

The AI Architecture of Versu

Richard Prideaux Evans and Emily Short

Imperial College, London

1 A Simulationist Interactive Drama

Versu is an improvisational play, rather than an interactive story. The player is encouraged to *perform* her character, to improvise within the dramatic situation that she has been thrown into. The smallest comment, the slightest look, even *not* saying something - these moment-to-moment actions are noticed by the other participants and amplified. In this respect, Versu resembles *Façade*. But while *Façade* uses an architecture built around beats and joint-behaviours, Versu is an agent-driven simulation in which each of the NPCs make decisions independently.

When building an interactive narrative system, one of the fundamental design decisions is whether the individual agents or a centralised drama manager gets to decide what happens [21]. At one end of the spectrum, a *strong story system* is one in which the drama manager makes all the decisions. The NPCs have no individual autonomy - they are just puppets of the drama manager. At the other end of the spectrum is the *strong autonomy system*, in which the NPCs make decisions based on their own individual preferences, unaware of authorial narrative goals. *Façade* occupies the middle ground on this axis: a centralised drama manager chooses the next beat, but the individual agents and joint behaviours have some limited control over how the beat is played out.

Versu, by contrast, takes the strong autonomy approach. Each character chooses his next action based on his own individual beliefs and desires. There is a centralised drama manager, but it is rare indeed for the drama manager to override the characters' autonomy and force them do something. Instead, the drama manager typically operates at a higher level - by providing suggestions, or tweaking the desires of the participants.

Why did we choose a strong autonomy approach in Versu? There were two main reasons. Firstly, a true simulation provides much more opportunity for *replayability*. In *Façade*, many of the behaviours are hard-coded to the particular characters who are involved in them. In *Façade* there is no general making-cocktails activity. Instead, there is the *particular* activity of Trip-making-cocktails. Because the scripts and characters are entangled together, it would be hugely nontrivial to replace a character in *Façade* with another (replacing Trip with Captain Kirk, say). It would involve rewriting most of the behaviours.

In Versu, the social practices are authored to be *agnostic* about which characters are assigned to which roles. This means that we can assign various different characters to the roles, and *everything just works*. In a Versu-authored version of the *Façade* situation, you could play Grace, or Trip, or the guest. You could assign different characters to the various roles, and see what happens.

The simulation makes a clear distinction between *roles* in a story and the *characters* playing those roles. A romance might have a hero, a heroine, a friend, and a jealous rival. The player is free to assign various characters to those roles. She could make Mr Darcy be the hero, and Elizabeth Bennett be the heroine - or, she could make Mr Collins be the hero, and Miss Bates be the heroine. The episode will play out very differently depending on which characters are playing which roles. If there are n roles and $k(> n)$ characters to play those roles, we have $k!/(k-n)!$ permutations.

Relatedly, the scripts in *Façade* assume that the player is always playing the guest, and the hosts are always played by NPCs. If you wanted to rework it so that you could play Trip or Grace, this would involve a major rewrite. In *Versu*, because the social practices are authored to be *character-agnostic*, you can play the same story from multiple perspectives. You can try out the job interview out from the perspective of the interviewee, or the interviewer. You can even play it as the drama manager. An episode with 2 roles could be played by two humans in multiplayer, one human playing either role, or both roles played by the AI. More generally, an episode with n roles has 2^n permutations of player-npc assignments. The first reason, then, that we committed to a simulationist architecture is because we wanted to maximise replayability.

The second reason for choosing strong autonomy is that a simulationist architecture allows the player more control over the outcome. A simulated system has clear rules which the player can learn and internalize. Once she has confidence that she has understood the internal mechanisms, she can use these to *anticipate* the consequences of future action, and plan how to achieve her goals. A non-simulated system risks being just a series of arbitrary puzzles, in which the player is forced to guess the changing whims of the designer. A simulation, by contrast, uses the same models repeatedly. The player can build up confidence that she *understands* the underlying system - and increased understanding can yield increased control.

2 Addressing the Design Questions Raised by *Façade*

Some of the crucial initial design decisions in *Versu* were made by looking hard at what *Façade* achieved. We focused on *Façade* in particular because it is such a substantial achievement. In [8], when evaluating the successes and failures of *Façade*, Mateas and Stern mention three outstanding issues in particular:

- **The speed of content production** and global agency. Because of the intricate animation overlaying and parallel behaviours which needed to be authored for most actions, adding a new piece of content to *Façade* was a time-consuming task. In the end, after 3+ years in development, they only had time to author 27 beats. The amount of global agency (the ability for the player to affect the overall arc of the story) is limited by the amount of content, so in the end the player did not have as much ability to affect the outcome of the story as the authors had initially hoped.

- **Feedback.** Façade involved three social “head-games”, played one after the other (an affinity game, a hot-button game, and a therapy game). By design, the state of each game was not communicated directly (via numbers of spreadsheets or sliders), but indirectly by gesture and tone of voice. (The authors wanted to maintain the sense that this was a drama rather than a computer game). But this design decision made it very hard for the player to tell the state of the simulation.
- **Interface.** In Façade, the player can type any text she wishes. But the parser will attempt to shoe-horn all the player’s sentences into one of 30 parametrized discourse acts. Unfortunately, the player’s utterance can not always be fitted into one of these 30 specific actions - and even if it could, the parser often can not see how it could. The player can feel like she is fighting the parser, rather than using it effortlessly as a tool to communicate with.

We tried to make sure that Versu had good answers to the three issues which Mateas and Stern identified:

- The **speed of content production.** To speed up content production, we eschewed Façade’s 3D procedurally animated characters for (procedurally-generated) text and static images.
- **Feedback.** We added various visualisations to the UI (User Interface) so the player can see at a glance the state of the simulation.
- **Interface.** We replaced the parser with a simpler menu interface. The affordances provided by each social practice were displayed explicitly to the player. She acts by clicking on a button.

We shall go through each of these three decisions in turn. But first, a point of clarification: it may look like each of these responses to issues in Façade are just simplifications that allowed us to *avoid* the hard problems that Mateas and Stern were brave enough to tackle head on. But these decisions were not just a cop-out - they were pursued in order to allow us to be *more ambitious in our simulational goals*. We will return to this point repeatedly below.

2.1 Text Output

Façade wholeheartedly embraced the “holodeck” vision of interactive drama in which character behaviour is rendered realistically in multiple modalities: 3D characters, parallel animation, recorded speech. This, of course, is one of the main reasons why it took so long to add a new behaviour: it is time-consuming getting the animations blending correctly and achieving synchronization with other actors.

Now there is nothing wrong with realistic rendering of characters, but neither is it *necessary*. We agree with Salen and Zimmerman [26] that the level of immersion does not necessarily increase with extra levels of realism. A text output can be just as immersive as a 3D animated environment. To think otherwise is to be seduced by what Salen and Zimmerman call the “*immersive fallacy*”. Versu does not use fancy 3D animation or voice actors - the output is dynamically

generated text and static images. Text output certainly makes it quicker to produce behaviour (in a later section, we shall describe how we managed to produce an order of magnitude more behaviours than *Façade* in a shorter timeframe). But that is not its only advantage. It is not just that text is cheaper than 3D animation - it is also more *expressive*.

Text as an Expressive Medium: Interiority Before we started developing *Versu*, when working on *The Sims 3*, we came across a revealing situation which highlighted the advantages of text output for revealing interiority. There was a chronically shy Sim who was hosting a party. Some of the guests had rung the door, and were waiting to be ushered in. The shy Sim was sitting on the couch, deciding what to do. Debugging his internal state, we could see that he was *conflicted*: the norms of social propriety dictated that you should answer the door when invited guests come over. But his own chronic shyness gave him a strong countervailing reason for *not* answering the door: he very much wanted to be alone. Within the decision-making system that we were using, the Sim had a hard choice between answering the door and refusing to do so. But neither of these options captured what the Sim wanted to express: what should have happened is that the Sim answered the door *reluctantly*. Here, we express his internal conflict through an *adverbial modifier*.

Now in a 3D game with animated polygonal characters, adding an adverbial modifier to an action is a hugely expensive process: we would need a separate animation for answering-the-door-reluctantly - and we would also need a separate walk cycle for reluctant walking. Given a large set of animated actions, each adverbial modifier we add would require a prohibitively large number of additional animations. But in a *text* game, adverbial modifiers are much cheaper: we can modify a verb by simply *appending* an adverb to the sentence. Adverbial modifiers are useful in many ways: we can use them to express internal state, to express the reason for action, and also to express individual personality.

Text as an Expressive Medium: Individuality We wanted each character in *Versu* to have a unique personality and we wanted their individuality to be expressed *throughout* their actions. Text output made it feasible for each character to have a unique text override for many actions. For example, in most 3D games, each character uses the same generic walk cycle. But in *Versu*, each character has an individual way of walking: Brown *swaggers*, Frank Quinn *walks ponderously*, George Wickham *strides*, Lady Catherine *hobbles*, while the pug dog *waddles*.

2.2 Feedback

When designing ways to help the player understand the social simulation, we were guided by Noah Wardrip-Fruin's concept of the *Sim City effect* [34]. When playing a game which simulates some aspect of experience that the player is already familiar with, the player starts by using her own model of how it works.

But the simulation will inevitably diverge from reality in various ways. If things go badly, the divergence between the player's understanding of the phenomenon and the simulation's model of the phenomenon will prevent the player from understanding or manipulating the system. The *Sim City effect* occurs when the user-interface helps the player to transition from her original model of how the thing actually works to how the simulation models it. If this works properly, the player ends up with an accurate model of how the simulation models the phenomenon, without having had to read a manual or textbook.

Our simulation is based on fine-grained emotional states, relationships, and social practices. We made sure that the user interface exposed these to the player transparently¹.



Fig. 1. Each character's emotional state is displayed along with explanatory text.

To help the player understand the characters' moods and relationships, we added a portrait of each character at the bottom of the screen. (See Fig. 2). Each character has various emotional states he can be in (based on Ekman's typology[4]), and each character has a different portrait for each emotional state.

¹ A danger with exposing the simulation internals is that the experience starts to seem less like a drama and more like a computer program. We worked hard to make sure that the emotions, relationship states and social practices were exposed in a way that kept the player immersed within the world we were creating.

When the player clicks on a character portrait, she sees *why* that character is in that particular mood: each character remembers who the emotion is directed towards (e.g. I am annoyed with Brown), and the event which prompted the emotional change (e.g. Brown's insult).

Our simulation is unusual in that there are multiple independent social practices running concurrently (this is described in detail below). To help the player understand the state of the various social practices that are currently in play, we organised the affordances around the practices which initiated them. For example, if the player's character is in the middle of a dinner-party, and Brown has just made a rude remark, there will be two social practices running concurrently: the dinner party (providing affordances to *Eat, Drink, etc*) and the current conversation (providing affordances to *Disapprove of Brown, Forgive him, etc*). The affordances are arranged in categories, grouped by the social practice that instantiated them, so that the player begins to understand the underlying simulation state. The text for each social practice is carefully worded to display its current state.

3 Architecture Overview

Our simulation is built up out of two types of objects: agents and social practices. (See Fig. 3).

A **social practice** describes a type of recurring social situation. Some social practices (e.g. a conversation, a meal, a game) only exist for a short time, while others (e.g. a family, the moral community) can last much longer.

A practice coordinates agents via the *roles* they are playing. For example, a greeting practice sees the two participants under the descriptions of *greeter* and *recipient*.

The main function of the social practice is to describe the actions the agents can do in that situation. A greeting practice, for example, tells the greeter how he can greet the recipient. It also tells the recipient the various ways she can respond.

The practice provides the agent with a set of *suggested actions*, but it is up to the agent himself to decide which action to perform, using utility-based reactive action selection. This is described in Section IX below.

In Versu, we allow *multiple practices to exist concurrently*. During a dinner party, for example, there will be multiple practices operating at once:

- eating and drinking (and commenting on the meal)
- the conversation about politics
- the rising flirtation between Frank and Lucy
- responding to the fact that Mr Quinn has spilled the soup

Each of these practices provides multiple affordances. The agent’s set of options is the *union* of the affordances from each of the practices he is participating in.

Some practices are organised into states, so that they can provide different affordances in different situations. But a social practice is significantly more powerful than a Finite-State Machine, in two main ways. First, each practice can store arbitrary persistent data², while the only memory a state machine has is the state it is in³. Second, the only possible effect of a Finite-State Machine’s action is transitioning from one state to another. A Versu action can do much more than change the state of the practice: performing an action can result in *any sentence* being added to the world database. The results of adding new sentences can be that relationships are updated, new beliefs or desires are formed, old practices are deleted or new practices are spawned.

² The whist game, for example, stores which cards have been assigned to which players, which suit is trumps, whose turn it is to play, and the score. See [10] for an early description of how practices need their own memory to “keep score”.

³ Because the Finite-State Machine’s memory is limited by the number of states, it is not Turing-complete.

4 The Architecture

In this diagram, boxes represent data and ovals represent procedures which operate on that data:

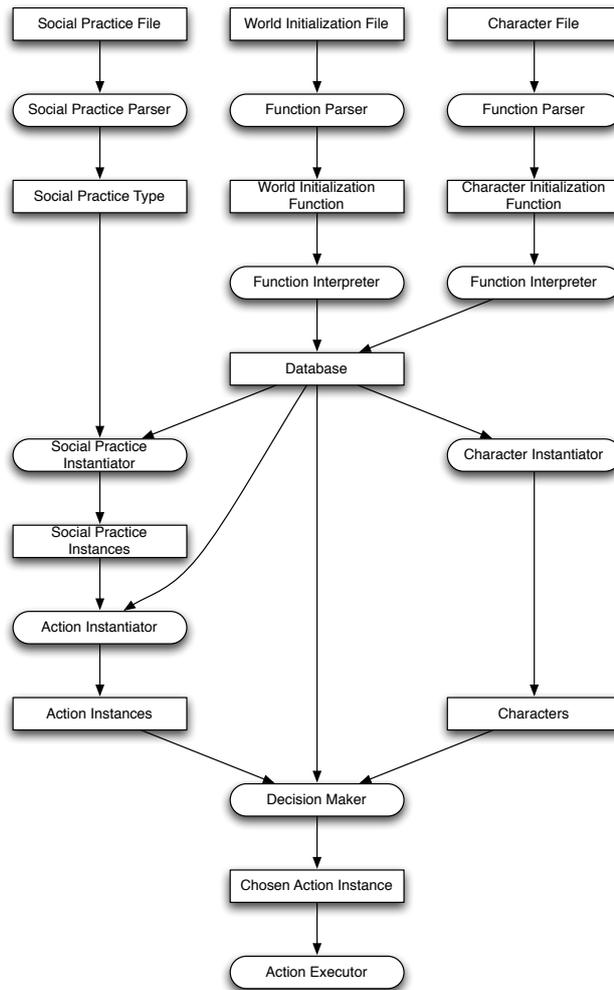


Fig. 2. The architecture of the simulator. Data is placed in rectangles, and processes in ovals. This diagram shows the information flow when an NPC makes a decision. The same architecture is used for player choice - except the Action Instances are sent directly to the user-interface, rather than to the Decision Maker.

We shall start from the top of the diagram and work downwards. Creating an episode involves three types of scripts:

- Scripts defining the social practices that can be instantiated during the episode
- Scripts defining the initial state of the characters that may be participating in the episode
- A script defining the initial world state

All scripts are authored in a high-level domain-specific-language designed specifically for this simulation: **Praxis**⁴.

When the episode starts, we execute the World Initialisation function, and then execute the Character Initialisation function for each character who was selected to participate in the episode. At this point, the Database contains the initial world state.

Social practices can be parameterized by arguments, so they can be multiply instantiated with different values for the arguments. Each practice type specifies a set of actions together with their preconditions. The actions available to a character at any given moment are determined by all the actions of the currently instantiated practices whose preconditions are satisfied.

Agents score each available action (see section IX) and then execute the highest scoring action. Actions may modify the database and/or generate text.

The diagram is almost symmetric, with social practices on the left hand side, and characters on the right hand side. The reason it is not *entirely* symmetric is that there are multiple social practice instances for each social practice type (e.g. there may be two instances of the whist game practice type occurring simultaneously in two different rooms), but there is only *one* character instance for each character file.

The diagram does not call out the drama manager explicitly. This is because each episode’s drama manager is implemented as a special type of social-practice. A drama manager is not a new type of entity: it is just a particular type of practice⁵.

5 How We Represent the World

5.1 The world is everything that is the case

In the Praxis system, the simulation-state is entirely determined by a set of sentences in a modal logic⁶. The set of sentences is the complete simulation

⁴ Praxis contains a number of decisions which are logically independent. First, the decision to model social practices as first-class objects. Second, the use of a strongly-typed logic-formalism to model simulational state. Third, the use of Exclusion Logic as the logic of choice.

⁵ The drama manager is described in Section XI.

⁶ A modal logic, in the broad modern sense, is a logic which contains non truth-functional operators for talking about relational structures (see [1], p. xi). Previously, the term “modal logic” was restricted to logics which treat the standard modalities of necessity, possibility, knowledge, belief, time, deontic modalities, etc.

state. There are no objects, or pointers, as traditionally conceived. Representing the world-state as a set of sentences has a number of advantages:

- **Visibility.** The entire world-state is completely open to inspection. Nothing is hidden. If you want to know, for example, if there is anyone in the current simulation state who is in the same room as someone they dislike, you just need to form the query. There is no need to ever write code to access the state of the world because the entire state of the world is represented in a uniform manner and is *already* open to view.
- **Debuggability.** You can place *logical* breakpoints to detect which practice is responsible for making a fact true. This is much more powerful than traditional code breakpoints or data breakpoints.
- **Serializability.** Because the world-state is represented in a uniform manner, it is trivial to serialize and deserialize world-state

The main advantage is visibility. We found, in previous simulations we have worked on, that the main factor which makes it hard to improve the quality of the AI is the difficulty in seeing the entire simulation state. Bugs lurk in the darkness. In this architecture, where the world is a set of sentences, nothing is hidden.

The simulator comes with a run-time Inspector, which gives developers complete access to the internal state of the simulation. The Inspector allows them to:

- find out what is true
- print all facts about an object, agent or process
- find all instantiations of a term with free-variables (e.g. find me everybody who Brown has annoyed)
- find out why an action’s preconditions have failed
- find out what is causing a fact to become true

5.2 Sentences, Practices, Agents and Affordances

The world is a set of sentences in a formal logic. Sentences which contain the distinguished **process** keyword make practices active. Sentences containing the distinguished **agent** keyword makes agents active. These practices provide affordances to the agents. When an agent chooses to perform one of these affordances, the world-state is changed: sentences are added and removed from the database. The database changes can change the set of available practices, and the loop begins again. In summary:

- Sentences being true make practices and agents active
- Practices propose affordances to agents
- Performing affordances updates sentences in the database (which might mean that different practices are now available, providing different affordances)

6 Exclusion Logic

The world-state is defined as a set of sentences in a logic called *Exclusion Logic* [5]. This modal logic is particularly well-suited for modelling simulation state in general, and social practices in particular.

Given a set S of symbols, the literals⁷ in Exclusion Logic are defined as all expressions of type X in:

$$X ::= S \mid S.X \mid S!X$$

We use the $!$ and $.$ operators to build up trees of expressions. For example:

- brown.sex!male : Brown is male
- brown.class!upper : Brown is upper class
- process.dinner.dining_room.participant.brown: Brown is one of the people having dinner in the dining room
- process.dinner.dining_room.participant.lucy : Lucy is one of the people having dinner in the dining room

Asserting that $A.B$ is claiming both that A , and that *one* of the ways in which A is B . Saying that $A!B$, by contrast, is to say that B is the *only* way in which A is the case. The semantics of Exclusion Logic is described in [5].

The two things that distinguish Exclusion Logic from traditional predicate logic are:

- The ability to directly represent tree structures
- The exclusion operator

6.1 Exclusion Logic Literals Represent Tree Structures

Consider the following facts about Brown:

- brown.sex!male
- brown.class!upper
- brown.in!dining_room
- brown.relationship.lucy.evaluation.attractive!40
- brown.relationship.lucy.evaluation.humour!20

This is a declarative representation of a tree structure, implemented as a trie[9]. A group of shared literals has a shared prefix (in this case, “brown”), and the sub-tree can be referred to directly by its prefix. The sub-tree can be removed

⁷ The database (representing the current world state) is just a collection of ground literals. But the *queries* that can be expressed in the Praxis language are all propositions of type P , using the familiar connectives:

$$P ::= X \mid P \wedge P \mid P \vee P \mid \neg P \mid P \rightarrow P \mid (\forall v)P \mid (\exists v)P$$

in one fell swoop by deleting its associated prefix. We can e.g. remove all of the facts about Brown by deleting the term “brown”⁸. A prefix (referring to a subtree) is the Praxis equivalent of an *object* in an object-oriented programming language.

The structure of literals allows us to express the *life-time of data* in a natural way. If we wish a piece of data d to exist for just the life-time of an object t , then we make the prefix of t be the prefix of d . For example, if we want the beliefs of an agent to exist just as long as the agent, then we place the beliefs inside the agent:

```
mr_collins.beliefs.clergymen_should_marry
```

Or if we want the state of a game to exist just as long as the game itself, then we place the state inside the social practice for the game:

```
process.whist.data.whose_move!brown
```

A related advantage of Exclusion Logic is that we get a form of *automatic currying*[32] which simplifies queries. If, for example, Brown is married to Elizabeth, then we might have “brown.married.elizabeth” in the database. In Exclusion Logic, if we want to find out whether Brown is married, we can query the sub-term directly - we just ask if “brown.married”. In traditional predicate logic, if married is a two-place predicate, then we need to fill in the extra argument place with a free variable - we would need to find out if there exists an X such that “married(brown, X)” - this is more inefficient as well as being more verbose.

6.2 Automatic Removal of Invalid Data

The exclusion operator supports the automatic clean-up of data which is no longer referenced.⁹ For example, a social practice p might have two states: a and b . State a might have two pieces of information, $x1$ and $x2$. State b might have three pieces of information $y1$, $y2$ and $y3$. Being in state a would be represented as:

$$p!a.x1 \wedge p!a.x2$$

State b would be represented as:

$$p!b.y1 \wedge p!b.y2 \wedge p!b.y3$$

Now if we are in state a (because the statement $p!a$ is true), and we switch to state b (by inserting $p!b$ into the database), all the local data from state a is *automatically* removed from the database according to the update rules for Exclusion Logic.

⁸ Compare this with Prolog - where it is much harder to remove all sentences containing a particular symbol. You can remove all predicates of a *certain arity*, but would have to separately remove the various different groups of predicates of different arities.

⁹ This is a form of simplified belief-revision, or garbage-collection.

6.3 Simple Postconditions

When expressing the pre- and post-conditions of an action, STRIPS¹⁰ has to explicitly describe the propositions that are removed when an action is performed:

```
action move(A, X, Y)
  preconditions
    at(A, X)
  postconditions
    add at(A, Y)
    remove at(A, X)
```

STRIPS finesses the frame-problem[12] by using the closed-world assumption: anything that is not explicitly specified as changing is assumed to stay the same. But there is still residual awkwardness here: we need to explicitly state that when A moves from X to Y , A is no longer at X . It might seem obvious *to us* that if A is now at Y , he is no longer at X - but we need to explicitly tell the system this. As Jeff Orkin writes [19]:

It may seem strange that we have to delete one assignment and then add another, rather than simply changing the value. STRIPS needs to do this, because there is nothing in formal logic to constrain a variable to only one value.

Exclusion Logic is a formal logic designed to express directly the natural idea that certain variables can only have one value. The interpretation of the exclusion operator means we do *not* need to specify the facts that are no longer true:

```
action move(A, X, Y)
  preconditions
    A.at!X
  postconditions
    add A.at!Y
```

When using Exclusion Logic, the postconditions are shorter and less error-prone. The “!” operator makes it clear that something can only be at one place at a time.

6.4 The Exclusion Operator Helps the Author Specify Her Intent

The semantics of the exclusion operator remove various error-prone book-keeping tasks from the implementer. But perhaps the exclusion operator’s main benefit is that it allows the Praxis script-writer to specify her intent more precisely. When we specify that, for example¹¹:

¹⁰ These comments apply equally to PDDL (the Planning Domain Definition Language).

¹¹ This is a typing judgement in Praxis, saying that a variable A (of type *agent*) has a unique sex G (of type *gender*)

```
A(agent).sex!G(gender)
```

We are saying that an agent has *exactly one* gender. This exclusion-information is available to the type-checker, which will rule out any piece of code which suggests that an agent can have multiple genders.

Some modern logic-programming languages are starting to add the ability to specify uniqueness properties of predicates [31], but they treat uniqueness properties as meta-linguistic predicates. Praxis is the first language to treat exclusion as a first-class element of the language.

6.5 Support Tools for Praxis

A domain-specific language needs a number of support tools before it becomes really usable for production work. Praxis comes with a number of tools¹² to make the script-writer's life easier:

- A type-checking system to find errors early. Praxis is strongly, but implicitly typed¹³: the author does not have to specify the types of all variables - instead, the system will infer the types of all variables and complain if a consistent assignment cannot be found.
- An Inspector for giving the author run-time access to the precise state of the simulation
- A Playback facility. The game stores the exact set of actions chosen by the player during a game, and writes them to a file. This allows the author to reproduce exactly a previous play-through. This playback feature required that we kept the simulator *fully deterministic* at all times.
- A Stress-Test tool which runs hundreds of instances of the game simultaneously, with all characters controlled by the computer. By running multiple instances of the game at high speed, we are able to find bugs and anomalies quickly.

7 Social Practices

7.1 Regulative Versus Constitutive Views of Social Practice

At the heart of the technical architecture is a commitment to a *constitutive* view of social practices. I will explain what this means by contrasting it with the alternative *regulative* view of social practices.

¹² When designing the inspection tools, we were heavily indebted to the sophisticated authoring tools in Inform 7.

¹³ Compare ML and Haskell.

The Regulative View of Social Practice Imagine a group of agents, each with his own set of goals and available actions. At any decision point, an agent chooses the action which best satisfies his goals (or expected goal satisfaction, once we take probabilities into account).

In this individualist picture, a social practice is just a set of *restrictions* on available actions which allows us to collectively increase our utility. For example, the driving-on-the-left practice restricts our actions (we can no longer drive on the right). We are prepared to accept this limitation on our freedom because it lowers the probability of a collision.

The Constitutive View of Social Practice According to the regulative picture, agents could act before they participated in practices. They already had goals, and already understood what actions were available to them. These were already given. All the social practices does is allow us to solve various coordination problems.

The constitutive view (first articulated explicitly by John Rawls in [20])¹⁴ rejects this assumption. According to the constitutive view, the action is *only* available within the practice. A nice example from Rawls’ original paper is the game of chess: you can move a carved piece of ivory from one square to another, but you can only *move your pawn to King 4* if you are playing chess. Again, I can utter a series of noises which sounds like “Ay doo”, but this only constitutes a *marriage-vow* in the context of a wedding-ceremony. The action is only available in the practice.

One way to see the need for a constitutive view of practice is to consider the vast array of actions I could possibly do now. Sitting here right now, I could lend the stranger on my left 10 pounds; I could tell the other stranger on my right that Paris is the capital of France; I could ring up my wife and enumerate the prime numbers.

With an infinite number of actions available to me, why am I not *overwhelmed* with choice? How do I ever find the time to make a decision?

In the constitutive view of practice, the affordances are always embedded in the practices. *The agent does not see the action as available unless he is already participating in the practice which makes it visible.* The agent isn’t overwhelmed by an infinite number of choices because *he only sees the affordances that are provided by the social practices he is in.* It is this constitutive model of social practices which lies at the heart of our simulation. In our implementation, we take this idea literally: every affordance is contained within a practice and is only available if that practice is instantiated.

7.2 Role Agnosticism

Tomasello [33] has argued that one of the critical differences between the great apes and humans is that, although both apes and humans can participate in

¹⁴ But this view has a long history and can certainly be traced back to Wittgenstein [29] and Heidegger[3], and arguably to Hegel[2] and beyond.

practices, only humans are able to effortlessly switch roles. As soon as a human infant (as young as two) participates in a practice, she adopts a birds-eye view of the practice, which means she is able, after participating once, to suggest we play again, *with roles reversed*. Other creatures do not exhibit the ability to play a practice from multiple roles. A great ape can participate in many sorts of practice - but he is *locked in* to the role which he played. He cannot see the practice from a bird's eye perspective. This is a distinguishing features of humans.

One of our big goals in Versu was that each practice would be *role-agnostic*: it would be authored without knowing which roles were being played by NPCs and which by human players.

7.3 Representing Social Practices

A social practice is a hierarchical collection of affordances, providing various options to its participants (who are characterised solely in terms of the roles they are playing).

A social practice is represented with the keyword `process`. Processes are specified with declarations, for example:

```
process.greet.X(agent).Y(agent)
  action "Greet"
    preconditions
      // They must be co-located
      X.in!L and Y.in!L
    postconditions
      text "[X] says 'Hi' to [Y obj]"
end
```

Here the term associated with the process is

```
process.greet.X.Y.
```

The processes can then be instantiated any number of times by adding sentences to the knowledge base. For example, if we add the following assertion:

```
process.greet.jack.jill
```

then the process will be active with substitutions [jack/X, jill/Y]. If, furthermore, Jack and Jill are in the same location:

```
jack.in!hill
jill.in!hill
```

Now the preconditions of the action are satisfied and the Greet action will be available to Jack on Jill. If we added another term:

```
process.greet.jill.jack
```

then *another* instance of the process will be active with substitutions [jill/X, jack/Y], and the Greet action will be available to Jill on Jack.

The language for describing actions has a number of the features described in PDDL[14]. In particular:

- Disjunctive preconditions
- Negation in preconditions (using negation-as-failure)
- Quantified preconditions (both universal and existential quantifiers, can be nested arbitrarily)
- Expression evaluation (the ability to perform numeric calculations)
- Domain axioms (the ability to define new predicates and relations in terms of existing relations)
- Conditional effects in postconditions

7.4 Respecting the Normative

Many of the practices we authored had their own individual sense of the normative. During a conversation, you *should* respond when spoken to, you *should* respect the salient topics, etc [25] [30]. During a meal, you *should* be polite about the food, etc. But the player still has a choice - she can always violate the norms if she wants to! The major requirements for modelling the normative were that:

- NPCs understand what they should and shouldn't do
- NPCs, if left to their own devices, should respect the norms (unless they have some particularly acute personality deviation which overrides their urge to respect the social mores)
- But the player should be free to violate these norms at any time!
- If the player violates a norm, the others should notice and respond accordingly

To get NPCs to respect norms, we add postconditions to norm-violating actions to mark that a norm has been violated. We similarly add postconditions to actions which should be performed to mark that a requirement has been respected. We give most¹⁵ agents strong desires to respect the social norms. When deciding what to do, the agents will see the consequences of their actions. If they see a norm-violation consequence, that will be a major disincentive.

Although NPCs typically do not violate norms, the player is allowed to do whatever she likes. If the player does step outside the bounds of propriety, the NPCs should notice and respond accordingly. Getting drunk, insulting the wine, refusing to answer when spoken to - all these norm-violations are only fun to play if they are *noticed*. When a norm-violation occurs, the practice which kept track of the norm spawns a sub-practice whose job is to mark that a violation has occurred. This responsive practice will provide options to the others: disapproving, forgiving, getting angry, and even (in extreme circumstances) evicting the character altogether.

¹⁵ The drama manager will occasionally lower these desires for certain agents (e.g. our rakish poet, Brown) when it wants them to behave outlandishly for dramatic purposes.

8 Agents

8.1 Autonomy

When an agent is deciding between various possible actions, he looks at the consequences of each action, and chooses the action which best satisfies his desires. He uses forward-chaining, rather than goal-directed backward chaining¹⁶.

This is a form of utility-based reactive action-selection ([11, ?]), rather than a full-blown planner. But it is an unusual utility-based method in that it is highly responsive to the fine details of the simulation.

In most systems which use utility-based decision-making, the agent's *estimation* of the consequences of the action is much simpler than the *actual* consequences. For example: in *The Sims*, when a Sim decides to go to the toilet, the actual consequences of the action are:

- Routing into the bathroom
- If there is someone already in the room, he expresses frustration, and exits
- Otherwise, he locks the door, sits down, and relieves his Bladder motive

These are the actual consequences. But the estimated consequences are much simpler. When considering going to the toilet, all he sees about the future is:

- He will relieve his Bladder motive

This discrepancy (between the actual and the estimated consequences) creates all sorts of issues. One particularly aggravating problem is that the estimated consequences miss all the *conditional effects*: effects which may or may not happen depending on various other aspects of the simulation state. For example: whether or not the going-to-the-toilet will be successful depends on whether there is somebody already in the bathroom. But the Sim does not consider this when planning; he thinks the action will always be successful. If there is in fact someone else in the bathroom, the Sim will attempt to use the toilet, but be thwarted by the other person. Then he will try to choose another action; his Bladder motive will still be unsatisfied, and he will attempt to use the toilet again... This behaviour can repeat indefinitely¹⁷. Now there are various kludges we can put in to avoid this particular problem. But the best way to fix this general *class* of problems is to address the root cause: instead of using a simplified estimation of the consequences, compute the *actual* consequences for decision-making.

¹⁶ SHOP[17] uses a similar approach: “Since SHOP always knows the complete world-state at each step of the planning process, it can use considerably more expressivity in its domain representation than most planners.”

¹⁷ The Sims is not an isolated example. *FEAR* contains exactly the same discrepancy between the planner's understanding of the action effects and the actual gameplay effects. In [19], notice the two separate fields in the Action class: `m.effects` for the planner's understanding of the effects, and the `ActivateAction()` function for the actual gameplay effects.

The NPCs in Versu look at the actual consequences of an action when deciding what to do. When considering an action, they actually execute the results of the action - rather than some crude approximation. Then they evaluate this future world state with respect to their desires. Then we *undo* the consequences of the world-state¹⁸.

This sort of decision-making is *broad* rather than *deep*. It doesn't look at the long-term consequences of an action - but it looks at *all* the short-term consequences. By looking at a broad range of features, it is able to make decisions which would typically only be available to long-term planners. For example, the NPCs are able to play a strong game of whist. When considering the various cards the whist-player can play, the decision-maker computes the various features of a move (whether it counts as winning-the-trick, whether it counts as throwing-away-a-card, trumping, getting rid of a suit, etc). These conditional-effects determine the score of playing the card. Using such simple appraisals, and giving the NPCs suitably-weighted desires to perform actions which satisfy these appraisals, is *all that is needed* for the NPCs to automatically play a strong game. There is no need for any separate sui-generis whist-playing decision-procedure.

Our short-term planner elegantly handles large dynamic sets of goals, allowing characters to select actions that advance multiple goals simultaneously.

Utility is computed by summing the satisfied desires. A want is a desire to make a sentence true - and that sentence can be any sentence of exclusion logic - to any level of complexity. So, for example, Brown (our rakish poet) likes annoying upper-class men. He wants it to be the case that there exists an *Other* such that:

```
Other.sex!male and
Other.class!upper and
is_displeased_with.Other.brown
```

Each want comes with an associated utility modifier. This want is tagged with a utility-modifier of 20. Every separate instantiation of this desire gives an additional 20 score. If, for example, Brown can see that one remark would simultaneously annoy three upper-class men, then that remark would score three times higher than a remark that just annoyed one.

8.2 Individuality

When attempting to model individual personalities, one common method is to implement a small finite set of personality traits, and define a character as a

¹⁸ It is only because we are able to undo actions that this approach is workable. The language instruction primitives of the Praxis language were designed to be efficiently undo-able. If we couldn't undo an action, the only way we could compute the actual consequences would be by making a copy of the entire simulation state, performing the action in that copy, and then throwing it away. But copying the entire simulation state is prohibitively expensive when we are considering so many actions for so many agents.

combination of these orthogonal traits [6]. We wanted a more expressive system, in which there were an *infinite* number of personalities - as many personalities as there are sentences in a language.

The fact that the planner looks at the *entire simulation state* means that the range of things that agents can desire is very wide. Some examples of individual desires:

- The doctor is sexist: he wants $\exists X \text{ leader!}X \wedge X.\text{sex!male}$
- Brown enjoys annoying upper-class men: he wants $\exists X \text{ is_displeased_with.}X.\text{brown} \wedge X.\text{sex!male} \wedge X.\text{class!upper}$.
- Peter doesn't like to be alone: he hates it when $\exists L \text{ peter.in!}L \wedge \forall X \text{ X.character} \wedge X \neq \text{peter} \rightarrow \text{not}X.\text{in!}L$

The expressive range of the logic is what allows us to specify such fine-grained personalities.

8.3 Relationships

We use role-evaluation (based on Sacks' Membership Categorization Devices [24]) to model many different sorts of relationships described.

At any moment, we are simultaneously participating in many different practices. In "Pride and Prejudice", Darcy, for example, is simultaneously a member of the gentry, a friend of Mr Bingley, a potential suitor. For each role he is playing, one crucial question is *how well* he is playing that role. Different people have different evaluations of how well someone is playing a role.

This concept, **role-evaluation**, is at the heart of the relationship model. Just as agents can be participating in multiple social practices concurrently, just so an agent can have multiple concurrent views about another agent.

Some of the role-evaluations we use for our Jane Austen episodes include:

- how well-bred someone is
- how properly he is behaving
- how attractive someone is (evaluating someone as a potential romantic partner)
- how generous, intelligent, amusing, etc
- how well they are performing their familial role (father, mother, daughter, etc)
- whether they are a good husband or wife

In "Pride and Prejudice", characters evaluate each other according to their success at these various roles:

- The Bingley sisters find Elizabeth lacking in style, taste, beauty
- They judge Jane to have low family connections
- Mrs Bennet judges Charlotte to be plain

Characters remember the *reason* for these evaluations¹⁹:

¹⁹ The evaluations are stored inside the agent in terms of the form *Agent.relationship.Evaluated.role.Role!Value!Explanation*.

- Mr Bingley is a good suitor because he was so affable at the dance
- Mr Darcy is a bad suitor because he was rather rude at the dance
- Jane is a bad catch because her family is so badly connected

Characters make these evaluations public when prompted (or unprompted):

- Mrs Bennet says Darcy is a bad match for Elizabeth on account of his rudeness
- The Bingley sisters say Jane is a bad match for Mr Bingley on account of her low connections

These public evaluations can be communicated from one character to another.

- Mr Wickham tells Elizabeth that Darcy is dishonourable and she believes him

Sometimes, people disagree about their evaluations of a character:

- Elizabeth and Jane disagree about whether Mr Wickham’s dark hints mean that Darcy is blameworthy

These evaluations affect the characters’ autonomous behaviour:

- Characters will be invited over if they are evaluated sufficiently highly
- Characters will propose if they evaluate the other as a suitable match
- Characters will display gratitude for doing favours (e.g. the wife and daughters are grateful to Mr Bennet for paying Mr Bingley a visit)

These evaluations also affect the *tone* of their autonomous behaviour:

- Miss Bennet is not gracious when Mrs Bennet apologizes because she is not valued highly
- Their effusiveness in looking after somebody depends on their evaluation (e.g. the Bingley sisters are less effusive in their concern for Jane because they do not rate her highly)

Updating Role-Evaluations Characters can acquire evaluations in three ways:

- They can start with the evaluation hard-coded into them.
- They can acquire the evaluation when they interpret one of the other’s actions
- They can hear someone else’s evaluation and decide to believe it

Characters acquire new evaluations of others when they see them performing actions, and interpret those actions in a particular light. These interpretations are themselves actions which the characters decide to do - they have a choice how to interpret others’ actions.

Relationship States Relationship evaluations are multiple and *asymmetric*: x may judge y according to multiple different roles, and x 's views on y may not be the same as y 's views on x .

But as well as these multiple views, we also model a single *symmetric* notion: the public relationship state between the characters. This is the official long-term stance between the characters: whether they are friends, lovers, siblings, or enemies.

8.4 Emotions

We used a fine-grained set of emotional states, based on Ekman's work [4]. An advantage of a text-representation was that it was easy to express the various emotional states. It would be much harder to express the fine-grained distinction between e.g. being embarrassed and humiliated if we had to show it in a 3D face. In order for an emotional state to be intelligible to another, the character should be able to *explain* it. To do this, the character remembers who the emotion is directed towards (e.g. I am annoyed with Brown), and the event which prompted the emotional change (e.g. Brown's insult).

An agent changes emotional state when performing an action. Reactions, in particular, are a rich source of emotional state changes.

Our representation of agent's emotional states is simple and straightforward: the agent has only one emotional state at a time, and any new emotional state always overrides the previous one. The agent also remembers his *previous* emotional state, so that we can have autonomous decisions based on mood-switching. An agent, for example, may not like to laugh when he is already in a bad mood.

8.5 Beliefs

The world-state is shared amongst the agents. We do not, for memory reasons, give each agent his own separate representation of the world. Instead, we give them all access to the one authoritative world model. This means, of course, that misunderstandings etc cannot be implemented fully.

For *specific cases* where we want false beliefs, or factual disagreements, we store individual beliefs for that specific issue. So, for example, early on in the ghost story, when there are various spooky (but inconclusive) happenings, the characters can disagree about whether these events are caused by a ghost or there is some more scientific explanation. Another example: typically, inter-personal evaluations are shared and accessible to all. But if we want some people to know - and others to be unaware - that Frank loves Lucy, then we store this as a belief, and gate certain actions on whether the actor *believes* that X loves Y , rather than on whether it is *true* that X loves Y .

8.6 Character Arcs

The most complex part of a character description file is the character-arc: a story-arc for that individual character, describing how his objectives and emotions change over time. This arc references only facts about that individual

character, so it can be brought into any story in which the character is placed. Our philandering poet, Brown, for example, might choose to take another mistress - but once he has seduced her, we might want him to start to feel bored and trapped once again. If a social practice provides choices about external low-level actions, the character arc represents internal high-level choices: does Brown want to take another mistress, or focus on improving his poetry? The author designs the character arc to give the character choices about what he wants to *be* - not just what he wants to *do*.

The character arc also specifies a variety of possible epilogues for the character. Each epilogue describes the end-state of the game from the perspective of that character's defining life-choices. If Brown decides to distract himself from his malaise by taking another mistress, this could end up with his ditching her, or with him deciding - to his own great surprise - that he will remain committed to her after all. The character-arc has its own sense of drama, separate from the story: it is here that the character may achieve self-awareness, sink into despair, or transform himself.

True character transformation comes about in the moments where a character decides to set aside an old want, adopt a new one, or act in contravention of his own desires. This ability - the ability to choose an affordance that is not what the character simulation would prefer - is not available to NPCs - it is only available to the player. The natural result of this is that the characters controlled by humans have opportunities for change and development, while characters controlled by NPCs will not exercise those opportunities; instead, they will continue to play supporting roles.

The mixing of character arcs with episode story management also produces a *productive interference* when it comes to the meaning of the stories as they're experienced and interpreted by the player. For instance, the story manager of the murder mystery might dictate that the characters can identify and confront the murderer and then choose either to turn him in to the law or to help conceal his guilt. These might be the only possibilities recognized by the episode structure. However, different motivations might manifest themselves as a result of the character arcs. For instance, if the character who discovers the murderer's guilt is in betrothed to the murderer's son, this presents a motivational question *to the player*: should she push to convict out of a love of justice, knowing that this will swamp the family in scandal, lead to the end of her engagement, and cause a disappointing end to her personal story arc?

Such dilemmas as these are not explicitly modeled by Versu; rather, they fall out of the creative interference between characters who have personal motivations and desires for the outcome, and narrative cruxes that force dramatic change.

9 The Core Model of Beliefs, Emotions, Relationships and Evaluations

9.1 Inter-Practice Communication via the Core Model

Social practices typically track internally one or several variables pertinent to that practice: for instance, a dinner practice might keep track of how many guests had been served wine, or which course it was; a whist practice might track which agents were playing in the game, what cards they had been dealt, and which suit was trumps.

This kind of internal information, however, does not allow the different practices to communicate with and affect one another. Instead, the agent information described above – consisting of relationship states, beliefs, emotions and evaluations – serves to communicate between practices.

To be effective – that is, to cause repercussions for other characters and changes within the story – a character’s actions within Versu need to change one or more of these core model elements.

Changes to emotional states might last for a few turns and might affect the way the character is described doing things (and her appearance on screen), but a character will experience many emotions in the course of a game. Unless a given mood affects one of the longer-term decisions, its unlikely to determine how the story ends.

By contrast, characters evaluations of others play into relationship state decisions. A character who evaluates her flirt as cruel or bumbling will have the opportunity to begin a “breaking up” practice, which will end the romantic relationship state between these characters and produce a negative “rejected flirt” relationship state instead. It does not matter to the core model *how* the character’s flirt made himself unacceptable: it might be that he clumsily spilled his wine at dinner, or stepped on her foot during a dance, or said something inconsiderate to her mother. It might be that he committed one large transgressions or a series of smaller ones. Regardless of the originating practice, when the character’s evaluation of him becomes too negative, the opportunity to break up will present itself.

Relationship states and character self-evaluations are the most significant and lasting part of the core model.

During play, a relationship state may function to make particular actions available: for instance, the practice allowing a couple alone together to kiss will become functional only if the pair are in a romantic relationship state.

In addition, each character in the drama begins with certain relationship or self-belief goals. For instance:

- Miss Bates begins wishing to be friends with someone
- The poet Brown, who has a chip on his shoulder about his illegitimacy, may have a desire to form an inimical relationship with an upper-class man
- Lucy starts out hoping either to find a protector or else to become more confident in herself

Succeeding or failing to achieve these goals affects story outcomes, as each story concludes by narrating both how the extrinsic episode ended (was the murderer identified? was he tried?) and how the character arc went for the player character (did Elizabeth conclude the story engaged to Mr Darcy?).

9.2 Using the Core Model to Promote Player Agency

Most of the player’s agency in the game therefore comes from using the affordances made available by the social practices to affect the core model in some fashion. In order to maximize this sense of agency, we identified the following design goals:

- Player actions should be rewarded either with information about the world state or changes *to* the world state.
- Changes to short-term qualities should happen frequently.
- Changes to short-term qualities should lead to a chance for the player to back down or persist in an attempt. For instance, if the player’s character has evaluated another as somewhat attractive, this should lead to opportunities to either flirt with or ignore the object of affection – allowing the player to indicate whether it is really her intention to try to develop a relationship there.
- Changes to long-term qualities should be strongly marked. If a player becomes someone’s friend, flirt, enemy, etc., that event should entail several moves of interaction and be marked out clearly by the user interface. (Currently, making a friend, flirt, or enemy is an achievement which is strongly noted when it occurs.) These events are key to the story and should be noted as significant accomplishments.
- Relationship state changes should require active choice from both characters. It should never be possible for an NPC unilaterally to change his or her relationship to the player, though the NPC might offer the player an unavoidable choice, either prong of which will have a significant state-changing outcome (e.g. “marry me *or* we will break up”).

9.3 Using the Core Model with Autonomous Agents

The use of autonomous agents also introduces an additional design goal into the system. As explored above, each agent may have arbitrarily complex preferences. Lucy likes to evaluate other characters positively. Miss Bates likes to be in a good mood. Mr Quinn likes to have a bad opinion of Frank Quinn. However, these preferences can only produce significantly different behaviour between agents *if the range of available affordances, and the effect of those affordances, is sufficiently great*. Otherwise, agents will be comparatively scoring just a handful of options, and they are unlikely to produce distinct behaviour, even if their formulae for scoring differ widely. This produced an additional design goal:

- Individual actions should ideally produce change to *multiple* qualities or types of quality.

This is especially easy to demonstrate with respect to conversation. A given line of conversation may accomplish any of the following tasks:

- shift the speaker to a new belief about the world
- communicate a belief
- shift the speaker to a new emotion
- communicate one or more emotions
- shift the speaker to a new evaluation of another character
- communicate an evaluation of another character
- shift the speaker to a new self-evaluation
- communicate the self-evaluation of the speaker
- count as an atomic action to which others may react

For example: Mr Collins may speak about how Lady Catherine, his patroness, has complimented the sermons he preached at her parish. This quip²⁰ is marked to do each of the following things:

- communicate that Mr Collins has a positive self-evaluation about intelligence, inviting listeners to accept or reject this view of Mr Collins
- communicate that Mr Collins admires Lady Catherine’s status and patronage, inviting listeners to accept or reject this view of Lady Catherine
- shift Mr Collins to the “pleased” emotional state, because he enjoys dwelling on the compliments he has received

Likewise, Lucy might say, “Oh no! I’m afraid there must be a ghost here!” when she has not previously believed in ghosts. That quip would have the following effects:

- shift to a belief in ghosts, changing Lucy’s mentality on this point for the remainder of the game
- communicate a belief in ghosts, allowing other characters to accept this belief or rebut it
- shift to a fearful emotion
- communicate a fearful emotion, allowing other characters to offer Lucy comfort

...so that it would be possible for other characters to respond by reassuring her, telling her that she’s superstitious to believe such a thing, or agreeing that there probably is a ghost.

Coding speech and other social actions in this way allows agent-driven characters to make a more nuanced use of their various wants. It also produces texts in which characters appear to react to both surface and subtext, as in this final example: Mrs Elton might say, “What a handsome parlour you have; it is *almost* as graciously appointed as my sister’s!” This quip would accomplish each of the following:

²⁰ Dialog is authored in units called *quips*, which are combinations of a text template together with a collection of possible effects on social and emotional state.

- communicate a moderately positive status view of the person addressed
- communicate a superior status view of her own family
- constitute a “be complimentary” action

Some characters might respond to the “compliment” by thanking Mrs Elton; others might reply by being self-deprecating about their status or by putting down Mrs Elton’s family in order to correct her self-evaluation that they disagreed with.

10 The Story Manager

Unlike some other systems [22], Versu does not have a general-purpose drama manager which dynamically combines story elements to produce a wide variety of different stories. Instead each individual episode has its own *individual story manager*, which encodes the author’s understanding of the narrative goals for that particular episode. A particular story (say, a murder mystery) has certain key recognisable moments (the victim makes himself unpopular, the victim is killed, the body is discovered), and the story manager is responsible for making sure these events happen at the right moment.

The story manager for an episode is a high-level director who does not like to micro-manage. Given a stock of characters and a set of social practices, all our story manager likes to do is initiate practices, watch their progress, and insert occasional changes. It leaves the performance of those practices, and the individual decisions, to the individual autonomous NPCs.

The story manager is a reactive process (itself implemented as a social practice). It starts by creating characters and placing them in initial social situations. Once these characters have been given some interesting goals, it often leaves those autonomous characters to their own devices for some time, before the next intervention. The story manager watches what is going on, spawning new social practices, and tweaking individual goals, to keep things moving.

For example, in the murder-mystery, the story manager wants, after the meal has finished, for the people to gather together to perform *some sort* of group activity: reading together, music, whist. But it doesn’t mind which *particular* activity - so it spawns different practices on different occasions, leading to significantly different run-throughs.

Our story managers are significantly less ambitious than some systems [22]. They do not plan ahead, anticipating the narrative consequences of various dramatic moves, scoring each move according to narratological criteria. Instead, our story-managers are reactive processes, hand-crafted for each episode. This gives the author strong control over the outcome and quality of the story, at the expense of emergence at the narrative level.

11 Related Theory: Computational Models of Social Practice

11.1 Schank and Abelson's Scripts

Schank and Abelson's work on scripts ([27, ?]) has been hugely influential. They were one of the first in the AI community to articulate the important idea that an individual action is not intelligible on its own: its intelligibility comes from the social practice in which it is embedded. They used the term *script* to describe a computer model of a routine social practice: eating at a restaurant, travelling on a bus, visiting a museum. A script is a state graph containing a distinguished path which is marked as "normal". (For example, in the restaurant script, the normal path involves the customer ordering, eating the meal, and then paying for it).

The script describes *coordination of multiple actors*: characters were assigned to roles and the script understood which actions were expected for each role in each state. The script also achieves *continuity over time*: an individual agent's sequence of actions over time is intelligible as a sequence of causally-linked actions as the script travels through various states.

Schank and Abelson's theory accommodates the important idea that multiple scripts can be running *concurrently*, and can *interfere* with each other. One example they give is:

John was eating in a dining car. The train stopped short. John's soup spilled.

Here, the eating script and the being-on-a-train script are running concurrently. A problem in the train script then causes an interference in the eating script.

Schank and Abelson's work on scripts was a major source of inspiration to us. But our model of social practice is different in a number of ways. First, Schank and Abelson used scripts as a way of understanding natural language stories, while we use social practices as a way of guiding autonomous behaviour in an interactive system. Second, Schank and Abelson use so-called "scruffy" methods to model social practices (Conceptual Dependency Theory is a graph-based representation without a formal semantics), while we model the entire simulation state declaratively using a modal logic. (In Versu, we tackle "scruffy" research problems with "neat" methods). Third, a Schankian script describes a social practice from a *particular perspective*: from the viewpoint of one distinguished role. Schank and Abelson are explicit about this ([27], p. 210):

A script must be written from one particular role's point of view. A customer sees a restaurant one way; a cook sees it another way.

Again [28] p.152:

A script takes the point of view of one of these players, and it often changes when it is viewed from another player's point of view.

In Versu, by contrast, a social practice is authored from a birds-eye perspective: the domain-specific-language supports an authoring style in which practices are *agnostic* about which particular character is playing which role. In Versu, a restaurant script is written *once* and incorporates both the customer’s and the cook’s perspectives.

11.2 Moses and Tenenholtz’s Work on Normative Systems

Moses and Tenenholtz [16] have also developed a computational model of social practices. They define a *normative system* as a set of *restrictions* on available actions such that, when these restrictions are respected, there is a *guarantee* that a certain desirable condition obtains. For example: on a busy road, the desirable condition might be that no cars hit each other, and the restriction might be that all cars drive on the right hand side of the road. Part of the power of their work is that, using a type of modal propositional logic, they can *prove* that certain norms guarantee certain desirable properties.

Their approach is related to ours in that they use formal logic to describe social systems. But there is one fundamental difference: they see social systems as *restrictive* rather than *constitutive*. Imagine an agent who already has a set of available actions. A normative system, in their sense, provides a *restriction* on the set of actions. The restriction on the agent’s freedom is offset by the (provable) desirable properties of everyone obeying that restriction: forcing me to drive on the right is a restriction on my ability to drive on the left, but the guarantee that I can drive without collision offsets that restriction. Versu, by contrast, uses a *constitutive* model of social practice in which social practices *make new actions available*: placing a card down on the table only counts as *trumping with the Jack of Spades* within the context of the whist game in which it is embedded.

12 Related Interactive Systems

Starting with Tale-Spin[15], there have been many attempts to generate narrative using a simulationist agent-driven architecture. The recurring problem with these attempts has been that the generated stories lacked narrative coherence. It has proven very hard for an author to achieve any sort of narrative control by fiddling with the parameters of individual agents. When building Versu, we were hoping to find a spot in design space that has the generativity of simulation, while also having a satisfying degree of narrative coherence and author-ability.

In a recent paper [21], Mark Riedl and Vadim Bulitko provide a clear taxonomy for describing design choices in interactive narrative systems. Two fundamental questions that they are consider are:

1. **Authorial intent.** To what extent does the human author’s storytelling intent constrain the narrative? How much of the story is decided in advance by the author, and how much is generated by the player and computer during play?

2. **Virtual character autonomy.** Does each individual character make up his own mind, or is each character merely a puppet controlled by a centralised drama manager?

In terms of authorial intent, Versu is somewhere in the middle between a manually authored choose-your-own adventure, on the one hand, and an automatically-generated emergent narrative, on the other. Each episode in Versu comes with its own episode-specific drama-manager: a reactive agent which mostly sits back and watches, occasionally intervening to push the narrative forward. In this respect, Versu is rather like Façade: for each episode, there is a specific situation which has been carefully authored, but the exact path taken through the narrative landscape - the how and the why - is up to the player to determine²¹.

In terms of character autonomy, Versu is strongly simulationist. Each character chooses his next action based on his own individual beliefs and desires. It is very rare indeed for the drama manager to override the characters' autonomy and force them to do something. Instead, the drama manager typically operates at a higher level - by creating new social practices, or tweaking the desires of the participants.

Unusually, the drama manager is also modelled as an autonomous agent, and chooses which (meta-level) action to do based on her own (meta-level) desires. This means that in some episodes, the player can actually *be* the drama manager.

12.1 Comparisons with Façade

Playing the same scene from multiple perspectives Whereas Façade chose a middle-ground between the story-driven and agent-driven approaches, Versu is much more heavily agent-driven and simulationist. At the heart of Versu's simulation of social practices is a distinction between the *roles* in the social practice and the *characters* who are assigned to those roles. Because the player can play different roles in a situation, and can assign many different permutations of characters to the roles, a Versu situation has much more variation and replayability than the Façade scenario. If we were to implement a version of the Façade scenario in Versu, the player would not be constrained to playing the guest - she could also play Trip or Grace. Further, in a Versu version of Façade, the player could assign different characters to the roles. The player could assign, say, Mr Darcy to play the male host, and Elizabeth Bennett to play the female host, and see how differently it plays out.

The difference between joint behaviours and social practices A story-driven interactive narrative lets a single drama manager determine what happens next. (This provides continuity at the cost of emergence). At the opposite extreme, an agent-driven interactive narrative lets each individual agent determine

²¹ Versu's drama manager is rather less ambitious. Façade models a sense of rising tension and chooses the next beat by finding the one which most closely matches the intended current tension. We do not do this.

what he will do next. (This provides emergence at the cost of continuity). One of the most striking architectural ideas in *Façade* is the use of joint-behaviours (JBs) which coordinate a group of agents and are *intermediate* between individual agents (on the one hand) and a single drama manager (on the other). A joint behaviour can express synchronisation between its participants and can enforce continuity between individual behaviours. But a joint behaviour, as Mateas and Stern use it, is more restrictive than a social practice in that:

- A joint behaviour (JB) is a way of coordinating *NPCs* - there is no equivalent of a joint behaviour for coordinating *PCs* - or for coordinating a mixture of *PCs* and *NPCs*
- When an NPC is deciding whether to enter a joint behaviour, this decision is not based on his individual personality or desires. Deciding whether to join in depends only on whether the NPC *can* participate (whether he has an individual behaviour which matches the specification of the joint behaviour) - and not on whether he *wants* to²².

A social practice is like a joint behaviour in that it is responsible for coordination and continuity. But it is more general in two ways. First, a joint behaviour coordinates *NPCs* only, while a social practice can coordinate both *NPCs* and *PCs* together. Second, a social practice does not wrest control from the individual agent - it always respects the individual agents' autonomy. It provides *suggestions* - but leaves it up to the individual agent what to do. In *Versu*, the social practice is a coordinating entity which is intermediate between individual agents and a drama manager - but this coordinating entity does not wrest control from the individual agents. In *Versu*, it is always the individual agents who decide what to do.

What this discussion shows is that the broad question of virtual character autonomy turns out, on closer inspection, to divide into three separate questions:

- What sort of entity controls *decision-making*?
- What sort of entity provides *coordination* between agents?
- What sort of entity provides *continuity* over time between the actions of a single agent?

Façade answers all three questions in the same way: the joint-behaviour provides coordination, continuity, and is also responsible for making decisions. In *Versu*, there are separate answers to these questions: the social practice provides coordination and continuity, while the *individual agent* makes the decisions.

²² See [7] p.75: "If all agents respond with intention-to enter messages, this signals that all agents in the team have found appropriate behaviors in their local behavior libraries." The only way in ABL for an individual agent's personal preferences to affect the decision to join in is if that preference is added as an explicit *precondition*: "Note that the preconditions can also be used to add personality-specific tests as to whether the Woggle feels like playing follow the leader" ([7] p.78). As Mateas later acknowledges, "A possible objection to pushing coordination into a believable agent language is that coordination should be personality-specific; by providing a generic coordination solution in the language, the language is taking away this degree of authorial control" ([7] p.226).

The cost of content production The scenario in *Façade* takes about 20 minutes for the player to complete. This episode took 3+ man-years to create. A comparable 20 minute length episode in *Versu* takes about 2 months to create. *Versu* also contains longer episodes. The ghost-story, which takes 45 to 60 minutes to complete, took 6 months to produce.

In terms of the number of behaviours, *Façade* has 30 parametrized speech acts created during the 3+ years in development. *Versu*, by contrast, has more than 1000 parameterised actions, authored in 1 year in development.

We attribute our faster content production time to two main factors. First, using text output rather than 3D animated characters saved us a lot of production time. Second, the domain-specific language in which we authored behaviours (*Praxis*) was a very high-level declarative language for creating content. A high-level declarative language can express behaviour more compactly than a procedural language²³.

12.2 Comparisons with Prom Week

Prom Week[13] is a social simulation of high-school student social-life developed by a group at UCSC²⁴. At a high-level, *Prom Week* has a lot in common with *Versu*: they are both aiming to provide interesting individual characters in dramatic situations. But at the ontological level, there are some fundamental differences.

Activity in *Prom Week* involves a sequence of discrete speech acts. In *Versu*, individual actions are *coordinated* by social practices which provide meaning to *sequences* of actions. For example, a game of whist involves a *sequence of actions by multiple participants*. These actions make sense as a whole because they are all contributing to the one unifying practice: the game.

A concrete example: suppose one character suggests to some others that they retire to the drawing room to listen to some music. Now in *Prom Week*, this request would be an individual speech act: others would accept or reject this proposal, and then the speech act would be over. But in *Versu*, this suggestion is *part of a larger social practice*: the group deciding what they should do next. This larger practice involves the group as a whole achieving consensus (or failing to achieve consensus) on what they should do. Others may agree with the proposal to listen to music, or they may suggest an alternative pastime (reading, dancing, whist). Others may take sides, attempt to dominate, or back down. In *Versu*, the simple request speech-act is embedded within a larger practice which gives it *intelligibility* and provides *continuity*. In *Prom Week*, by contrast, behaviour is just a sequence of isolated and unrelated speech-acts.

²³ *Façade* is authored in ABL, a procedural domain-specific language built on top of Java.

²⁴ We were advisors on this project, and have had many fruitful discussions with the developers over the years.

12.3 Situating Versu in Design Space

We conclude this section by relating Versu to other games in terms of the three separable aspects of character autonomy²⁵:

- What sort of entity controls *decision-making*?
- What sort of entity provides *coordination* between agents?
- What sort of entity provides *continuity* over time between the actions of a single agent?

	Decision Making	Coordination	Continuity
CYOA	DM	DM	DM
The Sims	Agent	Speech Act	-
Prom Week	Agent	Speech Act	-
Façade	DM/JB	JB	JB
Versu	Agent	Practice	Practice

Fig. 3. Different ways of handling three aspects of character autonomy.

In a Choose-Your-Own-Adventure (CYOA), there is one entity (the static pre-authored story-graph, functioning as a non-reactive Drama Manager (DM)) which decides what happens next, provides coordination between agents, and continuity between action.

In *The Sims*, the individual agent decides what to do next. Coordination between agents is limited to individual speech-acts²⁶. In *The Sims*, there is no continuity between actions over time.

Prom Week is similar, in that decisions about what to do are always made by the individual agent. Coordination between two agents lasts for the duration of an individual speech-act and involves an initiating sentence followed by a response. Again, there is no continuity between actions over time.

Façade uses a Drama Manager to decide what happens next. It uses Joint Behaviours (JBs), described above, to achieve coordination between agents, and continuity over time.

Versu uses the *social practice* to achieve coordination and continuity, but always lets the individual agents decide what to do.

13 Limitations and Further Work

13.1 Limitations of the Agent Model

When using a utility-based decision-maker, tweaking the various desires (there are already over 300 desires in the system) so that the preferred action scores

²⁵ See Fig. 4.

²⁶ Performing a speech act spawns an invisible object that lasts only for the duration of the act, and coordinates the animations and responses of the two participants.

higher is a difficult and time-consuming tuning problem. The Inspector helps authors understand autonomy tuning problems, by showing the scores of each action - and the *reasons* why the action gets the score it does - but even with this tool, it is a difficult and unrewarding problem.

Another limitation with the system is the way agents' beliefs are expressed. To simplify the implementation, all beliefs are represented as sentences involving a two-place predicate and a pair of constants. We can represent Mrs Quinn's belief that Lucy is compromised by Frank Quinn, or Darcy's belief that the ghost was killed in the study, or Elizabeth Bennett's belief that she should not marry Mr Collins - but we cannot represent:

- Beliefs involving universal quantifiers e.g. "*Everyone* has become insane"
- Beliefs involving existential quantifiers e.g. "the murderer is one of the guests"
- Beliefs about others' beliefs e.g. "Mr Quinn believes that Lucy believes that Mrs Quinn is the murderer"

13.2 Limitations of Our Representation of Social Practices

Simplifying Assumptions of the Social Model Our model of social practices simplifies in two ways. First, the agents have a shared understanding of the state of the social practices. It is not possible in this model for two agents to have divergent understandings of the state of the situation (for example: disagreeing about whose move it is in a game of chess). Instead of modelling each individual agent's beliefs about the state of the practice, we just model the practice once - and give agents access to it. Second, even if we did give each agent his own individual model of the practice, so that they could diverge, there would still be a shared *understanding* of the practice: both agents would agree, for example, that greeting is something you do when you are sufficiently well acquainted. A deeper model of practices would allow individual agents to have their own *interpretations* of the practice.

Multiple Concurrent Practices Complexify Authoring One of the striking things about the architecture is that it allows multiple practices to exist concurrently. Most of the time, there are many practices running at once, each providing various options to the agents. It is because there are so many practices that the player has such a wide range of options at any time.

But this complexity comes at a price: the fact there are multiple concurrent practices complicates the tuning and debugging of the scenes. Each action from each practice needs to be scored against all the other actions from all the other concurrent practices.

Allowing multiple concurrent practices generates another, deeper authoring problem. Sometimes something happens that is so obviously important that all the other things that are going on should be forgotten, for the moment. For example, in our murder-mystery episode, when the dead body is discovered, there may be many other things that were going on: there may be a budding flirtation between two of the guests, some of the characters may have got drunk,

or violated one of the norms of Regency England, and be getting told off. But when the body is discovered, the affordances from these other practices should be suppressed. The *seriousness* of the situation should mean jokes and flirtations are not even considered. We currently use a rather simple mechanism for suppressing the affordances from other practices: a *dominating* practice is one which, when active, suppresses the affordances from any other practices. But this dominating mechanism is too broad and crude for the case in hand: there are *some* other practices which should co-exist with the discovery of the body: for example, weeping at the loss of a loved one. What we really want (but don't know quite how to implement) is Heidegger's idea of a *public mood*[3] which opens up a range of possibilities and closes off others. In our example, when the dead body is discovered, this should create a public mood of *shock*, and this mood should reveal various practices (grieving, examining the body, wondering who could have done such a thing) while obscuring others (drinking, joking, flirtation).

13.3 User-Generated Content

Once she is up to speed with Praxis, a writer can produce a 20 minute episode (with a rich variety of end-states and the ability to play from multiple perspectives) in 1-2 months. Although this is a significant increase in content-production speed over Façade, for example, the speed of content production is still an issue for us.

In order to make content production possible in a reasonable amount of time, we have built on top of the Praxis language an authoring tool called *Prompter*. Graham Nelson (author of the Inform language for interactive fiction [18]) has assisted in the design and implementation of Prompter, which allows writers rapidly to create scenes and dialogue in a format that resembles a play script. The script is marked up with additional text indicating the emotional and evaluative effects of a given piece of dialogue or action. The Prompter software then converts the script into raw Praxis.

Prompter-generated episodes can also include other, pure-Praxis files at need, which makes it possible to coordinate generated data with newly invented social practices, props, and behavior.

The existence of Prompter has significantly sped up our internal writing process, making it possible to create a substantially branching 20-minute episode in less than a week, rather than in 1-2 months as was formerly the case.

Acknowledgment

The authors would like to thank Tom Barnet-Lamb, Chaim Gingold, Ian Holmes, Ian Horswill, Michael Mateas, and Graham Nelson for feedback and guidance throughout the project.

References

1. Blackburn, P., de Rijke, M., Venema, Y.: *Modal Logic*. Cambridge University Press (2002)
2. Brandom, R.: *Making It Explicit*. Harvard University Press (1998)
3. Dreyfus, H.: *Being-in-the-world*. The MIT Press (1990)
4. Ekman, P.: *Basic Emotions* (1990)
5. Evans, R.: *Introducing Exclusion Logic as a Deontic Logic*. *Deontic Logic in Computer Science*. Springer (2010)
6. Evans, R.: *Representing Personality Traits as Conditionals*. *Proceedings of AISB (Artificial Intelligence and the Simulation of Behaviour)* pp. 64-82 (2008)
7. Mateas, M.: *Interactive Drama, Art, and Artificial Intelligence*. Ph.D. Thesis. School of Computer Science, Carnegie Mellon University (2002)
8. Mateas, M., Stern, A.: *Writing Façade: A Case Study in Procedural Authorship*. In *Second Person* ed. Harrigan and Fruin (2007)
9. Knuth, D.: *Digital Searching*. *The Art of Computer Programming Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley (1997)
10. Lewis, D.: *Scorekeeping in a Language Game*. *Journal of Philosophical Logic* p.379 (1979)
11. Maes, P.: *How To Do the Right Thing*. *Connection Science Journal* (1989)
12. McCarthy, J., Hayes, P.: *Some Philosophical Problems from the Standpoint of Artificial Intelligence*. *Machine Intelligence* 4 (1969)
13. McCoy, J., Mateas, M., Wardrip-Fruin, N.: *Comme il Faut: A System for Simulating Social Games Between Autonomous Characters*. *Proceedings of the 8th Digital Art and Culture Conference*, pp.1-8 (2009)
14. McDermott, D.: *PDDL - The Planning Domain Definition Language (Version 1.2)*. Yale Center for Computational Vision and Control (1998)
15. Meehan, J.R.: *Tale-Spin, An Interactive Program That Writes Stories*. *Proceedings of the Fifth IJCAI* (1977)
16. Moses, Y., Tenenholz, M.: *On Computational Aspects of Artificial Social Systems*. *Proc 11th DAI Workshop* pp. 108 - 131 (1992)
17. Nau, D., Cao, Y., Lotem, A., and Muoz-Avila, H.: "SHOP: Simple Hierarchical Ordered Planner." *IJCAI-99*, pp. 968-973 (1999)
18. Nelson, G., Short, E.: *The Inform 7 Manual*. <http://inform7.com/learn/man/index.html>
19. Orkin, J.: *Three States and a Plan: The AI of FEAR*.
20. Rawls, J.: *Two Concepts of Rules*. *Philosophical Review* LXIV pp. 3 - 32 (1955)
21. Riedl, M., Bulitko, V.: *Interactive Narrative: An Intelligent Systems Approach*. *AI Magazine* vol. 34(1) (2013)
22. Roberts, D. L., and Isbell, C. L.: *A Survey and Qualitative Analysis of Recent Advances in Drama Management*. *International Transactions on Systems Science and Applications, Special Issue on Agent Based Systems for Human Learning*, pp.179 - 204 (2008)
23. Rosenblatt, J.: *Maximising Expected Utility for Behaviour Arbitration*. *Australian Joint Conference on Artificial Intelligence* (1996)
24. Sacks, H.: *Lectures on Conversation*. Kluwer (1989)
25. Sacks, H., Schlegoff, E., Jefferson, G.: *A Simplest Systematics for the Organization of Turn-Taking for Conversation*. *Language*, vol.50 (1974)
26. Salen, K., Zimmerman, E.: *Rules of Play*. MIT Press (2003)

27. Schank, R., Abelson, R.: Scripts, Plans, Goals and Understanding. Artificial Intelligence Series (1977)
28. Schank, R., Abelson, R.: Scripts, Plans, and Knowledge. IJCAI (1975)
29. Schatzki, T, R.: Social Practices: A Wittgensteinian Approach to Human Activity and the Social. Cambridge University Press (1996)
30. Short, E.: NPC Conversation Systems. IF Theory Reader. Transcript On Press. pp.331 - 359 (2011)
31. Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of Mercury: an efficient purely declarative logic programming language. Journal of Logic Programming, vol.29 (1996)
32. Strachey, C.: Fundamental Concepts in Programming Languages. Higher-Order and Symbolic Computation 13: 11-49 (2000)
33. Tomasello, M: Origins of Human Communication. MIT Press (2008)
34. Wardrip-Fruin, N.: Expressive Processing. MIT Press (2007)