Praxish: A Rational Reconstruction of a Logic-Based DSL for Modeling Social Practices

James Dameris*, Rosaura Hernandez Roman*, Max Kreminski
Santa Clara University
{jdameris, rhernandez2, mkreminski}@scu.edu

Abstract
The Versu framework is historically notable for its full-featuredness as a suite of tools for creating highly responsive interactive dramas. However, it has also been lost for nearly a decade, and a similarly approachable and flexible simulationist interactive narrative authoring framework has not yet emerged to take its place. We therefore aim to introduce an open-source rational reconstruction of the Versu framework, drawing on publicly available documentation of Versu’s design and implementation to assemble a successor system with similar architecture and capabilities. Here, we present the first component of this system: Praxish, a reconstruction of the low-level exclusion logic language atop which the rest of Versu’s functionality is based.

Introduction
Versu (Evans and Short 2014a) was an ambitious authoring framework intended to enable the creation of highly responsive interactive dramas. It was created by the interactive narrative systems equivalent of a supergroup, consisting of Emily Short (the narrative designer behind such notable works of highly responsive interactive fiction as Galatea); Richard Evans (the AI lead for The Sims 3); and Graham Nelson (the creator of the Inform series of domain-specific languages for interactive fiction authoring). Versu began development in 2010 and was acquired in 2012 by Linden Lab (the makers of Second Life), where development continued until the project was cancelled in 2014 (Reed 2021).

Several papers and blog posts documenting the Versu framework were released, and a handful of example games made with Versu (including the Regency-era murder mystery A Family Supper (Figure 1) and the late Roman Empire thriller Blood & Laurels) were briefly made available on iPad. Today, however, both Versu itself and its example games are effectively lost: the games cannot be played on modern versions of the iPad operating system, and the framework has never been released, with Linden Lab declining to sell it back to its original creators or otherwise allow its continued development (Reed 2021).

Unusually among interactive narrative frameworks, Versu was intended to support both a strongly simulationist approach to authoring (in which fine-grained computational models of social situations and character motivations were allowed to drive the story in potentially unexpected directions) and a detailed, high-quality textual rendering of the story that emerged from gameplay. Players of Versu games could be allowed to drop into the shoes of any character in the story; different non-player characters could be made to play different narrative roles as the player desired, with the character’s underlying personality driving them to make appropriate-seeming decisions in a wide variety of situations; and characters could ideally even be transplanted from one story to another, all without compromising prose quality. Additionally, Versu’s content language Prompter (Nelson 2014) aimed to make it possible for even inexperienced programmers to write simulationist stories. This unusual set of features, combined with the framework’s apparent success at meeting these goals in a handful of medium-scale text games, causes Versu to stand out as something of a high-water mark for interactive drama to this day.

We believe that both intelligent narrative technologies researchers and interactive narrative designers would benefit from the public availability of a system like Versu: the former from its uniqueness as a point in the design space of AI-supported frameworks, the latter from its utility as a suite of tools for crafting responsive stories. For this reason, we are working to release a strongly Versu-inspired authoring framework as a freely available and open-source piece of software. The reconstruction of the entire framework (which consists of several distinct and individually nontrivial software components) is no small task, and so we are currently taking a piecemeal approach: building a complete reconstruction of Versu one significant component at a time, with the most foundational components prioritized first.

In this paper, we apply the methodology of rational reconstruction (Tease et al. 2012) to the Praxis logic language atop which the rest of the Versu framework is based. Rational reconstruction involves thoughtfully reconstructing a piece of software from its technical description to gain insight into what makes it work, as well as into what about it can be improved. In addition, the output of the reconstruction process is a piece of software which can be used as a foundation for future research and development, both by us and by others—an especially important result in this context due to the unavailability of Versu itself.
A Family Supper

whom he prefers to the here, and so he has contrived to promote that son at
the hero’s expense. The other son is a clod, of mean intelligence and
unappealing manners, who will do no credit to his position.

Miss Bates signals for another glass of wine, and drinks it off in a single
swig.

She starts to feel woozy.

🏆 Completed achievement: drunken night

Frank: What an intriguing story! I do so admire the creative genius.

Miss Bates succumbs to a sudden, drink-induced melancholy and begins
talking about her sad the world is.

Lucy gives Miss Bates a small, reassuring smile.

Miss Bates lets rip a heartfelt bloom.

Mr Quinn looks decidedly a

Mrs Quinn (to Miss Bates)

Some people do not know how to regulate their drink?

Figure 1: A screenshot of the Versu example game A Family Supper, taken from (Evans and Short 2014b). The story so
far is laid out like a play script, and what each character is
currently thinking about can be viewed by interacting with
the character portraits at the bottom of the screen.

Our recreation of Praxis is called Praxish. Praxish is im-
plemented in dependency-free vanilla JavaScript and can
consequently be deployed in a wide variety of contexts,
including frontend-only mobile and desktop web browser
games, as well as in a standalone desktop executable or on
the server side via Node.js. Additionally, Praxish is open
source.1 Like Praxis, Praxish is a relatively low-level lan-
guage that will eventually serve as the compilation target for
a higher-level and more approachable content authoring lan-
guage akin to Versu’s Prompter; however, it also implements
a sufficiently full set of features that recognizable Versu-like
storyworlds can be created in only the preliminary version of
Praxish that exists today. (Figure 2 shows which Versu com-
ponents have been recreated in Praxish in some form.)

Research Process

Rational reconstruction begins with a thorough review of
existing documentation and materials related to the soft-
ware being reconstructed. The canonical write-up of Versu
is a 2014 article in the journal IEEE Transactions on
Computational Intelligence and AI in Games (Evans and Short
2014a), while the longest and most detailed explication of
Versu’s features can be found in an informally published ar-
ticle that was presented at Imperial College London (Evans
and Short 2014b). Beyond these articles, we also reviewed
the full breadth of documentation available on the Versu
website (Short 2014), including a slide deck that gives sub-
stantial additional detail on the Praxis language in particu-
lar (Evans 2014), and played Blood & Laurels on an iPad
(belonging to a friend of the Versu developers) that has not
been updated since the game was released.

In the process, we found that the most distinctive feature of
Versu relative to other narrative-oriented social simulation
frameworks (especially at the Praxis layer) is its focus on
the separation of content into multiple distinct and composable
social practices. A social practice provides actions that
the various participating agents can choose to perform and
tracks the state of a particular social situation, such as a din-
ner party, a flirtatious side conversation, a game of tic-tac-
toe, or a bartender dealing with orders from multiple differ-
ent customers—all of which could coexist in the same physi-
cal space, suggesting appropriate actions to the participating
agents in each practice without requiring every practice to be
aware of all the others. Practices in Versu were specifically
intended to be role-agnostic (Evans and Short 2014b)—i.e.,
to allow any participating character to be played either by an
autonomous agent or a human player—and to be potentially
reasurable across different interactive stories or even entire
storyworlds, such that some practices might be equally use-
ful for stories set in realistic, fantasy, and science fictional
settings. This grouping of potential character actions into
cleanly composable and reusable social practices is there-
fore a primary goal for our work on Praxish.

We then set about an iterative process of development in-
volving the gradual construction of two example storyworlds
and the framework features needed to realize these story-
worlds. After reconstructing the most foundational compo-
nent of the whole framework (the logic database), we split
into two teams and attempted to author social practices sep-
ately, developing two distinct intermediate JSON formats for
representing practice types. We discussed the differences
and similarities between these two formats, merged them
into a single format, wrote code to support all of the Praxis
language features used in these example practices, and then
repeated this two-step process of divergent content authoring
and convergent framework code authoring for several itera-
tions, eventually ending up with the set of framework fea-
tures discussed here. Notably, this resulted in two distinct
example storyworlds with conceptually similar but some-
what divergent practice definitions; we discuss these a bit
more in the “Case Studies” section below.

Overall, we believe that this process of repeated diver-
gence and reflective convergence resulted in a more robust
framework—one that accommodates a wider range of use
cases than if we had attempted to construct only a single
framework and example storyworld from day one.

Implementation

Praxish consists of three major components: the underlying
logic database, which handles the storage, querying, and up-
dating of storyworld state; the social practice layer, which
handles the specification and maintenance of the rules that
govern character behavior in a particular storyworld; and the
characters layer, which handles the specification and perfom-
ance of the characters present in a particular story.

---

1https://github.com/mkremins/praxish
Figure 2: A copy of the Versu system diagram from (Evans and Short 2014a), with a green overlay indicating which components we have reimplemented in Praxish.

Logic Database

The logic database is Praxish’s most fundamental component. It stores ground sentences of exclusion logic (Evans 2010)\(^2\) (i.e., sentences that contain no variables), permits the insertion and deletion of sentences, and can be queried via Prolog-style unification (Siekmann 1989) of the various sentences in the database with a provided open sentence (i.e., a sentence containing at least one variable).

Ultimately, the majority of higher-level Praxish functionality consists of running queries against the logic database to discover which actions can be performed by a particular agent and updating the database to reflect any changes associated with the action an agent has chosen to perform.

Data Format

As in Praxis, a Praxish sentence is any of:

\[ X ::= S | S.X | S!X \]

...where \( S \) denotes a symbol (either a capitalized Variable or a lower-case constant). Sentences are stored in the database as trees of constant symbols.

The dot connector \( . \) asserts that the tree on the left-hand side has the subtree on the right-hand side. For instance, the sentence \( \text{bedroom.furniture.bed} \) asserts that \( \text{bedroom} \) has one or more subtrees, including \( \text{bedroom.furniture} \); which itself has one or more subtrees, including \( \text{bedroom.furniture.bed} \). This sentence could coexist in the database with sentences like \( \text{bedroom.occupant.trip} \) and \( \text{bedroom.furniture.desk} \), allowing for the specification of a hierarchical world model.

The exclamation point connector \( ! \) is like the dot connector, but asserts that the left-hand side has only a single subtree, namely that specified by the right-hand side. Inserting first \( \text{bedroom.lockState!locked} \), followed by \( \text{bedroom.lockState!unlocked} \), into the database will result in only the second sentence being retained. However, all other subtrees of \( \text{bedroom} \) will remain unaltered: for instance, \( \text{bedroom.furniture} \) is unchanged.

Querying via Unification

The database can be queried by unification of its sentences against an open sentence that contains one or more capitalized variables. Suppose we query with the open sentence \( \text{Room.occupant.Person} \) (which contains two variables, Room and Person) a database that contains the following sentences:

\[
\begin{align*}
\text{bedroom.furniture.bed} \\
\text{bedroom.occupant.trip} \\
\text{bedroom.lockState!unlocked} \\
\text{patio.occupant.grace}
\end{align*}
\]

This yields two sets of variable bindings: \{Room: bedroom, Person: trip\} and \{Room: patio, Person: grace\}.

Updating the Database

The database accepts two basic update commands: insert and delete. Insertion generally adds a single sentence to the database, but may delete existing sentences representing removed subtrees if a subtree is overwritten by an inserted sentence that contains the exclamation point (\( ! \)) connector. For instance, if the database initially looks like the following:

\[
\begin{align*}
\text{jukebox.color!red} \\
\text{jukebox.playState.song!allStar} \\
\text{jukebox.playState.songPart!beginning} \\
\text{jukebox.playState.nextUp!closingTime}
\end{align*}
\]

...then inserting \( \text{jukebox.playState!broken} \) will cause the database to look like the following, replacing all information about the jukebox’s playState with the opaque symbol \( \text{broken} \) (without affecting any other stored information):

\[
\begin{align*}
\text{jukebox.color!red} \\
\text{jukebox.playState!broken}
\end{align*}
\]

\(^2\)Exclusion logic was later rebranded by Evans as eremic logic (Evans 2016). We use the older name here for consistency with the Versu documentation.
Deletion, meanwhile, can be used to remove either a single sentence or all sentences that share a specific prefix. For instance, if the database initially looks like the following:

```
practice.debate.ben.pattie
practice.debate.ben.pattie.whoseTurn!pattie
practice.debate.ben.pattie.topic!agents
practice.greet.trip.gonzalo
```

...then deleting `practice.debate.ben.pattie` will cause the database to look like the following:

```
practice.greet.trip.gonzalo
```

...essentially discarding all information related to the ongoing debate between Ben and Pattie, while preserving everything else.

Deletion in exclusion logic is specifically intended to allow for the easy implementation of lifetimes for bundled data. For instance, many social practices (including conversations, games, and so on) store temporary information that is needed to organize the behavior of participating agents while the practice (or a particular temporary state of the practice) remains active, but that can safely be cleaned up as soon as the practice concludes. The ability to easily clean up this transitory state with a single delete command is key to keeping the database manageable when many practices are active in parallel.

Ultimately, almost all state updates at higher layers of the Praxish stack are eventually implemented in terms of insert and delete commands issued to the underlying logic database.

### Social Practices

The next layer of Praxish provides affordances for authoring structured social practices, which guide the behavior of characters (both player and non-player) by providing these characters with a set of actions they can perform. From a technical perspective, practices are similar to finite state machines: each practice instance encapsulates some state that will be cleaned up when the practice instance is removed; actions offered by a particular practice are typically made available only when the practice is in a particular state; and taking an action offered by a particular practice will typically advance that practice’s state in some way.

Like Praxis, Praxish implements a constitutive view of social practices. Unlike the regulative view, in which social practices shape a person’s behavior by restricting what actions they can take, practices in the constitutive view shape a participant’s action possibilities in an additive way: the only meaningful actions that characters can take are those provided by the practices they are participating in (Evans 2016). In other words, the set of actions available to a character in Praxish is exactly the union of the sets of actions provided by each of the practices in which that character is a participant.

### Practice Format

Practices are defined as JSON objects with several required and several optional keys. At minimum, a practice must have:

- A string **id** that uniquely identifies this practice type
- A list of string **roles**, containing at least one entry, that contains the names of the practice’s uniquely identifying instance variables
- A list of **actions** containing at least one action definition, specifying the actions that this practice provides to its participants (format detailed further below)

Optionally, a practice may also have:

- A human-readable string **name** describing the practice, which may contain variable substitutions from the practice’s roles
- A list of static **data** sentences, which are inserted into the database under the `practiceData.PracticeID` prefix when the practice definition is registered
- A list of **init** commands to run when an instance of the practice is spawned, generally used to insert additional instance-level data into the database
- A list of zero or more **functions**, each of which is made available to the `call` command when the practice definition is registered

For an example of a basic practice definition that provides a single action, see Figure 3.

### Action Conditions

Every action definition consists of two major parts: a list of **conditions** that must be simultaneously fulfilled for the action to be available, and a list of **outcomes** that are executed when the action is performed by an agent. If the conditions for an action can be fulfilled in multiple different ways at the same time, then multiple different versions of the action are made available for the acting agent to choose between.

In considering whether a particular action definition should result in any concrete action possibilities for the currently acting character, the action instantiator first establishes bindings for all of the role variables from the practice that provides the action (for instance, Greeter and
Greeted in the greet practice outlined in Figure 3) and for the special variable Actor (which always indicates the ID of the currently acting character). It then attempts to unify the action’s conditions with one another subject to the initial variable bindings, establishing at least one set of internally consistent bindings for the initially unbound variables in the action’s conditions. For every set of bindings that can be established, a single instance of the action is made available.

In the current version of Praxish, several different condition types are allowed:

- Basic propositional conditions, which check that a certain sentence exists in the database
- Equality conditions, which check that two arguments (typically variables) are equal
- Negated conditions, which check that another condition does not hold

The original Praxis language implemented a slightly wider range of condition types, including disjunctive conditions, conditions that evaluate mathematical expressions such as greater-than, and conditions containing universal quantifiers. This range of conditions was inspired by those available within PDDL (McDermott 2000), and we eventually intend for our framework to support this full range of features, but mathematical expressions and universally quantified conditions are both left as (relatively straightforward) priorities for future work, since we found no immediate need for them in the example storyworlds that we created so far. Disjunctive conditions can already be implemented in terms of multiple different actions with similar but slightly disjoint conditions and identical player-visible names; this adds authoring burden, but because we intend for Praxish to be a compilation target for a higher-level and more approachable authoring language in the future (and because we found little need for disjunction in authoring our example content), we found that no new Praxish-level features are needed to support disjunction at this time.

**Action Outcomes** An action’s outcomes are simply a list of commands to execute in order when the action is performed. Commands include basic insert and delete commands that update the underlying logic database directly, as well as a call command that can be used to call a named function with zero or more parameters. As described in the following section, the call command can be used to implement conditional execution of action effects via functions that execute different commands depending on the values of their parameters.

The insert command can also be used to spawn new practice instances. Inserting a sentence into the database of the form practice.PracticeID.RoleBindings (for instance, practice.greet.trip.gonzalo in the case of a simple greeting practice) will cause a new instance of the named practice to be spawned with the provided role bindings, potentially running the corresponding practice’s instance initializer (which can perform additional database updates) in the process. In this way, actions in one practice can spawn subpractices that are used to furnish characters with responses or reactions to an action that was just performed: for instance, asking a question to a specific character in a group conversation practice might spawn a subpractice that provides the targeted character with several different ways of answering (or pointedly refusing to answer) the question that was just asked.

**Callable Functions** Functions behave similarly to actions, but can be invoked via the call command at any time (particularly within the effects of an action or the body of another function) to perform a sequence of parametrized commands. The following call command, for instance:

call setShouldRespond battler beatrice

...would invoke the setShouldRespond function (perhaps provided by some sort of group conversation practice) with the parameters battler and beatrice—i.e., the character who is now socially obliged to respond to another character’s direct question and the asker to whom a response is owed.

Functions are also capable of conditionally executing effects. This is because they contain not just a single list of conditions and a single list of effects, but an ordered list of cases, each of which has its own separate list of conditions that must be satisfied and effects that will be run if these conditions are met. Execution of a function proceeds down the list of cases, trying to find a case whose conditions all hold; as soon as an appropriate case is found, the body of the case is executed with the bindings established by the function’s parameters and the case’s conditions, and the search process stops. If no matching case is found, the function does not execute any code.

For example, a function intended to check the state of a tic-tac-toe board after every move and update the game’s overall status might have several cases:

- One that matches a victory by horizontal row, and marks the game over with the appropriate player as the victor if one is found
- One that matches a victory by vertical column
- One that matches a victory by diagonal (from the top left to the bottom right)
- One that matches a victory by diagonal (from the top right to the bottom left)
- One that matches a tied board, and marks the game over with no player as the winner

This function could then be called in the effects of every action that updates the board state to ensure that endgame conditions are always marked as soon as they are met.

These case-based functions (which were not present in Praxis) are inspired by, and behave similarly to, pattern-matching functions in functional programming languages like Haskell (Wadler 1987). They allow actions to execute conditional effects without the addition of any new conditional-execution features (beyond unification, which is already used to determine which actions are possible) to the core Praxish language.
Characters
In parallel to the social practice layer, an additional set of Praxish language features supports the definition and performance of non-player characters. This is the most preliminary of Praxish’s three existing components and the most likely to change in future versions of the language, but nevertheless already implements a significant portion of the character definition functionality provided by Praxis.

Character Format Characters in the present version of Praxish are defined as JSON objects with several allowed keys, all of which are optional except the id:

1. An id string uniquely identifying this character
2. A human-readable name string, substituted for the id in text presented to the player
3. A list of zero or more goals, each consisting of a list of conditions and an associated scalar utility score (described further below)
4. A boundToPractice string indicating the ID of a social practice to which the character is “bound”, if any (described further below)

Utility-Based Action Selection Non-player characters in Praxish behave by default as reactive utility-based agents that perform a single step of lookahead on the available actions (essentially simulating the immediate consequences of each available action) before deciding what action to perform. Agents choose the action with the highest available utility score to perform; if multiple actions are tied for highest score, one of these is chosen uniformly at random.

Utility scores are assigned to actions on the basis of character goals. A character goal consists of two components: a set of conditions that must all be simultaneously true for the goal to be satisfied, and a scalar utility score that the character “earns” when this goal is satisfied. For instance, a romantic comedy character kaguya who wants to impress her crush by ordering the same drink as them at the bar might have a goal represented as follows:

```json
{utility: 5,
conditions: [
  "practice.bar.B.patron.kaguya.order!Drink",
  "kaguya.ship.Crush.evaluation.crush",
  "Crush.preferences.favoriteDrink!Drink"]}
```

... and consequently would derive 5 utility from any action that results in her placing an order for the drink Drink at any arbitrary bar B where the Drink in question is also the favorite drink of a character she views as a Crush.

The overall utility of an action is determined by summing together the utility scores of every character goal that the action would cause to be satisfied. Consequently, characters will generally prefer to take actions that satisfy multiple of their goals at once if such actions are available.

Because goal satisfaction is evaluated via unification in the same way as action or function case conditions, a single goal can be satisfied in multiple different ways. When this is the case, the character earns additional utility for each instance of the goal that is satisfied.

Practice-Bound Agents Sometimes it is useful to include agents in a storyworld that are not traditional human-like social actors, but that are still capable of taking a narrow range of autonomous actions. Examples include both drama managers (which should take only metanarrative actions, perhaps intervening at certain key moments to advance the story to its next high-level plot beat or to adjust the motivations of a central character) and diegetic automata that should not participate in the vast majority of social practices (such as a jukebox, which should only take actions that involve playing through a song selected by another agent).

In Praxish, these kinds of behaviors are implemented by practice-bound agents: agents that are restricted to taking only actions provided by a specific named practice. A character definition may specify (via the boundToPractice key) the string ID of a practice to which this character should be bound, and the character will thereafter be unable to perform actions provided by any practices besides the specified one. This is a novel feature not available in the original Praxis language, and we discuss its implications in greater detail in the “Key Differences from Praxis” section below.

Player Characters Any agent may be substituted by a human player. Whenever an agent played by the player is allowed to act, the player is asked to directly select an action for the agent to perform from the complete list of actions available to that agent.

Key Differences from Praxis
Though Praxish began as a one-to-one reimplementaion of the original Praxis language from available documentation, we soon encountered several points at which it made sense to deviate from Praxis slightly in functionality. This section briefly discusses our key deviations from Praxis and the motivation behind each.

Conditionality via Pattern-Matching Functions
The original Praxis language provides an if command that can be used in an action’s outcomes (or in the body of a function) to check the value of a conditional, then execute a list of body commands if the conditional evaluates to true. However, because part of our vision for Praxish is that it will eventually serve as the compilation target for a higher-level and more approachable content language, we would prefer to minimize the number of distinct language features that are supported at the Praxish layer and to push less critical features up the stack to higher-level layers of our framework.

Consequently, we deviate from Praxis in providing only a single Praxish-level conditional primitive: the pattern-matching behavior exhibited in action preconditions, extended to the behavior of callable functions. A function that matches on its parameters and executes a different case depending on these parameters’ values can be used to implement arbitrarily complicated conditional effects without necessitating the addition of a separate if command, and a higher-level authoring tool or language can directly compile conditional effects in action definitions down to the invocation of an equivalent anonymous function if we decide that we want to implement something like if later on.
Practice-Bound Agents

A need for autonomous agents that do not participate fully in normal human social practices (including extradiegetic drama managers and diegetic automata) arose multiple times during our authoring of example content. Though these agents can be implemented by adding an extra condition to almost all actions (such that the vast majority of actions in a storyworld check whether the acting agent is a normal character rather than an automaton before allowing the action to proceed), in practice the need to manually insert these conditions substantially increases the authoring burden (Jones 2023) on practice authors. Manual insertion of actor type preconditions also limits portability of practices from one storyworld to the next, since there is no universal way to declare to the system that certain agents should not be treated as normal characters. Therefore, though it adds to the overall complexity of the core Praxish language, we still found the addition of a practice-bound agents feature to be worthwhile due to how much authoring burden it alleviates.

Practice-bound agents should also be excluded from selection as targets for normal social interactions, since they don’t represent typical embodied characters. For instance, characters usually shouldn’t try to flirt with the jukebox or pursue a rivalry with the drama manager. We initially believed that this would require us to explicitly exclude practice-bound agents from selection as targets of actions outside their bound practice: a nontrivial problem, given that the targets of an action are not explicitly tracked by a language-level feature in Praxish in the way that the currently acting agent is tracked by the actor variable. Fortunately, however, we discovered that the natural style of practice authoring (in which agents must generally take an explicit action to enter a practice before they can participate in it, or else must be inserted into a specific role within the practice when the practice is first spawned) tends to exclude practice-bound agents from appearing as potential targets for other practices’ actions as well, even without any additional language-level functionality or practice authoring conventions. Because a practice-bound agent will never initiate its own participation in other practices, it will generally not be subject to actions by other agents in other practices either.

In a Praxish-based game that permits players to select which character to play as, developers may allow players to select a practice-bound agent as the player character. This enables the creation of unusual social simulation-driven play experiences such as Juke Joint (Ryan et al. 2016), in which the player plays as a haunted jukebox with very limited affordances for interaction, and Cozy Mystery Construction Kit (Kreminski et al. 2019), in which one of the players is cast in the role of a drama manager who takes exclusively extradiegetic rather than in-character actions.

No Typechecking

Praxish was a statically typed language that enforced type annotations (provided by content authors) on the symbols that could be bound to certain roles, both in social practices and as arguments to callable functions. Praxish, on the other hand, makes no attempt to implement typechecking or type annotations, largely for the sake of language simplicity and ease of implementation. Though we believe that typechecking would eventually be nice to have as an assistance feature for content authors, we plan to implement typechecking later in the higher-level authoring tools rather than include it in the lower-level Praxish language, at least for now.

Case Studies

As discussed in the “Research Process” section above, we split into two content authoring teams during the research process and created two distinct but similar example storyworlds while developing Praxish. Both feature the same general setting (a bar that contains a jukebox and some space for playing tabletop games) and both feature some shared social practices, though they differ in the details.

Each storyworld also presents a slightly different user interface. One, created to test autonomous character behavior, simply simulates and prints out a sequence of autonomous character actions in the browser console to illustrate the nondeterministic and interleaved progression of the various social practices active in the world. The other, created to test player interaction, allows the player to create several named characters, then illustrates simple round-robin turn-taking in which the player is repeatedly allowed to choose the next action of each character.

Together, these example storyworlds (both available as part of this paper’s associated software artifact) demonstrate a wide variety of Praxish functionality:

- The greet practice demonstrates how practices provide actions only to their participants; how multiple instances of the same practice type can exist simultaneously between different sets of characters; and how a practice can remove itself from the database once it concludes.
- The tendBar practice demonstrates role asymmetry within practices (the bartender has access to a very different set of actions than the other characters), as well as the capacity for characters to independently (or) join and leave practices by walking up to and away from the bar. Two characters in the non-interactive demo also exhibit preferences for specific kinds of drinks, while the bartender has a preference for serving one of the customer characters over the other.
- In the non-interactive demo, the ticTacToe practice makes extensive use of a callable pattern-matching function to determine whether an endgame state has been reached every time a character makes a move.
- In the non-interactive demo, the jukebox, which plays through a requested song autonomously once another character makes a request, is a practice-bound agent: it can take actions, but only those provided by the jukebox practice to which it is bound, and it will never be selected as the target for normal social interactions.
- In the interactive demo, the fight practice demonstrates both how practices can be spawned by other practices, as well as how multiple different causes can all lead

3That said, we would love to play a game in which it makes sense to do either of these things.
to the same effect: there are several different ways for
characters to become angry at one another, but all of them
can result in a bar fight if the angered character decides
to escalate to a physical attack.

Both example storyworlds contain only a few hundred lines of practice and character definition JSON, split across four distinct practice types in the non-interactive demo and six in the interactive demo. We found that our pace of content authoring dramatically accelerated as framework features solidified, eventually enabling the development of multiple complex practices per day.

Related Work

We are aware of two prior non-Versu projects that made use of features from the Versu AI architecture: the now-defunct Character Engine by Spirit AI (a company whose founding team included Versu’s Emily Short) made use of Praxis-like social practices to structure agent behaviors, while the experimental AI-based game MKULTRA (Horswill 2018) was built around an exclusion logic database. Unfortunately, the Character Engine framework is now unavailable, and MKULTRA does not replicate features of the Versu architecture beyond the foundational database layer.

More broadly, there exist a number of logic-based domain-specific languages for specifying the behavior of characters in narrative-oriented social simulations. These all differ substantially from Praxish in terms of compositionality: where Praxish (like Praxis before it) aims to cleanly separate social behaviors into distinct social practices that can be reused from one story to the next, instantiated multiple times within a single story, and potentially even ported from one storyworld to another, other systems typically lump together all of the actions that are possible for characters to take into a single storyworld definition that does not consist of multiple readily separable practice definitions.

Ceptr (Martens 2015) is a logic language for interactive storytelling based on linear rather than exclusion logic, which provides a different set of solutions to the problems of modeling stateful social situations. Actions in Ceptr (called rules) may produce and consume resources that influence which actions are and are not available, and rules are grouped into stages that can be used to represent different high-level social contexts or successive states of a progressing social practice. However, the story may not be in multiple different stages at the same time, so stages cannot be used to cleanly separate social behaviors into composable or reusable practices as Versu practice definitions permit. Additionally, Ceptr does not provide any built-in mechanism for intelligent decision-making by non-player characters when multiple actions are possible. Ceptr does offer an approachable graphical authoring tool (Card and Martens 2019) that may serve as inspiration for our own future work on higher-level authoring tools for Praxish.

The Comme il Faut line of systems (McCoy et al. 2014), including successor systems Ensemble (Samuel et al. 2015) and Kismet (Summerville and Samuel 2020), are all based on a two-stage model of character decision-making. Characters in these systems first form high-level volitions toward one another (generally relationship status intents such as “start dating” or “become rivals”), with their propensity to pursue specific volitions calculated via storyworld-specific influence rules, and then take actions that are specified to advance these volitions. These volitions, however, are relatively limited: unlike in Praxish, characters cannot intend arbitrarily complicated sets of logical conditions, limiting the expressive flexibility of character goals. Nevertheless, Ensemble in particular is associated with a graphical authoring tool that may further influence our own future work on Praxish-based authoring tools.

Finally, the recently released social simulation game City of Gangsters (Zubek et al. 2021) is backed by a logic language called BotL that is largely used for reactive querying and state updates. Queries are periodically executed against the state of the world to discover whether certain logical patterns are met, in a fashion similar to story sifting (Kreminski, Wardrip-Fruin, and Mateas 2023), and relationship states for certain characters are updated if they are. Unlike the other systems discussed here (including Praxish), BotL is not responsible for handling the complete character action selection loop in CoG, and nothing like the definition of distinct, composable and reusable social practices is attempted. However, CoG does provide an interesting alternative take on how social norms could be encoded in a logic language for narrative-oriented simulation.

Conclusion and Future Work

We have presented Praxish: an open-source rational reconstruction of the Praxis logic language, which served as the foundation for the ambitious and now-defunct proprietary interactive narrative authoring framework Versu. Praxish can be used to define interactive storyworlds in terms of multiple distinct, composable, reusable and role-agnostic social practices that can operate in parallel to provide characters with a wide variety of action possibilities, as well as highly expressive character goals that drive reactive utility-based action selection by non-player characters.

In the future, we plan to continue our Versu reconstruction efforts by introducing a higher-level and more approachable authoring language that compiles down to Praxish, akin to Versu’s Prompter (Nelson 2014). This may be accompanied by other approachable authoring tools, including tools for creating NPCs, that go beyond Versu’s original affordances.

We also plan to create one or more larger storygames in Praxish as a case study. In particular, we aim to try reproducing a game that is similar in scale and storyworld to either A Family Supper, Blood & Laurels, or another Versu example game, to achieve and eventually demonstrate feature-parity with the original Versu framework. These efforts may be accompanied by one or more authoring studies (Hargood and Green 2023) intended to evaluate the usability of Praxish and our other authoring tools for content authors.

Acknowledgements

Thanks to Jacob Garbe for preserving, and allowing us to play, a working installation of the Versu example game Blood & Laurels.
References
McCoy, J.; Treanor, M.; Samuel, B.; Reed, A. A.; Mateas, M.; and Wardrip-Fruin, N. 2014. Social story worlds with Comme il Faut. IEEE Transactions on Computational Intelligence and AI in Games, 6(2): 97–112.