Hierarchical Petri Nets for Story Plots Featuring Virtual Humans

Daniel Balas, Cyril Brom, Adam Abonyi, Jakub Gemrot

Charles University in Prague, Faculty of Mathematics and Physics, Department of Software and Computer Science Education
Malostranske nam. 25, Praha 1, 118 00, Czech Republic
addabis@gmail.com, brom@ksvi.mff.cuni.cz, adam.abonyi@gmail.com, jakub.gemrot@gmail.com

Abstract

Petri Nets can be used for a retrospective analysis of a computer game story, for representing plots in serious games as well as for monitoring the course of the story, as recently demonstrated by several authors. It was not clear, however, whether Petri Nets can be used for representing plots and for their unfolding in games that feature large worlds inhabited by virtual humans. This paper demonstrates that this is indeed possible, presenting several scenarios from the serious game *Karo*, an on-going project, which features both virtual humans driven by reactive planning and a story manager unfolding story plots represented and controlled by Petri Nets. However, a specific refinement of Petri Nets, a hierarchical model capitalising on Timed Coloured Petri Nets and non-deterministic FSMs, had to be developed for this purpose. This refinement is described here, including general discussion showing why individual features of this refinement are needed – directions for game developers considering whether to use this technique, or not.

Introduction

One useful way of conceiving control architectures of computer games emphasising a story, or storytelling applications in general, is the two-layered "individual characters—story manager" conceptual framework (e.g. Szilas 2007; Fig. 1). This is basically another reincarnation of a "divide and conquer" mechanism. Capitalising on this distinction, a designer can easily distinguish in his or her mind between the issues of a) specifying background behaviour of virtual characters and objects in general, and b) a high-level plot that is to clothe this simple behaviour in a narrative, e.g. by altering top-level goals of the characters. Distinction between a world state and a story state can be made, the former being represented within a world simulator, the latter by the story manager. Further, some issues can be easily solved when one adopts the central view offered by a story manager abandoning the notion of strong autonomy, which pops up naturally when thinking at the level of individual characters, e.g. the issue of synchronising during joint-behaviour (see e.g. Mateas 2002).

This paper concerns itself with the question of how to represent plots of games featuring middle- or large-size virtual worlds inhabited by virtual characters, for instance RPGs, i.e. with the "story manager layer". Recently, there has been huge interest in planning techniques (to mention just a couple of works: Reidl and Stern 2006; Cavazza et al. 2002; note that not all of them employ the two-layered framework). Basically, planning has opened the field of automatic story construction; being largely inaccessible by reactive methods, it helps to avoid the problem with combinatorial branching during unfolding of a story, and it also offers new possibilities in solving "narrative-interactive" tension. Nevertheless, planning is unlikely to dominate in the field of gaming AI in the next decade, especially for games featuring middle- or large-size worlds. Until one luckily proves P=NP, the limited computational resources will be the stumbling block. Another, sometimes overlooked, fact is that often a pre-specified story is needed.

Reactive techniques have been used for years both for controlling virtual characters as well as unfolding the story. Their disadvantages, which motivated the use of planning at the first place, are well known. However, for large worlds, we have no other choice. A branch of reactive techniques widely used at the story manager layer is deterministic finite state machines (dFSMs; e.g. Sheldon 2004; Silva et al. 2003). Sometimes, so-called branching graphs or branching trees are used; though they are not dFSMs literally, they are very similar conceptually. Generally, dFSM can be also regarded as a type of rule-based systems, but because even STRIPS-like planning can be realised within a rule-based system, it is more useful to think in terms of dFSMs then in unconstrained rules—dFSMs better shape the thinking. However, it was argued that dFSMs cannot cope well with representing plots for large worlds (Brom et al., accepted). Most notably, there is

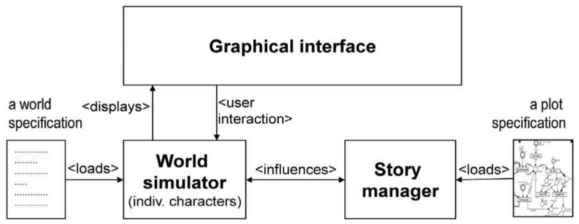

Fig. 1. Two-layered control architecture of a "typical" storytelling application. Simulator of the virtual world features individual characters and controls their background behaviour. This behaviour is modified for the narrative purpose by the story manager. Each component uses its own representation.

a problem with parallelism; stories can evolve in parallel, they can be even perceived in parallel (think of a multiplayer RPG for instance); and another problem consists in specifying how the story manager can influence the world and vice versa, i.e., in specifying the "story manager—world" interface.

Petri Nets are another technique, which, conceptually, is a sort of rule-based system. It was proposed that they could cope with these issues (Brom and Abonyi 2006). However, to our knowledge, they were never employed as a plot representation technique and a mechanism for evolving a story in a large game featuring virtual characters. Hence it was not clear whether they can be really used as proposed. Petri Nets were used for specification of behaviour of individual characters (e.g. Blackwell and von Konsky 2001). Natkin and Vega (2003) used them to perform a retrospective analysis of a computer game story. They were used for plot monitoring in a simple game (Delmas et al. 2007). In our previous work, we used them for prototyping purposes (Brom and Abonyi 2006), and as a technique for representing plots in the serious game Europe 2045 (Brom et al. 2007a), a statistical coarse-grained simulation without virtual characters.

Recently, for our on-going project, a serious game Karo featuring a middle size world inhabited by virtual humans, we have decided to use this technique. So far, we have (1) created a novel Petri Nets refinement, a hierarchical model (PN-model in the further text), (2) developed a Petri Nets engine unfolding stories based on plot specifications given in the PN-model, (3) designed and implemented an interface between this engine and a simulator of virtual worlds, through which the story manager influences the world and vice versa (see again Fig. 1), (4) prototyped several scenarios to verify that the method works. In these scenarios, background behaviour of virtual characters is driven by hierarchical if-then rules, while their high-level goals (and the state of the world in general) are altered by the story manager with the engine underpinned by the PN-model.

The goal of this paper is to demonstrate this technique and to discuss the requirements demanding its use. The matter is that on one hand, the PN-model is quite robust and can be applied to a greater variety of story plots than canonical Petri Nets or dFSMs, but on the other hand, the PN-model loses intuitiveness of these techniques. Actually, while the PN-model stems from Petri Nets, it also has a taste of non-deterministic hierarchical finite state machines (nHFSM) and general reactive rule based systems, being a hybrid mechanism of its own sort, understanding and implementing of which can be demanding. Thence, the objective of this paper is not only to show how a decades old technique, Petri Nets, can be used for story managers, giving the game designers something that has advantages over dFSMs and is ready to use for practice (as opposed to a planning approach), but also to provide some hints when to use it and when not to.

The rest of the paper proceeds as follows. First, Petri Nets are overviewed in general. Next, the requirements imposed on the PN-model are detailed. Now, the PN-model is introduced: first, the details of its usage are described from the designer's perspective, and second, its formal specification (i.e. for the purposes of programmers) is given. Finally, example scenarios are described.

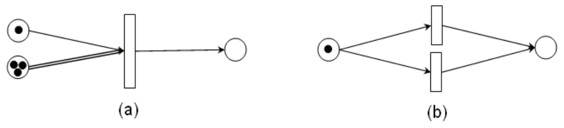

Fig. 2. Petri Nets examples. a) The action generates one token if there is one token in the upper container and two in the lower container. b) The two actions are in conflict. Had there been two tokens in the container, the actions could run in parallel.

Petri Nets

Petri Nets is a specification technique frequently used in software engineering. A basic Petri Net variant consists of containers (or places, represented by a circle: ○), tokens ("the pellets": ●), actions (or transitions, ▭), and a transition function (⟶). The containers contain the tokens. If a sufficient number of tokens is contained in specific containers, an action is triggered. After firing an action, the tokens that helped to fire this action are removed, and some new tokens are generated (Fig. 2a). Which tokens fire which action and which action generates tokens to which containers is specified by the transition function (and depicted by arrows). At one instant, several containers can contain tokens, which allows for concurrent triggering of actions. Obviously, a conflict between two applications of the transition function may appear (see Fig. 2b). Such conflict can be solved in various ways, e.g. using priorities. The basic kind of Petri Nets can be extended by introducing different types of containers, tokens, and transition functions. For example, tokens can have a state, a modification typically called Coloured Petri Nets ("colour" meaning "state"). For more thorough introduction to Petri Nets, we recommend the reader to consult (Petri Nets World, 2007).

The advantages of Petri Nets in storytelling have been discussed in depth in (Brom et al., accepted). For brevity, only the main benefits will be recapitulated:

a) Petri Nets are formal (i.e. exact), and yet graphical, which makes them easily intelligible. (This holds for dFSMs too, but not for rule-based systems in general; and as already suggested, the intelligibility of the PN-model is also disputable.)
b) The story plots can be branching (this holds for dFSMs too).
c) The episodes can happen in parallel. For example, in an RPG, Petri Nets can represent naturally that one episode is to be triggered in village *A* while another in village *B*, independently, but at the same time. (This is the main advantage of Petri Nets over a dFSM, which can be just in one state, and each state can represent just one episode.)

d) Petri Nets fit well for the description of large plots, especially if hierarchical nesting is exploited (dFSMs can be also hierarchically nested, but large plots typically demand parallel episodes – see Point c).

On the other hand, as already mentioned, Petri Nets fit naturally for stories with pre-scripted plots, with limited aspects of emergent narrative, as opposed to purely emergent narrative, or automatic story construction. They also do not address the problem of combinatorial explosion of possible story branches explicitly; this must be solved by the designer in an *ad hoc* way similarly to the dFSMs.

Karo game

Karo is a simulation featuring virtual humans intended as a serious game for civics. The motivation is to help students to understand the dynamics of social relationships. Presently, the game should be regarded as a research prototype, not a full-fledged application. For prototyping purposes, we use a dummy case-study scenario, which has no educational aspect. For the purpose of this paper, it is important that the scenario features 7 characters tied up by the story plot.

Technically, Karo is built upon Java framework IVE (Brom et al. 2007b) serving as a world simulator. During the work presented here, the IVE has been extended with a story manager with an engine using the PN-model. The IVE features grid worlds only; however, this is sufficient for our present purpose. The IVE was intentionally designed for simulations of large virtual environments inhabited by tens of virtual humans. Conceptually, the characters are driven by BDI architecture (Bratman 1987), that is implemented using fuzzy if-then rules. The IVE also uses level-of-detail technique for simplifying space and behaviour of objects and virtual humans at the places unimportant at a given instant (Brom et al. 2007b). Nevertheless, this feature has not been used in Karo.

Narrative in *Karo*. The narrative in the case-study scenario is partly emergent and parallel, very character-oriented and goes into the detail. The story contains many actors; some of them being background characters completely driven by world simulator and never directly influencing the story (e.g. inhabitants of a village, who seem to have their own lives). On the other hand, behaviour of main characters can be influenced by the story manager. We decided to solve “interactive—narrative” tension by limiting the degree of interactivity available to a user. The user is allowed to participate only by changing emotions, moods, and relationships of the characters, and by modifying the environment slightly, e.g. by moving objects. Nevertheless, these small influences can change the outcome of the story radically. Our working hypothesis is that this approach has a strong educational potential because it allows a student to perceive that small, seemingly similar actions can have totally different outcome; however, this issue remains to be investigated.

The scenario is a dummy fantasy set taking place in a fictional medieval town and in the surrounding rural area. It features seven main characters and several background actors. Importantly, characters can be located in different parts of the world at the same time. In the beginning, Anne Greyfox, a maiden, who is the daughter of a local baker, Bryant, is visited by her friend, Nerys Robertson. After a short conversation Anne and Nerys decide to go to the Lonely Tree, a romantic place outside the town. Possibly they are warned about the place by Bryant, which may influence their caution later.

The girls go through the town, finally coming to the house of Aunt Dawson, a goodhearted old woman. She asks them if they can take a lunch to her son Timmy, a city guard. After they agree, Jonas, Nerys’ little brother, comes here. He is a spoiled child, fattish and rude. He tells the girls that he is coming with them. Nerys, depending on her mood, either ignores him or forbids him to follow them. He, however, heard where the girls were going, so if he is not allowed to follow them, he decides to go to the Lonely Tree on his own. He does not know that they are going to Timmy’s watch post first, which is quite a detour, so he arrives to the Lonely Tree sooner.

Near the Lonely Tree, local villain and murderer, Rob The Robber, is hiding. When the girls arrive, they see him and naturally he decides to kill them. In the end, the girls are either alive, and Rob the Robber is dead, or other way round, and also Jonas is either alive or dead, depending on how the user was influencing the story and on a random element—someone who is passing by the Lonely Tree can hear Rob’s voice and go to the town for help.

Requirements, Issues, Solutions

It is important to realise that a game featuring virtual characters and a middle- or large-size world has a different requirements than a game either with a small world (e.g. *Façade*) or with a large world but without virtual characters (e.g. *Europe 2045*). This section elaborates on this issue, summarising these requirements, and outlining conceptual solutions. The PN-model presented in the next section has been designed to accord with these solutions.

(1) Given the story can be large and can evolve in parallel, there is a need for structuring the story to help the author to organise it and the player/s to orient within it.

Conceptual solution: We decompose a story plot hierarchically. We call a description of a part of a story a *plot*. Each plot can be divided into a group of so called *subplots*. A *plot* is a *superplot* of all of its subplots. For each story, there is only one plot without a superplot; a *root plot*, which represents the whole story. Every other *plot* has exactly one superplot.

We say that a plot is *active* in a certain moment if the story manager currently evolves the story using this plot. When a plot is active, its superplot is also active. A plot is *finished* when it was active in the past. A finished plot can be reactivated in the future again. Two plots are *mutually exclusive* when they cannot be both active at a time. Plot *A* is a *successor* of *B* when they are mutually exclusive and *A* becomes finished in the moment *B* becomes active. We denote (*B*, *C*) *succession options* of plot *A* when plots *B*, *C* are of the same depth and both are successors of *A*.

The described structure resembles a hierarchical non-deterministic FSM, an idea elaborated in the next section.

(2) When separating high-level story logic from low-level if-then rules driving the actors, a natural question pops up, where the low layer ends and the higher layer begins.

Solution: The question is immensely complex; we have just adopted an intuitive heuristic, which proved useful:

(2a) Characters. Stories usually focus only on few important characters. Other characters have only a small role, either they become important during a small episode (e.g. Aunt Dawson), or they have no importance for the story, constituting only a believable background. Capitalising on this distinction, when having a root plot, we say that a character can be either *important* or *unimportant* in each of its subplots. Importance is an inherited property, meaning that if a character is important in a plot, it is important in all its subplots. We work with three types of characters. (a) The *main character* is a character important in the root plot. (b) A character is *temporarily important* (t-important) if it is important in some plots and unimportant in others. (c) A character is *minor* if it is not important in any plot.

Let *A1* be a plot, where character *C* is unimportant, and let *A2* be a subplot of *A1*, where *C* is important. When *A2* is activated, we say that character *C* is *promoted*; when *A2* ceases to be active, C is *demoted*. While all characters need to be represented by the world manager, only the main characters and the promoted t-important characters need to be represented by the story manager. Conceptually, the *promotion* of a character is a process of finding a suitable unimportant character, which will become important for the duration of a given subplot; by analogy, a character becomes a "value" of a "character variable", an issue elaborated further in Point (3). Dividing the characters into described groups is a major step of the story design.

(2b) Behaviour. We distinguish between daily, routine behaviour, and behaviour being unfolded for the purpose of the story: typically an unusual behaviour with a dramatic impact. While the former is fully controlled by the world simulator, the latter is controlled by the story manager. It follows that only the important characters can perform the latter. For instance "Anne is doing the housework" would be the part of the world specification, whereas the decision

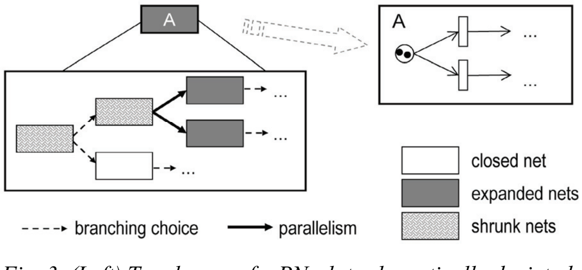

Fig. 3. (Left) Two layers of a PN-plot schematically depicted. Notice the "between subnets" parallelism. (Right) The root net is detailed; notice the "within-plot" parallelism.

"would Bryant warn the girls?" is important to the story, so it has to be a part of the plot specification. Note, however, that even unimportant characters can influence the story since they can interact with important characters on the world manager level, which is an aspect of emergent narrative.

(3) There is a question how the story manager—world simulator interface should look like. E.g. how the story manager can cause "Rob Robber emerges from the shadows after all important characters arrive at the scene"?

Solution. Basically, we allowed the story manager to monitor the state of the world, and based on this state, to *trigger* an action that changes this state. Importantly, this necessitated a) creation of a language for describing states of the world that would work with variable quantification ("all" in the example above is actually the universal quantifier), b) representing objects and characters for the purpose of the story manager, i.e. for questioning features of these characters ("position of all characters is the scene *X*"), and changing them (the top-level goal of R. Robber is changed to from "hiding" to "murder"). We specified a sort of *deictic* variables that points from the story manager layer to the virtual world layer, as detailed in the next section.

Petri Nets in *Karo*

This section describes the PN-model we have developed to address the requirements of story plots of Karo. One way of thinking about the PN-model is that it is (roughly) a hierarchy of interconnected non-deterministic FSMs, in which each state represents a subplot and encapsulates one Petri Net, in which actions can, apart from other things, start evaluation of other non-deterministic FSMs (Fig. 3). Additionally, tokens in these Petri Nets can carry added information, and the transition function of the Petri Nets is generalised.

The PN-model can be employed in two ways; first, as a design tool for a game designer, second, as a formal underpinning of the story manager. These two perspectives will be detailed in turn. Additionally, on several implemented subplots from *Karo*, it will be demonstrated how several particular features that are hard to address by canonical Petri Nets and dFSMs can be realised by the PN-model, namely a) parallelism, b) demoting/promoting of a character, c) queries of the story manager on the story world using variable quantification.

PN-model from the perspective of a designer

We have adopted the following design process: A) a designer writes a narrative (as is the case of the scenario of *Karo* described above), B) based on the narrative, s/he creates a plot specification in a semi-graphical language we developed that capitalises on the PN-model (this language is not detailed here for brevity), C) this specification is converted manually by a programmer into the code (an authoring tool is a future work). In parallel, D) the designer describes the background behaviour of characters, which is

E) converted by the programmer into reactive if-then rules. This section briefly outlines a methodology we have developed to and found useful for Point B. Basically:

(i) The whole narrative becomes the root plot. The designer decomposes it into subplots. These can be further decomposed to smaller subplots, until atomic plots are reached. An atomic plot is such a plot that can be described by, more or less, a *single* sentence from the original narrative that does not make a sense to decompose further (if one finds that a sentence should be decomposed, one should do it in the verbal narrative at the first place), e.g. "The girls go through the town, finally coming to the house of Aunt Dawson."

(ii) Plots at each level of the hierarchy are organised by AND, OR, and XOR connectives, that is, succession options are created.

Finally, the designer will end up with the narrative parcelled out into single sentences connected in a much like nFSM way (or activity diagram, if one prefers the UML terminology), each original sentence becoming an atomic subplot. Fig 4a shows the subplots of the root plot of the Karo story. Compare it with Fig 4b detailing the countryside subplot. While 4a is a simple non-branching FSM, 4b shows both branching and parallelism. Note the inherent combinatorial explosion problem: though hierarchical decomposition helps to manage the story design, one still has to specify each individual story branch on his/her own.

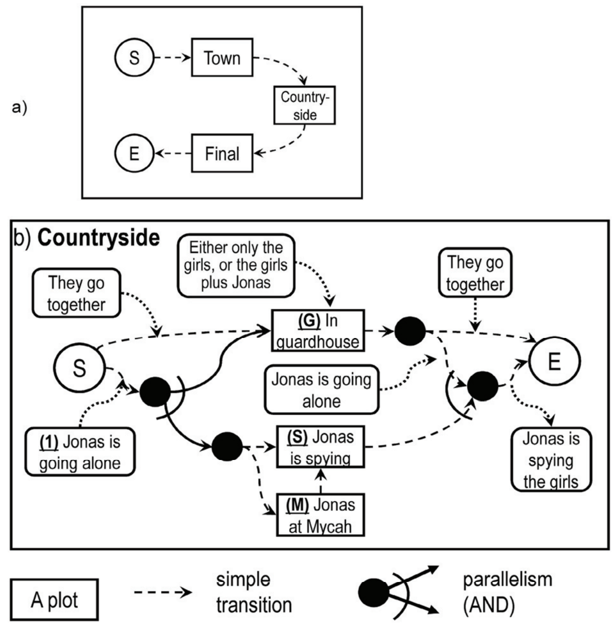

Fig. 4a. High-level structure of the Karo story is depicted. 4b. The countryside plot is detailed. All of the subplots depicted are atomic. The description of each subplot is shortened for brevity (i.e. no whole sentences). The plot branches just after it starts. If (1) holds, the story will evolve in parallel (G and (S or (M+S)).

PN-model from the perspective of a programmer

Point C above states that a programmer has to convert the designer's specification into the formal plot description for the story manager, i.e. describe the plot using the PN-model formalism (using XML in our case). While the gross hierarchical structure can be converted straightforwardly, each verbal description embedded within a subplot of the designer's specification has to be replaced by a Petri Net. The description of the PN-model follows.

PN-Plot. Let *Nets*, *Containers*, *Tokens* and *Triggers* denote the set of all nets, containers, tokens, and triggers, respectively, used in a given plot. A *net* is a quintuple $(C, T, s, M_0, S) \subseteq Nets$, where:

- $C \subseteq Containers$ is a set of *containers* of this net.
- $T \subseteq Triggers$ is a set of *triggers* of this net.
- $s \in \{closed, expanded, shrunk\}$ is a state of the net.
- $M_0 \colon C \to P(Tokens)$ is an initial marking of the net.
- $S \subseteq Nets$ is a set of *subnets*. A net is a *supernet* of all $s \in S$. One net has zero supernets, a *root net*, all other nets have exactly one supernet.
- If a net is *closed* or *shrunk*, all of its subnets are of the same state. If a net is *expanded*, its supernet is also *expanded*. Initial state of each net is *closed*, except for the root net, which is always *expanded*. The typical change of the state of a net is: *closed → expanded → shrunk*, but if needed, the net can be expanded several times.

The definition formalizes the notion of plots as discussed in the previous section, Point (1), and also underpins the graphical language for designers. The root plot is represented by the root net. Closed nets represent plots that have not been active yet, expanded nets correspond to active plots, and shrunk nets to finished plots (Fig. 3, left).

At one level of the hierarchy, after a net becomes shrunk, a choice which net to expand next can be made, a method for representing branching stories. This is the classical dFSM approach. We extend it; at one level of the hierarchy, more nets can be expanded, an idea borrowed from non-deterministic FSMs and a method for representing parallel stories (events specified by those two nets can be happening simultaneously; see again Fig. 4b). Later, we will see that there is yet another method allowing for the parallelism.

Tokens. A token is a quadruple $(i, c, a, p) \subseteq Tokens$, where:

- i is a unique identifier of the token
- c is a colour of the token
- a is an age
- p is an object parameter.

A token is the main information unit in the Petri Nets. Generally, the colour is used as a categorisation of information stored in the token, a reminiscence of the concept of colour in the Coloured Petri Nets (actually, we have developed a typing mechanism common in higher programming languages, which uses colour as a type). A token's age enables Petri Nets to perceive time, a notion coming from Timed Petri Nets. The object parameter is stemming from Requirements (2a) and (3), it presents a method for making an unimportant character temporarily important, i.e. for selection of a character into a certain role, which is important in a given subplot. The object parameter binds a character with a token: it is a deictic variable pointing to the character (or generally to any story-important object). This is an extremely important mechanism enabling the story manager to perceive temporarily some objects from the virtual environment, and change their properties if needed.

Each token must be located in exactly one container.

Containers. A *container* is a tuple (i, C), where i is a unique identifier of a container, and C a set of tokens the container contains. The initial state of all containers of a given net is described by the function M_0 of this net.

Story state. The *story state* is given by a combination of states of all nets, binding of tokens with physical objects and assigning tokens to containers.

Atomic triggers. An *atomic trigger* is a method for stating "if the world and story is in valid state, we can do some action", a mechanism derived from the transition function *plus* actions of canonical Petri Nets. Let S denote the set of all possible story states, and W the set of all possible world states. Then, an atomic trigger is a pair (c, a):

- $c : W \times S \to \{0; 1\}$ is a boolean *trigger condition*.
- $a : W \times S \to W \times S$ is a *trigger action*.

We say that a trigger is *applicable* iff its condition is met, and that a trigger *fires* when the action is being executed. The applicability of a trigger does not necessarily mean that the action is executed.

Note that a trigger changes both the current story state *and* the world state. Hence, triggers are able of working in both important directions: the world state can influence the story state and vice versa. Moreover, the story state can influence itself, which is a natural property, and the world state can influence itself. The latter, however, should not be described by the triggers, but representations held by the world manager.

Note also, that it is possible to define an atomic trigger with *true* condition, hence always firing, or with an action that is *identity*, i.e. has no effect. These triggers act as conditions and unconditional actions together with other triggers *inside* so-called composite triggers, as described later.

How exactly is specified a condition and an action of a trigger depends on the trigger's type. There are five basic types of atomic triggers that work only with the PN-model and thus only with the story state with an obvious meaning: MOVE_TOKEN trigger, DELETE_TOKEN trigger, CREATE_TOKEN trigger, EXPAND_NET trigger, and SHRINK_NET trigger. After a net is expanded, tokens are generated using M_o. Firing of the SHRINK_NET trigger deletes all tokens from the shrunk nets, stopping a sub-plot in effect.

Additionally, we use about 10 atomic triggers handling various elements of the virtual world; i.e. they present the window through which the story manager monitors the world. While some of them are implementation dependent, others basically change *object parameter* of a token or change properties of an object bound to a token in this parameter.

Importantly, the triggers present the second mechanism for representing parallelism (see Fig. 2b; 3/Right). Now, we are confronted with the "with-in" plot parallelism, as opposed to the first method described above, which is concerned with parallelism among plots.

Composite triggers. Currently, we have six composite triggers. Three are roughly similar to logical connectives and the other three are roughly similar to quantifiers. *Connective triggers* simply take a given set of triggers (either atomic or composite) and compose their conditions and actions based on a logical operation.

- CONJUNCTIVE trigger $And(\tau_1, \ldots, \tau_n)$ has a condition $c(w, s) \coloneqq c_{\tau_1}(w, s) \wedge \ldots \wedge c_{\tau_n}(w, s)$ and an action $a(w, s) \coloneqq (a_{\tau_1} \circ \ldots \circ a_{\tau_n})(w, s)$. The sign denotes an action composition function, which is discussed below.
- DISJUNCTIVE trigger $Or(\tau_1, \ldots, \tau_n)$ has a condition $c(w, s) \coloneqq c_{\tau_1}(w, s) \vee \ldots \vee c_{\tau_n}(w, s)$ and an action $a(w, s) \coloneqq (a_1 \circ \ldots \circ a_i)(w, s)$, where $a_1, \ldots, a_i$ are actions of those $\tau_1, \ldots, \tau_n$ that are applicable.
- NEGATION trigger $Not(\tau_1, \ldots, \tau_n)$ has a condition $c(w, s) \coloneqq \neg c_{\tau_1}(w, s) \wedge \ldots \wedge \neg c_{\tau_n}(w, s)$ and an empty action. This trigger has to be used in composition with triggers that produce an action.

A *quantification trigger* takes one trigger (either atomic or composite) as an argument, and additionally a variable name, and a finite domain (for instance, a domain can be "all main characters"). This variable is substituted in the trigger's argument for every value in the domain:

- UNIVERSAL QUANTIFIER trigger $All(\tau, var, dom)$ has a condition
  $$c(w, s) \coloneqq \forall \alpha \in dom{:}\, c_{\tau[var:=\alpha]}(w, s)$$
  and an action
  $$a(w, s) \coloneqq (a_{\tau[var:=\beta_1]} \circ \ldots \circ a_{\tau[var:=\beta_i]})(w, s)$$
  where $dom = \{\beta_1, \ldots, \beta_i\}$. This trigger is used when we need to fire the trigger τ for all members of the domain.
- EXISTENCE QUANTIFIER trigger $Exists(\tau, var, dom)$ has a condition
  $$c(w, s) \coloneqq \exists \alpha \in dom{:}\, c_{\tau[var:=\alpha]}(w, s)$$
  and an action
  $$a(w, s) \coloneqq (a_{\tau[var:=\beta]})(w, s)$$
  where $\beta \in dom$ and $c_{\tau[var:=\beta]}(w, s) = 1$. If the trigger τ is applicable for more values, one is chosen randomly to fire.
- A trigger $Foreach(\tau, var, dom)$ has a condition
  $$c(w, s) \coloneqq \exists \alpha \in dom{:}\, c_{\tau[var:=\alpha]}(w, s)$$
  and an action
  $$a(w, s) \coloneqq (a_{\tau[var:=\beta_{i_1}]} \circ \ldots \circ a_{\tau[var:=\beta_{i_1}]})(w, s)$$
  where $\beta_{i_1}, \ldots, \beta_{i_j} \in dom$ is a maximal set of values satisfying the condition $(c_{\tau[var:=\beta_{i_k}]}(w, s) = 1) \forall k \in \{1, \ldots, j\}$. In other words *Foreach* is like the existence trigger, but it fires τ for each value from the domain for which τ is applicable.

These definitions introduce the language demanded in Req. 3. There are two problems remaining: First, changing the order of actions in compositions may change the outcome. Second, two triggers can collide (see Fig. 2b again).

However, solving these issues on a rigor basis is out of scope of this paper. We think that if the story is carefully designed, these conflicts can be avoided. In (Brom et al. 2007a) we actually introduced a mechanism for solving the conflicts, but it was avoided by the designer: he opted to solve the conflicts on his own.

As a sidenote, we have two auxiliary triggers with special behaviour, which is roughly similar to metaprogramming—ONCE trigger can be applied just once, FIRED trigger on the other hand is bound to another trigger, and is applicable iff this trigger was applied at least once. When composed with another triggers, FIRED trigger enable the programmer to express the situations where the application of some triggers need to be ordered. This mechanism has turned out to be needed quite often for the plot specifications. Formal description of this behaviour is out of the scope of this paper; simply said it is possible to emulate it by adding containers and tokens, but it is more convenient to use these triggers.

The quantifier triggers push the PN-model from the original Petri Nets towards a general reactive rule-based system. This has two negative consequences. First, these triggers slow down the evaluation. Second, they cannot be easily portrayed graphically, as opposed to simple transition function from original Petri Nets. Thence, a designer can specify graphically only the gross structure of the plot, as described by the methodology above, but not individual Petri Nets embedded within the atomic plots. As s/he is not expected to use the formalisation of the PN-model directly, the methodology demands him/her to specify the plot using a single-sentence narrative; however, since verbal description is not exact description, misunderstandings between the designer and the programmer can arise. On the other hand, without quantifier triggers one cannot easily express rules like "after all important characters arrive at location *X*, start event *Y*". Apparently, there is a trade-off between complexity of expressive power of triggers and intuitiveness of their graphical portrayal. Needless to say that neither canonical dFSM nor Petri Nets possess triggers with powerful expressiveness.

Implementation

The story manager is presently fully implemented, including a debugger of stories. The scenario detailed above is implemented as well, except for some background characters' behaviour. The whole scenario is described by about 20 nets with the hierarchy depth of 3. Together, the nets contain about 100 triggers (without triggers nested in composition), 10 containers and 20 distinct tokens.

The story manager works as follows. After it loads the plot specification, it creates its internal representation. Then, it starts the execution of the plot, changing its state, step-based-step, in hand with the world manager simulating the virtual world. Execution of the PN-model resembles the original. All applicable triggers are selected and one of them is randomly chosen to fire. Closed and shrunk nets are not evaluated at all, a method for execution optimization and elimination of unwanted trigger firing.

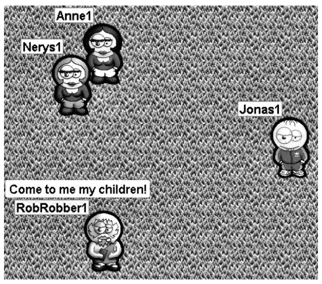

Fig. 5. A screenshot from the prototype – the final scene.

For illustrative purposes, we now detail several example situations from the Karo narrative, some of which are difficult to represent without our model. For brevity, we will use pseudo-formal form for description of actual triggers along with quoted statements describing complex triggers descriptions of which are out of the scope of this paper:

Bryant warns the girls. This action is actually represented by a trigger conjunction stating "If it is randomly decided that Bryant warns the girls, a token is put to a specific container, but this action can be done only once." The pseudo-formal form of the trigger is: "Random decision" & CREATE_TOKEN(container *c*, token *t*) & ONCE.

If the girls do not want to take Jonas with them, he goes another way. There are two subnets for the two girls' options (i.e. branching), one of them ("not to take") having two successions describing the two parallel subplots (Fig. 4b). Expansion of two parallel subnets is described by conjunction of two EXPAND_NET triggers along with a ONCE trigger.

The event, when Rob Robber emerges from the shadows starts when all important characters arrive at the scene (Fig. 4b). The condition is expressed by a universal quantifier trigger with a domain comprising of the story's main characters, and a trigger argument being a condition on a character location. This quantifier is then conjunct with change of a Rob Robber's goal. In the pseudo-formal form: ONCE & "Change the Rob's goal" & All("character *c* is in the Lonely Tree location", *c*, story_characters).

When Rob Robber is intimidating the girls, someone who is passing by overhears his threats and goes to the town to get help. The selection is performed by an existence quantifier trigger. The character chosen is "stored" in a token as the object parameter, so this information can be used later in the story. Pseudo-formal form would then be: ONCE & Exists("character *c* is near the Lonely Tree" & "change *c*'s goal to *get_help*" & CREATE_TOKEN("some container", "token containing *c* as object parameter"), *c*, helpful_characters). This trigger promotes exactly one unimportant character to t-important character. Demotion of the character is simple: the net removes the token with this character from the specific container and changes the character's goal to its former one (or possibly lets the character perform some transition behaviour).

Conclusion and Discussion

This paper has demonstrated a novel technique, based on Petri Nets, for representing plots and controlling their evolution in games emphasising a story and featuring large worlds inhabited by virtual characters, like RPGs. The main advantage of the technique is that at the same time:

(1) It allows for describing plots of stories that can evolve in parallel. This cannot be done easily by classical deterministic FSMs, but this can be achieved with a non-deterministic FSM as well as canonical Petri Nets.
(2) It allows for hierarchical decomposition of a story, which helps with speeding up the evaluation and simplifying the design; hierarchical dFSMs and nFSMs can also help in this way, but not canonical Petri Nets.
(3) It explicitly addresses the issue of story-manager—world interface by providing triggers allowing for monitoring the state of the world and by possessing tokens coupled with deictic variables by which the story manager can access world objects. Neither d/nFSM nor canonical Petri Nets possess such a mechanism; however, they can be augmented with it in principle.
(4) The expressive power of the triggers is that of the first-order logic, which allows for formulating complicated expressions.

In general, we think the technique is ready for use in practice. Its features make it especially suitable for large worlds. However, point (4) complicates the design process and slows down the execution. Hence, if the expressive power of first-order logic is not needed, simple triggers should be opted for. Actually, the more the triggers are simplified, the more the technique starts to resemble hierarchical nFSMs. In general, in small words, dFSMs or Beat approach (Mateas 2002) can be sufficient. If a parallelism is needed in a small world, a flat Coloured Timed Petri Nets model can do their job.

It has to be also kept in mind that the technique is reactive, having all the advantages and disadvantages of this approach, similarly to other techniques discussed here. Most notably, one cannot expect miracles concerning combinatorial complexity of branching; this has to be addressed manually by a designer. We think that it may be possible to overcome this by using off-line planning, i.e. to construct branches automatically to some extent during the design process; this is, however, out of our scope now.

Acknowledgement

This research was partially supported by the Program "Information Society" under project 1ET100300517, by student grant GA UK No. 1053/2007/A-INF/MFF, and by the Ministry of Education of the Czech Republic (Res. Project MSM0021620838). The authors also thank to three anonymous reviewers for their helpful comments.

References

Blackwell, L., and von Konsky, B. 2001. Petri Net Script: A Visual Language for Describing Action, Behaviour and Plot. In *Proc. 24th ACSC*, 26-37. ACM Int. Conf. Proc. Ser.

Bratman M. E. 1987. *Intention, plans, and practical reason.* Cambridge, Mass: Harvard University Press.

Brom C., Abonyi A. 2006. Petri-Nets for Game Plot. In *Proc. of AISB*, Vol. 3. 6–13. AISB press.

Brom C., Sisler V., Holan T. 2007a. Story Manager in 'Europe 2045' Uses Petri Nets. In *Proc. 4th ICVS*, LNCS 4871. 38-50. Springer-Verlag.

Brom C., Sery O., Poch T. 2007b. Simulation Level-of-detail for Virtual Humans. In *Proc. IVA'07*, LNCS 4722. 1-14. Springer-Verlag. IVE framework is available at: http://urtax.ms.mff.cuni.cz/ive/public/about.php [1.7.2008]

Brom C., Balas D., Abonyi A., Holan T., Sisler C., Galambos L. 2008. Petri Nets for Representing Story Plots in Serious Games. In *AISB Journal*. Forthcoming.

Cavazza M., Charles F., Mead S. J. 2002. Planning Characters' Behaviour in Interactive Storytelling. In *Jn of Visualization and Computer Animation*, Vol. 13, 121–131.

Delmas, G., Champagnat, R., Augeraud, M. 2007. Plot Monitoring for Interactive Narrative Games. In *Proc. of ACE 07*, 17-20.

Mateas, M. 2002. *Interactive Drama, Art and Artificial Intelligence*. Ph.D. Dissertation. Department of Computer Science, Carnegie Mellon University.

Natkin, S., Vega, L. 2003. Petri Net Modelling for the Analysis of the Ordering of Actions in Computer Games. In *Proc. of Game-ON*, 82–92.

Petri Nets World: Petri Nets World, a web collection. http://www.informatik.uni-hamburg.de/TGI/PetriNets/ [6.7.2008]

Reidl, M. O., Stern, A. 2006. Believable agents and Intelligent Story Adaptation for Interactive Storytelling. In *Proc. of TIDSE*, LNCS 4326, 1-12. Springer-Verlag.

Sheldon, L. 2004. Chap. 7 and 14. *Character Development and Storytelling.* Thompson Course Technology.

Silva A., Raimundo G., Paiva A. 2003. Tell Me That Bit Again... Bringing Interactivity to a Virtual Storyteller. In *Proc. of Virtual Storytelling II*, 146–155. Springer-Verlag.

Szilas, N. 2007. BEcoll: Towards an Author Friendly Behaviour Narrative. In *Proc. 4th ICVS*, LNCS 4871, 102-113. Springer-Verlag.