A Visual Modelling Tool for Micro-Machinations

Ferdy van den Hoed
f.k.vdhoed@gmail.com

Abstract
Cost of producing games is an increasing problem and industry wide accepted tools are missing (Neil, 2012). The young game industry could also benefit from the streamlined processes other branches of development uses. Quality of game design requires many iterations and because designers and programmers communicate differently they need to reiterate implementations. This report builds on the progress Machinations and Micro-Machinations achieved. Machinations is an existing framework for designing game economies that includes a user-friendly tool that facilitates paper prototyping. Micro-Machinations is a programming language formalizing Machinations’ meaning that can also be used for software prototyping. This report presents the Micro-Machinations Visual Modelling Tool which combines the best of both worlds, bringing the advantages of Micro-Machinations to game design.

1 Introduction
Technology improvements from recent years greatly improved the processes of software development. Unfortunately, game design processes are lacking in comparison and make it more difficult and complex to deliver high quality in this aspect. Quality in game design is a time costly process because of the long design, evaluation and iteration cycles. These cycles could be shortened if designers had early and understandable feedback about the inner workings of their game’s economy. They could also greatly benefit from the live programming technique. With this technique the feedback loop between modelling code and the observable behaviour is shortened. (Lieberman & Fry, 1995) Designers lack a common vocabulary for gameplay and they are also missing tools to explore design alternatives more easily. There has been progress in these aspects, but an industry standard has not risen yet. The gap between design and implementation also contributes to the long development cycles. The communication between programmers and designers is not satisfactory and both sides lose valuable time when programmers implement adjustments best understood by designers. The lack of formal models makes Procedural Content Generation (PCG) and Automated Game Design (AGD) methods also less feasible because these systems need to be created by programmers.

Machinations (Adams & Dormans, 2012) is a visual language that is a semi-formal model for expressing the inner workings of a games internal economy. It comes with an interactive tool for designers that support the designs and balance in the early stages of development. Micro-Machinations (Klint & van Rozen, 2013) is a continuation of Machinations that aims to formalize its model. It uses the live programming technique and comes with an API that allows programmers to manage game economies. This paper presents a prototype that takes the best of both to create a visual modelling tool from Micro-Machinations like the one Machinations uses. This will allow designers to create and balance the economy and elevate work from programmers since Micro-Machinations can be used as software artefact.

The structure of this reports is as follows. Section 2 describes the Machinations framework, what it is and what it is for and how to use it. Section 3 describes Micro-Machinations and the improvements
that it made to the Machinations framework using an example. Section 4 describes the new system and demonstrates the design goals and implementation of the Visual Modelling Tool for Micro-Machinations. Section 5 presents a comparison of the different systems and will describe the direction for future work. Section 6 conclude this report.

2 Machinations

2.1 The Machinations Framework

Machinations is a visual language to express the flow of resources in a games economy. It does this by using diagrams with nodes and edges (connections). It helps designers design, understand and balance. “It is based on the theoretical notion that structural features of game mechanics are for a large part responsible for the dynamic gameplay of the game as a whole.” (Dormans, Simulating Mechanics to Study Emergence in Games, 2011) Because of the visual nature, Machinations can help identify important structural features that create emergent gameplay. One of those structures are feedback loops which the framework put on the foreground. (Dormans, 2012) argues that by using MDE techniques, the efficiency of development could increase and raise the overall quality.

Testing and validating the mechanics created by the designers is very important for game development. Machinations comes with a digital tool that makes the diagrams interactable, so they can be tested and iterated upon (Adams & Dormans, 2012). Although the diagrams are dynamic and interactable they are design artefacts only and are not suitable as software requirement.

The Machinations framework also comes with a pattern library for recurring structures. These are used to capture objective standards of quality in architecture (Adams & Dormans, 2012) (Dormans, 2012).

2.2 Diagram Elements Condensed

Machinations uses nodes and edges to express the internal game economy. These elements dictate the rules how resources flow. (Klint & van Rozen, 2013) describe the most important elements of the language in the table below.\(^1\)

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty pool</td>
<td>Pool with resource</td>
</tr>
<tr>
<td>Flow edge flow of rate 1</td>
<td>Flow edge unlimited flow rate</td>
</tr>
<tr>
<td>Passive pool</td>
<td>Interactive pool</td>
</tr>
</tbody>
</table>

\(^1\) For consistency we have added hyper-edges, gates and group boxes in this table as well. For the original table at (Klint & van Rozen, 2013)

\(^2\) For conciseness we only give an informal description here, closely adhering to (Adams & Dormans, 2012)
Automatic (*) nodes act automatically, once every step. Start (s) nodes are active in the initial state but become passive afterwards.

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 (&) interpret 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.

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 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 targets of their triggers.

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.

Hyper-edges connect an origin node to a target label of an edge. A label modifier indicates how state changes in the origin node modify the current value of the target label at a current time step as indicated by the state connection’s own label. The new value takes effect in the next time step.

Gates appear as diamond shapes and often have multiple outputs. Instead of a flow rate, each output is labelled with a probability or a condition. The first type of outputs are referred to as probable outputs while the others are referred to as conditional outputs. All outputs of a single gate must be of the same type: When one output is probable, all must be probable, and when one output is conditional, all must be conditional.

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 can only pull, and the any modifier does not apply. If specified, the unit kinds on the inputs and outputs must match the converter’s unit kinds.

Group Box inserts a resizeable dotted-line box in the diagram for illustration purposes. Has no effect on the simulation.

Table 2.1: Elements of Machinations condensed

2.3 Example

Let’s say we want to make an action RPG. The player needs to be able to kill enemies and the enemies need to spawn in to the world. Figure 2.1 show how you could model this with
Machinations. Enemies spawn automatically from the “Spawn” source and the player is able to kill it via the user interactable “Kill” drain. As you can see the kill is not guaranteed since it has a probable flow connection from “Enemies” to “Kill”.

Figure 2.1: Basic player enemy interaction. (pools, flow edges, drains, sources and group boxes)

Usually in an RPG the player has some form of progression by means of experience points, see figure 2.2. We have replaced the “Kill” drain with a converter since we want to convert the killed enemies in experience points and added the pool “Exp” to store them. The more experience a player has the easier it becomes to kill enemies. This is expressed in the diagram via the hyper-edge from “Exp” to the flow edge between “Enemies” and “Kill”.

Figure 2.2: Player progression. (converters and hyper-edges)

This RPG lacks the role-playing aspect, so let’s change the means by which we kill enemies and give the player a bow as weapon. We have added the pool “Arrows” to store our ammo and a user interactable drain “Shoot” see figure 2.3. We have changed activation modifier of “Kill” to passive and added a trigger edge from “Shoot” to “Kill”. Now when the player activates “Shoot” it will automatically activate the “Kill”. We could not add a flow edge between “Arrow” and “Kill” directly because the converter would convert an arrow to experience even if the probable edge between “Enemy” and “Kill” would not let a resource flow.
Having a different way to attack enemies is nice but since arrows are consumable there needs to be a way to gain them. We have added a new pool “Arrows” on the top right side of the screen to store arrows dropped by enemies see figure 2.4. There already exist a pool with this name for the player, this is not a problem for Machinations but it is confusing to humans. Therefore, we have added a group box “Player” and “World”. Group boxes have no effect on the simulation the diagrams become more easily readable for designers. To make the enemy drop loot we have added a flow edge between “Kill” and “Arrows” inside the world group box. The label of this edge is a multiplier and tries 10 times to add a resource to its target node with a 20% change of success. We also changed the activation modifier of “Arrows” inside the player group box from passive to user and added a flow edge from “Arrows” inside world to “Arrows” inside player so the player will pick up all the arrows laying around the world when he activates his “Arrow” pool.

Regaining arrows was a necessary step to prolong the game but unfortunately the diagram could still end up in a deadlock. When the player has shot all his arrows there is no way to get more since the only way they spawn is by killing enemies with arrows. In figure 2.5 you can see the added “Gold” pools both for the player and the world. There is now a flow edge between the worlds “Spawn” and “Gold” so gold spawns automatically in the world. Like with arrows we added a flow edge between both the “Gold” pools. We also added “Buy” so the player can convert gold into arrows.
Now that we have introduced the gold resource let's make the enemy drop it as well see figure 2.6. We have removed the flow edge between “Kill” and the worlds “Arrows” and added the gate “Loot” and the flow connection between “Kill” and “Loot”. We have added two nameless sources and a trigger edge between them and “Loot” with different probabilities. When an enemy is killed the gate will have a 60% chance activate one of those sources and spawn loot.

Figure 2.7 shows the complete diagram of the game. There are lots of ways to expand this diagram, add a different skill to kill an enemy or a perk to modify the kill change or loot drops. We could add different types of enemies, add a health system or even add multiple players.
2.4 Benefits and Limitations

Diagrams are interactable and can be used as an early prototype, so designers can test if their design results in the intended emergent gameplay. Because of the digital nature of the tool, diagrams are easily iterated upon. It is not uncommon to first create an abstract diagram and add more detailed with every iteration. Furthermore, the tool can run diagrams automatically and they are even options to run simulated players. The option to include charts gives tangible data which is essential for making informed decisions. All this combined allows flaws in the design to surface early and helps reduce development cycles.

The Machinations framework uses models, this forces the designers to think about their game at an abstract level. The visual nature of the framework helps designers to see the structural features and more importantly it shows how these features interconnect. This can help designers to identify what features are affected by changing a part of the diagram. The framework comes with a pattern library including common structures for game design. The use of patterns helps designers to understand and solve design problems. These patterns can be implemented in several ways, so they are not rigid and do not hinder the creative process.

In Machinations, the diagrams cannot be changed during runtime. This forces the designer to stop the current run and lose the progression that was made. Also, every part of the diagram needs to be modelled in the specific elements that make up these structures. If we would add a second player to the diagram shown in figure 2.7 we would need to duplicate all the elements inside the “Player” group box and reconnect them with those contained inside the “World”. It is great to see how the entire diagram connected, but this results in very large diagrams and a repetition of work. It is also not very maintainable for a change in player features results in a change to all “Player” group boxes.

The tool is a standalone application and there are no hooks into it, so it cannot be used as software requirement. Because the edges contain resources and their length can differ it is possible that resources are distributed out of sync, this makes the results less reliable. Furthermore, there are a couple of ambiguous elements like hyper edges. It is essential to see how edges influence the distribution of resources, but there is no definitive way to interpret in which order these are executed. This results that it is difficult to analyse diagrams, not only for designers but also for software. This makes the usage of mixed initiative techniques described by (Dormans, 2012) less feasible.
In conclusion, the framework and the tool give designers a lot of insight on how the mechanics interact but for the tool to reach industry standards more development is needed (Neil, 2012).

3 Micro-Machinations

3.1 The Micro-Machinations DSL

“Micro-Machinations (MM) is an evolutionary continuation of Machinations aiming at software prototyping and validation. MM is a formalized extended subset of Machinations, that brings a level of precision (and reduction of non-determinism) to the elements of the design notation that enables not only simulation but also formal analysis of game design.” (Klint & van Rozen, 2013)

Visual languages are a great tool for humans to see the interworking and connections of complex systems. Unfortunately, they are not easily parsed and analysed by computers, that is why Micro-Machinations represents the model in a textual format. The Micro-Machinations DSL was created to solve this problem. “Micro-Machinations is intended as embedded scripting language for game engines that enables interaction between the economic rules and the so-called in-game entities that are characterized by one or more atomic resources.” (Klint & van Rozen, 2013) This makes the diagrams created in Micro-Machinations more valuable to the entire development cycle of games than those created in Machinations.

The textual representation is easily parsed and allows the diagrams to be analysed, which increases the chance of design flaws to surface even earlier in the design process. The method that is used to analyse Micro-Machinations is described in (Klint & van Rozen, 2013) (van Rozen & van der Storm, Augustus 2017). Because Micro-Machinations can be parsed it is able to resolve conflict and therefore can assure that modelling actions keep the diagrams valid.

This allows Micro-Machinations to use the live programming technique and enables users to modify the model during runtime. Allowing designers to make changes on the fly and get immediate feedback reduces iteration cycles (Klint & van Rozen, 2013) (van Rozen & Dormans, 2014).

3.2 Micro-Machinations Additional Elements

Micro-Machinations uses most of the elements of Machinations described in section 2.2. Probable gates are not yet implemented, it removes group boxes and it replaces edges and hyper-edges with expressions. It also introduces new elements to the existing selection that help with modularity. These help to encapsulate elements that together form entities like players. Changes to such an entity must be done once, this eliminates repetitive work. A beneficial side effect is that they help with scalability, diagrams are less cluttered with duplicate entities. (Klint & van Rozen, 2013) introduced the newly added elements. A modified version can be seen in table 3.1.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited</td>
<td>Limited capacity</td>
</tr>
<tr>
<td>capacity</td>
<td>Full pool</td>
</tr>
<tr>
<td>Pools</td>
<td>Pools may specify a maximum capacity for resources, which can never be exceeded, that is visually a prefix max followed by an integer.</td>
</tr>
<tr>
<td>Definition</td>
<td>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. There is no visual representation for this element.</td>
</tr>
</tbody>
</table>
A type pool is a pool that contains all the elements of its type definition. This makes definitions a resource that can be stored inside a pool. The behavior of the internal elements of every resource of a type pool are the same, but every resource in a type pool is its own named diagram with its individual elements.

A reference, represented by a circle with a dashed line, is an alias that refers to a node that is defined externally. 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 points into the node, an output modifier points out of the node, and an in-/output modifier does both.

An interface makes internal elements of a type pool available to the outside, and can be used by connecting resource connections. Visually, an interface is a small circle at the border of a typed pool 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 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.

Table 3.1: Micro-Machinations newly introduces elements.

3.3 Example

Let’s look again at our game described at section 2.3 and make it a multiplayer game. This means we need to separate the “Player” group box from the “World” group box. Figure 3.1 (a) shows the “Player” definition. Note that we have greyed out the edges that were running through multiple group boxes, we will reconnect those later.

Figure 3.1 (b) shows the entities of the world but the name of the definition is called “Game”. In Machinations there was only one diagram for all entities, but Micro-Machinations separates them. The “Game” definition is like that one diagram of Machinations and is the starting point of our Micro-Machinations diagram. Generally, here we store the elements that belong to the game world. We have added a pool “Players” with the pool type of the “Player” definition that will represent our players.

---

4 Note that although Micro-Machination has a visual representation of elements the diagrams are expressed in textual form only.
players. Although the “Player” definition contains multiple elements it can still be seen as a resource which can be stored inside a pool.

Now that we separated our entities we need to reconnect them. Micro-Machinations introduced references, we use these nodes to represent pools of other definitions. We have added the “Gold”, “Arrows” and “Enemies” references to the “Player” definition, these will represent the pool defined in the “Game” definition. We have also added a new reference called “Loot”. Micro-Machinations does not allow other elements than pools to be represented externally so we need this pool to transfer resources from the “Player” definition to the “Game” definition.

![Image](https://via.placeholder.com/150)

Figure 3.2: Reconnection the player to the world. (references, interfaces and bindings)

Because we have added reference nodes to the “Player” definition the “Players” pool of the “World” definition show the corresponding interface nodes see figure 3.2 (b). We use these interfaces in combination with the binding edges to reconnect the player with the world. We have also added the pool “Loot” with the automatic activation type and connected it with the “Loot” gate. When a player shoots and kills an enemy, the resource will flow from “Kill” inside the “Player” definition to the “Loot” pool inside the “Game” definition. Then the resources inside the “Loot” pool of the “World” definition will automatically be pushed to the “Loot” gate and loot will be generated. When the player activates the “Gold” or “Arrow” pool inside the “Player” definition, the resources contained inside the corresponding “Game” pools will flow from the “Game” to the “Player” definition.

Now that we have reconnected the player and the world we can add multiple players by adding resources to the “Players” pool inside the “World”. These players will share the resources inside “Gold”, “Arrows”, “Loot” and “Enemies” defined in the “World” definition but each player will have his own resources defined in the “Player” definition.

### 3.4 Benefits and Limitations

Micro-Machinations made great steps in formalizing the Machinations language and the current C# API allows programmers to manage game economies with it. One of the greatest achievements is that it is possible to make changes in the design during runtime which can shorten the iteration cycles. Unfortunately, the DSL still has drawbacks. There is no visual interface like the Machinations tool for designers to use and programmers still need to implement the design.

Encapsulation hides elements in the diagrams. This makes diagrams more readable but does not show the complete interaction between mechanics. The replacement of edges and hyper-edges for

---

5 Micro-Machinations replaces hyper-edges with expressions but they can still be visually expressed.
expressions is another drawback. This removes the complete overview of feedback loops which are an important element in Machinations.

4 The Micro-Machinations Visual Modelling Tool

4.1 Design goals
The Micro-Machinations Visual Modelling Tool (MMVMT) should approach the functionality that the Machinations Tool had. Furthermore, it should incorporate the key features that separate the Micro-Machination framework of that of its predecessor.

The current implementation will focus on; visuals, scalability and feedback to the user.

Visuals: A tool is as useful as its ability to be used by its target audience. It is important that the tool can be used and understood by designers. Therefore it is essential that the new tool will express itself visually.

Scalability: Because multiple instances of game entities need to be expressed in all the elements that entity is composed of Machinations diagrams can easily blow out of proportion and make them difficult to use and understand. Furthermore, the Machinations diagrams can only function as design artefacts and the need to maintain large and complex diagrams is not needed. The new tool should improve readability and be able to handle larger and more complex diagrams.

Feedback: For designers to improve their productivity it is essential they understand what the meaning is of what they create and how they use the tool. Therefore it is necessary that the tool can give feedback to the designers about the systems they create and where possible assist and advise.

4.2 Feature design
Model Driven Engineering
Because it is possible to change the model while the diagrams are interactable it is important that the new tool clearly distinct model- and diagram interactions.

To encourage Model Driven Engineering and avoid confusion the new tool will have two windows, one dedicated for the model actions (2) and the other one for interaction with the diagram (3) see: figure 4.1.
(a) **Current implementation**

Figure 4.1: MM VMT separate windows, left toolbar, middle model window, right state window.
(1) Tool window, (1.1) Model-tab, (1.1.1) Graph selector, (1.1.2) Node/Edge Inspector, (2) Model window, (2.1) Model grid, (2.2.1) Model tab bar, (2.2.2) Add definition button, (3) State window, (3.1) State grid, (3.2) State tab bar, (3.3) State error window.

Model actions:
The tool enables the designer to place, move and remove all the nodes and edges on the Model grid figure 4.1 (2.1) supported by Micro-Machinations. The tool window holds the Graph selector and here the designer can select what element to place. When a node or edge is selected, with the

(b) **Mock up**

*Figure 4.1: MM VMT separate windows, left toolbar, middle model window, right state window.*
(1) Tool window, (1.1) Model-tab, (1.1.1) Graph selector, (1.1.2) Node/Edge Inspector, (2) Model window, (2.1) Model grid, (2.2.1) Model tab bar, (2.2.2) Add definition button, (3) Model error window, (3) State window, (3.1) State grid, (3.2) State tab bar, (3.3) State error window.
mouse icon, the inspector will show the properties of that element and can be modified figure 4.1 (1.1.2).

State actions:
In the State grid designers can interact with nodes that have the type user as when property. The designer is also allowed to move elements.

Encapsulation
In Machinations diagrams quickly grow to large and got clutter even when they stayed fairly simple, encapsulation has the same risk in when the display of large number of Definitions and Instances are handled poorly. Encapsulation is one of the focus points of Micro-Machinations (Klint & van Rozen, 2013) and while this helps with scalability they have the risk to destroy the visual connection between elements.

Like other applications MMVMT will use a tab system to separate the different Definitions and Instances see figure 4.1 (2.2.2) and (3.2). To keep designers in their flow and not distract them or waste their time with unnecessary opened tabs the Definitions and Instanced will also be displayed in a list view for easy access see figure 4.2. These views can be accessed from within the Tool window.

![Figure 4.2: Tree views, left model, right state.](image)

Connections
Feedback loops where one of the structural features that the Machinations framework put on the foreground (Adams & Dormans, 2012), but the way edges and hyper-edges were implemented made the diagrams non-deterministic. Micro-Machinations replaced them with expressions which come with the benefit that they are very powerful and can be expressed visually like edges and hyper-edges. Unfortunately, they are more difficult to create and are harder to read. Therefore, the tool should guide the designer in creating these edges and make these connections visible to the designer by other means than expressions alone.

Feedback to the Designer
One of the advantages of digital games, in comparison to their physical counterparts, is their ability to assure the player does not take invalid actions. It is important that the tool communicates invalid actions the designer may take so he has a better understanding why his action was invalid and gains insights in to correct his mistakes. Pointing out flaws is a great gain, but it would be more constructive to have the tool make suggestions for possible resolutions.

The tool will do this by via the error window see figure 4.3. The Model and State window will both have a dedicated window, so it is clear if the error came from a model- or a state action. The errors
will also be clickable, and the tool will take the designer to the correct window and will highlight the elements in question.

Figure 4.3: Model Error window.

4.3 Benefits and Limitations
The current iteration of the MMVMT is in prototype phase, but this does not mean it is not functional. The prototype supports the placement, movement and remove functionality for all the elements described in section 2.2 and 2.3. The tool also supports changes in the properties of these elements that can result in a valid diagram. And maybe one the most important features of Micro-Machinations these changes can be made during runtime. At the moment the tab feature describe in the previous section is also implemented. It is also possible to save and load diagrams to the .mm file type which could be understood by people custom to read code.

Not only it is possible to model diagrams with the tool they are also interactable, the resource connections distribute the resources accordingly and the values of the pools show the correct values. When the designer makes changes to the model of an instance each instance is updated. The tab system is also implemented for the instances.

Although the prototype is functional some usability features are missing. Currently the Error system, tree view and expression wizard described in the last section are not yet implemented. The creation of hyper edges can be achieved by using the expression, but it is not possible to do this visually, also the only visual feedback is in expression form which means not all feedback loops are seen at a glance. Because the tool is an early prototype it did not have the luxury to go through a polish phase and the small but important UX features like; undo/redo, grid snap, grid resize, and label placement are missing.
5 Discussion

5.1 Machinations Tool vs Micro-Machinations Visual Modelling Tool

The aim of the MMVMT is to replace the function that the Machinations tool had and expand beyond. Right now, it is a standalone application but eventually we can incorporate it inside the Unity Engine to let it manage real game economies instead of simulating them like Machinations did. This is possible because Micro-Machinations has a general API which can already interface with other software by programmers.

One of the main goals of Micro-Machinations was the ability to shorten the feedback cycles by enabling changes during run-time. The MMVMT allows designers to do this. This is a great improvement in contrast with Machinations where the designer is forced to stop the tool, make changes and restart the diagram from its initial state.

Another goal was to reduce redundant work by providing encapsulation. In the MMVMT designers only need to model and maintain entities once. This is a great improvement on Machinations where all the elements of an entity needed to be modelled individually. Encapsulation makes Micro-Machinations diagrams less cluttered than their Machinations counterpart. This has the downside that a complete overview of the entire diagram is not available in Micro-Machinations.

Micro-Machinations replaced edges and hyper edges with expressions. Expressions are more powerful and less prone to ambiguous interpretation but come with the cost of being harder to read, write and understand. The current iteration of MMVMT does not guide the designer in this aspect and further development is needed. Right now, the MMVMT lacks the visual connection of hyper-edges.

Because changes during runtime are only possible if the program can validate the diagrams, Micro-Machinations can be analysed. This has several benefits for the designer. One of those is feedback the MMVMT can give the designer, this will give him a better understanding of the outcome of his actions. Currently the API gives feedback, but it is not shown to the designer. In a later stage of the development it is possible to take the analysis even further and give designers suggestions in how to improve their economy. There has been made an effort towards this aspect for Micro-Machinations. (van Rozen, 2015)

Micro-Machinations is completely event driven and records can be made of all the changes in a diagram. This provides even more data that can be analysed. This even allows designers to balance the diagrams, run it with the same events and see how they impact the economy directly. Another benefit of the event-based nature is that this allows the Micro-Machinations Tool to support multiple users.

Machinations supports charts, and this is its way to provide the designer with data. Unfortunately, because of the non-deterministic nature of Machinations the results are not that reliable. Charts are a feature missing of the MMVMT and should be kept in mind for future iterations.

The Machinations Tool allows the designer to add simple AI to simulate player behaviour. The MMVMT lacks such a feature at the moment. Micro-Machinations is open source and has an API, so programmers have a far greater control. Since the MMVMT aims to close the gap between programmer and designer the tool should make automatic runs with AI possible.

Machinations also supports a colour code mode. This made it possible to have one node that can hold multiple resource types. Micro-Machinations does not and will not support this feature. It loses some of the conveniences colour mode contributed but it makes the diagrams more readable.
Micro-Machinations did not inherit some elements from Machinations, namely; traders, registers, end conditions and delays. Traders and registers were created as single elements for the convenience for designers and can be created with other elements (Adams & Dormans, 2012). Micro-Machinations does not support the time modes Machinations did because it was a root cause of the ambiguity.

The Machinations tool is a product that has had multiple iterations and has a lot of small UX features. When the MMVMT comes out of the prototype phase the tool should benefit from the effort that Machinations had put into this aspect.

5.2 Future work
The next iteration of the MMVMT should focus on the development of the missing features described in section 4.2 and start to introduce UX features. This is needed so the tool can be used and tested by the target audience.

When the tool is out of the prototype phase, and can be used as valid prototype method, development can start on more ambitious features. One of those is the integration within Unity itself so that Micro-Machinations can be used to manage economies of games without the direct usage of the API and makes it useful to designers.

This may allow the tool to be widely used and create large amounts of data which later can be analysed. This data could be used to find commonly used patterns and enhance the pattern library. These patterns could be used by the tool to make suggestions such as provided by MeDeA (van Rozen, 2015). This will open possibilities for PCG or AGD suggested by (Dormans, 2011).

6 Conclusion
Micro-Machinations is a great evolutionary step from Machinations and helped diagrams to be more suitable for formal analysis of game designs. The API allowed programmers to manage economies and make changes during runtime which sorted iteration cycles (Klint & van Rozen, 2013). The current prototype of the MMVMT is a great first step that takes the best of Machinations visual tool and the Micro-Machinations API. Although further development is needed to make the MMVMT usable practically in the industry it shows great potential.

Acknowledgements
Thanks to Riemer van Rozen for answering my many questions about Micro-Machinations and for giving feedback and suggestions about my approach and implementation. I would like to thank Anders Bouwer for the support and suggestions for writing this paper. Thanks Firebrush studios and the RAAK MKB Project Live Game Design to provide the opportunity to work on this project.

References


