Villanelle: Towards Authorable Autonomous Characters in Interactive Narrative

Chris Martens* martens@csc.ncsu.edu
Owais Iqbal* omiqbal@ncsu.edu
Sasha Azad* sasha.azad@ncsu.edu
Maddie Ingling* mainglin@ncsu.edu
Anthony Mosolf* amosolf@ncsu.edu
Emma McCamey† mccameyec@vcu.edu
Johanna Timmer‡ timmerjanna@gmail.com

*North Carolina State University †Virginia Commonwealth University ‡ Gwinnett College

Abstract

Innovations in intelligent narrative technologies have continued to accelerate, but elude adoption by a broad and diverse audience of storytellers. Authoring accessibility for these technologies is an open challenge, largely due to tensions between story adaptability and author control, explainability, and predictability. We describe ongoing efforts on the *Villanelle* project, an approach to autonomous character authoring that integrates scripting with generativity, using a logic-based foundation that unifies these approaches with reasoning principles that carry through the authoring process. We outline our proposal for thinking about authoring *languages* rather than tools, describe our proposal for such a language, and discuss current implementation progess, including a block-based authoring tool and two ongoing game projects using the framework. We discuss key design challenges and a future roadmap for accessible autonomous character authorship.

Introduction

From hobbyist writers to AAA game developers, storyellers using interactive media increasingly yearn for the ability to make their story-worlds feel "alive:" to create an open-ended narrative possibility space, to employ systems that shift and recombine narrative elements in response to player decisions, and to allow narrative to emerge organically while preserving authorial decisions about key beats and pacing. The intelligent narrative research community has largely been focused on advances in procedural narrative that can bring these effects about: the planning-based systems that regenerate narrative arcs based on player decisions [Ayl99, RY10, PCC10], the social practice systems encoded in CiF and Versu [MTS$^+$10, ES14a], the combinatorial content selection algorithms of Ice-Bound [RGWFM14], and many other examples demonstrate the impressive range of technology available for narrative generation.

However, *authoring accessibility* for these systems remains an open challenge with a deficit of attention. With the introduction of authoring tools like Twine, the appeal of telling stories with interactive media is broader and more diverse than ever, but the authorability of generative narratives is not keeping pace with this demand, generally privileging technical sophistication over expressivity and accessibility [Sal16]. As a community, we have an opportunity to broaden and diversify our field with the voices of more women and LGBTQ+ storytellers, who disproportionately don't have a traditional programming background—but to do so, we need to shift our focus to the problem of authoring.

In this paper, we examine interactive narrative authoring challenges specifically around **autonomous characters**, i.e. story characters (and other aspects of the narrative environment external to the player) that exhibit observable behavior

In: H. Wu, M. Si, A. Jhala (eds.): Proceedings of the Joint Workshop on Intelligent Narrative Technologies and Workshop on Intelligent Cinematography and Editing, Edmonton, Canada, 11-2018, published at http://ceur-ws.org

without player intervention. We outline these challenges and present approaches to solving them, discuss the shortcomings of current approaches, and propose a new framework called *Villanelle* in development.

Villanelle is being developed through a practice-driven process, using small examples and medium-scale game projects to drive the expressiveness needs of the foundational theory and tool interface. This approach has exposed a wide gap between the affordances of popular narrative modeling techniques and our design goals as interactive narrative authors. Our proposal addresses this gap with a hybrid scripting/planning model implemented as a minimal, easy-to-learn syntax. This approach combines insights from narrative planning techniques with insights from scripting languages, particularly behavior trees (BTs), a common game AI authoring tool in commercial games [Isl05]. Whereas most planning-based systems integrate a representation formalism with a search procedure, we separate these two ideas and start with "plan notation" as a baseline for an author's expressive palette, allowing them to sequentially compose operators in a planning domain. Observing the frequent need to author contingencies for nondeterministic outcomes of action, we then integrate mechanisms for branching and looping (inspired by behavior trees).

We then demonstrate how this language lays a foundation on which to build accessible tooling such as a "block-based" behavior editing tool. We have found that this formalism not only provides a minimal, elegant core calculus for describing agent behaviors but also provides a mechanism for authoring player-facing content, including two different user interface approaches: "CYOA-style" hand-authored finite choices and "parser-style" combinatorial actions generated by the planning domain. Finally, we outline a proposal for re-incorporating the "search" part of planning as a procedure that authors may invoke at the leaves of their behavior trees and discuss challenges with intertwining generated plans with believable reactivity.

Villanelle has not been formally evaluated by its target audience yet, and we are actively seeking input from the IN research community on how its design can best facilitate the needs of diverse research on autonomous virtual characters. Our intention with this paper is to bring authoring tool design challenges to the fore of discussion within the intelligent narrative community.

Related Work

Plan-based approaches to narrative generation [You99, CCM02] form one theoretical anchor for this work. However, planning only tells one half of the story for *interactive* narrative; choices must still be made about the execution of plans, particularly when the environment is evolving. Efforts to integrate narrative planning into interactive narrative [RY10, PCC10] traditionally take the approach of generating a plan before play begins, then mediating the player's input one step at a time: if the player takes an action that invalidates the preconditions of future steps of the plan, a *replan* mechanism is invoked. However, authors are given few affordances to control the replanning behavior, while in interactive stories, disruption of default behaviors is usually the common case. In our work, by contrast, we adopt the *representation formalism* of plans while devising a new approach to authoring and execution in which "plan failure" may be explicitly anticipated and invoke pre-authored decision making.

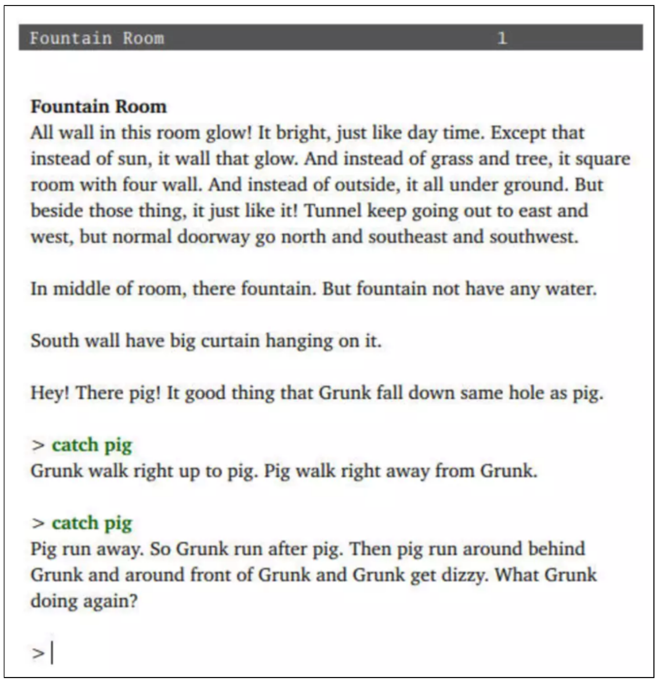
Our work is also anchored in so-called "reactive" or scripting-based approaches to interactive story authoring. The classic interactive narrative work *Facade* [MS03b] was authored using ABL [MS02], a behavior scripting language with facilities for sequencing behaviors, conditioning behaviors on properties of the narrative state, and executing sub-behaviors in parallel (e.g. for modeling a character gesturing while speaking a sentence). ScriptEase [COM$^+$07] is another, more recent approach to character scripting. A related approach is *behavior trees*, commonly used in AAA games and occasionally in narrative work [KFZ$^+$15]. While we adopt similar constructs (sequencing, selection, conditioning, etc.) to these systems, none have seen widespread public adoption that would serve as a measure of accessibility, and the language we present aims to improve the transparency and simplicity of such a language while retaining and building upon their expressive power. We also plan to incorporate the generative capacity of planning in such a way that it can be invokes by authors as a black-box process.

A number of proposals have been made for interactive narrative authoring tools, such as Scenejo [SWM06], Comme il Faut [MTS$^+$10], Prism [CKM$^+$08], Narratoria [VV08], and Versu [ES14b]. These tools have a wide range of focal points; for instance, Scenejo integrates the AiML scripting language for dialog, and Comme il Faut (CiF) present a domain-specific language for social interaction rules. The Villanelle project is highly inspired by these projects, especially CiF and Versu's approach to rule-based modeling. Unlike these works, our focus specifically on *autonomous characters* leads us to design considerations not found in these tools. We also make an effort to improve on these predecssors with (a) a simpler core formalism, to minimize the constructs a beginner has to learn to get started; (b) a more general range of expressivity paired with a library of common actions; and (c) an emphasis on open, portable, and legible definitions for the core technologies so that they may be easily re-implemented and reused in other systems.

Project Goals

The goal of this project is to develop generalizable, transferable ideas that facilitate developing real, playable works of interactive narrative with autonomous characters. Although story generation, drama management, and autonomous characters can be realized and rendered to players in contexts as rich and interactive as real-time virtual or live-action experiences, for this project we are currently limiting the scope to turn-based, text-based adventures with a range of player interfaces for choosing actions (choice-based, structured action composition, or free form text). See Figure 6 for screenshots of the player's view of games we are currently developing using our authoring tool.

We are particularly motivated by *interactive fiction*, or interactive experience that primarily communicate through text. Parser-based interactive fiction in the style of Infocom's games has an active community of hobbyists producing works such as *Lost Pig* (see Figure 1, in which the player controls an ogre in search of a lost pig. Apart from the charming style of the player character's narration, the appeal of the game can be seen in its scripting of non-player characters, such as the pig (seen exhibiting behavior in the screenshot) and a tinkergnome that engages in various tics and crafting activities while the player speaks to them. These examples speak to the way autonomous NPCs can enrich a player's experience, and we hope to simultaneously enable easier authoring and a more expressive range of behavior than current IF authoring tools (such as Inform 7 [Ree10]) currently support.


```
Fountain Room                                1
```

Fountain Room
All wall in this room glow! It bright, just like day time. Except that instead of sun, it wall that glow. And instead of grass and tree, it square room with four wall. And instead of outside, it all under ground. But beside those thing, it just like it! Tunnel keep going out to east and west, but normal doorway go north and southeast and southwest.

In middle of room, there fountain. But fountain not have any water.

South wall have big curtain hanging on it.

Hey! There pig! It good thing that Grunk fall down same hole as pig.

> catch pig
Grunk walk right up to pig. Pig walk right away from Grunk.

> catch pig
Pig run away. So Grunk run after pig. Then pig run around behind Grunk and around front of Grunk and Grunk get dizzy. What Grunk doing again?

>|

Figure 1: A screenshot of the game *Lost Pig*. The pig's actions execute each time the player issues a command.

Accessible Authoring

While an *accessible* tool is often easy to recognize when we see one, we need to more carefully define what we mean and how this goal informs our tool design. We thus break down accessibility into four components: (1) tool learning curve, e.g. how readily a newcomer to the tool can get hands-on experience with the tool and engage in self-directed learning; (2) expressiveness, i.e. how readily authors can identify natural ways to express the story idioms they want to express; (3) reasoning, i.e. given an author's design goals, how much certainty can they develop that their authored project satisfies those goals (and how easy is it to identify and resolve mismatches, i.e. "debug" the project); and finally, (4) scalability, or how well do all of the above principles scale to bigger and more complex projects? We then map these criteria onto four features of a tool that supports them: (1) a minimal necessary feature set (to support a low barrier to entry); (2) an extensible, sensible library of defaults and composable examples (to support expressiveness); (3) support for logical specifications (to support reasoning); and (4) support for composition and reuse (to support scalability).

Approach

We define our approach by two high-level features that distinguish it from similar efforts: first, that it is *practice-driven*, i.e. facilitates the development of real, playable works of interactive fiction by authors with stories to tell but who may not have a programming background. Second, our approach is *language-based*, meaning that the research contribution is not specifically embedded in the software *tool* we are building but in the transferable, generalizable authoring *language* that it embodies. In this paper we make an attempt to separate the formal definition of the language from its realization as an editing environment and engine for running character scripts.

Practice-driven. As mentioned above, we are driven by works of text-based interactive fiction such as *Lost Pig* and *Galatea* [Sho00], as well as by more complex media experiences like Facade [MS03a], in which characters are responsive to player input but also to motivations, stimuli, and interiority that the player does not control. In addition to our own games, we are also in the process of engaging with interactive fiction authors through surveys and social media discussions to solicit input on the kinds of stories they would like to be able to tell with better tools for creating these characters.

Language-based. Rather than focusing purely on the idea of an authoring *tool* as a black-box piece of software, we instead take the view that authoring *languages* should be designed and published independently from the tools that compile, interpret, and render them with an engine. This view is not unusual: in some sense, Inform 7, Twine 2, and the generative art system Processing take this approach as well, providing "domain-specific languages" for the practice in question. Twine and Processing (unlike Inform) provide macro systems and libraries within existing general purpose languages (JavaScript and Java, respectively), and Processing even provides support for multiple host languages (including Python and Javascript). Separating language from tooling and even host language dependencies makes tools more adaptable to the existing workflows and environments that many potential users are already familiar with, while not precluding the development of a standalone tool for programming novices.
Integrating Scripting and Generativity. Whereas most planning-based systems integrate a representation formalism (story events as preconditions and effects in propositional logic) with a search procedure (generate a partially-ordered set of events satisfying narrative constraints), we separate these two ideas and start with "plan notation" as a baseline for an author's expressive pallette, allowing them to sequentially compose operators in a planning domain. Inspired by behavior trees, we then observe the usefulness of *reactivity* in characters, i.e. the ability to break from planned sequences to respond to more immediate events, and incorporate mechanisms for branching and looping into the plan notation.

Villanelle: An Authoring Framework

Villanelle is a software project representing a collection of ongoing efforts to improve character authorship for interactive narratives. The project takes a *language-based approach* to authoring, which means we first design a core calculus with a minimal set of orthogonal constructs about which we can reason formally, then we design authoring tools, syntactic sugar, and editor support on top of the linguistic foundation.

In Villanelle, authors provide a world description (initial story state), a set of atomic story event or action templates, a set of hand-authored behaviors, and a narrative control graph, all of which will have varying options for automated generation. Villanelle compiles these components into a runnable interactive story rendered in the browser, by default in a turn-based mode wherein the world updates whenever the player takes an action. Each time the player acts, successor world states are computed based on the action's effect, the non-player character behaviors, and narrative control scripts.

Eventually, our goal is for authors to be able to author stories using Villanelle through a *structure editor*, a syntax-error-free programming environment made of composable GUI elements, which makes programming-like abilities more accessible to an audience without programming background.[1] The system as a complete authoring tool will consist of a world editor, atomic action editor, character and narrative script editor, execution engine, and a suite of analysis and debugging tools.

Plans as authorable notation

The planning approach to narrative generation brings many ideas into a single framework. At a high level, we can subdivide these ideas into *representation* and *search*. In terms of representation, the approach prescribes that we describe narrative states in terms of logical predicates and narrative events or actions in terms of transformations on those states, i.e. the preconditions and effects that an event would have, usually generalized over some set of term parameters to describe event *schema*. *Search* refers to the planning algorithm itself, which accepts a starting and ending state (and potentially some other constraints) and returns a valid plan transforming one into the other.

Another aspect of representation in planning is the notation for plans themselves. Usually rendered to a viewer as a sequence of plan operations instantiated with ground terms, these structures have many interesting properties: they are always *valid* sequences of operations, assuming that nothing can modify the state except for the processes responsible for executing the plan. They also track dependency relationships between actions so that a *partial order* structure can be understood, rather than a strict sequence, allowing some events to permute around one another or to occur in parallel.

In Villanelle, we separate the facility of domain representation and plan notation from *planning* as a process. Doing so gives us the ability to consider plan notation as an authoring language, whose expressiveness we can then extend.

Consider the example in Figure 2 based on classic text adventures in which characters can move, take objects, and give objects to other characters. Given these action specifications, we can write down sequences of actions such as

```
move(ang, den, foyer); take(ang, wand);
move(ang, foyer, den); give(ang, wand, beryl)
```

This sequence constitutes a valid plan for certain conditions that we can extract from the plan itself, and it has similarly inferrable composite effects. Specifically, we can give this plan the following specification:

[1]Some people refer to these systems as "code-free," but we prefer to think of it as better usability design for writing code.

```
% Character C moves from location L to L'
move(C : character, L : location; L' : location):
  pre: at(C, L), adjacent(L, L')
  eff: at(C, L'), adjacent(L, L')

% Character C takes object O from location L
take(C : character, O : object; L : location):
  pre: at(C, L), at(O, L)
  eff: at(C, L), has(C, O)

% Character Giver gives object O to character Receiver in location L
give(Giver : character, O : object, Receiver : character; L : location):
  pre: at(Giver, L), at(Receiver, L), has(Giver, O)
  eff: at(Giver, L), at(Receiver, L), has(Receiver, O)
```

Figure 2: A small sample domain defining three actions. Identifiers beginning with capital letters are parameters to the action (universally quantified variables), and parameters separated by a semicolon (;) are "implicit" parameters that may be inferred given the parameters preceding the semicolon.

```
pre: at(ang, den), at(wand, foyer), at(beryl, den)
eff: at(ang, den), at(beryl, den), has(beryl, wand)
```

Next, we can abstract over plans to accept term parameters the same way that the basic operators accept such parameters:

```
fetchGive(A, O, B; L1, L2) =
  move(A, L1, L2); take(A, O); move(A, L2, L1);
  give(A, O, B)
```

This behavior can also be given a general specification of its preconditions and effects, expressed in terms of its parameters:

```
pre: at(A, L1), at(L2, O), at(B, L1)
eff: at(A, L1), at(B, L1), has(B, O)
```

As long as each operator has a specification, this *extrapolated* specification can be inferred algorithmically. The specification allows an author to view the `fetchGive` script as a black box: as long as the preconditions are satisfied, the script can run and produce known effects, and thus it can be used *compositionally* in future behaviors as though it were an atomic action. Although the story becomes more complex when considering multiple simultaneous behaviors, this concept of compositionality is a key part of supporting *author reasoning* about behaviors.

BTL: A behavior authoring language

The sequence language described above already supports authoring non-interactive simulations in a way that is amenable to an easy learning curve, rendering intermediate states after each action for immediate feedback, and tool support to check that a script is valid (executable) within a given starting environment. However, generally the premise of interactive narrative supposes stories that emerge from interaction and are not necessarily pre-planned. In this section, we extend the language above with constructs that support reaction and interaction between multiple plans executing within the same environment. Building on the formalism of sequential plans described above, we designed a "scripting language" within Villanelle which extends sequential behaviors with conditional tests, branching, and iteration.

Construct	Abs. syntax	Concrete syntax
Operators	a	`op(args)`
Sequences	$\pi ; \pi$	`sequence {b1; ...; bn}`
Selectors	$\pi + \pi$	`selector {b1, ..., bn}`
Abstractions	$\Lambda x.\ \pi$	`f(x1...xn){ b }`
Conditions	$?\phi.\ \pi$	`preconditon: P { b }`
Iteration	ϕ^*	`repeat { b }`

Figure 3: The syntax for BTL describing behaviors π in abstract notation and `b` in concrete ASCII notation.

Conditional Branching. The executability of a hand-authored plan depends on the idea that it is the *only* plan being executed, and nothing else is manipulating the state of the world. Otherwise, external state changers could invalidate the premises of the actions making up the plan—for example, if some agent (player or NPC) picks up the shed key from the attic, then another character's action to do so will fail. In a fully generative architecture, such a failure would trigger a "re-plan" phase, resulting in a new behavior whose relationship to the original one is hard to know and impossible to pre-author.

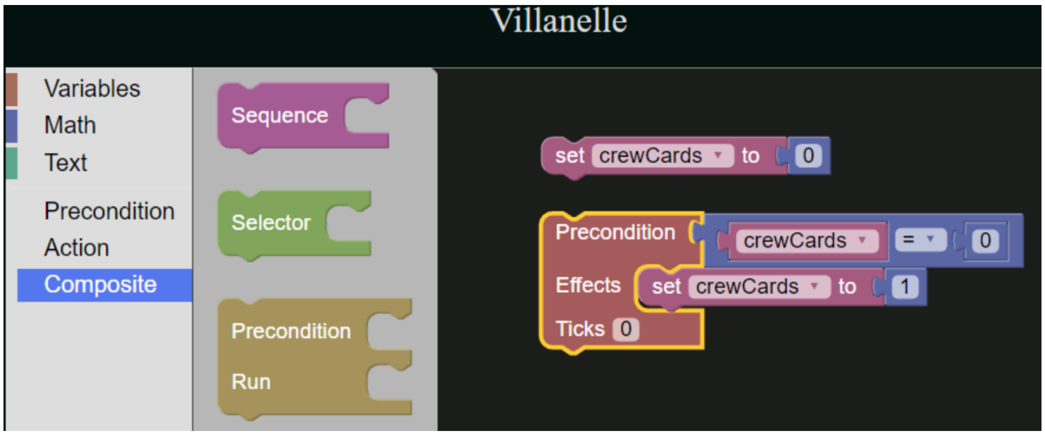
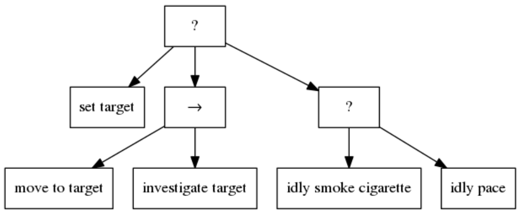


Figure 5: The Blockly interface to authoring Villanelle behavior expressions.

In practice, many games industry professionals use *behavior trees* (BTs) to author responsive behaviors that handle environmental uncertainty like this. Behavior trees, in addition to action sequences, contain *selectors* that pick between several alternate courses of action, and *conditional tests* that sense the current world state and proceed only under specific circumstances. The combination of selectors and tests gives way to an idea similar to "conditional" or "contingency" planning. See Figure 4 for a BT example whose semantics matches our notion of conditional branching.

Iteration. By default, when a plan is finished executing, no more behavior is generated. On the other hand, behavior trees re-execute from the root after each time they reach a leaf node of the tree. We allow the author to choose which behavior they want by introducing an explicit iteration construct that can be attached to any top-level behavior. For example, the example in Figure 4 implicitly assumes top-level iteration to make the agent stay responsive to changes in its environment.



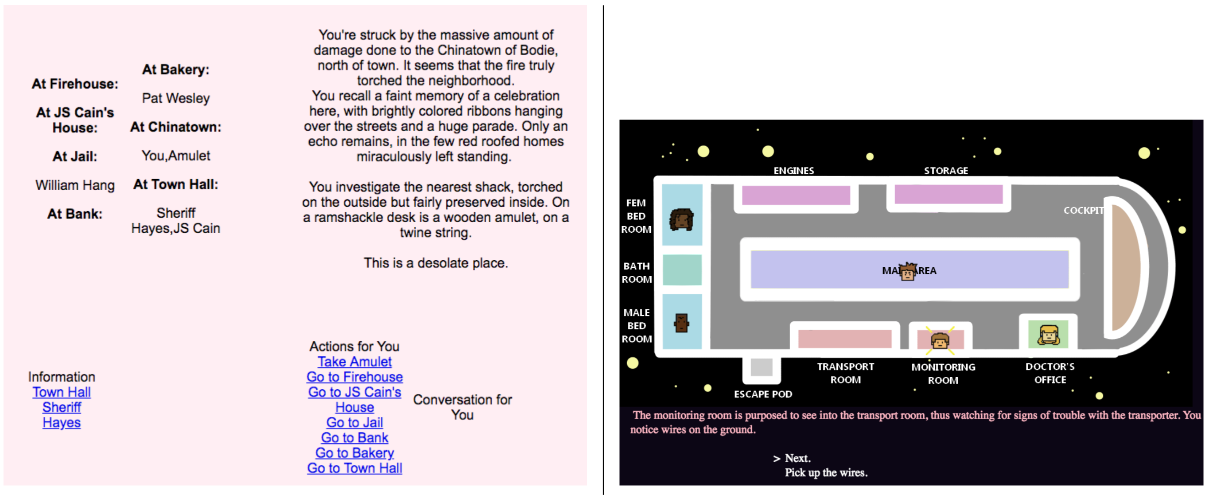
```
investigate_noise(C) = selector {
  set_target(C),
  sequence { move_to_target(C); investigate_target(C) },
  selector {smoke(C), pace(C)}
}
```

Figure 4: A example behavior tree for a noise-investigation behavior, and a corresponding BTL expression. The tree is evaluated in preorder traversal. Leaf nodes specify actions in the world (such as moving to a target), which can succeed or fail. Interior nodes combine children into composite behaviors. The arrow (→) is sequencing, and the question (?) is selection.

BTL. Putting these pieces together, we arrive at the language *BTL* whose syntax is summarized in Figure 3. In subsequent sections, we demonstrate examples using this syntax. It is outside the scope of this paper to provide a formal semantics for the language, but we will provide a brief intuition. Similar to behavior trees, each behavior expression can succeed or fail. *Sequences* run by executing each subexpression in sequence and succeed when the last element succeeds. *Selectors* try each subexpression in order and succeed as soon as the first one succeeds. *Conditions* succeed (and execute the subexpression) only when its first argument, a logical formula characterizing properties of the narrative state, is true in its current environment. Iterations fail if their subexpression ever fails and otherwise continues running.

Authoring Interface

Computer science education research has demonstrated that, for programming novices, it is helpful to provide code-editing environments that eliminate syntax errors by providing a GUI and palette of syntax constructs that snap together only for well-formed programs. Fortunately, with a formal syntax definition, it is easy to create such a GUI tool using Google's Blockly tool. In Figure 5, we show our prototype of a GUI editor for BTL behaviors. The editor generates corresponding JavaScript code that runs in our web-based engine.


At Firehouse:

At Bakery:

Pat Wesley

At JS Cain's House:

At Chinatown:

At Jail:

You,Amulet

William Hang

At Town Hall:

At Bank:

Sheriff Hayes,JS Cain

You're struck by the massive amount of damage done to the Chinatown of Bodie, north of town. It seems that the fire truly torched the neighborhood. You recall a faint memory of a celebration here, with brightly colored ribbons hanging over the streets and a huge parade. Only an echo remains, in the few red roofed homes miraculously left standing.

You investigate the nearest shack, torched on the outside but fairly preserved inside. On a ramshackle desk is a wooden amulet, on a twine string.

This is a desolate place.

Information
Town Hall
Sheriff
Hayes

Actions for You
Take Amulet
Go to Firehouse
Go to JS Cain's House
Go to Jail
Go to Bank
Go to Bakery
Go to Town Hall

Conversation for You

The monitoring room is purposed to see into the transport room, thus watching for signs of trouble with the transporter. You notice wires on the ground.

> Next.
Pick up the wires.

Figure 6: Screenshots of *The Mystery of Bodie Township* and *Space Escape*.

Examples

The Mystery of Bodie Township

The Mystery of Bodie Township is a detective story game set in the historic Bodie, California, a legendary gold rush town. The original version of the game was created in Twine as a promotion for the website of the actual historic site. You play the grandchild of a deceased resident and arrive in town to find the aftermath of a severe arson that burned down a neighborhood. Your role is to talk to everyone in town to uncover enough evidence to accuse the correct perpetrator.

The game contains a relatively large cast of characters in an interconnected social web. In the Twine game, character behavior is static: each visit to a character yields the same story and conversation text, and characters appear as stationary within the parts of town they start in, doing nothing but awaiting the player's actions. We therefore found it an excellent candidate to adapt to Villanelle's character engine, sketching out characters' personalities, goals, and motivations to inspire us to script what they would be doing in the game as the player works to uncover the mystery. For example, the actual perpetrators of the crime may be trying to hide incriminating evidence, obtain a car, and leave town; the sherriff may be going door to door to collect information.

Because this game is largely about information and knowledge, much of which is exchanged through conversation, we created a conversation engine to track characters' mental states. These states are mappings from *topics* (short, unique identifiers) to *quips* (each a unique identifier paired with some longer display text), a system adapted from Emily Short's description of the conversation system used in Emily Short's *Galatea* [Sho00]. Characters also track who they've told what: each character also maps quips to a list of "people told" so that they don't repeat themselves in conversation. Finally, characters track topics with *empty* lists of associated quips to represent that they are aware of the topic, but don't know anything about it.

Knowledge gives characters the ability to *ask about* and *tell about* topics with other characters. Asking about a topic might prompt a tell response from the other character (depending on trust level), unless they don't have any available quips that they haven't already told you. Telling someone about a topic is possible only when the teller hasn't already told them, and its effects are to add the topic to the tellee's knowledge base—at which point they can then tell other characters, spreading knowledge in a grapevine fashion.

Characters can learn things outside of conversation, as well. Interactive fiction frequently gives players the option to *look* at their surroundings and *examine* specific things within it. These actions make less sense from an NPC point of view, usually, but in this game, we actually give these actions *knowledge effects*: looking adds awareness of certain objects to the characters' knowledge base, and examining them can add quips to their knowledge of that object's topic. For example, a character might observe a locked hatch at the firehouse. This makes the topic **hatch** available for asking other characters about. A sample script that's authorable with these actions is:

```
look(bayard, firehouse); examine(bayard, hatch);
move(bayard, townhall); ask(bayard, sheriff, hatch);
give(sheriff, hatch-key, bayard)
```

```
precondition: (location_of_player == engines)
sequence:
  displayDescriptionAction: {text:
    "You enter the engines room. It is dark except for the blinking
    dim red lights on the consoles."};
  addUserAction {text: "Make your way to the cockpit.",
      effect: location_of_player == cockpit}
```

Figure 7: Code for giving the player an interaction choice

Player Interaction

Using the operator specifications in the Action Library, the engine can generate a choice corresponding to every player action whose preconditions are met. These choices can then be presented to the player as hyperlinks (or, with additional tooling, a parser interface could be introduced). We treat the player character and non-player-characters symmetrically in the sense that they all have access to the same library of actions: NPC scripts consist of sequences of the same library actions afforded to the player. See Figure 6 for a screenshot of the player's view, which also exposes the current locations of the other characters.

Space Escape

Space Escape is an in-progress science fiction game demonstrating Villanelle's facility for inter-character coordination. The narrative premise is that a crew of characters with different skills, personalities, and relationships to one another need to coordinate to diagnose a malfunctioning spaceship and fix several subsystems. Tonally, the game aims to be funny, tense, and emotional, causing the player to form attachments to the characters, learn about their insecurities, desires, and love interests, and ultimately face a difficult dilemma: it is not possible to fix all subsystems of the ship, so even in the best case, the player will need to decide how to allocate the two remaining escape pods among the five crew members. See Figure 6 for a screenshot.

Space Escape is our first exploration of a game with a relatively large cast of characters, each with scripted, looping behaviors that control aspects of their personality: where they prefer to be on the ship, how they will respond to certain dialog from certain other characters, and how they respond under stress. An additional trick is that the game uses behavior scripts as a game mechanic: the player, by assigning tasks to crew members, is in fact editing the behavior script for that character (in a similar vein to Inform 7's "ask ⟨someone⟩ to ⟨command⟩" construct). The skill challenge in the game is one of getting to know your crew members' competencies and listening to their needs: assigning the wrong tasks to the wrong people, or using force or caring language in the wrong circumstance, will elevate resentment and slow progress.

Player Interaction

In *Space Escape*, rather than generating the player interface as the set of all possible actions, we demonstrate a way to hand-author the player's choices in specific scenarios. Authors can customize the set of choices available to players under certain circumstances using the behavior syntax. Figure 7 shows an example for providing a navigation option. This system affords a Twine-like CYOA interface, but with more generality due to the expressive precondition language—certain actions may only be available when the player has certain items, for example, and the effects of player actions can make arbitrary changes to the narrative state.

Open design challenges

Reasoning about composite behaviors

While the system presented can extrapolate conditions and effects of plans (sequential behaviors), extrapolating them to full behaviors with selectors and tests is a work in progress. Some selector-based behaviors have the same end result no matter which path through the tree is taken, but others would result in a disjunction of possible outcomes. A viable approach for handling selectors will probably rely on a combination of author annotations and automatic effect inference.

Extrapolating specifications for iterated behaviors is an open challenge as well. Many behavior engineering patterns based on Behavior Trees rely on a complete path through the tree executing once per game step, and different branches execute based on the effects of previous branches, such as the hide-and-seek behavior from Figure 4. Defining an abstract representation of the preconditions and effects for such a behavior also remains an open challenge.

Finally, most games (including both of our examples) involve running multiple behaviors simultaneously, which manipulate a shared state (the game environment). While it is possible for the author to combine all character-specific behaviors into a single "global" behavior, this approach is non-compositional and does not scale with the size of the project. Therefore, Villanelle's approach is to allow "attaching" trees to specific agents and to run a randomized round-robin scheduler for executing the next step of each agent's plan. But this process, like programming with concurrency, can create a large number of possible behavior interleavings that is hard to predict, even though many examples may be intended to always evolve in a synchronous manner. The problem of designing tools that allow authors to state their intent for concurrent behaviors, observe when that intent is violated by the program, and debug the program, is an open problem in formal methods whose solution would greatly benefit this project.

Interruptable Sequences

Related to concurrent behaviors is the concept of *reactivity* in behavior trees. In some BT implementations, a new traversal through the tree is executed from the root at every game step, *even between steps of a sequence*. This permits behavior with an increased level of apparent "reactivity," e.g. a character who stops to pick up a hundred dollar bill on the ground while walking to a destination. However, affording this expressiveness leads to a huge loss for reasoning: if a sequence can be interrupted at any step, the narrative state space explodes with every possible interleaving of alternate behaviors with steps of the sequence. Furthermore, an agent can appear to "forget what it was doing" if it stops to pick up the bill and then executes down a completely different branch. These failure modes and others are documented in a detailed developer blog post [Ang14] that proposes using a notion of stored state or memory. An alternative that could integrate better with planning might be to use a model of tunable "commitment" to plans, as in Eger and Martens [EM18]. with authorable priorities to use for a heuristic of plan comparison.

Support for Behavior Debugging and Repair

Scripting and generating character behaviors gives rise to interesting, sometimes surprising emergent behavior, a desired effect of these systems. However, this same property makes it difficult to tell when an authored system is right, and what the culprit is when things go wrong. Twitter account `@TheStrangeLog` documents thousands of examples of "the strange poetry of changelogs and patch notes" from games that implement unpredictable behavior, such as "Fixed issue where a demon eating a skewer of human skulls would unexpectedly dance when spoken to." These behaviors evoke a certain surrealist tone that has become a pastiche of the emergent narrative genre, but to evolve as a storytelling medium (especially in serious genres), we need tools that give us more control over the generative possibility space. Villanelle's static precondition/effect specifications are one possible tool, but likely we will need other standard software engineering techniques, such as generative testing, assertions, and suggestions for repairing behavior scripts to match stated specifications. Making these tools accessible and comprehensible to authors is an open research challenge whose payoff would be narrative designers equipped with robust, systemic mental models of the generative spaces that they are authoring.

Future work

We are in the process of conducting an iterative, community-informed design-and-evaluation process for Villanelle, planning for frequent requests for feedback from our target audience. We have conducted an initial, informal survey of interactive fiction developers whose responses have informed the first draft of our framework. The next step will be to design the next step of engagement with this audience by staging demos and side-by-side comparisons of our design proposals to solicit additional feedback. Meanwhile, we are working with writers and others with little programming background to develop our two showcase examples.

An eventual goal for the project is to port the core technology (BTL and specifications) to popular frameworks for interactive narrative creation. We have selected Inform 7 and Unity as two testbeds with which to integrate; the differences between these tools will no doubt provide an educational test of our claim to platform-independence.

Conclusion

We have presented *Villanelle*, an interactive narrative authoring framework whose goal is to support rapid self-directed learning by authors, sensible but extensible defaults, composition and reuse, and reasoning about composite behaviors. Villanelle consists of a platform-independent behavior language BTL, a block-based editing tool, and execution engine, as well as proposals for extending this framework with integrated run-time plan generation. We have showed two game projects using the framework to showcase and motivate its capabilities, discussed ongoing design challenges, and invite feedback and collaboraton from the intelligent narrative community on these issues.

References

[Ang14] Bobby Anguelov. Synchronized behavior trees, 2014.

[Ayl99] Ruth Aylett. Narrative in Virtual Environments - Towards Emergent Narrative. In *Working Notes of the Narrative Intelligence Symposium*, 1999.

[CCM02] Marc Cavazza, Fred Charles, and Steven J Mead. Character-based interactive storytelling. *IEEE Intelligent systems*, 17(4):17–24, 2002.

[CKM$^+$08] Yun-Gyung Cheong, Yeo-Jin Kim, Wook-Hee Min, Eok-Soo Shim, and Jin-Young Kim. Prism: A framework for authoring interactive narratives. In *Joint International Conference on Interactive Digital Storytelling*, pages 297–308. Springer, 2008.

[COM$^+$07] Maria Cutumisu, Curtis Onuczko, Matthew McNaughton, Thomas Roy, Jonathan Schaeffer, Allan Schumacher, Jeff Siegel, Duane Szafron, Kevin Waugh, Mike Carbonaro, et al. Scriptease: A generative/adaptive programming paradigm for game scripting. *Science of Computer Programming*, 67(1):32–58, 2007.

[EM18] Markus Eger and Chris Martens. Keeping the story straight: A comparison of commitment strategies for a social deduction game. In *AIIDE*, pages 30–36, 2018.

[ES14a] Richard Evans and Emily Short. Versu—A Simulationist Storytelling System. *Transactions on Computational Intelligence and AI in Games*, 6(2):113–130, 2014.

[ES14b] Richard Evans and Emily Short. Versu—a simulationist storytelling system. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(2):113–130, 2014.

[Isl05] Damian Isla. Handling complexity in the halo 2 ai. In *Proceedings of the 2005 Game Developers Conference*, 2005.

[KFZ$^+$15] Mubbasir Kapadia, Jessica Falk, Fabio Zünd, Marcel Marti, Robert W Sumner, and Markus Gross. Computer-assisted authoring of interactive narratives. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*, pages 85–92. ACM, 2015.

[MS02] Michael Mateas and Andrew Stern. A behavior language for story-based believable agents. *IEEE Intelligent Systems*, 17(4):39–47, 2002.

[MS03a] Michael Mateas and Andrew Stern. Façade: An Experiment in Building a Fully-Realized Interactive Drama. In *Game Developers Conference*, volume 2, 2003.

[MS03b] Michael Mateas and Andrew Stern. Integrating plot, character and natural language processing in the interactive drama façade. In *Proceedings of the 1st International Conference on Technologies for Interactive Digital Storytelling and Entertainment (TIDSE-03)*, volume 2, 2003.

[MTS$^+$10] Josh McCoy, Mike Treanor, Ben Samuel, Brandon Tearse, Michael Mateas, and Noah Wardrip-Fruin. Authoring game-based interactive narrative using social games and comme il faut. In *Proceedings of the 4th International Conference & Festival of the Electronic Literature Organization: Archive & Innovate*. Citeseer, 2010.

[PCC10] Julie Porteous, Marc Cavazza, and Fred Charles. Applying planning to interactive storytelling: Narrative control using state constraints. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 1(2):10, 2010.

[Ree10] Aaron Reed. *Creating interactive fiction with Inform 7*. Cengage Learning, 2010.

[RGWFM14] Aaron A Reed, Jacob Garbe, Noah Wardrip-Fruin, and Michael Mateas. Ice-bound: Combining richly-realized story with expressive gameplay. In *FDG*, 2014.

[RY10] Mark O Riedl and Robert Michael Young. Narrative planning: Balancing plot and character. *Journal of Artificial Intelligence Research*, 39:217–268, 2010.

[Sal16] Anastasia Salter. Code before content? brogrammer culture in games and electronic literature. *Hyperrhiz: New Media Cultures*, (Hyperrhiz 17), 2016.

[Sho00] Emily Short. Galatea. *Electronic Literature Collection: Volume One*, 1, 2000.

[SWM06] Ulrike Spierling, Sebastian A Weiß, and Wolfgang Müller. Towards accessible authoring tools for interactive storytelling. In *International Conference on Technologies for Interactive Digital Storytelling and Entertainment*, pages 169–180. Springer, 2006.

[VV08] Martin Van Velsen. Narratoria, an authoring suite for digital interactive narrative. In *FLAIRS Conference*, pages 394–395, 2008.

[You99] R Michael Young. Notes on the use of plan structures in the creation of interactive plot. In *AAAI Fall Symposium on Narrative Intelligence*, pages 164–167, 1999.