Mek: Mechanics Prototyping Tool for 2D Tile-Based Turn-Based Deterministic Games

Rokas Volkovas
QMUL
London, UK
r.volkovas@qmul.ac.uk

Michael Fairbank
University of Essex
Colchester, UK
m.fairbank@essex.ac.uk

John R. Woodward
QMUL
London, UK
j.woodward@qmul.ac.uk

Simon Lucas
QMUL
London, UK
simon.lucas@qmul.ac.uk

***Abstract*—There are few digital tools to help designers create game mechanics. A general language to express game mechanics is necessary for rapid game design iteration. The first iteration of a mechanics-focused language, together with its interfacing tool, are introduced in this paper. The language is restricted to two-dimensional, turn-based, tile-based, deterministic, complete-information games. The tool is compared to the existing alternatives for game mechanics prototyping and shown to be capable of succinctly implementing a range of well-known game mechanics.**

I. Introduction

Existing research in tool-assisted-game-design is primarily dedicated to the development of game generation algorithms. Frameworks such as GVGAI [1] are determined to provide a groundwork for the development of general AI or complete game generation automation. As such, the languages used for the game specification are flexible enough to describe games with significantly different player goals, but too rigid to be used for describing new interactions.

The most flexible tool for a digital game designer remains a general purpose programming language, which requires a good knowledge of programming and is a significant barrier of entry. This is not the case with other disciplines in game development, such as visual art or sound design. To begin improving the situation towards one where game designers have an environment – equivalent to the environment of what `Photoshop` is to visual artists – a language, abstracting the common tedious details of digital games is introduced here.

`MekLang`, the introduced language, sacrifices some description generality for its uniformity through enforcing the mechanics to fall into the two-dimensional (2D), turn-based, tile-based, complete-information categories. In contrast to other existing attempts, the language emphasizes development of mechanics, rather than games.

A. Games vs Mechanics

For a more precise discussion of the details and the importance of mechanics, it is necessary to draw a clear line between what is meant by `games` and `mechanics` within this document. A mechanic is a state transition definition. In other words, a mechanic describes how the game is allowed to change over time or with every move available. A game, on the other hand, is a set of mechanics with an end state – a state where no moves can be made.

978-1-7281-1884-0/19/$31.00 ©2019 IEEE

Consider an infinite `Chess` game board. The way the bishop moves is a mechanic. Specifically, the ability to move the piece anywhere on the board, as long as it is within a straight diagonal line from the current position, is the mechanic. Similarly, adding a knight on the same infinite board, increases the number of mechanics to three: movement of the bishop, movement of the knight and their interaction with each other. That is, a combination of mechanics is also a mechanic. The limitation of the board to an `8x8` grid introduces the mechanic of pieces not being able to move off the board. Mechanics by themselves are not games. Adding an end state – which is also a mechanic – is what makes the complete set of mechanics a game.

B. Analysis Implications

The explicit separation of the game and its mechanics is important for both design and analysis purposes. In Chess, a player wins the game by check-mating the opponent's King (making it unable to escape a check). Now consider a Chess variant where the player wins by getting her own King check-mated. These game versions can be argued to be significantly different in terms of how the players will approach playing them. Hypothetically, assuming there is a perfect artificial measure of fun, one game might be found to be fun and the other as far away from fun as possible. The only mechanic that was changed was the end condition. In this situation, no practical information is gained about the inner workings of the game, other than its overall quality. In other words, the game designer learns nothing about the mechanics, the building blocks of the game.

Here, we argue, that mechanics can and should be separated from games and analyzed as distinct units of game design. Much like in art, a teacher can tell whether or where the basic shapes a student is drawing are well presented, one should be able to tell whether a mechanic, in isolation, is fit for the purpose it is being implemented for. Mechanics build up games like stepping stones, and they alone can give solid indication on how the game will function. Structured mechanic analysis will provide a groundwork for experimentation through rapid prototyping and is most sensible when the mechanics are defined in a mechanics-focused language. Section IV describes the first incarnation of such a language, created for the purpose of prototyping mechanics.

Tool	Language	Players	Game Creation	Goal	Restrictions
Mek	Visual	N/A	No	Game Mechanics Prototyping	DD
Machinations	Visual	N/A	No	Game Systems Design	Abstract
Ludi	Text	2	Automated	Automate Board Game Design	2P, DD
ANGELINA	Text	1	Automated	Automate Game Creation	Depends on version
GVGAI	Text	1-2	Manual & Auto	Testbed for General AI	2D Sprite-based

Table I: Game Design Tool Comparison

II. Existing Work

This section explores the inner-workings of the alternative tools, identifying their strengths and weaknesses for mechanics prototyping. Table I compares the features of `Mek` with those the most prominent existing game design tools. Notes:

- DD = Discrete, Deterministic
- Discrete = turn-based, tile-based
- All languages are custom (not build directly on another)
- *Players* is the number of players supported in games
- *ANGELINA* creates different genre games per version
- *Machinations* helps designing conceptual systems

MekLang is conceptually most similar to the *Machinations* tool in terms of its purpose. The other tools listed use game design as a fertile ground for AI experimentation (*Ludi*, *ANGELINA* and *GVGAI*). There exist other tools that can be use for game design by proxy, such as *PuzzleScript*, *RPGMaker* or any other engine geared towards creating games. However, these tools are ignored as not being focused on advancing game design explicitly.

A. Machinations

Joris Dormans' *Machinations* [2][3] tool can easily be considered the most formalized design tool implemented to date. It has the look of UML diagrams. The tool focuses on the ability to express what is called game *e conomies*, which are the transfer of game numerical values (resources) from one container to another over the course of the game. This focus bias allows the designer to quickly express high-level resource interactions common in most traditional board games with the ability to define value transitions using common mathematical operations.

The language expressiveness power, however, comes at the cost of abstracting game details to sometimes unrecognizable game implementations – not only does it separate the level design from the mechanics, but it removes level design entirely. The removal of level design allows game designers familiar with the system to discuss concepts quickly, however the design of even a relatively basic resource-focused game like *Monopoly* depends heavily on the "level" (board) representation. The lack of level design expressiveness also limits the ability of automation, both creating new mechanics and analyzing the existing ones, not knowing how they would interact in the full game. On the other hand, adding such capabilities seems to be within reasonable reach of the language, but without the source code availability or author's support this is uncertain.

B. Ludi

Ludi [4] is a framework devised specifically for the game mechanics delivery automation (the polar opposite of manual design). It focuses on finding *fun* game rule-sets using an evolutionary-algorithm based system. *Ludi* is a good example of the existing interest of analyzing mechanics. However, while it uses a language that could potentially be used by game designers to explore their designs, its definition is geared towards the automated design iteration abilities. The system was shown to even be capable of producing a game that was released commercially [5].

The restrictions it imposes onto the created games are that they have to be turn-based, grid-based, deterministic, two-player and perfect information. The most significant of these is the two-player requirement. It restricts designers when considered in comparison to other mechanics or general languages. The rules are expressed using a text-based scripting language, which features the ability to define a set of distinct board types: traditional 8-neighbour square grid, hexagons, triangles and some less familiar ones. The focus of the language towards design automation is apparent when reading the described game definitions. The definitions are very flexible in how they can be written out, but may be hard to parse for someone not familiar with the language features in-depth. Nonetheless, the system shows what features lend themselves more easily to AI analysis and design metrics.

C. ANGELINA

ANGELINA is an example of game design being approached with another distinct set of motivations. It is a system created to produce complete games autonomously, using procedural generation creatively (e.g. looking up images based on text) and has gone through a number of iterations, with the latest one [6] making 1990s dungeon crawling RPG-like 3D games.

In stark contrast to the systems described above, it attempts to entirely remove the designer from game development. While at first glance, this would rule it out from being valid consideration or a comparison to tools aimed to aid game designers, it is important to recognize that as the field becomes more and more automated, the approaches made towards automation help gain insight into what features to include or avoid in order to make even designer-centric systems more expandable. From this point of view, it is useful to note how *ANGELINA* handles game resource management, both using pre-designed pieces for levels and objects combined with evolution. The system does not directly expand on any of the specific algorithms but tackles the issue of gluing them all together in a cohesive manner.

D. GVGAI

The General Video Game Artificial Intelligence (*GVGAI*) framework [1] is a system created with the goal of encouraging AI researchers to focus their attention on general problem solving methods in order to allow them to be more readily transferable to new domains. Through defining a common interface to a large number of distinct games, the framework facilitates exploration of a number of game development areas, including: general AI agents playing unseen games [7], generating levels [8] and, most relevantly, generating game rules [9].

The rule generation in its case is treated as a problem to be solved, given a specific level. The generation is done on the game description language used in *GVGAI*, which is evolved from [10]. The language is succinct in terms of what it aims to represent, which is relatively complex games, but this feature comes at the cost of flexibility, as the rules specified rely on engine-available features. This approach does not align well with manual design of new mechanics. The framework is also directed more towards generality, which allows for creative research, such as mechanics recommender systems [11], but might deter in-depth focus for better or worse.

E. Design Considerations

The tools outlined in the previous paragraphs were found to be the most prominent examples of the state of the art tools available to game designers comparable to Mek. This is not to indicate that no other work has been done towards the same goal, only to showcase the work that's practically usable. A number of primarily philosophical considerations have also been discussed, including [12], which outlines a method of potentially more useful prototype iteration in their developed engine [13], encouraging both human and machine play-testing. Or [14], which describes a system to express game ideas and concepts instead of concrete designs.

Any type of game-specific design tools, such as [15] [16], or even use of machine learning to define and combine mechanics from existing games [17] [18] is excluded from consideration as being either not directly usable or too specific to be useful for game designers. Also not covered was material such as the Stanford GDL [19], which could be considered a precursor to the language used in GVGAI framework.

III. Language Requirements and Restrictions

Prior to describing the details of the language, it is important to understand the requirements outlined for it and their significance. Intuitively, every digital game can be described as a Markov Process, with every distinct frame of a game being a different state. However, this naive implementation yields impractically vast numbers of states and their transitions.

A. Scalability

While Markov Process representation may be practically applied to anything with few states, making changes is a tedious process and the implementation does not scale to larger variations of the same game/mechanic. Consider implementing an adjacently moving piece in `2x2` sized grid-world. This results in `4` possible states (indicating unit position) with `8` connections between the states (`2` adjacent connections from each tile). The setup grants complete flexibility over how the connections between the states are connected.

Now consider expanding the grid-world size to `3x3`. The movement mechanic is identical, but the expansion requires adding `5` extra states, with `20` connections. The representation complexity (number of needed states and connections) increases quadratically with the grid size. This is undesirable. Thus the language should be capable of scaling mechanics to any board size.

B. Discrete 2D and Information Completeness

With the conceptual goal in place, the domain of game mechanics needs to be defined. The language is chosen to specialize in two-dimensional, turn-based, tile-based, complete-information games. These categories were chosen due to the ease of information presentation, computational efficiency and large range of distinct existing games falling under these categories. That is, these restrictions strike a favourable balance between the implementation complexity and mechanic definition flexibility.

The following games – and more importantly, their mechanics – fall under the defined restrictions: *Chess*, *Snake*, *Othello*, *Tetris*, match-3 games (*Bejeweled*, *Candy Crush* etc.), *Sudoku*, *Sokoban* and many others. These games are all mechanically distinct – they have mechanics which are naturally associated with those specific games, yet they all share the common features of having a limited size two-dimensional grid if square tiles, are played in turns and present all gameplay information to the player at all times.

C. Mechanics Focus

The purpose of the developed language is to make the design of mechanics more accessible. This core principle establishes that the language is *not* for making games, but creating the mechanics. These mechanics can then be taken outside of the language confines to make full-games.

The tools to support game development are not the same ones that support mechanics development. This may be difficult to grasp, since no widespread tools support mechanics development exist. Games need: mechanics, art, sound, general programming etc. To help the understanding, mechanics are to be looked at as what 3D models are to games – individual pieces of art which can be presented by themselves and need some small (relative to the model creation) amount of work to be utilized in a game. If a model looks good in modelling software, then it will likely look good in game. This is not guaranteed though, due to the possibility of game-sensitive issues, such as clashing art-styles with other models. The same holds for game mechanics. Yet, there are no tools anywhere near as expressive as the state of the art modelling software. The ultimate goal here is to build one. The tool is a small step towards this idealistic goal.

IV. Mek

Mek is the proposed visual mechanics prototyping tool. *MekLang* is the language Mek uses to implement mechanics. Understanding MekLang is easier through first understanding Mek. Figure 1 shows how the complete interface of Mek looks like. The interface is a combination of a number of individual parts. The inner workings of all the interface parts and their interactions are described in this section.

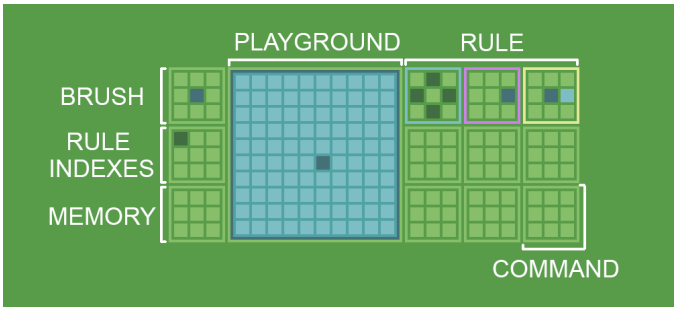

Figure 1: Complete Mek interface

A. Playground

The playground is a `10x10` grid of tiles. In Figure 1, it is seen as the large grid center-left of the image. This grid is used to represent the current board state, upon which the designer-implemented mechanics operate. A playground tile can be one of the `9` colors shown in Figure 2. The colors are labeled with their respective color indexes.

Figure 2: Available playground tile colors with their indexes.

B. Command

A command is a `3x3` grid of tiles. Figure 3 shows how one appears in Mek. This grid is used to define an operation to be done on the playground. To change the command behaviour, each one of the tiles can be clicked using the mouse. When clicked, the command tile changes color to the next available one, cycling back when the last color is reached. The number of available colors differs depending on the type of a command, but by default command tiles all look like ones in Figure 3. The type of a command is indicated visually by the color of the outline of the grid. Table II shows all the existing primary command types and their associated colors. The behaviour associated with each command and their variations are specified in Section V.

Figure 3: Command: a `3x3` grid of tiles.

C. Command cycling

The color cycling operation happening on command tile click is equivalent to the following modulo operation:

$$ColorIndex_n = (ColorIndex + 1)\%|AvailableColors|$$

Figure 4 shows the sequence of enumerated single command states. Specifically, the command labeled `0` is the default state of the `WRITE`-type command (the type is indicated by the yellow border color). Once the top-left tile is left-clicked, the command transforms into the one labeled `1`. Left-clicking the top-left tile again transforms the command into the one labeled `2`. When the command state labeled `9` is reached, the next left-click on the same tile leads back to the grid labeled `1`. If the tile is right-clicked, the states in the image are traversed in the opposite direction (`2` to `1`, `1`, to `9`, `9` to `8` etc.). Middle-clicking the tile, resets it back to the color seen in state `0`. This describes the color cycling for the command of type `WRITE`. Other commands might have less colors available. Unavailable colors are skipped.

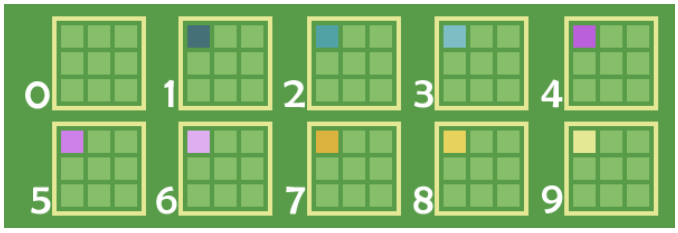

Figure 4: Command color cycling. From `0` to `9`: command grid transformation after each top left tile click.

D. Command Execution

A command can be executed. Executing a command means performing an action on the board (playground + memory, introduced later) state. This action may or may not transform the board state. The nature of the action depends on the type of the command. For example, execution of a `WRITE` command, would cause certain tiles on the playground be overwritten with colors of the ones contained within the command, which is showcased in Figure 5. The `WRITE` command with (the `5x5` version of) the playground is shown the left. When the tile highlighted in white is clicked, the playground becomes the one shown on the right. The execution of the command meant copying the non-light green contents of the command grid to the playground, relative to the clicked tile location.

Command Type	Outline Color	Tile Color Set
`WRITE`	Yellow	All, except dark green
`CHECK`	Light Purple	All
`SHIFT`	Dark Green	Dark green only
`ROTATE`	Light Blue	All
`CYCLE`	Orange	All, except dark green
`CALL`	Dark Blue	Dark green only

Table II: List of all the primary command types and their associated outline colors and available tile color sets.

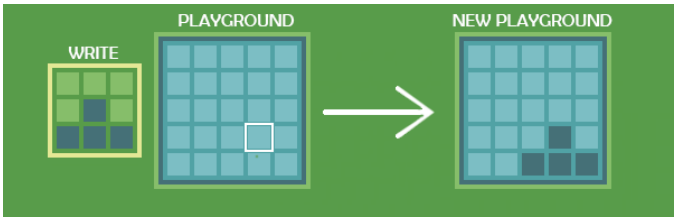

Figure 5: `WRITE` command execution: when the tile highlighted in white is clicked, the left playground transforms into the one on the right.

E. Rule

A rule is a list of commands. The order in which the commands interact with the board state depends on its rule position. Figure 6 shows how a `3`-command rule would appear in Mek. In the figure, there are three commands each with a different type (indicated by the different outline colors). The rule can be executed. Executing a rule means executing its commands in order, one by one. Some commands may terminate the rule execution. In this case the remaining commands are not executed.

Figure 6: Rule: a list of commands. The command outlines indicate their types: 1. `CHECK` 2. `WRITE` 3. `CALL`

In Mek, rules are `9` commands long. Figure 7 shows how a rule looks like in Mek. By default, all rule commands are empty (light green command outline). The number labels indicate the order in which they are executed. In the figure, the commands `5` to `9` are empty. Empty rules are skipped when a rule is executed. Listing 1 shows the pseudo-code equivalent of the rule logic shown in Figure 7.

```
func execute_rule(focus):
    #ROTATE
    for rotation in [0, 2, 4, 6]:
        focus.save()
        focus.rotate(rotation)
        #SHIFT
        focus.shift(RIGHT)
        #CHECK
        var tile = focus.tiles[CENTER]
        if tile.color = DARK_BLUE:
            #WRITE
            focus.tiles[CENTER].color = L_BLUE
            focus.tiles[LEFT].color = D_BLUE
        focus.reset()
```

Listing 1: Example rule execution pseudo-code. Lines starting with # indicate the beginning specific command type functionality.

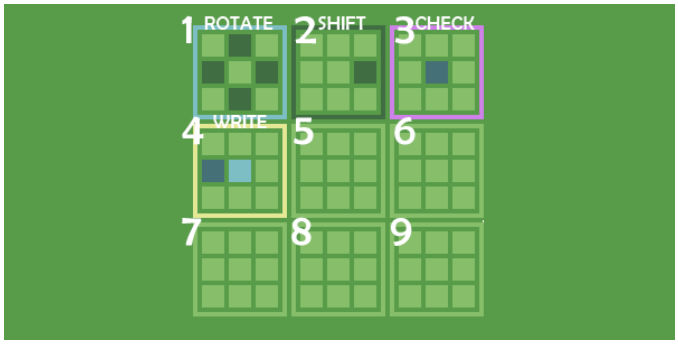

Figure 7: A rule in Mek: a list of `9` commands.

The pseudo-code makes use of the focus concept, explained in Section IV-G. In the pseudo-code, the focus is an object, containing a reference to the tiles of the playground the commands are being executed upon.

F. MekLang Process

MekLang is the language Mek uses to implement the mechanics. The mechanics are built out of rules and commands. The mechanics operate on the playground. At its simplest form, MekLang in Mek works like this:

1) A tile on the playground is clicked
2) The available rules are executed
3) The process repeats

G. Focus

The `3x3` grid of playground tiles each command operates upon are said to have the *Focus* on them. The focus has two parameters: position and rotation. By default, before command execution, the focus center location is set to match the location of the clicked tile. The focus position, as well as the rotation, can both be changed by specific commands, which are described later. Moving the focus affects which tiles are seen by the executed commands, whereas rotating the focus affects how they are seen.

H. Rule Index Grid

One rule may not be enough for more complex mechanics. To combat this, Mek has `9` rules available. However, there is not enough screen space to show them all at the same time. To view and modify the different rules, the rule index grid is used. The rule index grid is shown in Figure 8. The shown tiles are indexed by the rule numbers they lead to. Clicking the tile changes the visual rule commands to the ones of the selected rule. This is only a visual change for ease of interaction.

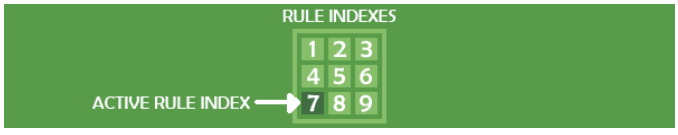

Figure 8: Rule index grid: the dark green colored tile indicates the currently visible rule index.

I. Tile Toggling Example

A small step above in terms of complexity in comparison to the tile coloring example is the tile toggling mechanic:

- when a tile is clicked, toggle its color
  - toggling a light blue tile turns it dark blue
  - toggling a dark blue tile turns it light blue

The simplest way to implement the mechanic is to use two rules, one for each toggle scenario. Figure 9 shows how the first three commands of the two rules would look like. When a tile is clicked, both rules are executed in order.

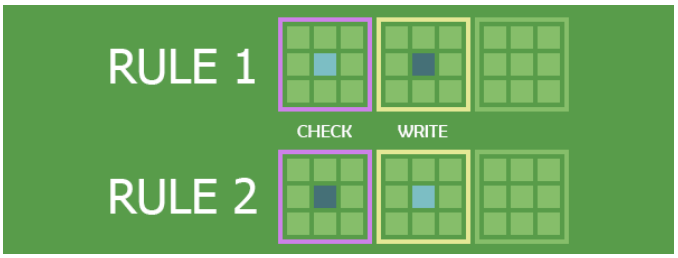

Figure 9: Toggle mechanic rules: `CHECK` commands followed by `WRITE` commands of opposite color.

The figure utilizes the already used but yet to be defined command, which is the `CHECK` command (purple command border). This command compares the values of the tiles focused on with the ones in the command. If the tiles mismatch, the rule being executed terminates. These rules implement the toggle mechanic.

J. Implementation of Existing Games

The examples presented above are a useful when describing MekLang, but are still not representative of the language's flexibility. The language was used to implement a number of well known games (or at least their core mechanics). Their representation complexity, in terms of the number of rules and commands used within the implementation, in MekLang is shown in Table III.

Note, that since MekLang is capable of implementing *Game of Life*, it is Turing Complete, which implies that given endless memory it is capable of implementing all computer programs. However, this feature does not mean that doing so is simple enough in comparison to the alternatives. On the other hand, the number of commands and rules used to implement the different games arguably should give a solid indication of the language capabilities.

Game	Rules	Commands
Sliding Puzzle	1	3
Nim (Subtraction)	2	8
Solitaire (Marbles)	3	9
Sokoban (Pushing)	3	9
Game of Life	4	16
Noughts and Crosses	5	19
Othello / Reversi	5	21
Connect 4	6	26

Table III: Known game implementation complexity

V. MekLang

To make the implementation of all the games listed in Table III possible a number of specific command types were used. All the different commands, along with the additional behaviours of MekLang are covered in this section.

A. Memory

Memory is an additional `3x3` tile grid that the commands can operate on. It has all the same features that the playground does, except its size. Memory and playground together form the entire board state. The memory grid is seen in the bottom-left corner of Figure 1.

B. Modes of Operation

There are two modes of operation in Mek. The modes are:

1) `BRUSH`
2) `NORMAL`

The modes affect what rules are executed when a tile is clicked. In `BRUSH` mode, only one rule, with a single command is executed. The command type is set to `WRITE` and cannot be changed. The purpose of this mode is to be able to quickly change the board state without executing the implemented mechanics. The implemented mechanics are executed in the `NORMAL` mode. The modes can be switched between at any point a tile click is expected.

C. Execution Scheduling

When executed, some commands may complete their action immediately, or schedule it to be done after all the rules are executed. Immediately performed actions affect what the subsequent commands interact with; delayed actions do not. If multiple commands schedule overlapping actions, they are applied in the scheduling order.

D. `WRITE`

The simplest command is of the `WRITE` type. When a `WRITE` command is executed, the tiles at the focus location are overwritten with the contents of the command grid. Light green command grid tiles are ignored, thus keeping the original value of the target grid. It has a number of variations[1]. Including the default, the variations are:

- `WRITE`
- `WRITE TO MEMORY`
- `WRITE FROM MEMORY`
- `WRITE FROM PLAYGROUND`
- `WRITE INSTANT`
- `WRITE TO MEMORY INSTANT`

The variations specify the colors that are to be used for writing as well as the target grid along with whether or not the writing should be delayed to the end of the turn. In the case of the `WRITE`, `WRITE TO MEMORY`, `WRITE INSTANT` and `WRITE TO MEMORY INSTANT`, the colors to be written are the ones specified in the command grid.

[1]Command variations have distinct shades of the base command color. They can be thought of as functions with different parameters.

E. CHECK

The CHECK command compares the color contents of the target grid with the source ones and if they do not match, terminates the current rule, does nothing otherwise. Light green source tiles are skipped and as such, if all the source tiles are light green, the command is equivalent to an `if true` instruction. These are the command variations:

- CHECK
- CHECK NOT
- CHECK MEMORY
- CHECK MEMORY NOT
- CHECK WITH MEMORY
- CHECK WITH MEMORY NOT

The source tiles are the command tiles if the command type is CHECK, CHECK NOT, CHECK MEMORY or CHECK MEMORY NOT. Otherwise, the source tiles are the memory tiles. The NOT variations compare the source and target tiles and terminate the current rule if they match. When the source tiles are not the command tiles themselves, the command tile colors indicate the referenced tile index.

F. Color Enumeration Mapping

Specifically, the colors in Figure 2 are transformed into numbers from 1 to 9 and mapped onto the source tile enumerations, which are equivalent to the positional command enumeration in Figure 7.

G. SHIFT

The SHIFT command translates the focus location in all, within the command specified, directions. Specifically, the operation of this command is as follows: if a tile is colored dark green (the only alternative color for this command), the subsequent commands are executed with the focus shifted by one tile in the direction of the coloured neighbour relative to the center tile.

Consider Figure 10. On the left, the SHIFT command (dark green outline) is shown with the bottom-right tile in dark green. Next to it is the playground. The white 3x3 rectangle outline represents the focus location before the command execution. The playground on the right shows the focus location after the command execution.

If multiple tiles are dark green, the execution of the subsequent commands repeats for each direction, resetting the pre-shift focus location before every shift. If the center tile is colored or no tiles are colored, the following commands are executed without shifting.

H. ROTATE

Similarly to the SHIFT command, the ROTATE command executes the subsequent commands with the focus rotated. Specifically, dark green command grid tiles indicate the amount by which the subsequent commands should be rotated clock-wise around the center tile. Coloring the top center tile applies no rotation, the top-right one has the commands rotate by one tile clock-wise (45 degrees), the right tile has them rotate by two tiles and so on. If the center tile is dark green colored or no tiles are colored, the subsequent commands are executed without rotation.

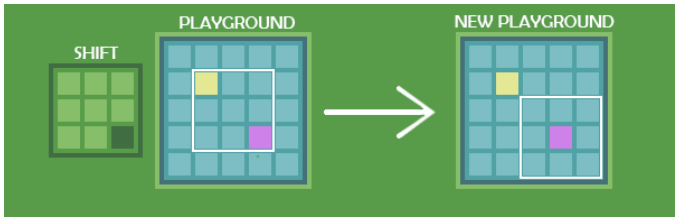

Figure 10: SHIFT command: the focus moves from the center of the playground, one step in the south-east direction, because of the dark green tile location in the SHIFT command grid.

Consider Figure 11. Here, a three-command rule on the left starts with a ROTATE command (light blue command border), followed by a SHIFT and WRITE. When the rule is executed, the commands executed are those seen on the right – for each dark green colored tile, a row of rotated subsequent commands is generated. This forking also happens when multiple tiles in the SHIFT command are colored dark green.

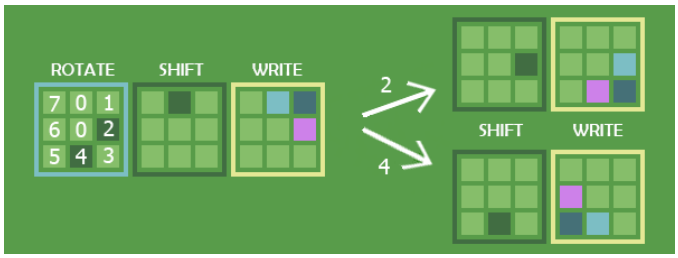

Figure 11: ROTATE command: generates multiple rules from the following commands, which are all rotated clock-wise based on the dark green command tile index.

I. CALL

The CALL command is used to indicate when another rule should be called. Executing a CALL command, executes the rule indicated by the dark green tile index. Upon completion of the rule, it returns to continue with the commands following the CALL command, using the same indication as the rule index grid (Figure 8). Only one dark green colored tile is allowed to be used in the command. For example, the CALL command in Figure 6 would call rule 5.

J. CYCLE

The CYCLE command cycles the target tiles the same way a click does on a command tile. The tiles are cycled by the amount equal to the index of the command color (see Section V-F). For example, if the center command tile is colored light blue, the center of the focus tiles will be cycled by 2 colors (the light blue color index is 2). These are the command variations:

- CYCLE
- CYCLE INSTANT
- CYCLE MEMORY

- CYCLE MEMORY INSTANT

VI. Limitations

A. MekLang

Even though the introduced language is flexible enough to implement a range of distinct recognizable mechanics and games, the flexibility is nowhere near that of a general purpose language. That is, the cost of making game mechanics prototyping more accessible is the loss of access to established tools and libraries. Furthermore, the poor performance of the language is also a considerable drawback. Depending on the complexity of the logic implemented, the logic may take a while to compute. A while here is considered up to a second, making rapid changes to the board state unobtainable.

B. Interface

Figure 1 shows the interface of Mek, with all the described parts labeled. This initial implementation limits the number of rules and commands available to the user, primarily due to the visual representation restrictions. This inhibits and biases the designer creativity most significantly. A good example for this is the inability to implement the ko-rule of *Go*, which requires remembering all the distinct game states that have already occurred. This example could easily be implemented given multiple playgrounds and the ability to select the targeted one.

C. Game Making

Moreover, making games in this initial implementation is considerably more difficult compared to making individual mechanics. Unlike the previous limitations, this one appears by design – the language is focused on allowing rapid mechanic iteration. The only reason games (set of mechanics with an end condition) can be made in it is because end conditions are mechanics too. The impact of the described limitations can also still be reduced, which is the primary goal of the future work.

VII. Conclusion and Further Work

In this document, a language for prototyping mechanics for two-dimensional, turn-based, tile-based, complete-information games was introduced. The language was described in detail, shown to be capable of implementing a range of distinct mechanics (and games) and the limitations of the implemented system were outlined.

With the ability to succinctly describe a range of mechanics, the opportunities to more easily explore them open up. Specifically, the future work is laid out to be the the implementation of additional features, which allow automating the design of the mechanics and comparing the mechanics statistically in parallel with reducing the friction of using the language for its main purpose – prototyping mechanics.

The additional features are expected to be: the removal of the grid size limitations (both of the playground and the commands), support for API-level access through a webserver, ability to setup multiple playgrounds as well as the support for custom images instead of preset range of colors.

VIII. Demo

A working demonstration with all the mentioned implemented mechanic setups can be found on the author's website[2]. The tool shown there is a demonstration of the described concepts and is not yet intended for professional use.

Acknowledgment

This work was funded by the EPSRC CDT in Intelligent Games and Game Intelligence (IGGI) EP/L015846/1.

References

[1] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Simon M Lucas, and Tom Schaul. General video game ai: Competition, challenges and opportunities. In *Thirtieth AAAI Conference on Artificial Intelligence*, pages 4335–4337, 2016.

[2] Joris Dormans. Machinations: Elemental feedback structures for game design. In *Proceedings of the GAMEON-NA Conference*, pages 33–40, 2009.

[3] The designer's notebook: Machinations, a new way to design game mechanics.

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

[5] Cameron Browne. Yavalath. In *Evolutionary Game Design*, pages 75–85. Springer, 2011.

[6] Michael Cook and Simon Colton. Ludus ex machina: Building a 3d game designer that competes alongside humans. In *ICCC*, pages 54–62, 2014.

[7] Cristina Guerrero-Romero, Annie Louis, and Diego Perez-Liebana. Beyond playing to win: Diversifying heuristics for gvgai. In *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*, pages 118–125. IEEE, 2017.

[8] Xenija Neufeld, Sanaz Mostaghim, and Diego Perez-Liebana. Procedural level generation with answer set programming for general video game playing. In *Computer Science and Electronic Engineering Conference (CEEC), 2015 7th*, pages 207–212. IEEE, 2015.

[9] Ahmed Khalifa, Michael Cerny Green, Diego Perez-Liebana, and Julian Togelius. General video game rule generation. In *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*, pages 170–177. IEEE, 2017.

[10] Marc Ebner, John Levine, Simon M Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. Towards a video game description language. 2013.

[11] Tiago Machado, Ivan Bravi, Zhu Wang, Andy Nealen, and Julian Togelius. Shopping for game mechanics, 2016.

[12] Adam M Smith, Mark J Nelson, and Michael Mateas. Computational support for play testing game sketches. In *AIIDE*, 2009.

[13] Adam M Smith, Mark J Nelson, and Michael Mateas. Prototyping games with biped. In *AIIDE*, 2009.

[14] Mike Treanor, Bryan Blackford, Michael Mateas, and Ian Bogost. Game-o-matic: Generating videogames that represent ideas. In *Proceedings of the The third workshop on Procedural Content Generation in Games*, page 11. ACM, 2012.

[15] Matthew Guzdial and Mark Riedl. Learning to blend computer game levels. *arXiv preprint arXiv:1603.02738*, 2016.

[16] 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. ACM, 2016.

[17] Gabriella AB Barros, Michael Cerny Green, Antonios Liapis, and Julian Togelius. Data-driven design: A case for maximalist game design. *arXiv preprint arXiv:1805.12475*, 2018.

[18] Matthew Guzdial and Mark Riedl. Automated game design via conceptual expansion. *arXiv preprint arXiv:1809.02232*, 2018.

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

[2] https://rokasv.com/mek-paper-1/