Modeling Game Mechanics with Ceptre

Chris Martens, Alexander Card, Henry Crain, and Asha Khatri

***Abstract*—Game description languages have a variety of uses, including formal reasoning about the emergent consequences of a game's mechanics, implementation of artificial intelligence decision-making where the game's rules make up the space of possible actions, automated game and level generation, and game prototyping for the sake of low-time-investment design and tinkering. However, in practice, a new game description language has been invented for almost every new use case, without providing formal underpinnings that follow generalizable principles and can be reasoned about separately from the specific software implementation of the language.**

Ceptre is a language that attempts to break this pattern, based on an old idea known as multiset rewriting. This paper describes the language formally, through example, and in a tutorial style, then demonstrates its use for writing formal specifications of game mechanics so that they may be interactively explored, queried, and analyzed in a computational framework. Ceptre allows designers to step through executions, interact with the mechanics from the standpoint of a player, run random simulated playthroughs, collect and analyze data from said playthroughs, and formally verify mathematical properties of the mechanics, and it has been used in a number of research projects since its inception, for applications such as procedural narrative generation, formal game modeling, and game AI.

***Index Terms*—prototyping tools, game modeling, game description languages, multiset rewriting**

I. Introduction

Game designers and developers think and communicate using a number of well-established formal abstractions, such as finite state machines, behavior trees, cellular automata, context-free grammars, game trees, and state-space graphs. These abstractions exist on the whiteboard as design tools as well as in codebases as implemented programs in, essentially, domain specific languages for authoring specifications within the terminology of those abstractions (e.g. alphabets and production rules for context-free grammars). The existence of these formalisms as *implementation independent languages* helps us document and transmit knowledge about any number of phenomena of use to the creative practice of digital games, including automated non-player character behavior, automated opponent strategy, player modeling, procredural content generation, and other topics generally studied under the heading of "Game AI."

This paper argues for adding another formalism to the Game AI formalism cannon, *multiset rewriting*, that happens to correspond nicely to some common design patterns in game AI and game design research. We describe Ceptre, a programming language originally introduced in 2015 [38], which implements multiset rewriting over user-defined, typed domains. Ceptre was designed in reaction to a lack of tools for prototyping game mechanics across a flexible range of operational logics [69] without commitment to a specific mode of rendering game state and player interaction. The implementation includes a novel language feature called *stages*, as well as mechanisms for authoring typed domains, interacting with simulations, and inspecting causal traces, all of which were developed to support use cases discovered by the author during development.

Ceptre can be thought of as consisting of two parts: (1) a computational language; that is, a formally-describable set of language constructs and definitions for their meaning; and (2) an implementation of this language as a set of software tools, comprising an authoring system intended for use in specific contexts (e.g. game mechanic design, narrative modeling, and procedural generation). This two-part conceptualization enables us to define the semantics of the language in an implementation-independent way so that it is replicable by others, and so that additional use contexts (and corresponding features and tools) may be built atop it.

Ceptre the *language* is fundamentally a variant of multiset rewriting [29], which can be stated succinctly as a formal system and independent from its implementation. Ceptre implements a flexible, domain-independent version of multiset rewriting with user-defined types and relations. It has in common with many other description languages the feature of describing state transitions in terms of *preconditions* and *effects*, as seen in languages such as PDDL [25], Ludi [7], Cygnus [65], and in game creation tools such as Kodu [43], GameMaker [73], and Stencyl [36]. We briefly cover its relationship to these systems in Section II. Ceptre the *toolset* currently consists of a parser, interpreter, and command-like based interactive execution engine, as well as a web-based structure editor for a simplified subset of the language.

In this paper, we describe both the language and toolset in detail, first through example and then through formal definition. We review applications of the language to a number of domains, such as narrative generation [40], AI for interactive fiction [13], quest generation [2], and formal reasoning about game mechanics [39]. Thus, our key contributions in this paper are:

- A tutorial explanation of multiset rewriting as applied to game modeling, and a process for mapping game modeling problems to the affordances of multiset rewriting;
- The technical contribution of describing Ceptre's semantics implementation-independent, reproducible way, such that other programmers and researchers may reimplement the underlying ideas in their languages and development contexts of choice;
- A summary of empirical and mathematical evidence of Ceptre's important properties, including usability and compositionality;
- A discussion of Ceptre's formal properties, and the strengths and limitations of those properties;

- A discussion of how Ceptre has been used since its inception, and guidance for future use in the computational study of digital and analog games.

II. Related Work

Because Ceptre is based on a general-purpose, domain-independent formalism (much like state machines), but its implementation and tool support has domain-specific purposes, it can be difficult to narrow down the set of existing formalisms and tools to compare to. Here, we identify two main overlapping categories of related work: **game description languages**, which span more than just the multiset rewriting paradigm; and **multiset rewriting languages** for specification and modeling, which span more than just games as a domain.

A. Game Description Languages

A number of efforts have been made to define specification languages specific to game rules under the banner of "game description languages" (GDLs), beginning with the Stanford GDL, which was used to specify and reason about the rules of board games for the sake of "general game-playing" as an AI problem [20]. Game description languages have been developed for a number of reasons: to specify formally the design space of mechanics for games for the purpose of automated generation of rules and mechanics [6], [65], levels [28], and tutorials [22]; to reason formally about the emergent consequences of games [6], [66]; to prototype and compare game ruleset iterations [62], [63]; to model an AI agent's internal understanding of the game world for the sake of robust decision-making behavior [20]; and more. In 2013, a Dagstuhl meeting of digital games researchers identified a need for GDLs specific to video games and outlined a proposal called VGDL [17], which was later implemented in Python as PyVGDL [61] and now forms the basis for the IEEE GVGAI (general video game AI) series of competitions [34], [54]. Other GDLs that address video game design specifically include EGGG [51], Gamelan [53], Machinations [16] (and MicroMachinations [30]), and Cygnus, the underlying DSL of the Gemini game generation system [64], [65].

The GDL approach entails creating affordances for specification that are unique to games: built-in constructs may include player control, entity movement and collision, or piece movement on a grid (in the case of board games). In Mateas and Wardrip-Fruin's parlance, each GDL embeds a fixed set of *operational logics* [69], or underlying code abstractions for common sets of game mechanics, that they support. This approaches trades generality for a more rigid definition of what constitutes a game in their specific applications.

By contrast, formalisms such as state machines, regular languages, and context-free grammars have provided long-running foundations for much of computer science, including game AI. We argue that game AI would benefit tremendously from formal languages defined with the same level of formal rigor as these other foundational tools if we want them to outlast the specific research projects in which they have been applied to date. Accordingly, Ceptre is based on *multiset rewriting*, a general mathematical toolset that can be succinctly defined in a reproducible way. Multiset rewriting nicely coincides with a variety of common abstractions found in games, but nothing is built into it specific to a particular notion of what a game is made out of, similar to other formalisms such as context-free grammars and state machines. Support for programmer-defined types and constants ensures domain flexibility. At the same time, multiset rewriting is *limited* enough (in contrast to full-featured, general-purpose programming languages) that we can still formally analyze and reason about games specified in it.

TABLE I
Mapping from game design concepts to formal constructs.

Game design concept	Mathematical construct
Game states	Multisets of predicates
Game rules	Multiset rewriting rules
Game state transition	Rule application
Playtrace	Sequence of rule applications

B. Multiset Rewriting- and Linear Logic-Based Modeling

It follows from the general applicability of multiset rewriting that it has been used in a number of other domains outside of games. Two languages in particular, K [58] and Maude [12], have been used extensively for modeling and reasoning about the semantics of programming languages. In the programming languages community, encoding and formally proving properties of a language is a common research activity known as *mechanizing metatheory* [24], and there are many tools that support this activity, including interactive proof assistants and logical frameworks [57]. Ceptre's origins lie within this tradition: its direct predecessor is Celf [60], based on CLF, the Concurrent Logical Framework [70], [10]. Celf builds on a tradition of linear logic programming languages, including the Linear Logical Framework (LLF) [9], Lygon [72], and Forum [44], each of which implements an overlapping set of features to model and reason about domains as diverse as molecular biology [11], voting protocols [15], and cryptographic security protocols [5].

Other rule-based systems bear a strong resemblance to MSRW, but fall outside the formal definition. These include the NetLogo [67] and AgentSheets [56] tools, also touted for cross-domain complex system modeling, and Kappa [4], a rule-based system designed for molecular systems biology modeling, as well as PuzzleScript [32], whose pattern-matching syntax was indirectly influenced by linear logic (citing personal communication). One could argue that the Planning Domain Definition Language (PDDL) used for specifying problems for AI planners [21] is also similar, given that both share a syntax of specifying preconditions and effects of actions in terms of logical predicates, but there are subtle formal distinctions that distinguish their applicability in different use cases—see Section V-A for discussion.

III. Modeling Game Mechanics with Multiset Rewrite Rules

In Table I, we outline our representational approach, mapping concepts in game design to the formal structures of

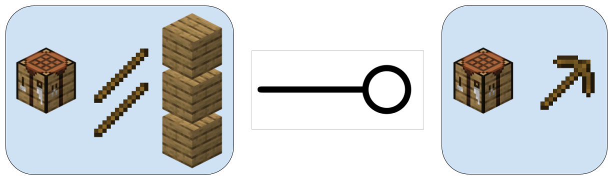


Fig. 1. A multiset rewriting rule describing the Minecraft recipe for a wooden pickaxe: a crafting table, two sticks, and three planks can be rewritten into a crafting table and a wooden pickaxe.

multiset rewriting [29] (MSRW). *Multiset rewriting* is a mathematical formalism in which systems can be specified as sets of rules that rewrite states, where states are represented as multisets (i.e. sets whose elements can occur more than once). The general process for modeling a game in MSRW is to design a set of predicates that constitute game states, describe game states as *multisets* of those predicates, and describe game rules (mechanics) as multiset rewriting rules. *Rewriting* a state means pattern matching on it and replacing the matched subset with a new set. At its core, a multiset rewriting-based model is composed of these rules of the form $A \multimap B$, where A and B are multisets. This technique turns out to be a powerful representation scheme for games, as we will show.

To explain MSRW formally, we will develop a running example based on *Minecraft* [47], a video game in which resources found in the game environment are collected and used in “crafting recipes” to create new items. As a crafting game, Minecraft gives us a reason to talk specifically about multiplicity of set elements, which we will use to directly represent the quantity of a resource. See Figure 1 for an example rewrite rule based on the Minecraft recipe for crafting a wooden pickaxe from two sticks and three planks.[1] A complete implementation of this example in Ceptre is available on GitHub.

To start with the bare minimum amount of formalism, we will use uninterpreted symbols (identifiers written in mathsf format) as the elements of multisets, which we will refer to as *propositional atoms* or *atomic propositions* (atomic because they cannot be broken into constituent pieces). These symbols also nicely correspond to *resources* in the Minecraft example. However, we will eventually expand multiset elements to include predicates in first-order logic that accept arguments, which enables us to represent much more than resource-exchange mechanics.

A. Game States as Multisets

A *game state* keeps track of all of the variables in the game world and their current value. In a relational model, the association between a variable and its value is usually represented as a logical predicate $p(var, val)$. However, a “nullary” predicate p (one that has no arguments) can also represent a part of a game state. In the multiset rewriting model, it can be helpful to think of such a predicate p as a *resource*. Its absence represents a lack of that resource in the current state; its presence—and quantity!—represents that resource’s availability.

[1]This example model depicts a simplification of Minecraft recipes in which the 2D grid layout of ingredients is discounted.

Rule name	Rule
chop_tree	$\mathsf{tree} \multimap \mathsf{wood}$
chop_wood	$\mathsf{wood} \multimap \mathsf{plank}^4$
chop_plank	$\mathsf{plank} \multimap \mathsf{stick}^4$
craft_table	$\mathsf{plank}^4 \multimap \mathsf{table}$
craft_wood_pickaxe	$\mathsf{table}, \mathsf{plank}^3, \mathsf{stick}^2 \multimap \mathsf{table}, \mathsf{wood\_pickaxe}$
craft_stone_pickaxe	$\mathsf{table}, \mathsf{cobble}^3, \mathsf{stick}^2 \multimap \mathsf{table}, \mathsf{stone\_pickaxe}$
mine_stone	$\mathsf{wood\_pickaxe}, \mathsf{stone} \multimap \mathsf{wood\_pickaxe}, \mathsf{cobble}$

Fig. 2. A selection of (simplified) Minecraft crafting recipes in multiset rewriting notation. Each tree can be chopped to produce wood; each unit of wood can be divided into four planks; each plank can be further divided into four sticks. Four planks can be used to create a crafting table. With the crafting table, two stick, and three planks, we can create a wooden pickaxe (and we retain the crafting table). The wooden pickaxe can then be used to mine stone (converting it to cobble). Cobble and sticks can be used (if we have the crafting table) to craft a stone pickaxe.

For example, a Minecraft environment containing two trees and three stones could be represented by the following multiset:

$$\{\mathsf{tree}, \mathsf{tree}, \mathsf{stone}, \mathsf{stone}, \mathsf{stone}\}$$

Sometimes multisets are written more concisely by annotating each predicate with its multiplicity as an exponent:

$$\{\mathsf{tree}^2, \mathsf{stone}^3\}$$

This notation closely mirrors the depiction of a player’s inventory in Minecraft.

Syntactically, we define multisets Δ as comma-separated lists of propositions p. We then stipulate that these lists form a monoid under the comma operator (,), with the empty multiset $\emptyset$ as a unit element. That is, they obey associativity, commutativity, and unit laws as follows:

$$\begin{array}{rcl} \Delta_1, (\Delta_2, \Delta_3) & = & (\Delta_1, \Delta_2), \Delta_3 \\ \Delta_1, \Delta_2 & = & \Delta_2, \Delta_1 \\ \Delta, \emptyset = & \Delta & = \emptyset, \Delta \end{array}$$

This algebraic formulation allows us to write our rewrite rules without concern for the order in which elements appear in the state.

B. Game Mechanics as Multiset Rewrite Rules

In the Minecraft setting, modeled after early human tool construction and use, certain resources are found in the environment around the player, such as trees and stones. Players can then *craft* combinations of these materials into tools, such as a crafting table (needed for all subsequent crafting tasks), pickaxes, and forges, which can then be used for further mining and crafting.

{tree, tree, stone, stone, stone}

$\to_{\mathsf{chop\_tree}}$

{wood, tree, stone, stone, stone}

$\to_{\mathsf{chop\_tree}}$

{wood, wood, stone, stone, stone}

$\to_{\mathsf{chop\_wood}}$

{plank, plank, plank, plank, wood, stone,
stone, stone}

$\to_{\mathsf{craft\_table}}$

{table, wood, stone, stone, stone}

$\to_{\mathsf{chop\_wood}}$

{table, plank, plank, plank, plank, stone,
stone, stone}

$\to_{\mathsf{chop\_plank}}$

{table, stick, stick, stick, stick, plank,
plank, plank, stone, stone, stone}

$\to_{\mathsf{craft\_wood\_pickaxe}}$

{table, wood_pickaxe, stick, stick,
stone, stone, stone}

$\to^{3}_{\mathsf{mine\_stone}}$

{table, wood_pickaxe, stick, stick, cobble,
cobble, cobble}

$\to_{\mathsf{craft\_stone\_pickaxe}}$

{table, wood_pickaxe, stone_pickaxe}

Fig. 3. A sequence of transitions that shows how to craft the stone pickaxe from two trees and three stones. Resources that are matched by the succeeding transition are highlighted, and new resources produced as a result of applying the transition are in green. A superscript on a rule r^n indicates that the rule r is applied n times in sequence.

The transformation of resources is a perfect fit for multiset rewriting, in which rules are given that *replace* some set of resources with another. A rule $A \multimap B$, where A and B are both multisets, indicates that we can *replace* the resources represent A with with those represented by B. Figure 2 gives some of Minecraft's recipes in this representation scheme.

We will refer to the A part of a rule $A \multimap B$ as its *antecedent*, and the B part as its *consequent*. It will also be helpful to introduce a name for each rule. We will refer the preceding rules as `chop_tree`, `chop_wood`, `chop_plank`, `craft_table`, `craft_wood_pickaxe`, `craft_stone_pickaxe`, and `mine_cobble` respectively. For each rule name r corresponding to a rewrite rule $A \multimap B$, we say that r has the *signature* $A \multimap B$.

C. Game Actions as Transitions

A rule, or game mechanic, represents a general schema for how a game action may take place. In our example, all of the rules have been used to represent player actions, although rules can also represent engine-controlled actions as well (such as the laws of physics).

Next we draw a distinction between such a rule schema and its *instantiation* as an action on a particular game state, which we refer to as a *transition*. This distinction will be particularly useful when we introduce rules that range over parameters.

For now, we can think of a transition as a matching up between resources in the game state Δ and a given rule's antecedent. For example, if our Minecraft inventory contains a crafting table, two sticks, and three planks, then we can craft a wooden pickaxe using the `craft_wood_pickaxe` rule. If we have *more* than enough of each of these ingredients, then we need to specify which ones we are going to use. One way to write this concept formally is to say that our original game state Δ can be split (and rearranged) into an equivalent state Δ_A, Δ', where Δ_A matches the antecedent A of the rule we want to apply, and Δ' is whatever remains. We say a transition is *enabled* when its antecedent—the A part of the rule $A \multimap B$—is matched by some subset of the current state.

Definition 1 (Transition)**.** *If $\Delta = A, \Delta'$, and there exists a rule $r : A \multimap B$, then $A, \Delta' \to_r$ is an enabled transition.*

We can then define the *application* of a transition $A, \Delta_1 \to_{r:A\multimap B} \Delta_2$, representing how a transition transforms one state into another:

Definition 2 (Transition Application)**.** *If $\Delta = A, \Delta'$, and there exists a rule $r : A \multimap B$, then*

$$\Delta \to_{r:A\multimap B} B, \Delta'$$

For example,

$$\{\mathsf{plank}, \mathsf{tree}^2, \mathsf{stone}\} \to_{\mathsf{chop\_tree}}$$
$$\{\mathsf{wood}, \mathsf{plank}, \mathsf{tree}, \mathsf{stone}\}$$

Transition systems like this are common in formal language definitions, including more widely-used formalisms such as finite-state automata. However, multiset rewriting has the distinct advantage over this approach that states are not considered *monolithic*: we can look inside them and refer to changing only those components that matter for the rule, leaving the rest alone. In other words, multiset rewriting rules allow us to represent how, for any given game mechanic, only a subset of the state actually changes. This property, formally defined by transition application above, is sometimes known as a *frame property* an is an important aspect of the *compositionality* afforded by such models, as we will revisit in Section V-A.

If we iterate our transition relation by repeatedly applying rules, we arrive at a notion of *transitive closure* $\Delta \to^* \Delta'$, which says that Δ can be transformed into Δ' through repeated application of rules. See Figure 3 for an example.

After applying this sequence of transitions to reach that last state, no more rules apply. We refer to this condition as *quiescence*.

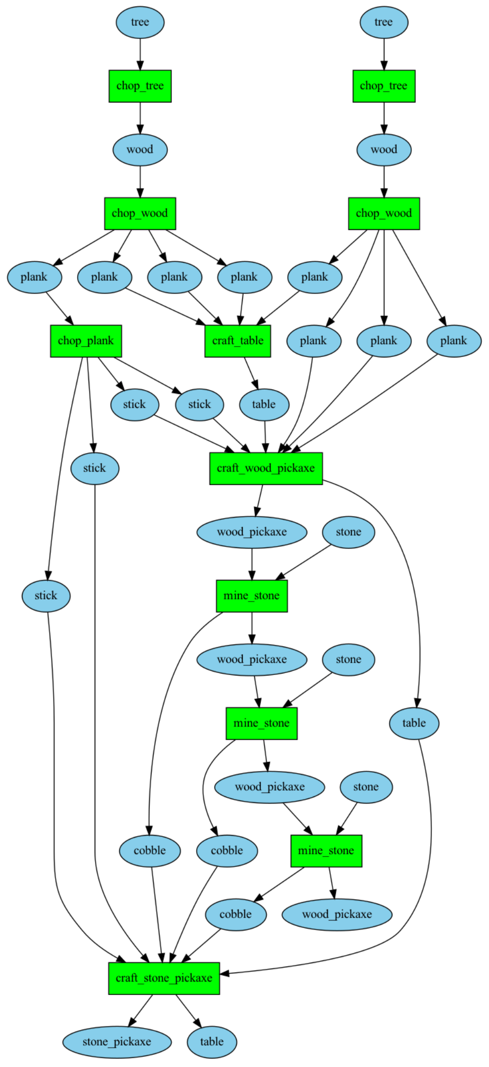


Fig. 4. A causal trace of Minecraft gameplay.

D. Gameplay as Causal Execution Trace

As an execution semantics, the transition relation $\Delta \to_r \Delta'$ is not deterministic; for any given game state, several rules may apply. For example, in the state $\{\mathsf{plank}^5\}$, both the craft_table and chop_plank rules apply. If we apply the first of these rules, the resulting state after rule application will be $\{\mathsf{table}, \mathsf{plank}\}$. If we choose the second, it will be $\{\mathsf{stick}^4, \mathsf{plank}^4\}$. The selection of the rule in this case can be thought of as delegated to an external decision-making source (such as a player or AI selection algorithm). In this sense, a collection of rules and an initial game state establish a *space of play*, where at any point, a rule that applies represents a valid move for the decision-maker to make.

Any given transition sequence represents a play trace, or a particular selection of actions (made by all decision-makers), resulting in a record or a history of the game state's evolution. Because we independently track which resources are used and generated by a given rule, we can actually represent this record with more structure than a simple sequence. Figure 4 shows a play trace using the running Minecraft example, which was presented linearly in Figure 3. Oval nodes represent resources (elements of the game state), and rectangular nodes represent transitions. An edge from a resource to a transition means that the transition consumed the resource, and an edge from a transition to a resource means that the transition produced the resource.

This graph structure allows us to observe *parallelism* in the play trace: for instance, the two initial sequences of chopping a tree followed by chopping wood into planks, have no dependencies (edges) connecting them, indicating that these sequences could be performed in arbitrary order with respect to one another (or simultaneously). Likewise, the structure reveals causal dependence and independence: we can see that chopping a plank into four sticks allowed us to split this resource so that half of it could be used for crafting the wood pickaxe, and then later, the rest could be used for crafting the stone pickaxe. This notion of independence between actions has been formalized in prior work through the notion of "concurrent equality" [70].

E. Term Quantification

The Minecraft-based example, with its emphasis on resource transformation and crafting has been useful to us for illustrating the basic principles of multiset rewriting, but fundamentally, this system is limited and difficult to scale. Most practical uses of multiset rewriting depend on predicate logic as a basis for multiset elements, which enables us to quantify over terms and write rule schema that may apply to multiple different instances. That is, for example, instead of representing a player inventory as a multiset whose elements are atomic resources (i.e. nullary predicates) such as plank, we would like the option to represent game state as a multiset of *predicates* that can take arguments, such as $\mathsf{inventory}(\mathsf{planks}, 3)$, where plank and 3 are *terms* and inventory is a predicate.

As soon as we introduce term arguments, it becomes important to be able to quantify over them: for example, now the chop_wood rule's antecedent needs to decrement the number of wood blocks and increment the number of planks in the player's inventory. To be fully general, we need to introduce a quantifier form $\forall x.\phi$ (for rules ϕ) so that rules can range over term variables. Then, a general decrementing rule can be written:

$$\begin{array}{rc} \mathsf{chop\_wood}_0 : & \forall W, P \\ & \mathsf{inventory}(\mathsf{wood}, W), \mathsf{inventory}(\mathsf{plank}, P) \\ \multimap & \mathsf{inventory}(\mathsf{wood}, W-1), \\ & \mathsf{inventory}(\mathsf{plank}, P+4) \end{array}$$

But what if we don't have any wood to chop? The rule ought no longer to apply, but technically 0 is a valid instantiation for the variable W. One approach to fixing this problem is to include an additional premise to the antecedent, $W > 0$, but this premise has a different nature to the others, since inequality is defined by general mathematical principles rather than by the presence of specific resources. We will introduce the ability to include predicates of this kind later, but for now, we present a more elegant solution based on *pattern matching*.

Predicates can be restricted to take arguments of certain *types*. The definition of types is discussed in section IV. We can then restrict W and P to be natural numbers and write

$$\begin{aligned}\mathsf{chop\_wood} : \quad & \forall W, P \\ & \mathsf{inventory}(\mathsf{wood}, W+1), \mathsf{inventory}(\mathsf{plank}, P) \\ \multimap \quad & \mathsf{inventory}(\mathsf{wood}, W), \mathsf{inventory}(\mathsf{plank}, P+4)\end{aligned}$$

This rule will only *apply* when there is a resource of the form $\mathsf{inventory}(\mathsf{wood}, W+1)$. That is, the second argument has to *match the pattern* of the successor of another natural number, i.e. be greater than 0.

We elide the formal definitions necessary to make this kind of pattern matching and transition semantics precise. In the next section, we describe our implementation of a programming language called Ceptre that enables a user to write computer-interpretable definitions of this nature.

F. Substitution

Once we introduce quantification, as above, we introduce predicates whose arguments can be *logic variables*. A rule over such predicates cannot apply to a state directly: first, we need to *match* concrete elements of the state to the abstract state *pattern* in the precondition of a rule. To define how a rule transforms a state, then, we need to describe how to *substitute* concrete terms for variables.

Definition 3 (Ground)**.** *We define any term, predicate, or rule as* ground *iff it contains no logic variables—that is, it only refers to concrete terms.*

For example, $\mathsf{at}(\mathsf{alice}, \mathsf{wonderland})$ is ground, but $\mathsf{at}(X, Y)$ (in the context of a rule where X and Y are bound variables) is not.

Multiset rewriting programs maintain the invariants that every state (multiset) consists only of ground predicates. That is, a rule should never introduce variables into the state. To maintain this invariant, we require that the right-hand side of a rule not contain any variables that don't also exist on the left-hand side.

Determining whether a rule applies now becomes equivalent to finding ground terms in the state that can be substituted for variables, so that the resulting predicates match the left-hand side of the rule:

Definition 4 (Rule Application)**.** *A rule $r : \forall \vec{x}.\ A \multimap B$* applies *to a state S iff there exist ground terms $\vec{t}$ such that $[t/x]A$ is a subset of S.*

The notation $[t/x]A$ can be pronounced "t for x in A" and refers to *substitution*, resulting in a new multiset A' in which all instances of x have been replaced by t.

Definition 5 (Substitution)**.** *$[t/x]M$, pronounced "substitute t for x in M," computes a new predicate in which each instance of the variable x occurring in M is replaced with t. M can be a predicate, term, proposition, or rule.*

For example, $[\mathsf{wood}/X]\mathsf{inventory}(X, 5)$ results in $\mathsf{inventory}(\mathsf{wood}, 5)$. We can also use similar notation for the simultaneous substitution of multiple terms for multiple variables, e.g. $[\mathsf{wood}/X, 5/Y]\mathsf{inventory}(X, Y) = \mathsf{inventory}(\mathsf{wood}, 5)$.

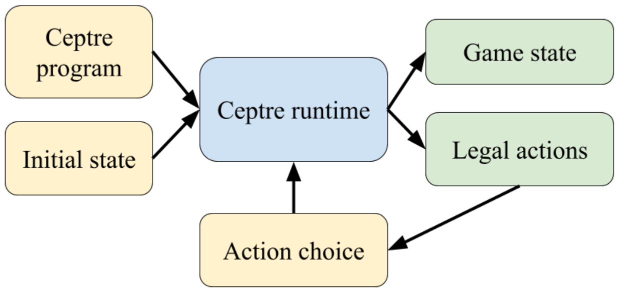


Fig. 5. A typical game modeling workflow with Ceptre. Light yellow nodes represent human-controlled input; light green nodes represent what Ceptre generates at runtime.

Defining substitution gives us the means to describe the operational semantics of the state rewrite:

Definition 6 (State Transition)**.** *A state S can rewrite to a state S' under rule $r : \forall \vec{x}.\ A \multimap B$ iff r matches S with substitution $[t/x]$, and $S' = S - [t/x]A \cup [t/x]B$.*

For example, suppose the state contains

$$\mathsf{inventory}(\mathsf{wood}, 3), \mathsf{inventory}(\mathsf{plank}, 0)$$

If we have the chop_wood rule as given previously, then the substitution $[2/W, 0/P]$ allows us to instantiate the rule as:

$$\begin{aligned}\mathsf{chop\_wood} : \quad & \mathsf{inventory}(\mathsf{wood}, 3), \mathsf{inventory}(\mathsf{plank}, 0) \\ \multimap \quad & \mathsf{inventory}(\mathsf{wood}, 2), \mathsf{inventory}(\mathsf{plank}, 4)\end{aligned}$$

Now the left-hand side of the rule exactly matches the two atoms in our state and we can apply the previous semantics for multiset rewriting.

IV. The Ceptre Language

Ceptre is a programming language implementation of the multiset-rewriting formalism described previously. The language includes constructs for user-defined sets of **terms**, first-order **predicates** that range over those terms, and **rewrite rules** that manipulate world states described in terms of (multisets of) those predicates. Authors write programs that consist of *signatures* of multiset rewriting rules, representing the space of possible actions for a specific game (as discussed in the previous section). Programs also include *initial game states*, represented in ceptre as *contexts* (multisets of ground predicates). Ceptre can then be run as an interpreter for this model that shows what specific actions apply at each step in the game trace, and either permits an action to be selected interactively or chooses nondeterministically from the legal actions. A diagram of the modeling workflow can be found in Figure 5.

We detail the components of a Ceptre program next, using a simplified model of the board game Pandemic [33] as a running example. A complete implementation of this example

Fig. 6. The Pandemic board game, set up with role assignments, player city card hands, and pieces in their initial location.

Predicate form	Argument types	Meaning
adjacent C C'	C, C' : city	C is adjacent on the map to C'
res_ctr C	C : city	C is a research center
disease C	C : city	C has a disease cube
at P C	P : player, C : city	P is located at C
turn P	P : player	P has a turn
hand P A	P : player, A : card	P has A in their hand
city_card A C	A : card, C : city	A is the city card for C
discard A	A : card	A is in the discard pile
draw P	P : player	P is obliged to draw a card

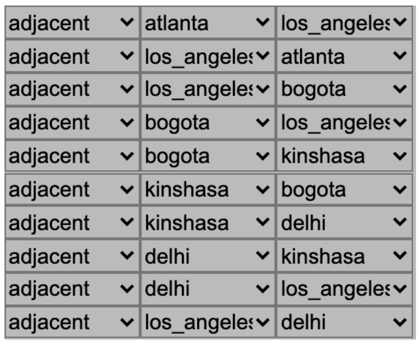
TABLE II
PREDICATES DEFINED FOR THE PANDEMIC EXAMPLE.

for web-based Ceptre is available on GitHub, and can be downloaded and imported into the hosted version of the web editor on Glitch.

A. The Ceptre Editor

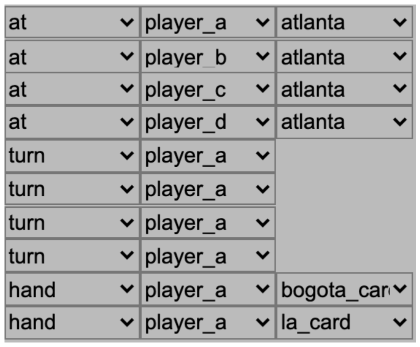
Since our target audience for Ceptre is game *designers*, not only programmers, we searched for design principles that would support this audience. Research on novice-friendly programming environments supports the use of *structure editors*, editors in which programmers manipulate program structure directly rather than text which is parsed into programs [45]. Structure editors include block-based programming tools, which have a demonstrable positive effect on learning for programming novices [71], leading to their adoption in introductory programming environments such as Scratch [37], Greenfoot [31], and Alice [27].

In light of these findings, we designed the Ceptre Editor, a web-based programming tool for a subset of the Ceptre language. The Ceptre editor uses standard HTML elements (drop-down menus, buttons, and checkboxes) to support direct editing of program structure, requiring the user to type only when first naming identifiers. After an identifier has been named, it will appear as an option in drop-down menus in appropriate locations. We will illustrate the worked example in this section with screenshots of Ceptre Editor programs in lieu of text-based syntax, though the program can be written in either format (and text-based Ceptre code can be automatically generated from the editor).



adjacent	atlanta	los_angele
adjacent	los_angele	atlanta
adjacent	los_angele	bogota
adjacent	bogota	los_angele
adjacent	bogota	kinshasa
adjacent	kinshasa	bogota
adjacent	kinshasa	delhi
adjacent	delhi	kinshasa
adjacent	delhi	los_angele
adjacent	los_angele	delhi

Fig. 7. Initial state predicates corresponding to the game map.



at	player_a	atlanta
at	player_b	atlanta
at	player_c	atlanta
at	player_d	atlanta
turn	player_a	
turn	player_a	
turn	player_a	
turn	player_a	
hand	player_a	bogota_car
hand	player_a	la_card

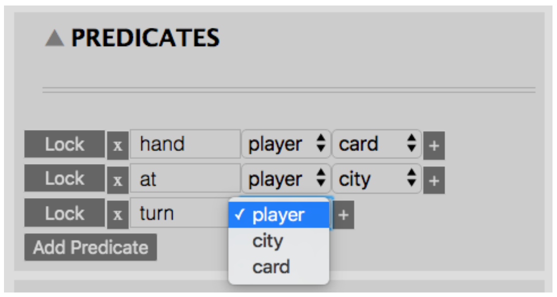
Fig. 8. Initial state predicates corresponding to the initial game configuration.

B. Game States as Multisets

A depiction of the *Pandemic* board game in play is shown in Figure 6. The game is cooperative, with each player contributing to the overall goal of mitigating and eradicating disease pandemics by traveling to infected locations (where level of infection is represented by an integral quantity of "disease cubes"), treating the sick (which removes a disease cube), and researching cures (by building research centers). Between each player's turn, the game randomly adds infection to cities and potentially propagates the disease to neighboring cities. Each player has four actions available on their turn.

For the sake of concise demonstration, we model a simplified rule set based on Pandemic. In our model, each player has the same set of abilities, that there is only one type of disease, and consists of a much smaller world map with just 5 cities. A game state consists of the contents of the deck of cards, the location of each player, the quantity of disease cubes on each city, the number of outbreaks that have occurred so far, and so on. In our simplified model, We can map each of these state variables to a logical predicate, e.g. the location C of players P to predicates $at(P, C)$. The full table mapping game state components is given in Table II.

Given this choice of vocabulary for representing game states, a specific game state, including the initial configuration, can be represented by a multiset of *ground* predicates—that is, predicates whose arguments are concrete terms, with no variables. The code in Figure 7 shows how the multiset representing (part of) the initial world map is written in the Ceptre Editor, and Figure 8 shows the same for (part of) an initial game state in which a player starts off with four turns.



hand	player	card	
at	player	city	
turn	✓ player		

Fig. 9. Defining predicates by giving their argument types in the Ceptre Editor.

C. Declaring Type and Term Constants

In order to facilitate readable and bug-free domain authoring, Ceptre programs include *declarations* for typed terms and predicates, which the author uses to define the valid syntax of game rules. For example, to represent "a character is at a location," the predicate `at` must take two arguments, a character and a location. This design means early failure in the case of mistakes like misspelled argument names, swapped arguments, or missing arguments.

Users define types by providing an identifier for the type and an enumeration of what elements are members of that type. For example, when specifying the rules of *Pandemic*, we can create a type for cities called `city`. Then, we can create *term constants* for each individual city, declaring each to have the type `city`. The Ceptre editor supports *only* these simple enumerative types and calls them "sets."

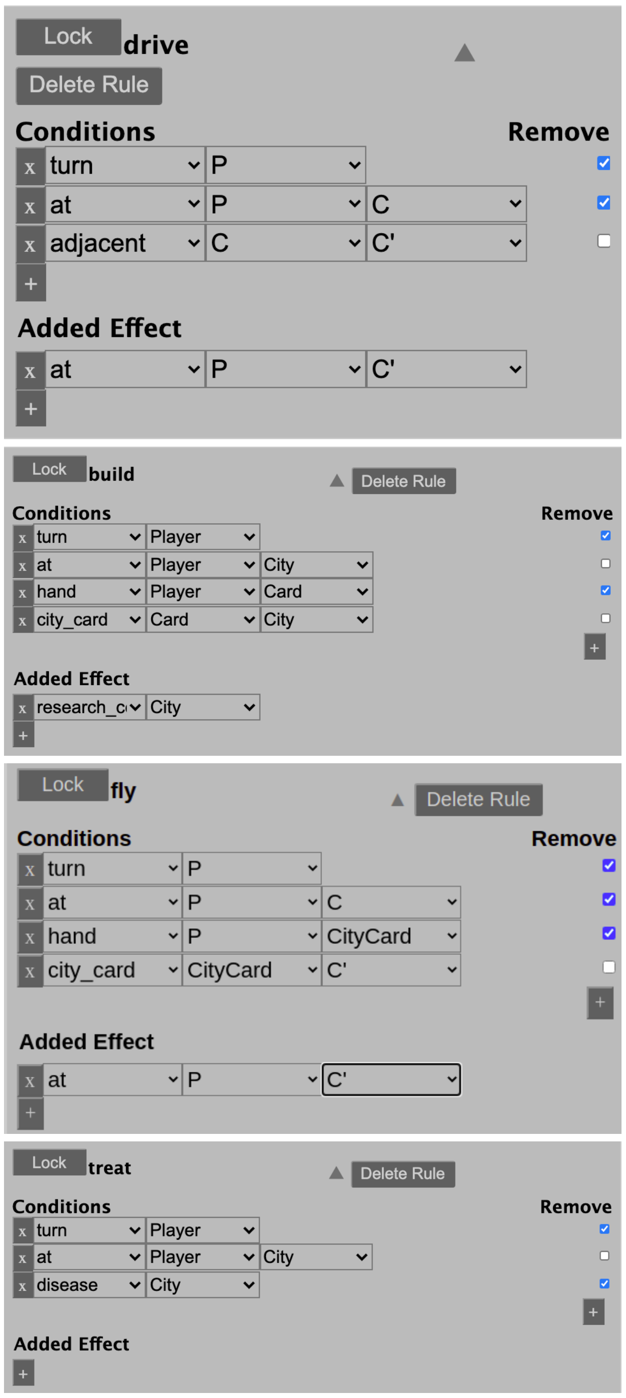
Once the user has declared the types (or sets) of term constants, they can define predicates by giving their type signatures. This process is depicted in Figure 9. The dropdown menus in the Ceptre Editor are automatically populated with the sets defined by the user.

D. Game Mechanics as Multiset Rewrite Rules

After defining our representation language, we can write rules that correspond to gameplay mechanics in Pandemic. For example, on each player's turn, they have four available actions (represented by the presence of four "turn" tokens for a given player). They can spend these actions on any of several choices of moves, including moving around the map, treating disease cases (removing disease cubes from the map), and building research centers. The "drive" move, for example, allows a player to spend a turn to move directly to any adjacent location on the map. Using the mathematical syntax from Section III, we can represent this action as follows:

$$\forall p, c, c'.\ \mathsf{turn}(p), \mathsf{at}(p, c), \mathsf{adjacent}(c, c') \quad \multimap \\ \mathsf{at}(p, c'), \mathsf{adjacent}(c, c')$$

This expression is represented by the first snippet of Ceptre code shown in Figure 10. Note that rather than requiring the author to repeat "permanent" information, like the edges between adjacent cities, in both the preconditions and effects of the rule, we give every condition a check-box for whether it should be *removed* from the state or not. Since the player must spend a turn, and since they will no longer be *at* their previous location after making this move, those predicates are removed, but the adjacency predicate remains.



drive

Conditions			Remove
turn	P		✓
at	P	C	✓
adjacent	C	C'	☐

Added Effect		
at	P	C'

build

Conditions			Remove
turn	Player		✓
at	Player	City	☐
hand	Player	Card	✓
city_card	Card	City	☐

Added Effect	
research_c	City

fly

Conditions			Remove
turn	P		✓
at	P	C	✓
hand	P	CityCard	✓
city_card	CityCard	C'	☐

Added Effect		
at	P	C'

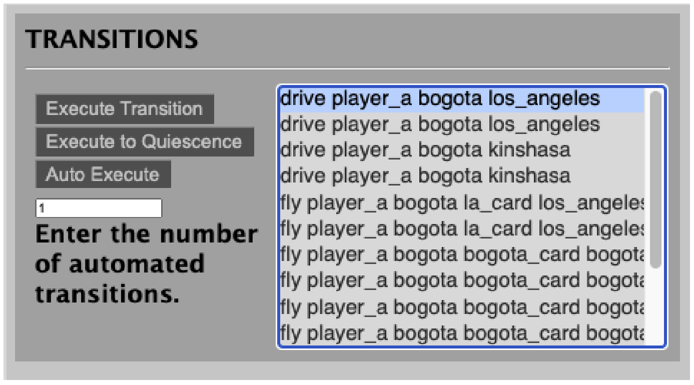
treat

Conditions			Remove
turn	Player		✓
at	Player	City	☐
disease	City		✓

Added Effect

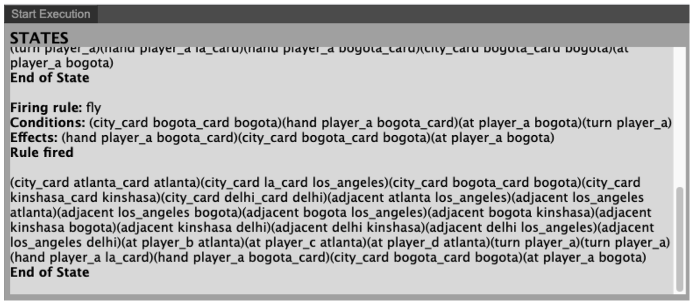
Fig. 10. Player moves in Pandemic (drive, build, fly, and treat), represented as a Ceptre rules.

Three additional examples are given in Figure 10, showing encodings of the Pandemic rules that enable players to fly to non-adjacent locations (if they hold the corresponding city card), treat disease cases in their current location, and build research centers in cities for which they also possess the city card.



```
drive player_a bogota los_angeles
drive player_a bogota los_angeles
drive player_a bogota kinshasa
drive player_a bogota kinshasa
fly player_a bogota la_card los_angele
fly player_a bogota la_card los_angele
fly player_a bogota bogota_card bogot
fly player_a bogota bogota_card bogot
fly player_a bogota bogota_card bogot
fly player_a bogota bogota_card bogot
```

Fig. 11. The Ceptre Editor interface for browsing and selecting transitions manually or initiating automatic selection.



(turn player_a)(hand player_a la_card)(hand player_a bogota_card)(city_card bogota_card bogota)(at player_a bogota)
End of State

Firing rule: fly
Conditions: (city_card bogota_card bogota)(hand player_a bogota_card)(at player_a bogota)(turn player_a)
Effects: (hand player_a bogota_card)(city_card bogota_card bogota)(at player_a bogota)
Rule fired

(city_card atlanta_card atlanta)(city_card la_card los_angeles)(city_card bogota_card bogota)(city_card kinshasa_card kinshasa)(city_card delhi_card delhi)(adjacent atlanta los_angeles)(adjacent los_angeles atlanta)(adjacent los_angeles bogota)(adjacent bogota los_angeles)(adjacent bogota kinshasa)(adjacent kinshasa bogota)(adjacent kinshasa delhi)(adjacent delhi kinshasa)(adjacent delhi los_angeles)(adjacent los_angeles delhi)(at player_b atlanta)(at player_c atlanta)(at player_d atlanta)(turn player_a)(turn player_a)(hand player_a la_card)(hand player_a bogota_card)(city_card bogota_card bogota)(at player_a bogota)
End of State

Fig. 12. The Ceptre Editor interface for viewing the sequence of states and actions.

E. Execution

Running a Ceptre program is possible after the rules and initial state have been provided. Because execution is nondeterministic, the system enables the user to “steer” the simulation manually, or alternatively to allow uniform random selection at each step from the space of all possible transitions.

A *transition* in Ceptre is a single step of program execution, or equivalently, an *instantiated* rule: that is, a rule where all variables have been substituted with specific terms so that its precondition matches a segment of the state. Transitions are thus presented as the name of a rule followed by its “arguments,” or substituted terms, that can replace the rule’s logic variables to make its precondition match the state. Many such transitions may be possible in any given state, so Ceptre generates a list of all possible transitions. When the Ceptre rules represent game mechanics, the list of possible transitions represents the set of “moves” available to the player. See Figure 11 for a screenshot of how Ceptre would generate the available moves for our encoding of Pandemic. [2]

Transitions can be selected manually, if the program is run in *interactive* mode, or automatically at random. As transitions are selected, the state evolves, according to the semantics described in Section III. A trace of this activity is shown to the user in the form of a stream of alternating actions and states

[2]Because Ceptre distinguishes between copies of the “same” atom in the multiset, in some cases, identical transitions will be listed more than once. For interactive Ceptre programs, this can be regarded as a presentational quirk that can be remedied on the rendering side where it is undesired.

```
stage play = {
   % Rules for player actions...
} #interactive play.

stage play -o stage auto.

stage auto = {
   % Rules for automatic actions...
}

stage auto -o stage play.
```

Fig. 13. Skeleton of a stage-based Ceptre program.

(see Figure 12). The simulation stops running when the user chooses to halt execution or it reaches natural quiescence.

F. Stages

Certain idioms are difficult to express in pure multiset rewriting with the simple quiescence semantics (stop when no more rules apply) that we have given in the previous section. In the example of Pandemic, for instance, after the player completes a full move, the “game” or external world gets a turn to propagate diseases and pass control over to the next player in sequence. Making one set of rules interactive (controlled by the player) and one set of rules automatic (running the “game’s turn”) is a key element of most game design, but not possible to represent in the multiset rewriting formalism.

In order to support interactive/automatic turn alternation, among other common game rule idioms, we invent a language construct called a *stage*, which is a named collection of rules that runs to quiescence, and inter-stage rules, which pass control among stages. For a sketch of how stage-level programming looks, see Figure 13.

The full, text-based version of Ceptre supports stages, and we have implemented a stage-based version of Pandemic, available online, as a supplement to the program presented here. However, the Ceptre Editor currently lacks stages as a feature, so the worked example shown here focuses on what we can represent without them. A more complete discussion of stages can be found in Martens’s Ph.D. thesis [39].

G. Implementation

Both of Ceptre’s implementations rely on standard rewrite-rule-based language implementation techniques, such as term matching to enumerate the set of possible rule applications [55]. The text-based language is written in Standard ML and uses YACC for parsing. The web-based editor is written in entirely client-side HTML and JavaScript. The source for both implementations is on GitHub: text-based, web editor. Downloadable binaries are available for text-based Ceptre.

V. Evaluating Ceptre’s Usability and Usefulness

Ceptre is a multidisciplinary project, spanning human-computer interaction (HCI), programming languages (PL), formal logic, and game design, so its evaluation requires a multidisciplinary approach. Traditionally, programming languages and logics are evaluated through formal proof: mathematical definitions lend themselves to asking formal questions like whether it is possible to write certain programs or prove certain theorems. HCI gives us tools to understand a language in terms of its affordances for human interaction, which of course depends not only mathematical properties of the language, but also its syntax, tooling, ecosystem, and contexts in which the language is being learned. All of these factors together affect the degree to which people can express their intended meaning through the language, reason about the programs they write, and gain useful insights from their models.

Thus, we discuss evaluation in three ways: first, by describing some formal properties of the language that allow us to compare and contrast it with other formalisms; second, by describing an empirical, qualitative study of the Ceptre Editor tool; and finally, by describing in less formal terms the way that Ceptre has been used by and influenced other researchers, practitioners, and game designers.

A. Formal Properties

1) Compositionality: A key idea from programming language design *compositionality*, which enables programmers to reason about (and write code for) programs in terms of their constituent parts independently. This idea shows up in many forms, including the idea of "context-free" grammars for parsing, as well as implications for the implementation of language interpreters, type systems, static analyses, and compiler transformations [26]. Compositionality is often seen as a key design requirement for programming languages, program synthesis engines, and compilers, as evidenced by a sample of highly influential work in these fields ([19], [1], [49]).

Formally, a compositionality property for a function f and combinator $\cdot$ is one that takes the form, "$f(X \cdot Y) = f(X)\hat{\cdot}f(Y)$." X, Y, and $X \cdot Y$ are all members of some set A closed under the combinator $\cdot$, and $\hat{\cdot}$ is a translation of $\cdot$ to the codomain of f (i.e., if $f : A \to B$, then $\_ \cdot \_ : A \times A \to A$, and $\_ \hat{\cdot} \_ : B \times B \to B$).

Ceptre is compositional in the following senses: first, *compositionality of program execution* can be stated as follows:

Compositionality of Program Execution. For any contexts Δ, Δ', and Γ, if

$$\Delta \to^* \Delta'$$

then

$$\Delta, \Gamma \to^* \Delta', \Gamma$$

That is, adding some *extra stuff* (here represented by Γ) to the program as it runs does not change the fact that it can run the same way it ran *without* that extra stuff. Note that, because the transition relation $\to$ is nondeterministic (the same context can evolve to multiple different successor contexts), this does not rule out the possibility that Γ might contribute new ways for the program to evolve. It is entirely possible that there is some program trace that witnesses $\Delta, \Gamma \to^* \Theta$, where $\Delta \not\to^* \Theta$—but it is always *possible* to evolve the program so that the elements of Γ don't interact with Δ.

Notably, this property does *not* hold of languages like PDDL, which are superficially similar to multiset rewriting—actions in PDDL specify logical facts that are true before and after the execution of the action—but which also permit "deletion" of facts that do not appear in the premises (preconditions) of the rule. As a consequence, an action would *have a different effect* on the world state depending on whether or not the deleted fact is true before the execution of that action.

A second sense in which Ceptre is compositional has implications for the programmer's ability to reason about programs as they write them. In essence, any given rule within a Ceptre program (or, more accurately, within an individual stage) can be reasoned about independently of the other rules in the program. That is, within the program

$$\begin{aligned} &\ldots \mathsf{Pre} \ldots \\ \mathsf{rule} \ :\ & A \multimap B \\ &\ldots \mathsf{Post} \ldots \end{aligned}$$

The rule rule has the same meaning (the same effect on the context if it is selected during execution) as it would in any other program where Pre and Post were different. This feature distinguishes Ceptre from other game description languages like Kodu and Puzzlescript, in which the order of rules has meaning (related to priority when more than one rule applies). In these settings, a rule's meaning can only be determined in the context of all rules that precede it. Similar complexities arise if we add probabilities or weights to rules. Ceptre's introduction of stages attempts to contain this complexity by introducing ordering effects at stage boundaries, but not within an individual stage.

2) Decidability of Reachability: One common reaction to Ceptre's surface-level similarity to the planning formalism PDDL is an interest in static analysis of game specifications: could we, for instance, write a set of rules and then use a search algorithm to definitively prove whether a certain game state is reachable from a starting state?

This question is formally known as *reachability* (can we reach one state from another), and the question of its decidability depends on which fragment of multiset rewriting we are considering. For example, the fragment that corresponds with planning problems does indeed support search using similar algorithms. This fragment limits the term language to be finite by disallowing e.g. numeric constants in rules (such as those seen in Section III-E). For an analysis of reachability problem decidability in a similar formalism (Petri nets), see prior work [23], [18].

This decidable fragment is necessarily Turing-incomplete, and more specifically, it limits the programmer's ability to define and process data in common (inductively-defined) formats like lists and numbers. In essence, when deciding which features to support, we make a tradeoff between the expressiveness of the language and our ability to systematically analyze programs in that language. The Ceptre implementation described in this paper supports more expressiveness at the cost of decidable analysis, but it is possible that in the future we will implement decision procedures for the fragments of the language that support it.

B. Human Subjects Evaluation of the Ceptre Editor

We conducted a formative study, described in depth in previous work [8], to understand how well the web-based programming environment for Ceptre described in Section IV supports learners who are new to Ceptre in understanding and building on examples. We briefly summarize the method and results of this study here as evidence of Ceptre's usability.

We recruited 8 participants without requiring any prior programming or logic experience. In practice, all participants had some prior programming experience (though some were novices), but most had no formal logic training. Participants completed a brief, 15 minute guided tutorial based on the "Blocks World" domain common in symbolic AI research [59], in which a robot arm can pick up and place stackable blocks. We gave participants a guided tutorial in which they created the block type, the "on" predicate, and first rule (for picking up a block). We then asked them to complete the rest of the model on their own. Following a think-aloud protocol [68], we asked participants to extend the tutorial program several times while talking through their thought process as they worked. The three program extension tasks were to add (1) a second robot arm, (2) an additional block, and (3) a termination condition for the program.

We collected the programs authored by participants, the amount of time it took them to complete each task, and both screen and audio recordings (later transcribed) of their think-aloud narration. We found that participants unanimously succeeded in completing the blocks world tutorial in the time given (15 minutes), and that the majority of the participants (7 out of 8), were able to complete each of the subsequent extensions within 30 minutes. Participants' behaviors with the tool and think-aloud narration demonstrated that participants understood both the logic of the program structure and the semantics of the model: for example, several participants experimented with the model by running tests without our prompting them to. Additionally, over the course of participants' engagement with the tool, the way that they described Ceptre programs changed. For example, at first, participants read the states and transitions between states directly from the program, e.g. "*on c a and clear c*" and "*pickup from block c a left*." But as they continued running the program and constructing extensions to the model, they began referring to the program and its execution using the language of the representation domain, e.g. "*let's pick up c with our left arm*" and "*so c is on a, and c is clear*." The results of this study support our hypothesis that the Ceptre language, supported by the web-based editing environment, allows novice users construct, understand, and modify a model after completing a 15-minute guided tutorial.

C. Applications

The first author originally developed Ceptre as a Ph.D. thesis project [39], and it has since been used in several published projects within their own lab, such as character simulation-based narrative generation, modeling the structure of phatic conversation [48], and generating quests in Minecraft [2]. The project of specifying games using multiset rewrite rules has also had indirect influence on other projects, including the Cygnus game definition language underlying the Gemini game generation engine [65], the Puzzlescript language for writing 2D tile-based puzzle games [32] (which itself has been used in game generation and modeling research [35], [41]), Microsoft's TextWorld system [13] for designing and evaluating reinforcement learning algorithms for understanding of virtual worlds, and a natural language interface developed for a Wesleyan University honors thesis [50].

Finally, we speculate that Ceptre would make a compelling tool to support many applications of general interest within computational game design support as future research projects. These include authoring rule-based AI systems, as a simpler alternative to reactive planning-style systems (e.g. ABL [42]) or Rete rules [46]; interactive simulations to support complex systems understanding and informal science learning, as has been achieved with visual programming environments in NetLogo [67] and AgentSheets [56] (see also related work on modeling biological systems with multiset rewriting [3]); as a Game Description Language for tuning game designs based on formally-defined properties like balance and drama, as in Cameron Browne's work [6]; formal reasoning about play traces, as in Playspecs [52]; and formal reasoning about progressions and branching narratives, as in the "scenario validation" problem described by Dong et al. [14].

VI. Discussion

The design of Ceptre represents an effort to follow certain language design design principles in order to achieve a way of expressing game mechanics with minimal cognitive overhead. We now discuss the principles that we believe have made this project a success, as well as limitations of adhering to these principles as design constraints. We put forth these design principles as an example that others may follow if they wish to replicate similar results in other language design projects.

1) Language Minimalism: Multiset rewriting fundamentally consists of just two constructs, the multiset (joining together predicates on either side of a rule) and the rewrite rule. The Ceptre language adds to this framework conservatively, introducing user-defined types, typed predicate declarations, and (in the full command-line version) Prolog-style relational definitions and stages. These 5-6 ideas are all that is necessary to understand for a beginner to get started with the formalism, yet it is also all one needs to build up rich and complex simulations.

Language minimalism as a concept is not just about the number of language constructs, but also how easy it is to formally describe their operational semantics, i.e. how a program that uses them will behave when it is run. This paper provides the formal semantics of the language in order to demonstrate that it is possible to describe the meaning of each language construct concisely and with simple mathematical machinery. We believe that this paper is the *first* in the space of game modeling (or game definition) languages to do so, but we hope we are not the last!

2) Genre-Agnostic Extensibility: Unlike game description languages like VGDL and PuzzleScript, Ceptre does not assume a specific underlying set of game constructs or operational logics specific to particular game genres (e.g. collision, enemies, tile grids, spatial movement), or indeed even specific to games. All of these concepts can be encoded by the programmer, but there are no built-in assumptions about their meaning.

We see this property of generality as a strength in the sense that Ceptre permits game designers to think experimentally and flexibly about what a game can contain or represent: since Ceptre can model cryptographic protocols and ecological systems, for example, it is easier to use it to imagine and prototype games about decoding encrypted messages or healing disrupted ecosystems. It also makes it possible to combine, compare, and remix games that use different operational logics, since it provides a common basis for encoding all of them. These use cases are demonstrated by the myriad corresponding examples in the Ceptre code base repository, many of which are documented in publications.

Of course, generality can also be seen as a weakness, since genre- and operational logic-specificity enables specific tooling (such as visual rendering) that leans on the assumptions of those logics to provide designers with visualizations and reasoning modes specific to, e.g., tile-based puzzle games or collisions in 2D space. However, we think it is possible to have one's cake an eat it, too: an avenue of ongoing work for Ceptre is to design a puzzle game subset (collection of predicate definitions and rule templates) and corresponding visualization and testing tools for this subset.

3) Limitations and Tradeoffs: As discussed briefly in Section V-A, the preservation of certain formal properties, like decidability of reachability from one state to another, is in tension with the expressiveness of the language used to describe that program, like the usage of non-finite data structures (e.g. natural numbers). Ceptre's design chooses to support more expressivity in the case of its data structure language, but on the other hand, it limits expressivity in the rule language (e.g. by not supporting negation or rule prioritization) in order to support other formal properties, like compositionality. We believe that reasonable language designers can make different decisions about each of these tradeoffs, but it is important to understand the consequences of these decisions. Within Ceptre's implementation, we make different choices for different language fragments: the "pure" multiset rewriting fragment of the language has limited expressiveness, but high amenability to analysis; adding stages affords more expressiveness at the expense of amenability to formal analysis. Within a stage, we can reason about every rule set compositionally, but staged programs as a whole sacrifice compositionality in the interest of expressive power.

VII. Conclusion

With this paper, we contribute a tutorial explanation of multiset rewriting as a reproducible formalism in which objects and conditions in games can be represented as multisets of first-order logic predicates and actions that modify these multisets. We also contribute a technical description of Ceptre, a programming language that implements this formalism, and its web-based structure editor, which allows non-expert users to express complex procedural ideas in this formalism without struggling with unfamiliar syntax. Ceptre has been and can be used to rapidly prototype and analyze a wide variety of games and non-game systems, facilitated by its minimalism as a language and formal properties like compositionality. In the long term, we hope to see multiset rewriting adopted in many programming languages and contexts, similar to context-free string grammars and cellular automata, with diverse ecosystems of tools built atop it to facilitate interdisciplinary community-building and cross-domain procedural literacy.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1755922. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Amal Ahmed. Compositional compiler verification for a multi-language world. In *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[2] Ryan Alexander and Chris Martens. Deriving quests from open world mechanics. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, pages 1–7, 2017.

[3] Stefano Bistarelli, Iliano Cervesato, Gabriele Lenzini, Roberto Marangoni, and Fabio Martinelli. On representing biological systems through multiset rewriting. In *International Conference on Computer Aided Systems Theory*, pages 415–426. Springer, 2003.

[4] Pierre Boutillier, Mutaamba Maasha, Xing Li, Hector F Medina-Abarca, Jean Krivine, Jérôme Feret, Ioana Cristescu, Angus G Forbes, and Walter Fontana. The Kappa platform for rule-based modeling. *Bioinformatics*, 34(13):i583–i592, 06 2018.

[5] Marco Bozzano and Giorgio Delzanno. Automatic verification of secrecy properties for linear logic specifications of cryptographic protocols. *Journal of Symbolic Computation*, 38(5):1375–1415, 2004.

[6] Cameron Browne and Frederic Maire. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):1–16, 2010.

[7] Cameron Browne, Dennis J. N. J. Soemers, Eric Piette, Matthew Stephenson, and Walter Crist. Ludi language reference. Online at https://ludii.games/downloads/LudiiLanguageReference.pdf, 2021.

[8] Alexander Card and Chris Martens. The ceptre editor: A structure editor for rule-based system simulation. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 133–137. IEEE, 2019.

[9] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and computation*, 179(1):19–75, 2002.

[10] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework ii: Examples and applications. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2003.

[11] Kaustuv Chaudhuri and Joëlle Despeyroux. A hybrid linear logic for constrained transition systems with applications to molecular biology. *arXiv preprint arXiv:1310.4310*, 2013.

[12] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and José F Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[13] Marc-Alexandre Côté, Akos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, et al. Textworld: A learning environment for text-based games. In *Workshop on Computer Games*, pages 41–75. Springer, 2018.

[14] Kim Dung Dang, Ronan Champagnat, and Michel Augeraud. Modeling of interactive storytelling and validation of scenario by means of linear logic. In *Joint International Conference on Interactive Digital Storytelling*, pages 153–164. Springer, 2010.

[15] Henry DeYoung and Carsten Schürmann. Linear logical voting protocols. In *International Conference on E-Voting and Identity*, pages 53–70. Springer, 2011.

[16] Joris Dormans. Simulating mechanics to study emergence in games. In *Workshops at the Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011.

[17] Marc Ebner, John Levine, Simon M Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. Towards a video game description language. *Artificial and Computational Intelligence in Games (Dagstuhl Follow-Ups)*, 2013.

[18] Javier Esparza and Mogens Nielsen. Decidability issues for petri nets. *BRICS Report Series*, 1(8), 1994.

[19] Ian Foster. Compositional parallel programming languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):454–476, 1996.

[20] Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI magazine*, 26(2):62–62, 2005.

[21] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl: The planning domain definition language. Technical report, Yale Center for Computational Vision and Control, 1998.

[22] Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, Tiago Machado, Andy Nealen, and Julian Togelius. Atdelfi: automatically designing legible, full instructions for games. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, pages 1–10, 2018.

[23] Michel Henri Théodore Hack. *Decidability questions for Petri Nets*. PhD thesis, Massachusetts Institute of Technology, 1976.

[24] Robert Harper and Daniel R Licata. Mechanizing metatheory in a logical framework. *Journal of functional programming*, 17(4-5):613–673, 2007.

[25] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. An introduction to the planning domain definition language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(2):1–187, 2019.

[26] Theo MV Janssen and Barbara H Partee. Compositionality. In *Handbook of logic and language*, pages 417–473. Elsevier, 1997.

[27] Caitlin Kelleher, Dennis Cosgrove, David Culyba, Clifton Forlines, Jason Pratt, and Randy Pausch. Alice2: programming without syntax errors. *User Interface Software and Technology*, 01 2002.

[28] Ahmed Khalifa, Diego Perez-Liebana, Simon M Lucas, and Julian Togelius. General video game level generation. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 253–259, 2016.

[29] Athar Kharal, Mansoor H. Alshehri, Nasser Bin Turki, and Faisal Z Duraihem. Generalized mapping for multiset rewriting systems. *Soft Computing*, 25(17):11439–11448, 2021.

[30] Paul Klint and Riemer Van Rozen. Micro-machinations. In *International Conference on Software Language Engineering*, pages 36–55. Springer, 2013.

[31] Michael Kölling. The greenfoot programming environment. *Trans. Comput. Educ.*, 10(4):14:1–14:21, November 2010.

[32] Stephen Lavelle. Puzzlescript. Online at https://www.puzzlescript.net.

[33] Matt Leacock. Pandemic. Board game published by Z-Man Games, 2003.

[34] John Levine, Clare Bates Congdon, Marc Ebner, Graham Kendall, Simon M Lucas, Risto Miikkulainen, Tom Schaul, and Tommy Thompson. General video game playing. *Artificial and Computational Intelligence in Games (Dagstuhl Follow-Ups)*, 2013.

[35] Chong-U Lim and D Fox Harrell. An approach to general videogame evaluation and automatic generation using a description language. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[36] Stencyl LLC. Gamemaker. Online at https://www.stencyl.com/.

[37] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16, 2010.

[38] Chris Martens. Ceptre: A language for modeling generative interactive systems. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[39] Chris Martens. *Programming interactive worlds with linear logic*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, 2015.

[40] Chris Martens, Joao F Ferreira, Anne-Gwenn Bosser, and Marc Cavazza. Generative story worlds as linear logic programs. In *Seventh Intelligent Narrative Technologies Workshop*, 2014.

[41] Chris Martens, Aaron Williams, Ryan S Alexander, and Chinmaya Dabral. Generating puzzle progressions to study mental model matching. In *AIIDE Workshops*, 2018.

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

[43] Microsoft Research. Kodu. Online at https://www.microsoft.com/en-us/research/project/kodu/.

[44] Dale Miller. A multiple-conclusion meta-logic. In *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 272–281. IEEE, 1994.

[45] Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann. Evolution of novice programming environments: The structure editors of carnegie mellon university. *Interactive Learning Environments*, 4(2):140–158, 1994.

[46] Ian Millington and John Funge. *Artificial intelligence for games*. CRC Press, 2018.

[47] Mojang. Minecraft. Online at https://www.mojang.com/games/, 2011.

[48] Hannah Morrison and Chris Martens. "how was your weekend?" a generative model of phatic conversation. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, pages 1–7, 2017.

[49] Maxwell Nye, Armando Solar-Lezama, Josh Tenenbaum, and Brenden M Lake. Learning compositional rules via neural program synthesis. *Advances in Neural Information Processing Systems*, 33:10832–10842, 2020.

[50] Samantha Ong. Talking to ceptre: A natural language interface. *Wesleyan University Honors Theses*, 2018.

[51] Jon Orwant. *EGGG: The Extensible Graphical Game Generator*. PhD thesis, Massachusetts Institute of Technology, 2000.

[52] Joseph Osborn, Ben Samuel, Michael Mateas, and Noah Wardrip-Fruin. Playspecs: Regular expressions for game play traces. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 11, pages 170–176, 2015.

[53] Joseph Carter Osborn, April Grow, and Michael Mateas. Modular computational critics for games. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[54] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon M Lucas, Adrien Couëtoux, Jerry Lee, Chong-U Lim, and Tommy Thompson. The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(3):229–243, 2015.

[55] Peter Raulefs, J Siekmann, Peter Szabó, and E Unvericht. A short survey on the state of the art in matching and unification problems. *ACM Sigsam Bulletin*, 13(2):14–20, 1979.

[56] Alexander Repenning, Andri Ioannidou, and John Zola. Agentsheets: End-user programmable simulations. *Journal of Artificial Societies and Social Simulation*, 3(3):351–358, 2000.

[57] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, Zachary Tatlock, et al. Qed at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, 5(2-3):102–281, 2019.

[58] Grigore Rosu. K: a rewrite-based framework for modular language design, semantics, analysis and implementation—version 2. Technical report, University of Illinois at Urbana-Champaign, 2006.

[59] Stuart Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson Education, Inc, 2003.

[60] Anders Schack-Nielsen and Carsten Schürmann. Celf–a logical framework for deductive and concurrent systems (system description). In *International Joint Conference on Automated Reasoning*, pages 320–326. Springer, 2008.

[61] Tom Schaul. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, 2013.

[62] Adam M Smith, Mark J Nelson, and Michael Mateas. Computational support for play testing game sketches. In *Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2009.

[63] Adam M Smith, Mark J Nelson, and Michael Mateas. Ludocore: A logical game engine for modeling videogames. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 91–98. IEEE, 2010.

[64] Adam Summerville, Chris Martens, Sarah Harmon, Michael Mateas, Joseph Osborn, Noah Wardrip-Fruin, and Arnav Jhala. From mechanics to meaning. *IEEE Transactions on Games*, 11(1):69–78, 2017.

[65] Adam Summerville, Chris Martens, Ben Samuel, Joseph Osborn, Noah Wardrip-Fruin, and Michael Mateas. Gemini: Bidirectional generation and analysis of games via asp. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 14, 2018.

[66] Michael Thielscher. Gdl-iii: A description language for epistemic general game playing. In *The IJCAI-16 workshop on general game playing*, page 31, 2017.

[67] Seth Tisue and Uri Wilensky. Netlogo: A simple environment for modeling complexity. In *International conference on complex systems*, volume 21, pages 16–21. Boston, MA, 2004.

[68] Maarten Van Someren, Yvonne F Barnard, and J Sandberg. The think aloud method: a practical approach to modelling cognitive. *London: AcademicPress*, 11, 1994.

[69] Noah Wardrip-Fruin and Michael Mateas. Defining operational logics. In *DiGRA &#3909 - Proceedings of the 2009 DiGRA International Conference: Breaking New Ground: Innovation in Games, Play, Practice and Theory*, volume 5, September 2009.

[70] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework i: Judgments and properties. Technical report, Carnegie Mellon University, 2003.

[71] David Weintrop and Uri Wilensky. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Trans. Comput. Educ.*, 18(1):3:1–3:25, October 2017.

[72] Michael Winikoff and James Harland. Implementing the linear logic programming language lygon. In *ILPS*, pages 66–80. Citeseer, 1995.

[73] YoYo Games and Mark Overmars. GameMaker Studio. Online at https://www.yoyogames.com/en/gamemaker.