Social Modeling via Logic Programming in *City of Gangsters*

Robert Zubek[1], Ian Horswill[2], Ethan Robison[3], Matthew Viglione[1]
[1] SomaSim, [2] Northwestern University, [3] Naughty Dog
robert@somasim.com, ian@northwestern.edu, ethan@ethanrobison.com, matt@somasim.com

Abstract

City of Gangsters is a commercial strategy game with significant social modeling mechanics: it is a tycoon management game, where the player needs to work their social connections with a network of roughly 1200 NPCs to get things done, and NPC opinions about the player modulate the player's ability to succeed.

We found logic programming to be well suited to our knowledge representation problem, including the need to perform inferences over a relationship network with more than a thousand active characters, and to provide the player with meaningful feedback about the consequences of their actions in the social space.

In this paper we present the technical details of this social modeling problem, the details of our logic programming implementation, and how this interacts with the game's design and its social and material economies.

Introduction

City of Gangsters (*CoG*) is a commercial "mafia management" game developed by SomaSim and collaborators, and shipping in Q3 2021. Social interaction is foundational to the game – many game mechanics are accessed via conversations with NPCs, and modulated by the NPC's stance towards the player. Moreover, the player's actions have social consequences: news about what the player did spreads through the social networks of interested characters, and may impact how various people interact with the player in the future.

This design necessitated the development of several independent AI components, and in this paper we will discuss two: a logic programming subsystem for social inference, and the application of constraint satisfaction for procedural content generation. These resulted in two success stories, and one lesson in system limitations learned for the future.

About the Game

CoG is set in the 1920s USA during Prohibition, and the player's goal is to try their hand at building and running a successful but illicit alcohol empire that eventually takes over the entire city. Alcohol production and distribution are illegal at that time, and therefore highly profitable, and the player's job is to build and operate a variety of production facilities, set up and manage supply chains, find trustworthy customers, run distribution operations, expand their territory and sphere of influence, and manage a ruthless crew that will get all this done and keep other competing outfits at bay.

Since the core business operation is completely illegal, the game relies on the black-market dynamics of the real world, such as secrecy and relationship management: to get anything done, *"you gotta know a guy"*. In order to buy illegal ingredients or sell off alcoholic beverages, you must have contacts who trust you and who will work with you, because nobody will deal with someone unknown who could rat them out. Those contacts in turn can be built by getting introduced by mutual friends, or by slowly building up rapport over time – and easily destroyed through acts of malice or thoughtlessness towards that person or people they care about. Similarly, the game encodes a few social norms appropriate for the setting, such as revenge (e.g., *"you will pay with an eye for an eye"*), and familial reciprocity (e.g., *"you helped my brother, and now I will help you"*) which are the focus of this paper.

Social and Material Economies

In *CoG* the player operates within a two-level economy. The nominal level is the traditional *material economy* of a tycoon

Figure 1. Screenshot of *City of Gangsters*

game, concerned with the optimization of resource production, supply chains, deliveries, and cashflow.

But this rests on a foundation of a *social economy* of relationships and favors. Buying, smuggling, running speakeasies, and getting anything else done involves talking to NPCs, and the player's action space is shaped by the relationship with that person and any favors they owe. This web of relationships is crucial to manage well, because *most of the important game systems are modulated by the social economy*. Not just business dealings, but also traditional game systems like tech trees, level-ups, quests, loot drops, and other systems, are accessed by talking to people and modulated by relationships, and the player who rushes headlong into combat, and makes enemies, will find themselves quickly cut off from being able to get anything done.

Due to space constraints, we will limit our description of the game to the above, but for more details about the game's design and how it interacted with AI development, please see our parallel design paper (Robison et al. 2021). We also only focus on social modeling here, and omit other AI components used in this game, such as opponent AI that gangs use for decision-making or combat. We welcome the interested reader to experience these directly in the game.

Social Modeling

Before we turn to the core issue of social inference, and the additional issues of constraint satisfaction, let us briefly introduce the social data model that forms the foundation.

The game presents the player with large, procedurally-generated (PCG) cities. In addition to physical city generation, its population is also rendered via PCG to create a large, "lived-in" social context of businesses, rival gangs, troublemakers, and police officers, all interconnected via webs of family ties and friendships, and presenting the player with a variety of opportunities and challenges.

In the typical case, like the *"Chicago"* map, the city will be populated by about 1200 NPCs which can interact with the player and each other (plus a larger number of non-interactive characters which only exist as "window dressing", and which will not be counted here). These NPCs are part of long-standing families that may have been in the city for a couple of generations, and a typical NPC will have a number of siblings, cousins, and other family members in different professions (legal, illegal, organized crime, law enforcement, and so on).

Relationship Model

The graph of relationships linking everybody together is a labeled, directed graph $G = \langle C, R \rangle$ where C is the set of characters (player or NPC), and R is the set of relationship edges. Each $r \in R$ is a tuple of the form $\langle x, y, t, v, H, c \rangle$, where:

- $x, y \in C$, such that r models how x feels about y
- t is a relationship type (e.g., *brother, friend*, etc.)
- $v \in \mathbb{R}$ is a scalar valence value, with positive or negative values for positive or negative opinions
- H is the history affecting the relationship (described below)
- c is our stand-in for extra context data that is used by the game but is not germane to this discussion

Consequently, each relationship edge is dyadic and unidirectional, to represent the basic asymmetry of interpersonal opinions. Edges are only added for characters who know each other (either from existing PCG connections, or because they've become acquainted at runtime), and when two characters meet, each of them adds a corresponding relationship, e.g. when x and y meet, both $x \rightarrow y$ and $y \rightarrow x$ edges are added separately.

History Elements

Each relationship is annotated with a history of events that influence the relationship. Each history H can be considered as a list of history elements $H = \{h_0, \ldots, h_n\}$, which may be empty, and each element is a tuple of the form: $h = \langle \Delta v, a, t, x, e, c \rangle$ where:

- $\Delta v \in \mathbb{R}$ is the change in valence due to this element
- $a \in C$ is the actor and $t \in C$ is the target
- x is an optional expiration time for this element
- e is a set of text strings, for in-character and out-of-character explanations of this action's effect on the relationship valence
- c is additional context (e.g., which quest spawned this history item) which is not material here

Actor and target elements are optional and can be left empty, which creates two slightly different flavors of history elements. The simpler flavor is just a "buff", which comes with just a valence modifier, an optional expiration, and a human-readable explanation. For example, a simple buff might be explained to the user as *"+5 due to trait: friendly"* or *"-15: they just don't like the look of you (for another 3 turns / 15 days)"*. Expiration allows for encoding of both *episodic* and *semantic* knowledge, i.e., some memories will be transient (e.g., being threatened), while some will relate to immutable facts (e.g., this person is family).

The more interesting flavor is an "external action" element. This one extends the "buff" flavor with pointers to the *actor* and *target*. For example, a social history element might include the player as the actor and some third party as the target, and be explained to the player as: *"-10: they heard that you beat up their family member"*. Figure 2 illustrates these kinds of history elements annotating a dyadic relationship.

Relationship Valence

Valence is a scalar which is typically in the range [-50, 50] (corresponding to negative-to-positive feeling), and is calculated from relationship history, i.e., $v = \sum_{h \in H} \Delta v_h$.

This valence is the common (but not sole) driver enabling and disabling various opportunities. For example, as the player travels around the game world, they can chat up store owners or their local beat cop, but if the relationship is merely neutral, they can't expect to get very far. However, as the player works on improving the relationships, NPCs may drop their façade and start letting the player know about other, less public information: business owners may talk about buying or selling illegal alcohols, or the beat cop might indicate willingness to take a bribe. A particularly good relationship with someone starts accruing favors, which can be redeemed for opportunities: warm introductions to useful people, actions that expand the player's territory, leads on ruffians to hire as crew, and many more.

Model Consequences

Each social history element can be considered as tetradic: it's applied to the dyadic relationship, but it can also reference an optional third and fourth party. This is useful for representing a history of the player doing something to someone, which affected the relationship between the player and some other people (or maybe some other unrelated pair of people).

Taken to its extreme, having a list of tetradic annotations on each relationship means that each relationship, nominally dyadic, can be essentially *n-ary* – it is influenced by a history of interactions between an arbitrary number of agents. Furthermore, because of expiration timestamps, the arity and valence of the relationship change fluidly over time. It's a powerful construct on which we lean heavily.

This enables the desirable *"you gotta know a guy"* kind of gameplay. In order to keep the action space open, the player needs to continually build new relationships, and try to affect existing relationships by doing things that those NPCs might appreciate: for example, if the player has only one supplier of a key ingredient, they may wish to keep them happy such as by agreeing to their quests or by hiring their family as crew members. And because of different expiration horizons, the total relationship valence will naturally ebb over time if left unattended.

Consequently, one of the major strategies in the game is *working the social network* – that is, building up relationships with a range of people, finding opportunities to make them happy, pushing them to gain favors, and then redeeming those favors for additional introductions to other useful people, which will reveal a breadth of new opportunities and open many new doors for the player.

Social Inference

We can now return to effects such as *"you will pay with an eye for an eye"*, or *"you helped my brother"*. These are the kinds of indirect effects where the player's (or anyone's) actions on one person can affect their standing with another person, perhaps someone who they didn't even know existed, by inserting relevant history elements.

Here is one example of this effect, observed in a recent play-through. The human player started extorting businesses in his territory, as gangsters do, but as he drove around to a different part of town a little bit later, suddenly he got ambushed and attacked by a previously-unknown corner hooligan. A quick conversation showed that the hooligan was the brother of one of the people who was extorted, and he didn't take it lightly.

Under the hood, the news of the player's extortion propagated to the target's family members, and that was enough for the hooligan brother to build a one-sided negative relationship towards the player, before the player ever had a chance to meet him.

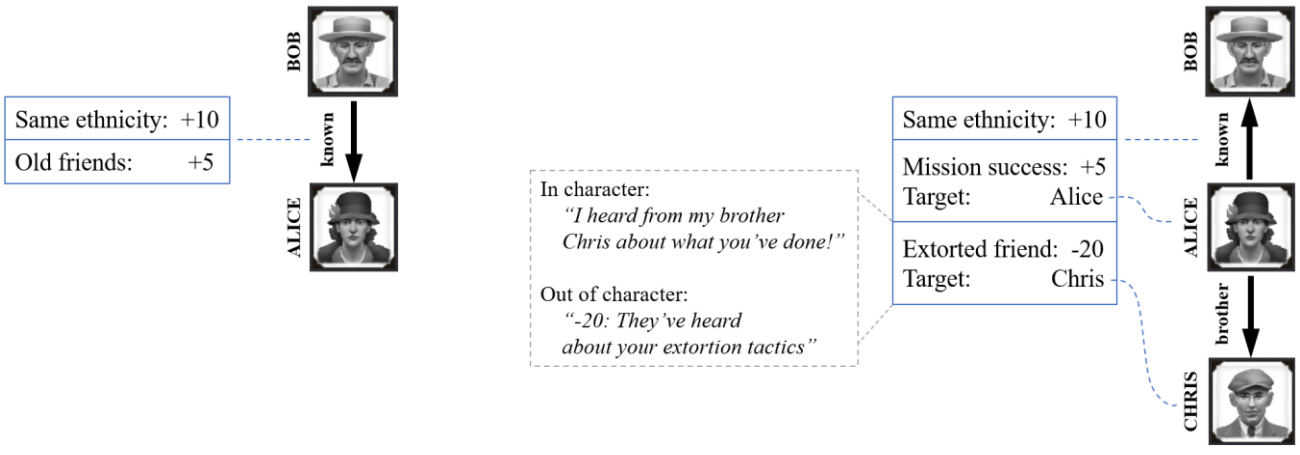

Figure 2. Relationship between three people, with history of buffs and social actions. Left: simple buffs. Right: social action between Bob and Chris affecting relationship between Alice and Bob, with player-facing copy.

These kinds of effects are implemented by having the system observe activities in the world, and then change relationships between the actor and other relevant NPCs by stacking on new social history elements on those relationships. This is done in three steps: querying the relationship knowledge graph to see if some specific patterns have been detected, figuring out how to respond, and finally modifying the relevant relationship and possibly informing the player.

We wanted to implement querying and response as generic rules, so that we could apply them broadly across various situations.

Logic Programming and Social Inference

Logic programming languages allow algorithms and data to be expressed in a logic-like formalism. In its simplest form, the author provides basic facts and rules encoded as first-order Horn clauses. The equivalent of subroutines are predicates, and calls to predicates are treated as queries to find values of the variables in the query that will make the query true. Queries can also be used as iterators to find all possible solutions to the query.

Logic programs can be interfaced to the game's native data structures by writing a small number of primitive predicates in the host language (C#) to perform low-level queries of the game state. High-level queries and inference rules can then be written against these primitives. The system executes them automatically, without the need to write bespoke loops. If the organization of the low-level data is changed, the primitives can be updated, without having to change the high-level code.

Social Norm Encoding

We encoded our social norms as logic rules, which scan the relationship graph for specific link structure that spans any kinds of persons. For example, a rule like *"eye for an eye"* could be cast as the rule: X should retaliate against Y if Y committed a violent action A against some member F of X's family. The game can then periodically query: find all X,Y such that X should retaliate against Y.

Queries of rules like this are performed routinely in response to character actions. Most fail, but when matches are found, the main game takes the appropriate action. In this case, the result would be to add a new element to the relationship between persons X and Y with a drastically negative valence.

BotL

Queries and inference rules are written in BotL, a logic programming language designed by Ian Horswill for real-time execution inside game engines. BotL was originally designed with two goals in mind: to be a more accessible Prolog for an undergraduate class on social simulation, and to be a proof of concept that logic programming could be sufficiently performant for a real-time game with hundreds or thousands of NPCs. From early on in *CoG* prototyping, we had been considering using logic programming, but had found existing implementations inappropriate for performance or interoperability reasons. BotL makes a number of departures from traditional LP systems to allow this.

Unified object system. BotL and the underlying game engine share a single object system and memory manager, namely the Common Language Runtime (Miller & Ragsdale, 2004) on which the Unity3D (Unity Technologies, 2004) game engine is built. LP languages traditionally use bespoke object representations and memory managers, meaning there are then two heaps, two garbage collectors, many stacks, and two radically different representations of objects. When interoperating with C or C# code, each codebase must understand both implementations. All other changes listed below are either forced by or enabled by this design constraint.

Compound objects are opaque to the unifier. Forcing the logic programming system to use the native object system makes standard Prolog-style data structures impossible. In particular, (1) logic variables cannot be embedded in compound objects such as arrays and record types, and (2) the unifier cannot look inside compound objects for the pur-

```
// inference about violence goes along family links of any valence
socialInference(Interlocutor, Target, Human, $SocialLink.Family, "violence", "violence-inf") <--
  linkedVia(Interlocutor, Target, $SocialLink.Family),
  historyIncludes(Target, Human, "violence");

// respect from helping someone else (i.e. finishing a quest) follows all links
socialInference(Interlocutor, Target, Human, AnyLink, "quest-complete", "quest-complete-inf") <--
  linkedVia(Interlocutor, Target, AnyLink),
  historyIncludes(Target, Human, "quest-complete");

// X and Y have EventType in their history together
historyIncludes(X, Y, EventType) <-- historyExists(X, Y, History), includes(History, EventType);
```

Figure 3. Two examples of relationship-specific queries in BotL (edited for clarity).

pose of pattern matching. The language thus bears superficial similarities to DATALOG (Greco, 2015). Since previous social simulation systems (Samuel, *et al.* 2015; McCoy, *et al.* 2012; McCoy, *et al.* 2011; Evans, Short 2014; Evans 2009) didn't allow this either, it was not a problem for our use cases.

Higher-order predicates are macros. The lack of support for traditional Prolog data structures means BotL is not fully homoiconic (Kay, 1969). This means that higher-order predicates, which take abstract syntax trees of source code and execute them at run-time, cannot be implemented. BotL mitigates this limitation by allowing higher-order predicates as compile-time macros that perform source-to-source transformation into equivalent first-order code. These are used to implement: logical connectives (and/or/not); so-called "all-solutions" predicates that, given a query to perform, generate a collection of all unique solutions to the query; aggregation predicates that fold a function over all solutions to a query; and optimization predicates that find the minimal or maximal solution to some query given an objective function.

Support for functional expression. Since BotL does not support the use of function terms as record constructors ("structures" in Prolog), functional expressions are free to be used to represent traditional function calls. In Prolog, the call `p(f(1))` would call predicate `p` with a record structure of type `f` containing the single slot `1`. In BotL, it means to call the function `f` with `1` as an argument and pass its return value as an argument to the predicate `p`. There is also support for calling user-defined predicates intended to denote functions in functional style: if `f(I,O)` is a two-argument predicate intended to represent a function from `I` to `O`, the expression `p(f(1))` will be converted by the compiler into the conjunction: `f(1, Temp), p(Temp).`

Static memory allocation. The primary performance constraints on the BotL implementation were that (1) it run in a small memory footprint, so it could be deployed to consoles and mobile devices, and (2) that it never trigger a garbage collection, since GC causes dropped frames, which in turn cause complaints from reviewers and users. On the one hand, logic programming languages are a good match for this application because they fully obey stack discipline: when the system backtracks, all memory allocated since the choice point being backtracked can be freed as a block. On the other hand, modern host language VMs such as the CLR and the JVM severely limit what kinds of data can be stack allocated. As a result, the BotL VM maintains its own stacks for all its internal data structures. These stacks are statically allocated, meaning that once the VM itself is instantiated, further calls into BotL code effectively run in constant space.

Hybrid logic/functional VM. BotL runs in an exotic byte-coded virtual machine. It is a variation of the Vienna Abstract Machine (Krall & Neumerkel, 1990), specifically VAM_{2P}, augmented with a separate instruction set for evaluating functional expressions that appears as arguments in predicate calls.

Calling sequences for logic programming involve the caller pushing arguments on the call stack and the callee then rescanning those arguments to determine if they match the argument pattern of the left-hand side of rule from which they are compiled. If so, the callee's code continues on to call the predicates making up the right-hand side of the rule. If not, the system tries the next rule for the predicate being called. Compiled rules therefore have the structure of a series of match opcodes followed by a series of calling opcodes.

Following the VAM_{2P}, BotL runs these two sections in lock step: it runs the push opcodes of the caller synchronously with the corresponding match opcode of the callee; the VM maintains *two program counters*: one for the caller and one for the callee, and the inner loop of the VM dispatches not on a single opcode but on the *(caller-opcode, callee-opcode) pair*. This allows the system to determine at run-time that a given argument need not be computed because it will be matched with a wildcard on the other side. It also allows for early outs from pattern matching: if the first argument does not match, the subsequent arguments are never even pushed on the stack.

Benefits of Logic Programming

Logic programming allowed us to formulate our social norms as *queries* on a city-wide knowledge graph, using a logical form decoupled from the game internals. This had several positive effects:

- Hiding execution details: letting us focus on the specific patterns we wanted to detect, instead of developing code for traversing data structures
- Iteration robustness: queries written in a domain-specific language (DSL) were easy to modify and extend as design changed
- Code separation: changing game internals required modifying only the primitive predicates, not the inference rules or queries

Additionally, the developer-cost profile of this setup was very encouraging: even though there was a one-time cost to integrating BotL into the game, once that was done, adding or changing inferences in the existing setup had negligible marginal costs.

Constraint Satisfaction

Another logic-based AI system employed in *CoG* was the constraint satisfaction system *CatSAT* (Horswill, 2018), which is a highly-performant randomized solver for an ASP-like language.

Procedural content generation is a natural application for constraint solvers, as PCG content typically needs to satisfy

a variety of simultaneous positive or negative constraints. In our case, we used the solver in two specific cases: generating personality traits, and generating family members over time.

Personality Trait Generation

CoG uses PCG to generate physical cities, and also to populate them with entire families of interconnected individuals. We needed these individuals to have personality traits, following some specific desiderata:

- Each individual gets $n \approx 3$ unique traits from a bag of $m \approx 20$ total traits
- Some traits are incompatible with others and must not be picked at the same time
- Some traits entail others and must be picked together
- Trait inheritance: some random traits from parents will be selected and must be included in the solution

This problem forms a natural fit for constraint satisfaction, and was successfully used to generate all traits for all characters, giving them coherent personalities as well as long-term patterns that are shared among family members.

Population Generation

In addition to trait generation for individuals, we experimented with using the solver to generate entire populations of individuals, by generating families from random individuals using constraints such as age (children need to be sufficiently younger from their parents and separated from each other), setting-appropriate marriages (they needed to have similar age, similar ethnicity, and opposite gender), and so on.

Interestingly, this approach was not successful. The computational problem of building 100+ family trees containing 1200+ people, connected via intermarriages that simultaneously satisfied the various constraints, was too large for the solver to run within acceptable performance bounds. In the end, we took this as a lesson about what kinds of problems are the best fit for solvers – and instead re-implemented family generation as forward-simulation over a period of several generations.

Related Work

The last decade has seen a growth of interest in social simulation in game AI. Recent systems can be divided roughly into reputation systems (Sellers, 2008), large-scale simulations with hundreds of NPCs (Adams & Adams, 2006; Ryan, 2018), which use forward-simulation, and smaller-scale systems with 5-20 NPCs, which use symbolic reasoning (McCoy, *et al.* 2012; McCoy, *et al.* 2011; Samuel, *et al.* 2015; Evans, Short 2014). *The Sims 3* (Evans, 2009) is an interesting transitional case, as it used both forward simulation and a simple rule-based system, although detailed social simulation was only performed on the characters in the current "lot." *CoG* supports significantly more characters than any of these systems and demonstrates that symbolic reasoning can be scaled to much larger groups. Of these systems, *CoG* is most similar to *Versu* (Evans & Short, 2014) and *CiF*/*Ensemble* (Samuel, *et al.* 2015).

Versu is implemented in what can generally be considered a logic programming language, *Praxis* (Evans & Short, 2014). *Versu* also heavily emphasizes characters' evaluations of one anothers' actions. It differs from *CoG* in the logic on which it is based (eremic logic (Evans, 2010) rather than FOL), its smaller scale, and its genre (interactive fiction rather than a tycoon game).

Like *CoG*, *CiF* (McCoy, *et al.* 2011), and its successor, *Ensemble* (Samuel, *et al.* 2015), represent social state in terms of a weighted graph of relationships and a history of social actions, then query that using symbolic rules. They differ from *CoG* in that they implement a much more complex social simulation (hundreds of rules rather than tens) for a much smaller group of characters (tens rather than a thousand). And as befits a puzzle game, *Prom Week* focuses less on explaining the causes of NPC responses than does *CoG*.

Finally, the arguments made here for logic programming are very similar to the arguments made in entity-component systems (Bilas, 2002) for placing all game objects in a relational database: the use of a general query language allows developers to quickly react to design changes. However, the architecture used here shows that such query languages can be supported without having to literally store all game state in a database.

Conclusions

The resulting system exceeded our expectations. It allowed for a city with a very rich, "lived-in" feel, where everything gets entangled in the webs of relationships and interpersonal histories. For the player, these effects are impossible to escape, and present interesting challenges and opportunities.

As for the technical approach, LP proved to be a very effective solution. BotL was robust to unexpected design changes, and allowed for easy extensions. In fact, the use of LP opened up additional design options – once we had a system that worked well and was easy to extend, we decided to double down on these social mechanics and use them more widely than we initially planned, with excellent outcomes.

However, the use of a LP system is not without its challenges. Perhaps the main challenge is that both the initial integration and the continuing use require someone with appropriate expertise in AI and LP specifically. This can be a great opportunity for collaboration with academia, but otherwise might present a staffing difficulty. Secondly, the standard challenge of embedded DSLs applies: they are exogenous to popular game engines, and their lack of debugging and tracing tooling provides integration challenges.

References

Adams, T., & Adams, Z. 2006. Slaves to Armok: God of Blood Chapter II: Dwarf Fortress. Bay 12 Games.

Bilas, S. 2002. A Data-Driven Game Object System. In *Game Developer's Conference (GDC 2002)*. San Francisco.

Evans, R. 2009. AI Challenges in Sims 3. In *Artificial Intelligence and Interactive Digital Entertainment*. Stanford, CA: AAAI Press.

Evans, R. 2010. Introducing Exclusion Logic as a Deontic Logic. In *Deontic Logic in Computer Science, Proceedings of the 10th International Conference, DEON 2010, Lecture Notes in Computer Science Volume 6181* (pp. 179–195). Fiesole, Italy: Springer.

Evans, R., & Short, E. 2014. Versu - A Simulationist Storytelling System. *IEEE Transactions on Computational Intelligence and AI in Games*, *6*(2), 113–130.

Greco, S. 2015. *Datalog and Logic Databases*. Morgan & Claypool.

Horswill, I. 2018. CatSAT: A Practical, Embedded, SAT Language for Runtime PCG. In *AIIDE-18*. AAAI Press.

Kay, A. 1969. *The Reactive Engine*. University of Utah.

Krall, A., & Neumerkel, U. 1990. The Vienna Abstract Machine. In *International Workshop on Programming Language Implementation and Logic Programming. PLIP 1990. Lecture Notes in Computer Science, vol. 456.* Berlin, Heidelberg: Springer.

McCoy, Josh, Treanor, M., Samuel, B., & Reed, A. A. 2012. Prom Week. Santa Cruz, California: Expressive Inteligence Studio at UC Santa Cruz.

McCoy, Joshua, Treanor, M., Samuel, B., Wardrip-Fruin, N., & Mateas, M. 2011. Comme il Faut: A System for Authoring Playable Social Models. In V. Bulitko & M. O. Riedl (Eds.), *Proceedings of the 7th AI and Interactive Digital Entertainment*. Stanford, CA: AAAI Press.

Miller, J. S., & Ragsdale, S. 2004. *The Common Language Infrastructure Annotated Standard*. Addison-Wesley.

Robison, E., Viglione, M., Zubek. R., Horswill, I. 2021. AI Design Lessons for Social Modeling at Scale. Forthcoming.

Ryan, J. 2018. *Curating Simulated Storyworlds*. University of California, Santa Cruz.

Samuel, B., Reed, A. A., Maddaloni, P., Mateas, M., & Wardrip-Fruin, N. 2015. The Ensemble Engine: Next-Generation Social Physics. In *Proceedings of the 10th International Conference on the Foundations of Digital Games (FDG 2015)* (pp. 22–25).

Sellers, M. 2008. Otello: A next-generation reputation system for humans and npcs. In *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008* (pp. 149–154).

Unity Technologies. 2004. Unity 3D. San Francisco, CA.