An Answer Set Encoding for Narrative Planning with Theory of Mind

Molly Siler, Stephen G. Ware

Narrative Intelligence Lab, Department of Computer Science, University of Kentucky
329 Rose Street
Lexington, Kentucky 40506
molly.siler@uky.edu, sgware@cs.uky.edu

Abstract

There has been much research into making planning-based story generators more efficient; however, the question remains whether the same efficiency could be achieved by reducing the problem to a more widely-studied search problem and leveraging existing solvers. We investigate this question for the narrative planning formalism used by Sabre, which models character goals and beliefs with deeply-nested theory of mind. We use answer set programming to develop a declarative implementation of the same planning formalism. Benchmarking our implementation, we find that existing, specialized planners remain the state of the art for solving their target problems as quickly as possible. However, the compactness and modularity of our approach will make it easier for researchers to develop prototype generators for new solution spaces that build on existing models.

1 Introduction

Machine-learning-based approaches have seen resounding success in recent narrative generation research, taking advantage of improvements in large language models (LLMs) (Sun et al. 2023; Kumaran, Rowe, and Lester 2024). Compared to symbolic narrative generators, they avoid the need for extensive knowledge engineering to define the possible structure and content of stories. The two paradigms can complement each other in neurosymbolic systems, with symbolic models being used to guide long-term coherence in LLM-generated stories (Martin 2021). However, we also argue that the trait of symbolic approaches often considered a weakness—the fact that they require the system builder to commit to a precise, explicit definition of the story space—makes them interesting for study in their own right. These approaches can contribute to narratology by providing testable formal models of narrative structure and human narrative reasoning (Mateas and Sengers 1999).

Narrative planners (Young et al. 2013) are a family of symbolic narrative generation algorithms that use the propositional state and action model of classical planning to represent events within a story world. They are distinguished from classical planners by how they define a valid solution—not only requiring a goal to be achieved, but also constraining the plan to be story-like with properties such as the appearance of character autonomy (Riedl and Young 2010).

One area of study is refining the constraints on character behavior (Ware et al. 2014; Sanghrajka, Young, and Thorne 2022) or overall story structure (Bae and Young 2008; Cardona-Rivera and Li 2016) to incorporate new narrative properties into the solution definition. As Dabral and Martens (2020) point out, many works in this area involve a large amount of engineering effort using an imperative programming language to build a new planner satisfying the solution definition of choice. Dabral and Martens instead argue for plan generation in a declarative language, allowing for a concise and more easily extensible codebase that promotes faster implementation of new narrative constraints.

This argument makes declarative narrative planning attractive when the objective is to study new solution spaces in a research setting. There is another comparison to be made, however, in terms of computational performance. On one hand, specialized planners use search strategies that take advantage of story-specific properties (Birchmeier and Ware 2025; Senanayake and Ware 2025); on the other, a declarative approach benefits from powerful solvers from a more heavily-studied research area (Horswill 2021), some of which also allow for custom heuristics (Gebser et al. 2013).

This paper introduces a new declarative encoding of narrative planning written using answer set programming (ASP) (Brewka, Eiter, and Truszczyński 2011). One of our contributions is the encoding itself. ASP-based narrative planners (Dabral and Martens 2020; Siler and Ware 2020) and plan validators (Wang and Kreminski 2025) have been written before, but ours is novel in that it incorporates features introduced in Sabre (Ware and Siler 2021). These features include *triggers*, state updates that occur automatically similar to the notion of natural exogenous actions (Reiter 1996) or rigid actions (Subrahmanian and Zaniolo 1995); but most importantly, a model of character belief where theory of mind (one character believes that another believes that...) can be deeply nested (Shirvani, Ware, and Farrell 2017). Although the Dabral and Martens (2020) encoding is versatile, it generates partial-order plans without explicitly tracking state, and using that paradigm to represent a Sabre-style belief model is still an open problem; we chose to develop an encoding for total-order plans with explicit state to handle the belief model with a preestablished method.

Our encoding makes the Sabre model more accessible not only for researchers modifying the constraints to develop new planner variants, but also for authors of new planning domains. Rather than requiring a problem instance to fully specify the values of fluents in the initial state, our encoding accepts an arbitrary proposition over the fluents as a *most general initial state* (Ware and Young 2010); a plan can be generated if it is legal in any fully-specified initial state consistent with the proposition. This allows domain authors to experiment with the range of stories their domains can generate by effectively specifying a whole class of problem instances in one, testing whether and how a set of starting conditions could lead to a particular outcome.

Another contribution of this paper is benchmarking our ASP planner on a suite of established narrative planning problems. We find that our encoding can generate plans at a speed that is useful for researchers prototyping new solution spaces, but fails to outperform specialized planners. Our study is not conclusive about the overall performance potential of declarative approaches but highlights the complementary value of imperative-language planners.

2 Related Work

Our work is at the intersection of several research areas.

Narrative planning solution spaces Our formulation of narrative planning traces its ancestry to IPOCL (Riedl and Young 2010), which extends classical planning to add a notion of intentionality: Actions are taken by characters with goals, and a character's action must lead to the achievement of the character's goal. CPOCL (Ware et al. 2014) modifies the intentionality requirement so a character action must be part of a *hypothetical* plan to achieve a character goal, but can fail to achieve the goal in the executed plan. The ASP planner by Dabral and Martens (2020) bases its solution definition on CPOCL's. The model defined by Shirvani, Ware, and Farrell (2017), and implemented by Sabre (Ware and Siler 2021) and by our encoding, combines intentionality with a model of characters' beliefs. In a graph of possible worlds, a character's beliefs in one state are represented by a pointer to another state, allowing for recursive theory of mind. Other belief models in narrative planning include those that represent beliefs about the world state only (Sanghrajka, Young, and Thorne 2022), a non-recursive layer of theory of mind (Christensen, Nelson, and Cardona-Rivera 2020), or a separation of beliefs into public and private knowledge (Teutenberg and Porteous 2015).

Open-world planning The problem of planning stories from partially-specified initial states was formulated by Riedl and Young (2005). Their algorithm, Initial State Revision, modifies partial-order planning to handle initial states where some facts have undetermined truth values; the truth values become determined when used to satisfy preconditions. After originally using a classical planner as its base, Initial State Revision was later implemented (Riedl and Young 2006) in a narrative planner incorporating character intentionality. Ware and Young (2010) proposed a generalization of the algorithm in which the initial state was an arbitrary formula and consistency was checked with a SAT solver, similar to conformant planning (Hoffmann and Brafman 2006) except ensuring that the plan solved the problem from *some* rather than *all* completions of the initial state. Open-world planning is reminiscent of automated problem modification models such as excuse generation (Göbelbecker et al. 2010): taking an instance where a plan could not be found, and highlighting reasons for failure by showing modifications to the initial state, goal, or operators that would have made the problem solvable. RoleModel (Chen et al. 2010) and Retcon (Horswill 2022) are non-planner authoring tools that also use retroactive constraining of story worlds as the story develops.

Answer set planning The use of ASP for (non-narrative) planning dates back to Subrahmanian and Zaniolo (1995) and Lifschitz (2002), the latter of whom likened it to the emerging success of SAT-based planning (Kautz and Selman 1992) but with added synergies between ASP's non-monotonicity and planning's dynamics of how facts change or persist between actions. See Son et al. (2023) for a survey of later developments. In the area of multiagent epistemic planning, Burigana et al. (2020) present an ASP encoding of agent beliefs as a state-graph structure similar to our encoding of the Shirvani, Ware, and Farrell (2017) model.

Logic programming for procedural narrative Logic programming in general is the basis for a variety of narrative applications outside of planning. RoleModel (Chen et al. 2010) uses it to generate stories that reinterpret events in light of character roles (e.g., aggressors and victims). The Celf linear logic programming language has been used to define story worlds and analyze their causal structures (Martens et al. 2014), and the Ceptre language to analyze the gameplay spaces resulting from interactive narrative mechanics (Martens 2015). Social simulation is a particularly successful narrative application of logic programming that has seen commercial use (Evans and Short 2013; Zubek et al. 2021; Horswill and Hill 2024).

3 Narrative Planning Background

This section describes the narrative planning formalism that we translated to ASP. The formalism includes Sabre-specific (Ware and Siler 2021) features that extend more typical definitions of narrative planning.

A narrative planning problem defines a set of *fluents*, variables over a multi-valued domain. The possible values for a fluent may be booleans, numbers, or story-world objects. A special category of story-world objects is *characters*.

To define the other features of a narrative planning problem, we must refer to the notion of *state*. A state specifies an assignment of values to all fluents. For each character, a state specifies the character's beliefs, which are also a state; i.e., to find the character's beliefs about a proposition in a state, one can check the true value of the proposition in the belief state. Similarly, the state representing a character's beliefs can also point to other states representing the character's beliefs about others' beliefs, and so on.

A narrative planning problem defines any number of *triggers*. A trigger has *preconditions* and *effects*. The precon-

ditions are a proposition over the fluent assignments and/or beliefs in a state; when the proposition is true in a state, the trigger is automatically applied to derive a new state. The effects are changes to fluent assignments and/or beliefs that become true in the state resulting from the trigger; other fluents and beliefs retain their original value.

A narrative planning problem also defines a set of *actions*. Actions have preconditions and effects, but unlike triggers, they are chosen by the planner rather than happening automatically; an action's preconditions define when it is legal rather than when it is necessary. An action can have one or more *consenting characters*; these will be used in later definitions, but intuitively, they are the characters who "take" the action. An action also has an *observability condition*, a proposition, for each character. In a state where the action's observability condition holds for a character, if the action is taken, the character *observes* the action—the action is applied to the state representing the character's beliefs. This means that the action effects will become true in the belief state as well as the original state, and observations will happen recursively—characters within that belief state will observe the action if the observability conditions hold there.

A narrative planning problem provides a *utility function* for each character as well as an *author utility function*. A utility function is a mapping from states to numeric values; increasing a utility value is akin to achieving a goal in standard planning formalisms.

Finally, a narrative planning problem specifies a *most general initial state*, a proposition over fluent assignments and beliefs.

A solution to a narrative planning problem consists centrally of an action sequence $\{a_1, a_2, \cdots, a_n\}$ that satisfies a few properties.

First, there must exist a sequence of states $\{s_0, s_1, s_2, \cdots, s_n\}$ where:

- the most-general-initial-state proposition holds in s_0;
- each action a_i is legal in state s_{i-1};
- each state s_{i+1} is the result of applying the effects of a_{i+1} to state s_i, plus the effects of any triggers that were enabled; and
- the value of the author utility function is higher in state s_n than in s_0.

Second, for each consenting character c of an action a_i, there must exist an *explanation* for c taking a_i. Intuitively, an explanation shows how the action furthers the character's own interests by contributing to the character's plan to achieve a goal; it consists of an action sequence $\{a'_1, a'_2, \cdots, a'_m\}$ with a state sequence $\{s'_0, s'_1, s'_2, \cdots, s'_m\}$ where:

- s'_0 is the state representing c's beliefs in s_{i-1}, i.e., just before the action a_i was taken;
- $a'_1 = a_i$;
- each action a'_j is legal in state s'_{j-1};
- each state s'_{j+1} is the result of applying the effects of a'_{j+1} to state s'_j, plus the effects of any triggers that were enabled;
- if an action a'_j has consenting character $c' \neq c$, there exists an explanation for c' taking a'_j;
- the value of c's utility function is higher in state s'_m than in s'_0; and
- a'_1 is necessary for achieving the utility increase in s'_m, as defined below.

There have been a variety of ways to define the last bullet point, requiring that an action in some way contributes to the character's plan rather than being redundant. The Shirvani, Ware, and Farrell (2017) redundancy check involves checking causal links between the action and the utility increase, similar to the concept of *well-justification* proposed by Fink and Yang (1992). The Sabre version (Ware and Siler 2021) accepts an explanation if and only if no strict subsequence of the explanation achieves a utility increase at least as high, what Fink and Yang call *perfect justification*. We adopt a redundancy check that is stronger than the first one without the increase in complexity class needed by the second, using Fink and Yang's *greedy justification*.

In short, the check concludes that an action is necessary for the character's plan if and only if skipping the action, and skipping any actions prevented by previous skips, would result in a worse outcome for the character. More formally, given the candidate explanation $\{a'_1, a'_2, \cdots, a'_m\}$ starting in state s'_0 and ending in s'_m, a'_1 passes the greedy justification test and its explanation is accepted if there exists a state sequence $\{s''_1, s''_2, \cdots, s''_\ell\}$ where:

- $s''_1 = s'_0$;
- for $k > 1$, if a'_k is legal and explained in s''_{k-1}, then s''_k is the result of taking a'_k in s''_{k-1};
- for $k > 1$, if a'_k is not both legal and explained in s''_{k-1}, then $s''_k = s''_{k-1}$;
- the character's utility in s''_ℓ is lower than in s'_m.

In summary, a narrative plan is a solution if it improves the author's utility, and any given consenting character for an action can foresee a plan to increase their own utility starting with that action, such that the action is actually necessary to the character's plan.

4 ASP Background

We briefly summarize the ASP paradigm and the ASP-Core-2 (Calimeri et al. 2020) language as it is used in our later code examples. A more comprehensive introduction to ASP can be found in the article by Brewka, Eiter, and Truszczyński (2011).

A basic ASP problem consists of rules of the form `head :- body.`, where the head is a literal and the body is a comma-separated list of literals, treated as a conjunction. If the body holds, the head is derived. A solution to an ASP problem—an answer set—is a maximal set of literals that can be mutually derived from the rules.

A head with no body (i.e., `head.`) is treated as always true; a body with no head (i.e., `:- body.`) is called a *constraint*, and the body is forbidden from holding in an answer set.

For instance, given the rules `x.` and `y :- x.`, the single answer set consists of `x` due to the first rule, and `y` due to the second rule and the derivation of `x`. If we add the rule `:- y.`, there are no answer sets due to the contradiction between the derivation of `y` and the constraint against `y`.

A literal in a body may be negated; for a literal `x`, the negation is written as `not x`. Negations in ASP hold by default; i.e., `not x` should be interpreted as "`x` cannot be derived". For instance, given an ASP problem consisting of the single rule `y :- not x.`, the sole answer set consists only of `y`. If we keep the aforementioned rule but also have `x :- not y.` then there are two answer sets: one consisting only of `y`, and one consisting only of `x`, as we can apply either rule and the derivation will prevent the other rule from applying.

Literals can take the form of predications, such as `x(a)`. A predication with constants, such as numbers or lowercase terms, are handled like non-predicated variables, but this syntax also allows for predications over variables which are specified using uppercase. For instance, the rule `y(A) :- x(A).` is equivalent to the set of rules where constants are substituted for `A`; in an ASP problem where `x(1)` and `x(2)` are derived, the aforementioned rule will cause `y(1)` and `y(2)` to be derived as well. ASP solvers typically ground the program before searching for solutions; that is, they replace rules containing variables so there are only constants.

A *choice rule* has a head of the form `c1{ choices }c2 :- body`, where `choices` is a set of literals and `c1` and `c2` are numbers; when `body` holds, anywhere between `c1` and `c2` literals from the set may be derived. For instance, given the rules `z.` and `1{ x; y }2 :- z.`, the problem's answer sets are `x, z` and `y, z` and `x, y, z`. A choice rule may also be defined using variables, with a condition specifying what values the variable can take; for instance, given the rule `1{ y(A) : x(A) }1.`, if `x(a)` and `x(b)` hold, then one of either `y(a)` or `y(b)` is derived.

A percent symbol (`%`) at the start of a line denotes a comment.

5 Encoding and Solving

This section discusses how we encoded the narrative planning problem definition from Section 3 in an answer set program. We will describe features of the encoding at a conceptual level, showing code fragments when most relevant; the full codebase is available online.[1] Some code fragments are modified from the original when necessary to provide concise illustrations.

As input, the program takes an ASP representation of the problem instance to be solved, separate from the general ASP representation for narrative planning we describe later in this section. The ASP problem instance declares the characters, actions, triggers, fluents and their possible values, most general initial state, utility functions, and the logical expressions needed to define these, such as the propositions that make up action preconditions. We currently use Sabre's built-in parser to read and preprocess the problem file, with a custom printer to output the ASP rendering.

Listing 1: Example action declaration in a problem instance.

```
action(buy(teen,shoes)).

precondition(buy(teen,shoes), proposition1).
% Representing at(teen)==mall
comparison(proposition1).
operator(proposition1, equals).
left(proposition1, at(teen)).
right(proposition1, mall).

causes(buy(teen,shoes), at(shoes),
    teen).

consenting(buy(teen,shoes),
    teen).

observability(buy(teen,shoes),
    teen, true).

observability(buy(teen,shoes),
    mom, proposition2).
% Representing at(mom)==mall)
comparison(proposition2).
operator(proposition2, equals).
left(proposition2, at(mom)).
right(proposition2, mall).
```

For instance, Listing 1 shows an excerpt from an example ASP problem instance declaring an action where a consenting character, the *teen*, *buy*s some *shoes*; the precondition is that the teen must be at the *mall*, the effect is that the teen now possesses the shoes, and another character, the *mom*, observes the action only if also at the mall.

The problem-instance rules do not directly cause any reasoning to be performed; the actual semantics in the context of the plan result from domain-independent rules described in the rest of this section. We designed these rules to yield answer sets that map one-to-one with unique combinations of the following: a plan, a completed initial state from which the plan is a solution, and a set of explanations for all actions' consenting characters.

To generate these answer sets without prior knowledge of the size of the solution (e.g., number of actions and triggers required), we use the multi-shot solving (Gebser et al. 2019) functionality of the Clingo (Gebser et al. 2018) solver. This functionality allows for changes to program constraints and incremental additions of ground rules while keeping the preprocessing work already done by the grounder rather than starting grounding over from scratch. We divide a plan into units we call *nodes*, whose contents we will explain later in the section. Starting from $n = 0$, we try generating n nodes with a constraint that the plan must be completed at the nth node; if no plan is found, we add rules for generating the $n + 1$st node while keeping the rules for generating nodes $0, \cdots, n$, and we replace the constraint requiring a solution at the nth node with a constraint requiring a solution at the $n + 1$st.

[1] https://cs.uky.edu/%7Esgware/projects/aspnp/

Listing 2: Initial state assignment.

```
node(n).
base_state(n, s(n)).

node_type(n, initial) :- n=0.

% Choose values for all fluents
1{ assigned(S, F, V) :
    fluent_domain(F, V) }1 :-
        node_type(n, initial),
        base_state(n, S),
        fluent(F).
% fluent_domain(F, V) and fluent(F) are
%   from the problem instance

% Assignments must fit problem instance
:- node_type(n, initial), base_state(n, S),
    init(P), not holds(S, P).
% init(P) is from the problem instance
% holds(S, P) is defined elsewhere
```

Nodes are associated with one of several types prescribing their meaning in the final plan representation and which rules apply to them. For $n = 0$ only, the node generated is a *initial node*. Nodes contain states, including one which we will call the node's *base state* to distinguish it from states generated for auxiliary purposes. The initial node's purpose is to hold the initial state of the plan as its base state. A choice rule, shown in Listing 2, directs an arbitrary value to be assigned to each fluent in this state. A series of rules, not shown in the listing, determine what relevant propositions hold in the state via the `holds(S, P)` predicate; one of these propositions is the one defining the most general initial state, supplied as `P` in a literal `init(P)` given in the specific planning problem. A constraint, the last rule in the listing, only allows answer sets where the fluent assignments generated by the choice rule are consistent with that proposition. In other words, an answer set takes the most general initial state and completes it into a fully-specified initial state.

We track a character `C`'s beliefs in a state `S` using a literal of the form `beliefs(S, C, S')`, where `S'` is the state the character believes to be true. By default, as in Sabre, characters' initial beliefs about the values of fluents are assumed to match the true values of fluents; if a character has only these default beliefs, then `S'=S`. However, the problem instance may allow a character's initial beliefs to differ from the true fluent values. We generate and constrain these beliefs in the same way as the true initial fluent values; if wrong beliefs for the character are indeed generated, we define a separate state for `S'` within the initial node. Likewise, if a character has theory-of-mind beliefs—beliefs about the beliefs of another character—that differ from their own, we repeat this process recursively to create additional layers of belief states. We omit code fragments for these rules due to their length.

In later nodes, fluent values in states do not need to be generated and constrained in this way, as they follow deterministically from the interaction of previous states and actions or triggers. A node is assigned the type of *trigger node* if the preconditions for a trigger are met in any state associated with the previous node, including the base state but also the beliefs. The trigger effects are applied to those states to get the states for the new node; any fluent values not changed by triggers remain the same as before. These rules are illustrated in Listings 3 and 4. The `generated_at(S, N)` predicate is defined for states `S` associated with nodes `N`; the `derived_from(S', S)` predicate is defined for states `S'` that should copy their default values, besides effects, from an earlier state `S`.

Listing 3: Detecting and applying triggers.

```
node_type(n, trigger) :- occurs(n, T),
    trigger(T).
occurs(n, T) :- trigger(T),
    generated_at(S, n-1),
    precondition(T, P), holds(S, P).
effect_applies(n, S) :- trigger(T),
    occurs(n, T),
    generated_at(S, n-1),
    precondition(T, P),
    holds(S, P).
% generated_at(S, N) is defined elsewhere
```

Listing 4: Updating states based on effects.

```
assigned(S', F, V) :-
    occurs(n, E),
    causes(E, F, V),
    derived_from(S', S),
    effect_applies(n, S).
changed(S', F) :-
    derived_from(S', S),
    assigned(S, F, V1),
    assigned(S', F, V2),
    V1!=V2.
assigned(S', F, V) :-
    derived_from(S', S),
    assigned(S, F, V),
    not changed(S', F).
% derived_from(S', S) is defined elsewhere
% omitted: edge cases for modifying beliefs
```

A node is assigned the type of *action node* if no other node type is required. A choice rule for action nodes causes an action to be added to the plan from among those whose preconditions were met; besides the selection of initial state values, this is the only nondeterministic choice in the program.

An action applies in the base state; recursively, the action's observability to characters is checked to determine whether the action should propagate to belief states. These dynamics are illustrated in Listing 5. Like trigger effects, action effects result in updates to fluent assignments as modeled in Listing 4.

The plan generation process needs to produce explanations as well as the executed plan. At each node, our encoding maintains a list of prior action nodes that have not finished being explained yet. For an answer set to be considered a solution, this list must be empty at the final node and

Listing 5: Generating and applying actions.

```
node_type(n, action) :-
    not node_type(n, initial),
    not node_type(n, trigger),
    not node_type(n, explanation_start).
1{ occurs(n, A) :
    action(A), possible(S, A) }1 :-
        node_type(n, action),
        base_state(n, S'),
        derived_from(S', S).
effect_applies(n, S) :-
    occurs(n, A), action(A),
    base_state(n, S).
effect_applies(n, S') :-
    occurs(n, A), action(A),
    effect_applies(n, S), S'!=S,
    beliefs(S, C, S'),
    observability(A, C, P), holds(S, P).
% derived_from(S', S) is defined elsewhere
```

Listing 6: Initializing explanations.

```
node_type(n, explanation_start) :-
    node_type(n-1, action),
    occurs(n-1, A),
    consenting(A, C).
% consenting(A, C) is from problem instance
derived_from(S, S'') :-
    node_type(n, explanation_start),
    base_state(n, S),
    occurs(n-1, A),
    consenting(A, C).
    base_state(n-1, S'),
    beliefs(S', C, S'')
occurs(n, A) :-
    node_type(n, explanation_start),
    occurs(n-1, A).
% omitted: handling multi-character actions
```

the author utility must have increased.

The first node in an explanation, the *explanation start node*, models the perspective of the character whose action is being explained. Given the character `C` taking action `A` from state `S`, we take `beliefs(S, C, S')` and apply the effects of `A` to `S'`; the result becomes the base state for the explanation start node. A fragment of this process is illustrated in Listing 6. After the explanation start node, action nodes expand the explanation in the same manner as expanding the executed plan.

Figure 1 shows a subset of an example plan to illustrate the relationships between nodes and states. Node 0 has the initial state as its base state, including pointers to initial beliefs for the characters. In this story domain, we start out with the *teen* character at *home* and the *mom* character at the *party*, but the teen believes the mom is at *work* and the mom believes the teen is at *school*. Node 1 represents the result of an action where the teen goes to the *mall*. The base state is copied, except the location of the teen is updated in the new base state. Because the teen observes the action, the location update also takes place in the teen's beliefs; but because the mom does not observe the action, the mom's beliefs remain the same. Now the action must be explained for the teen. Node 2 is an explanation start node, representing the teen's expected result of the Node 1 action instead of the actual result; the base state is copied from the teen's original beliefs, except with the location update applied. Node 3 is another action node but it is part of the new explanation instead of the original author plan; after going to the mall, the teen intends to buy the *shoes*.

For each node within an explanation, there is one more type of state that is tracked outside of the base state and its belief states). These special states are responsible for tracking the greedy justification of the explained action, i.e., showing that the character plan would fail without that action and the later actions it enables. We use a constraint to reject solutions containing an action shown to be unnecessary for its own explanation (code fragments omitted for brevity).

6 Experiments

We benchmarked our ASP planner on a machine with 512 GB RAM and an Intel Xeon w3-2425 processor.

Fully Specified Initial State

As a baseline for comparing our ASP planner, we ran the Sabre benchmark suite by Ware and Farrell (2023) on our machine. We used the Ware and Farrell benchmarking tool to run Sabre 10 times each in different combinations of search strategy (goal-first, explanation-first, or A*) and heuristic (sum-graph, relaxed-plan-graph, or reachability).

Similarly, we parsed the problems from the benchmark suite into their ASP counterparts and ran our planner 10 times each for different built-in Clingo configurations (`auto`, `jumpy`, `handy`, `crafty`, and `tweety`).

Table 1 shows the results of this benchmarking. For each problem instance, the "auth. limit" and "char. limit" columns show the limits on author and character plan length, respectively, placed on both the Sabre and ASP planners. The "exp. nesting" column shows the maximum depth of explanations-within-explanations allowed—e.g., the author plan has depth 0, an explanation for a character's action in the author plan has depth 1, and an explanation for a character's action within that explanation has depth 2. We used the limits suggested in the original Sabre benchmark report.

The remaining columns show the average CPU times in seconds over a set of trials. The "def." columns show the times for the Sabre default planner configuration of A* search with a relaxed plan graph heuristic, and for the default Clingo configuration `auto`. The "best" columns show the times for the fastest configuration for each individual problem. ASP times include grounding as well as the main search process; grounding is a significant portion of Clingo's runtime because of the optimizations it makes during this phase.

Some problems from the benchmark suite are omitted: `aladdin`, `gramma`, `hospital`, `lovers`, and `western`. Our ASP planner was unable to solve these because the grounding size went beyond Clingo's built-in limit on the number of IDs that can be assigned to objects. Solving these problems in ASP would require either a more compact encoding or an answer set solver without this ID limit.

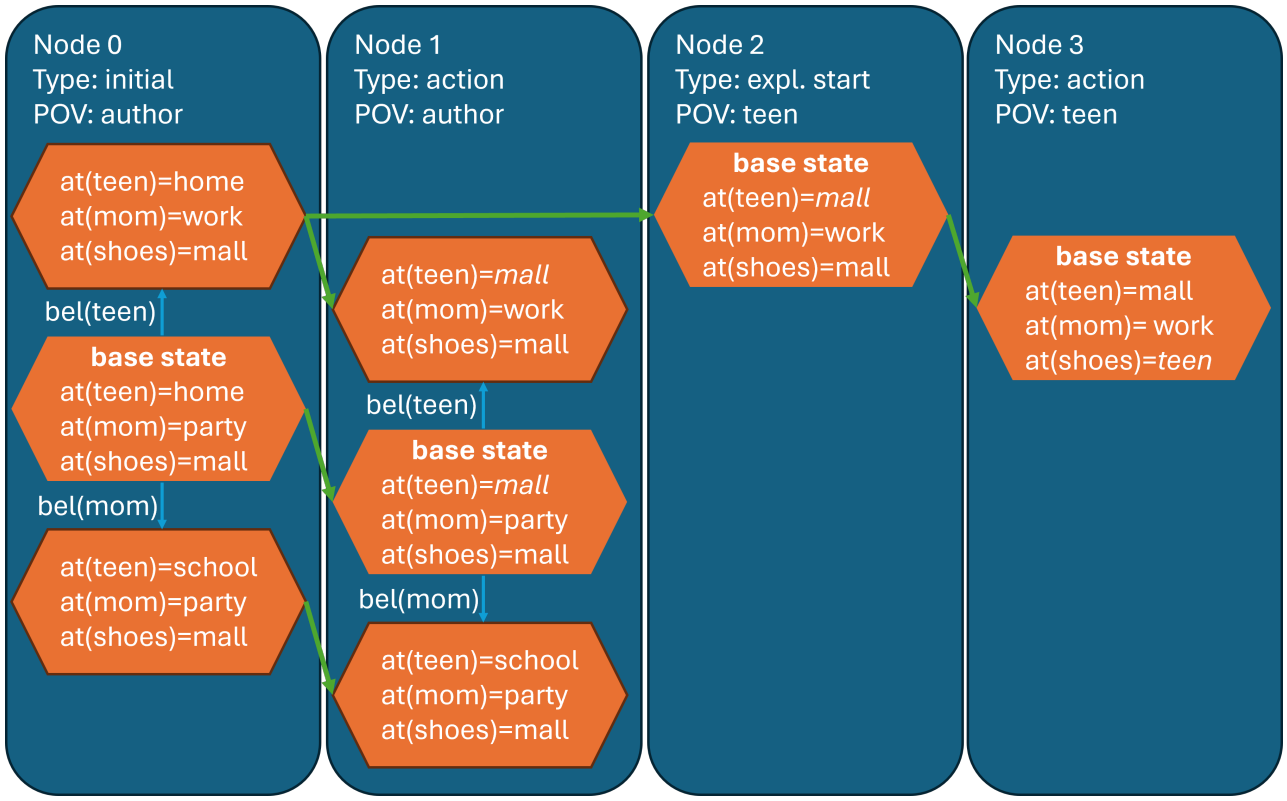

Figure 1: A conceptual illustration of the structure of nodes and states in a generated plan. Blue rounded rectangles represent nodes. Orange hexagons represent states and the fluent assignments are shown inside. Cyan arrows annotated *bel([character])* indicate a character's beliefs from a state; for brevity, only one layer of beliefs is shown. Green arrows indicate that a state copies its fluent values from another, except for values changed by actions, which are italicized.

Furthermore, the Sabre benchmark suite includes some variations of problems that require specific author utilities, but our benchmarking includes only versions of the problem that allow any author utility increase.

Among the benchmark problems solved successfully in ASP, the average ASP time for `basketball` and `raiders` was faster by a few seconds than Sabre given default configurations for both, but an optimal Sabre configuration was the fastest for all problems. The default Clingo configuration was faster than other Clingo configurations, except in the case of `deerhunter` and `fantasy` where `trendy` was the fastest and `raiders` where `jumpy` was the fastest.

Partially Specified Initial State

To explore our ASP planner's potential for handling problems with nondeterministic initial states, we created a set of synthetic benchmarks by modifying some of the original benchmarks from the previous section. We chose `basketball`, `jailbreak`, and `raiders` because of their original runtimes—short enough to collect larger volumes of data but long enough to more confidently attribute changes to solving complexity rather than overhead—as well as their structural similarities that we used to modify all of them in the same way.

Specifically, because these problems all focus on characters moving between locations and using items, we chose to create partially-specified initial states by removing specifications for the starting locations of characters and items. For each original problem, for each value of k from 1 to the number of location fluents, we generated ten new problem instances by randomly selecting k location fluents to leave unspecified in the initial state. The other fluent values were the same as in the original problem instance.

Figure 2 shows the average times taken to solve problems for each value of k using the Clingo default configuration. With the exception of the first fluents removed in `basketball`, runtimes showed a decreasing trend as more fluents had their initial value unspecified. This trend is the case because more general initial states permit a wider set of solutions, and the ASP solver was able to effectively take advantage of the new available solutions.

To examine this trend more closely, in Figure 3 we plot the unspecified initial fluent values against the average length of a solution in terms of our encoding's *nodes*, which correspond to actions, triggers, and plan or explanation initializations as discussed in Section 5. The node counts, like the runtimes, became smaller as the problem became more general. Although the trend is closer to linear than to exponential like in the case of runtime, this makes sense because

Table 1: Sabre and ASP benchmarking results. Times are in seconds.

problem	auth. limit	char. limit	exp. nesting	Sabre def.	Sabre best	ASP def.	ASP best
basketball	7	5	2	607.2	2.2	601.7	601.7
bribery	5	5	2	< 0.1	< 0.1	0.3	0.3
deerhunter	10	6	1	4.2	< 0.1	26567.8	17893.6
fantasy	9	3	2	4.0	0.9	2542.7	2155.2
jailbreak	7	6	1	20.3	0.2	15.4	15.4
raiders	7	4	1	0.3	< 0.1	1607.1	1327.4
secretagent	8	8	1	< 0.1	< 0.1	48722.1	48722.1
space	9	3	1	< 0.1	< 0.1	1.1	1.1
treasure	4	4	3	< 0.1	< 0.1	122.1	122.1

the ASP solver must explore an exponentially larger space of candidate plans as our planner iteratively increases the node limit; recall that in our multi-shot solving process we first determine whether there are any plans at node count 1 before moving on to node count 2 and so on. Therefore, the runtime savings for adding unspecified fluent values are likely a direct result of new solutions becoming available at shorter node limits.

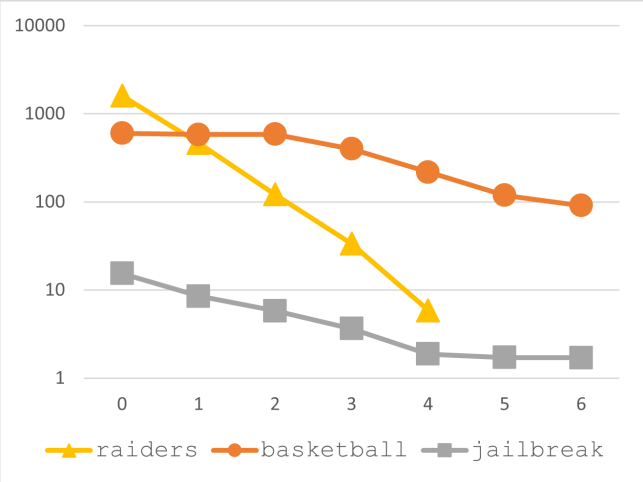

Figure 2: Number of unspecified initial fluent values (horizontal axis) versus average ASP solve time (vertical axis; note the logarithmic scale). There are fewer data points for raiders because there are fewer fluents whose initial value specifications could be removed.

Our data collection shows that, when the domain author will accept a variety of initial states, combining them all into one most general initial state can offer computational savings over committing to initial states individually. However, this approach creates a risk of generating stories that are unsatisfying narratively. For instance, in the jailbreak domain, one possible end condition is the police catching the protagonist with contraband; when the solver could choose most or all starting character and item locations, it would start the story with the protagonist holding a smuggled weapon directly in front the police, allowing the police to catch them and end the story in a single action.

Future development of nondeterministic narrative planners would be best supported by new benchmarks that can be varied in the level of initial state specification like ours, but are designed to prevent initial states that have trivial solutions. However, there are also research opportunities for placing the responsibility on the planner itself to evaluate story quality and select stories that satisfy structural requirements other than simply goal achievement.

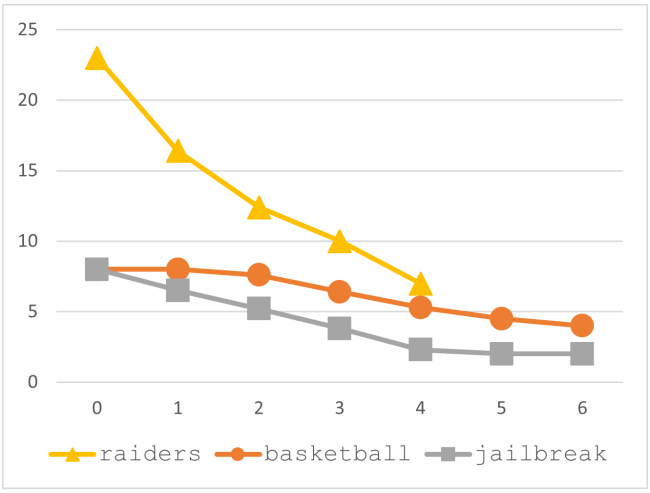

Figure 3: Number of unspecified initial fluent values (horizontal axis) versus average nodes in the solution (vertical axis).

7 Limitations

The primary limitation of our approach is its computational intensity, with most solve times for standard problems being orders of magnitude slower than their Sabre counterparts. Although it offers a modifiable starting point for exploration of new problem definitions, our implementation is not competitive in its current form as a solver for well-studied narrative planning formulations. It is not yet clear whether specialized planners have an inherent advantage over ASP for speed or if optimizations to our ASP planner could eventually help it become competitive; in a classical planning setting, custom ASP solver configurations have been able to produce orders-of-magnitude speedups over the base configuration (Gebser et al. 2013).

8 Conclusions

This paper introduced the first combination of ASP with a Sabre-style narrative planning model. The synthesis brings a nested theory-of-mind model to answer set narrative planning. Conversely, it brings nondeterministic initial state to Sabre-style planning, and is the first implementation of nondeterministic narrative planning to handle the original Ware and Young (2010) vision of a most general initial state as an arbitrary proposition. The declarative format simplifies the process of modifying the narrative planning model; the domain-independent portion of the ASP encoding consists of about 115 unground rules, fewer than the number of *classes* in the imperative Sabre code. This allows narrative planning researchers to prototype new features before implementing them in a conventional planner, and allows domain authors to add simpler requirements *ad hoc*; for instance, a PDDL-style state trajectory constraint (Porteous and Cavazza 2009) can be placed on plans with the addition of a single unground rule.

We found that Sabre itself outperforms our ASP planner on established benchmark problems. However, it is possible that the speed of specialized planners and the versatility of ASP could be combined using a hybrid approach. For instance, in a conventional narrative planner modified to handle initial states that are only partially specified, an ASP encoding could be used for plan verification rather than plan generation: The planner's original algorithm explores candidate plans, and the ASP encoding is used to check whether there exists a completion of the initial state where plan constraints like action legality are met. A similar approach has been used successfully with SAT encodings in nondeterministic variations of classical planning (Hoffmann and Brafman 2006).

Integrating nondeterministic planning with action explanations raises new challenges. First, once a deterministic narrative planner explains an action, the explanation is guaranteed to be usable in the eventual solution; but in nondeterministic planning, an explanation for one action may later be invalidated because it is not compatible with any of the same initial states as the explanations for another action. Second, many planning heuristics that are effective for deterministic planning are undefined in the nondeterministic setting and effective replacements must be found. Our future work includes investigating the viability of the hybrid approach and developing these heuristics.

Acknowledgments

This paper benefited from feedback from Gage Birchmeier and the AIIDE program committee; code from Mira Fisher; and a discussion on ASP representations with Chinmaya Dabral, Chris Martens, Adam Smith, and David Thue.

This material is based upon work supported by the U.S. National Science Foundation under Grant No. 2145153 and the U.S. Army Research Office under Grant No. W911NF-24-1-0195. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Army Research Office.

References

Bae, B.-C.; and Young, R. M. 2008. A use of flashback and foreshadowing for surprise arousal in narrative using a plan-based approach. In *International Conference on Interactive Digital Storytelling*, volume 1, 156–167.

Birchmeier, G.; and Ware, S. G. 2025. Speeding up narrative planning with Causal Width Search and Pruning. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 21.

Brewka, G.; Eiter, T.; and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM*, 54(12): 92–103.

Burigana, A.; Fabiano, F.; Dovier, A.; and Pontelli, E. 2020. Modelling multi-agent epistemic planning in ASP. *Theory and Practice of Logic Programming*, 20(5): 593–608.

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Maratea, M.; Ricca, F.; and Schaub, T. 2020. ASP-Core-2 input language format. *Theory and Practice of Logic Programming*, 20(2): 294–309.

Cardona-Rivera, R. E.; and Li, B. 2016. PLOTSHOT: Generating discourse-constrained stories around photos. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 12, 2–8.

Chen, S.; Smith, A. M.; Jhala, A.; Wardrip-Fruin, N.; and Mateas, M. 2010. RoleModel: Towards a formal model of dramatic roles for story generation. In *Intelligent Narrative Technologies Workshop*, volume 3.

Christensen, M.; Nelson, J.; and Cardona-Rivera, R. 2020. Using domain compilation to add belief to narrative planners. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, 38–44.

Dabral, C.; and Martens, C. 2020. Generating explorable narrative spaces with answer set programming. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, 45–51.

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

Fink, E.; and Yang, Q. 1992. Formalizing plan justifications. In *Conference of the Canadian Society for Computational Studies of Intelligence*, volume 9, 9–14.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Lühne, P.; Obermeier, P.; Ostrowski, M.; Romero, J.; Schaub, T.; Schellhorn, S.; and Wanko, P. 2018. The Potsdam Answer Set Solving Collection 5.0. *Künstliche Intelligenz*, 32: 181–182.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, 19(1): 27–82.

Gebser, M.; Kaufmann, B.; Romero, J.; Otero, R.; Schaub, T.; and Wanko, P. 2013. Domain-specific heuristics in answer set programming. In *AAAI Conference on Artificial Intelligence*, volume 27, 350–356.

Göbelbecker, M.; Keller, T.; Eyerich, P.; Brenner, M.; and Nebel, B. 2010. Coming up with good excuses: What to do when no plan can be found. In *International Conference on Automated Planning and Scheduling*, volume 20, 81–88.

Hoffmann, J.; and Brafman, R. I. 2006. Conformant planning via heuristic forward search: A new approach. *Artificial Intelligence*, 170(6-7): 507–541.

Horswill, I. 2021. Answer set programming for PCG: The good, the bad, and the ugly. In *Workshop on Programming Languages in Entertainment*.

Horswill, I. 2022. Retcon: A least-commitment storyworld system. In *Experimental AI in Games Workshop*, volume 9.

Horswill, I.; and Hill, S. 2024. Fast, declarative, character simulation using bottom-up logic programming. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 20, 54–64.

Kautz, H.; and Selman, B. 1992. Planning as satisfiability. In *European Conference on Artificial Intelligence*, volume 10, 359–363.

Kumaran, V.; Rowe, J.; and Lester, J. 2024. NarrativeGenie: Generating narrative beats and dynamic storytelling with large language models. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 20, 76–86.

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2): 39–54.

Martens, C. 2015. Ceptre: A language for modeling generative interactive systems. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 11, 51–57.

Martens, C.; Ferreira, J. F.; Bosser, A.-G.; and Cavazza, M. 2014. Generative story worlds as linear logic programs. In *Intelligent Narrative Technologies Workshop*, volume 7.

Martin, L. J. 2021. *Neurosymbolic automated story generation*. Ph.D. thesis, Georgia Institute of Technology.

Mateas, M.; and Sengers, P. 1999. Narrative intelligence. In *AAAI Fall Symposium on Narrative Intelligence*, 1–10.

Porteous, J.; and Cavazza, M. 2009. Controlling narrative generation with planning trajectories: the role of constraints. In *International Conference on Interactive Digital Storytelling*, volume 2, 234–245.

Reiter, R. 1996. Natural actions, concurrency and continuous time in the situation calculus. In *International Conference on Principles of Knowledge Representation and Reasoning*, volume 5, 2–13.

Riedl, M. O.; and Young, R. M. 2005. Open-world planning for story generation. In *International Joint Conference on Artificial Intelligence*, volume 19, 1719–1720.

Riedl, M. O.; and Young, R. M. 2006. Story planning as exploratory creativity: Techniques for expanding the narrative search space. *New Generation Computing*, 24(3): 303–323.

Riedl, M. O.; and Young, R. M. 2010. Narrative planning: Balancing plot and character. *Journal of Artificial Intelligence Research*, 39(1): 217–268.

Sanghrajka, R.; Young, R. M.; and Thorne, B. 2022. HeadSpace: Incorporating action failure and character beliefs into narrative planning. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 18, 171–178.

Senanayake, L.; and Ware, S. G. 2025. Speeding up narrative planning using Fog of War Pruning. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 21.

Shirvani, A.; Ware, S.; and Farrell, R. 2017. A possible worlds model of belief for state-space narrative planning. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 13, 101–107.

Siler, M.; and Ware, S. G. 2020. A good story is one in a million: Solution density in narrative generation problems. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, 123–129.

Son, T. C.; Pontelli, E.; Balduccini, M.; and Schaub, T. 2023. Answer set planning: A survey. *Theory and Practice of Logic Programming*, 23(1): 226–298.

Subrahmanian, V.; and Zaniolo, C. 1995. Relating stable models and AI planning domains. In *International Conference on Logic Programming*, volume 12, 233–247.

Sun, Y.; Li, Z.; Fang, K.; Lee, C. H.; and Asadipour, A. 2023. Language as reality: A co-creative storytelling game experience in 1001 Nights using generative AI. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 19, 425–434.

Teutenberg, J.; and Porteous, J. 2015. Incorporating global and local knowledge in intentional narrative planning. In *International Conference on Autonomous Agents and Multiagent Systems*, 1539–1546.

Wang, Y.; and Kreminski, M. 2025. Can LLMs generate good stories? Insights and challenges from a narrative planning perspective. arXiv preprint arXiv:2506.10161.

Ware, S. G.; and Farrell, R. 2023. A collection of benchmark problems for the Sabre narrative planner. Technical report, Narrative Intelligence Lab, University of Kentucky.

Ware, S. G.; and Siler, M. 2021. Sabre: A narrative planner supporting intention and deep theory of mind. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 17, 99–106.

Ware, S. G.; and Young, R. M. 2010. Rethinking traditional planning assumptions to facilitate narrative generation. In *AAAI Fall Symposium on Computational Models of Narrative*, 71–72.

Ware, S. G.; Young, R. M.; Harrison, B.; and Roberts, D. L. 2014. A computational model of narrative conflict at the fabula level. *IEEE Transactions on Computational Intelligence and Artificial Intelligence in Games*, 6(3): 271–288.

Young, R. M.; Ware, S. G.; Cassell, B. A.; and Robertson, J. 2013. Plans and planning in narrative generation: A review of plan-based approaches to the generation of story, discourse and interactivity in narratives. *Sprache und Datenverarbeitung, Special Issue on Formal and Computational Models of Narrative*, 37(1-2): 41–64.

Zubek, R.; Horswill, I.; Robison, E.; and Viglione, M. 2021. Social modeling via logic programming in *City of Gangsters*. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 17, 220–226.