick while the population of plants is less than 200, and that *plant* ceases to exist once harvested (the logic of which is dictated by the `Harvested` event)." This fragment governs plants – and as such changes to the logic of when new plants start, or end can be implemented with minor changes to the code.

Semantics of Temporal Entities

General ontologies include very abstract high-level categories, usually this includes some kind of top-level category of which all other categories are instances. For example, in *DOLCE* everything maps to a Particular (Borgo et al. 2022), in *Cyc* everything maps to a Thing (Lenat et al. 1985; Ramachandran et al. 2005), and in *BFO* everything maps to a Continuant (Otte et al. 2021). We focus here on temporal categories, leaving others, such as a top category or BFO's distinction between Material and Immaterial concepts, to future work. Specifically, the temporal categories will include timepoints, events, continuants, (stateful) features, and temporal relationships.

Time and Temporal-Types

Since games are generally discrete-time simulations, we will assume that the set of **timepoints** $\mathbb{T}$ is order-isomorphic to the natural numbers.

A **temporal entity** is any object that can be considered to exist at some timepoints but not others. Events, persons, and friendships are temporal entities. Natural numbers and the color green are not.

A **temporal type** $\boldsymbol{X} = (X, E_X)$ is a set X of temporal entities, equipped with an existence relation $E_X \subseteq X \times \mathbb{T}$ such that xE_Xt holds precisely when $x \in X$ exists at timepoint t. We will write $x \in \boldsymbol{X}$ (x is an element of the temporal type) to mean $x \in X$ (x is an element of the underlying set).

Note that the existence relation E_X is defined relative to a given run of the simulation. However, what we want is for different runs to have different patterns of entities being created and destroyed at different timepoints. This could be easily remedied by making E_X a ternary relation between entities, timepoints, and simulations runs. However, we believe this notation would make the definitions more cumbersome without improving clarity.

Given some temporal entity $x \in \boldsymbol{X}$, the **lifetime** of the entity is the set of timepoints $L_{\boldsymbol{X}}(x) = \{\, t \mid xE_Xt \,\}$ during which the entity exists. Conversely, the **population** $P_X(t) = \{\, x \mid xE_Xt \,\}$ is the set of entities from the type that exist at timepoint t. We also say that the entity x has a **start** timepoint t where xE_Xt but not $xE_X(t-1)$, where $t-1$ is the timepoint immediately before t. Conversely, it has an **end** timepoint t where not xE_Xt but $xE_X(t-1)$.

Temporal Kinds

We will distinguish three different kinds of temporal types based on the structure of their instances.

We will say temporal type $\boldsymbol{X}$ is an **event type** if for all $x \in \boldsymbol{X}$, $L_{\boldsymbol{X}}(x)$ is a singleton, an **existent type**, if $L_{\boldsymbol{X}}(x)$ is an interval, or an **intermittent type** otherwise (as depicted in

Figure 1). Events start and end share the same timepoint, existents have unique start and end timepoints, and intermittent types can have multiple starts and ends – allowing them to come into and out of existence throughout the simulation.

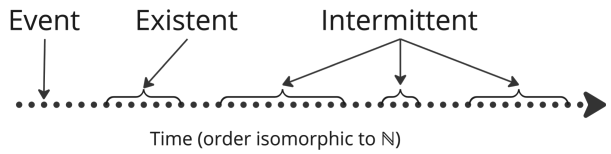

Figure 1: Time and the 3 Temporal kinds

Relationships in Time

A **temporal relation** on temporal types $\boldsymbol{X} = (X, E_X)$, $\boldsymbol{Y} = (Y, E_Y)$ is a temporal type $R = (X \times Y, E_{XY})$ for which at any timepoint t, $(x, y)E_{XY}t \Rightarrow xE_Xt \wedge yE_Yt$, that is, if the relation holds at t then both x and y must exist at that same time t. We will define the induced relation $R_t = \{(x, y) \mid (x, y)Rt\}$ to be the pairs of entities for which R holds at time t.

We extend the notions of one-to-one, many-to-one, etc. relations to temporal relations in the natural way: R is one-to-one if R_t is one-to-one for all t, many-to- one if R_t is many-to-one for all t, etc. Similarly, for homogenous temporal relationships (i.e. relations between a temporal type and itself) R is symmetric if R_t is symmetric for all t. As a special case for homogenous relations, we say that R is exclusive if R_t is both symmetric and one-to-one.

Features of Temporal Entities

A V-valued **feature** of a temporal type $\boldsymbol{X}$ is a function $F: X \times \mathbb{T} \to V$, where $(x, t, v) \in F \Rightarrow xE_Xt$, i.e. F is only defined for a given x within its lifetime. As with relations, we define $F_t: X \to V$ to be $F_t(x) = F(x, t)$.

A V-valued **affinity** between temporal types $\boldsymbol{X}$ and $\boldsymbol{Y}$ is a function $A: X \times Y \times \mathbb{T} \to V$ where $(x, y, t, v) \in A \Rightarrow xE_Xt \wedge yE_Yt$, i.e. A is only defined for a given x and y within their lifetimes. Again, we define $A_t: X \times Y \to V$ to be $A_t(x, y) = A(x, y, t)$.

As with relations, we extend notions such as symmetry to affinities in the natural way, i.e. A is symmetric if A_t is symmetric at all timepoints t.

The Simulog Language

Once again, the goal of Simulog is to support declarative assertions about time, change, and causation in addition to conventional logic programming rules and predicates, and to do so with sufficient performance to support large numbers of entities with acceptable frame rates.

Simulog is an embedded language within C#, meaning that Simulog code is itself C# code. It is strongly typed and compiles to native code. It provides first-class support for temporal entities, including events, existents, and temporal relationships, as well as features of existents.

Simulog works by compiling declarations about temporal entities into sets of predicates and rules for an underlying Datalog-like language (TED), which is then compiled into C#. These predicates can be used freely in the underlying logic programming language and/or C# code. The system is effectively a large in-memory database that can be freely queried and updated in Datalog.

Simulog surfaces the existence predicates of temporal types as two different Datalog predicates, one for the population at the current timestep, and which therefore includes only the currently existing entities of that type, and another (the type's "chronicle"), giving the lifetimes of all instances throughout the history of the simulation. Since the chronicle is potentially huge, Simulog only generates chronicling code for types whose chronicles are used by the game code.

Timepoints

Simulog assumes a `Timepoint` data type implemented in C#. Simulog has been used with both integer and floating-point time units, but `Timepoint` may also be defined as something more exotic. It must define a `Current` value giving the timepoint value of the current game tick, and an `Eschaton` value representing the maximum possible `Timepoint` value.

Developers may wish to add additional operations such as subtraction to yield a time interval, or addition of `Timepoints` to intervals. However, Simulog does not require these.

Features

Features of temporal entities are the roles, parameters, or state variables, or other information that describes the entity. Simulog does not have a "feature" type *per se*, however each of the temporal kinds below can be declared to carry some specific set of features with it.

Events

`Event` expressions introduce new types of events and are of the form:

```
Event(typeName, features …)
      .OccursWhen(conditions).Causes(effects)
```

The optional `OccursWhen` and `Causes` clauses describe the causal relationships between the event and the rest of the simulation. The former defines a set of sufficient conditions for triggering (causing) the event, and the latter defines a set of effects it causes.

Listing 2: Character Ends `VoixDeLaVille.cs`

```
1  DieOfOldAge = Event("DieOfOldAge", person)
2                .OccursWhen(Character[person], AgeOf, age > 60, PerMonth(0.003f));
3  DieMidLife = Event("DieMidLife", person)
4               .OccursWhen(Character[person], AgeOf, age > 18, age <= 60, PerMonth(0.001f));
5  Character.EndWhen(DieOfOldAge).EndWhen(DieMidLife);
```

A Chronicled Event is created automatically when the simulation includes rules that refer to the history of the event (by saying, e.g., "`Event(`*features*…`).At(`*time*`)`…").

Effects and Causation

Several clauses in Simulog, such as the `Causes` clause of the `Event` declaration, define effects that are caused by some event in the simulation. Such effects may be:

- Causing (instantiating) events, including the creation and destruction of temporal entities.
- Updating the features of entities.
- Asserting or retracting statements in the underlying logic programming language.

Listing 2 gives event definitions governing character death in a TotT-like simulator. `DieOfOldAge` (lines 1-2) occurs with a 0.3% chance per month for characters over 60. `DieMidLife` (lines 3-4) has a 0.1% change per month for characters ages 18-60. Line 5 states that these destroy the `Character`. This could also be achieved by adding `.Causes(Character.End[person])` to lines 1 or 2 and 3 or 4.

Existents

Existent expressions introduce new types of existents and are of the form:

```
Exists(typeName, object, features …)
       .StartWhen(conditions)
       .StartCauses(effects)
       .EndWhen(conditions)
       .EndCauses(effects)
```

Again, the optional `Start/EndWhen` clauses specify when to create/destroy instances of the type and `Start/EndCauses` specify effects that are triggered by creation and destruction. The type can be queried using several Datalog predicates, *type*`[`*object*`,` *features* …`]`, which is true when an instance of the type with the specified features exists in this tick, and *type*`[`*object*`,` *features* …`,` *start*`,` *end*`]` holds when an instance with the specified features had the specified lifetime. As well, *type*`.Start/.End[`*object*`,` *features* …`]` holds when an instance with the specified features was created/destroyed this tick.

Listing 3 shows the definition of the `Place` existent in a TotT-like. Places here represent buildings, not spatial positions, which are called "locations" in the system. Lines 1-2 declare the type and its features; ignore the `.Indexed` annotations which govern code optimization. Lines 3-5 define when places should be created and destroyed: they are created when the `CreatedLocation` event occurs and destroyed with a probability of 10% per year when the location is more than 40 years old and not listed as permanent.

Temporal Relations

Temporal relations are, again, binary relations that can vary over time. They are an intermittent type because a given instance of the relationship, such as a friendship between two characters can be created, destroyed, and then recreated. While intermittent temporal types could be made into first class objects, the only relevant applications that we have found thus far are for relationships. Temporal relations are defined by `Relationship` expressions:

```
Relationship(name, left, right)
             .StartWhen(conditions)
             .StartCauses(effects)
             .EndWhen(conditions)
             .EndCauses(effects)
```

where *left* and *right* specify the types of the arguments to the relation. As with existents, relationships automatically define `Start` and `End` events, and both chronicled and unchronicled existence predicates.

Listing 3: Place Existence `VoixDeLaVille.cs`

```
1  Place = Exists("Place", location, locationType.Indexed, locationCategory.Indexed,
2                  position.Indexed, businessStatus.Indexed, founding)
3      .StartWhen(CreatedLocation)
4      .EndWhen(Place.Attributes, !In(locationType, permanentLocationTypes),
5               PlaceAgeOf, age > 40, PerYear(0.1f));
```

Listing 4: Spark and Charge Affinities `VoixDeLaVille.cs`

```
1  Interaction = Event("Interaction", person, otherPerson, interactionType);
2  Charge = Affinity("Charge", person, otherPerson, charge).Decay(0.8f)
3                   .UpdateWhen(Interaction, In(interactionType, platonicInteractions),
4                                InteractionAffinityDelta[interactionType, charge]);
5  Spark = Affinity("Spark ", person, otherPerson, spark).Decay(0.8f)
6                  .UpdateWhen(Interaction, In(interactionType, romanticInteractions),
7                              InteractionAffinityDelta[interactionType, spark])
8                  .UpdateWhen(InteractionOfType(Insulting), IsRomantic, spark == -750);
```

There are also specialized declarations that create symmetric relationships and exclusive (symmetric, one-to-one) relationships.

Affinities

The `Affinity` declaration introduces a new numeric affinity between pairs of objects:

```
Affinity(name, left, right, value)
        .UpdateWhen(conditions)
        .UpdateCauses(effects)
        .Decay(decayRate)
```

Again, *left* and *right* specify the types of the arguments to the affinity, and *value* specifies the type of its value. While general affinities are supported, `int` and `float` values are the most common use case we have found and as such all of the example affinities found in Listing 4 will be integer affinities. The optional `UpdateWhen` and `UpdateCauses` specify causes and effects of changes to the affinity. And the optional `Decay` clause – which is only valid on `int` and `float` affinities – specifies that the value of the affinity should asymptotically decay to zero in the absence of updates.

Listing 4 shows the definitions of two affinities between characters used in TotT and similar simulators, and how they update based on interactions between those characters. `Charge` is platonic attraction and `Spark` is romantic attraction. Both decay slowly when characters don't interact but increase or decrease when the appropriate kinds of interactions occur between the two characters.

The Interaction event as well as the Spark and Charge affinities are very rough proxies for platonic and romantic attraction which are heavily inspired by TotT mechanisms of the same name. Line 1 shows the definition of the Interaction event, not including the conditions under which it occurs. Lines 2-4 show the definition of the Charge affinity (an integer affinity), the decay rate, and the conditions under which the affinity should update – translating the conditions to roughly "update when an interaction of the platonic type occurs, and update by the default delta amount in `InteractionAffinityDelta`.".

Lines 5-8 show the definition of the Spark affinity – a similar definition to Charge – with the additional update condition for insulting (an interaction type that can happen platonically, but when it happens between romantic partners has a different effect than the default assigned to that interaction).

Although this example implies a use case of person-to-person affinity – and while we noted in the definition of temporal relations and affinities that ***X*** and ***Y*** must be temporal entities (things that exist in time) – 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 – the needs here being abstract concepts like hunger or want to go to the movies not temporal entities.

Affinity Relationships

Affinities can be thought of as graded relationships. Whereas Relationships are binary, they either hold or don't between two objects at a given point in time, affinities assign a numeric strength. A common use case is to build a binary Relationship from a graded Affinity by thresholding it, perhaps with hysteresis. Given an affinity *a*, we can create a relationship from it using the declaration:

```
a.Relationship(name, startThresh, endThresh)
```

This creates a relationship that starts when the affinity goes above *startThresh* and is destroyed when it falls below *endThresh*.

Listing 5 gives the definitions of four relationships between characters. Three – `Friend`, `Enemy`, and `Romantic` – are derived directly from affinities. Because their end thresholds are smaller in absolute value than their start thresholds, they have some hysteresis. Lines 4-6 define the `Lover` relation to start when two characters are friends, romantic toward each other, and each not lovers with anyone else. Lines 7 says the death of either character ends the relationship. Changing `ExclusiveRelationship` to just

Listing 5: Affinity Relationships `VoixDeLaVille.cs`

```
Friend = Charge.Relationship("Friend", 5000, 4000);
Enemy = Charge.Relationship("Enemy", -6000, -3000);
Romantic = Spark.Relationship("Romantic", 7000, 6000);
Lover = ExclusiveRelationship("Lover", person, otherPerson)
    .StartWhen(Friend[person, otherPerson], Friend[otherPerson, person],
               Romantic[person, otherPerson], Romantic[otherPerson, person])
    .EndWhen(Character.End[person], Character[otherPerson]);
```

`Relationship` would switch the simulation to support polyamory. Adding additional `EndWhen` clauses would add situations that could end the relationship.

Implementation

For this implementation, the declarative language that Simulog is embedded in is TED – a high-performance, bottom-up logic programming language inspired by Datalog and embedded in C#, see Figure 2:

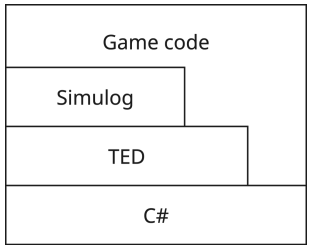

Figure 2: Hierarchy of Simulog embeddings and backing for various functions in a game codebase

Simulog is strongly typed, supports higher-order predicates, parallel execution, and optional native code compilation among several other features. Embedding allows transparent C# interoperation, since Simulog code is itself C# code. It also allows Simulog to use IDE support for C#, such as type checking, refactoring, and so on. Simulog is open source and available at https://github.com/SamuelHill/Simulog.

Evaluation

Much of the appeal of logic programming for game developers lies in its conciseness and abstractness, which facilitates rapid development and experimentation (Zubek et al. 2021; Smith and Mateas 2011). Simulog aims to facilitate even higher-level abstractions within its domain of competence while still allowing extremely high performance. We will discuss each of these issues in turn.

Performance

We have used Simulog to build 3 systems: a real time Sims-like environment supporting several thousand NPCs running needs-based AI, a small dwarf-like for testing behavior trees, and a town-like (TotT inspired city simulator) called Voix de la Ville (VdlV). We will focus our performance evaluation on the latter as descriptions of the capabilities of each system would be confusing (as each is used as the testbed for various improvements or features to this technique) and the complexity of VdlV acts as a compelling case study.

Voix de la Ville supports the following existents, affinities, and temporal relations:

Temporal Type	Count
Characters	~400
Locations	~100
Housing assignments	~400
Employment	~300
Friendships	~2,500
Romances	~400
Lovers	~100
Spark instances	~15,000
Charge instances	~60,000

Counts are approximate since they change every tick, but these are representative of a city with 400 characters (where we end testing). Voix de la Ville also includes a large number of events, too many to list here.

We can achieve the performance shown in Figure 3 on an Apple M3 Pro when running VdlVs compiled Simulog code in parallel for ~50,000 ticks. When just looking at the ticks where the population was over 400, we find an average execution time of 45.6ms or roughly 22fps with a standard deviation of 7.7ms (meaning 99.7% of the time we at least have ~15fps). We can achieve 60fps or higher up until a population of over 200 characters. This performance, given the complexity of the system, is something we find to be acceptable for a laptop.

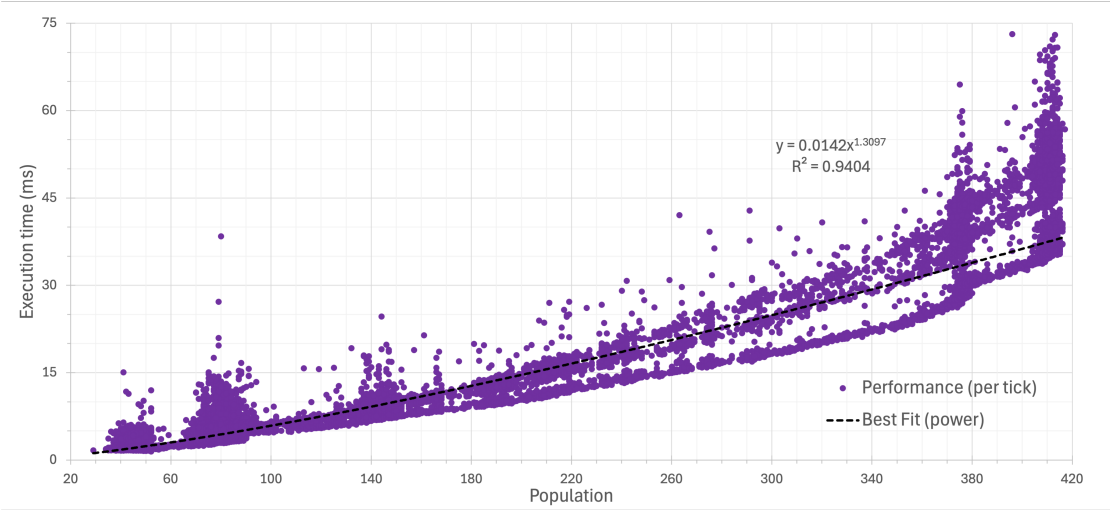

Figure 3: Voix de la Ville performance plotted against population.

Abstraction

To evaluate the abstraction level of Simulog code we use inverse code size as a proxy for abstraction level, that is, we take longer line counts to indicate less abstraction.

First, we compare VdlV, whose core simulator is 403 lines of combined Simulog and TED code (including 49 lines of comments). This compiles to 8,365 lines of C# code. The machine generated C# code is less dense than that which would be written by a human programmer. However, even with a conservative assumption that the machine generated code is 5x longer than a human would write, the declarative version is 4x more compact.

To evaluate Simulog relative to TED, we can compare the Simulog code for the single existent in Listing 1 to the TED code it expands into shown in Listing 6. Here we see that Simulog is roughly 3x more compact than TED. Note that this is the *minimum* possible TED code. The actual TED code generate by Simulog could be twice as long depending on how the existent is used by the surrounding code. For example, if the Population Count is used this adds 4 more lines.

Related Work

The only specific work we are aware of combining logic programming and ontologies with games is (Lapeyrade and Rey 2023) that used Prolog and an ontology coded in OWL to play Wumpus. They report execution times in the 10-20ms range for a single character. On the face of it, this is orders of magnitude slower than Simulog. However, playing Wumpus is a very different problem from large-scale character simulation and it's unclear how much can be inferred from this.

One can argue that ontologies have always formed a core part of text-based interactive fiction games even if they haven't necessarily been described in such terms. Nelson's *Inform 7* language (Nelson 2006a; 2006b), which is the current most popular IF language, can be seen as a hybrid declarative/procedural language combined with an upper ontology specialized to interactive fiction. Its standard library has built-in support for concepts such as agents, rooms, portals, devices, switches, gravity, containment, locking, lighting and darkness, etc.

Similarly, there has been some use of logic programming, without ontologies, in game development. *MKULTRA* (Horswill 2018) was written primarily in Prolog, save for the graphics and UI code. The *Lume* system (Mason et al. 2019) was also Prolog-based. *Versu* (Evans and Short 2014)

Listing 6: Plant Existence Equivalent in TED

```
1  Plant = Predicate("Plant", plant, startTime, endTime, state);
2  PlantStart = Predicate("PlantStart", plant).If(NeedNewPlant, MakeNewPlant[plant]);
3  PlantEnd = Predicate("PlantEnd", plant).If(Harvested);
4  Plant.Add[existent, true, time, Eschaton].If(PlantStart, CurrentTimePoint[time]);
5  Plant.Set(existent, end, time).If(PlantEnd, CurrentTimePoint[time]);
6  Plant.Set(existent, exists, false).If(PlantEnd);
```

used a custom logic programming language, *Praxis* (Evans 2010), as did *City of Gangsters* (2021). TED (Horswill and Hill 2024), the target language into which Simulog compiles, was used in *Rise of Industry 2* (2025).

While not technically logic programming, a number of games have used production systems or other rule-based systems. These include *The Sims 3* (2009) used a rule-based system to script the interactions between situations, personality traits, and actions available to a given character (Evans 2009). Several other systems have used forward-chaining rule-based systems such as *Comme Il Faut* (McCoy et al. 2011; 2010), the social simulation engine (implemented in JavaScript) upon which *Prom Week* (McCoy et al. 2012) was built, and the *Ensemble Engine* (Samuel et al. 2015), *CiF's* successor. *Façade's* (Mateas and Stern 2005; 2003) internal working memory included a forward-chaining production system.

As mentioned previously, there are relatively few deployed games that involve large-scale character simulation. Of the commercial games, the best known is *Dwarf Fortress* (Adams and Adams 2006). *RimWorld* (Sylvester 2018) has similar gameplay in some ways, although the character counts are much lower. *City of Gangsters* (2021) involves on the order of 1000 NPCs per game. In the research literature, *Bad News* (Samuel et al. 2016) and *Talk of the Town* (Ryan 2018) simulate a small-town's history over 140 years, ending with character counts on the order of 400 NPCs.

There are smaller numbers of tool intended to facilitate building large-scale character simulations. Johnson-Bey *Neighborly* (Johnson-Bey et al. 2022) can be used as a Python-based construction set for batch simulators like *Talk of the Town*. *Kismet* (Summerville and Samuel 2021) is a rapid-prototyping system for social simulations. However, its focus was on allowing casual users to build social simulations. It cannot support large-scale simulations.

Future Work

We believe high level declarative modeling is a fruitful approach to large-scale character simulation. We have focused on modeling of temporal entities here because they occur in nearly all games and nearly all games model time as a discrete series of ticks, be they integer or floating-point values. That said, there are a few other systems that may be fruitful to model.

Space

The natural complement to modeling temporal entities is to model spatial ones. Here, the primary difficulty is that games do have relatively different models of space. Most obviously, some games are 2D, others 3D. Some have integer-value coordinates, some continuous. Some treat objects as being single tiles or treat positions as single tiles. Others allow objects that span tiles or are arbitrary polygons or NURBS.

This may argue for an ontology that ignores metric issues entirely and instead uses a qualitative model of space. As mentions previously, most text-based interactive fiction games already models space this way, in terms of containers, part/whole relationships, and so on.

Action

Simulog does not presently model actions because it's difficult to divorce the representation of actions from the action-selection architecture in which they will be used (e.g. behavior trees vs. utilities vs. GOAP, etc.). Nonetheless, a minimal but extensible system that could represent preconditions and effects of actions could be very useful. *Kismet* (Summerville and Samuel 2021) could be fruitful model for this.

Time

There are several incremental improvements that could be made to Simulog's temporal entity system. Temporal relationships and affinities could be extended to n-ary relations and symmetric affinities (e.g. representing employment as a ternary character/role/employer relation in VdlV). Existents could be modified to support predestined death times: this is a common design pattern in games; rather than choosing each tick whether a character should die, you decide in advance when they'll die by sampling some distribution.

Simulog events are currently instantaneous. Durative events that have distinct start and end timepoints might be useful. These would have more complex causal properties, i.e. effects that happen at the start, vs. the end or all through the duration of the action. Support for recurring events (the sun rising every day) or intermittent events (hammering a nail) might be useful.

Explicit support for temporal level of detail, whereby different objects are simulated at different tick rates, would also be useful if it can be formulated in a sufficiently game-independent manner.

Conclusion

Researchers and practitioners have looked to logic programming and other declarative programming techniques for ways to write abstract, high-level code for various tasks in the development of video games. However, it is challenging to make these systems run in real-time in video games.

Simulog provides an even higher-level (more abstract) language than traditional logic programming, specialized to describing temporal entities in large-scale character simulations. It provides direct support for temporal entities, events, and causal relations. Yet it does so very efficiently and has been shown to support hundreds of characters at video framerate.

Appendix

The full set of temporal kinds, relationships, and features is:

Temporal Kind	Simulog Predicates
event	Event
existent	Existent
intermittent	Relationship
intermittent	SymmetricRelationship
intermittent	ExclusiveRelationship
intermittent	AffinityRelationship
intermittent	IntegerAffinity
intermittent	FloatAffinity
intermittent	GeneralAffinity

Events and each of the flavors of relationships can be chronicled, which makes a special case version of these types with support for storing the full history of that predicate.

References

Adams, Tarn, and Zach Adams. 2006. Slaves to Armok: God of Blood Chapter II: Dwarf Fortress. Bay 12 Games, released.

Borgo, Stefano, Roberta Ferrario, Aldo Gangemi, et al. 2022. "DOLCE: A Descriptive Ontology for Linguistic and Cognitive Engineering." Applied Ontology 17 (1): 45–69. https://doi.org/10.3233/AO-210259.

Evans, Richard. 2009. "AI Challenges in Sims 3." Artificial Intelligence and Interactive Digital Entertainment.

Evans, Richard. 2010. "Introducing Exclusion Logic as a Deontic Logic." In Deontic Logic in Computer Science, David Hutchison, Takeo Kanade, Josef Kittler, et al., vol. 6181, edited by Guido Governatori and Giovanni Sartor. Lecture Notes in Computer Science. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-14183-6_14.

Evans, Richard, and Emily Short. 2014. "Versu—A Simulationist Storytelling System." IEEE Transactions on Computational Intelligence and AI in Games 6 (2): 113–30. https://doi.org/10.1109/TCIAIG.2013.2287297.

Gebser, Martin, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2012. Answer Set Solving in Practice. In Synthesis Lectures on Artificial Intelligence and Machine Learning. https://doi.org/10.2200/S00457ED1V01Y201211AIM019.

Horswill, Ian. 2018. "Postmortem: MKULTRA, An Experimental AI-Based Game." Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 14 (1): 1. https://doi.org/10.1609/aiide.v14i1.13027.

Horswill, Ian, and Samuel Hill. 2024. "Fast, Declarative, Character Simulation Using Bottom-Up Logic Programming." Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment. https://doi.org/10.1609/aiide.v20i1.31866.

Johnson-Bey, Shi, Mark J. Nelson, and Michael Mateas. 2022. "Neighborly: A Sandbox for Simulation-Based Emergent Narrative." 2022 IEEE Conference on Games (CoG), August 21, 425–32. https://doi.org/10.1109/CoG51982.2022.9893631.

Lapeyrade, Sylvain, and Christophe Rey. 2023. "Non-Player Char-Acter Decision-Making With Prolog and Ontologies." Paper presented at IEEE Conference on Games. https://doi.org/DOI: 10.1109/COG57401.2023.10333221.

Lenat, Douglas B., Mayank Prakash, and Mary Shepherd. 1985. "CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlenecks." Articles. AI Magazine 6 (4): 65. https://doi.org/10.1609/aimag.v6i4.510.

Mason, Stacey, Ceri Stagg, and Noah Wardrip-Fruin. 2019. "Lume: A System for Procedural Story Generation." Proceedings of the 14th International Conference on the Foundations of Digital Games, August 26, 1–9. https://doi.org/10.1145/3337722.3337759.

Mateas, Michael, and Andrew Stern. 2003. Façade: An Experiment in Building a Fully-Realized Interactive Drama.

Mateas, Michael, and Andrew Stern. 2005. Façade. Released. https://www.playablstudios.com/facade.

Maxis. 2009. The Sims 3. Maxis, released.

Maxis. 2014. The Sims 4. Maxis, released.

McCoy, Josh, Mike Treanor, Ben Samuel, Aaron A. Reed, Noah Wardrip-Fruin, and Michael Mateas. 2012. "Prom Week." Proceedings of the International Conference on the Foundations of Digital Games (New York, NY, USA), FDG '12, May 29, 235–37. https://doi.org/10.1145/2282338.2282384.

McCoy, Josh, Mike Treanor, Ben Samuel, Brandon Tearse, Michael Mateas, and Noah Wardrip-Fruin. 2010. "Comme Il Faut 2: A Fully Realized Model for Socially-Oriented Gameplay." Proceedings of the Intelligent Narrative Technologies III Workshop, June 18, 1–8. https://doi.org/10.1145/1822309.1822319.

McCoy, Joshua, Mike Treanor, Ben Samuel, Noah Wardrip-Fruin, and Michael Mateas. 2011. "Comme Il Faut: A System for Authoring Playable Social Models." Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 7 (1): 158–63. https://doi.org/10.1609/aiide.v7i1.12454.

Miller, James, and Susann Ragsdale. 2003. The Common Language Infrastructure Annotated Standard. Addison-Wesley Professional.

Nelson, Graham. 2006a. Inform 7. Released.

Nelson, Graham. 2006b. "NATURAL LANGUAGE, SEMANTIC ANALYSIS AND INTERACTIVE FICTION." White Paper.

Otte, J Neil, John Beverley, and Alan Ruttenberg. 2021. Basic Formal Ontology: Case Studies. August 11.

Ramachandran, Deepak, Pace Reagan, and Keith Goolsbey. 2005. First-Orderized ResearchCyc: Expressivity and Efficiency in a Common-Sense Ontology. January 1.

Ryan, James. 2018. "Curating Simulated Storyworlds." University of California Santa Crus.

Samuel, Ben, Aaron A Reed, Paul Maddaloni, Michael Mateas, and Noah Wardrip-Fruin. 2015. “The Ensemble Engine: Next-Generation Social Physics.” Paper presented at Foundations of Digital Games. Proceedings of the Tenth International Conference on the Foundations of Digital Games, June 22.

Samuel, Ben, James Ryan, Adam Summerville, Michael Mateas, and Noah Wardrip-Fruin. 2016. “Bad News: An Experiment in Computationally Assisted Performance.” Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 10045 LNCS (November). https://doi.org/10.1007/978-3-319-48279-8_10.

Smith, Adam M, and Michael Mateas. 2011. “Answer Set Programming for Procedural Content Generation : A Design Space Approach.” IEEE Transactions on Computational Intelligence and AI in Games 3 (3): 3. https://doi.org/10.1109/TCIAIG.2011.2158545.

SomaSim. 2021. City of Gangsters. SomaSim, released.

SomaSim. 2025. Rise of Industry 2. SomaSim, released.

Summerville, Adam, and Ben Samuel. 2021. Kismet: A Small Social Simulation Language. January 1. https://www.academia.edu/101779750/Kismet_A_Small_Social_Simulation_Language.

Sylvester, Tynan. 2018. RimWorld. Ludeon Studios, released October.

Zubek, Robert, Ian Horswill, Ethan Robison, and Matthew Viglione. 2021. “Social Modeling via Logic Programming in City of Gangsters.” Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 17 (1): 220–26. https://doi.org/10.1609/aiide.v17i1.18912.