Translating Between Game Generators with Asterism and Ceptre

Cynthia Li
Independent researcher
San Jose, CA, USA
licynthia.x@gmail.com

Joseph C. Osborn
Pomona College
Claremont, CA, USA
joseph.osborn@pomona.edu

ABSTRACT

In this paper, we present in-progress work that converts games made with Ceptre, a genre-agnostic game description language, into graphical games using the framework of operational logics. Our preliminary code targets the translation of tilemap-based dungeon crawlers, but we present strategies for generalizing this process to other Ceptre games and Asterism engines. We gesture at the potential of operational logics and Asterism as a tool to communicate across the many frameworks surrounding game development and playing.

CCS CONCEPTS

• **Software and its engineering** → **Domain specific languages**; **Source code generation**; • **Applied computing** → **Computer games**.

KEYWORDS

game generators, operational logics, formal models

ACM Reference Format:
Cynthia Li and Joseph C. Osborn. 2024. Translating Between Game Generators with Asterism and Ceptre. In *Proceedings of the 19th International Conference on the Foundations of Digital Games (FDG 2024), May 21–24, 2024, Worcester, MA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3649921.3659847

1 INTRODUCTION

There exist a wide range of game description languages and game-making tools, all with distinct priorities and goals. Here, we focus on two of them:

- Asterism, a tool for making game engines using the framework of operational logics [8], and
- Ceptre, a genre-agnostic programming language designed by Chris Martens that manipulates its state through rewrite rules [6].

Ceptre allows for high-level descriptions of game rules using multiset rewriting, and a user can define their own types rather than being beholden to a pre-existing schema of supported game mechanics. It's meant to be simple to learn and use, with high flexibility when it comes to *what* is being modelled. However, that flexibility means it's difficult to commit to *one* graphical representation in particular—so Ceptre doesn't at all, preferring to focus on its formal representation. This preference for the command line means that Ceptre doesn't easily allow programmers to create *graphical* games. Moreover, the process of translating between Ceptre and Asterism's distinct languages for game description shows the potential for theorizing a more general approach to communication between game generation tools.

Our ongoing research seeks to convert Ceptre games to Asterism ones as a preliminary case study; this work is still in progress and much of it is currently speculative. This paper theorizes a process for how this translation might take place, suggesting structural similarities between the two tools, and bridging the abstraction gap between Ceptre and Asterism through annotated extra information. This process would allow for the generation of graphical games from high-level, abstract rewrite rules.

2 BACKGROUND

2.1 Related Work

There are many formal game description languages and game generation tools: Game Description Language is a high-level, abstract language oriented toward planning [13]; VideoGame Description Language provides a closed but detailed language for creating tilemap-based 2D games [11]; Gemini generates games that reason about the meaning of the objects within it using the game description language Cygnus [12]; ANGELINA primarily focuses on story-based arcade games, metroidvanias, and small 3D games with coherent themes [2, 3].

This litany of tools shows us the breadth of game generation research. However, they all have distinct purposes and models of game-ness, and many of them are targeted for particular mainstream genres of games—metroidvanias, platformers, dungeon crawlers, etc. Their one-off nature means that we are less able to discuss how they are related, esepcially when it comes to modeling and verification across different frameworks. Work has been done on mapping meaning across different games: Mirgati and Guzdial [7] manually match tiles in *Kid Icarus* and *Super Mario Bros.* together by function, then generate games from the shared, unified tileset. Bentley and Osborn [1] describe a set of affordances for *The Legend of Zelda*, and Sarkar et al. [10] expand this work to describe a wider set of NES side-scrolling platformers. However, the game genres here are fairly restricted.

2.2 Ceptre and Asterism

We choose Ceptre and Asterism specifically because they gesture at a potential to encompass games that look very different from the ones used above, supported by their underlying frameworks and formal properties. Ceptre was written in reaction to game

FDG 2024, May 21–24, 2024, Worcester, MA, USA

ACM ISBN 979-8-4007-0955-5/24/05
https://doi.org/10.1145/3649921.3659847

description languages that only targeted a slim range of games and game mechanics. Its high-level descriptions mean it can describe a wide variety of games with less effort on the writer's part, since a user can define their own types rather than being beholden to a pre-existing schema of supported game mechanics. This allows us to quickly write high-level descriptions of game rules using the formalism of multiset rewriting.

Asterism, on the other hand, is a library that operationalizes operational logics (OLs) in Rust, providing a computational definition of individual OLs [8]. OLs are a framework from games studies developed by Noah Warduip-Fruin and Michael Mateas, and further expanded on by Joseph Osborn. Like Ceptre, OLs seek to be widely applicable to games, but they focus on fine-grained implementation details of meaning-making through games.

OLs aren't logics in the mathematical sense—instead, they model "mechanics" as a meshing together of related abstract processes. OLs identify processes (individually referred to as *logics*) that are repeated across many entities throughout a game, and describe how those processes interact with each other [9]. For example, the event of a character jumping in a 2D sidescrolling platformer invokes both a control logic (a player pressing the button assigned to the "jump" action) and a physics logic (applying vertical acceleration to the character).

Asterism allows a user to compose OLs as described above to create game engines, which act as rule schemas for concrete game worlds. When data is given to a ruleset, those engines produce games. Osborn et al. [9] name the ways logics are woven together, describing communication channels, operational integrations, and structural syntheses:

- *Structural syntheses* ensure that game state is consistent across logics, creating a cohesive game world;
- *operational integrations* mediate the way logics react to events triggered by other logics; and
- *communication channels* express the data that logics provide to the player.

The engines generated by Asterism can then generate games. This approach allows a greater number of engines to be built from the same base parts. OLs can also describe a much wider variety of games than general game description languages because logics slice across game "mechanics," separating them into systems that can instead be relocated into different kinds of games. This genre-agnosticism makes it an attractive match for Ceptre, although the limited current work on Asterism means it isn't quite as flexible.

3 COMBINING FRAMEWORKS

We choose Ceptre and Asterism because they both seek to be relatively agnostic to operational logics—or at least, support a wide range of them, beyond what description languages like VDGL prioritize. As opposed to GDL and its focus on planning and goals, they're both more interested in game rules and state changes in the core game loop. These similarities drive our inquiry here.

Ceptre exists at a higher level of abstraction than Asterism does, so its game descriptions require less information. Our task is to annotate and map what does match across game schemas, then fill in any other data that might be missing to replicate a game in Asterism. However, this work of translating *all* Ceptre games to *any* Asterism engine is an incredibly complex task. The exact scope of this question is uncertain, and defining it would require more exploration.

For this paper, we focus on translating annotated Ceptre games to one specific Asterism engine, `boxsy` (see Figure 1). `boxsy` is a minimal game engine based on Adam LeDoux's Bitsy [5] for simple tile-based dungeon crawlers. It provides a player, characters, tiles, resources, and rooms. Players and characters can have resources, and both tiles and characters can link to other rooms. The player can collect resources and traverse links on collision with the characters or tiles they belong to.

To generate a `boxsy` game, we first write an annotated Ceptre schema, which can be played using the software; a playtrace is shown in Figure 2. We seek to reverse-engineer a `boxsy` game out of this Ceptre schema. The Ceptre code and mock-up code for a generated `boxsy` game can be found in the supplemental materials for this paper.

We choose this engine because it highlights linking logics and resource logics, which Ceptre handles more easily than more fine-grained OLs like physics and collision. Ceptre *already* generates a playtrace as a transition system, and its rewrite rules support

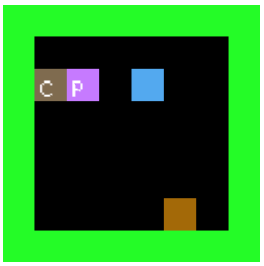

Figure 1
A simple dungeon crawler game made with `boxsy`.

```
0: (quiesce)
1: (take_tile_exit p1 r1 r2)
?- 1

0: (quiesce)
1: (take_rsrc p1 r2 c1 rocks 1 0)
2: (take_char_exit p1 r2 c1 r3)
3: (take_tile_exit p1 r2 r1)
?- 1

0: (quiesce)
1: (take_rsrc p1 r2 c1 rocks 0 1)
2: (take_char_exit p1 r2 c1 r3)
3: (take_tile_exit p1 r2 r1)
?- 1

0: (quiesce)
1: (take_char_exit p1 r2 c1 r3)
2: (take_tile_exit p1 r2 r1)
?- 1

0: (quiesce)
1: (take_rsrc p1 r3 c2 stones 1 0)
2: (take_tile_exit p1 r3 r2)
?- 0
```

Figure 2
The Ceptre version of this game played to quiescence.

pattern-matching to support resource management. While the language *can* be used to express the events generated by collision logics, graph-shaped logics are more directly structurally supported by the tool.

3.1 Generating Games

Before translating between these two tools, we must first generate an Asterism engine. If a game genre can be defined as "a *conventional configuration of operational logics* in relation to one another" [4], then we start by choosing the OLs involved in the genre for our game. We define structural syntheses across those OLs, producing types that show which concepts belong to the same game object across logics—for example, a collision body, a set of controls, and a resource pool of health would create a player character. We describe how those types *react* to one another with operational integrations. While structural syntheses are passive and ontological, operational integrations can operate like event-reaction pairs. We might declare that when the arrow keys are pressed, a character moves in response. This level of composition, where structural syntheses are combined with operational integrations, create game *schema*, rulesets for games whose actual content can differ. Providing data for the schema above would then generate a playable game. `boxsy` uses a closed set of operational integrations, and only requires the user to plug in its data; however, not all Asterism engines must be constructed this way.

In contrast, Ceptre is used specifically for describing game schema, then defining the data involved in an initial state, creating a *single* game that can be played, although that initial state can be easily swapped out for another one. A Ceptre schema contains the types and rules of Asterism engines sans the explicit connection to OLs. We write a Ceptre program by defining data types and predicates, then combining those predicates in rewrite rules, which change pieces of the program's state by identifying the patterns on the left hand side and replacing them with the right hand side. We define an initial state with a set of starting predicates, then can run the program, handling non-determinism of what rules can be applied with either user input or randomness. Using the `boxsy` engine gives us another advantage in that it already structurally resembles this process, providing us with a heavily restricted set of operational integrations. It's difficult for a user of the game engine to make a game with a behavior that the engine hasn't defined already, similarly to Ceptre's separation of rule definitions from initial state.

3.2 Translation

We define the following four types of annotations.

`synthesis` identifies types which are things in the game, and what logics are attached to them:

```
%*** synthesis { control, collision, resource }
player_id: type.
p1: player_id.
```

We map the logics in the annotations to the mappings produced by the `boxsy` engine. Since `boxsy` is quite small, this mapping is one-to-one and works without any more qualifiers. A more complex engine might prefer annotating Ceptre types directly with the `boxsy` type in addition to these logics.

Additionally, the `data` keyword marks types that further describe the syntheses above:

```
%*** data { resource }
rsrc_id: type.
rocks: rsrc_id.
stones: rsrc_id.
```

The `query` keyword identifies *predicates* that reinforce the structural syntheses of the involved types. They link the types with the data that's associated with them:

```
%*** query { linking }
player_in_room player_id room_id: pred.
```

`integration` annotations mark rewrite rules, which in this case serve as operational integrations, which in `boxsy` are simply called events. They comprise of two parts—a precondition which allows for the event to take place (the left hand side of the rule), and the actual result of the event on the other side:

```
stage play {
  %*** integration { collision, linking, control }
  take_tile_exit:
    player_in_room P R * $tile_link R (exit R')
      -o player_in_room P R'.
  % ...
}
```

Here, we identify integrations by looking at the mappings of rewrite rules and match them directly with events in `boxsy`, and assume compatibility because there aren't that many possible events to match on. If events could be applicable to multiple entities with differing behavior, we look at the types and predicates involved to clarify. For example, a rewrite rule named `take_tile_exit` applies specifically when a player touches a tile, but has the same "logic signature" as `take_char_exit`. However, since the former involves the `tile_id` type, we can check the rewrite rule to see what syntheses are involved. For more complex engines, if there's overlap between signatures that could map to different *kinds* of events offered by the same logic, we can do more precise matching using annotations of logics' *provided concepts* when specifics are needed. Again, however, `boxsy` is simple enough that this is not necessary.

Everything else—communication channels, level generation, etc—are beyond the scope of Ceptre entirely and must be decided by `boxsy`. The leap in abstraction between Ceptre's text interface and `boxsy`'s (very simple) graphical one means there's some amount of level and asset generation that has to happen between them. Those processes must respect the bounds that Ceptre sets (i.e. two characters, three rooms, a link between room 1 and 2 activated by a particular tile, etc), but concrete data like map layout and character positions can be fairly arbitrary. More sophisticated techniques for level and art asset generation is out of scope here, and has been discussed in depth elsewhere.

3.3 Roadblocks

Structurally, Ceptre and Asterism are quite similar, breaking up into roughly the same levels of types-schema-game data. The main difficulty of translating between them is the difference between their fine-grained-ness and abstraction levels. For one, the game

update loop of many Asterism engines revolve around a process of branching out to a specific logic to do work, then returning to the main loop, separating work out to individual logics instead of as a cohesive whole. Ceptre makes no such distinction; early attempts at writing Ceptre games in an "Asterism-y way" were clumsy and convoluted. Ceptre prefers a more concise approach, but its rigid formalism and structure allows us to *infer* OL-style relations so long as we know what logics its datatypes are associated with. This means the conversion process requires a great deal more information when moving from Ceptre's rewrite rules to Asterism's more complicated game loop.

We also sometimes need to *fabricate* information when reaching levels of detail that Ceptre doesn't care about. Ceptre *can* be used to model a collision system, but doing so in full would be clumsy since Ceptre lacks support for arithmetic or the procedural programming style necessary for those calculations, *especially* when they're not even useful to focus on for the game system at hand. Instead, when talking about collision in `boxsy`, we focus on the actual collisions/contacts and restitutions that occur instead of the actual math of the collision update loop. The control logic is also often simply implied by *Ceptre's actual interface*, rather than keeping track of literal player inputs as we would in `boxsy`. The Ceptre version of the game has no understanding of an `ActionID` as `boxsy` defines it, and will not care about individual key presses. This must be filled in by the translation process, whether through procedural generation (in the case of level maps, entity placement, and geometry), manually generating this content (leaving a user to fill in missing data themselves), or through pre-set default values (e.g. for input schemes).

4 CONCLUSION

Our research is preliminary, gesturing at what precisely the relationship between Ceptre and Asterism might be. This paper discusses a potential strategy for translating between these two tools, and gestures at its formal implications.

Other work in this vein might use a different configuration of operational logics or different Asterism engines, generalizing further to encompass a wider variety of Ceptre games. `boxsy` was selected specifically for its pre-existing similarity to Ceptre; how might other engines translate? Additionally, since our mapping is *structural* rather than meaning-based, we consider what might be revealed when converting an entirely different Ceptre schema to the same Asterism engine, or a Ceptre schema to a different Asterism engine. These alternate Ceptre games or Asterism engines might not even require the concepts of characters or tiles or players altogether.

The *reverse* process may also be valuable, cutting away information from Asterism games to form Ceptre schema. Since formal work has already been done with Ceptre [6], more work on specifying Asterism's formal aspects—e.g. formally describing operational logics—would be useful, with the goal of interoperability with Ceptre's formalism in mind.

ACKNOWLEDGMENTS

We would like to thank Chris Martens for their work on Ceptre, and for helping us with the software.

REFERENCES

[1] Gerard R. Bentley and Joseph C. Osborn. 2019. The Videogame Affordances Corpus. In *Proceedings of the AIIDE Workshop on Experimental AI in Games.*
[2] Michael Cook, Simon Colton, and Jeremy Gow. 2017. The ANGELINA Videogame Design System—Part I. *IEEE Transactions on Computational Intelligence and AI in Games* 9, 2 (2017), 192–203. https://doi.org/10.1109/TCIAIG.2016.2520256
[3] Michael Cook, Simon Colton, and Jeremy Gow. 2017. The ANGELINA Videogame Design System—Part II. *IEEE Transactions on Computational Intelligence and AI in Games* 9, 3 (2017), 254–266. https://doi.org/10.1109/TCIAIG.2016.2520305
[4] Tamara Duplantis, Isaac Karth, Max Kreminski, Adam M Smith, and Michael Mateas. 2021. A genre-specific game description language for game boy rpgs. In *2021 IEEE Conference on Games (CoG).* IEEE, 1–8.
[5] Adam LeDoux. 2024. bitsy. http://bitsy.org/. Accessed 11 January 2024.
[6] Chris Martens, Alexander Card, Henry Crain, and Asha Khatri. 2023. Modeling Game Mechanics with Ceptre. *IEEE Transactions on Games* (2023), 1–14. https://doi.org/10.1109/TG.2023.3292982
[7] Negar Mirgati and Matthew Guzdial. 2023. Joint Level Generation and Translation Using Gameplay Videos. arXiv:2306.16662 [cs.CV]
[8] Joseph C. Osborn, Cynthia Li, and Katiana Wieser. 2021. Asterism: Operational logics as a game engine engine. In *Workshop on Programming Languages in Interactive Entertainment.*
[9] Joseph C. Osborn, Noah Wardrip-Fruin, and Michael Mateas. 2017. Refining Operational Logics. In *Proceedings of the 12th International Conference on the Foundations of Digital Games* (Hyannis, Massachusetts) *(FDG '17).* Association for Computing Machinery, New York, NY, USA, Article 27, 10 pages. https://doi.org/10.1145/3102071.3102107
[10] Anurag Sarkar, Adam Summerville, Sam Snodgrass, Gerard Bentley, and Joseph Osborn. 2020. Exploring Level Blending across Platformers via Paths and Affordances. arXiv:2009.06356 [cs.LG]
[11] Tom Schaul. 2013. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Inteligence in Games (CIG).* 1–8. https://doi.org/10.1109/CIG.2013.6633610
[12] Adam Summerville, Chris Martens, Ben Samuel, Joseph Osborn, Noah Wardrip-Fruin, and Michael Mateas. 2018. Gemini: Bidirectional Generation and Analysis of Games via ASP. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 14, 1 (Sep. 2018), 123–129. https://doi.org/10.1609/aiide.v14i1.13013
[13] Michael Thielscher. 2011. The General Game Playing Description Language is Universal. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two* (Barcelona, Catalonia, Spain) *(IJCAI'11).* AAAI Press, 1107–1112.