Centrifuge: A Visual Tool for Authoring Sifting Patterns for Character-Based Simulationist Story Worlds

Shi Johnson-Bey, Michael Mateas
University of California Santa Cruz
{shijbey, mmateas}@ucsc.edu

Abstract

Finding characters and events of interest in a large story world with possibly hundreds of characters can be challenging. Story sifting is the method of sorting through a story world's data to find the desired content. Until now, this process has mainly required someone to use text-based programming or query languages. However, this may prohibit those who prefer a more visual experience when story sifting. We present preliminary work on Centrifuge, a proof-of-concept graphical editing tool that enables users to query a character-based simulated story world for narratively intriguing groupings of characters, character relationships, events, and other entities. Our system presents users with various nodes representing entities in the simulated world or components of the underlying query language's syntax. Users may author queries by dropping nodes onto the canvas and dragging connections between nodes. Then Centrifuge is responsible for translating their configuration of nodes into a valid query that looks for matching patterns in a simulation data database. Query patterns are meant to be reusable and allow users to hierarchically build new patterns from preexisting ones.

Introduction

Character-based story worlds use interactions between virtual characters to produce engaging stories for players to enjoy. And a story world is considered simulationist if relies on the bottom-up interactions between autonomous non-player characters for content generation. On the spectrum of narrative generation systems, these story worlds depend heavily on character autonomy, with the goal being to have captivating series of events, or character relationships, emerge organically from the interactions of various subsystems (Riedl and Bulitko 2013). These emergent occurrences are called *emergent narratives*, and they feel like authentic events happening within the story world because they arise naturally in the system instead of via explicit scripted logic (Ryan 2018).

Relying purely on subsystem interactions to produce valuable narrative content can be risky. It assumes that whoever engages with the story world will discover the desired content in a sea of other simultaneous events. This assumption may not be a problem for simulations with relatively few characters, but for simulations that have on the order of tens to hundreds, the chances of a player being at the right place when an interesting scenario occurs are slim. However, there are solutions to this. One solution is to use narrative planners to ensure interesting, believable, and coherent events occur around the player (Riedl and Young 2010). Another solution and the one at the center of this project is story sifting. Story sifting is the process of finding engaging narratives, including narratively interesting characters and sequences of events, out of a sea of simulated history (Ryan 2018; Kreminski, Dickinson, and Wardrip-Fruin 2019). It is one of four proposed open design challenges for interactive emergent narrative (Ryan, Mateas, and Wardrip-Fruin 2015).

At the moment, there is very little in the literature directly addressing story sifting. Story sifting applications have ranged from manual live-coding (Samuel et al. 2016), to procedural pattern matching (Ryan 2018), to query language-based co-creativity tools (Kreminski, Dickinson, and Wardrip-Fruin 2019). While effective, each of these solutions required that the user have experience programming with a general-purpose language or a domain-specific language. In the case of *Bad News* (Samuel et al. 2016), the wizard player needed to have intimate knowledge of the underlying simulation code to navigate the space of simulated content. These projects effectively employed story sifting, but their approaches could be prohibitive to those with little or no programming background. Therefore, we wanted to make a visual authoring tool to enable users to craft sifting patterns without programming language syntax or knowledge of simulation-specific code structure.

Visual programming tools are a popular solution for teaching programming principles and providing users with a less technical alternative for coding. Languages like Scratch make programming and computational thinking more accessible to people who want to learn without opening the flood gates of syntax errors, and other programming-related woes (Fesakis and Serafeim 2009; Aivaloglou and Hermans 2016). Moreover, popular game engines such as Unity, Unreal Engine, and CryEngine provide visual programming workflows as an alternative to programming in C#, C++, or various computer graphics shader languages.

Our tool, Centrifuge, is a work-in-progress graphical sifting pattern editing tool explicitly tailored for character-

based simulationist story worlds. Our system presents users with various nodes and a drag-and-drop interface to construct arbitrarily complex sifting patterns without writing the sifting code manually. Behind the scenes, Centrifuge compiles patterns into a lower-level logic-based query language. So, users have the power to query a story world for narratively intriguing groupings of characters, character relationships, events, and other entities. Our goal is to create a tool that allows someone without programming experience to find narratively intriguing scenarios in a simulated storyworld with potentially hundreds of characters.

In this paper, we present our journey thus far with implementing our sifting pattern editor. We briefly overview the underlying query language, describe the tool, our pattern nodes, and how they connect to produce queries. Then we describe our case study implementing the editor to sift content from the *Talk of the Town* social simulation, the same simulation the underlies *Bad News* (Samuel et al. 2016). Finally, we discuss lessons learned, plans for the future, and potential impacts for a tool like this.

Related Work

Emergent Narrative (EN) has an extensive history. Aylett played an early role in defining EN (Aylett 1999) and formalizing it in various character-based systems (Aylett et al. 2005; 2006). ENs are one of the core pleasures of games that use social simulation as a core mechanic. Commercial examples of social simulations include *The Sims*, *Dwarf Fortress*, and *Blood & Laurels* (which used the *Versu* system (Evans and Short 2013)). Academic examples of social simulation systems and partnering media experiences include *Talk of the Town* (Ryan et al. 2015) and *Bad News* (Samuel et al. 2016), and *Comme il Faut* (McCoy et al. 2011) and *Prom Week* (McCoy et al. 2012).

Complementary to the implementation of emergent narrative systems have been efforts to sift for narrative patterns and characterize ENs. Existing examples of story sifting pattern tools include *Felt* (Kreminski, Dickinson, and Wardrip-Fruin 2019) and its application to the narrative co-authoring experience *Why Are We Like This?* (Kreminski et al. 2020). In the game *Bad News*, one player must surface simulation information to the actor by using the "wizard console" to perform manual story sifting via live Python coding. Similarly *Sheldon County* (Ryan 2018) used procedural python code to perform narrative pattern matching when constructing Podcasts from in-simulation events.

Our pattern editor also builds on past efforts toward making query languages and logic programming more accessible to general audiences. An early example of visual logic programming is (Ladret and Rueher 1991)'s work on creating VLP, a visual logic programming interface for Prolog. (Banyasad and Cox 2001; 2013) presented a visual logic programming language called *Lograph*. Their main goal was to contribute to the availability of tools for assisting design engineering efforts. Along these same lines, (Febbraro, Reale, and Ricca 2010) developed a graphical interface for writing answer-set programming (ASP) programs. Visual database query interfaces saw early work on end-user accessibility with Query-By-Example (QBE). QBE, created by Moshé M. Zloof, was the first visual database query language (PRESS 1975; Johnson 1999). More contemporary work includes (Vargas et al. 2019)'s implementation of *RDF Explorer* for querying graph-based databases. Visual programming languages are popular solutions for programming education. Our work builds on these approaches by presenting designers with a visual node-based interface to author narrative patterns compiled into a query language.

```
[:find ?person :in $ % :where [?person "sim/
    type" "person"]]
```

Listing 1: This basic query returns an array of entity identifiers for all entities with a "sim/type" attribute equal to "person."

Background: Query Language Overview

Our tool translates the node-based sifting patterns into a textual queries used with a database/query language called DataScript[1]. It is an open-source in-memory database solution and Datalog query engine, and has been used in other story sifting projects, namely *Felt* (Kreminski, Dickinson, and Wardrip-Fruin 2019) and *Why Are We Like This* (Kreminski et al. 2020). The following is a summary of DataScript's query syntax with examples. Our goal is to support as much of the DataScript syntax as necessary. As of this writing, Centrifuge supports most of the basic DataScript syntax needed to create moderately complex patterns. We cover mainly the parts of syntax relevant to our purposes.

DataScript databases are sets of immutable facts called datoms. Datoms are EAVT four-tuples containing the following:

- An entity ID (E)
- An attribute name (A)
- A value for the attribute (V)
- A transaction ID (T)

An entity in the database is the set of datoms that all refer to the same entity ID (E). Our case study stores simulated objects such as people, businesses, and events as DataScript entities. An example entity is given in table 1. Datom attribute values can be simple values (numbers, strings, boolean), JavaScript objects, or references to other entities in the database. Along with datoms, DataScript also uses optional schemas that allow users to specify the types of data associated with certain attributes. Users may specify the data types, cardinality (does the attribute map to a single or an array of values), and if the attribute may be used as alternative unique identifier for entities.

Once a database has entries, users may take advantage of its query language to find matching datoms and entities. Listing 1 shows a basic query containing the major sections that we use in this project. The first section is the *find-spec* where users define which variables should be returned by

[1] https://github.com/tonsky/datascript

E	A	V	T
56	“sim/type”	“person”	1234
56	“person/name”	“Korra”	1234
56	“person/age”	21	1234
56	“person/occupation”	:23	1234
23	“sim/type”	“occupation”	5678
23	“occupation/name”	“Avatar”	5678

Table 1: Example DataScript database containing two entities and a total of six datoms. The colon in “:23” distinguishes this value as a reference to another entity instead of it being a numerical value.

the query. The find spec starts with the keyword, ***:find*** and is followed by one or more variable names. Variable names have to start with the ‘*?*’ character and may be bound to entity IDs, attribute values, or the results of some query functions. The next major section of the query is the *inputs* section, indicated by the ***:in*** keyword. This section remains static for all of our queries, but the ‘$’ and ‘*%*’ are important built-in variables that map to the default database input and specified query rules, respectively. The ‘$’ is implicit in the query syntax and can mostly be ignored unless doing a more complicated operation using multiple database instances. The rule variable,‘*%*’, lets DataScript know we plan to use query rules. We discuss query rules in a later section as they are how we encapsulate sifting patterns to be hierarchical and reusable. Finally, there is the *where-clauses* portion, indicated by the ***:where*** keyword, that contains all of the conditions for filtering results.

DataScript *Where* Clauses

The most simple *where clause* is made up of three parts, an entity ID, attribute name, and attribute value. Listing 1 shows a where clause that bind the entity ID to the *?person* variable and tries to find a datom with an attribute name *“sim/type”* and value *“person”*. Each part of this type of where clause may be a constant or a variable. Variables are case sensitive and so *?person_0* is different from *?Person_0*. All the variables in *where clauses* implicitly join and unify to the same value. Where clauses may also take the form of range predicates ($<, >, \neq, \leq, \geq$), logical and/not/or clauses, logical not-join/or-join clauses, query functions, and query rules.

Range Predicates Range predicates check if given values pass a specified inequality/equality operation. Range predicates accept both variables and constants. Out tool makes heavy use of these for filtering. Listing 2 shows a query checking if a character is a given age.

Logical Clauses Logical clauses wrap other where clauses inside a logical and/not/or operation, changing how the internal clauses relate to the external where clauses. By default all clauses are implicitly and-joined, but *not* and *or* allow for additional complexity.

Logical join syntax is not fully supported by our tool, but it is used heavily in other projects such as *Felt* (Kreminski, Dickinson, and Wardrip-Fruin 2019). These operate identically to their non-join versions. However, one can specify which variables within the clause need to unify with the greater surrounding clause. See Listing 4

```
[:find ?person
 :in
    $ %
 :where
 [?person "person/age" ?age]
 [(< ?age 45)]
]
```

Listing 2: This query returns an array of entity identifiers for all entities with a “person/age” attribute less than 45.

```
(and [?person "person/gender" "female"]
     [?person "person/college_graduate" true
    ])
...

(not [?person "person/gender" "female"]
     [?person "person/college_graduate" true
    ])
...

(or  [?person "person/gender" "female"]
     [?person "person/college_graduate" true
    ])
...
```

Listing 3: Examples of and/not/or where clause syntax. Each keyword encapsulates other where clauses and modifies the output accordingly.

```
[:find ?person
 :in
    $ %
 :where
 [?person "sim/type" "person"]
 (not-join [?person]
    [?friend "person/friends" ?person]
    [?friend "person/tags" "villain"])
]
```

Listing 4: Query returning the an array of entity identifiers for all person entities who do not have a friend who is tagged as a villain.

Query Functions Query functions (See Listing 5) accept variables and constants as parameters, run a piece of JavaScript/ClojureScript code and bind the result to a variable. They have a similar syntax to predicate clauses, but they accept an additional parameter placed outside the parentheses.

```
[:find ?person
 :in
    $ %
 :where
 [?person "person/friend" ?friend]
```

```
 [(count ?friend) ?friend_count]
 [(> ?friend_count 10)]
]
```

Listing 5: Query returning an array of all people who have more than ten friends. Here *count* is a function that accepts a single parameter and returns the number of unique times *?friend* is matched. This query assumes that "person/friend" was specified as having a *cardinality/many* in the schema.

Query Rules Finally, there are query rules which are pre-authored groupings of *where clauses* passed into a DataScript query via the '%' character in the *input* section of the query. Rules are given names and may be defined multiple times. All instances of a rule must accept the same parameters, and during query-time, all versions must be satisfied for the result to pass as valid. This procedure is similar to other logic programming languages such as Prolog.

```
[(farmer_with_big_family [?person]
    [?person "person/occupation" ?occupation
    ]
    [?occupation "occupation/name" "Farmer"]
    [?person "person/family/kids" ?kid]
    [(count ?kid)] ?kid_count)
    [(> ?kid_count 4)]
]
```

Listing 6: Example query rule testing if the person is a farmer with a large family label

System Implementation

Our tool is a cross-platform desktop editor application where users author DataScript-based sifting patterns using interconnected nodes. Users can create new patterns, save patterns, and open/edit multiple patterns at once. The core feature of the graphical user interface (GUI) is the node canvas, where users add/remove nodes from their patterns, rearrange nodes, and drag connections between nodes. When users want to make more complicated patterns, they may organize them into a workspace directory and open the directory within Centrifuge's workspace explorer. Centrifuge then compiles the existing pattern files and uses them to create custom query rule nodes. These query rule nodes are designed to add reusability to the editing experience and simplify the visual space by abstracting logic behind a single node.

Centrifuge's purpose is to assist users in constructing sifting patterns by abstracting away the text-based syntax for a graphical representation. The GUI draws inspiration from Visual Studio Code, Unity's Shader Graph, and Unreal Engine's Blueprints. A screenshot of the GUI can be seen in Figure 1.

Our node design is heavily influenced by DataScript's EAVT data model and query syntax. Each node has a set of input/output ports where links connect specific bits of information between nodes. Ports have associated types although there is no type checking currently implemented in our system. We split nodes into the following types:

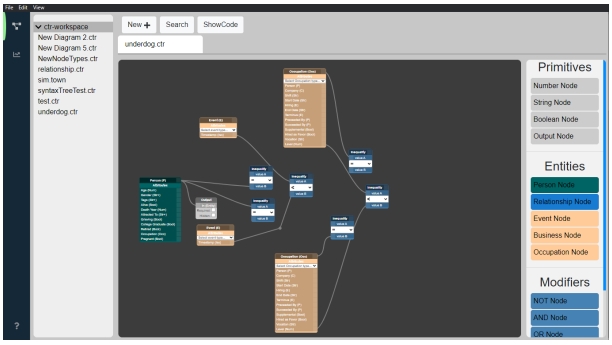


Figure 1: Screenshot of the Centrifuge editor. Users drag and drop node types from the right panel to the grey canvas in the middle. The pattern displayed is a version of the "underdog" or "rags-to-riches" story pattern where a person goes from a lower socioeconomic status to a significantly higher one.

Node Types

Constant Nodes Our node design is heavily influenced by DataScript's EAVT data model and query syntax. Each node has a set of input/output ports where links connect specific bits of information between nodes. Ports have associated types. However, there is no type-checking currently implemented in our system. We split nodes into the following types:

Entity Nodes Entity nodes represent a collection of related datoms in the database. They can represent objects in the simulation, such as people, places, and events. These nodes are the base for creating sifting patterns as they supply the attribute values and are the objects that a query seeks to find. Our tool creates database queries that return entity IDs that can be used to extract all associated data at a future time. Entity nodes do not accept incoming connections; they only provide output connections to other nodes. Aside from a label, they have a collection of output ports, and each port is associated with a different attribute of that entity type. Each port has an associated type based on the data that is stored. Right now, it is up to the person configuring the nodes to explicitly define what type of port is associated with each attribute. Ports can be mapped to one or more constant values or entity identifier references. Entity ports are unique because they supply the connecting node with the variable name bound to that attribute's value.

Variable names in the query are based on the names assigned to the entity nodes. Entity nodes' names are autogenerated but can be modified by the user to improve clarity. Derivative variable names such as those associated with an attribute value are given a generated name which includes the entity name as a prefix and the attribute name as a suffix. For instance, a variable mapped to a person entity's age will have the following variable binding:

```
[?person_0 "person/age" ?person_0_age]
```

In this case, "age" is the attribute name. So, we append it to the end of the variable name *?person_0* with an underscore. All variable names in DataScript start with a question mark

(?), and we add this mark behind the scenes if not already present.

Range Predicate (Inequality) Nodes Range predicate nodes translate to DataScript's range predicate syntax and perform inequality/equality checks on two given inputs. Inputs may be constant values or entity attributes. Range predicate nodes are one of the core methods of filtering the results of a query. They have a single output port for chaining their resulting output inside other nodes — for example, chaining a series of range predicates and logical nodes.

Functional Nodes Functional nodes represent pieces of custom JavaScript code that handle some cases that the DataScript API does not support. For example, cardinality counting is not supported in the JavaScript version, and so we wrote a *count* node that provides that functionality. In our experience, functional nodes require hand-authoring as they usually serve a particular purpose and will probably implement functionality not provided by the query language. Since they are made as needed, there is no set number of input ports, but there should be only one output port that binds a variable name for the function's output.

Logical Nodes Logical nodes include NOT, AND, and OR nodes. These nodes wrap the output of nodes passed to them in their respective clauses. For example, the NOT node will produce code where clauses from input connections are wrapped in a "(not ...)."

There are also logical-join nodes that provide functionality for *not-join* and *or-join* syntax. These are two powerful operations that allow query designers to check for all instances of outputs where a set of clauses in not the case, any set of clauses could be the case. Logical join nodes encapsulate variables and let the user decide which ones should unify with the greater query. In a later section, we present an example using not-join with a sequence of events.

Query Rule Nodes Query rule nodes represent authored patterns. They expose inputs specified by the sifting pattern's variable nodes and translate into DataScript clauses of the form:

```
(rule_name ?var1 ?var2)
```

The user defines the rule's name in the GUI, and the variables used in the rule header are specified using variable nodes, which are explained next. Query rule nodes allow for pattern reuse, and users can build more complex patterns using simpler patterns as building blocks. We are working on a system to generate these using the intermediate JSON format to which sifting patterns are compiled.

Variable Nodes Finally, variable nodes are a special type of node that defines how the pattern interacts at compile time. Patterns may be used as query rules in other patterns, or they may be the primary entry point for a query. The variable nodes' job is to determine which variables need to (A) be returned by the query or (B) need to unify with the outer context if it is used as a rule. Variable nodes only accept a single input from an Entity node. Not all entity nodes need to be attached to a variable node. Only those that are intended to be output for the rule/query.

Pattern Compilation

A completed pattern is essentially a dependency graph between components of information needed to form a query. The directed nature of links in our sifting patterns results in a directed acyclic graph with information flowing from the left to the right. The compilation starts with running a topological sort to determine a processing order for nodes. Then we convert each node to a text string using an associated template. Nodes chain their string outputs to produce any nested query structure. So when traversing the tree to produce the final query, only the leaf nodes' output is included in the final output. The compilation result is an intermediate JSON format that allows the tool to convert patterns to either full-queries or query rules referenced by a full query.

Case Study: Talk of the Town

We wanted to explore the process of tailoring Centrifuge to an existing character-based social simulation, so we selected *Talk of the Town* as a case study. *Talk of the Town* is a character-based simulationist story generator that uses social simulation to simulate a virtual town and its townspeople over a 100+ year period(Ryan 2018; Ryan and Mateas 2019). Characters have families, homes, and relationships. They also follow routines based on their occupations. We chose *Talk of the Town* for our case study for the following reasons. First, it is a sophisticated example of a character-based, simulated story world. Second, the townspeople are constantly creating a variety of life events. Third, the character interactions give rise to various emergent scenarios — for example, rivalries, asymmetric friendships, and love triangles. Talk of the Town has shown that its generated content is interesting enough to support the award-winning AI-based experience *Bad News*. In this section, describe our experience designing nodes to fit DataScript's syntax and *Talk of the Town's* data models. This case study served both as an experience in tailored configuration and iterative design.

We had to create a custom pipeline that transferred simulation data from a *Talk of the Town* instance to a DataScript database instance. *Talk of the Town* was not configured for its simulation data to be exported or hooked into for external analysis. So, we had to write a custom code module that saved the state of the simulation out to file. We then loaded the relevant data (events, people, businesses, relationships, occupations) into a DataScript database instance. We also defined a schema for the data, specifying which attributes were mapped to multiple values and which attributes referenced other entities in the database. DataScript's data model forced us to flatten the way we represented simulation entities. Nested objects had to be stored at the same level, and entity references replaced where they once were in the data hierarchy

While most node types work regardless of the simulation, we needed custom entity nodes that reflected *Talk of the Town's* world model. Entities could be events, people, businesses, relationships, occupations. Each of those entity types has different attributes. Therefore was had to hand-code the port configuration, types, and labels for each entity type. Some entities, such as events, have dynamic attribute

ports based on the type of event. We hand-coded all of the attribute ports and their associated labels (see Figure 2).

Talk of the Town-specific Nodes

Person Nodes Person nodes represent individual townspeople. Users have access to all of a person's attributes. However, some attributes like familial and social connections are only available when using Person nodes in conjunction with Social Connection nodes.

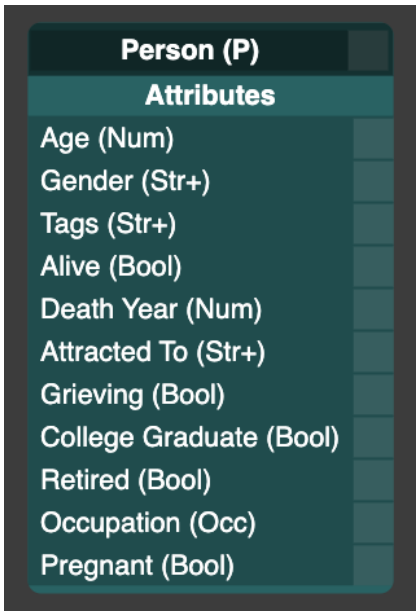


Figure 2: Person node in Centrifuge. Each output port on the right side of the node corresponds to an attribute stored in the database that users can use for filtering results.

Business Nodes Businesses are one of the core objects in *Talk of the Town*. They drive population growth and decay and are responsible for tragic story instances such as people being evicted from their homes to build a new business. When using business nodes, designers have the option to filter by a business type. Choosing to do so may also modify the attribute ports available on the business node. Some businesses like Law Firms have unique attributes that we dynamically hide/show based on the selected business type.

Occupation Nodes Occupation nodes are entities that represent job roles that townspeople have held throughout their lives. These nodes are essential for calculating a person's trajectory in their life. Occupations determine a person's routine, social circle, and the growth/decay of their relationships. So, they are a helpful tool when inspecting the social aspects of characters' lives. Occupation nodes also have a type selector that dynamically changes some of the ports to give unique options based on the selected type.

Event Nodes Event nodes represent recorded events that have transpired in the simulation. They have timestamps and associated characters that participated in the event. Event nodes have an event type selection drop-down that dynamically chances the output ports to match the information associated with this entity. Users can build queries that look for sequences of events in the simulation and retrieve the characters associated with those events.

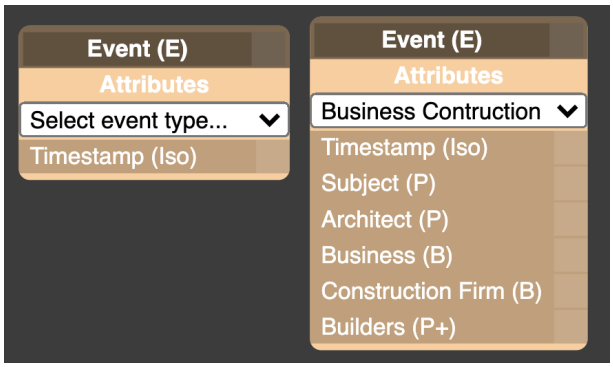


Figure 3: General event node (left) and an event node with its type set to Business Construction, revealing specialized attribute ports.

Relationship Nodes Relationships between characters in *Talk of the Town* are entities that gauge the amount of romantic and platonic affinity one character has for another. Relationships in *Talk of the Town* are directed. Therefore, one relationship entity only represents one character's feelings about another and does not represent the shared reciprocal feelings. For one to sift for two characters' feelings toward each other requires two separate relationship nodes. Relationship nodes are separate from Social Connection nodes because they provide quantitative values about relationships, not just the semantic relationship.

Social Connection Nodes Since *Talk of the Town* is a social simulation, we need to have a node that represents social connections between characters. Characters can be related to each other in a multitude of ways. They can be friends, enemies, family, coworkers, or some other social association between characters. Social Connection nodes are one of our *special nodes* and are helpful to designers when specifying social or familial relationships between People nodes in their pattern. For example, they can check if two people are friends, family, enemies, married to one another, or some other social arrangement. This node does not map to a specific entity but was made specifically for working with *Talk of the Town* as it simplifies pattern representation.

Authoring Patterns

We tried authoring patterns based on those used with *Felt*(Kreminski, Dickinson, and Wardrip-Fruin 2019). In figure 5 we demonstrate an example sifting pattern that looks for a series of layoff events where the same character gets laid-off from their job. The particular part is that a retirement event must have occurred between these events. Ideally, this sifting pattern would find a character who is "down on their luck" and constantly gets fired from their various jobs.

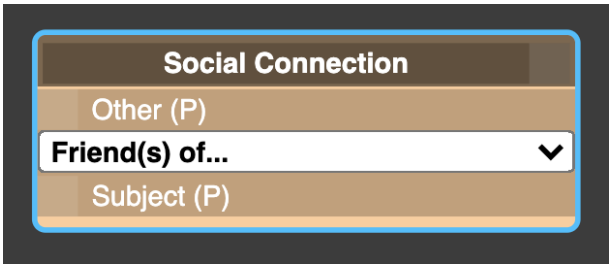


Figure 4: Social Connection node with the "friends" connection selected. Users can click the drop-down to select from many more types of character relationships.

This query is made possible thanks to the not-join node, which finds instances of the layoff events and characters that do not have an intermediate retirement event. If this same pattern were implemented in python code, it would require nesting *for-loops* over the sets of events and characters. The final query, while not as succinct as one written by hand, is still valid DataScript and could be fed into a running instance containing *Talk of the Town* simulation data.

Discussion

Creating a custom pipeline for a specific story world requires much hands-on coding with the underlying node-diagram library. Our experience thus far with adapting the sifting pattern nodes to *Talk of the Town* has been tedious as we had to write custom code to make the simulation data work with the database and make the pattern nodes fit the simulation. As we explained prior, importing *Talk of the Town's* data into the database required a custom pipeline that involved a database schema and a procedure for replacing nested objects with DataScript entity references. The process is time-consuming and is most likely prohibitive to anyone in the industry planning to use this particular approach. However, experiences/games that are built with DataScript in mind, such as *Why Are We Like This?* (Kreminski et al. 2020), may have a more effortless experience. Future iterations of this project could benefit from a domain-specific configuration language that simplifies the node authoring process. Perhaps something like JSON schema could facilitate both data conversion and node generation.

The concept of having a more designer-friendly tool that houses a repository of sifting patterns is exciting. We imagine using sifting patterns to analyze the frequency of desired pieces of content appearing in their story worlds. The sifting patterns would help a world designer characterize the generative space of emergent narratives in their simulation. Expressive range analysis (ERA) is vital for evaluating procedural content generation (PCG) systems (Smith and Whitehead 2010). ERA requires that designers define metrics to evaluate the output of a generative system. For narrative simulations that rely on emergent narratives, ERA is essential because, during any given play-through, there is no guarantee that there will be appropriately intriguing content for the end-user.

Moreover, there is no guarantee that the content will satisfy the designer's aesthetic goals. However, with proper analysis, designers may better understand the potential distribution of content, better understand the inner workings of the generator, and adjust the configuration of the simulation to fit their goals. However, before we can reach that state, designers need better tools to query the narrative output of their simulators. This tool would help in that effort.

Conclusion and Future Work

In this paper, We describe our preliminary work on developing a sifting pattern editing tool. We also described our experience adapting it for *Talk of the Town*. The project is still in progress, but it does provide an example of what a visual sifting pattern language could look like. Users can author patterns and see their query-language equivalent without needing to know any DataScript. Not all of the features that we planned are currently available as of this writing. Specifically, pattern re-use is an important feature that we believe will help users write more complex patterns without being overwhelmed by the multitude of nodes and links. Also, reusable patterns will add a level of modularity that currently does not exist. In their concluding remarks, Kreminski (Kreminski, Dickinson, and Wardrip-Fruin 2019) mentions the potential for sifting pattern authors to become overwhelmed with larger patterns. Perhaps the visual hierarchy created by our tool could help prevent that.

For now, our tool only works with a specific version of Talk of the Town. Future work could investigate strategies for specifying nodes in a configuration format like YAML or JSON and auto-generate nodes used for authoring. The current workflow involves creating all the associated code for defining a new node type. This procedure is not friendly to expanding this framework for other simulations. The experience with authoring nodes for specific simulations could benefit from a higher-level configuration language that removes the burden of extending the underlying APIs.

In the future, we would like to run further investigations with this tool and perhaps get more qualitative feedback from individuals who are asked to complete some authoring task using it. Observing how they author patterns for structured tasks and free-form exploration would give us a better sense of the expressivity of our tool and potential areas of improvement. Suppose user tests demonstrate that Centrifuge is more helpful than authoring sifting patterns in code. In that case, more time should be invested in developer experience features such as better type checking between ports, error reporting, and query result presentation.

Currently, there is no interface for running the queries on a database and visualizing the results. The presented experience feels incomplete without a way for users to inspect the entities that their sifting patterns find. Future work will investigate intuitive designs for presenting the results of queries produced by sifting patterns.

References

Aivaloglou, E., and Hermans, F. 2016. How kids code and how we know: An exploratory study on the scratch reposi-

tory. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, 53–61.

Aylett, R. S.; Louchart, S.; Dias, J.; Paiva, A.; and Vala, M. 2005. Fearnot!–an experiment in emergent narrative. In *International Workshop on Intelligent Virtual Agents*, 305–316. Springer.

Aylett, R.; Louchart, S.; Dias, J.; Paiva, A.; Vala, M.; Woods, S.; and Hall, L. 2006. Unscripted narrative for affectively driven characters. *IEEE Computer Graphics and Applications* 26(3):42–52.

Aylett, R. 1999. Narrative in virtual environments-towards emergent narrative. In *Proceedings of the AAAI fall symposium on narrative intelligence*, 83–86.

Banyasad, O., and Cox, P. T. 2001. Implementing lograph. Technical report, Report CS-2001-05, Faculty of Computer Science, Dalhousie University.

Banyasad, O., and Cox, P. T. 2013. Design and implementation of an editor/interpreter for a visual logic programming language. *International Journal of Software Engineering and Knowledge Engineering* 23(06):801–838.

Evans, R., and Short, E. 2013. Versu—a simulationist storytelling system. *IEEE Transactions on Computational Intelligence and AI in Games* 6(2):113–130.

Febbraro, O.; Reale, K.; and Ricca, F. 2010. A visual interface for drawing asp programs. In *CILC*.

Fesakis, G., and Serafeim, K. 2009. Influence of the familiarization with" scratch" on future teachers' opinions and attitudes about programming and ict in education. *Acm SIGCSE Bulletin* 41(3):258–262.

Johnson, S. 1999. Query-by-example (qbe). *Database Management Systems. New York, NY: McGraw-Hill Publisher*.

Kreminski, M.; Dickinson, M.; Mateas, M.; and Wardrip-Fruin, N. 2020. Why are we like this?: The ai architecture of a co-creative storytelling game. In *International Conference on the Foundations of Digital Games*, 1–4.

Kreminski, M.; Dickinson, M.; and Wardrip-Fruin, N. 2019. Felt: a simple story sifter. In *International Conference on Interactive Digital Storytelling*, 267–281. Springer.

Ladret, D., and Rueher, M. 1991. Vlp: a visual logic programming language. *Journal of Visual Languages & Computing* 2(2):163–188.

McCoy, J.; Treanor, M.; Samuel, B.; Wardrip-Fruin, N.; and Mateas, M. 2011. Comme il faut: A system for authoring playable social models. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 6.

McCoy, J.; Treanor, M.; Samuel, B.; Reed, A. A.; Wardrip-Fruin, N.; and Mateas, M. 2012. Prom week. In *Proceedings of the International Conference on the Foundations of Digital Games*, 235–237.

PRESS, A. 1975. Conference or the american federation of information processing societies, inc.

Riedl, M. O., and Bulitko, V. 2013. Interactive narrative: An intelligent systems approach. *Ai Magazine* 34(1):67–67.

Riedl, M. O., and Young, R. M. 2010. Narrative planning: Balancing plot and character. *Journal of Artificial Intelligence Research* 39:217–268.

Ryan, J., and Mateas, M. 2019. Simulating character knowledge phenomena in talk of the town. In *Game AI Pro 360*. CRC Press. 135–150.

Ryan, J.; Summerville, A.; Mateas, M.; and Wardrip-Fruin, N. 2015. Toward characters who observe, tell, misremember, and lie. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 11.

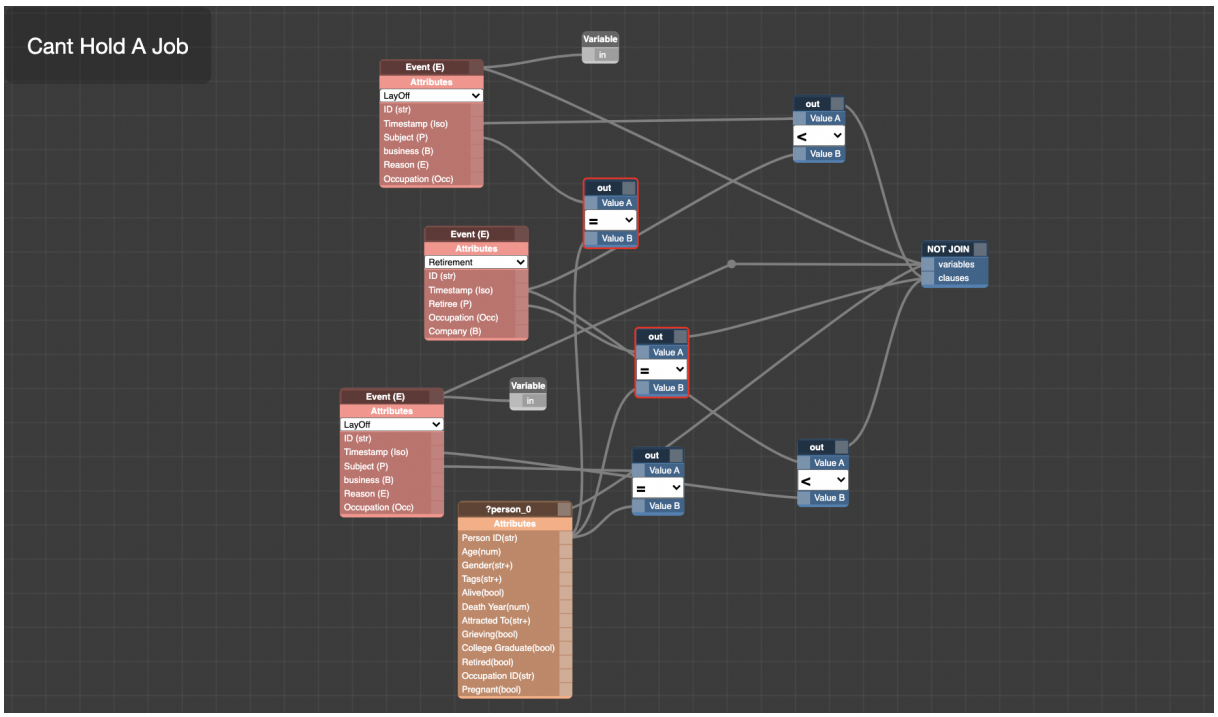
Ryan, J. O.; Mateas, M.; and Wardrip-Fruin, N. 2015. Open design challenges for interactive emergent narrative. In *International Conference on Interactive Digital Storytelling*, 14–26. Springer.

Ryan, J. 2018. *Curating simulated storyworlds*. Ph.D. Dissertation, UC Santa Cruz.

Samuel, B.; Ryan, J.; Summerville, A. J.; Mateas, M.; and Wardrip-Fruin, N. 2016. Bad news: An experiment in computationally assisted performance. In *International Conference on Interactive Digital Storytelling*, 108–120. Springer.

Smith, G., and Whitehead, J. 2010. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 1–7.

Vargas, H.; Buil-Aranda, C.; Hogan, A.; and López, C. 2019. Rdf explorer: A visual sparql query builder. In *International Semantic Web Conference*, 647–663. Springer.




```
[
:find ?event0 ?event1
:in $ %
:where
[?event_0 "sim/type" "event"]
[?event_0 "event/subject" ?event_0_subject]
[?event_1 "sim/type" "event"]
[?event_1 "event/subject" ?event_1_subject]
[?person_0 "sim/type" "person"]
[?person_0 "person/id" ?person_0_id]
[(= ?event_0_subject ?person_0_id)]
[(= ?event_1_subject ?person_0_id)]
(not-join [?person_0 ?event_0 ?event_1]
    [?event_2 "sim/type" "event"]
    [?event_2 "event/subject" ?event_2_subject]
    [?person_0 "person/id" ?person_0_id]
    [(= ?event_2_subject ?person_0_id)]
    [?event_0 "event/timestamp" ?event_0_timestamp]
    [?event_1 "event/timestamp" ?event_1_timestamp]
    [?event_2 "event/timestamp" ?event_2_timestamp]
    [(< ?event_0_timestamp ?event_2_timestamp)]
    [(< ?event_2_timestamp ?event_1_timestamp)])
]
```

Figure 5: An example query that looks for a sequence of Layoff events where the same character is laid off from a job without any retirement events happening in that character's life between the two layoff events. (Top) node representation. (bottom) The resulting query syntax.