Simulating Mechanics to Study Emergence in Games

Joris Dormans
Amsterdam University of Applied Sciences
Duivendrechtsekade 36-38
1096 AH, Amsterdam, the Netherlands

Abstract

This paper presents the latest version of the Machinations framework. This framework uses diagrams to represent the flow of tangible and abstract resources through a game. This flow represents the mechanics that make up a game's interbal economy and has a large impact on the emergent gameplay of most simulation games, strategy games and board games. This paper shows how Machinations diagrams can be used simulate and balance games before they are built.

Games that display dynamic, emergent behavior are hard to design. One of the biggest challenges is that the mechanics of these games require a delicate balance. Developers must rely on frequent game testing to test for this balance. This requires much time, but also a functional prototype of the game. This paper looks into simulating games in early stages of development, before prototypes are built, in order to design dynamic but balanced game mechanics effectively and efficiently. To this end it utilizes the most recent version of the Machinations framework for representing game mechanics. The Machinations tool has been extended to simulate dynamic games and to collect data from a multitude of simulated play session. In the second section of the paper, the tool is put to the test by simulating and balancing the game *SimWar*, which has been described by Will Wright, but never was built.

The Machinations Framework

The *Machinations framework* formalizes a particular view on games as rule-based, dynamic systems. It focuses on game mechanics and the relation of these mechanics and the dynamic gameplay that emerges from them. 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. Game mechanics and their structural features are not immediately visible in most games. Some mechanics might be close to a game's surface, but many are obscured by the game's system. Only a detailed study of a game's mechanics can shed a light on the game's structure. Unfortunately, the models that are used to represent game mechanics, such as representations in code, finite state diagrams or Petri nets, are complex and not really accessible for non-programmers. What is more, these are ill-suited to represent games at a sufficient level of abstraction, on which structural features, such as feedback loops, become immediately apparent. To this end, the Machinations framework includes a diagrammatic language: *Machinations diagrams* which are designed to represent game mechanics in a way that is accessible, yet retains the same structural features and dynamic behavior of the game it represents. Earlier versions of the framework were presented elsewhere (Dormans 2008; 2009). The version presented here uses the same core elements as the previous version, although there have been minor changes and improvements. The main difference is a change in the resource connections which now can have only one label, where it could have two before (see below). In addition, support for artificial players, charts representing game state over time and collecting data from many simulated sessions are new additions.

The theoretical vision that drives the Machinations framework is that gameplay is ultimately determined by the flow of tangible and abstract resources through the game system. Machinations diagrams represent these flows and foreground the feedback structures that might exist within the game system. It is these feedback structures that for a large part determine the dynamic behavior of games (Hunicke, LeBlanc, and Zubek 2004; Salen and Zimmerman 2004). This is consistent with findings in the science of complexity that studies dynamic and emergent behavior in a wide variety of complex systems (Wolfram 2002; Fromm 2005).

Machinations diagrams have an exact and consistent syntax. This means that the diagrams can be interpreted by a computer, in fact, the Machinations software tool implements diagrams. In other words, they are interactive and dynamic, just like the games they are modeling, and can be executed in a similar way. It allows dynamic models of games to be designed and tested quickly and efficiently. The tool also allows the designer to quickly gather quantitative data from simulated play sessions. Unfortunately this property of the software tool does not translate to paper; the interactive tool, and the examples discussed in this paper, can be found on the Machinations web page: `http://www.jorisdormans.nl/machinations`.

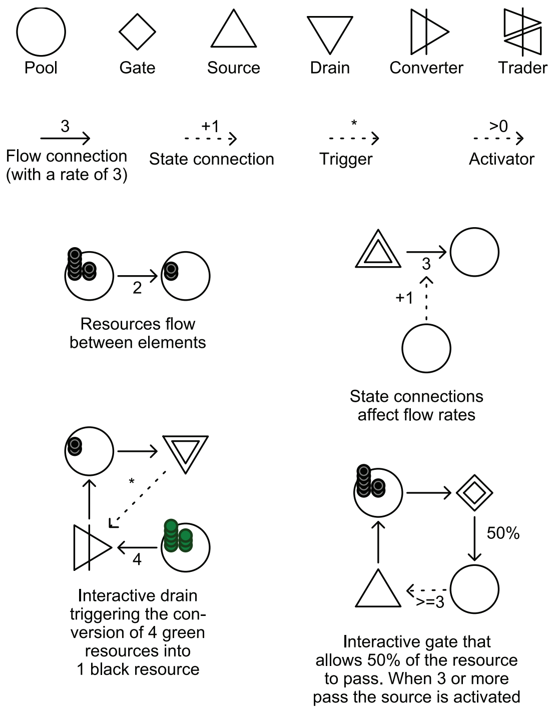
Pool
Gate
Source
Drain
Converter
Trader
3
Flow connection (with a rate of 3)
+1
State connection
*
Trigger
>0
Activator
2
Resources flow between elements
3
+1
State connections affect flow rates
*
4
Interactive drain triggering the conversion of 4 green resources into 1 black resource
50%
>=3
Interactive gate that allows 50% of the resource to pass. When 3 or more pass the source is activated

Figure 1: Elements of Machinations diagrams and example constructions.

Resource Flow

The Machinations framework utilizes the idea of 'internal economy' (Adams and Rollings 2007) to model activity, interaction and communication between game parts within the game system. A game's economic system is dominated by the flow of resources. In games resources can be anything: from money and property in *Monopoly*, via ammo and health in a first person shooter game, to experience points and equipment in a role-playing game. Even more abstract aspects of games, such as player skill level and strategic position can be modeled through the use of resources. A game's internal economy consists of these resources as well as the entities or actions that cause them to be produced, consumed and exchanged. In the case of *Risk*, territories, armies and cards are the main resources, the players' option to build will produce more armies, while with an attack the player risks armies to gain territories and cards. This makes building and attacking important actions that affect the economic state of the game.

Figure 1 contains the basic elements found in Machinations diagrams and some example constructions. The flow of resources is dictated by *resource connections*. These are represented as arrows that connect different elements in a diagram. Resource connections can have a label that determines the flow rate. Many labels are numbers, but the Machinations framework also uses simple expressions or icons to represent flow rates based on randomness, skill or other uncertain factors outside the game mechanics. When the label is omitted the flow rate of a resource connection is considered to be one.

The most basic elements connected by resource connections are *pools*. These are represented as an open circle. A pools collects resources that flow into it. Pools can be used to represent any collection of resources in a game. The resources themselves are represented as small circles. Different colors can be used to denote different types of resources. When there are more than twenty-five resources on a single pool, the resources are represented as a single number indicating their total.

Machinations diagrams are close to Petri nets. In a Machinations diagram resources can move from element to element, just like tokens in a Petri net move between places. The state of a Machinations diagram is determined by the distribution of resources over the entire diagram. This also means that Machinations diagrams are time-based. Resources can actually flow from one element to another over time. The time step at which the diagrams operate is arbitrary. In the online implementation the time step defaults to one second, which means that elements are checked and activated once every second. This value can easily adjusted.

Elements in a Machinations diagram can be *interactive*. Interactive elements are depicted with a double outline and represent player actions. In the online version of the Machinations tool a user can click an interactive element to *fire* it. A firing pool, for example, will try to pull resources through its inputs. The number of resources pulled is determined by the rate of the individual input resource connection.

Communication of State

An important difference between Machinations diagrams and Petri nets is that in a Machinations diagram the distribution of resources represent the current state of a game, and this distribution can alter the subsequent flow or (de)activate nodes. These relations are indicated by *state connections*, which are represented as dotted arrows (see figure 1) with labels. State connections indicate how the state of a pool, the number of resources on it, affects other elements in the diagram.

There are two special instances of state connections: triggers and activators. *Triggers* are state connections with a star ('*') as its label. An element that has an outgoing trigger, will activate the elements that trigger leads to when it is *satisfied*: when it received the number of resources along all of its inputs as indicated by their respective flow rates.

An *activator* is a state connection that has a condition as its label. This condition can be written down as a simple expression (for example '==0', '<3', '>=4' or '!=2') or a range of values (for example '3-6') if the state of the elements where the activator originates meets this condition then the element where the activator ends can fire, otherwise it is inhibited.

Controlling Resource Flow

Pools are not the only possible elements in a Machinations diagram. *Gates* are another type of element. In contrast to a pool a gate does not collect resources, instead it immediately redistributes them. Gates are represented as diamond shapes and can have multiple outputs. Instead of a flow rate each output will either have a probability or a condition specified by the output's label. The first type of outputs are referred to as *probable outputs* while the other are referred to as *conditional outputs*. All outputs of a single gate must be of the same type: when one output is probable, all are considered to be probable and when one output is conditional, all are considered to be conditional.

Probabilities can be represented as percentages (for example '20%') or weights indicated by single numbers (for example '1' or '3'). In the first case a resource flowing into a gate will have a probability equal to the percentage indicated by each output, the sum of these probabilities should not add up to more than 100 percent. If the total is less than 100 percent there is a chance that the resource will not be sent along any output and is destroyed. In the latter case the chance that a resource will flow through a particular output is equal to the weight of that output divided by the sum of the weights of all outputs of the gate.

An output is conditional when its label represents an expression (such as '>3' or '=0' or '3-5'), then all conditions are checked every time a resource arrives at the gate and one resource is sent along every output which condition is met. This might lead to duplication of resources, or, when no condition is met, to the destruction of the resource.

Four Economic Functions

In their discussion of a game's internal economy represented by the flow of resources, Adams and Rollings identify four basic economic functions: sources, drains, converters and traders (Adams and Rollings 2007). Sources create resources, drains destroy resources. Converters replace one type of resource for another, while traders allow the exchange of resources between players or game elements. In theory, a pool or combination of pools and gates can fulfill all these functions, but for clarity it makes sense to introduce special elements to represent sources, drains, converters and traders.

Sources are elements that produce resources. In *Risk*, the building action is a source: it produces armies. Likewise passing 'Go' in *Monopoly* also is a source: it generates money. Some sources are automatic, while other sources need to be activated, either by the actions of the player or some other occurrence in the game. The rate at which a source produces resources is a fundamental property of a source. Adams and Rollings distinguish between 'limited' and 'unlimited' sources (Adams and Rollings 2007). A limited source can be easily represented by a pool without inputs that starts with a number of resource on it. To represent unlimited sources, the Machinations framework includes a special source element represented as a triangle pointing upwards.

Drains are elements that consume resources. In an adventure game where you can cross hot lava at the cost of loss of health points, the lava acts as a *drain*. Like sources, drains can be automatic, activated by player actions or other events in the game. And, they have different types of rates too: some drains consume resources at a steady rate while others consume resources at random rates or intervals. The Machinations framework includes a special drain element represented as a triangle pointing downwards.

Converters convert one resource into another. In a shooter game, killing enemies might invoke a converter. In this case ammunition is used in an attempt to kill, which in turn, when the enemy is put down, might be converted in new ammunition or health packs dropped by the enemy. Converters act exactly as a drain that triggers a source, consuming one resource to produce another. As with sources and drains, converters can have different types of rates to consume and produce resources. The Machinations framework represents a converter as a vertical line over a triangle pointing to the right.

Traders are elements that cause resources to change ownership: two players could use a trader to exchange resources. The board game *Settlers of Catan* is built around a trading mechanism allowing players to trade the five types of resource cards among each other against exchange rates they establish among themselves. A player that has many of timber cards, might for example decide to exchange three timber cards for two wool cards of another player. Compared to converters, traders are relatively rare; most of the time, the behavior of a trader is implemented as a converter. For example, depending on the implementation, the merchants in many computer role-playing games where players can barter for goods and equipment are converters not traders. The Machinations framework represents a trader as a vertical line over two triangles pointing left and right.

Automated Machinations Diagrams

The online Machinations tool allows users to draw Machinations diagrams and also run diagrams. While running, the resources in a diagram flow from element to element and flow rates change according to their distribution. In a running diagram nodes have one of three activation modes:

1. Automatic nodes fire once every time step. They are marked with a star.
2. Interactive elements fire when clicked by the user. They have a double outline.
3. Passive elements do not fire at all, unless they are activated by a trigger.

The automated version of the diagrams also introduces three new elements: end conditions, charts, and artificial players. End conditions specify when a diagram has reached an end state. Usually such a state is reached when a specified number of resources is collected or when a particular resource is completely drained. End conditions need to be activated through an activator. End conditions can be used to set goals or build simple timers to limit the game's length. Diagrams that have end conditions are suited to 'quick run': instead of displaying the dynamic behavior as it develops over time, the tool runs the game to its completion immediately. Diagrams can also be quick run several times in succession, in this case the tool will keep track of which end condition was reached how many times.

Charts can be used to plot the state of pools into a graph. Pools and graphs are connected using state connections, but to avoid visual clutter the tool represents these connections as two small arrows, one leading out of the pool and one leading into the graph. The color of these arrows corresponds with the color of the lines in the graph. The data collected by these graphs can be exported as simple comma separated values to be analyzed further by other tools.

Artificial players allow the use of the Machinations tool to simulate players interacting with the diagram. This introduces the possibility of automated multiple tests runs. The implementation of artificial players is rudimentary, but effective. Basically the artificial player has a list of options to activate a specified element and either goes through these options in sequence, or works down the list testing a specified probability for each option until it finds one element to activate. These options might be affected by the state of a pool. For example, the artificial player script for a diagram that contains one pool labeled 'upgrades' and two other elements labeled 'invest' and 'run' might read:

```
invest = 100 - upgrades * 30
run = 100
```

Automated Machinations diagrams offer the opportunity to collect data on the behavior of a game system before the game is built. It allows designers to test typical playing strategies. The artificial players do not have very advanced artificial intelligence, but they can still easily be programmed to follow certain strategies, and will happily do so over thousands of runs. As will become clear in the discussion of *SimWar*, this can be a valuable tool in identifying dominant strategies and testing the balance in a game. Artifical players can be activated and deactivated individually, allowing the user to define different artificial players set up to represent and experiment with different strategies within a single diagram.

Case study: *SimWar*

The Machinations framework can be used to study existing games and support the design of new games. To illustrate the use of the framework I choose to discuss a game of some renown within the design community, yet has never been built:*SimWar*. *SimWar* was presented during the Game Developers Conference in 2003 by game designer Will Wright, who is well-known for his published simulation games: *SimCity*, *The Sims*, etc (Wright 2003). *SimWar* is a hypothetical, minimalistic war game that features only three units: factories, defensive units, and offensive units. These units can be built by spending an unspecified resource that is produced by factories. The more factories a player has the more resources come available to build new units. Only offensive units can move around the map. When an offensive unit meets an enemy defensive unit there is a fifty percent chance that one destroys the other and vice versa. Figure 2 is a visual summary of the game and includes the respective building costs of the three units. During his presentation Will Wright argued that this minimal real-time strategy game still presents the player with some interesting choices, and displays dynamic behavior that is not unlike the behavior found in other games within the same genre. Most notably Wrights argued that a 'rock-paper-scissors' mechanism affects the three units: building factories trumps building defenses, building defenses trumps building offensive units, whereas building of-

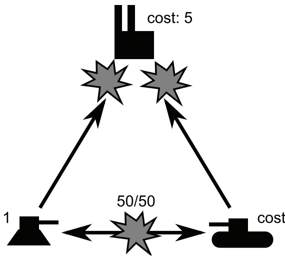

Figure 2: *SimWar* summary, after (Wright 2003)

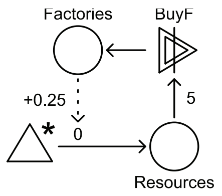

Figure 3: The production mechanism of *SimWar*.

fensive units trumps building factories. Wright describes a short-term versus long-term trade-off and a high-risk/high-reward strategy that recalls the 'rush' and 'turtle' strategies found in many real-time strategies.

Modeling *SimWar*

Building up a model *SimWar* using Machinations diagrams, is best done in few steps. Starting with the production mechanism, a pool is used to represent a player's resources. The pool is filled by a source, the production rate is initially zero, but is increased by 0.25 for every factory the player builds. Factories are build by clicking the interactive converter labels 'BuyF'. Figure 3 contains this diagram. The structure creates a positive feedback loop: the more factories a player builds the quicker resources are produced which in turn can be use to build even more factories. This particular feedback is very common in games; the mechanics in figure 3 constitute a recurrent pattern called a dynamic engine (Dormans 2009). Notice, that this structure requires players to start with at least 5 resources or 1 factory otherwise they can never start producing.

Resources are also used to buy offensive and defensive units. The mechanics for this are represented by figure 4. This diagram makes use of color coded resources. The resources produced by the converter labeled 'BuyD' are black while the resources produced by 'BuyO' are green as indicated by the color of their respective outputs. This means that black resources (representing defensive units) and green resources (representing offensive units) are both gathered on the 'Defending' pool. However, by clicking the 'Attack' gate, all green resources are pulled towards the 'Attacking' pool.

Figure 5 illustrates how combat between two players is modeled. Each attacking unit of one player (red on the left) increases the chance a defending unit of another player (blue on the right) is destroyed, and vice versa. In addition, attacking units increase the chance a factory is destroyed, but that

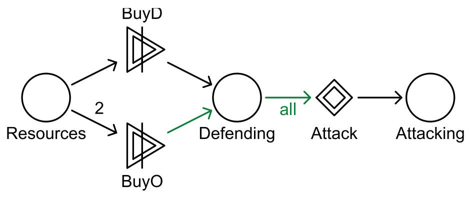

Figure 4: Offensive and defensive units in *SimWar*.

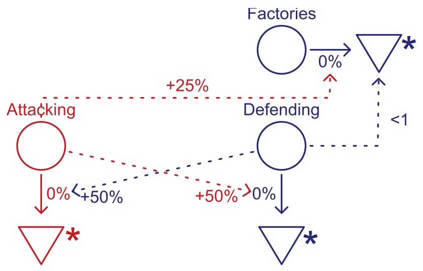

Figure 5: Combat in *SimWar*.

drain is only active when the defending player has no defending units left.

Combining these elements a model can be created for a two player version of *SimWar* (see figure 6). One player controls the red and orange elements on the left side of the diagram, while the other player controls the blue and green elements on the right side of the diagram. Both sides are symmetrical. Note that, in contrast to figure 3, the supply of resources is ultimately limited (as it is in most RTS games). This is to prevent the game from potentially dragging on for ever. If both players run out of resources before they managed to destroy the other, the game ends in a draw.

Simulating *SimWar*

Figure 7 displays the relative strength of each player as it developed over time during a simulated session. The strength was measured by adding five for every factory the player owns plus one for each offensive and defensive units. The chart displays what might be called the fingerprint of an interesting match.

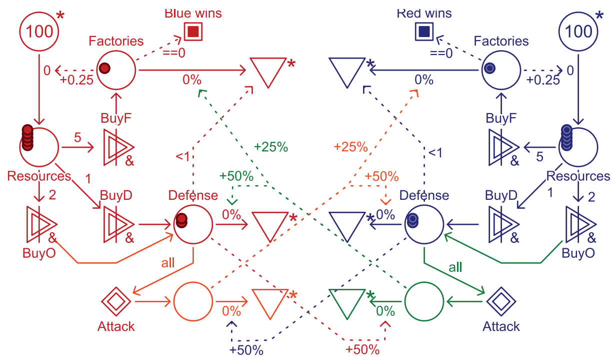

Figure 6: A complete Machinations diagram for *SimWar*.

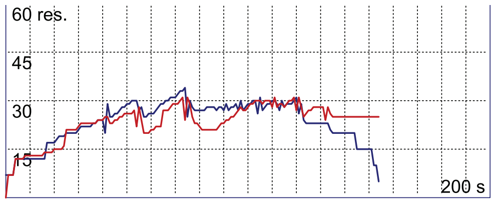

Figure 7: A chart showing the relative strength of each player as it developed over time during a simulates session eventually won by the red player.

This particular session was played by two artificial players set up to follow what might be called a ‘turtling’ strategy, favoring factories and defensive units over offensive units. The script these players followed was:

```
Attack = Defense*5-30
BuyF = 100-Factories*30
BuyD = 100-Defense*15
BuyO = Factories*20+Resources*2
```

Another type of artificial player was created by setting up the script to follow a ‘rushing’ strategy, by building one factory before directing all resources towards building offensive units:

```
Attack = Defense*10-70
BuyF = 200-Factories*100
BuyD = 100-Defense*50
BuyO = 100
```

The ‘rushing’ strategy proved to be very unsuccessful. Out of one thousand simulated session, the ‘rushing’ strategy managed to win only twice. Figure 8 plots the strengths of both players over the session and also indicates when attacks where launched. The ‘rushing’ player (red) builds up a large attack, but does not recover once its units are lost. After that attack, it is fairly easy for the ‘turtling’ player (blue) to defeat red with a series of smaller attacks. Most published real-time-strategies games are balanced towards rushing strategies, as these tend to be harder to execute, and mastered later by players. In order to balance the game a number of ‘tweaks’ were tested: I increased the costs for factories and defensive units, and decreased the cost for offensive units and run the simulation one-thousand time for every modification (see table 1). Surprisingly, increasing the cost for defensive units seem to have little effect. Even when a defensive unit costs more than an offensive unit, making it really poor choice, the turtle strategy remained dominant. This leads to the conclusion that the balance between rushing and turtling strategy is mostly affected by the balance between production and offensive units, and little by the balance between offensive and defensive units.

Discussion

To summarize, the Machinations framework allows designers to model and simulate games in an early stage of development. To simulate a game the complexity of the diagrams to represent it grows quickly. It remains to be determined

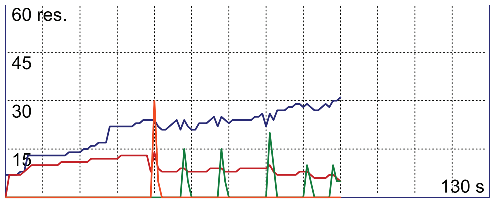

Figure 8: A chart showing a rushing player (red) against a turtling player (blue). The orange spikes indicate attacks waves launched by red, the green spikes indicate attack waves launched by blue.

Tweak	Avg. time	Rush wins	Turtle wins	Draws
No tweaks	69.79s	2	997	1
Factory cost 6	76.33s	7	993	0
Factory cost 7	88.22s	107	889	4
Factory cost 8	91.86s	312	676	12
Factory cost 9	83.58s	557	426	17
Factory cost 10	68.03s	745	228	27
Offense cost 1.5	65.60s	164	836	0
Offense cost 1.4	54.72s	439	561	0
Offense cost 1.3	45.72s	583	417	0
Offense cost 1.2	32.73s	806	194	0
Defense cost 1.5	70.02s	16	983	1
Defense cost 2.0	74.35s	88	911	1
Defense cost 2.5	74.12s	196	803	1

Table 1: Tweaks to *SimWar*'s economy and the effects for "turtling" versus "rushing" strategies.

how easy it is for designers to read the diagrams. However, use of the diagrams with several groups of technical and non-technical students indicates that with some practice, the framework is fairly comprehensible: they were able to get good results. In addition, research in recurrent design patterns in the game's internal economy resulted a small set of patterns that form important building blocks. Understanding these blocks and seeing the larger patterns constitutes an important skill for the use of Machinations diagrams. Machinations diagrams are interactive: they can produce play data even before a prototype is built. The question is how reliable is this data and how relevant are these results for the further development of the game?

Obviously, the simulation of *SimWar* through a Machinations diagram is not quite the same thing is implementing the game. Machinations diagrams focus on a game's internal economy, it does not represent level design or tactical maneuvering that have an equal impact on the gameplay. Although the number of resources available to a player might be dictated by level design, and the two locations for offensive units to be in in a way is reminiscent of tactical maneuvering in a real real-time strategy game. As such this particular technique seems to be more relevant for strategy games, simulation games and board games, as in these types of games internal economy plays a more important role than in certain other types of games.

However, the Machinations framework does foreground feedback structures that play an important role in the design process. By making designers aware of these structures and the effects they have, it might help them to understand the complexity and dynamic of the game in later stages. It is naive to assume that the tweaks suggested in this paper translate to a perfectly balanced game when implemented. The impact different levels and player skill will have on a full game must be taken into account. Instead, the real benefits of doing simulating the game at this points are twofold: 1) it will give designers an indication if a system has the potential for creating particular dynamic effects, and 2) it will give designers some ideas of what values to change in order to tune the system. For example, from the simulation of *SimWar* it does appear that the economic system does generate dynamic gameplay, that allow different types of strategies and without to many obvious choices. In additions, it is important to establish that the price of the defensive units has very little impact on the balance between the 'rushing' and 'turtling' strategies. Designers that are aware that this balance depends more strongly the balance between the costs for the factories and the offensive units will be able to tune the finished game with greater efficiency and accuracy.

In this way the Machinations framework can be used to shape and experiment with game mechanics effectively. When a the design goals of a particular game dictate that a particular dynamic effect is wanted or most be avoided, automated Machinations diagrams seem a useful tool to help design the targeted dynamic behavior. At the same time, as Machinations visualize structures that are otherwise difficult to grasp and understand, it can help designers to grow more confident and enhance their skills in designing complex systems. In that way, an automated tool like Machinations, might actually act as a navigational tool to chart relatively unexplored areas of the design space of games that were previously inaccessible.

References

Adams, E., and Rollings, A. 2007. *Fundamentals of Game Design*. Upper Saddle River, NJ: Pearson Education, Inc.

Dormans, J. 2008. Visualizing game mechanics and emergent gameplay. In *Proceedings of the Meaningful Play Conference*.

Dormans, J. 2009. Machinations: Elemental feedback structures for game design. In *Proceedings of the GAMEON-NA Conference*.

Fromm, J. 2005. Types and forms of emergence.

Hunicke, R.; LeBlanc, M.; and Zubek, R. 2004. Mda: A formal approach to game design and game research. In *Proceedings of the AAAI-04 Workshop on Challenges*, 1–5.

Salen, K., and Zimmerman, E. 2004. *Rules of Play: Game Design Fundamentals*. Cambridge, MA: The MIT Press.

Wolfram, S. 2002. *A New Kind of Science*. Champaign: Wolfram Media Inc.

Wright, W. 2003. Dynamics for designers. Presentation at the Game Developers Conference.