In the previous chapter we have seen that there are no consistent and widely accepted methodologies available for the development of games. Yet, the number of attempts and calls for such endeavors, indicate that a more formal approach to game design is warranted. In this chapter I propose a formal framework that focuses on discrete game mechanics called ‘Machinations’. Initial steps towards the current framework were presented at the Meaningful Play Conference in 2008 and the GAME ON-NA Conference in 2009 (Dormans, 2008b, 2009). The last sections appeared earlier in a paper presented at the Workshop on Artificial Intelligence in the Design Process at the AIIDE Conference in 2011 (Dormans, 2011d).

Taking into account the discussion of existing methodologies and the nature of games in the previous chapters, the Machinations framework presented here adheres to the following requirements:

1. A formal framework for game mechanics should formulate a clear theoretical vision on the structure of game mechanics and their relation to quality in games.

2. A formal framework for game mechanics should not only be an analytical tool for existing games; rather it should have a direct application for the development of new games.

3. A formal framework for game mechanics should provide a good return on the investment to learn it, either by keeping the investment low, or by making the return high, but preferably both.

The first section presents an overview of the Machinations framework and its various components. Sections 4.2 through 4.5 describe the Machinations diagrams that are part of the framework and how they can be used to represent game mechanics. The sections 4.6 to 4.8 describe how, using Machinations diagrams, feedback structures, and recurrent patterns in these structures, can be analyzed and related to a game’s dynamic behavior. In sections 4.9 and 4.10, I will discuss how the Machinations software tool, developed as part of this research, implements Machinations diagrams, and how it can be used to simulate, and generate quantitative data in order to balance games.
concludes with a case study. The Machinations framework is used to explore the theoretical game SimWar described by Will Wright, which, to my knowledge, has never been implemented. With this example I hope to illustrate that the Machinations framework can be used to simulate and analyze games even before they are built.

A word of caution: the Machinations framework is a lot to take in at once. The framework comprises many interrelated concepts that are best understood in unison. This means that there is no real natural starting point to explain all these concepts. I advise the reader to occasionally refer back to earlier concepts. I also like to point out that appendix A presents a single page overview of the most important concepts. In addition, many of the diagram examples can be found online in an interactive version at the Machinations wiki page: www.jorisdormans.nl/machinations/wiki.

4.1 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 designers. What is more, they 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 Machinations diagrams which are designed to represent game mechanics in a way that is accessible, yet retains the structural features and dynamic behavior of the games they represent.

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. This is consistent with findings in the science of complexity that studies dynamic and emergent behavior in a wide variety of complex systems (see section 1.5). By using Machinations diagrams a designer can give game systems a shape which would normally remain invisible. The main premise is that through Machinations diagrams, structure and quality in game mechanics become tangible.

Machinations diagrams are designed to capture game mechanics. As such, they are not only a design tool; they are also useful as an analytical tool to compare and analyze existing games. The Machinations framework allows us to
observe recurrent patterns across many different games. Machinations diagrams are a medium to express and reason about these patterns.

Machinations diagrams can be drawn with any tool. The language was designed to draw easily on a computer, or on paper. At the same time, the syntax of the language is exact. It describes unambiguously how different elements interact. This allowed the development of a Machinations software tool, which can be used to simulate and experiment with game systems. With this tool, a user can run and interact with a Machinations diagram. To a certain extent, a user can play a game represented as a Machinations diagram. In addition, the Machinations tool allows users to define ‘artificial players’ that interact with a diagram automatically. This means that games can be simulated without any actual interaction of real users. Such a simulation can run very quickly, allowing a user to experiment and gather quantitative data from thousands of simulated play sessions quickly and efficiently. To support this, the tool features automatic charts to collect data from each simulated play session.

Figure 4.1 is an overview of the Machinations framework and its most important components. It summarizes the discussion above.

Machinations diagrams create an abstracted perspective on game mechanics and are often used to focus on certain aspects of a game. The framework does not dictate how much detail a diagram should capture or what aspects of the game economy one should represent. Using Machinations diagrams many different aspects can be captured at many different levels of detail. The best perspective and level of abstraction is largely determined by context and purpose. Often it is sufficient to model games from the perspective of a single player, even if the game is actually played by multiple players. In these cases, it is often fairly easy to imagine how a diagram might be duplicated and combined to represent the multiplayer situation. In other cases it is useful to model one player at a
higher level of detail than other players. Likewise, particular aspects of games such as taking turns might be abstracted away. At a high level of abstraction, there often is little difference in the effects of players acting simultaneously and asynchronously, or in alternating turns. I have tried to keep the level of detail low and the level of abstraction high in the examples used in this chapter. This way the structural features of the internal economy are best foregrounded, which best serves the purpose of examining the effects of these structures on emergent gameplay. For this reason the natural scope for Machinations diagram is that of a single player and his or her individual perspective on the game system. Although it is certainly possible to model multiplayer systems, the framework, as it currently stands, does not include features designed to support multiplayer games in particular. For example, the main input device for interaction with a Machinations diagram is the mouse; there is no support for multiple input devices. Likewise, turn taking is geared towards a single player only: the system responds to every turn in a similar way. There is no support for alternating turns for a number of players; and it does not prevent the players to take actions outside 'their turn'.

The Machinations framework focuses on rules and mechanics and does not take into account all elements of game design. Most importantly, the Machinations framework excludes all elements of level design. As such, the framework is more effective for games that do not rely on level design that much. This includes most board games, strategy games and management simulation games. While the framework will still be applicable to games that rely more on level design, for these games the framework can only describe a part of the picture. Level design will be the subject of the next chapter.

Moreover, The Machinations framework does not involve the player or any cultural dimension of representation through games. The framework treats games as complex state machines: interactive devices that can be in many different states, and whose current state determines the transition to a new state. This is an approach that can be and has been critiqued: without players games would be, quite literally, meaningless. The formal rule systems of games are subject to constant change and reinterpretation by players. A formal approach always runs the risk of turning a blind eye towards this dynamic and important dimension of games (see Malaby, 2007). However, building game systems is an important task of a game designer. It is this system that codifies the player’s possible interaction and generates individual game experiences. The aim of the Machinations framework is to understand the elementary structures that contribute to emergent gameplay and that ultimately facilitates the expressive and dynamic nature of games.

Finally, one more word of caution: when one sets out to model anything as complex as games, a model can never do justice to the true complexity of the reality of gameplay. The best models succeed in stripping down the complexity of the original by leaving out, or abstracting away, many important details. This is certainly the case with the framework and diagrams I present here.

---

1This type of mechanics can be modeled using the current framework, but it does require fairly elaborate constructions. There is no 'native' support for this type of features.
However, any model is a tool that can help us understand and work with complex systems. To be able to use the model to the best effect, understanding the concepts that informed the creation of the model is required. As any model, the Machinations framework and diagrams only facilitate understanding; they are never a substitute for it.

### 4.2 Flow of Resources

The Machinations framework utilizes the idea of ‘internal economy’ (Adams & 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. More abstract aspects of games, such as player skill level and strategic position, can also 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 player’s option to build will produce more armies, while with an attack the player risks armies to gain territories and cards.

In order to model a game’s internal economy Machinations diagrams uses several types of nodes, that pull, push, gather and distribute resources. Resource connections determine how resources move between elements and while state connections determine how the current distribution of resources modifies other elements in the diagram. Together, these elements form the essential core of Machinations diagrams. This section explains the basics of resource flow. The next section discusses different types of state connections. Sections 4.4 and 4.5 discuss more specialized nodes used to represent common operations in the game economy. An two-page visual overview of these core elements can be found in appendix A.

Machinations diagrams represent the flow of resources between different game elements. In this respect, Machinations diagrams are dynamic: just like tokens in a Petri net can move between places, resources in a Machinations diagram can move between nodes. The flow of resources is dictated by resource connections

---

2 The history of the notion of internal economy within games is somewhat fragmented. It appears in a chapter title in Rollings & Adams (2003) which is a precursor to Adams & Rollings (2007), but the notion itself is not really discussed in that book. There are a few examples of the use of the term or the synonym ‘in-game economy’ in the context of games, such as in Simpson (2000), Burke (2005) and McGuire & Jenkins (2009). In these three cases the term is reserved for trade and inventory systems (as opposed to, for example, combat systems). McGuire and Jenkins even include ‘commodity flow graphs’ that visualize the flow of resources through a game, but they do not foreground feedback structures as Machinations graphs do (see discussion of their work in section 3.7). It is not until the notion was discussed in more detail in Adams & Rollings (2007) that it started to encompass more types of resources (such as health) than a strict interpretation of ‘economic’ would allow, and that a game’s internal economy could actually include combat systems, leveling systems, etc., as well. Since then, the term appears in lectures and syllabuses that follow Adams and Rollings book. It is in this wider sense that the term is used here.
Figure 4.2: In a Machinations diagram the distribution of resources over nodes represents the state of a game. Resource connections indicate how resources might be redistributed.

represented as solid arrows connecting the nodes of the diagram. The state of the game corresponds to the distribution of resources over the nodes (see figure 4.2). Thus on one hand, a Machinations diagram represents a single state a game can be in, but on the other it determines subsequent states that are its direct result. The nodes and connections of the diagram visualize the structure that ultimately shapes the probability space of game states. The static version of Machinations diagrams accompanying this text obviously cannot display the state changes over time. The distribution of resources in a static Machinations diagrams always represent a game’s initial state, unless stated otherwise.

In a Machinations diagram the nodes are active: they can fire and by firing they cause resources to be redistributed. A node in a Machinations diagram can be in one of three different activation modes:

1. A node can fire automatically: it fires at intervals determined by the diagram’s time mode (see below). All automatic nodes fire simultaneously.

2. A node can be interactive, which means that it represents a player action and fires in response to that action. In the dynamic version of a Machinations diagram, interactive nodes fire after users click on them.

3. A node can be passive, which means it can only fire in response to a trigger generated by another element (see section 4.3 below). Passive nodes still accumulate resources.

As Machinations diagrams are dynamic it is important to understand how they handle time. There are three different time modes for Machinations diagrams:

1. In synchronous time mode all automatic nodes are activated at regular intervals of arbitrary length specified by the user. All interactive nodes clicked by players fire at the next time step, at the same time when automatic nodes fire. In this mode all actions in one time step take place simultaneously. It is possible to for a user to activate multiple interactive nodes during a time step, but during a time step, each interactive node can only be activated once.
2. In *asynchronous time mode* automatic nodes in the diagram are still activated at regular intervals of arbitrary length specified by the user. However, players can activate interactive nodes at any time within the intervals and the resulting actions are executed immediately. In this case an interactive node can be activated multiple times during a time step.

3. Alternatively, a Machinations diagram can be in *turn-based mode*. In this mode time steps do not occur at regular intervals. Instead a new time step occurs after the player has executed a specified number of actions. This is implemented by assigning a number of action points to each interactive node and allotting players a fixed budget of action points each turn. After all action points are used, all automatic nodes fire and a new turn starts.³

The flow of resources from node to node is instantaneous: resources simply disappear and reappear at their new location. However, in order to help designers understand how the internal economy works, the Machinations tools can be set to visualize the resource flow by animating the movement of the resources along the resource connections. This is a visual aid only and does not affect the diagram in any way.

Resource connections, that dictate how resources are distributed when nodes fire, have a *label* that determines the flow rate. Many labels are numbers indicating a fixed flow rate, 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 1.

The most basic nodes connected by resource connections are pools. They are represented as open circles. A pool collects resources that flow into it. Pools can be used to represent any collection of resources in a game. For example, the money possessed by a player of *Monopoly* can be represented as a pool, the money in the ‘bank’ can be represented as another pool. The resources themselves are represented as small dots. Different colors can be used to denote different types of resources. Alternatively, to avoid visual clutter, the number of resources on a pool can be represented as a number. Resource connections that lead into a node are considered to be its inputs while resource connection that lead out of a node are considered to be its outputs.

The activation mode of a pool is indicated by its visual appearance. Passive pools have a single outline, interactive pools have a double outline, automatic pools have a single outline and are marked with a star (‘*’), see figure 4.4. When a pool fires it will try to pull resources through its inputs. The number of resources pulled is determined by the rate of the individual input resource connection. Alternatively, a pool can be set in ‘push mode’. In this mode, a pool will push resources along its outputs when fired. Again, the number of resources pushed is determined by the flow rate of the output resource connection. A pool

³It is possible to create interactive nodes that cost no action points to activate. When all interactive nodes cost no action points, except a single ‘end turn’ action (that has no other effect), this can be used to create a game where players can take any number of actions until they are done.
in push mode is marked with a ‘p’ (see figure 4.4). A pool that has only outputs is always considered to be in push mode, in which case the marker is omitted.

It might happen that two pools try to pull resources from the same source simultaneously, while there are not enough resources to serve both pools. For example, in figure 4.5 every time step pool B automatically pulls one resource from A and both C and D attempt to pull one resource from B. This means that after one time step, B will have one resource and C and D will both try to pull it. How this is resolved depends on the the time mode. In synchronous time mode, neither C nor D can pull the resource. After two iterations when B has pulled a second resource, both C and D will pull one resource from B. While the diagram runs, C and D will both pull a resource once every two time steps simultaneously. As A starts with nine resources, after nine time steps C and D will have four resources and one resource will remain on B. The state of the diagram will then no longer change.

In asynchronous or turn-based mode, either C or D will pull a resource one. Which pool has priority is initially random; subsequently, the priority alternates every time step. This means that C and D will both pull one resource from B on alternating time steps and eventually there will be four resources on C and five on D, or vice versa.

**Figure 4.3:** resource connections in a Machinations diagram.

**Figure 4.4:** Different activation modes of a pool.

**Figure 4.5:** How simultaneous pulls are handled in a Machinations diagram depends on the diagram’s time mode.
4.3 Flow of Information

Machinations diagrams are dynamic beyond the flow of resources. Resources flow rates can be modified by changes in resource distribution. In addition, nodes can be triggered, activated or deactivated in response to changes in resource distribution. These modifications are indicated by a second class of edges called state connections. State connections indicate how the current state of a node, the number of resources on it, affects target elements in the diagram. State connections are represented as dotted arrows. Labels and the type of elements they connect specify the type of state connection. There are four types of state connections that are characterized by the type of elements they connect and their labels:

1. Label modifiers connect a source node to a target label \((L)\) of a resource connection or a state connection. It indicates how state changes of the source node \((\Delta S)\) modify the value of the target label as indicated by the modifier’s label \((M)\). The new value takes effect on the subsequent time step. Thus, the next value of label that is the target of a number \((n)\) of label modifiers is given by the following formula:

\[
L_{t+1} = L_t + \sum_{i=1}^{n} (M_i \Delta S_i)
\]

For example, in figure 4.6 every resource added to pool A adds 2 to value of the resource flow between pools B and C. Thus the first time B is activated, 1 resource flows to A and 3 resources flow to C, the second time, 1 resource still flows to A, but now 5 resources flow to C. The label of a label modifier always starts with a plus or minus symbol indicating incrementation or decrementation.

Label modifiers are frequently used to model different aspects of game behavior. For example, a pool might be used to represent a player’s accumulated property in a game of Monopoly. The more property a player has, the more likely it is that player will collect money from other players.

This can be represented by the diagram in figure 4.7.\footnote{Note that many details are omitted in this diagram, in particular the diagram does not}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure46.png}
\caption{A label modifier affecting the flow rate between two pools. In effect: \(\text{Flow}_{BC} = 3 + 2A\)}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure47.png}
\caption{In Monopoly the state of your property positively affects the chance other player’s money flows to you.}
\end{figure}
Figure 4.8: Node modifiers affect the number of resources on a pool. In effect: $C = 3A - 2B$

Figure 4.9: In Settlers of Catan your score is determined by the number of villages and cities you possess (amongst other things).

case the exact value of the label modifier is unspecified, it only indicates that the effect on the random flow rate is positive.

2. **Node modifiers** connect two nodes. They indicate how state changes of the source node ($\Delta S$) modify the number of resources on the target node ($N$) as indicated by the modifier's label ($M$). The state changes to the target node are further processed during the subsequent time step. Thus, the new number of resources on a node that is the target of a number ($n$) of node modifiers is given by the following formula:

$$N_{t+1} = N_t + \sum_{i=1}^{n} (M_i \Delta S_i)$$

Figure 4.8 illustrates a node with two modifiers. By using negative node modifiers or redistributing resources from a node that has positive input node modifiers it becomes possible that the number of resources on a node becomes negative. In this case, the negative number of resources indicates a shortage. No resources can be pulled from a node that has a shortage, and resources that flow into a node with a shortage are used to compensate for the shortage first.

Node modifier can have labels that are fractions, for example '+1/3' or '-2/4'. In this case the number of resources of a target node is modified by the value indicated by the fraction's numerator every time there is a change to the number resources on the source divided by the fraction's denominator and rounded down. Thus when a source node changes from 7 to 8, the number of resources on the target is lowered by 2 if the modifier is -2/4, but if the modifier is +1/3 the number of resources on the target node does not change.

A simple, real-life example of the use of node modifiers can be found in Settlers of Catan where players gain 1 point for every village in their possession and 2 points for every city in their possession (see figure 4.9).
Node modifiers change the number of resources on other nodes. In effect, the use of node modifiers causes production or consumption of resources. These effects can also be achieved with a slightly more elaborate, but also somewhat cumbersome construction. This construction includes other elements that are introduced later. However, as the use of node modifiers is quite common, they are useful ‘syntactic sugar’: simpler notations of more elaborate constructions. For this reason node modifiers are here treated as a particular type of state modifiers.

3. **Triggers** are state connections that connect two nodes or connects a source node to the label of a resource connection. Triggers fire when all the inputs of its source node become *satisfied*: when each input passed the number of resources to the node as indicated by its flow rate. A firing trigger will in turn fire its target. When the target is a resource connection, it will pull resources as indicated by its flow rate. A node that has no inputs will fire outgoing triggers whenever it fires (either automatically, or in response to a player action, or to another trigger). Triggers are identified by their label which is a star (\(^*\)).

Triggers are commonly used in games to react to redistribution of resources. For example, in MONOPOLY players might transfer money to the bank in order to ‘trigger’ the transfer of property from the bank into their possession. This can be represented as the diagram in figure 4.10.

4. **Activators** connect two nodes. They activate or inhibit their target node based on the state of their source node and a specific condition. The activator’s label specifies this condition. Conditions are written as an arithmetic expression (for example ‘==0’, ‘<3’, ‘>=4’ or ‘!=2’) or a range of values (for example ‘3-6’). If the state of the source node meets this condition then the target node is activated (it can fire). When the condition is not met the target node is inhibited (it cannot fire).

---

5This means that resource connections have a memory, they keep track of how many resources were requested and how many were delivered to pulling resources.
Activators are used to model many different game mechanics. For example, in the board game Caylus players place their laborers (a resource) at particular buildings on the board to enable them to execute special actions associated with that building. For example, a player might place a laborer at a goldmine in order to collect gold (see figure 4.11). However, as indicated by the trigger in figure 4.11, in Caylus every time a player exercise this option, the laborer is returned to the player’s laborer pool.

4.4 Controlling Resource Flow

Pools are not the only possible nodes in a Machinations diagram. Gates are another type of node. In contrast to a pool a gate does not collect resources, instead it immediately redistributes them. Gates are represented as diamond shapes that often have have multiple outputs (see figure 4.12). Instead of a flow rate, each output is labeled with a probability or a condition. 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 must be probable and when one output is conditional, all must 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. Gates with probable outputs can be used to represent chances and risks. For example, in Risk players risk armies in order to gain territories. This type of risk can easily be represented by a gate with probable outputs indicating the rates for success or failure.

An output is conditional when it is labeled with a condition (such as ‘>3’ or ‘==0’ or ‘3-5’). In this case, all conditions are checked every time a resource arrives at the gate and one resource is sent along every output whose condition is met. As the conditions might overlap this can lead to duplication of resources, or, when no condition is met, to the destruction of the resource.

Like pools, gates have three activation modes: gates can be passive, interactive or automatic. Interactive gates also have a double outline and automatic gates are also marked with a star. When a gate has no inputs, it triggers every time it fires, this way gates can be used to produce triggers either automatically or in response to player actions.

Furthermore, gates have one of two different distribution modes: deterministic distribution and random distribution. A deterministic gate will distribute resources evenly according to the distribution probabilities indicated by percentages or weights if it has probable outputs. When it has conditional outputs it will count the number of resources that have passed through it every time step and uses that number to check the conditions of its outputs.\(^6\) A deterministic gate has no special symbol and is represented as a small open diamond.

A random gate generates a random value to determine where it will distribute incoming resources. When it has probable outputs it will generate a suitable number (either a value between 0 and 100 percent, or a number below the total weights of the outputs). When its outputs are conditional it will produce a value between 1 and 6 to check against the conditions, just as if the diagram rolled a normal six-sided die.\(^7\) Random gates are marked with a dice symbol.

Gates might have only one output. Gates with one output act exactly the same way as gates with multiple outputs. The gates on the middle row of figure 4.12 will (from left to right) randomly let 30 percent of all the resources pass, immediately pass the resource to the output regardless of the output’s flow rate, and let only the first two resources pass.

All output state connections from a gate are triggers; gates do not accumulate resources and therefore label modifiers, node modifiers and activators originating from a gate have no function. The triggers are activated instead of redirecting

---

\(^6\)It can be convenient to think of a deterministic gate with conditional outputs as ‘counting gate’.

\(^7\)This value can be set represent other types of random distribution if needed.
resources. These triggers can also be conditional or probabilistic. In this way gates can be used to control the flow of resources (see figure 4.13).

4.5 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 & Rollings, 2007, 331-340). 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 is useful to introduce special nodes to represent sources, drains, converters and traders.

Sources are nodes 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. Health packs are sources of health in shooter games. As any node in a Machinations diagram, sources have an activation mode that is either passive, interactive or automatic. An example of an automatic source is the steady regeneration of the protective shields of the player’s star fighter in Star Wars: X-Wing Alliance. The rate at which a source produces resources is a fundamental property of a source. The building action in Risk and the passing of ‘Go’ in Monopoly are examples of sources activated by user actions and game events respectively. Adams and Rollings distinguish between ‘limited’ and ‘unlimited’ sources (Adams & Rollings, 2007, 333). A limited source can easily be represented as a pool without inputs that starts with a number of resources on it. An unlimited source can be represented as a pool without inputs but with a sufficiently large number of resources on it (or rather an infinite number of resources). To represent unlimited sources, the Machinations framework includes a special source node represented as a triangle pointing upwards (see figure 4.14). Note that Adam and Rollings’ notion of a ‘limited source’ is still represented as a pool with no inputs.

Drains are nodes 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. Being underwater in most games causes a resource representing breath to be drained away. The rate of a drain is determined by the flow rate of its input resource connection. Some drains consume resources at a steady rate while oth-
ers consume resources at random rates or intervals. Drains could be represented as a pool with no outputs, but the Machinations framework includes a special drain node represented as a triangle pointing downwards (see figure 4.14).

Converters convert one resource into another. In Monopoly the option to buy property acts as a converter: the resource money is converted into another resource: property. In a shooter game, killing enemies might also 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 nodes that cause resources to change ownership when fired: 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; in many games, what appears to be a trader is often 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. These merchants will happily buy whatever the player has to offer and seem to have an unlimited supply of handy items and money to buy the player’s unwanted loot (Castronova, 2005, 198-199). Fallout 3, where all traders’ supplies are limited, is an exception. A trading mechanism can be constructed by two gates connected by a trigger ensuring that when one resource is received the other is returned in exchange. The Machinations framework represents a trader as a vertical line over two triangles pointing left and right.

The difference between converters and traders is not always immediately
clear. Especially from the perspective of an individual player, converters and traders will almost have the same function: pass a number of resources to it, and get a number of other resources in return. Yet, there is an important difference. As pointed out above: a converter can be seen as a combination of a drain and a source. Using a converter resources are actually consumed and produced, and therefore the total number of resources in the game might change. Whereas with a trader the number of resources always stays the same (also see Adams & Rollings, 2007, 334).

The economic function of a particular game element can change according to the perspective of the diagram. For example, figure 4.15 represents MONOPOLY from the perspective of an individual player. Only in the perspective of an individual player rent income is a source. From the perspective of the entire game rent acts as trader: money simply exchanges ownership. In this case, the bank should be modeled as an individual pool with which money is only traded. After all, the boxed game comes with a finite supply of play money. All of these would constitute valid models of the same game. As mentioned in the introduction, the Machinations framework allows one to model a game with different levels of detail and abstraction. The important question is: what does one try to model? For a basic understanding of behavior of MONOPOLY a simple, limited perspective as in figure 4.15 might suffice. Especially when one can imagine how this same structure is repeated for every player. But for a more thorough analysis more detail might be required.

8When a converter is replaced by two pools as the equivalents of a source and a drain suggest the number of resources also stays the same. However, if one considers resources that are trapped on a pool without outputs and which state no longer affect the game to be inactive then the number active resources might change.

9For an example of such an analysis see http://www.jorisdormans.nl/machinations/wiki/index.php?title=Tutorial_1. In this example the model for MONOPOLY includes two players. The only reason I can imagine why anyone would want to model the bank is in a detailed study to find out how much money should be included in the published game.
4.6 Feedback Structures in Games

The structure of a game’s internal economy plays an important role in a game’s dynamic behavior and gameplay. Just as feedback plays an important role in any dynamic system (see section 1.5), it plays an equally important role in a game’s internal economy. The idea of applying the concept of feedback to games is not new. During his 1999 lecture at The Game Developers Conference Marc LeBlanc introduced feedback loops to the game design world (1999). Since then, feedback loops have been discussed by a number of influential designers, including Salen & Zimmerman (2004), Adams & Rollings (2007) and Fullerton (2008). A classic example of feedback in games can be found in MONOPOLY where the money spent to buy property is returned with a profit because more property will generate more income. This feedback loop can be easily read from the Machinations diagram of MONOPOLY (figure 4.15): it is formed by the closed circuit of resource and state connections between the money and property pools. More specifically, for feedback to exist, a close circuit of connections is required that consists of at least one state connection that is not an activator. A closed circuit of resource connections can only create a loop of resources. To change the rate of the flow, at least one label modifier, trigger or activator that changes or triggers the production or consumption of the resources must be part of the loop. Note that for this reason a closed circuit of resource connections that includes a converter or trader also constitutes a feedback loop, as their equivalent constructions do include a state connection (see figure 4.14).

As is the case in classic control theory (DiStefano III et al., 1967), Marc LeBlanc distinguishes between two types of feedback: positive and negative feedback (also see section 1.5). Positive feedback strengthens itself and destabilizes a system. Positive feedback occurs when an effect fuels itself. In the MONOPOLY example above, the feedback is positive because investing money will generate more money. Positive feedback amplifies small differences between players: a player that has a lucky break early in the game will find this luck amplified over time: a player that by chance gets the option to get more money or good property early in a game of MONOPOLY is very likely to win.10 Positive feedback can be applied to ‘positive’ game effects but also to ‘negative’ game effects, as is the case with losing pieces in CHESS, which increases the chances of losing more pieces, and which will eventually make you lose. LeBlanc suggests that positive feedback drives the game to a conclusion and magnifies early successes (see also Salen & Zimmerman, 2004, 224–225).

Negative feedback is the opposite of positive feedback. It stabilizes a game by diminishing differences between players, by applying a penalty to the player who has done something that takes him closer to his goal and winning the game, or by giving advantages to the trailing players. Many racing games use negative feedback to keep a race close, either by giving trailing players more advantages or by hindering leading players. This effect is often described as ‘rubber-banding’.

10Which, incidentally, is exactly the point the original designers of MONOPOLY’s ancestor THE LANDLORD GAME were trying to make: it was a critique of capitalism that favors those that possess capital and works to widen the gap between the rich and the poor (Fron et al., 2007).
It can be implemented by blatantly giving trailing players better acceleration and more grip, or more subtly as is the case in *Super Mario Kart* by having the most effective weapons in the game affect cars in front of the player that uses them. LeBlanc points out that in most multiplayer games that allow direct interaction, some sort of negative feedback is already in place: rational players will target the leader more than any other player. As one might expect, negative feedback can prolong a game and magnifies late successes.

Control theory, in almost all cases, strives for negative feedback while avoiding positive feedback, as it aims to create stable systems. A large part of control theory concerns itself with determining and optimizing the stability of the system. For games the situation is, of course, very different. Positive feedback loops are much more frequent in games because, in general, players do not want to play a game that is stable and drags on forever. Negative feedback does have an application within games: most games with only positive feedback will seem too random to many players as they will be unable to catch the player who took an early lead; negative feedback is often used to balance out early successes.

Just as one feedback loop will only create weak emergence in Jochen Fromm’s typology (see section 1.5), most games need multiple feedback loops to display truly interesting emergent behavior. The game *Risk* is an excellent illustration of this as in this game four feedback loops interact.

The core feedback loop in *Risk* involves the resources armies and territories. Figure 4.16 depicts this core feedback loop. The label ‘+1/3’ of the label modifier that sets the output flow rate of the source ‘build’ indicates that the output of the source is improved by one for every three territories the player has.

The second feedback loop in *Risk* is formed by the ‘cards’ that are gained from a successful attack (see figure 4.17). Only one card can be gained every turn, thus the flow of cards passes through a limiter gate first. Collecting a set of three cards can be used to generate new armies. Not every set generates armies, and some sets generate more than others. In figure 4.17 these effects are indicated by the random symbol labeling the output of the converter that converts cards into armies.

The third feedback loop is activated when a player manages to capture a continent, which will give the player bonus armies every turn (see figure 4.18). In *Risk* predefined groups of territories form continents as indicated by the design of the game board. In the diagram this level of detail is not possible, instead
the construction is represented as a pool connected to another pool with a node modifier. In this particular case, seven territories will count as one continent which will in turn activate the bonus source.

Finally, the fourth feedback loop is activated by the loss of territories due to the actions of other players (figure 4.18). Which player is going to attack which other player is dependent on many factors, including those players strategies and preferences. Sometimes it is opportune to prey on weaker players in order to gain territories or cards, sometimes it is important to oppose stronger players to keep them from winning. In the diagram this is indicated by the multiplayer dynamic label (an icon depicting two pawns) affecting the resource flow to the drain on the right. The number of continents a player captured has a strong influence on this. As in general, players do recognize that other players that have a continent have a big advantage and will usually act against that more vigor.

The player might also lose armies to actions of other players. I have chosen to omit them from the diagram to avoid too much clutter. It should not affect the argument too much. The important thing is, that in Risk there is some form of friction caused by other players, and the strength of this friction increases when the player has captured continents. This type of friction is a good example of the negative feedback that can almost always be found in multiplayer games where players can act against each other as pointed out by LeBlanc (see above).

The first three feedback loops in Risk all are positive: more territories or cards will lead to more armies which will lead to more territories and cards. Yet they are not the same. The feedback of cards is much slower that the feedback of territories as a player can get only one card each turn, but at the same time the feedback of the cards is also much stronger. Feedback from capturing continents operates faster and even stronger. These properties are important characteristics of the feedback loops that have a big impact on the dynamics of the game. Players are more willing to risk an attack when it is likely that the next card they will get completes a valuable set: it does not improve their chances of winning a battle but it will increase the reward if they do. Likewise the chance of capturing a continent can inspire a player to take more risk than the

**Figure 4.17:** The second feedback in Risk involving cards and armies.
Figure 4.18: The third feedback loop in Risk: bonus armies through the possession of continents.

Figure 4.19: The fourth feedback loop in Risk: negative feedback caused by capturing continents. Note that in this figure the label modifier affecting the resource connection from the territories pool to the opposition drain does create a closed circuit as both nodes at the end and the start might be affected by the changing flow rate.
player should. In Risk the player’s risks and rewards constantly shift, making the ability to understand these dynamics and to read the game a decisive skill in this game. These three positive feedback loops play an important role but simply classifying them as positive does not do justice to the subtlety of the mechanics.

### 4.7 Feedback Profiles

To really appreciate the feedback structure of Risk the differences between the three positive feedback loops must be explored in more detail. The feedback of capturing territories to be able to build more armies is straightforward, fairly slow and involves a considerable investment of armies. Often players lose more armies in an attempt to conquer territories than they will regain with one build. This leads to the common strategy to build during multiple, consecutive turns. The feedback involving the cards requires a considerably larger effort on the part of the player. Players can only gain one card during a single turn, no matter how many attacks were successful. However, depending on the cards the player draws, the return might be much higher. The feedback involving continents is very fast and with a high return. Players will receive bonus armies every turn no matter whether they choose to build or to attack. The feedback is so strong and obvious that it will typically inspire fierce counter measures from other players.

Table 4.1 lists seven characteristics that, together with the determinability characteristic discussed below, form a more detailed profile of a feedback loop. At a first glance some of these characteristics might seem overlapping, but they are not. It is easy to confuse positive feedback with constructive feedback and negative feedback with destructive feedback. However, positive destructive feedback does exist. For example, losing pieces in a game of Chess will increase the chance of losing more pieces and losing the game. Likewise, the board game Power Grid employs a mechanism in which the game leaders have to invest more resources to build up and fuel their network of power plants: negative feedback on a constructive effect.

The strength of a feedback loop is an informal indication of its impact on the game. Strength cannot be attributed to a single characteristic: it is the result of several. For example, permanent feedback with a little return can have a strong effect on the game. The effects of a feedback loops on a game can drastically change with these characteristics. Feedback that is indirect, slow but with a lot of return and not durable has a strong destabilizing effect. In this way even negative feedback can be used to destabilize a system if it is applied erratically or when its effects are strong, but slow and indirect.

In many games the profile of a feedback loop is also affected by factors such as chance, player skill and social interaction. Table 4.2 lists the different types on determinability used in the Machinations framework and the icons used to...
<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td>Positive</td>
<td>Amplifies differences, destabilizes a game.</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td>Dampens differences, stabilizes or balances a game.</td>
</tr>
<tr>
<td><strong>Effect</strong></td>
<td>Constructive</td>
<td>Operates on a game effect that helps a player win.</td>
</tr>
<tr>
<td></td>
<td>Destructive</td>
<td>Operates on a game effect that will make a player lose.</td>
</tr>
<tr>
<td><strong>Investment</strong></td>
<td>High</td>
<td>Many resources must be invested to activate the feedback.</td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>Few resources must be invested to activate the feedback.</td>
</tr>
<tr>
<td><strong>Return</strong></td>
<td>High</td>
<td>The net gain is high</td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>The net gain is low</td>
</tr>
<tr>
<td></td>
<td>Insufficient</td>
<td>The gain does not outweigh the investment (net gain is negative).</td>
</tr>
<tr>
<td><strong>Speed</strong></td>
<td>Immediate</td>
<td>The feedback is in effect immediately.</td>
</tr>
<tr>
<td></td>
<td>Fast</td>
<td>The feedback takes a little time to take effect.</td>
</tr>
<tr>
<td></td>
<td>Slow</td>
<td>The feedback takes a lot of time to take effect.</td>
</tr>
<tr>
<td><strong>Range</strong></td>
<td>Short</td>
<td>The feedback operates directly over a few steps.</td>
</tr>
<tr>
<td></td>
<td>Long</td>
<td>The feedback operates indirectly over many steps.</td>
</tr>
<tr>
<td><strong>Durability</strong></td>
<td>None</td>
<td>The feedback works only once.</td>
</tr>
<tr>
<td></td>
<td>Limited</td>
<td>The feedback works only over a short period of time.</td>
</tr>
<tr>
<td></td>
<td>Extended</td>
<td>The feedback works over a long period of time.</td>
</tr>
<tr>
<td></td>
<td>Permanent</td>
<td>The effect of the feedback is permanent.</td>
</tr>
</tbody>
</table>

Table 4.1: Feedback Characteristics
denote them. These icons can be used to annotate resource connections and gates in a diagram. A single feedback loop can be affected by multiple and different types of nondeterministic resource connections or gates. For example, the feedback through cards in Risk is affected by a random gate and a random flow, increasing its unpredictability.

The profile of multiplayer feedback in a game that allows direct player interaction, like Risk, can change over time. As LeBlanc already pointed out, it often is negative feedback as players act stronger, or even conspire against the leader. At the same time, it can also be positive as in certain circumstances, as mentioned above, it can be beneficial to prey on the weaker player.

The skill of players in performing a particular task can also be a decisive factor in the nature of feedback, as is the case for many computer games. For example, Tetris gets more difficult as the blocks pile up, the rate at which players can get rid of the blocks is determined by their skill. Skillful players will be able to keep up with the game much longer than players with less skill. Here player skill is a factor on the operational or tactical level of the game. In games of chance, tactics, or games that involve only deterministic feedback, a whole set of strategic skills can be quite decisive for the outcome. However, that is a result of a player’s understanding of the game’s feedback structures as a whole, and as such it is not an element that can or needs to be modeled within the structure. This feedback loop in Tetris is also affected by randomness. The shape of the block is randomly determined by the game. Although, the skill is

<table>
<thead>
<tr>
<th>Type</th>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic</td>
<td>(none)</td>
<td>Given a certain game state, the mechanism will always act the same.</td>
</tr>
<tr>
<td>Random</td>
<td><img src="image.png" alt="Random Icon" /></td>
<td>The mechanism depends on random factors. The randomness can affect speed and/or return of a feedback loop, or the possibility of feedback occurring at all. It can create an infrequent return. Random feedback is difficult for the player to assess, and increases the chance of deadlocks.</td>
</tr>
<tr>
<td>Multiplayer-dynamic</td>
<td><img src="image.png" alt="Multiplayer Icon" /></td>
<td>The type, strength, and/or game effect of the mechanism are affected by the direct interaction between players.</td>
</tr>
<tr>
<td>Meta-dynamic</td>
<td><img src="image.png" alt="Meta-dynamic Icon" /></td>
<td>The type, strength, and/or game effect of the mechanism are affected by the tactical or strategic interaction between players.</td>
</tr>
<tr>
<td>Player skill</td>
<td><img src="image.png" alt="Player skill Icon" /></td>
<td>The type, strength, and/or game effect of the mechanism are affected by the player's manual skill in executing the action.</td>
</tr>
</tbody>
</table>

Table 4.2: Determinability
generally more decisive in Tetris, the player just might get lucky.

Games that feature only deterministic feedback can still show surprising unpredictable outcomes, as emergence itself can be a source of unexpected and hard to predict behavior. In fact, it is my conviction that a well-designed game is built on only a handful feedback loops and relies on chance, multiplayer dynamic, and skill only when it needs to and refrains from using randomness as an easy source of uncertainty.

A feedback loop’s characteristics and determinability form the feedback’s profile. While a profile like this can be very helpful in identifying the nature of feedback in a game, it does little to reveal the interaction between different feedback loops. This is where diagrams, such as figure 4.20, excel. Many of the characteristics of feedback loops described above can be read from the diagrams. The effect of the feedback is directly related to the constructive or destructive nature of the feedback loop, whereas return and investment depends on the number of resources involved. A feedback loop that consists of almost only state connections and triggers, and few interactive nodes, is likely to have a high speed. Range can be read from the number of elements involved in the feedback loop, speed from the number of iterations required to activate the feedback. The return of a feedback loop must be read from the modifiers of the arrows that create the closed circuit, as some of these modifiers might be nondeterministic the return is more difficult to assess or actually becomes uncertain. The type of feedback (positive or negative) is perhaps the most difficult to read from a static representation, and requires careful inspection of the diagram, but this is possible, too. Note that the plus symbols in the diagrams do not indicate positive feedback, only that there is positive correlation between the number of resources in the pool and the label it is affecting. A positive correlation can induce negative or positive feedback.

4.8 Feedback Analysis and Recurrent Patterns

An analysis of a game’s feedback loops can be used to identify structural strengths and flaws in its design. To create interesting and varied gameplay feedback is an important device, and most successful games incorporate two or more, but not that many more, feedback loops in its main structure. Structural flaws, or ‘bad smells’ in analogy to software engineering, are constructions that are best avoided. If we take Risks again as our example, we can identify one of its problems from play experience: building as often as you can is an effective, almost dominant, strategy. To counter this strategy the game includes a special rule preventing the players from building on more than three subsequent turns. Inspection of the feedback structure of the game suggests other ways of resolving the problem. Attacking feeds into a triple positive feedback structure (through territories, cards and continents), which is a strength of it its design, but apparently the feedback is not effective enough. Strengthening the feedback of territories will help only a little as building is part of the same feedback loop and will probably encourage the unwanted behavior. Either the feedback through cards or the feedback through continents needs to be improved. The
Figure 4.20: A Machinations diagram for Settlers of Catan, which is won by collecting ten points. The game’s five resources are collapsed into one for this diagram. Normally, a player has to pay with a particular set of resources to perform an action. The relative value of these different resources varies as production of each individual resource is subjected to chance. Settlers of Catan has three main feedback loops: 1) The slow, expensive but durable increase of production through the investment in roads, villages, and cities. 2) Buying cards, which is fast, unpredictable and has no durability. And 3) through trade with other players which is subject to multiplayer-dynamics.
The card feedback loop involves two random factors: success of attack and the blind draw of the card itself. This makes the feedback unpredictable and very hard for the player to assess. In general, involving too much randomness in the same loop is best avoided, especially when this randomness affects different steps in the loop. It is very hard to balance and predict the feedback of such a loop, so reducing the randomness, for example by allowing the winner a pick of three open cards, will help a lot.

Alternatively the feedback through the capture of continents can be improved. The problem with this feedback is that it has a high return, is permanent, direct and fast: it is very obvious and will inspire strong reaction by opposing players, in other words it acts as a red flag. Combined with a relative high investment, it constitutes an effective but risky strategy. The strength and the obviousness of the feedback invites a strong negative feedback. This creates a feedback loop that is too crude: it is either on and going strong or it is off. Either the player succeeds in taking and keeping a continent and has a very good shot at winning, or the player is hit hard and loses what usually is a considerable investment. By making the feedback less strong, and perhaps increase the number of continents (or rather regions) for players to conquer, a more subtle feedback loop is created that will pay-out more often without unbalancing the game too much.\footnote{On the other hand, the game is called ‘risk’ for a reason, risk taking is an intended part of the game play. How much risk is suitable for this game is also a matter of taste.}

Looking at feedback structures in games, many recurrent patterns emerge. For example, both Monopoly and Risk share a similar structured, positive feedback loop that can be found in many other games as well. This pattern, which I call a dynamic engine, revolves around a source producing one type of resource, which can be converted to improve the source. Figure 4.21 depicts this elemental feedback pattern, using the generic names energy and upgrades for the two resources involved. In Monopoly these resources are money and property respectively, whereas in Risk these are armies and territories. The pattern can be found in many more games. In StarCraft the player invests minerals to build SUV units to mine more minerals.

Settlers of Catan (see figure 4.20) has a more complicated implementation of this pattern, one where a player needs to build roads before that player can build villages, and where villages can be upgraded to cities. In this case the

![Figure 4.21: The dynamic engine pattern.](image-url)
dynamic engine is also part of a engine building pattern (see appendix B).

A dynamic engine has a very typical gameplay signature. When play begins players will invest most of their energy in upgrades for a while. At a certain point, players start to use the energy elsewhere or, when that is the set goal for the game, simply collect it. When one plots the output of energy over time in a graph, this leads to a sharply cornered line (see figure 4.22). This signature is recurrent in all games that use a dynamic engine, although it might be obscured by the randomness or nondeterministic behavior caused by other feedback structures. For example, with one strategy in StarCraft called "turtling", players invest a lot in building their base, before starting to build an offensive force to attack the enemy. When these players start attacking their offensive output is usually quite large. In the opposite strategy, called the “Zerg rush” after one of the game’s playable factions, the players invest little in their base, instead they start building offensive units as fast as they can in a bid to overpower their opponents before they have built up adequate defense (see figure 4.23). The effectiveness of the latter strategy depends on the speed of the attacking player, but also on the balance between offensive capabilities, defensive capabilities and the building costs of the units involved. Section 4.11 will discuss the balance between these two strategies in more detail.

Twelve more recurrent feedback patterns are described and discussed in appendix B.

4.9 Implementing Machinations Diagrams

The online Machinations tool does not only allow users to draw Machinations diagrams, it can also run diagrams. While running, the resources in a diagram flow from node to node and flow rates change according to their distribution. The digital version of the diagrams introduces an extra activation mode, two different modes the nodes can use to push or pull resources, three new types of nodes, and the concept of color-coded resources. All of these are discussed in this section.

The new activation mode digital Machinations diagrams introduce is the

---

13 See the tool’s web page: http://www.jorisdormans.nl/machinations

![Figure 4.22](image-url): The gameplay signature of a dynamic engine.

![Figure 4.23](image-url): The turtle and rush strategies in StarCraft are the result of a dynamic engine.
Figure 4.24: Extra modes and nodes in automated Machinations diagrams.

‘starting action’ mode. Nodes in this mode fire once when the diagram is started and are marked with an ‘s’ instead of the star used to mark automatic nodes (see figure 4.24).

The pull modes of a node specify in more detail how a node pulls resources from another node. There are two different pull modes:

1. By default, a node pulls as much resources as it can, up to the flow rates of its inputs. If not all resources are available, it still pulls those that are.

2. Alternatively, a node can be set to pull all or none resources. In this mode, when not all resources are available, none are pulled. Nodes that are in ‘all or none’ pull mode are marked with an ‘&’ sign (see figure 4.24).

These modes also apply to pushing modes: by default a pushing node sends as many resources out along its output resource connection up to outputs’ flow rate. A pushing node in ‘all or none’ mode only sends resources when it can supply all of its outputs. This means that nodes in push mode might be marked with both a ‘p’ and a ‘&’.

The three new nodes are: end conditions, charts, and artificial players. End conditions specify when a diagram has reached an end state and can stop further execution. Usually such a state is reached when a specified number of resources is collected or when a particular resource is completely drained (see figure 4.25). End conditions need to be activated through an activator, they do not have a activation mode as other nodes do. 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, it runs the game to its completion immediately. Diagrams can also be run several times in succession, in this case the tool will show 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 node modifiers, but to avoid visual clutter the tool represents these state 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 (see figure 4.26). 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 node and either goes through these options in sequence, or works down the list testing a specified probability for each option until it finds one node to activate. These options might be affected by the state of a pool. For example, the artificial player script for figure 4.26 reads:

\[
\text{invest} = 100 - \text{upgrades} \times 30 \\
\text{run} = 100
\]

This script will initially cause the artificial player to invest, but with every upgrade the chances it will invest are decreased by thirty percent. If it does not invest, there is a one-hundred percent chance it will run instead.

In the digital Machinations tool the color of resources is meaningful. If a resource connection has a different color than the color of the pool then only those
resources which color matches the color of the resource connection can be pulled through that resource connections. This allows different types of resources to be stored on the same pool. Likewise sources and converters producing resources, produce resources in the respective colors of their outputs, when these outputs have a different color than the source or converter. I use the term color-coding to refer to this use of colored resources and resource connections in a Machinations diagram. Label modifiers, node modifiers, and activators which color is different from the color of the node they originate from, act according to the number of resources of that color on the pool. Figure 4.27 illustrates how this can be used in a diagram. In this figure source A produces a random number of orange and blue resources every time it is activated. Both are collected at pool B, the number of orange resources on B increases the number of blue resources produced and vice versa. The user can only activate drain C when there are at least 20 red and 20 blue resources on pool C. Color-coded resources are used in the case-study of SimWar later in this chapter.

Digital 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 in section 4.11, this can be a valuable tool in identifying dominant strategies and testing the balance in a game. Artificial 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.

4.10 Randomness and Nondeterministic Behavior

In many games, complexity is not the only source of nondeterministic behavior. As was argued in chapter 2, dice (or other random generators) are a good way to create nondeterministic behavior for those mechanics that are not the core of the gameplay. In this way, dice can be used to simulate the outcome of battles in Risk or Kriegsspiel on the one hand, or to simulate the conditions that affect the rate of production in a game like Settlers of Catan on the other. From the perspective of the Machinations framework, randomness is a good tool to inspire particular behavior from the players but it might also be used to obscure dominant gameplay signatures that originate from certain feedback structures, such as the dynamic engine pattern.
An account on how randomness can affect the behavior of the player is given by John Hopson (2001). He argues that consistent with the findings in behavioral psychology experiments, player behavior is affected by chance and the interval the player is awarded for actions. When a player has a chance to be awarded at regular intervals, the player’s attention and activity will spike at those intervals, where as when those intervals have random lengths, the player will be active most of the time. After all, the next action might lead to a new reward. As such, it is vital that in a Machinations diagram you can set up random values as well as random intervals.

The effect of randomness on the dynamic engine signature is illustrated with the following experiment. A simple racing game for two players utilizes a simple dynamic engine. The goal of the game is to collect thirty ‘distance resources’ by running. But the player can also choose to invest three energy to produce an upgrade which will increase the rate of production of energy (see figure 4.28). The energy production starts at a rate of 0.1 which means 1 resource is produced every ten seconds. Every upgrade improves this rate by 0.1. The artificial players controlling the black and gray diagrams are set up slightly differently. The black player will first buy four upgrades before it starts running. The gray player will buy only two upgrades. Obviously, black’s strategy is superior: black wins every time. The chart in figure 4.28 shows this trend: the black line reaches 30 before the gray line does.

When the energy sources are changed to have a probable output of 10 percent every second, and each upgrade will increase this probability by another 10 percent (see figure 4.29), the pattern is broken. Figures 4.30 and 4.31 show sample data generated by different runs. Gray now has a chance of about 23 percent to win. In effect, the randomness can counter the effect of the feedback loop as was also suggested by Ernest Adams and Andrew Rollings (2007, 387).

The Machinations tool allows the designer control over random values produced. As we have already seen, the percent sign (‘%’) is used to denote a probability. A source that has probable output is labeled ‘20%’ will have a twenty percent chance to produce a resource every time step. The tool can also simulate dice rolls by using a similar notation for dice rolls and calculations that is commonly used in pen-and-paper role-playing games. In these games ‘D6’ stands for a random number produced by a roll of a single six-sided die, where as ‘D6+3’ adds three to the same dice roll, and ‘2D6’ adds the results of two six-sided dice and thus will produce a number somewhere between two and twelve. Other types of dice can be used as well: ‘2D4+D8+D12’ indicates the result of two four-sided dice added with the results of an eight- and twelve-sided die. Unlike pen-and-paper role-playing games, the Machinations tool is not restricted to dice that are commercially available. It can use five-, seven or thirty-five-sided dice.

Intervals are created by using a slash (‘/’) for inputs and outputs. For example, a source that produces ‘D6/3’ resources will produce between one and six resources every three seconds. Intervals can also be random: a drain that has an input with a modifier that states ‘3/D6’ will drain 3 resources with an interval

---

14 I ran the diagram 1000 times which resulted in 768 wins for black and 232 wins for gray.
Figure 4.28: Two players racing for distance in a game with deterministic behavior.
Figure 4.29: Two players racing for distance in a game with random behavior.

Figure 4.30: Sample result of a race with random behavior.

Figure 4.31: Another sample result of a race with random behavior.
between one and six seconds.

4.11 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 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. 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 4.32 can be seen as 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 offensive 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 (see section 4.8).

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 an automatic source. The source’s production rate is initially zero, but is increased by 0.25 for every factory the player builds. Factories are built by clicking the interactive converter labeled ‘BuyF’, which will pull resources only when at least five are available. Figure 4.33 contains this diagram. The structure is a typical implementation of a dynamic

Figure 4.32: SimWar summary, after Wright (2003)
engine pattern that we have seen before. As all dynamic engines, it 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. Notice that, in this case, the structure requires players to start with at least 5 resources or 1 factory otherwise players can never start producing.

Resources are also used to buy offensive and defensive units. The mechanics for this are represented by figure 4.34. 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. Thus only offensive units can be used to launch an attack.

Figure 4.35 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 drain is only active when the defending player has no defending units left.

Combining the structures of each step, a model can be created for a two player version of SimWar (see figure 4.36). 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 4.33, the supply of resources is

---

**Figure 4.33:** The production mechanism of SimWar.

**Figure 4.34:** Offensive and defensive units in SimWar.
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.

Figure 4.37 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.

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:
Figure 4.36: A Machinations diagram for SimWAR. It features two players. One player controls the red and orange nodes on the left, the other the blue and green nodes on the right.

Figure 4.37: A chart showing the relative strength of each player as it developed over time during a simulates session eventually won by the red player.
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:

\[
\text{Attack} = \text{Defense} \times 10 - 70 \\
\text{BuyF} = 200 - \text{Factories} \times 100 \\
\text{BuyD} = 100 - \text{Defense} \times 50 \\
\text{BuyO} = 100
\]

The ‘rushing’ strategy proved to be very unsuccessful. Figure 4.38 plots the strengths of both players over a typical 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. Out of one thousand simulated session, the ‘rushing’ strategy managed to win only twice. Most published real-time-strategy games are balanced towards rushing strategies, as the latter 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 4.3). 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 a 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. Also note that increasing the factory costs initially increases the average game length, but decreases it for costs above eight. This can be explained by the fact that increasing factory cost slows the game as it takes more time to build up production capacity. At the same time a very high factory cost favors the rushing strategy, which tends to win faster than the turtling strategy. At higher factory costs the second effect dominates the first effect.

4.12 Conclusions

The Machinations framework formulates a clear theoretical vision on the structure of game mechanics and quality in games. Within the framework, gameplay is an emergent property of the system of rules. Machinations diagrams visualize those structures that directly contribute to emergent gameplay. Quality is attributed to the structure of the game mechanics, or rather to the feedback structures that are present within game mechanics. Up until now, feedback loops in games have been characterized as being either positive or negative.
Figure 4.38: 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.

<table>
<thead>
<tr>
<th>Tweak</th>
<th>Avg. Time</th>
<th>Rush Wins</th>
<th>Turtle Wins</th>
<th>Draws</th>
</tr>
</thead>
<tbody>
<tr>
<td>No tweaks</td>
<td>69.79s</td>
<td>2</td>
<td>997</td>
<td>1</td>
</tr>
<tr>
<td>Factory cost 6</td>
<td>76.33s</td>
<td>7</td>
<td>993</td>
<td>0</td>
</tr>
<tr>
<td>Factory cost 7</td>
<td>88.22s</td>
<td>107</td>
<td>889</td>
<td>4</td>
</tr>
<tr>
<td>Factory cost 8</td>
<td>91.86s</td>
<td>312</td>
<td>676</td>
<td>12</td>
</tr>
<tr>
<td>Factory cost 9</td>
<td>63.58s</td>
<td>557</td>
<td>426</td>
<td>17</td>
</tr>
<tr>
<td>Factory cost 10</td>
<td>68.03s</td>
<td>745</td>
<td>228</td>
<td>27</td>
</tr>
<tr>
<td>Offence cost 1.5</td>
<td>65.60s</td>
<td>184</td>
<td>836</td>
<td>0</td>
</tr>
<tr>
<td>Offence cost 1.4</td>
<td>54.72s</td>
<td>439</td>
<td>561</td>
<td>0</td>
</tr>
<tr>
<td>Offence cost 1.3</td>
<td>45.72s</td>
<td>583</td>
<td>417</td>
<td>0</td>
</tr>
<tr>
<td>Offence cost 1.2</td>
<td>32.73s</td>
<td>806</td>
<td>194</td>
<td>0</td>
</tr>
<tr>
<td>Defence cost 1.5</td>
<td>70.02s</td>
<td>16</td>
<td>983</td>
<td>1</td>
</tr>
<tr>
<td>Defence cost 2.0</td>
<td>74.35s</td>
<td>88</td>
<td>911</td>
<td>1</td>
</tr>
<tr>
<td>Defence cost 2.5</td>
<td>74.12s</td>
<td>196</td>
<td>803</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.3: Tweaks to SimWar’s economy and the effects for ‘turtling’ versus ‘rushing’ strategies.
However, in order to explain gameplay we need more characteristics of feedback. The Machinations framework proposes to characterize feedback loops by their type, effect, speed, range, investment, return and durability. In addition, several types of nondeterministic flows can affect a feedback loop. The delicate interaction between multiple feedback loops must be taken into account to get a complete picture of the dynamics of games as complex machines that generate an internal economy. Patterns such as the ones discussed in this chapter codify structures that have proven to be successful in the past.

Although the Machinations framework utilizes a number of key concepts and terms, it is not a design vocabulary that needs to be expanded in order to include previously unencountered structures. Machinations diagrams work with only a handful of elements which can be combined in infinite constructions to capture just about any game. When these basic elements are understood, a designer should be able to read any Machinations diagram. The framework offers a high range of expressiveness for little investment on the part of the designer.

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, the diagrams can be run: they are interactive and dynamic, just like the games they are modeling. It allows models of games to be tested and explored quickly and efficiently. The tool even allows the designer to quickly gather quantitative data from simulated play sessions, offering a high and concrete return. Unfortunately this dynamic, interactive property of the software tool does not translate to paper; the interactive tool, and many of the examples discussed in this chapter, can be found on the Machinations web page: http://www.jorisdormans.nl/machinations.

The Machinations framework is a design tool first and foremost, but it can be used to record existing and non-existing games equally well. For pragmatic reasons many of the examples discussed in this chapter are models of well-known, existing games: it is hard to show the relation between rule structures and emergent gameplay when the reader is unfamiliar with the latter. In practice, using the Machinations tool allows a designer to simulate and run a design many times before building a prototype. It can also be used to track down flaws and suggest improvements for prototypes and published games. This applications of the Machinations framework should help the designer to get more out of each iteration in a play-centric design process.

The list of feedback patterns presented in appendix B and on the accompanying website is neither definitive nor complete. It is best to consider this framework as a set of building blocks that can be used to build an infinite number of different structures, some of which are recurrent patterns that can be used to analyze existing games and explore new concepts alike.

There are some limitations to the use of Machination diagrams. The idea of internal economy suits some games better than others. In particular, it works very well for board games, strategy games and management simulation games. Games that rely more on level design and mechanics of progression are addressed in the following chapter. In games where economy is more abstract it can be difficult to determine the best level of abstraction and scope for the model, as
is the case for games such as Chess and Go. Those games seem to derive their emergent behavior less from an internal economy and more from the mechanics that govern tactical maneuvering, that fall outside the scope of this dissertation. Many games can be diagrammed in multiple ways depending on the designer’s focus and the diagram’s particular perspective. Still, feedback loops can go a long way in explaining the flow of a game, and should be consistent features even with different levels of abstraction and different perspectives.