A Pattern-Based Game Mechanics Design Assistant

Riemer van Rozen[*]
Amsterdam University of Applied Sciences
Duivendrechtsekade 36-38 1096 AH
Amsterdam, The Netherlands
r.a.van.rozen@hva.nl

ABSTRACT

Video game designers iteratively improve player experience by play testing game software and adjusting its design. Deciding how to improve gameplay is difficult and time-consuming because designers lack an effective means for exploring decision alternatives and modifying a game's mechanics. We aim to improve designer productivity and game quality by providing tools that speed-up the game design process. In particular, we wish to learn how patterns encoding common game design knowledge can help to improve design tools. Micro-Machinations (MM) is a language and software library that enables game designers to modify a game's mechanics at run-time. We propose a pattern-based approach for leveraging high-level design knowledge and facilitating the game design process with a *game design assistant*. We present the Mechanics Pattern Language (MPL) for encoding common MM structures and design intent, and a Mechanics Design Assistant (MeDeA) for analyzing, explaining and understanding existing mechanics, and generating, filtering, exploring and applying design alternatives for modifying mechanics. We implement MPL and MeDeA using the meta-programming language RASCAL, and evaluate them by modifying the mechanics of a prototype of Johnny Jetstream, a 2D shooter developed at IC3D Media.

1. INTRODUCTION

Video game designers work to improve player experience by iteratively play testing game software and adjusting the game's design. Improving gameplay is difficult and time-consuming because designers lack an effective means for quickly exploring design alternatives and modifying a game's mechanics. Specifically, designers cannot efficiently 1) author, modify and fine-tune mechanics—rules of games that influence gameplay—without help of programmers; 2) receive immediate feedback for comparing design intent to change result; and 3) explore decision alternatives efficiently.

[*]This work is part of the Automated Game Design Project, and this research is carried out at the SWAT group of CWI, Amsterdam, NL.

Since making well-informed design decisions is costly due to long iteration times, play testing is limited to fewer software versions than necessary for achieving the best game quality. We aim to improve designer productivity and game quality by providing tools that speed-up the game design process. In particular, we wish to learn how patterns encoding common game design knowledge can help to improve such tools.

Micro-Machinations (MM) is a visual language and software library that facilitates brief design iterations by enabling game designers to modify a game's mechanics while it is running, and to play test simultaneously [10, 21]. MM programs can be represented as visual diagrams that are directed graphs describing game-economic mechanics using various kinds of nodes and edges. A diagram works inside a game, steering its mechanics, and when set in motion through run-time and player interaction, the nodes redistribute resources along the edges between the nodes. Understanding the dynamics of diagrams is hard because designers combine elements in non-trivial ways, expressing design intent by providing choices, challenges, trade-offs and strategies for interesting gameplay. Adams and Dormans have suggested patterns for MM's evolutionary predecessor Machinations, as a mental framework for understanding, explaining and designing game mechanics, and they provide a mechanics design rationale with example diagrams [1].

We propose a pattern-based approach that assists designers in the discipline of modeling mechanics for modifying game software. We define parameterized micro-mechanism patterns (*patterns* for short) that capture a wide range of diagrams with shared structures and design intent. We provide a Mechanics Pattern Language (MPL) for programming patterns and a Game Mechanics Design Assistant (MeDeA) that recognizes patterns in existing diagrams, explains intent of design choices in understandable text and enables interactively and visually exploring design choices that can be step-by-step filtered and applied yielding new diagrams and software versions. We contribute the following:

- A Mechanics Pattern Language (MPL) for programming patterns that capture a wide range of diagrams with shared structures and design intent.
- A Mechanics Design Assistant (MeDeA) implemented in the RASCAL meta-programming language for analyzing, explaining and understanding existing mechanics and generating, filtering, exploring and applying design alternatives for modifying mechanics.

scope/concern	**language**	**tool/system**	**goals/functionality**
"2D games" (research platform)	PyVGDL [15]	PyVGDL lib.	Design games and analyze dynamics and learning algorithms
board games	GDL / Ludemes [3]	Ludi system	Playing, measuring, and synthesizing board games
mechanics of "board-games"	Prolog subset [16]	BIPED	Prototype complete board-like games and analyze their dynamics
mechanics (avatar-centric)	PDDL subset [23]	Generator sys.	Generate playable mechanics using a constraint solver+planner
mechanics (game-economic)	Machinations [1]	Flash Tool	Conceptually analyze mechanics designs (not software artefacts)
	MM, MPL [this paper]	MeDeA	Statically analyze & modify embedded mechanics using patterns
discrete domain games	Gamelan [14]	"Tool set"	Analyze Gamelan game dynamics against Computational critics
stories of Zelda-like 2D RPGs	Plot points sequence [7]	Game Forge	Generate & render playable game world & story configurations
stories in interactive drama	Praxis [6]	Prompter	Author & analyze stories in agent-based interactive drama

Table 1: Game Description Languages and tools for authoring, analyzing and modifying different game facets

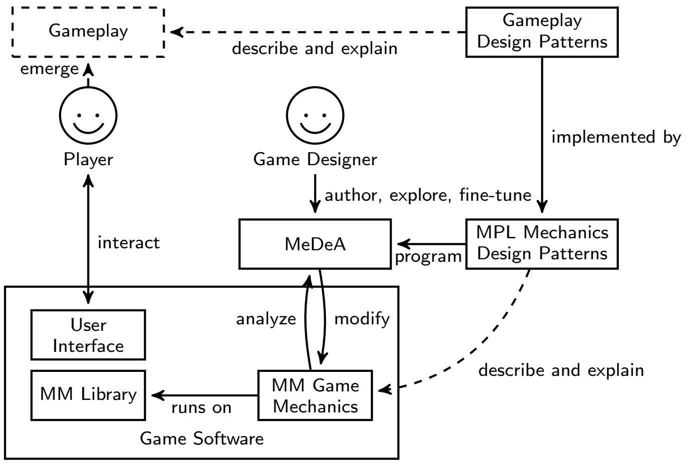


Figure 1: Relating design patterns to a game system

2. RELATED WORK

We relate our work to languages, game design patterns and design tools providing analysis or procedural generation techniques. Fig. 1 schematically shows how a designer modifies mechanics during play testing for improving the gameplay of a working game system, which includes both players and software. Before we elaborate on our approach for improving designer productivity we first sketch the research context.

Languages. Game designers have expressed the need for a common game design vocabulary [4]. Languages and tools are necessary for describing, communicating and improving game designs using both mental frameworks and authoring tools for constructing parts of game software. We relate our perspective to the Mechanics Dynamics and Aesthetics (MDA[1]) framework of Hunicke *et al.* who describe an approach to game design and game research [8]. As shown in Fig. 1, we view mechanics as part of the game software, dynamics as its run-time and player interaction, and aesthetics as good player experience (gameplay) that emerges. MeDeA is an authoring tool that uses Micro-Machinations (MM), a language for game-economic mechanics described by Klint, van Rozen and Dormans [10, 21]. MM is an evolutionary successor of Machinations, a mental framework for understanding the effect of game mechanics on gameplay introduced by Adams and Dormans [1]. MM and MPL are so-called Domain-Specific Languages (DSLs) [20], declarative languages that offer designers expressive power focussed specifically on game-economic mechanics [10]. Domains other than game development have benefited from DSLs and seen substantial gains in productivity and software quality at the cost of maintenance and education of DSL users [20]. Unlike ontologies [22] or mental frameworks that focus on analyzing and understanding games, DSLs also serve to modify software systems.

Patterns. Kreimeier made a general case for game design patterns [11]. Design patterns have been used to describe recurring patterns of gameplay [2] and game mechanics [1]. Björk *et al.* propose game design patterns for analyzing and describing gameplay that enable understanding and explaining games and how they relate, and together form a game design body of knowledge [2]. We view these patterns as *gameplay design patterns*, which can be used to understand what gameplay goals a completed game has, or should have while play testing during a game's construction. In contrast, *game mechanics patterns* can be used to describe how a game's mechanics work before it is built, and to modify them afterwards, offering a way to achieve gameplay goals. MPL can be used to model mechanics design patterns and MM offers a way to implement a game's mechanics aimed at improving gameplay as shown in Fig. 1. We view both pattern kinds as complementary, and discovering relationships between them as an exciting direction of future research. Unlike gameplay design patterns, MeDeA facilitates understanding mechanics by recognizing and visualizing patterns that run in software, and offering high-level explanations about their dynamics.

Tools. Game Description Languages (GDLs) are DSLs for the game domain. GDLs, tools and libraries have been described that offer game designers affordances for authoring, analyzing, understanding and modifying different game facets, e.g., measuring a game's qualities or by procedurally and/or iteratively generating content in a mixed-initiative way [17] or for automatic improvements. Comparing GDLs is hard because their goals, tools and notations differ, and their expressive power over specific or general abstractions and behaviors for achieving novel gameplay goals vary greatly. Therefore their effectiveness and usage scenarios range from game- or genre-specific, to more general and abstract. Table 1 shows a small list of representative GDLs. Disciplines include modeling *mechanics* of game-economies [10, 21], avatar acts [23] and board games [3, 16] but also *levels* [17], *missions* and *stories* of e.g., Zelda-like 2D games [7] or interactive drama [6]. Declarative textual notations have been proposed for specific classes of games. Nelson an Mateas' game design assistant facilitates iteratively prototyping micro-games with stock mechanics by authoring nouns, verbs and constraints [13]. Smith *et al.* propose prototyping abstract board-like games in a lightweight logic-

[1]Not to be confused with the Model-Driven Architecture (MDA), a software design approach for developing software systems by the Object Management Group (OMG) and a proprietary form of Model Driven Engineering (MDE).

Figure 2: Screenshot of a Johnny Jetstream prototype that embeds the Micro-Machinations library

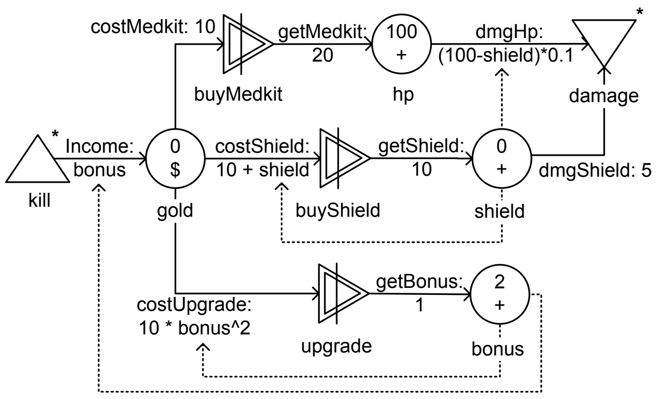
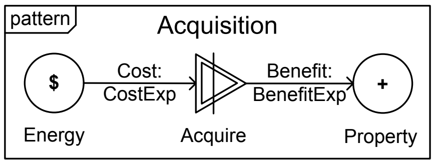


Figure 3: MM diagram of example mechanics of JJ



Intent: Activating converter ⟨Acquire⟩ costs ⟨CostExp⟩ resources from pool ⟨Energy⟩ as specified by resource connection ⟨Cost⟩ and yields ⟨BenefitExp⟩ resources in pool ⟨Property⟩ as specified by resource connection ⟨Benefit⟩.

Figure 4: Palette: an Acquisition pattern

based sketching language that models game state and mutation events on the BIPED system, which integrates the logical LUDOCORE engine and produces gameplay traces using answer-set-programming [16]. Browne and Maire demonstrate the feasibility of evolutionary techniques for closed-system combinatorial board games by measuring specific playability qualities in Ludi [3]. In contrast, Osborn *et al.* argue for a more general procedural language. They propose Gamelan for games over discrete domains (e.g., board, card games), and modular computational critics that can measure rule coverage, turn-taking fairness and existence of uninteresting strategies [14]. Evans and Short describe the Versu storytelling system, which is specific for interactive drama but also more generally reusable. Their Praxis language describes autonomous agents and social practices, and Prompter is a tool for debugging emergent stories [6].
MM is a visual and textual DSL aimed at reuse that raises the abstraction level above that of game- or genre-specific, tightly integrated languages. MM separates the concern of game economies and supports both closed systems and integration into larger systems composed of parts. In prior work, MM was analyzed against invariants using model-checking [10], and used to rapidly modify the mechanics of a tower defense game iteratively at run-time using the embeddable reusable MM library[2] [21]. MeDeA extends MM's tool set and the spectrum of mixed-initiative interactive game design tools with a novel and general pattern-based approach for deciding how to improve a game's mechanics, offering alternatives designers might otherwise overlook.

3. MECHANICS DESIGN ASSISTANT

We present a Game Mechanics Design Assistant (MeDeA) and evaluate it using game mechanics of Johnny Jetstream (JJ), a 2D side-scrolling shooter developed at Dutch game business IC3D Media that embeds MM. Fig. 2 shows a screenshot of JJ. MM programs can be represented as visual diagrams which are directed graphs that consist of two kinds of elements, *nodes* and *edges*. Both may be annotated with extra textual or visual information. These elements describe the rules of internal game economies, and define how *resources* are step-by-step propagated and redistributed through the graph. A diagram works inside a game, controlling its internal economy, and when set in motion through run-time and player interaction, the nodes redistribute resources along the edges through the graph.

[2]https://github.com/vrozen/MM-Lib

Fig. 3 shows a sample MM implementation part of JJ that is hard to read for novices. This is our running example. We introduce MM and the Mechanics Pattern Language (MPL) by analyzing it against patterns with MeDeA. The tool explains the design by generating explanations from patterns and templates explained in Section 3.1. Next, we describe the mixed-initiative decision making process of MeDeA in Section 3.2 and show how MeDeA also assists in authoring the example using the same patterns in Section 3.3. We sketch MeDeA's architecture in Section 3.4.

3.1 Pattern-Based Analysis and Explanation

We analyze and explain example mechanics of JJ using a collection of patterns we call our *pattern palette*. Although usually such a collection is called a catalogue, we will show that the term palette more closely resembles its uses. Our example palette is written in MPL, which enables palette composition and maintenance. It is based on a subset of patterns proposed by Adams and Dormans, to whom we refer for a high level discussion that omits programmed parameterized patterns as discussed here [1]. We introduce MM and MPL by example, briefly stating pattern intent. For conciseness we omit general explanations on motivation and applicability, focusing on how the patterns work. We do not claim these patterns are complete for explaining and authoring all interesting MM diagrams, and we expect that designers will have varying opinions on what makes a good palette. Here we give concise explanations. We provide a complete description[3] and a movie[4] as supplemental material. We now begin MeDeA's automated analysis.

[3]https://vrozen.github.io/fdg2015/fdg2015_rozen.pdf
[4]https://vrozen.github.io/fdg2015/MeDeA_analysis.mp4

role	A_1	A_2	A_3
Energy	gold	gold	gold
Acquire	buyMedkit	buyShield	upgrade
Property	hp	shield	bonus
Cost	costMedkit	costShield	costUpgrade
CostExp	10	10 + shield	10 * bonus^2
Benefit	getMedit	getShield	getBonus
BenefitExp	20	10	1

Table 2: Acquisition pattern cases in JJ

Acquisition. Designers can apply the Acquisition pattern for offering players a way to acquire property by spending currency. The visual representation of the Acquisition pattern is shown in Fig. 4. Visual MPL can be distinguished from MM by the a rectangle with a crooked edge pattern label on the top left, its name appearing in the top center. Patterns consist of named elements called *participants*. Acquisition has five participants, three nodes and two edges. Each participant name represents the *role* the participant plays in its context. We explain them one by one.

- **Energy** is a node of type *pool*, which abstracts from an in-game entity modeled by an MM diagram and can contain resources. Pools appear as a circle that contains an optional amount of starting resources, and zero or more *categories* that specify design intent. In our example palette, the category symbol (\$) marks that a node abstracts from currency such as gold, crystals or lumber, and is intended for spending.
- **Property** is another pool. Its category symbol (+) marks that it abstracts from a diagram node whose resources players desire to have such as health (hp).
- **Cost** is a *resource connection*, an edge with a *flow rate expression* that specifies the amount of resources that can flow, in this case how much the acquisition costs.
- **Benefit** is another resource connection that specifies how much Property the acquisition yields.
- **Acquire** is a *converter*, appearing as a triangle pointing to the right with a vertical line through the middle, that converts one type of resource into another. Acquire is the only node that *acts* in this pattern by *pulling* resources along its *input* (Cost), and *pushing* resources along its *output* (Benefit). Using the game's user interface, players can activate nodes that have an *interactive activation modifier*, visually marked with a double line, but converters only work when all resources on its inputs are available.

MeDeA analyzes our example diagram of Fig. 3 against the Acquisition pattern, and recognizes three cases. Fig. 5 shows pattern instances A_1, A_2 and A_3 (in light gray) that overlap in role Energy played by diagram pool gold (dark gray). Table 2 shows how roles are assigned over diagram elements and flow rates. MeDeA uses the pattern template to explain each pattern case to designers as shown in Table 3.

Dynamic Engine. Players require resources for activating one or more game-economic actions, in this case *gold*. The Dynamic Engine pattern, shown in Fig. 6, introduces income

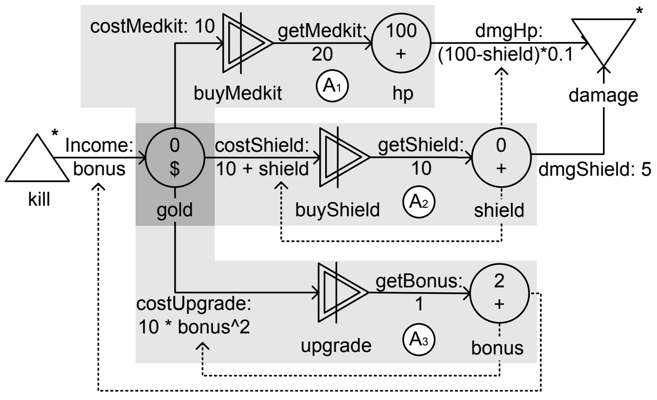
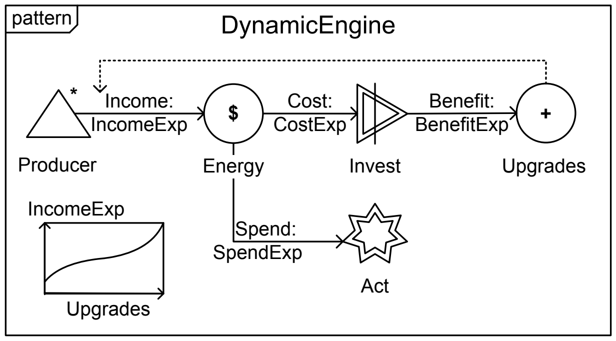


Figure 5: MM diagram showing three Acquisition cases

case	explanation
A_1	Activating converter *buyMedkit* costs *10* resources from pool *gold* as specified by resource connection *costMedit* and yields *20* resources in pool *hp* as specified by resource connection *getMedkit.*
A_2	Activating converter *buyShield* costs *10+shield* resources from pool *gold* as specified by resource connection *costShield* and yields *10* resources in pool*shield* as specified by resource connection *getShield.*
A_3	Activating converter *upgrade* costs *10*bonus*^{2} resources from pool *gold* as specified by resource connection *costUpgrade* and yields *1* resources in pool *hp* as specified by resource connection *getBonus.*

Table 3: MeDeA explains three Acquisition cases in JJ



Intent: Source ⟨Producer⟩ produces an adjustable flow ⟨Income⟩ of ⟨IncomeExp⟩ resources. Players can invest using converter ⟨Invest⟩ to improve the flow. **Apply when:** Apply Dynamic Engine for introducing a trade-off between spending currency ⟨Energy⟩ on long-term investment ⟨Invest⟩ and short-term gains ⟨Act⟩.

Figure 6: Palette: a Dynamic Engine pattern

and a tradeoff between long-term investments and short-term gains [1]. We explain its participants.

- **Producer** is a *source.* A *source* node, appearing as a triangle pointing up, is the only element that can generate resources. A source can be thought of as a pool with an infinite amount of resources. It can push any amount of resources, and therefore provides the flow rates specified by its outputs to the respective targets. The automatic activation modifier (*) of *Producer* specifies it automatically provides *IncomeExp* resources via resource connection *Income.*
- **Energy** is a pool specifying currency (\$) gained.

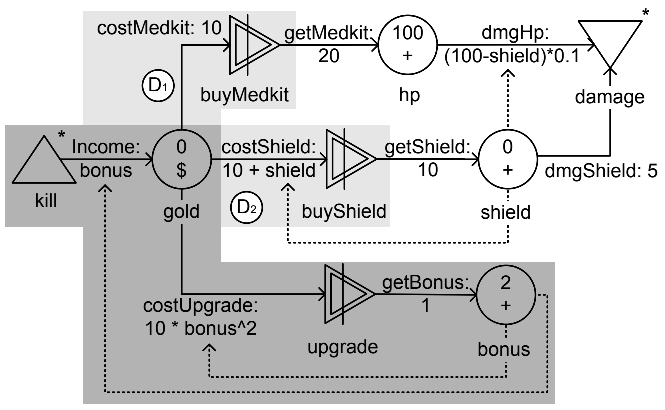
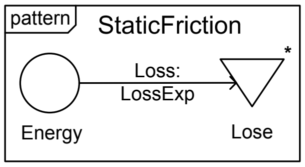


Figure 7: MM diagram showing two Dynamic Engine cases



Intent: Drain ⟨Lose⟩ causes a loss by pulling flow rate ⟨LossExp⟩ via resource connection ⟨Loss⟩ from pool ⟨Energy⟩.

Figure 8: Palette: a Static Friction pattern

- **Invest** is a converter converting Energy into Upgrades.
- **Act** is an *abstract node*, visually represented as a star, that represents any kind of node that players can activate for spending Energy. In this pattern it also represents an alternative user action to Invest.
- **Upgrades** is a pool whose resource amount influences *Income* positively and is marked as property (+).
- **Income** is a resource edge where its flow rate IncomeExp grows monotonically with Upgrades, as defined by the constraint on the left bottom of the diagram. This has to be the case for investment to be beneficial. A dashed edge signifying Upgrades is used in Income makes the feedback loop explicit.
- **Cost** is a resource edge specifying the cost of Invest from Energy.
- **Benefit** is resource edge specifying the benefit of Invest to Upgrades.
- **Spend** is a resource edge specifying the cost of Act.

MeDeA again analyzes our example diagram, now using the Dynamic Engine pattern and finds it twice as shown in Fig. 7, and the pattern roles are distributed as shown in Table 4. The difference between instances D_1 and D_2 is *buyMedkit* is replaced by *buyShield*. MeDeA's explanation of pattern case D_1 is shown in Table 5. We omit its explanation of D_2. We remark that in JJ *kill* happens automatically, but only when a player shoots an enemy, not every step.

Static Friction. So far our palette contains only patterns for gaining and converting resources. We need just one more pattern to explain all elements in the diagram. The Static Friction pattern is intended to counter positive effects a player tries to achieve, posing a challenge [1]. Its visual representation and pattern cases and MeDeA's explanations are shown in Fig. 8, Table 6 and Table 7. This concludes our concise explanation of MeDeA's pattern-based analysis.

role	D_1	D_2
Producer	kill	kill
Energy	gold	gold
Invest	upgrade	upgrade
Act	buyMedkit	buyShield
Upgrades	bonus	bonus
Income	income	income
IncomeExp	bonus	bonus
Cost	costUpgrade	costUpgrade
CostExp	10 + bonus^2	10 + bonus^2
Benefit	getBonus	getBonus
BenefitExp	1	1
Spend	costMedkit	costShield
SpendExp	10	10 + shield

Table 4: Dynamic Engine pattern cases in JJ

case	explanation
D_1	Source *kill* produces an adjustable flow *income* of *bonus* resources. Players can invest using converter *upgrade* to improve the flow. Apply Dynamic Engine for introducing a trade-off between spending currency *gold* on long-term investment *upgrade* and short-term gains *buyMedkit*.

Table 5: MeDeA explains a Dynamic Engine case in JJ

role	F_1	F_2
Energy	shield	hp
Loss	dmgShield	dmgHp
LossExp	5	(100-shield)*0.1
Lose	damage	damage

Table 6: Static Friction pattern cases in JJ

case	explanation
F_1	Drain *damage* causes a loss by pulling flow rate *(100-shield)*0.1* via resource connection *dmgHp* from pool *hp*.
F_2	Drain *damage* causes a loss by pulling flow rate *5* via resource connection *dmgShield* from pool *shield*.

Table 7: MeDeA explains Static Friction cases in JJ

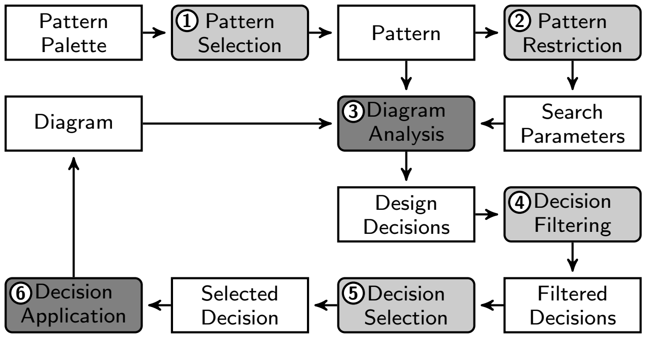
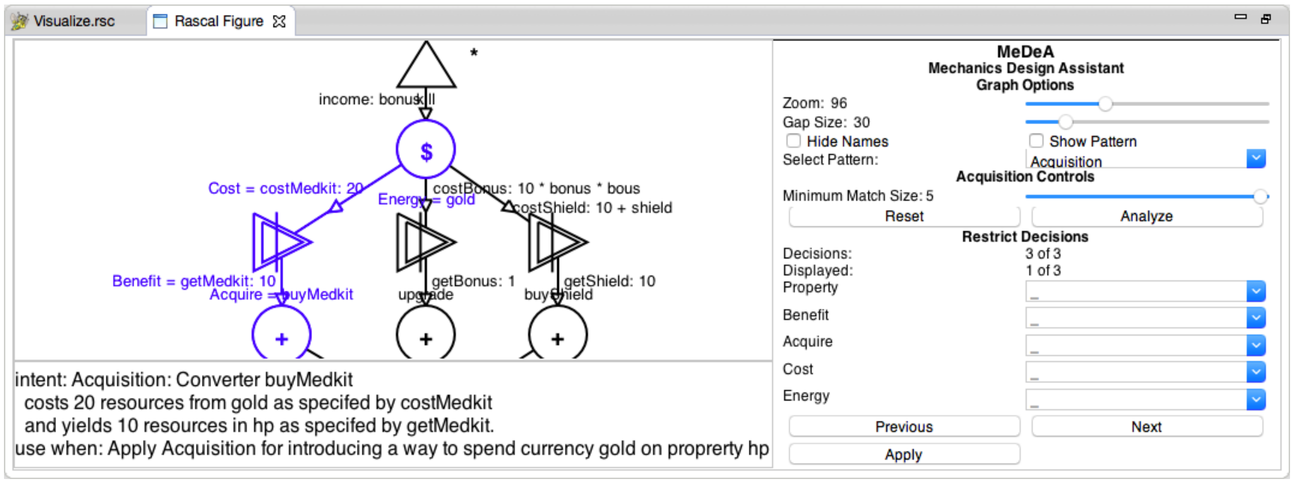


Figure 9: MeDeA iterative decision making process

3.2 Mixed-Initiative Design Decision Making

MeDeA facilitates the process of understanding and making game design decisions as we have seen in the previous section. It supports a process that consists of a sequence of simple steps as shown in Fig. 9. Rectangles represent data structures and rounded rectangles represent processes either



intent: Acquisition: Converter buyMedkit
costs 20 resources from gold as specifed by costMedkit
and yields 10 resources in hp as specifed by getMedkit.
use when: Apply Acquisition for introducing a way to spend currency gold on proprerty hp

Zoom: 96
Gap Size: 30
☐ Hide Names
☐ Show Pattern
Select Pattern: Acquisition
Minimum Match Size: 5
Decisions: 3 of 3
Displayed: 1 of 3
Property: _
Benefit: _
Acquire: _
Cost: _
Energy: _

Figure 10: MeDeA: A Pattern-Based Game Mechanics Design Assistant

primarily controlled by the user (light gray) or MeDeA (dark gray). By default MeDeA recognizes a full pattern in a diagram, associating a diagram element to each role in the pattern. However, MeDeA can also recognize partial matches, where not all roles in the pattern are represented, yielding a possibly large set of *extension points* to the diagram with respect to the pattern. Each extension point is associated with a set of pattern elements that did not match we call the *extension*. Given a designer's decision to apply the pattern, we reason that each extension is a possible *design decision*. The problem is that exploring a large set of decisions alternatives is not feasible. Our solution is to step-by-step filter the design alternatives by letting the designer associate roles to diagram elements for cherry picking from a restricted set of alternatives. We explain the decision-making process and detail the steps designers take.

1. **Pattern Selection.** Select a pattern from the palette displayed by MeDeA for modifying a diagram.
2. **Pattern Restriction.** Choose the minimum amount of pattern elements (minimum match size) for which roles must be mapped to diagram elements, affecting the size of the extension point. Optionally, restrict the search for design decisions by step-by-step assigning roles to names of elements in the diagram. This yields role constraints that target the search.
3. **Diagram Analysis.** Initiate the analysis of the diagram against the pattern. MeDeA generates a potentially large set of design decisions, every way the pattern applies to the diagram, given the minimum match size and predefined role constraints.
4. **Decision Filtering.** Filter the generated design decisions by further assigning roles to names of elements in the diagram. This yields additional role constraints that exclude generated decisions by filtering them out.
5. **Decision Selection.** Visually inspect the remaining design decisions, and select one that expresses design intent. MeDeA shows a visual rendering of diagram resulting from the design decision, providing colors for distinguishing between existing (black), found (blue), added/not found (green). For each visual representation MeDeA also shows the design intent specified by the pattern in which roles and amounts are replaced by their concrete names and values in the diagram.
6. **Decision Application.** Apply the selected design decision after providing MeDeA with names for all added elements (green) and flow amounts for all added edges. Additionally, found elements (blue) are optionally adjusted and replaced. MeDeA then then replaces the current diagram by the newly created one. This concludes the iteration, continue at step 1.

3.3 Assisted Game Mechanics Authoring

We now demonstrate how MeDeA assists in authoring[5] the mechanics of our running example shown in Fig. 3 by following the decision process of Fig. 9. We start with an empty diagram that we view as an empty canvas. First we select the Acquisition pattern from our palette. The pattern cannot match, because the canvas is empty, but we restrict the pattern to a minimum match size of zero, allowing us to add it. We provide MeDeA with the role names and edge flow rates specified by A_1 shown in Table 2 and apply the change. We again select Acquisition, restricting the pattern to a minimum match size of one, which yields seven decisions. We restrict the role *Energy* to *gold*, leaving four decisions. We select the decision where the other roles are added, providing values for them from A_2 as before and repeat these steps for A_3. Our diagram now has the elements highlighted with gray in Fig. 5.

Next we select the Dynamic Engine pattern from our palette and restrict the minimum match size to five. MeDeA offers twenty-seven design decisions. We associate the role *Upgrades* to *bonus*, *Energy* to *gold* and *Invest* to *upgrade*, leaving just three decisions. We pick one of two where *buyMedkit* or *buyShield* play role *Act* and apply it. The unassigned roles are *Producer* and *Income*, which we name *kill* and *income* respectively. We give *income* a flow rate of *bonus*, thereby satisfying the pattern constraint. Our diagram now contains the elements highlighted in gray in Fig. 7.

[5]`https://vrozen.github.io/fdg2015/MeDeA_authoring.mp4`

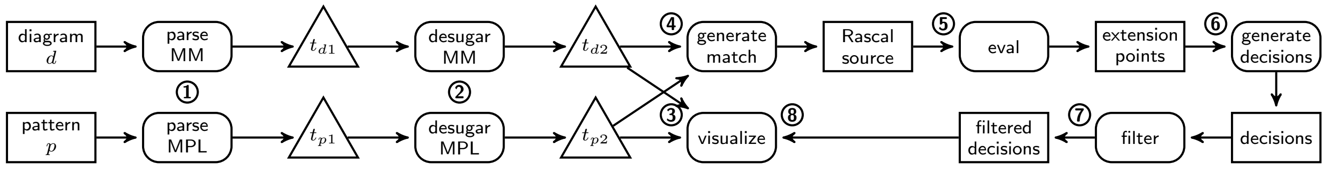


Figure 11: MeDeA: Model Transformation Steps

Next we select the Static Friction pattern from our palette, restricting *Energy* to *hp* yielding one design decision. We apply it, providing vales for pattern roles *Loss* and *Lose* as specified by F_1 shown in Table 6. Finally, we select Static Friction one more time for also applying Static Friction to *shield* using the values specified by F_2 for pattern role *Loss*. With these simple steps our diagram is now complete.

3.4 Tool Architecture

MeDeA is implemented in RASCAL, a functional metaprogramming language and language workbench for source code analysis and manipulation [9]. MeDeA's implementation[6] counts just 2.7K lines of code excluding comments and whitespace. Each of MeDeA's analysis and transformation steps is controlled from its UI shown in Fig. 10, which is programmed using the RASCAL Figure library as interactive visualization offering designers UI elements for mixed-initiative decision making as explained in Section 3.2.

Fig. 11 schematically shows the steps of how MeDeA processes diagrams and patterns. MM and MPL each have their own *grammar* for parsing ① textual programs that are represented as visual diagrams and patterns in this paper. MM and MPL parse trees are *imploded* against an Algebraic Data Type (ADT) such that we get Abstract Syntax Trees (ASTs). These ASTs are *desugared* ②, a transformation in which syntactic constructs added for user convenience are transposed to a more fundamental ADT that we visualize ③. In our case the desugared ADT is the same for MM and MPL, which is necessary for our approach because we rely on a RASCAL feature called *set matching* for our results.

We generate RASCAL programs ④ in which parameterized ADTs for patterns are matched against the ADTs of diagrams. Some ADT fields are parameters and some are constants, depending on the pattern, the diagram and optional user-supplied constraints for limiting the search space. When we evaluate ⑤ these programs with RASCAL it uses *backtracking* to bind the parameters to constants in every possible way, yielding a possibly large set of extension points to the diagram with respect to the pattern. From these extension points we generate design decisions ⑥, which are visualized as partial diagrams without values for the added diagram elements. Filtering ⑦ happens over this set using constraints posed by assigning diagram element names to roles. We again use RASCAL's matching to filter out unwanted solutions in which the roles (the pattern parameters) are bound to different constants. When applying a decision the current diagram is replaced with a new one ⑧ in which added elements have their variables bound to user supplied constants such as element names and flow rate expressions.

[6]`https://github.com/vrozen/MeDeA`

4. DISCUSSION

MeDeA simplifies authoring, understanding and modifying a game's game-economic mechanics, and for its users is an improvement over editing textual MM programs. However, the tool is currently limited to text or pattern-based authoring because RASCAL does not yet support visual edits on figures. Moreover, MeDeA is intended for game designers that can work as *gameplay engineers*, because mechanics modeling is an inherently complex technical discipline. MeDeA performs exhaustive calculations, which means that larger diagrams matched against patterns with fewer user-defined constraints result in larger search spaces, more extension points and longer calculation times. However, diagrams are composed of modular constructs called components [10] (abstracted away in this paper), which ensures that diagrams are relatively small. MeDeA supports a programmable extensible pattern palette for maintaining patterns, which mitigates the lack of MPL patterns mined from software.

5. CONCLUSION

We have presented a Mechanics Pattern Language (MPL) for programming patterns that capture a wide range of game-economic mechanics with shared structures and design intent, and a Mechanics Design Assistant (MeDeA) for analyzing, explaining and understanding existing mechanics that also supports exploring and applying decision alternatives for modifying mechanics. Its simple interface enables generating decisions from patterns, filtering and selecting using simple point-and-click controls, and all mechanics modifications result from applying generated design decisions. MeDeA is implemented in the RASCAL, a meta-programming language and language workbench. Of course, MeDeA is an academic prototype, and the case study on modifying the mechanics of Johnny Jetstream is a relatively small informal evaluation. Because the approach is general, embeddable, reusable and maintainable we believe it is a step towards industrially applicable game design tools. A more systematic evaluation is part of future work.

5.1 Future Work

Our pattern-based mixed-initiative model-driven game design approach can be extended by automating additional game disciplines and facets representing separated concerns.

- Our approach for generating alternative design decisions for modifying mechanics can also be used for fully automatic game design. It complements evolutionary approaches [18], which can also provide an alternative for filtering, and can drive a game generator such as the Game-O-Matic [19], Ludi [3] or the Angelina system [5]. Because MM is a reusable embeddable formalism that expresses extensible game economies in general, it is less dependent on specific generators and constrained input. Recipes can consist of names of patterns, resources and actions, and designers can analyze, modify and fine-tune the generated results.

- Modifying games *live*—while they run—may help tackle adaptivity challenges [12]. Modeling player behaviors and player experience respectively enable automating mechanics testing and play testing.

- The gap between high-level game design patterns and programming game mechanics may be bridged by further automating pattern-based transformations. Case studies on games embedding MM can help evaluate how MPL patterns relate to game design patterns, and how predictive they are for emergence. MPL can support pattern mining, relating palettes to existing game design knowledge for forming genre-specific pattern palettes of game-economic game design lore.

- A major challenge remains engineering languages and tools for different experts participating in game development processes. We argue for software language engineering of game languages which entails domain analyses and meta-programming of game design tools, e.g., writers require view points on storylines whereas game designers need view points on game mechanics, gameplay, levels and missions. Each of these view points could exist as separated concerns modified via *live* user interfaces designed to model, analyze and generate content for composing high quality games that are pieced together from sets of expressive, reusable, extensible, interoperable and embeddable formalisms.

Acknowledgements. I thank Paul Klint for discussions and proof reading, and the anonymous reviewers for their insightful comments that helped restructure this paper.

6. REFERENCES

[1] E. Adams and J. Dormans. *Game Mechanics: Advanced Game Design*. New Riders Publishing, Thousand Oaks, CA, USA, 1st edition, 2012.

[2] S. Björk, S. Lundgren, and J. Holopainen. Game Design Patterns. In *Level Up: Digital Games Research Conference 2003*, pages 4–6, 2003.

[3] C. Browne and F. Maire. Evolutionary Game Design. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(1):1–16, March 2010.

[4] D. Church. Formal Abstract Design Tools. *Game Developer*, 1999.

[5] M. Cook and S. Colton. Multi-faceted Evolution of Simple Arcade Games. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 289–296, Aug 2011.

[6] R. Evans and E. Short. Versu - A Simulationist Storytelling System. *Computational Intelligence and AI in Games, IEEE Transactions on*, 6(2):113–130, June 2014.

[7] K. Hartsook, A. Zook, S. Das, and M. Riedl. Toward Supporting Stories with Procedurally Generated Game Worlds. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 297–304, Aug 2011.

[8] R. Hunicke, M. Leblanc, and R. Zubek. MDA: A Formal Approach to Game Design and Game Research. In *In Proceedings of the Challenges in Games AI Workshop, Nineteenth National Conference of Artificial Intelligence*, pages 1–5. Press, 2004.

[9] P. Klint, T. v. d. Storm, and J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 168–177, 2009.

[10] P. Klint and R. van Rozen. Micro-Machinations: A DSL for Game Economies. In M. Erwig, R. Paige, and E. Van Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 36–55. Springer International Publishing, 2013.

[11] B. Kreimeier. The Case For Game Design Patterns. Gamasutra, 2002. `http://www.gamasutra.com/view/feature/4261/the_case_for_game_design_patterns.php?print=1`.

[12] R. Lopes and R. Bidarra. Adaptivity Challenges in Games and Simulations: a Survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(2):85–99, 2011.

[13] M. J. Nelson and M. Mateas. An Interactive Game-Design Assistant. In *Proceedings of the 2008 International Conference on Intelligent User Interfaces*, pages 90–98, 2008.

[14] J. C. Osborn, A. Grow, and M. Mateas. Modular Computational Critics for Games. In *AIIDE*, 2013.

[15] T. Schaul. A Video Game Description Language for Model-Based or Interactive Learning. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8, Aug 2013.

[16] A. M. Smith, M. J. Nelson, and M. Mateas. Computational Support for Play Testing Game Sketches. In *AIIDE*, 2009.

[17] G. Smith, J. Whitehead, and M. Mateas. Tanagra: An Intelligent Level Design Assistant for 2D Platformers. In *AIIDE*, 2010.

[18] J. Togelius and J. Schmidhuber. An Experiment in Automatic Game Design. In *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*, pages 111–118. IEEE, 2008.

[19] M. Treanor, B. Blackford, M. Mateas, and I. Bogost. Game-O-Matic: Generating Videogames That Represent Ideas. In *Workshop on Procedural Content Generation in Games*, PCG'12, pages 11:1–11:8, New York, NY, USA, 2012. ACM.

[20] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN NOTICES*, 35:26–36, 2000.

[21] R. van Rozen and J. Dormans. Adapting Game Mechanics with Micro-Machinations. In *Proceedings of the Foundations of Digital Games Conference*, 2014.

[22] J. P. Zagal, M. Mateas, C. Fernández-vara, B. Hochhalter, and N. Lichti. Towards an Ontological Language for Game Analysis. In *Proceedings of International DiGRA Conference*, pages 3–14, 2005.

[23] A. Zook and M. O. Riedl. Automatic Game Design via Mechanic Generation. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 530–537, 2014.

APPENDIX

This appendix is a supplement to the above paper.

A. MECHANICS DESIGN ASSISTANT

This section details the automated pattern-based analysis of Section 3 using three additional patterns.

A.1 Pattern-Based Analysis and Explanation

Static Friction. We detail the Static Friction pattern, which is intended to counter positive effects a player tries to achieve, posing a challenge [1]. Its visual representation and pattern cases and MeDeA's explanations are shown in Fig. 8, Table 6 and Table 7. We explain the participants.

- **Energy** is a pool that represents currency or property a player has (no symbol).
- **Loss** is a resource edge that specifies the (usually constant, i.e. static) flow rate of resources lost.
- **Lose** is a *drain* that acts automatically (*), e.g., when an enemy shoots the player's ship. A *drain* node, appearing as a triangle pointing down is the only element that can delete resources. Drains can be thought of as pools with dan infinite negative amount of resources, and have capacity to pull whatever resources are available, or whatever resources are pushed into them. Lose pulls all resources specified by Loss from Energy.

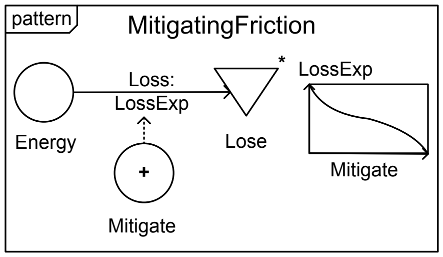
MeDeA finds two cases F_1 and F_2 of Static Friction in our example shown in Table 6 and explains them in Table 7.

Mitigating Friction. Mitigating Friction, a pattern shown in Fig. 12, is form of dynamic friction that enables players to counter the effect of losing resources [1]. Its roles differ from Static Friction as follows.

- **Mitigate** is an added pool whose resource amount negatively influences the amount of resources lost.
- **Loss** is a resource edge whose flow rate expression is modified such that it shrinks monotonically with Mitigate. Visually, this added constraint is shown inside the pattern by a diagram on the right. Additionally, a dashed edge signifies *Mitigate* is used in *Loss*, which makes the feedback loop explicit.

MeDeA finds one case F_3 of Mitigating Friction that extends Static Friction case F_2 where pool *shield* plays role *Mitigate* shown in Table 8. The explanation MeDeA gives is shown in Table 9.

Stopping. The Stopping pattern, shown in Fig. 13, is a way to express the law of diminishing returns [1]. Compared to the Acquisition pattern we just impose one extra constraint, namely that the flow amount *CostExp* on resource edge *Cost* increases monotonically with the amount of *Property* owned. Visually, this is represented by the constraint diagram appearing on the right. MeDeA finds two cases of Stopping that are also Acquisition cases A_2 and A_3. The explanations MeDeA gives are shown in Table 10.



Intent: Mitigates a loss caused by drain ⟨Lose⟩, because the flow rate ⟨LossExp⟩ on resource connection ⟨Loss⟩ decreases with the amount of resources in pool ⟨Mitigate⟩.

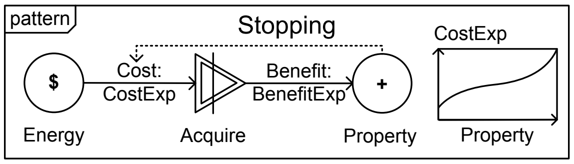
Figure 12: Palette: a Mitigating Friction pattern

role	F_1	F_2	F_3
Energy	shield	hp	hp
Loss	dmgShield	dmgHp	dmgHp
LossExp	5	(100-shield)*0.1	(100-shield)*0.1
Lose	damage	damage	damage
Mitigate			shield

Table 8: Friction pattern cases in JJ

case	explanation
F_1	Drain *damage* causes a loss by pulling flow rate *(100-shield)*0.1* via resource connection *dmgHp* from pool *hp*.
F_2	Drain *damage* causes a loss by pulling flow rate *5* via resource connection *dmgShield* from pool *shield*.
F_3	Mitigates a loss caused by drain *damage*, because the flow rate *(100-shield)*0.1* on resource connection *dmgHp* decreases with the amount of resources in pool *shield*.

Table 9: MeDeA explains Friction cases in JJ



Intent: Makes activating converter ⟨Acquire⟩ increasingly expensive because flow rate ⟨CostExp⟩ on resource edge ⟨Cost⟩ increases with the resource amount in pool ⟨Property⟩.

Figure 13: Palette: a Stopping pattern

case	explanation
A_2	Makes activating converter *buyShield* increasingly expensive because flow rate *10+shield* on resource edge *costShield* increases with the resource amount in pool *shield*.
A_3	Makes activating converter *upgrade* increasingly expensive because flow rate $10*bonus^2$ on resource edge *costUpgrade* increases with the resource amount in pool *bonus*.

Table 10: MeDeA explains two Stopping cases in JJ