Towards a Unified Language for Card Game Design

Riemer van Rozen
rozen@cwi.nl
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands

Anders Bouwer
a.j.bouwer@hva.nl
AUAS
Amsterdam, The Netherlands

Karel Millenaar
k.millenaar@hva.nl
FourceLabs and AUAS
Amsterdam, The Netherlands

ABSTRACT

Card game creation is a powerful tool for game design. Using playing cards, game designers can rapidly prototype and iteratively playtest a game’s core mechanisms to explore alternatives and improve the gameplay. However, this process is time-consuming, imprecise and challenging to steer and focus. We aim to empower designers with solutions that automate game design processes. In particular, we study to what extent a unified game design language can offer theoretical foundations, systematic techniques and practical solutions.

We propose a novel approach towards a solution that leverages the expressive power of playing cards. Initially focusing on well-known card games, we illustrate the steps for creating CardScript, a formal language and toolkit that supports game design processes.

The approach also has the potential to impact a wider research area. When fully developed, a unified language with a common tool set can enable reuse, and eventually support joint research agendas. We start the discussion by highlighting perspectives that relate open challenges to opportunities for future collaboration.

CCS CONCEPTS

- Software and its engineering → Domain specific languages;
  - Integrated and visual development environments;
- Applied computing → Computer games.

KEYWORDS

game design, design tools, domain-specific languages

ACM Reference Format:

1 INTRODUCTION

Card games provide powerful metaphors for rules of play. For game designers, rapid prototyping and effective playtesting are crucial for timely results [19, 21]. Many designers carry a prototyping kit with game pieces, dice, chips or a card deck. Using playing cards, they can rapidly create “cardboard prototypes”, abstract crude game representations for playtesting core mechanisms at a table in a mediated setting [10]. This practice has proven incredibly effective for flexibly changing the rules and rapidly improving the gameplay.
enables formalizing designs of collaborative games such as Hanabi, and helps in analyzing specifications by using simple metrics [24]. In meetings and workshops with Dutch indie game developers, we have obtained valuable feedback on presentations, demos, and tests. We report our experiences, starting with the following insights.

Designers create card games for a variety of reasons, some to publish, others to balance real-time RPGs. Visual interfaces are appreciated, but textual rule notations quickly become too complex. To avoid visual clutter, tools require modules and ways to hide and show them. Unfortunately, without a reusable semantics we cannot easily build on, extend or improve any of these tools. As a result, we cannot yet translate or reuse the game designs either.

To address these issues, we envision a set of complementary and compatible design tools built on a platform-independent card game formalism. Therefore, we take a step back from the more complex examples and initially focus on shedding and matching games. Next, we describe a structured approach towards a common solution.

3 TOWARDS CARDSCRIPT

We describe an approach and initial steps towards a practical solution named CardScript, a so-called Domain-Specific Language (DSL). A DSL is an executable specification language that offers domain experts a notation with improved expressive power over a restricted domain, in this case card games. This approach originates in the field of Software Engineering. We apply well-known techniques and use language workbench Rascal in its construction [8, 12]. First, we analyze the requirements of the DSL and its tools.

3.1 Requirements

Game designers are the main stakeholders and intended users. In vivid discussions we have obtained feedback and criticism that have contributed to the following preliminary set of requirements.

3.1.1 Card decks and table arrangements. Prototyping and playtesting usually happen at a table in a mediated setting. Designers author playing cards in a digital drawing tool. However, due to frequent changes to the symbols, this process costs too much time. They require a means to simplify and speed up the following activities:

- **R1** Express card decks, dimensions, symbols and layout.
- **R2** Describe table arrangements with piles, seats and locations.
- **R3** Modify multiple cards at once, procedurally.
- **R4** Store and load card deck designs digitally.
- **R5** Print out cards on paper for cutting cards manually.

3.1.2 Rules and moves. Prototyping revolves around improving the rules to explore gameplay. Designers need a formal notation that permits automating game design with the following activities:

- **R6** Express rules that determine how cards can move cards between piles, including player hands and piles on the table.
- **R7** Place conditions on rules that limit when moves can happen.
- **R8** Express scoring mechanisms and win conditions that attribute value to symbols, actions and results.
- **R9** Express game phases that scope rules for dealing, playing and exchanging cards, passing turns and scoring.
- **R10** Document the rules in tutorials and visual manuals.

3.1.3 Gameplay scenarios. In playtesting sessions, designers analyze if gameplay scenarios, i.e. sequences of player actions, lead to desirable player experiences. Designers need ways to prevent wasting time on poor quality designs. Instead of using spreadsheets for making analyses, they need appropriate visualizations and play-centric tools that support the following activities:

- **R11** Record and replay gameplay scenarios step-by-step with events, player actions and game states.
- **R13** Run specifications as programs, play them alone or together.
- **R14** Explore design alternatives, including what-if scenarios.
- **R15** Identify undesirable scenarios and poorly balanced play quickly, before wasting time on playtesting bad designs.
- **R16** Simulate playtest scenarios in automated playtesting, in order to learn if desirable dynamic sequences happen.
- **R17** Balance and fine-tune competing strategies, deck building.
- **R18** Alternate flexibly between activities and visual perspectives.
- **R19** Identify patterns that help speed up making changes.
- **R20** Apply patterns to introduce quantifiable effects.
- **R21** Obtain immediate feedback, live, with every design change.

Ultimately, designers need tools that seamlessly integrate physical and digital prototyping, e.g., using e-ink cards, augmented- or virtual reality. Next, we illustrate how to analyze problem domains.

3.2 Domain Analysis

We perform a focused domain analysis that compares the structure of shedding and matching card games such as Uno, Go Fish and Old Maid. In particular, we analyze the facets of card decks, table arrangements and moves. The result is a preliminary domain model, a visual dictionary that relates elements that physically exist in the real world to abstract concepts that capture essential properties [13].

This paper uses a simplified design of Old Maid as an illustrative example. Figure 2 shows an example of a game state after dealing. Each turn, a player picks a card whose symbols are hidden from their opponent’s hand. Players discard pairs of cards with matching symbols. The objective is not ending up with the Old Maid (©).

Figure 4 shows a partial domain model consisting of three UML class diagrams that relate the most important concepts we identify. Classes, shown as rectangles, signify named concepts. Associations between them, shown as lines, can be read in both directions,
although names appearing nearby indicate a preferred reading direction (•). Multiplicities appearing at each end indicate how many instances participate: one (omitted), one or more (1..*) or more (*).

First, we relate noteworthy concepts of card decks in Figure 4a. Cards are two-sided rectangular cardboard pieces with rounded edges, usually sized 64 x 89 mm. Cards contain symbols whose meaning is significant to how games are played. Each card has a card type that determines how these symbols appear on its face. Card types can have multiple dimensions of alternative symbols. Together cards can form decks, which can be reused.

Next, we analyze concepts of table arrangements, summarized in Figure 4b. Players occupy a seat at the table. Cards are arranged in piles on the table or in a player’s hand. Each pile has an orientation and an arrangement, which determine a card’s visibility and placement in the pile (abstracted away here). Finally, Figure 4c shows gameplay concepts. A collection of rules determines how each player can move a card from one pile to another.

For conciseness, we have to stop here. This preliminary domain model is sufficient to illustrate the approach. However, it still abstracts away many details. In previous design research iterations, tools have addressed additional facets [5, 24]. For instance, a player usually doubles as dealer in an initial game phase. For now, we also omit the back of cards, currency, scoring, win conditions, game phases, and the constraints rules place on moves. Future iterations can again gradually widen the scope of the analysis.

3.3 Technical Challenges

To realize the requirements of Section 3.1, we formulate technical challenges on developing an engine and tools for CardScript.

3.3.1 CardScript Engine. For creating a reusable engine, the following challenges need to be addressed with a set of components.

T1 CardScript. Textual DSL, exchange, storage format (R4) for expressing a) card decks (R1, R3); b) table arrangements (R2); c) rules (R6, R7); d) scoring (R8); and e) game phases (R9).

T2 Engine. Engine for executing DSL programs consisting of a) abstract syntax; b) an interpreter; and c) formal semantics for analyses with mathematical precision (R1–R21).

T3 Type checker. Analyzer for raising quality, checking context constraints and resolving names (R1–R3, R6–R9).

T4 Event log. DSL, exchange, storage format for expressing, recording and replaying dynamic interaction sequences (R11).

3.3.2 Tool set. We envision an extensible tool set for CardScript.

T5 Authoring tool. An interactive environment that integrates textual/visual notations, syntax highlighting, type checking.

3.4 Card Deck Generator

As an example, we concisely describe the design of a card deck generator. Instead of authoring playing cards by hand, CardScript allows designers to specify the properties of cards that the tool generates automatically. CardScript expresses collections of cards as the n-ary Cartesian product of the symbols appearing on them. We design the necessary components, including a type checker, and explain the textual syntax and the run-time data types.

3.4.1 Syntax. We create a textual syntax for CardScript, illustrated by the Simplified Old Maid example. The design of its deck, shown in Figure 3a, has only one dimension, the suit. The symbols are {♦, ♥, ♦, ♣} and the old maid {Ranked} (line 2). On each card, its symbol appears on the top left, center and bottom right (lines 3–6). The deck consists of generated cards myCards with two cards of each symbol, but only one old maid (line 8–9). Figure 3b shows the arrangement and illustrates a rule that lets players exchange cards (lines 7–9).

3.4.2 Engine. Based on our domain analysis of Section 3.2, we create a metamodel of the abstract syntax in Figure 5. Although the class diagrams of Figures 4a and 5 look very similar, there are several important differences. The classes represent software objects instead of real world ones, and associations are navigable in one direction only. Two added classes, Decl and Mult, represent the name and type of the card set, and how often symbols of its card type appear in the Cartesian product of the set.

3.4.3 Implementation. We implement the tool prototype using Rascal [12]. The tool counts only 430 source lines of code, and its sources are available on GitHub [26]. Generated PDFs of Old Maid and a classic French Deck appear in supplementary material.
4 DISCUSSION

We have proposed a structured approach towards a unified game design language that arises from collaboration with industry partners. Our preliminary results have not yet been thoroughly validated. However, we have demonstrated the approach is feasible, and our progress already includes useful insights, analyses and artifacts.

4.1 Challenges and Opportunities

Requirements for game design tools have also been described in related work [1, 17]. In order to identify common goals and needs for a unified game design language, we discuss additional perspectives that relate open challenges to opportunities for future research.

4.1.1 Automated playtesting. Analyzing the qualities of dynamic interaction sequences is necessary for automated playtesting, balancing, and fine-tuning. Many algorithms have been created and adapted for playing card games, e.g., Poker [4], Hanabi [7] and HearthStone [11]. The challenge is applying algorithms to assess the qualities of other games too, e.g., as done in Recycled [2]. CardScript serves to study general, compatible and reusable solutions.

4.1.2 General Gameplaying. Conversely, games have been studied as algorithmic challenges, e.g., in general game playing. Game Description Languages (GDLs) describe game-based challenges as a testbed for AI, e.g., Ludii [18], the Video Game Description Language (VGDL) [20], and the Card Game Description language (CGDL) [9]. Competitions serve to compare how well algorithms perform [15]. CardScript’s goals coincide with even more demanding challenges and real-world conditions. Organizers can further raise the difficulty by dynamically changing the rules during play.

4.1.3 Education. Digital games have been used to teach programming, computer science and game design. Educators use mods as examples that can be studied, adjusted and shared, e.g., with Scratch [23]. Card games are especially well suited as a subject for teaching young children [16]. CardScript can serve as an educational instrument. Coinciding challenges include creating coding environments for studying game design, documenting designs, creating manuals, and learning how to code.

4.1.4 Bodies of knowledge. Researchers have created catalogues describing ontologies and gameplay design patterns [3] to perform critical analyses and accumulate game design lore. However, bodies of knowledge usually omit working examples of source code that exemplify how to create mechanisms and how to reproduce dynamic interaction sequences. CardScript can serve this purpose. By adding working examples, researchers can construct annotated source code repositories. These repositories support empirical studies, e.g., for mining patterns and identifying best practices.

5 CONCLUSION

In this paper, we have made a case for a unified language for game design. We have proposed an approach that leverages playing cards, and described how CardScript and a reusable toolkit can be created. We invite feedback and discussion on its requirement and our progress in order to identify common objectives and next steps.

ACKNOWLEDGMENTS

We thank the DGA and CLICKNL for funding this work. We thank CodeGlue for their continued collaboration, and the workshop participants for their input and feedback. We thank the reviewers, Paul Klint and Daria Polak for their feedback on this paper.

REFERENCES