Case Study of Petri Nets in Game Design

Manuel Araújo
Departamento de Engenharia Informática
Faculdade de Ciências e Tecnologia
Universidade de Coimbra
Coimbra, Portugal
maraujo@student.dei.uc.pt

Licínio Roque
Departamento de Engenharia Informática
Faculdade de Ciências e Tecnologia
Universidade de Coimbra
Coimbra, Portugal
lir@dei.uc.pt

ABSTRACT

This paper describes an alternate approach to the modeling of game systems and game flow with Petri nets. Modeling languages usually used in this area are of limited efficiency when it comes to validating the underlying game systems. We provide a case study to show that Petri Nets can be used with advantages over other modeling languages. Their graphical notation is simple, yet it can be used to model complex game systems. Their mathematically defined structure enables the modeled system to be formally analyzed and its behavior simulation offers the possibility of detecting unwanted behaviors, loop-holes or balancing issues while still in the game design stage.

Keywords

Game design, Petri Nets, game flow modeling, simulation

INTRODUCTION

Visual communication is an essential part of software design. In game development projects, this sentence remains true. Be it for identifying and constructing use cases, charting the flow between menus, modeling a sequence of events or showing the relationship between actors in the game, diagrams and charts are extensively used throughout the conceptual and design phases of game development [9, 10]. The representations used for this purpose, however, present a number of limitations when it comes to the verification, validation and simulation of the underlying system [11, 17, 18]. This prevents designers from using these diagrams in a more effective way, such as for finding balancing issues or problems within the game's flow.

In this paper, we will explore Petri Nets for modeling a game systems and flow. We will try to show that this approach can be as simple and yet more powerful than other graphical tools used in this domain.

We will begin by analyzing modeling languages used in game design. Next we will describe Petri Nets structure, how they work and some useful extension to their basic structure. In the following section we will develop the case study to show how Petri Net can be used in describing game systems. We will then discuss some of their advantages and disadvantages when compared to the other modeling languages.

MODELING IN GAME DESIGN

Natural language is easy to read, but it is not easy to review a large natural language specification [15]. For that reason we use charts and diagrams to explain things visually and synthetically. This kind of communication can be as expressive as and easier to understand than the verbal descriptions used for the same information [15]. People doing game design make intensive use of visual communication [9, 10, 12, 13, 19]. The flow of the game is usually depicted visually instead of a pure textual approach [10, 13, 19]. We will now take a look at some of the most popular diagrams in this branch of software design.

Unified Modeling Language (UML) is a modeling language widely used in software specification and design. As of version 2.0, it comprises 13 diagrams types, divided into 6 categories [21]. A number of these can be of great use to game design [9]. Use cases are used to represent actors (player, external systems, etc.) and their actions or interactions with the system. They can be used to identify and collect requirements in the early stages of conception. Other diagrams, like sequence, class and activity diagrams, can help game designers further document and structure ideas and later help the programmers structure their code in the development phase.

As proposed in [9], although with a bigger focus on use case diagrams, UML makes the game designer divide his/her project in a number of different kinds of diagrams, each one serving its purpose. Those who are going to read those diagrams have to be able to understand all of them

and cope with possible semantic inconsistencies [18], since they define a range of possible interpretations instead of conveying an exact meaning [18, 20]. UML diagrams, especially when used at the conceptual level, lack formal semantics that prevent them from being used in rigorous model analysis [11, 17, 18]. In large, complex models this can present a problem since validation can be compromised. Although there are tools for analysis and simulation, these use intermediate models like graphs or Petri nets [11] or focus more on syntax and consistency checks [17]. There are also a number of proposals for the definition of formal semantics [11, 17, 20], but none of these are yet part of the UML standard.

Flowcharts can be used to explain the steps needed for a certain action to occur or simply to document the flow of the game [10, 12, 13, 16]. Decision trees are a type of flowchart that branch on every situation where the player can make a decision, effectively mapping all the possible decisions and outcomes in a game [16, 19]. Activity diagrams (as flowcharts) can be used to model sequential and concurrent processes, which can be useful for most games. When decision depends of external processes running in parallel their capabilities are limited and these diagrams become cumbersome.

Petri nets are both a mathematical and a graphical tool for modeling, analyzing and designing discrete event systems used in many different industries [1]. When compared with behavior modeling based on Statecharts, e.g, [15], Petri Nets tend to be, in general, a more economic representation when state-space complexity increases.

In [22], Brom and Abonyi describe a technique for "authoring a nonlinear plot and managing a story" according to this plot using Petri nets. This technique makes use of both the formal and graphical natures of Petri nets for drafting, simulating and building plots for a story-driven virtual reality application. It was later applied successfully to the story manager of serious games Europe 2045 [24] and Karo [23]. We believe that Petri nets can be applied to other areas of game design, not just plot description or story managing.

OVERVIEW OF PETRI NETS

Petri Nets were created by German mathematician Carl Adam Petri for the purpose of describing chemical processes [3]. They can be applied to many different areas, such as the modeling of production lines, distributed-database systems and communication networks or the design and analysis of workflows and business processes, among many others [1, 2]. They can also simulate such processes with the help of a number of computer applications [4, 5, 6].

Due to the fact that Petri nets can be described as a set of algebraic equations [3], they are considered as a powerful analysis tool [1, 2, 7]. Petri nets can be used to check for the existence of deadlocks or starvation, analyze concurrency between processes, precedence relations amongst events or the existence of appropriate synchronization [1]. They can also be used to measure the performance of the underlying system [1, 7].

Petri nets can be described both mathematical and graphically. For the sake of simplicity, this paper will not feature the mathematical description. Most of the Petri net diagrams present in this document were made using Yasper [5]. To illustrate a difference in notation, another diagram was made using Woped [6].

Graphically, Petri nets are described as a diagram with circles (places), bars or squares (transitions) and arrows (arcs) connecting them. Depending on the interpretation the designer wishes to give them, places can represent conditions, input/output data or resources. Transitions can be interpreted as events, tasks or clauses, among others [2]. Places can have multiple arcs from and to transitions and transitions can have multiple arcs from and to places. A transition can have arcs going back to its input places, symbolized in figure 1 by a double arrow. A place can hold one or more tokens, symbolized by one or more dots. Depending on the interpretation given to places, a token can represent resources or whether a condition is true or false [1]. In its basic incarnation, a transition is enabled and can fire when all the places that are connecting to it hold at least one token. When the transition fires, it removes (consumes) a token from all the incoming places and adds (produces) another token in all the outgoing places.

In the example given in figure 1, T1 has one input place

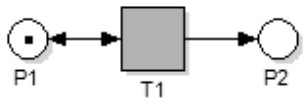

Figure 1: Simple Petri Net with two places and one

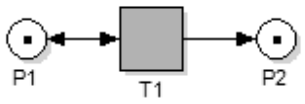

Figure 2: Result after transition has fired

(P1) and two output places (P1 and P2) and it is enabled because all of its input places have tokens. In figure 2, T1 has consumed a token from P1 and produced another in P1 and P2. The transition can fire again indefinitely because P1 has always one token.

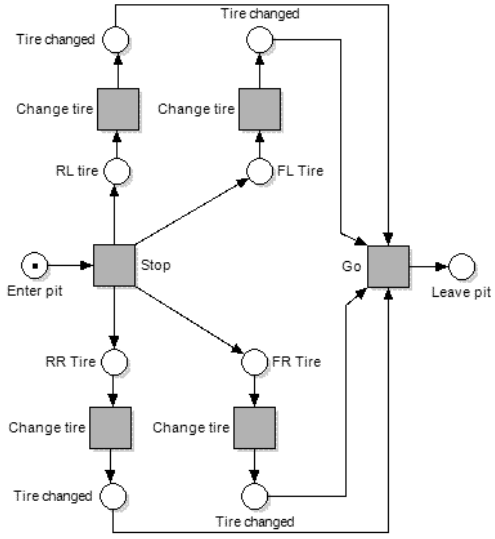

Figure 5: Pit stop. The car enters the pit, stops, has its tires changed and leaves.

More complex behaviors can be modeled by adding places and transitions to the diagram. In figure 3 we have a situation where a F1 car is making a pit stop to change all of its tires. When the Stop transition fires the places representing the tires – front and rear, left and right – receive a token. From that point on, all the tires are changed concurrently. The Go transition can only fire when all the "Tire changed" places that are connected to it hold a token (logical AND). This means that the car can only leave the pit when all the tires are in place.

Several extensions have been developed over the years in order to improve the Petri nets' capabilities or to simplify its

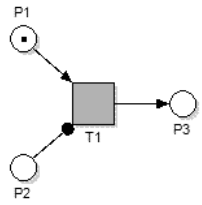

Figure 8: Petri net with inhibitor arc

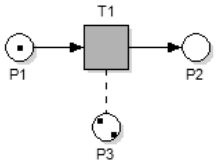

Figure 9: Net with reset arc. When T1 fires, P2 will be emptied.

design and readability.

One of such extensions is the inhibitor arc. This special kind of arc enables a transition when no tokens are present at the input place [1]. It is graphically represented by a line with a small circle where the arrow would be. In figure 4, T1 is enabled since P1 has a token and P2, who is connected to T1 trough an inhibitor arc, is empty.

Sometimes it may also be useful to delete tokens from places when certain transitions occur. When a stop condition is reached and you wish to stop processes running in parallel, for instance. For these cases there is a special kind of arc call the reset arc, usually represented by a dashed line [5]. When the transition connected to the arc fires, the place on the other end is emptied of its tokens.

Another useful extension involving arcs works by associating weights to them. Weights correspond to the number of tokens that must be removed or added from or to a place when a transition fires. The transition cannot fire while its input places do not have a number of tokens equal to the weight of the arc that connects them [1]. In the examples given up to now, we can consider all of the arcs to have a weight value of 1, since the transitions associated to them add/remove a single token to/from the corresponding place.

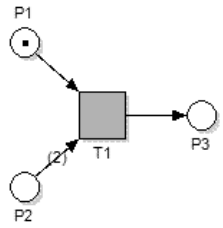

Figure 7: Arcs with weights. When fired, T1 consumes 3 tokens and produces 1

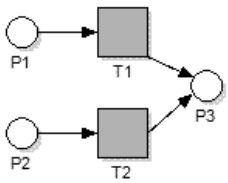

Figure 6: XOR with multiple transitions

Some extensions can simplify Petri nets greatly. In figure 7 we are modeling a logical XOR by using two different transitions. P3 will get a token if T1, T2 or both transitions are fired.

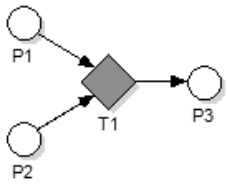

Figure 3: XOR-join transition

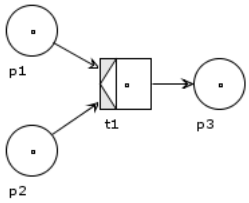

Figure 4: XOR-join transition alternate notation

In figure 8, however, we need only one transition to achieve the same result. Other transitions of this kind include XOR-splits (where we can choose from multiple outgoing places) and mixes of AND and XOR transitions. The notation for this type of transitions varies among Petri net modeling software between the diamond in figure 8 and the boxes in figure 9 [5, 6, 7].

Places can also function as branching points when they have more than one outgoing arc. In automatic simulation this transition is usually picked at random, but one can had guard expressions [8] or different probabilities [5] to transitions or arcs in order to simulate different behaviors.

Transitions can also be timed (timed Petri nets) and tokens can have different values and attributes throughout the net (colored Petri nets) [7].

Hierarchical structuring is a feature shared by many Petri Net design and simulation software [5, 8]. In order to keep the diagrams simple while modeling complex systems, we can divide a complex Petri Net into smaller, hierarchically related nets and spread them across separate diagrams. This makes it possible to model a large and complex system while still being able to explain the way it works in a general, simpler diagram.

Figure 10: Pit stop with hierarchies.

This can be illustrated by taking a look at the pit stop example given earlier in figure 3. We can hide the details of the tire changing process by moving it inside a subnet construct [5, 7].

This results in the diagram in figure 10. This construct takes the shape of a transition that aggregate the places and transitions we had before [7], simplifying both the reading and understanding of the diagram. Since the elements that were in figure 3 are still part of the Petri net, we can simulate the process in the exact the same way as we would if we didn't use hierarchies.

CASE STUDY OF PETRI NETS IN GAME DESIGN

We will now illustrate the use of Petri Nets as a modeling language in game design, the expressiveness of these diagrams and the ability to simulate the flow of the game with them, by resorting to a few examples. The diagrams present in this section are part of a game we are working on. The game's main objective is to promote an experimental and systematic research attitude in a simulated environment, managing knowledge needs and acquisition. The game is inspired by historical accounts of the methods followed for navigation and mapmaking of land coasts and maritime routes during the Portuguese Discoveries. Players play the role of a ship's captain on a mission to explore and chart an unknown world. The game's focus is mainly on the player's ability to evaluate his/hers situation based on how much he/she knows. Careful preparation, good information management and negotiation skills are essential to be able to chart the terrain and progress.

Example 1: Interaction with natives

Throughout the game, players will encounter situations where they will need to negotiate with the natives in order to get supplies or information from them. The way the natives respond to the player's requests is based on their past interactions with the player and the way he/she makes the request. The way envisioned for this process is loosely

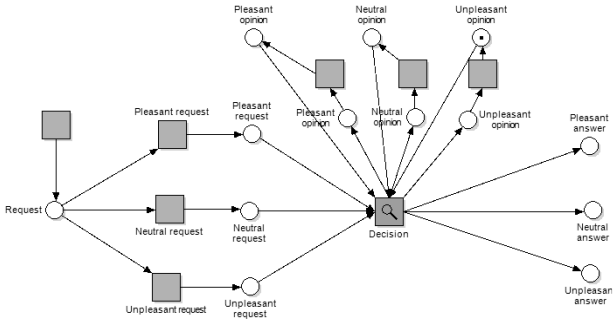

Figure 11: Request-answer process

based on a system present in an old strategy game called *Centurion: Defender of Rome* (developed by Bits of Magic and published by Electronic Arts in 1990). Every time the player enters a region of the map he/she doesn't control, there's an encounter between the player and the region's leader. Based on a number of factors, including the way the player talks to the barbarian leader, you will either end up in a battle, retreating or negotiating an alliance.

In our case, the general interaction process is described by the Petri Net diagram in figure 11. The player starts by making a request to the natives, which can be done in one of three ways: in a pleasant, neutral or unpleasant tone. Based on the way the player made the request and their opinion of him/her, the natives then give one of 3 possible answers, which can also be considered pleasant, neutral or unpleasant by the player (discount, normal price or overprice, for instance). The natives' opinion may change according to their previous thoughts on the player and his/her "tone". The whole decision process is represented in the diagram on figure 11, by the Decision subnet transition. This subnet takes the player's request and the natives' opinion as inputs and outputs the natives' answer and new opinion. The diagram could be further simplified by moving the transition choice for the request into the subnet. Still, we leave it like this to show the kind of requests the player can make at the "parent" diagram.

The place "Unpleasant opinion" already has a token, meaning that the natives' current opinion of the player is negative. When a request is made, both the token from this place and the player's request go into the Decision subnet. In this subnet, seen in figure 12, places marked with an "i" represent external inputs and places marked with an "o" represent external outputs. These correspond to places in the parent diagram. Inputs and outputs can only have one incoming or outgoing arc, so in this case there is an extra transition and place for each one of them. Transitions leading to answers fire according to the distribution of tokens in the request and opinion places. A pleasant answer, for instance, can be the result of a pleasant or neutral request and a pleasant opinion. Both these transitions also result in a positive opinion.

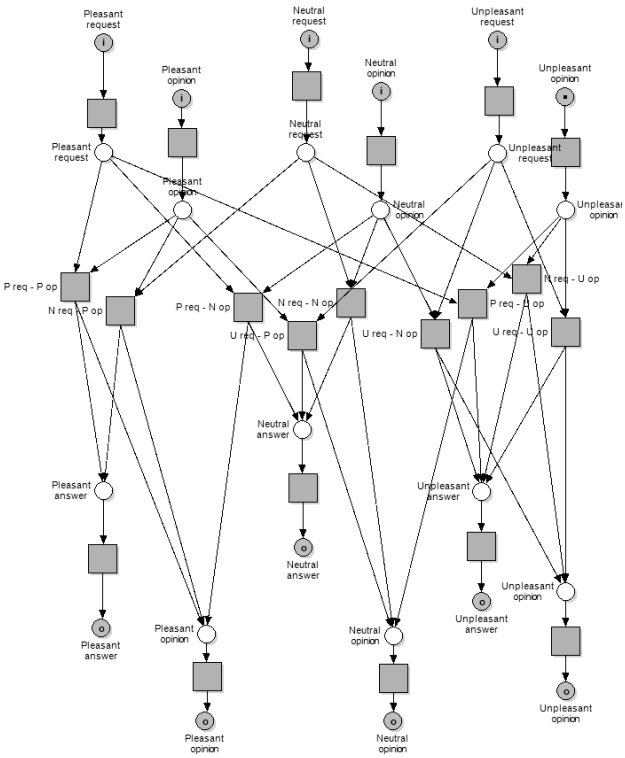

Figure 12: Decision subnet

Since "Unpleasant opinion" already has a token, we can tell by the outgoing arcs that the answer is going to be unpleasant no matter what the player says. Still, he can make the natives change their opinion to neutral by making a request in a pleasant tone. This is symbolized by the "P req - U op" transition. It can fire when both "Unpleasant opinion" and "Pleasant request" have a token. It then goes and deposits a token in "Unpleasant answer" and in "Neutral opinion".

This approach can still be extended by adding the neighboring countries' opinion to the equation. This way, a bad choice on the player's part can have an impact later on in the game by influencing the behavior of natives he/she hasn't even been in contact with. Simulation can be used in order to show how different kinds of requests and opinions lead to different results. It can also be used to count the number of tries the player has to make in order to change the natives' opinion to a positive one.

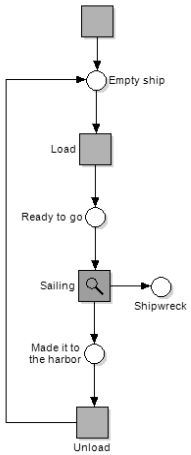

Figure 14: A merchant ship's journey

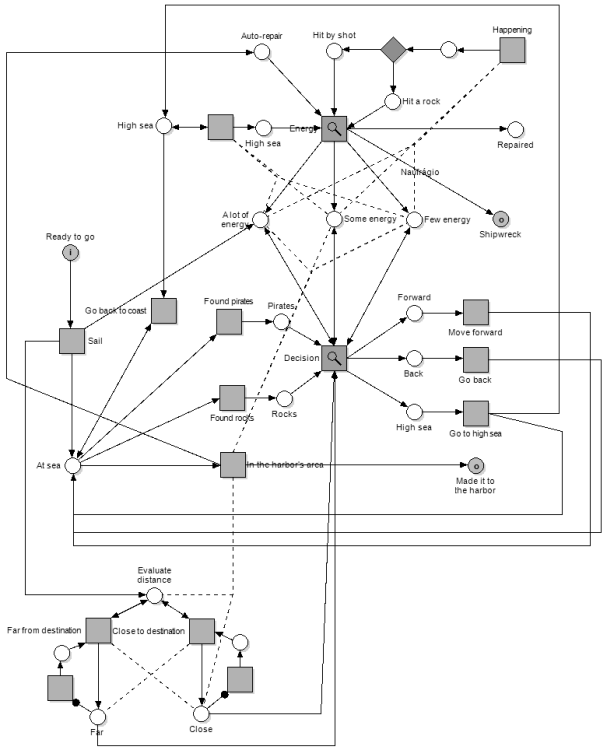

Figure 13: Sailing subnet. Further down the hierarchy there are two other subnets, Energy and Decision.

Example 2: Merchant ship

The merchant ship described here is projected as a computer-controlled unit, even though these diagrams can also easily describe the decision making process of a human player. This unit loads cargo from an harbor and takes it to another friendly harbor, where it unloads and reloads again. The ship is unarmed and cannot repair itself away from the harbor, which will affect the decisions it will make throughout its journeys. The ship starts empty, inside the harbor. After it loads the cargo, it's ready to set sail. Sailing can have two possible outcomes, or in this case, outputs: the ship either gets to an harbor (be it its destination or its starting point) or it is destroyed. Since the sailing aspect is the most complex, it was made into a subnet, in figure 14. Inside this subnet we have the possible actions from the unit, its location in respect to the destination, the game elements that can affect its journey and the decision making process.

To understand what is represented in the Sailing subnet some additional context is needed. In the game you have

two different regions: coastal and high sea. Ships in high sea take damage from the waves, but are safe from pirates and rocks. The sea doesn't damage the ship if it's close to the coast, but pirates and rocks may appear. This being said, the transition next to the place "High sea" should actually be a timed transition in order to simulate this aspect accordingly. Due to software constraints, though, that wasn't possible.

Taking a closer look at the Sailing subnet, while the ship is at sea, it will continuously evaluate the distance to its destination, as well as being in constant alert for pirates or rocks. It also starts its journey with full energy. If it finds pirates or rocks, it will enter the decision process. This process takes the finding, the ship's energy and the distance to it's destination as inputs. The energy is constantly monitored and will change along the trip if the ship moves into the high seas, takes a shot from a pirate or hits a rock. If energy reaches 0, that means the journey has ended in a shipwreck. Every time something happens, the energy that is used as input for the decision process is reset to insure the decision is made with up-to-date values. When the ship reaches the harbor area, it is automatically repaired. When this transition fires, it is important from a simulation point of view to clear both the distance tokens and the energy tokens with reset arcs. This way, when the ship returns to sail, all is back to the original condition.

The energy subnet in figure 15 has three places that symbolize the amount of energy the ship has. "Maximum energy" starts with 5 tokens, "medium energy" with 2 and "minimum energy" with 1. This makes the ship's total energy 10 tokens, since an extra token is added to "medium energy" and "minimum energy" when the transitions leading to them fire.

Inhibitor arcs are placed in the "suffered damage" transitions to insure that tokens are removed from the right

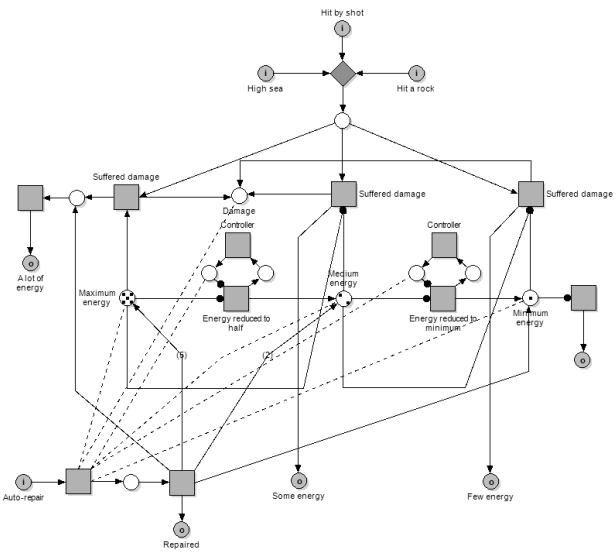

Figure 15: Energy subnet

place. The middle "suffered damage" transition, for instance, is there to remove tokens from "medium energy". That being the case, it's only logical that it can only fire when "maximum energy " is empty. When these transitions fire, they will also add a token to their respective output.

Between the max., med. and min,. energy places, there's a transition with two places named "controller". This is here to insure that the "energy reduced to half" and "energy reduced to minimum" transitions can only fire once, since the condition for them to fire is that the previous energy level is empty. By adding an extra condition, we prevent the transition from firing indefinitely.

When the ship is being repaired, these places are cleared, along with the damage and energy ones. Then, a final transition with outgoing weighted arcs is used to put the initial number of tokens back into the energy places before returning to the parent net.

The energy outputs from this subnet are valuable inputs for the decision making process. For this, like in the previous case, we have a decision subnet. As stated before, if the ship finds pirates or rocks, it will enter the decision process. The Decision subnet outputs the course of action the unit will take according to its position, its energy and the type of threat. If the harbor is close, the ship will keep on going no matter what. This means tokens from other inputs don't count in this case, so they will be cleared with the help of reset arcs.

If it is far, however, both the current energy level and the type of threat have to be taken into account. To simplify the diagram, we grouped this distance with the current energy using three other transitions. Being far and with only a few energy tokens left has the opposite effect of being close to the harbor, but it also doesn't need to know the type of threat, in order to decide what to do next. If on the other hand the unit as a lot or some energy, it might want to analyze the situation a little further. When the final transitions are fired, the output of this subnet is taken to the parent Sailing net, back into the "At sea" place, enabling its transitions once again and restarting this cycle.

We could still model and simulate this whole system without both the decision and energy subnets. These could be replaced by a simple XOR, giving the designer the power to choose what to do next. It is interesting, though, to give some extra detail to models in order to test some concepts.

Balancing issues can be detected in simulation of this system. If the journey ends in a shipwreck too many times, then there can be a problem with the damage or with the way decisions are made. If there are no shipwrecks, though, this can mean that the game is too easy.

ADVANTAGES AND INCONVENIENCES OF GAME MODELING WITH PETRI NETS

Once you know how transitions work, Petri Net diagrams can become easier to read and to understand. It has a simple

notation, being composed of only circles, bars or boxes and at most three types of arcs, when compared with UML's multiple diagram types [21] even if only using some of them for designing game systems [9]. Text descriptions are usually very short, and not all transitions/places need names (e.g., see figure 16).

The use of hierarchies of diagrams also enables them to represent complex systems in a simpler way while preserving or reinforcing semantics of each diagram. We can get the big picture of the game by looking at one or two diagrams, and then move down the hierarchy to understand the way specific things work.

The Petri Nets' ability to model system with concurrent operations and conflicts is well documented [1, 2, 3, 7]. This can be of great help for game designers trying to model real-time action and strategy games. In figure 14, for instance, we are modeling distance evaluation (albeit in a simple way), the ship's energy and what may affect it, its journey and its decision making process all at the same time. In multiplayer online games the complexity imposed by concurrent game events can pose even more complex modeling scenarios.

Being a mathematically structure, Petri Nets can be verified, validated and simulated with a number of analysis methods and tools [1, 7]. The diagrams that were made to model the game behavior can also be used to check for issues in the game design. Balancing the flow of a game is a task that is usually done only after some form of game prototyping enables playtesting [16]. Petri Net modeling gives game designers a way of finding problems before the game is in actual development. This can be especially important for simulating multiplayer or heterogeneous game designs whose dynamic properties are typically more difficult to foresee.

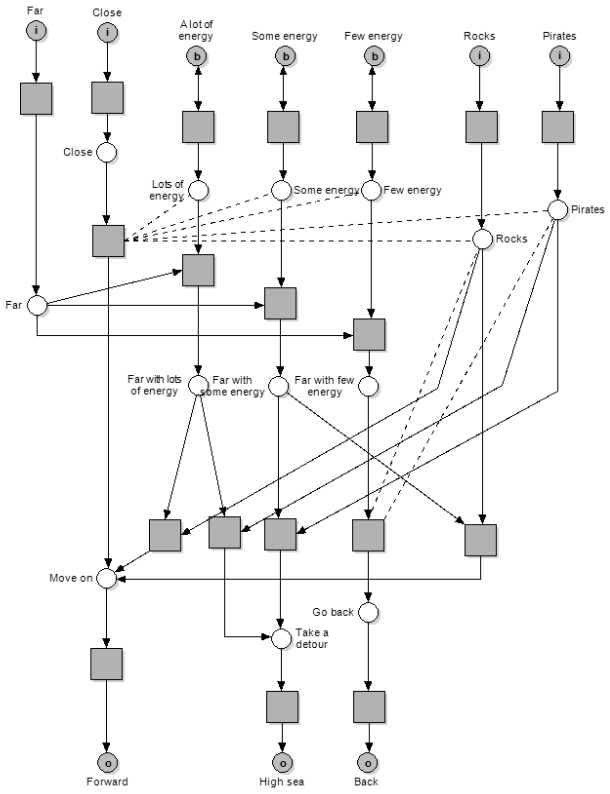

Figure 16: Decision subnet.
This outputs the course of action the unit will take.

Like in most graphical tools, growing complexity can become an issue. Even with hierarchies, some diagrams can grow to become very complex. In figure 12, for instance, the number of lines crossing paths hinders readability, even if the diagram is still fairly simple. Just by adding the neighbors' opinion to the equation, the increased complexity would likely make the diagram hard to read. Moving some sections to subnets can simplify the picture, but it makes it harder to view the whole system, unless each diagram retains a clear semantic role in the whole model

In figure 15 it is implied that cannon shots, rocks and waves do the exact same damage to the ship. From a representation point of view one could argue it doesn't really matter, but in simulation it can lead to awkward results. One could try to use different transitions with weighted arcs to represent different levels of damage. However, if the weight of the arc is bigger than the number of tokens present, the transition would never fire, leaving the ship undamaged. This is a limitation of using simple Petri Nets models that can be partly overcome with the adoption of an extension for colored tokens. This extension provides tokens with variable attributes and transitions that can choose the type of tokens to remove/add and change these attributes. This means that the diagram in figure 15, for example, could be simplified and model different types of damage by adding variables and conditions to tokens and transitions respectively.

CONCLUSIONS

In this paper we discussed the applicability of Petri Nets to the modeling of game systems and game flow. This is an area where Petri Nets can be of value, especially if we are concerned with modeling game scenarios with concurrency. As a graphical tool, we concluded that Petri Nets can be expressive and easy to understand. Their limited number of representation elements contrast with the large array of diagrams and concepts in other notations in popular modeling languages such as UML, making them easier to learn, although limited to describing system behavior. Petri nets can be useful for modeling decision making processes that depend of several preconditions. Simple agent behaviors as well as player's choices can be modeled and simulated. This is enhanced by a Petri net's capability of modeling concurrent operations. Petri Nets have formal semantics that enable them to be verified and simulated. This aspect can prove to be very useful for game designers, since they can evaluate some play time characteristics while

still in the design phase. We are still evaluating if the adoption of colored Petri Nets can help manage the needs for representing more complex situations retaining economy of representation.

ACKNOWLEDGMENTS

REFERENCES

1. Zou, M., and Zurawaski, R. *Introduction to Petri Nets in Flexible and Agile Automation*, Petri Nets in Flexible and Agile Automation. Kluwer Academic Publishers, 1995, 1-42.
2. Murata, T. *Petri Nets: Properties, Analysis And Applications*. Proceedings of the IEEE, Vol.77, No.4, April 1989, 541-580.
3. Petri, C., and Reisig, W. *Petri net* http://www.scholarpedia.org/article/Petri_net (retrieved on 12/02/09).
4. Petri Nets World: Petri Nets Tool Database Quick Overview. http://www. informatik.uni-hamburg.de/TGI/PetriNets/tools /quick.html (retrieved on 15/02/09).
5. Yasper User Guide. Available at http://www.yasper.org/ (retrieved on 09/02/09).
6. Workflow Petri Net Designer. http:// woped.ba-karlsruhe.de/woped/WorkflowNets (retrieved on 12/02/09).
7. van der Aalst, W. *The Application of Petri Nets to Workflow Management*. The Journal of Circuits, Systems and Computers, Vol. 8, No. 1, 1998, 21-66.
8. CPN Tools, Computer Tool for Coloured Petri Nets. http://www.daimi.au.dk/CPNTools/ (retrieved on 09/02/09).
9. Bethke, E. *Game Development and Production*. Wordware Publishing, 2003.
10. Crawford, C. *The Art of Computer Game Design*. McGraw-Hill/Osborne Media, 1984
11. Lian, J., Hu, Z. and Shatz, S. *Simulation-Based Analysis of UML Statechart Diagrams: Methods and Case Studies*. The Software Quality Journal (SQJ), Vol. 16, No. 1, March 2008, 45-78.
12. Lewinski, J. S. *Developer's Guide to Computer Game Design.* Wordware Publishing, 1999.
13. Rollings, A., and Adams, E. *Andrew Rollings and Ernest Adams on Game Design*. New Riders, 2003.
14. Wendt, S. *Modified Petri Nets as Flowcharts for Recursive Programs*. Software - Practice and Experience, Vol. 10, No. 11, 1980, 935-942.
15. Horrocks, I. *Constructing the User Interface with Statecharts*. Addison-Wesley Professional, 1999.
16. Salen, K., and Zimmerman, E. *Rules of Play: Game Design Fundamentals*. The MIT Press, 2003.
17. Aredo, D. B. *A Framework for Semantics of UML Sequence Diagrams in PVS*. Journal of Universal Computer Science, Vol. 8, No. 7, 2002, 674-697
18. Schattkowsky, T., and Forster, A. *On the Pitfalls of UML 2 Activity Modeling*. IEEE Computer Society, International Conference on Software Engineering, Proceedings of the International Workshop on Modeling in Software Engineering, page 8, 2007.
19. Crawford, C. *The Art of Interactive Design: A Euphonious and Illuminating Guide to Building Successful Software*. No Starch Press, 2002.
20. Szienk, M. Formal Semantics and Reasoning about UML Class Diagram. IEEE Computer Society, DEPCOS-RELCOMEX, Proceedings of the International Conference on Dependability of Computer Systems, 2006, 51 – 59.
21. Unified Modeling Language. http://en.wikipedia.org/wiki/Unified_Modeling_Language (retrieved on 17/02/09).
22. Brom, C., and Abonyi, A. Petri-Nets for Game Plot. Proceedings of AISB, Vol. 3. 6?13. AISB press.
23. Balas, D., Brom, C., Abonyi, A., and Gemrot, J. Hierarchical Petri Nets for Story Plots Featuring Virtual Humans. Proceedings of Artificial Intelligence and Interactive Digital Entertainment 2008, pages 2-9.
24. Brom, C., Sisler, V. and Holan, T. Story Manager in 'Europe 2045' Uses Petri Nets. International Conference on Virtual Storytelling 2007, pages 38-50.