Adapting Game Mechanics with Micro-Machinations

Riemer van Rozen, Joris Dormans

To cite this version:

Riemer van Rozen, Joris Dormans. Adapting Game Mechanics with Micro-Machinations. Foundations of Digital Games, Apr 2014, Aboard Royal Caribbean Liberty of the Seas, sailing from Ford Lauderdale, Florida, United States. hal-01110847

HAL Id: hal-01110847
https://inria.hal.science/hal-01110847v1

Submitted on 29 Jan 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Adapting Game Mechanics with Micro-Machinations

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

Joris Dormans[‡]
Amsterdam University of Applied Sciences
Duivendrechtsekade 36-38 1096 AH
Amsterdam, The Netherlands
j.dormans@hva.nl

ABSTRACT

In early game development phases game designers adjust game rules in a rapid, iterative and flexible way. In later phases, when software prototypes are available, play testing provides more detailed feedback about player experience. More often than not, the realized and the intended gameplay emerging from game software differ. Unfortunately, adjusting it is hard because designers lack a means for efficiently defining, fine-tuning and balancing game mechanics. The language Machinations provides a graphical notation for expressing the rules of game economies that fits with a designer's understanding and vocabulary, but is limited to design itself. Micro-Machinations (MM) formalizes the meaning of core language elements of Machinations enabling reasoning about alternative behaviors and assessing quality, making it also suitable for software development. We propose an approach for designing, embedding and adapting game mechanics iteratively in game software, and demonstrate how the game mechanics and the gameplay of a tower defense game can be easily changed and promptly play tested. The approach shows that MM enables the adaptability needed to reduce design iteration times, consequently increasing opportunities for quality improvements and reuse.

1. INTRODUCTION

In computer game development, developers face problems that arise from increased complexity. Challenges include increased development speed, changing technologies, and growing teams of experts with varying vocabularies. Teams may consist of artists, software engineers, domain experts and game designers who work together to create games. Practice is still catching up with the relatively new profession of the computer game designer. Game design, despite the large number of games produced today lacks a common vocabulary, methods for designing games and sharing knowledge and artifacts. The need for common design vocabularies [6,23], game design patterns [4], specialized game grammars [9,16], and computer assisted design tools [18,19] has been expressed for some time, but so far no tool, method, or framework has surfaced as an industry standard. As result, game design relies strongly on iterative prototyping, play-testing, and reprogramming parts to improve games.

Good gameplay is an emergent property of the game system defined by the game mechanics [10]. Therefore, gameplay can only be evaluated after the system has been built and set into motion through play. More often than not, the realized gameplay differs from the intended gameplay. Because a designer's understanding about how game rules affect the player is constantly changing, they have to make adaptations quickly and often. However, as software development progresses, making changes becomes harder and more time consuming. This seriously compromises the ability of the designer to design, play test, gain feedback and improve, which results in longer design iterations and missed opportunities for improving the quality. From a software engineering point-of-view gameplay adaptations represent a problematic stream of badly defined, poorly understood requirements that result in wasted effort, ineffective attempts at reuse, and repetitive and error-prone coding cycles, instead of focus on maintenance, libraries and tools.

We aim to accelerate the game development process by boosting game designer productivity and improving quality feedback. The language Machinations [1] provides a graphical notation for expressing the rules of game economies that is gaining popularity with designers. Micro-Machinations (MM) [14] formalizes the meaning of core language features of Machinations and adds modularity, making it also suitable for software development and formal analysis.

We propose a game design approach for adapting mechanics using MM, that aims for brief design iterations with informed and well-documented design decisions. The approach entails modeling game mechanics as embeddable software artifacts with MM. Additionally, it provides a means for adapting game mechanics at run-time using a library. We demonstrate that changes to the game economy of a tower defense game can be easily designed and embedded in software. Our approach shows that it is feasible to significantly reduce design iteration times, by improving flexibility and adaptability, thereby increasing opportunities for quality improvements and reuse.

[*]This work is part of the EQuA Project. http://www.equaproject.nl
[†]This research is performed at the SWAT group of CWI.
[‡]This work is part of the Automated Game Design Project.

2. BACKGROUND

2.1 Related Work

Language-oriented approaches for game development include Petri Nets [3,5], state machines [13], and rule based and constraint based systems [17, 19]. Additionally, configuration files have been used for loading constants, and interpreters that provide run-time adaptability using scripts have been embedded in games, e.g. Lua [22] and Python [8]. Commercial game engines also include script languages [12] and graphical languages [11]. Script languages are general purpose solutions that can be used for solving problems in many domains. In contrast, a Domain-Specific Language (DSL) focuses on providing increased expressiveness over solutions for a smaller set of problems. We adopt the DSL definition of van Deursen *et al.* [21] which states that: *"A Domain-Specific Language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain".* DSLs have advantages over script languages such as improved productivity and quality, division of labor. However, DSL approaches also come at a cost. Time and effort goes into analyzing a domain, choosing appropriate notations and meanings, learning the language and maintaining it. In comparison, using an existing script language is technically easier to achieve, e.g. through its meta-programming capabilities Lua has been used for creating internal DSLs [15].

2.2 Machinations Evolution

We give a brief history of the evolution of the Machinations language, its tools and frameworks, from its inception for game design to its formalization and use in game software.

Game Design. The original Machinations framework, intended solely for *game design*, helps designers to design, understand, and balance complex game systems [1, 10]. Its starting point is the notion of internal economy for games [2] that describes game dynamics in terms of distribution and flow of game resources. Game resources include tangible resources such as money, property, and food, but the concept also applies to abstract notions such as hit points, experience points, and strategic advantage. The framework uses a diagrammatic language to visualize a game's internal economy. Machinations diagrams foreground the structural characteristics of the game economy that are critical in the emergence of gameplay. In particular dynamic gameplay can be attributed to feedback loops in the internal economy [2, 20]. The framework augments paper prototyping, and can be used to assess game balance, emergent properties, and potential dominant strategies. Diagrams are abstract, dynamic, and playable representations of a game. Moreover, they are game design artifacts only, and not suitable as software requirements, since they lack a programmable semantics. Machinations is used in education and practice.

Formal Analysis. Micro-Machinations (MM) is a DSL that we describe in Section 2.3 intended for both *game design* and *software development*. It is a formalized extended subset of Machinations which adds a precise meaning to the design notation, enabling formal reasoning about alternative model behaviors and assessing their quality. MM also introduces new features, notably modularization, and has both a visual and a textual notation. Micro-Machinations Analysis in RASCAL (MM AiR) is a framework for analyzing MM that uses the RASCAL meta-programming language[1] and the SPIN model checker [14]. It offers an IDE that reads textual MM and displays visual MM for simulating models interactively or randomly and analyzing behaviors partially or exhaustively.

Software Development. Thus far, MM have only been analyzed. Here, we introduce a library for building games with MM. The Micro-Machinations Library (MM Lib) is a light-weight software library written in C++ for embedding MM in games and tools[2]. MM Lib tackles technical challenges related to interoperability, traceability and debugging. In particular, it enables embedding and adapting models in game software and replaying behavior traces. The library interprets textual MM as changes that are reflected at run-time in the game economy state, enabling adapting model elements between evaluation steps. Developers can integrate MM Lib using its simple embedding APIs, most notably for evaluating model changes, activating nodes, stepping to a next state and informing its context about changes to type definitions and instances.

2.3 Micro-Machinations

MM models are directed graphs consisting of *nodes* and *edges*, which can be annotated with extra information. They describe the rules of internal game economies and define how resources are redistributed step by step between nodes. We provide a description of modeling elements needed for the case study of Section 4, starting with the basic elements of Figure 1. For conciseness, in this paper we use only the visual variant. A more detailed explanation is provided in [14].

A *pool* is a named node, that abstracts from an in-game entity, and can contain *resources*, such as coins, crystals, health, etc. Visually, a pool is a circle with an integer in it representing the current amount of resources, and the initial amount at which a pool starts when first modeled.

A *resource connection* is an edge with an associated expression that defines the rate at which resources can flow between source and target nodes. During each transition or *step*, nodes can *act* once by redistributing resources along the resource connections of the model. The *inputs* of a node are resource connections whose arrowhead points to that node, and its *outputs* are those pointing away.

The *activation modifier* determines if a node can act. By default, nodes are *passive* (no symbol) and do not act unless activated by another node. *Interactive* (double line) nodes signify user actions that during a step can activate a node to act in the next state. *Automatic* (*) nodes act automatically, once every step. Nodes act either by pulling (default, no symbol) resources along their inputs or pushing (p) resources along their outputs. Nodes that have the *any modifier* (default, no symbol), interpret the flow rate expressions of their resource connections as upper bounds, and move as many resources as possible. Additionally, these nodes may process their resource connections independently and in any order. Nodes that instead have the *all modifier* (&) inter-

[1] http://www.rascal-mpl.org

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

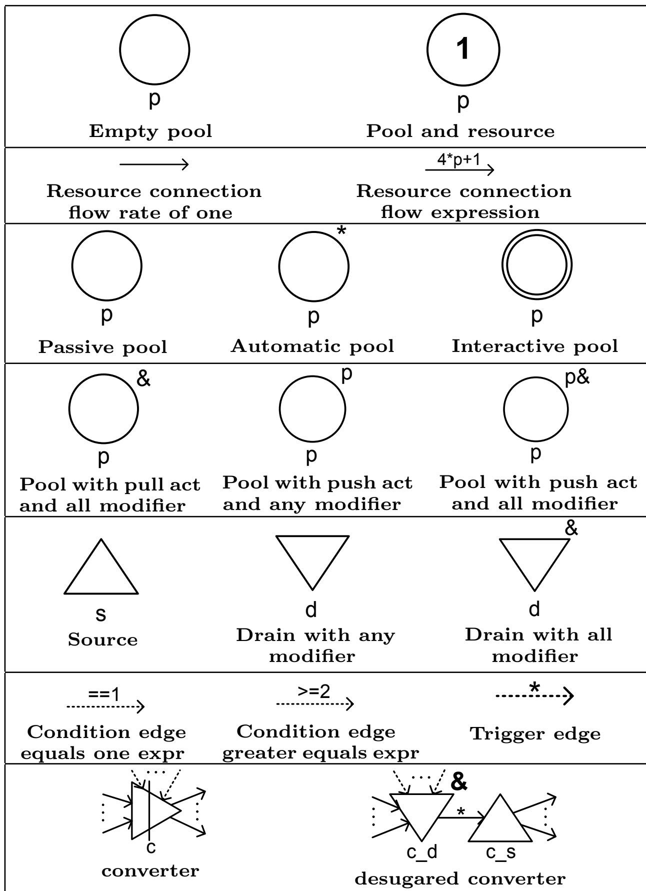


Figure 1: Visual Micro-Machinations of Basic Elements

pret them as strict requirements, and the associated flows all happen or none do.

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, and therefore always pushes all resources or all resources are pulled from it. The any modifier does not apply, and resources may never flow into a source. Also, infinite amounts may not flow from sources during a step.
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 an infinite negative amount of resources, and have capacity to pull whatever resources are available, or whatever resources are pushed into them. No resources can ever flow from a drain.

A node can only be active if all of its *conditions* are true. A *condition* is an edge appearing as a dashed arrow with an associated Boolean expression. Its source node is a pool that forms an implicit argument in the expression, and the condition applies to the target node. A *trigger* is an edge that appears as a dashed arrow with a multiply sign. The origin node of a trigger activates the target node when for each resource connection the source works on, there is a flow in the transition that is greater or equal to that of the associated flow rate expression. Additionally, automatic pulling nodes without inputs and automatic pushing nodes without outputs always activate their trigger targets.

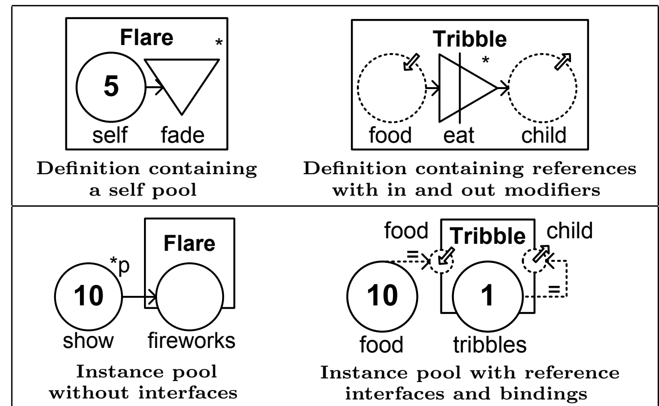


Figure 2: Visual Micro-Machinations of Modular Elements

Converters are nodes, appearing as a triangle pointing right with a vertical line through the middle, that consume one kind of resources and produce another. Converters are not core elements because they can be rewritten as a combination of a drain, a trigger and a source. Unlike basic node types, converters therefore take two steps to complete. Converters always implicitly have pull and all modifiers.

We now explain modularity features, shown in Figure 2. A *type definition* is a named diagram that functions as parameterized module for encapsulating elements. Type definitions define internal elements and how they can be used externally. A *reference*, represented by a circle with a dashed line, is an alias that refers to a node that defines it. Internal nodes annotated with an *interface modifier input*, *output* or *input/output* become interfaces on the instances of the type. The input modifier denotes that an interface accepts inputs, output implies it accepts outputs and input/output accepts both. Interface modifiers appear as an arrow in the top right corner of a node, where an input modifier point into the node, an output modifier points out of the node, and an in-/output modifier does both.

An *instance pool* is a pool whose resource type is a definition. It represents a set of instances, objects with individual instance data, whose shared interfaces are defined by that type, and can be bound to other models, acting as formal parameters. Additionally, the size of the set is the amount of resources in the pool. Visually, an instance pool appears as a circle and a rectangle. Instances are local to a pool and cannot flow out through resource edges. Resources flowing in create new instances, and those flowing out delete them. An *interface* makes internal elements of instances available to the outside, and can be used by connecting resource connections. Visually, an interface is a small circle at the border of an instance with its name under it. Input interfaces have an arrow pointing into the circle, outputs have an arrow pointing outward, and in-/outputs have a bidirectional arrow. The direction of the arrow implies the validity of the direction of the resource edges that connect to it. Only reference interfaces appear with a dashed line. References must be bound to definitions using edges called *bindings*, represented by dashed arrows annotated with an equal sign, that originate from a defining node and target a reference. When a type definition contains a pool named *self*, instances of this type end when the pool is empty.

occupation	discipline	main artifact	expected iteration activity level			
			concept	elaboration	construction	delivery
game design	gameplay design	gameplay goals	high	high	medium	low
	mechanics modeling	mechanics model	medium	high	high	high
	play test	aesthetics feed-back	medium	high	high	high
	mechanics test	behavior analysis	low	low	low	low
software engineering	domain modeling	name bindings	low	high	low	low
	implementation	game software	low	medium	high	low

Table 1: Disciplines, artifacts and expected iteration activity when using the Micro-Machinations approach

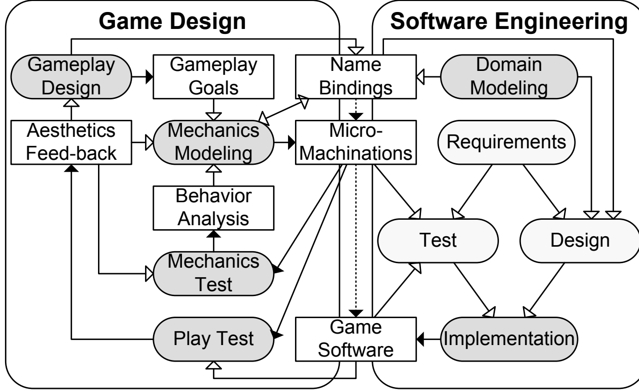


Figure 3: Game Design Approach for Adapting Mechanics

3. ADAPTING GAME MECHANICS

This section introduces a game design approach for designing, embedding and adapting game mechanics and gameplay of working software by embedding MM. Its goals are shortening design iterations, gaining immediate feedback and maximizing opportunities for quality improvements. We aim at flexible integration in processes that employ iterative development, agile practices and Scrum. The approach separates concerns, dividing the work between game designers and software engineers. Because a game designer's creative genius flourishes with expressive freedom, flexibly switching between activities, we describe only *what* artifacts game designers and software engineers create, but not *when*. Working on disciplines and artifacts is *optional* at each given time. Figure 3 shows disciplines (rounded rectangles) and artifacts (rectangles). Closed arrow heads denote an artifact is required for activities related to a discipline, whereas open arrow heads signify optional artifacts that provide added value. Table 1 describes expected iteration activity during phases of the project life cycle. These phases are 1) *concept*: when a game is conceived of and its feasibility is discussed, 2) *elaboration*: when plans are made concrete using models and diagrams documenting intended functionality, 3) *construction*: when game software is built, tested, balanced and fine-tuned, and 4) *delivery* when a product is prepared for its release. We explain the design disciplines one by one.

- **Gameplay Design:** *How should the game affect the player experience during play?* Designers describe the intended effect of rules on players [20] in what we call *gameplay goals*. Gameplay design includes taking *design decisions* concerning patterns, intended behavior and feedback loops. Activities may include paper prototyping and using other conceptual game design tools for assisting in the creative process, e.g. the Machinations tool [9].

- **Mechanics Modeling:** *What are the rules composing the mechanics?* The approach centers around the discipline of *mechanics modeling*, which involves designing an artifact using MM that we call the *mechanics model*. It describes the rules of the game, that when set in motion by run-time and player interaction aims to achieve the gameplay goals. The mechanics model is an embeddable *software artifact* that interacts with other parts of the software using its *name bindings*. It includes descriptions that relate design decisions and diagram elements to expected behavior and player interaction. Designers can add, change, balance and finetune mechanics at any development stage.

- **Play Test:** *How do the rules affect player experience?* During play testing designers validate if the mechanics model achieves the gameplay goals, gathering *aesthetics feedback* in order to make improvements. Traditionally play testing happens on paper prototypes in early stages. Mechanics models enable play testing without game software, but with the guarantee the models behave the same when embedded in games. Later, when game software is available play testing becomes less abstract. This approach enables play testing effectively throughout the development process.

- **Mechanics Test:** *How do the rules behave, and how can that be improved?* Mechanics models are not just embeddable in games. Tools such as MM AiR described in Section 2.2 can also interpret MM, enabling designers to analyze the behavior separately. Applications include interactively stepping through states and transitions, analyzing alternatives, programmable test setups prerecorded or programmed player actions for random simulation runs or exhaustive analysis, and reproducing states and transitions using MM traces [14].

For integrating the design and software engineering processes we relate shared artifacts to the software engineering disciplines of domain modeling and implementation.

- **Domain Modeling:** *What are the name bindings for embedding the mechanics in game software?* During domain analysis software engineers analyze and visualize important concepts and relationships, and agree with designers on concept names and activation points in a contract that we call *name bindings*. Changes to name bindings may cost time because they affect the software design, and integration points must be coded.

- **Implementation.** *How are the game mechanics integrated in software?* Software engineers build *game*

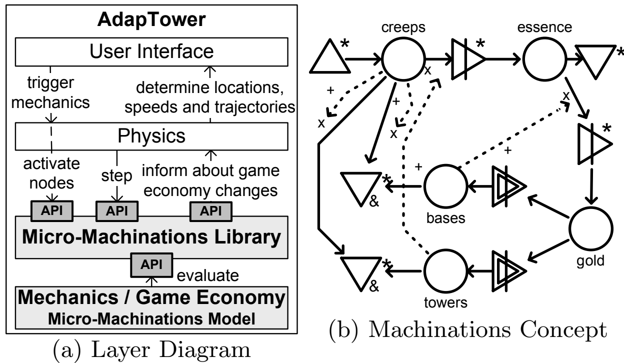
software glue libraries, program integration points specified in the name bindings, and integrate content and mechanics models with the rest of the game system. The MM Lib, introduced in Section 2.2, enables embedding the mechanics model. As soon as the first prototype runs the model, play testing can commence.

4. CASE STUDY: ADAPTOWER

4.1 Experimental setup

We demonstrate adaptability of the mechanics and gameplay of *AdapTower*, a prototype tower defense game built using the approach discussed in Section 3. The game, shown in Figure 4, is implemented in C# and embeds the MM Lib (which is C++) using a wrapper that marshals data to .NET.

Figure 4: Screenshot of Running AdapTower Prototype



(a) Layer Diagram (b) Machinations Concept

Figure 5: AdapTower Diagrams

Figure 5a shows the a partition of AdapTower in software layers. Players trigger mechanics in the User Interface (UI) layer. The Physics layer interprets user actions and tracks in-game entities, their location, speed and trajectories. It ensures the UI displays them accordingly. MM Lib interprets Physics calls to activate nodes, e.g. for collisions and time passing, and evaluates textual MM defining adaptations. MM Lib computes a next game economy state when the Physics calls the *step* API, and informs it about changes to definitions and instances with callbacks and messages.

4.2 Adapting AdapTower

Here we demonstrate adaptations to the game economy of AdapTower in a series of six design iterations. We provide visual MM definitions with additions and changes[3].

[3]The textual MM of AdapTower can be found at `https://github.com/vrozen/MM-Lib/tree/master/mm/tests/towers`

Design Iteration 0: Concept Phase

Gameplay Design. In AdapTower, the *creeps* spawn at random locations on the top of the screen and march downwards. Defensive *towers* shoot creeps and convert killed creeps into *essence*, a resource that falls down. *Bases* collect essence and convert it into *gold*, which can be used to build more towers and bases. Both are destroyed when they come into contact with the advancing creeps. To reach the objective of building a fixed number of bases the player needs to construct defensive configurations that minimize the risk of losing bases, but maximize the collection of essence. AdapTower's internal economy consists of two interconnected positive feedback loops. First, towers convert creeps into essence and bases convert essence into gold, which players use to buy more towers and bases. Second, the more creeps there are, the more likely it is they collide and destroy more towers, meaning more creeps will survive. Figure 5b shows a concept sketch modeled with the Machinations tool [1,9].

Design Iteration 1: Creeps, towers and bases

Mechanics Modeling. The first mechanics model version of the game economy of AdapTower consists of three MM models. The integrated game is modeled in Figure 6. We model *creeps*, *essence* and *gold* by pools, which are bound to *Tower* and *Base* instances on their shared interfaces using binding edges. Creeps enter the world by externally activating the interactive source *spawn* which pushes one resource along its edge to the pool *creeps*. The drain *missed* models *essence* disappearing from the world without being caught. The converters *buyTower* and *buyBase* consume 20 and 50 gold to respectively produce a tower and a base instance.

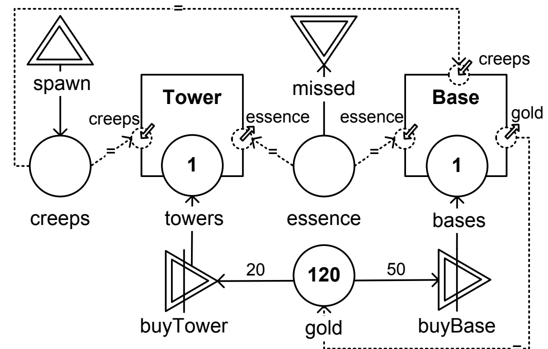


Figure 6: AdapTower Visual MM Definition

name	meaning	embedding
spawn	a creep enters the world	activate node
creeps	amount of creeps in the world	notify value
essence	amount of essence in the world	notify value
missed	essence leaves the world	activate node
towers	amount of towers in the world	notify new/del
bases	amount of bases in the world	notify new/del
gold	amount of gold the player has	notify value
buyTower	player buys a tower	activate node
buyBase	player buys a base	activate node

Table 2: Global Name Bindings

name	meaning	embedding
range	tower range in game yards	notify value
firePower	tower fire power in hit points	notify value
rotationSpeed	tower rotation speed degree/s	notify value
hitByCreep	physics: a creep hits a tower	activate node
killCreep	physics: a tower kills a creep	activate node
hitByCreep	physics: a creep hits a base	activate node
hitByEssence	physics: essence hits a base	activate node

Table 3: Tower and Base Type Definition Name Bindings

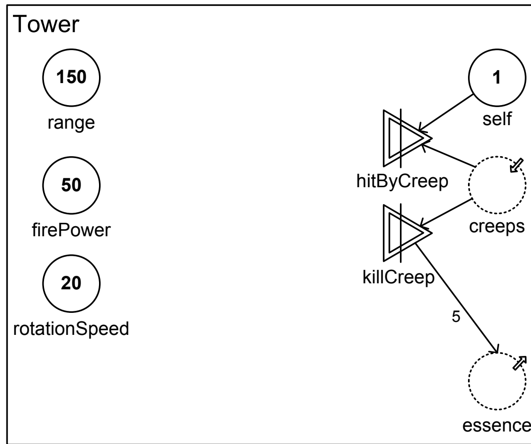


Figure 7: First Tower Visual MM Definition

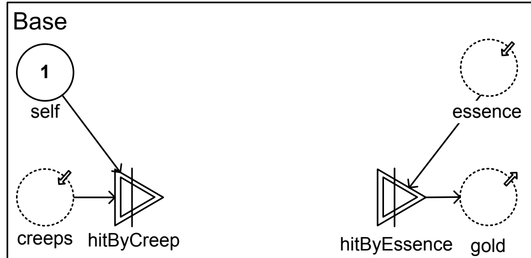


Figure 8: First Base Visual MM Definition

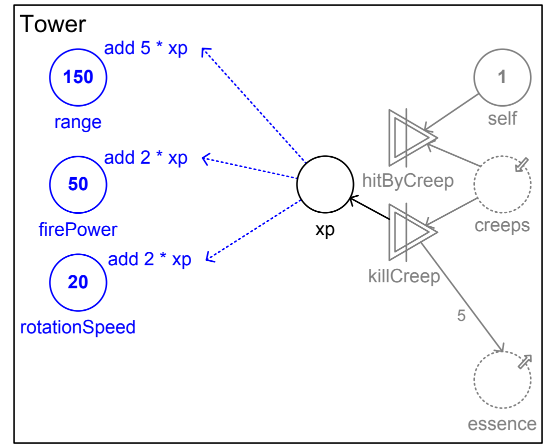


Figure 9: Second Tower Visual MM Definition

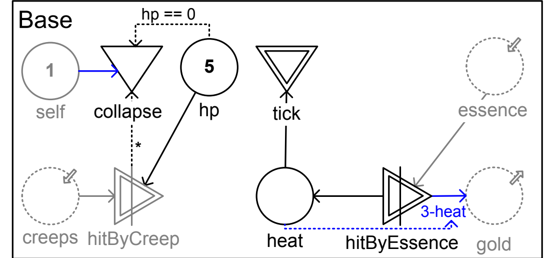


Figure 10: Second Base Visual MM Definition

Figure 7 models the first *Tower* type definition. Each tower has a *range* of 150, a *firePower* of 50 and a *rotationSpeed* of 20. When activated, converter *killCreep* pulls one resource from *creeps*, and produces five resources in *essence*. Figure 8 models the first *Base* type definition. When activated, converter *hitByEssence* pulls one resource from *essence* and generates one resource in *gold*. Tower and Base both contain a converter *hitByCreep*, which pulls one resource from external node *creeps* and one from *self*, collapsing the instance.

Domain Modeling. Table 2 and Table 3 show name bindings for embedding versions of the mechanics. Some interactive nodes are activated externally using their names, such as *spawn*, *missed*, *killCreep*, *hitByCreep*, and *hitByEssence*. Others, such as *buyTower* and *buyBase* are UI elements activated by the player. For of passive pools, such as *creeps*, *essence*, *towers*, *bases*, *gold*, *range*, *firePower* and *rotationSpeed* the game registers observers for using current values.

Play Test. The gameplay of the initial version works, but is not very exciting. Once players set up their bases and defenses, they simply need to wait and collect enough essence for quickly building the number of bases required to win.

Design Iteration 2: Towers gain experience

Mechanics Modeling. We adapt the Tower definition in Figure 9 by adding an experience pool *xp* and by changing the pools *range*, *firePower* and *rotationSpeed*, adding a bonus based on xp.

Gameplay Design. The rationale behind this change is that adding another feedback loop improves positioning effectively towers and speeds up the end game.

Design Iteration 3: Bases have health

Mechanics Modeling. We adapt the Base definition in Figure 10 by adding a pool *hp* that denotes hit points, starting at five resources. A resource is drained every time a creep hits the base, but bases only *collapse* when *hp* is empty. Additionally, bases gain heat in a pool called *heat* every time it is hit by essence. Heat inhibits the conversion from essence to gold. This change is represented by the modified flow from converter *hitByEssence* to reference *gold*. Heat diminishes over time, since every time drain *tick* activates, it pulls one resource from pool *heat*.

Gameplay Design. Hit points make bases more stable, whilst the heat mechanism introduces a negative feedback loop that diminishes the effectiveness of bases collecting a lot of essence. These changes aim to force players to spread bases for collection resources, and reduce the effectiveness of funneling all essence and creeps into a single place.

Design Iteration 4: Towers lose experience

Gameplay Design. We introduce two feedback loops to make towers more dynamic. The goal is that towers accumulate experience points much faster, but also lose them over time. At the same time the experience points inhibit the number of essence produced by each kill. The effect is that towers can go into a sort of *"killing spree"*, but these towers essence production is reduced at the same time.

Mechanics Modeling. We realize these effects by increasing the flow from converter *killCreep* to *xp* and adding a drain *tick* that pulls resources from pool *xp* in Figure 11. Additionally, the amount of generated essence is reduced by changing the flow expression to *5 - xp*.

Domain Modeling. The drain *tick* specifies the name of a node the game must activate once each second.

Design Iteration 5: Bases amplify essence

Play Test. One problem we notice is that the player is not encouraged to place bases at the top of the screen.

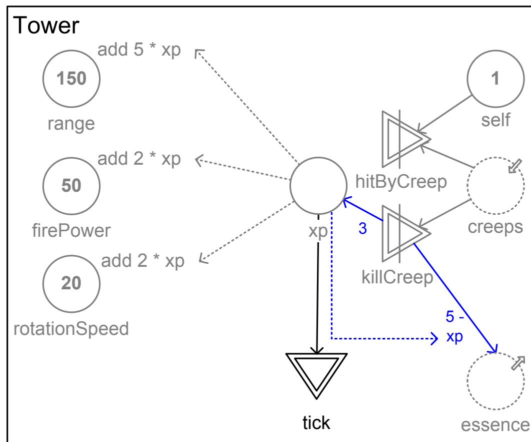


Figure 11: Third Tower Visual MM Definition

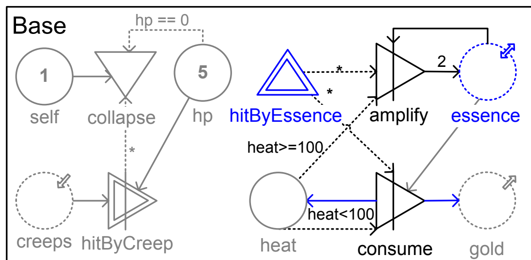


Figure 12: Third Base Visual MM Definition

Gameplay Design. To make positioning bases more interesting we introduce an aging mechanism. A base ages for every essence it converts into gold. Once a base has produced 100 gold it evolves into a new mode, duplicating essence instead. Because essence falls down, it is more effective to place newer bases below older bases, and this creates a high-risk reward strategy that encourages players to build their first bases in relatively exposed positions.

Mechanics Modeling. Figure 12 shows the third Base definition. We delete drain *tick*, reversing the decision of draining heat over time. We change *hitByEssence* into a drain and reference *essence* gains an output modifier. We add converters *amplify* and *consume* and triggers to activate them from *hitByEssence*. Consume returns one gold and one heat for each essence when heat is less than 100, but otherwise amplify instead returns two essence resources.

Design Iteration 6: Towers are upgradeable

Play Test. We see the problem that all towers act alike.

Gameplay Design. We try to overcome this by allowing players to chose different possible upgrades for their towers, for specializing individual towers in different ways.

Mechanics Modeling. The final tower model, shown in Figure 13, adds a new pool called *soulReap*, and increases the amount of essence for killing a creep to *5+soulReap*. Additionally, users can upgrade *range*, *firePower* and *rotationSpeed* and *soulReap* by respectively activating converters *upgradeRange*, *upgradePower*, *upgradeSpeed* and *upgradeSoulReap* that require more *xp* each upgrade.

Domain Modeling. Previous game adaptations only involved mechanics modeling, but choosing different tower upgrades is a feature not previously agreed upon. The MM Lib lets games observe type changes, enabling us to metaprogram the handy default of adding new interactive nodes

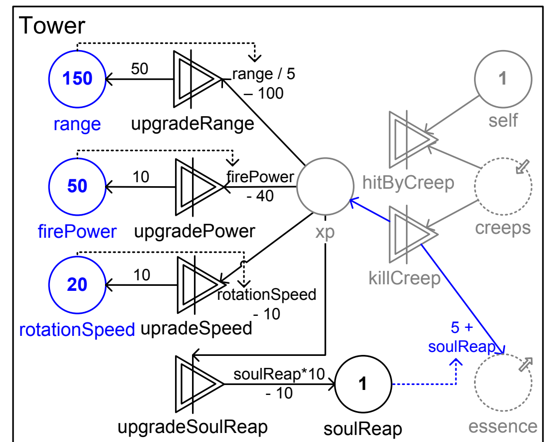


Figure 13: Fourth Tower Visual MM Definition

name	meaning	embedding
upgradeRange	upgrade range	activate node
upgradePower	upgrade power	activate node
upgradeSpeed	upgrade rotation speed	activate node
upgradeSoulReap	upgrade essence efficiency	activate node

Table 4: Additional Name Bindings for the Tower Definition

as clickable names near UI elements. In this case the game adds different upgrades near towers, and infers the name bindings of Table 4. This way, designers can continue mechanics modeling and play testing.

4.3 Experimental results

We showed that after the AdapTower prototype was built, its game economy could be modified using MM, with a flexibility reminiscent of the concept phase. Because we expect continuously high activity levels of mechanics modeling and play testing, as shown in Table 1, the approach is especially useful in later project stages. It allowed us to iterate and explore the design quickly, providing us with time to experiment and improve the game. Each iteration took us about 15 minutes to design gameplay and to model additions and changes of the mechanics in textual MM. We play tested briefly, and confirmed that we achieved the emergence we were looking for. Additionally, we spent only a fraction of the implementation time on the name bindings that integrate the model.

5. CONCLUSIONS

We proposed a game design approach for adapting game mechanics with MM. We applied the approach and implemented AdapTower, a prototype tower defense game. We demonstrated that its mechanics and emergent gameplay could be modified after its construction. We showed that MM is a suitable notation for making adaptations.

We argued that structured and clear artifacts and better, faster feedback about gameplay changes improve designer productivity. Our approach showed that it is feasible to significantly reduce design iteration times and accelerate the game development process by improving the adaptability, thereby increasing opportunities for quality improvements and reuse. We see our contributions as an important step towards practical game design methods for producing game software. However, our experience concerns a prototype and the technology still needs to mature before industry can adopt this approach. Future work entails the following:

- *Game design tools* can embed MM Lib and show editable visual MM, and send textual MM changes to adapt games like we proposed, e.g. graphical debuggers for tracing and replaying behavior, and integrated analysis tools like MM AiR. We envision a tool-of-tools approach, using language work-benches and meta-programming, for tailoring tools towards games.
- *Validation* requires that we apply tools and libraries in industrial case studies. Such experiences provide valuable information on the usage of language and tool features, and for making improvements.
- Comparable, reusable, analyzable game designs enable *pattern mining*. To do this, we first require a corpus of MM models to find empirical evidence of patterns, and an assessment of their gameplay qualities. Then, we can extract and compare them, e.g. for providing tools for mechanics comparison and prototype generation. Using extracted domain knowledge can also augment evolutionary techniques for mining patterns [7].
- MM do not necessarily describe the full run-time behavior of a game system, limiting analyzing it. Productivity and quality can be further improved by integrating MM with other DSLs, e.g. for locations and speed, and story-lines. Additionally, more accurate behavior predictions enable preventing bugs.
- Combining player satisfaction information with adaptable mechanics enables tailoring mechanics to the skill and enjoyment of individual players.

Acknowledgements.

We thank Paul Klint for proof reading this paper, and the anonymous reviewers for their insightful comments.

6. REFERENCES

[1] E. Adams and J. Dormans. *Game Mechanics: Advanced Game Design*. New Riders Publishing, 2012.

[2] E. Adams and A. Rollings. *Fundamentals of Game Design*. Pearson Education, Inc., 2007.

[3] M. Araújo and L. Roque. Modeling Games with Petri Nets. In *Proceedings of the 3rd annual DiGRA conference Breaking New Ground: Innovation in Games, Play, Practice and Theory*, 2009.

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

[5] C. Brom and A. Abonyi. Petri Nets for Game Plot. In *Proceedings of Artificial Intelligence and the Simulation of Behaviour (AISB)*, 2006.

[6] D. Church. Formal Abstract Design Tools. *Game Developer*, San Francisco, CA: CMP Media, 1999. http://www.gamasutra.com/view/feature/3357/.

[7] M. Cook, S. Colton, A. Raad, and J. Gow. Mechanic Miner: Reflection-Driven Game Mechanic Discovery and Level Design. In A. Esparcia-Alcázar, editor, *Applications of Evolutionary Computation*, volume 7835 of *Lecture Notes in Computer Science*, pages 284–293. Springer Berlin Heidelberg, 2013.

[8] B. Dawson. Game Scripting in Python. Game Developers Conference, 2002. http://www.gamasutra.com/view/feature/2963/.

[9] J. Dormans. Machinations: Elemental Feedback Patterns for Game Design. In J. Saur and M. Loper, editors, *GAME-ON-NA 2009: 5th International North American Conference on Intelligent Games and Simulation*, pages 33–40, 2009.

[10] J. Dormans. *Engineering Emergence: Applied Theory for Game Design*. PhD thesis, University of Amsterdam, 2012.

[11] Epic Games Unreal Developer Network. *Unreal Kismet User Guide*. http://udn.epicgames.com/Three/KismetUserGuide.html.

[12] Epic Games Unreal Developer Network. *UnrealScript Language Reference*. http://udn.epicgames.com/Three/UnrealScriptReference.html.

[13] D. Fu, R. Houlette, and J. Ludwig. An AI Modeling Tool for Designers and Developers. In *IEEE Aerospace Conference Proceedings*, Big Sky, Montana, 2007.

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

[15] M. Klotzbuecher and H. Bruyninckx. A Lightweight, Composable Metamodelling Language for Specification and Validation of Internal Domain Specific Languages. In R. Machado, R. Maciel, J. Rubin, and G. Botterweck, editors, *Model-Based Methodologies for Pervasive and Embedded Software*, volume 7706 of *Lecture Notes in Computer Science*, pages 58–68. Springer Berlin Heidelberg, 2013.

[16] R. Koster. A Grammar for Gameplay. http://www.raphkoster.com/gaming/atof/grammarofgameplay.pdf, 2005.

[17] M. McNaughton, M. Cutumisu, D. Szafron, J. Schaeffer, J. Redford, and D. Parker. ScriptEase: Generative Design Patterns for Computer Role-Playing Games. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 88–99, Washington, DC, USA, 2004. IEEE Computer Society.

[18] K. Neil. Game Design Tools: Time to Evaluate. In *Proceedings of the DiGRA Nordic Conference*, 2012.

[19] 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.

[20] K. Salen and E. Zimmerman. *Rules of Play: Game Design Fundamentals*. The MIT Press, 2003.

[21] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[22] B. H. Wasty, A. Semmo, M. Appeltauer, B. Steinert, and R. Hirschfeld. ContextLua: Dynamic Behavioral Variations in Computer Games. In *Proceedings of the 2nd International Workshop on Context-Oriented Programming*, COP '10, pages 5:1–5:6. ACM, 2010.

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