Ceptre: A Language for Modeling Generative Interactive Systems

Chris Martens
Carnegie Mellon University
cmartens@cs.cmu.edu

Abstract

We present a rule specification language called Ceptre, intended to enable rapid prototyping for experimental game mechanics, especially in domains that depend on procedural generation and multi-agent simulation.

Ceptre can be viewed as an explication of a new methodology for understanding games based on *linear logic*, a formal logic concerned with resource usage. We present a correspondence between *gameplay* and *proof search* in linear logic, building on prior work on generating narratives. In Ceptre, we introduce the ability to add interactivity selectively into a generative model, enabling inspection of intermediate states for debugging and exploration as well as a means of play.

We claim that this methodology can support game designers and researchers in designing, anaylzing, and debugging the core systems of their work in generative, multi-agent gameplay. To support this claim, we provide two case studies implemented in Ceptre, one from interactive narrative and one from a strategy-like domain.

Introduction

Today, game designers and developers have a wealth of tools available for creating executable prototypes. Freely-available engines such as Twine, Unity, PuzzleScript, and Inform 7 provide carefully-crafted interfaces to creating different subdomains of games. On the other hand, to invent interesting, novel mechanics in any of these tools typically requires the use of ill-specified, general purpose languages: in Twine, manipulating state requires dipping into JavaScript; in Unity, specifying any interactive behavior requires learning a language such as C# or JavaScript; Inform 7 contains its own general-purpose imperative, functional, and logic programming languages; and PuzzleScript simply prevents the author from going outside the well-specified 2D tile grid mechanisms.

The concept of *operational logics* (Mateas and Wardrip-Fruin 2009), the underlying substrate of meaningful state-change operations atop which mechanics are built, was recently proposed as a missing but critical component of game design and analysis. In most prototyping tools, there is a fixed operational logic (e.g. tile grids and layers in PuzzleScript, or text command-based navigation of a space in Inform 7) atop which the designer may be inventive in terms of *rules* and *appearance*, but all rules must be formed out of entities that mean something in the given operational logic (or use a much more complex language to subvert that default logic). We lack good specification tools for inventing *new* operational logics, or combining elements from several. In particular, many novel systems of play arise from the investigation of interactive storytelling (Mateas and Stern 2003), multi-agent social interaction (McCoy et al. 2010), and procedurally generated behavior (Hartsook et al. 2011), and no existing tools make it especially straightforward for a novice developer to codify and reason about those systems.

Ceptre is a proposal for an operational-logic-agnostic specification language that may form the basis of an accessible front-end tool. It is based on a tradition of *logical frameworks* (Harper, Honsell, and Plotkin 1993), which use logical formulas to represent the rules of a system (a *specification*). Then proof search may be used to simulate execution, answer queries, and perform analysis of the specification. In Ceptre, we specify games, generative systems, and narratives with similar payoff. Our logic of choice for representing games is *linear logic* (Girard 1987), unique among logics in its ability to model state change and actions without the need for a *frame rule* or other axiomatization of inertia for predicates that do not change in an action.

Using linear logic to specify a space of play closely resembles planning approaches. Planning has been used extensively to study games, especially in the interactive storytelling domain (Mateas and Stern 2003; Porteous, Cavazza, and Charles 2010; Medler and Magerko 2006) but also as a general mechanic description language (Zook and Riedl 2014). In contrast to this work, we position Ceptre as unique among game description languages in its combination of (a) direct correspondence to a pre-existing logic and proof theory, providing a portable and robust foundation for reasoning about games, and (b) use as a native authoring language, rather than as a library or auxiliary tool.

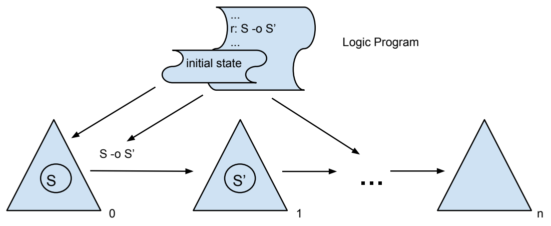
On the other hand, unlike prior executable description languages (e.g. (Dormans 2011; Osborn, Grow, and Mateas 2013)), we retain the planning-like ability to specify generalized rule schema that apply in a multi-agent setting, allowing for causal analysis of event sequences. These statements can be made more meaningful after an introduction to the language, so we postpone further discussion of related work to a later section. In the remainder of the paper, we illustrate the semantics and utility of Ceptre through two case studies: a generative story world and a skeletal dungeon-crawling adventure game.

Specifying Mechanics in Ceptre

Ceptre affords a game designer with three central mechanisms for authorship: (1) a term language in which to describe the game's primitive constructs and predicates that range over them; (2) a way to write local state transition rules with *replacement* semantics; (3) a way to delineate and coordinate separate components of the game with a system of *stages*. We recapitulate an interactive storytelling example used in prior work (Martens, Ferreira, and Bosser 2014) to demonstrate how Ceptre programs are structured and interpreted.

Bird's-Eye View

A Ceptre program is a collection of terms, predicates, and rules Σ, along with a specification of an initial state configuration Δ_0. To a first approximation, the execution of this program means repeatedly examining the current state configuration Δ_i (starting with Δ_0), selecting a rule r that applies to a subset S of Δ_i, and generating a new Δ_{i+1} which is Δ_i with S replaced by the rule's consequences S'. The rules in Σ are unordered, and several may apply, potentially giving rise to nondeterminism.



In the interactive storytelling domain, we can make an analogy between program execution and a dramatic production: the initial state will "set the scene" for the first act, and the engine itself works as a hybrid playwright and stage manager, looking up rules to determine the next scene–several of which might apply–then issuing cues to the actors involved. The role of the program author in this analogy is to describe the means by which one scene can transition to another. How to do so is described next.

Predicates and Rules

First, we need a specification of the primitives and what types of terms they operate over. We refer to this part of the specification as the *type header*. In the story domain, the types include characters, locations, and physical objects; the predicates include character locations, sentiments between characters such as affection and anger, possessions, and death. These predicates represent a subset of those used in the full domain. Here is a sample of the type header for this domain:

```
at character location : pred.
likes character character :  pred.
anger character character : pred.
has character object : pred.
depressed character : pred.
dead character : pred.
```

Next, we specify how the story state may evolve via rules of the form `rulename : A -o B`, where `A` and `B` each have the form p_1`*`...`*`p_n, where the p_is are predicates drawn from the above set. Predicates may contain variables that range over the term type corresponding to that predicate index. The meaning of `A -o B`, to a first approximation, is that whenever the predicates in `A` are present, they may be replaced with `B`. One example of a rule is:

```
do/compliment
  : at C L * at C' L * likes C C'
 -o at C L * at C' L * likes C C' * likes C' C.
```

The name of the rule is `do/compliment`—we'll prefix our rule names with `do/` in this example, by convention. The rest of the rule can be parsed as follows: `at C L` is a predicate applied to two variables `C` and `L` which must have the types `character` and `location`, as specified in the type header. Variables are indicated by starting with a capital letter, and they are implicitly quantified at the beginning of the rule, i.e. the above rule is really a *rule schema* that may be instantiated at any `C`, `C'`, and `L`. The logical operators in this rule are `*` and `-o`, pronounced *tensor* and *lolli*. The `*` operator conjoins predicates, and `-o` is the transition operator. Semantically, this rule gives meaning to a *compliment* action by saying that whenever a character likes another who shares their location, they may generate a resource representing affection in the other direction.

Note that because of the *replacement* semantics of the rule, we need to reiterate everything on the right-hand side of the `-o` that we don't want to disappear, such as the character locations and original `likes` fact. We use the syntactic sugar of prepending `$` to anything intended not to be removed in order to reduce this redundancy:

```
do/compliment
  : $at C L * $at C' L * $likes C C' -o likes C' C.
```

A more complex rule describes a `murder` action, using the `!` operator to indicate a permanent state:

```
do/murder
: anger C C' * anger C C' * anger C C' * anger C C' *
  $at C L * at C' L  * $has C weapon
              -o !dead C'.
```

(This rule consumes `C'`'s location, maintaining a global invariant that each character is mutually exclusively `at` a location or `!dead`.) Here we see a departure from planning formalisms: four instances of `anger C C'` mean something different from one. Here we are using an emotion not just as a precondition but as a resource, where if we have enough of it, we can exchange it for a drastic consequence. Whether or not we diffuse the anger, or choose to keep it by prepending `$` to the predicates, is an authorial choice. It may not matter, if there is nothing that makes *use* of a character's anger toward a dead character, but perhaps such rules could be narratively interesting (a monologue of resentful memories, perhaps). We do include a similar rule for grieving:

```
do/grieve : $at C L * likes C C' * dead C'
       -o depressed C * depressed C.
```

Initial States

While the rules of the story-world are parameterized over characters and locations, we must provide specific instances of these domains in order to apply the rules and run the program. For instance, we may declare that `juliet` is a character and `town` is a location. A *ground* predicate is one that contains no variables (e.g. `at juliet town` rather than `at C L`). A well-formed program's initial state is a multiset of ground predicates, and well-formed program rules maintain groundness of this set. That is, the variables mentioned on the right-hand side of a transition should be a subset of those that appear on the left. Here is a partial example of an initial state (called a "context" in Ceptre) from our story-world domain:

```
context init =
{ at romeo town, at montague mon_house, at capulet cap_house,
  at mercutio town, at nurse cap_house, at juliet town,
  at tybalt town, at apothecary town,

  anger montague capulet, anger capulet montague,
  anger tybalt romeo, anger capulet romeo, anger montague tybalt,

  likes mercutio romeo, likes romeo mercutio,
  likes montague romeo, likes capulet juliet,

  has tybalt weapon, has romeo weapon, has apothecary weapon }
```

At this point, we have described something that Ceptre can interpret as a runnable program, evolving the initial state forward by any rules in the signature that apply, using the directive `#trace init.` Note that this program is highly nondeterministic: several rules apply in any given world state, and the programmer must make no assumptions about which rule is selected when several apply – in other words, *rule ordering* is not part of the language semantics.[1] By default, the nondeterminism in rule selection manifests as uniformly random selection by the engine among all rules that apply. We can use this specification to randomly generate stories if we also provide termination conditions (story endings), but if we supplant the nondeterminism in the program with human interaction, we simply have an infinite game.

Interactive Rulesets

Ceptre offers a mechanism to replace the default random nondeterminism with a choice from an external source, e.g. standard input. The syntax for adding interaction is to wrap all of the rules in a *stage*, then add an `#interactive` directive:

```
stage allrules = {
  do/compliment : ...
  ...
}
#interactive allrules.
```

Upon running this program with the above initial state, a user is first prompted with these choices:

```
0: (do/insult/witnessed tybalt town romeo mercutio)
1: (do/insult/private tybalt town romeo)
2: (do/compliment/witnessed mercutio town romeo tybalt)
3: (do/compliment/private romeo town mercutio)
4: (do/compliment/private mercutio town romeo)
5: (do/formOpinion/dislike mercutio town juliet)
6: (do/formOpinion/dislike juliet town mercutio)
7: (do/formOpinion/like mercutio town juliet)
8: (do/formOpinion/like juliet town mercutio)
9: (do/travelTo montague romeo town mon_house)
10: (do/travelTo capulet juliet town cap_house)
11: (do/travelTo juliet nurse cap_house town)
12: (do/travelTo nurse juliet town cap_house)
```

New choices are generated based on the prior selection and its change to the state. The state as it evolves is written to a log file, which can be read in a separate text buffer to inform player decisions. (Ideally, these input and output mechanisms would be hooked up to a more intelligible, customizable user interface.)

Stages

The above interactive story-world has some peculiar features: for instance, it appears that we give the player control over not just characters' actions but also their *reactions*, which we would prefer to think of as involuntary, such as the `do/formOpinion` rules. We would like to for some of these rules to run automatically without player intervention. In our next iteration of the program, we will make use of a Ceptre feature called *stages*. Stages are a way of structuring a program in terms of independent components. Syntactically, a stage is a curly-brace-delimited set of rules with an associated name. Semantically, a stage is a unit of computation that runs to *quiescence*, i.e. no more rules are able to fire, at which point control may be transfered to another stage. The outer structure of our new program will be:

```
stage act = {
  % User-selected rules.
}
qui * stage act -o stage react.
stage react = {
  % Involuntary, reactive rules.
}
qui * stage react -o stage act.
```

The only new piece of syntax here is the `qui` predicate, which is a special token denoting quiescence of the program. All outer-level rules must have this form: upon quiescence, replace one `stage` resource with another (and possibly delete or add some additional state).

In this formulation of the game, the player may act arbitrarily many times before reactions are calculated (and in fact must do so until quiescence of the act stage). We can make a modification, editing each rule in the stage to accept one additional premise `tok`, which is generated whenever control is transfered to the stage. Doing so will offer one "turn" of input between potentially many steps of reaction computation. Turn tokens may be introduced to the reactive side of the program as well, if we wish to carry out a bounded number of reactive steps before returning control to the player. Thus concludes our first example of modeling an interactive, generative scenario in Ceptre.

Proofs as Traces

Having explained the central constructs of the programming language, we now introduce the theoretical motivation behind them: the central correspondence with proof theory mentioned in the introduction.

[1] Omitting rule ordering is by design: linear logic makes a firm commitment to *locality* in that a single rule's meaning does not depend on any other rules.

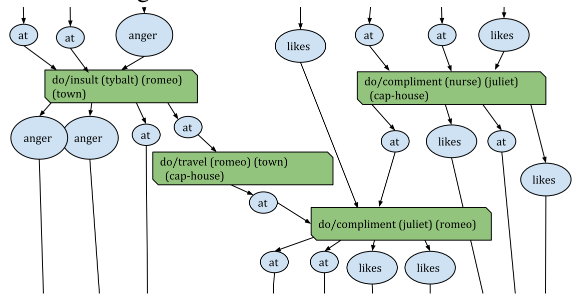
We refer to the standard logical notion of a *sequent* $\Delta \vdash A$, pronounced "Delta entails A." Δ stands for a state, i.e. a multiset of facts, as before, and A is some *goal formula*. Such a sequent is said to be *derivable* in linear logic if, using the sequent-based definition of linear logic's connectives (`*`, `-o`, and `!`), we can form a complete proof tree with that sequent as its root. A proof tree is either a leaf of the form $A \vdash A$ or an instance of a *left* or *right* rule that decomposes a connective on the corresponding side of the $\vdash$. For additional background on the semantics of logical sequents as they relate to interactive narratives, see (Bosser, Cavazza, and Champagnat 2010).

Sequent calculi are operationalized as programming languages by way of logic progamming, which leverages *computation as proof search* (Hodas and Miller 1994). A logic programming engine takes a sequent as input and creates a proof as output, or declares failure. The *process* of creating the proof can be computationally relevant, however. Since each connective can be decomposed via both left and right rules, the engine must decide whether to work on the right or the left at any given point. Analyzing the right-hand side of the sequent and choosing which rules to try based on it is called *backward chaining* or *goal-driven* proof search. On the other hand, looking to the left of the sequent for which currently-known facts produce new information is called *forward chaining* or *assumption-driven* search. It is the forward chaining process that allows Ceptre to model simulations: the transition semantics $\Delta \rightarrow \Delta'$ means that a proof step from the goal sequent $\Delta \vdash X$ to the sequent $\Delta' \vdash X$ is possible no matter what X is. Thus we can execute `#trace` directives on states without also specifying a goal, and choosing which rules to apply corresponds to moving "forward in time" with respect to the evolving proof state.

Causal Structure

The structure of the resulting proof depends on the search strategy used. For a *purely forward-chaining* strategy, the proof can be compactly represented as a sequence of rule applications, each of the form let $\langle x_1, \ldots, x_n \rangle = r\ y_1, \ldots y_m$, where the ys represent resources used (generated by prior rules) and the xs represent resources produced. (The r variable stands for the name of an author-defined rule, such as `do/compliment`.) Two rule applications that consume disjoint sets of resources from the same state can be said to happen *concurrently*, or independently. On the other hand, a rule that produces resources and another that consumes a subset of them can be said to be in a causal, or dependent, relationship. Less abstractly, if resources represent facts associated with particular game entities or characters, then independent rule applications represent potentially *concurrent* action by multiple such entities, and causally related rule applications represent either sequential action by a single actor, or synchronized interaction between two entities.

Ceptre program executions (modeled after the Celf logical framework (Schack-Nielsen and Schürmann 2008)) are recorded in a *trace* that keeps track of this dependency information. At the end of a single playthrough, the trace contains not just a linear record of which rules were applied, but a *structured term* giving rise to a directed, acyclic graph between rule applications, mapping onto their causal relationships. For instance, the story world described previously, run as a non-interactive simulation, may produce a trace containing the following structure:


at
at
anger
likes
at
at
likes
do/insult (tybalt) (romeo) (town)
do/compliment (nurse) (juliet) (cap-house)
anger
anger
at
at
at
likes
at
do/travel (romeo) (town) (cap-house)
likes
at
do/compliment (juliet) (romeo)
at
at
likes
likes

This image depicts a fragment of a structured trace in which two "scenes," one with Romeo and Tybalt and one with the Nurse and Juliet, happen concurrently, followed by Romeo traveling to the Capulet house to participate in a scene with Juliet. The resource dependencies that influence this flow of action are shown in the oval-shaped nodes. A trace of this form can be seen as describing the *conjunctive* branching present in the simulation, i.e. which events may happen together by partioning the state, while the original logic program additionally contains *disjunctive* branching, i.e. events that compete over common resources and thus represent alternatives in the flow of time.

The ability to extract this structure from a playthrough or generated artifact leads to several potential uses:

1. Debugging. If we want to understand why a rule fired despite our expectations, we have a complete causal record of its selection.
2. Analyzing feedback loops in game mechanics. If we unify all instances of a rule being fired in a trace, the resulting graph tells us which rules tend to directly precede or influence others, and cycles in the condensed graph may be analyzed for their cummulative or productive effects.
3. Experimental generative hypertext. From a given action node, each out-edge to a distinct action node may be interpreted as a link in a hypertext game representing the player "following" subsets of characters in the scene. (This interpretation was influenced by the experimental theatre piece *Tamara* (Krizanc 1997), in which audience members may choose to follow characters to different rooms when they exit a scene.) This idea has been implemented at `http://play.typesafety.net/`.

In summary, the connection between proof search and gameplay offers a new theoretical foundation for the computational nature of games, and bears a number of practical uses as a side effect.

Second Case Study: Dungeoncrawler

Next, to illustrate Ceptre's breadth, we examine a small dungeon-crawler-like game specification. This game will have three top-level actions: adventuring, resting, and shopping; and three central resource types: health, money, and weapons. In this setting, it is convenient to be able to perform arithmetic directly on resource quantities, so we will represent them numerically rather than through logical resources.

Ceptre allows the definition of permanent facts via backward-chaining predicates. For instance, we can define the arithmetic operation of addition capped at a certain maximum value as a predicate `cplus A B Cap C` which can be read as `A` plus `B` capped at `Cap` is `C`. We omit the definition of the predicate for brevity, but make use of it in later rules. We also use backward-chaining predicates to define a few constants, such as the player's maximum health and the damage and cost of various weapons:

```
max_hp 10.  damage sword 4.  cost sword 10.
```

We then define a initial context and an initial stage that sets up the game's starting state:

```
context init_ctx = {init_tok}.
stage init = {
  i : init_tok * max_hp N
   -o health N * treasure z * ndays z * weapon_damage 4.
}
```

We define the rest of the game using a "screen" idiom, with predicates representing the main, rest, adventure, and shop screens. (Some type header information is omitted.)

```
qui * stage init -o stage main * main_screen.

stage main = {
  do/rest : main_screen -o rest_screen.
  do/adventure : main_screen -o adventure_screen.
  do/shop : main_screen -o shop_screen.

  do/quit : main_screen -o quit.
}
#interactive main.

qui * stage main * $rest_screen -o stage rest.
qui * stage main * $shop_screen -o stage shop.
qui * stage main * $adventure_screen -o stage adventure.
qui * stage main * quit -o ().
```

The `rest` and `shop` stages allow recharging health (at the cost of an increment to the number of days) and upgrading one's weapon damage in exchange for treasure, respectively:

```
stage rest = {
  recharge : rest_screen
              * health HP * max_hp Max * recharge_hp Recharge
              * cplus HP Recharge Max N
              * ndays NDAYS
             -o health N * ndays (NDAYS + 1).
}
qui * stage rest -o stage main * main_screen.

stage shop = {
  leave : shop_screen -o main_screen.
  buy : treasure T * cost W C * damage_of W D * weapon_damage _
         * subtract T C (some T')
         -o treasure T' * weapon_damage D.
}
#interactive shop.
qui * stage shop * $main_screen -o stage main.
```

The adventure stages are the most complex part of the code, involving the random generation of a monster and random spoils are collected upon player victory. Spoils are only added to the treasure bank if the player does not flee from a fight in progress. To do all of this, we need an adventure initialization stage (`init`), a monster generating stage (`fight_init`), a way of responding to player actions in context (`fight_auto`), and a choice for the player between fighting and fleeing (`fight`):

```
stage adventure = {
  init : adventure_screen -o spoils z.
}
qui * stage adventure -o stage fight_init * fight_screen.

% drop_amount M N means a monster of size M can drop N coins
drop_amount nat nat : bwd.
drop_amount X X. % for now

stage fight_init = {
  init : fight_screen -o gen_monster * fight_in_progress.
  gen_a_monster : gen_monster * monster_size Size
                  -o monster Size * monster_hp Size.
}
qui * stage fight_init -o stage fight * choice.

try_fight : pred.
fight_in_progress : pred.
stage fight_auto = {
  fight/hit
    : try_fight * $fight_in_progress * monster_hp MHP * $weapon_damage D
        * subtract MHP D (some MHP') -o monster_hp MHP'.
  win
    : fight_in_progress * monster_hp MHP * $weapon_damage D
        * subtract MHP D none -o win_screen.
  fight/miss
    : try_fight * $fight_in_progress * $monster Size * health HP
          * subtract HP Size (some HP') -o health HP'.
  die_from_damages
    : health z * fight_in_progress -o die_screen.
  fight/die
    : try_fight * fight_in_progress * monster Size * health HP
          * subtract HP Size none -o die_screen.
}
choice : pred.
qui * stage fight_auto * $fight_in_progress -o stage fight * choice.
qui * stage fight_auto * $win_screen -o stage win.
qui * stage fight_auto * $die_screen -o stage die.

stage fight = {
  do_fight : choice * $fight_in_progress -o try_fight.
  do_flee  : choice * fight_in_progress -o flee_screen.
}
#interactive fight.
qui * stage fight * $fight_in_progress -o stage fight_auto.
qui * stage fight * $flee_screen -o stage flee.
```

Choosing `flee` takes you back to the main screen without any spoils:

```
stage flee = {
  % lose spoils
  do/flee : flee_screen * spoils X * monster _ * monster_hp _
            -o ().
}
qui * stage flee -o stage main * main_screen.
```

Finally, we need stages for winning and dying in combat:

```
go_home_or_continue : pred.
stage win = {
  win : win_screen * monster Size * drop_amount Size Drop
        -o drop Drop.
  collect_spoils : drop X * spoils Y * plus X Y Z
      -o spoils Z * go_home_or_continue.
  go_home : go_home_or_continue
      * spoils X * treasure Y * plus X Y Z
      -o treasure Z * main_screen.
  continue : go_home_or_continue -o fight_screen.
}
#interactive win.
qui * stage win * $main_screen -o stage main.
qui * stage win * $fight_screen -o stage fight_init.

end : pred.
stage die = {
  quit : die_screen -o end.
  restart : die_screen * monster_hp _
            * spoils _ * ndays _ * treasure _
            * weapon_damage _ -o init_tok.
}
#interactive die.
```

```
qui * stage die * end -o ().
qui * stage die * $init_tok -o stage init.
```

The program can then be run with the directive `#trace _ init init_ctx`.

The development of this example benefited extensively from the ability to specify interactivity modularly, occasionally making non-player-directed stages (like `fight_auto`) interactive to debug them. Additionally, we can test the design by "scripting" certain player strategies. For instance, we could augment the two rules in the `fight` stage to be deterministic, fighting when the monster can't kill us in one turn and fleeing otherwise:

```
stage fight = {
  do_fight : choice * $fight_in_progress * $monster Size
    * $health HP * Size < HP -o try_fight.
  do_flee  : choice * fight_in_progress * $monster Size
    * $health HP * Size >= HP -o flee_screen.
}
```

If we remove interactivity from this stage, then we get automated combat sequences that should never result in the player's death.

In summary, this case study illustrates the use of stages in Ceptre to program system interactions between multiple computational and human decision procedures in a strategy game setting.

Related Work

Linear logic has been used previously to study games, especially in the interactive storytelling domain (Dang et al. 2011; Champagnat, Prigent, and Estraillier 2005; Bosser, Cavazza, and Champagnat 2010). For instance, Dang et al. model story choices as linear logic propositions and use a theorem prover to validate the space of possible story outcomes. Related formalisms, such as Petri nets (Araújo and Roque 2009) and the Machinations framework (Dormans 2011), have been used to similar extent. These investigations are all limited to *atomic propositions*, however, meaning that a predicate like `at(C,L)` must be instantiated at every character and location to make use of it in the model, and rules cannot be described generically over such entities. Handling generic rule schema requires an extension to first-order logic, which we have included in Ceptre (but consequently we lose decidability of proof search in the language, so applications like exhaustive scenario analysis are not within scope of our project).

Several other systems have been proposed to address the problems of general (video) game description and the encoding of game mechanics for analysis. Our work has very similar goals to those listed in the Dagstuhl paper proposing VGDL (a video game description language) (Ebner et al. 2013), but Ceptre is not limited to describing games based on object collisions in 2D space. Gamelan (Osborn, Grow, and Mateas 2013), EGGG (Orwant 2000) and DisCo (Jarvinen et al. 1990) are proposals for domain-specific languages for describing game rules, but they do not provide a logic-based justification for the language design. Smith et al. have investigated computer-aided authoring of games through such means as constraint specification and procedural content generation (Smith, Nelson, and Mateas 2009; Smith 2012), which shares with our approach a use of logic programming as its foundation. In their setting, game rules are treated as *term-level objects* (as in, the indices of predicates) over which program rules are written, rather than directly encoded as rules themselves. This treatment separates the semantics of the logic program itself from the semantics of the game, meaning as a consequence that the object of formal analysis is distinct from the playable artifact (although they are connected semantically). We de-emphasize automatic generation via constraints in favor of stronger support for flexible hand-authoring of designs, and we prioritize in our programming language design a semantics that may be directly mapped onto games' operational meaning.

Finally, several game creation tools have gained popularity among hobby and novice game designers, making them attractive to our goals of striking a balance between expressive power and minimality of core concepts. These tools include Inform 7 (Nelson 2001), Kodu (MacLaurin 2011), and PuzzleScript (Lavelle 2013). We have carried out detailed studies of expressing idioms native to these tools in Ceptre and are hopeful that, despite the power sacrificed for generality, Ceptre can express the domain-specific idioms of these tools in a way that may lead to innovative recombination.

Conclusion

We have presented a linear logic-based framework for authoring and reasoning about generative game designs, and an implementation of these ideas in the Ceptre logic programming language, which models gameplay as proof search. We have demonstrated via two examples, a Shakespeare-inspired story world and a dungeon-crawler-inspired game, the use of this language as a generative system to which one may selectively add interactive components. We have carried out several additional case studies with this tool, including board game, garden and dialogue simulators, illustrating Ceptre's potential for inventing new operational logics. We are actively developing the language in the open and encourage contributions of examples from the game design community at large.

Going forward, we aim to expand the range of analytical tools for Ceptre specifications. We have in progress several candidate algorithms for checking programmer-specified game invariants, and tools for visualizing and manipulating causally-structured traces. These tools put to use in the context of the generative game examples well-described in Ceptre could lead to novel approaches to expressive range analysis (Smith and Whitehead 2010), for example.

Linear logic has been used independently to study numerous phenomena, such as memory management, concurrency, game theory, and quantum physics. Its continual rediscovery in these many domains leads us to believe it is one of the more "permanent ideas" in computer science, robust to advances the technology industry that might otherwise affect the way we formulate something as culturally and technologically contingent as video game design. By providing a logical underpinning to techniques used in planning-based and ad-hoc approaches to game description, we aim to provide a basis for extension and interoperability of game specifications at the language level.

Acknowledgements

This work was made possible by grant NPRP 09-667-1-100, from the Qatar National Research Fund (a member of the Qatar Foundation). The statements made herein are solely the responsibility of the authors.

References

Araújo, M., and Roque, L. 2009. Modeling games with Petri nets. *Breaking New Ground: Innovation in Games, Play, Practice and Theory. DIGRA2009. Londres, Royaume Uni.*

Bosser, A.-G.; Cavazza, M.; and Champagnat, R. 2010. Linear logic for non-linear storytelling. *ECAI Frontiers in Artificial Intelligence and Applications* 215.

Champagnat, R.; Prigent, A.; and Estraillier, P. 2005. Scenario building based on formal methods and adaptative execution. *ISAGA, Atlanta (USA)* 6.

Dang, K. D.; Hoffmann, S.; Champagnat, R.; and Spierling, U. 2011. How authors benefit from linear logic in the authoring process of interactive storyworlds. In *ICIDS*, 249–260.

Dormans, J. 2011. Simulating mechanics to study emergence in games. In *Artificial Intelligence in the Game Design Process*.

Ebner, M.; Levine, J.; Lucas, S. M.; Schaul, T.; Thompson, T.; and Togelius, J. 2013. Towards a video game description language. *Artificial and Computational Intelligence in Games* 6:85–100.

Girard, J.-Y. 1987. Linear logic. *Theoretical Computer Science* 50(1):1–102.

Harper, R.; Honsell, F.; and Plotkin, G. 1993. A framework for defining logics. *J. ACM* 40(1):143–184.

Hartsook, K.; Zook, A.; Das, S.; and Riedl, M. O. 2011. Toward supporting stories with procedurally generated game worlds. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, 297–304. IEEE.

Hodas, J. S., and Miller, D. 1994. Logic programming in a fragment of intuitionistic linear logic. *Information and computation* 110(2):327–365.

Jarvinen, H.; Kurki-Suonio, R.; Sakkinen, M.; and Systa, K. 1990. Object-oriented specification of reactive systems. In *Software Engineering, 1990. Proceedings., 12th International Conference on*, 63–71. IEEE.

Krizanc, J. 1997. Tamara: a play.

Lavelle, S. 2013. PuzzleScript game engine. `http://www.puzzlescript.net`. Accessed: 2015-05-31.

MacLaurin, M. B. 2011. The design of Kodu: a tiny visual programming language for children on the Xbox 360. *SIGPLAN Not.* 46(1):241–246.

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

Mateas, M., and Stern, A. 2003. Façade: An experiment in building a fully-realized interactive drama. In *Game Developers Conference*, 4–8.

Mateas, M., and Wardrip-Fruin, N. 2009. Defining operational logics. *Digital Games Research Association (DiGRA)*.

McCoy, J.; Treanor, M.; Samuel, B.; Tearse, B.; Mateas, M.; and Wardrip-Fruin, N. 2010. Comme il faut 2: A fully realized model for socially-oriented gameplay. In *Proceedings of the Intelligent Narrative Technologies III Workshop*, 10. ACM.

Medler, B., and Magerko, B. 2006. Scribe: a tool for authoring event driven interactive drama. In *Technologies for Interactive Digital Storytelling and Entertainment*. Springer. 139–150.

Nelson, G. 2001. *The Inform Designer's Manual*. Placet Solutions.

Orwant, J. 2000. Eggg: Automated programming for game generation. *IBM Systems Journal* 39(3.4):782–794.

Osborn, J. C.; Grow, A.; and Mateas, M. 2013. Modular computational critics for games. In *AIIDE*.

Porteous, J.; Cavazza, M.; and Charles, F. 2010. Applying planning to interactive storytelling: Narrative control using state constraints. *ACM Trans. Intell. Syst. Technol.* 1(2):10:1–10:21.

Schack-Nielsen, A., and Schürmann, C. 2008. Celf — a logical framework for deductive and concurrent systems (system description). In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'08)*, 320–326. Springer LNCS 5195.

Smith, G., and Whitehead, J. 2010. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 4. ACM.

Smith, A. M.; Nelson, M. J.; and Mateas, M. 2009. Computational support for play testing game sketches. In *AIIDE*.

Smith, A. M. 2012. *Mechanizing Exploratory Game Design*. Ph.D. Dissertation, University of California, Santa Cruz.

Zook, A., and Riedl, M. O. 2014. Automatic game design via mechanic generation. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*.