Formal Framework to improve the reliability of concurrent and collaborative learning games

I. Mounier[1,2], A.Yessad[1,2,*], T. Carron[1,2,3], F. Kordon[1 2], J-M. Labat[1,2]

[1]Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, 75005, Paris, France
[2]CNRS, UMR 7606, LIP6
[3]Université de Savoie, 73000, Chambéry, France

Abstract

Multi-player learning games are complex software applications resulting from a costly and complex engineering process, and involving multiple stakeholders (domain experts, teachers, game designers, programmers, testers, etc.). Moreover, they are dynamic systems that evolve over time and implement complex interactions between objects and players.

Usually, once a learning game is developed, testing activities are conducted by humans who explore the possible executions of the game's scenario to detect bugs. The complexity and the dynamic nature of multi-player learning games enforces the complexity of testing activities. Indeed, it is impracticable to explore manually all possible executions due to their huge number. Moreover, the test cannot verify some properties on multi-player and collaborative scenarios, such as paths leading to deadlock between learners or prevent learners to meet all objectives and win the game. This type of properties should be verified at the design stage.

We propose a framework enabling a formal modeling of game scenarios and an associated automatic verification of learning game's scenario at the design stage of the development process. We use Symmetric Petri nets as a modeling language and choose to verify properties by means of model checkers. This paper discusses the possibilities offered by this framework to verify learning game's properties before the programming stage.

Keywords: Concurrent and Collaborative learning Game; Scenario Modeling; Scenario Verification; Model Checking; Symmetric Petri Nets

Received on 29 December 2013, accepted on 05 March 2014, published on 22 May 2014

doi: 10.4108/sg.1.2.e4

1. Introduction

Context. Learning games can be defined as "(digital) games used for purposes other than mere entertainment" [1]. They are a way to help people to acquire domain knowledge and develop skills. Fabricatore [2] defines a learning game as : "[...] a virtual environment and a gaming experience in which the contents that we want to teach can be naturally embedded with some contextual relevance in terms of the game-playing [...]".

Learning games are complex software applications resulting from a costly and complex engineering process, involving multiple stakeholders (domain experts, game designers, designers, programmers, testers, etc.). In addition, the learning games implying multiple players are dynamic systems that evolve over time and implement complex interactions between objects and players. Once a learning game is developed, testing activities are conducted by humans who explore the possibles executions of the game to detects bugs.

Problem. The complexity and dynamic nature of multi-player learning games enforce the complexity of testing activities. Indeed, exploring all possible execution paths manually is impossible due to their large number. Also, multi-player learning games belong to the class of systems for which it is well known that testing activities are not sufficient to ensure reliability.

Moreover, testing activities do not allow to verify specification properties and are intrinsically performed too late because they require the game to be implemented first; thus, detected problems are costly to correct.

Contribution. To avoid costly testing procedures and improve the learning game reliability, we propose to perform automatic formal verification of scenarios of learning games at the design stage. Our objective is to ensure that a learning game satisfies properties which are extremely difficult to assess by means of

*Corresponding author. Amel.Yessad@lip6.fr tests only. Once the verification has been performed on an abstract specification, development starts from a validated design. It seems clear that after the formal verification, the test of learning games will be less costly.

This paper presents a verification approach enabling automatic verification of learning game properties. Among the available techniques, we chose the *Petri nets* to formally specify the learning game and the *model checking* techniques to verify properties.

Petri nets are a mathematical notation suitable for the modeling of concurrent and dynamic systems [3]. Due to the dynamic nature of learning games, we selected a particular Petri net model: Symmetric net with bags. Model checking is a powerful way to verify systems; it automatically provides complete proof of correctness, or explains, via a counter-example, why a system is not correct [4].

The paper presents our methodological approach that is illustrated thanks to two case studies based on real learning game as a proof of concept.

Content. Section 2 presents relevant properties for learning games. Section 3 details our approach. Then, we apply it to two case studies in section 4. Section 5 explains the automatic verification of properties before presenting some related work in section 6 .

2. Relevant Properties for Serious Games

Our work aims at verifying automatically (using model checking) properties of serious games at the design stage. We classify expected properties along two axes (see table 1). A serious game is a system which combines features relating to games and learning. Thus, the first classification axe deals with the type of a property:

- Learning property: it is related to the learning characteristics like the skills, the business process or the quizes .
- Gaming property: it is related to the fun like win a duel, avoid the monster or unlock the door.

The second axis defines the scope of a property (and therefore, the algorithms that are used for the scenario validation):

- Invariant are properties that are always verified in the game,i.e., in any state of the game.
- Reachability properties are the properties that are verified in at least a game state that is reached from the initial state. The occurrence or not of this state depends on the sequence of actions of the player.
- Temporal properties that are expressed using a temporal logic like CTL or LTL [5] and involve several states of the game and often express a sequence of game's states and define causal relations between states in the game.

	Learning-dependent	Game-dependent
Inv.	*"The learner can always improve his skills or at least maintain them" "The player can always call for help"*	*"It is always possible to perform an action before the game ends", "a player can always replay an action"*
Reach.	*"The player can acquire all skills", "The player reaches a quiz"*	*"The player can reach the virtual lab"* or *"The player can win (respectively lose) the game", "'win a duel", "avoid the monster" or "unlock the door"*
Temp.	*"The player can not complete the level as long as he does not have the competence C"*	*"The player must perform at least one game action before winning or loosing"*

Table 1. Classification of properties (invariant, reachability and temporal)

Table 1 provides instances of properties that show intersection between the two axes.

The problem is thus to be able to verify such types of properties. In this aim, we need (1) a formalization for expressing these constraints and properties of a serious game and (2) a operational framework for automatic verification.

3. Verification Framework

Today, the learning game's industry and even the game's industry in general use human testers to detect bugs in games. Obviously, this method is costly and unreliable (Most of games receive several patches after their release date). In our approach, we assume that the learning game's scenarios become more reliable and the development less costly if the scenarios' specifications are verified prior to implementation. This early verification is particularly adapted to verify properties such as the ones presented in section 2. Before presenting our verification approach, we propose a generic pattern describing a wide range of learning games that fits our approach. It concerns learning games that are organized on activities with inputs and outputs.

3.1. Generic Pattern for learning Games

Our research focuses on multi-players learning games where scenarios are composed of activities, often presented to players as challenges. An activity requires a player to have acquired some skills and some virtual objects and provides him with new skills and virtual objects (game state), depending on his performance. Thus, only players having the required skills and virtual objects (vo) may perform an activity.

Figure 1 shows an activity diagram of a learning game scenario. Activities can be performed in sequence (*e.g.* Act_1, then Act_2), in parallel (*e.g.* Act_2 and Act_3) or are collaborative (*e.g.* Act_6 requires players 1 and 2 to be performed).

3.2. Verification Method

Verification for software systems is commonly classified into three classes: simulation, algebraic methods and model checking. All can be applied to a design model, once the behavior of the system is appropriately specified.

Simulation is not well-adapted when we want to cover the whole execution space. Algebraic methods are difficult to operate and require highly skilled and experienced engineers. Model checking [6], is well-adapted to finite systems, despite an intrinsic combinatorial explosion problem: it is based on an exhaustive investigation of the system's state space and is fully automated.

We advocate that model checking is best suited for learning games. It is a good compromise between the accuracy of provided diagnostics and the automation/cost of the procedure because:

- it provides automatic verification of properties,
- it is more reliable than simulation (as well as tests on the final product with human testers),
- it requires little expertise in logical reasoning,
- it always terminates (with sufficient memory resources and when we consider finite systems) with a yes or no (then providing a useful counterexample) answer.

3.3. Symmetric Petri Nets with Bags (SNB)

Among the multiple variants of Petri Nets, we chose Colored nets that are necessary to get a reasonable sized specification, thanks to the use of colors to model data. Next, within the large variety of colored Petri Nets, we selected Symmetric Nets with Bags [7] where tokens can hold bags of colors. They support optimized model checking techniques [8]. Moreover, the notion of bags is relevant to modeling some dynamic aspects that are typical of learning games in a much simpler way than with most other colored Petri nets.

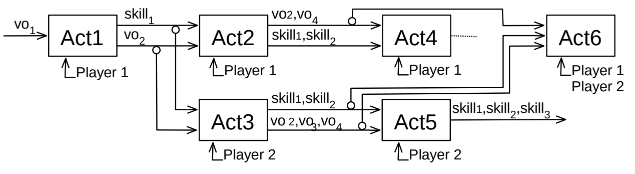


Figure 1. Scenario of a learning game

We provide here an informal presentation of Symmetric Nets and use them to model the generic pattern of learning game we presented.

Informal Definition and Example. A petri net is a bipartite graph composed of places (circles), that represent resources (*e.g*, the current state of a player in the game) and transitions (rectangles) that represent actions and consume resources to produce new ones. Some guards ([conditions] written near a rectangle) can be added to transitions.

Let us first present SNB by means of a simple example, the SaleStore (see Figure 2). People enter the sale store through an `airlock` with a capacity of two (of course, only a single person may enter too). Then, people may buy items (at most two but possibly zero if none fits their need) and leave with the acquired items. Let us note that this example has two scalable parameters: `P`, the number of involved people in the system and `G`, the number of available gifts in the warehouse.

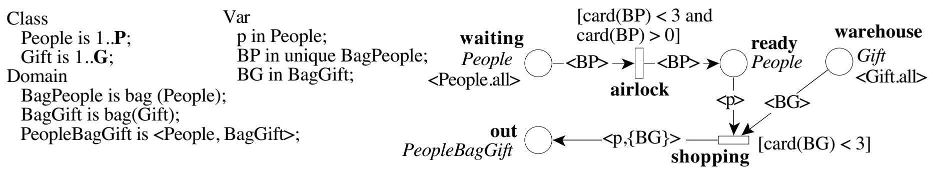
The model in Figure 2 illustrates most of the main features of SNB. First, there are several color types giving the place's domains: simple color types like `People` or `Gift` are called classes, while bags such as `BagPeople` or `BagGift` and cartesian products such as `PeopleBagGift` are built upon basic color classes.

Variables which are formal parameters for transition binding are declared in the `Var` section. A basic variable such as `p` can be bound to represent any element of `People`. A variable such as `BP` represents a multiset (or bag) of `People`; since it is tagged by the `unique` keyword, it can actually only be bound to a subset of `People` (each element in `BP` appears at most once). Variable `BG` is not tagged with `unique` keyword; it could be bound to any multiset of gifts (if the warehouse was configured to contain several instances of each gift for instance).

Transition guards can be used to constrain the cardinality of a Bag variable : the constraint $[card(BP) < 3$ and $card(BP) > 0]$ on `airlock` model its capacity of at most 2 people (the airlock does not operate empty), while the constraint on `shopping` bounds the number of gifts bought in the store by each person.

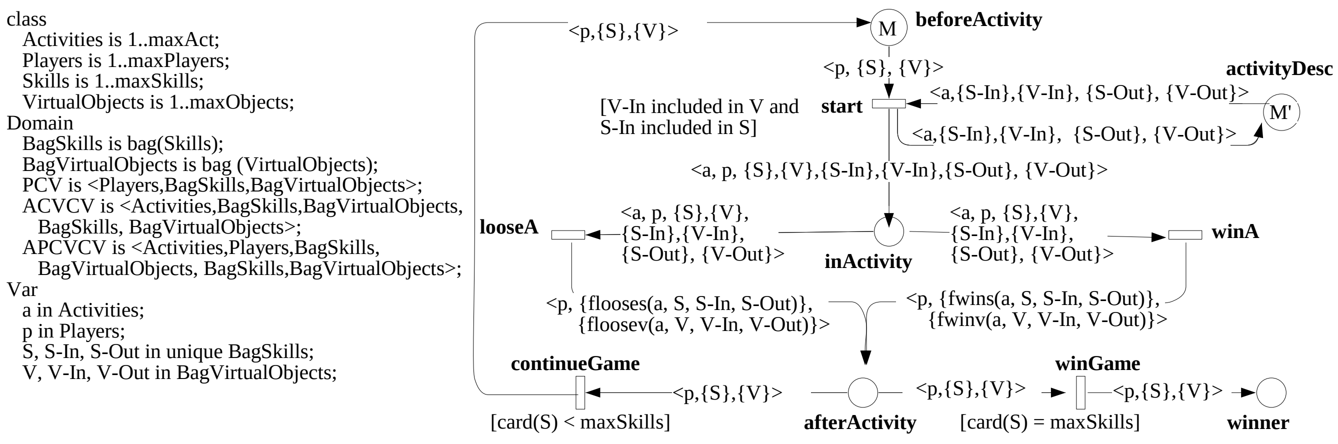
Let us now consider the SNB of Figure 3 that models a learning game activity. Place `beforeActivity` holds players and their context: skills and virtual objects (stored in bags)[1]. The initial marking `M` in place `beforeActivity` contains one token per player (identified by `p`) associated with its skills and virtual objects (sets `S` and `V` respectively). Place `activityDesc` holds the required skills and virtual objects for each activity. The initial marking `M′` in

[1]Here, only a set (not a bag) is required for skills, which is denoted by the keyword `unique` in the declaration of variables `S`, `S-In` and `S-Out`.



```
Class
   People is 1..P;
   Gift is 1..G;
Domain
   BagPeople is bag (People);
   BagGift is bag(Gift);
   PeopleBagGift is <People, BagGift>;
Var
   p in People;
   BP in unique BagPeople;
   BG in BagGift;
```

Figure 2. The SaleStore example modeled with a SNB



```
class
   Activities is 1..maxAct;
   Players is 1..maxPlayers;
   Skills is 1..maxSkills;
   VirtualObjects is 1..maxObjects;
Domain
   BagSkills is bag(Skills);
   BagVirtualObjects is bag (VirtualObjects);
   PCV is <Players,BagSkills,BagVirtualObjects>;
   ACVCV is <Activities,BagSkills,BagVirtualObjects,
      BagSkills, BagVirtualObjects>;
   APCVCV is <Activities,Players,BagSkills,
      BagVirtualObjects, BagSkills,BagVirtualObjects>;
Var
   a in Activities;
   p in Players;
   S, S-In, S-Out in unique BagSkills;
   V, V-In, V-Out in BagVirtualObjects;
```

<p,{S},{V}>
M
beforeActivity
<p, {S}, {V}>
activityDesc
[V-In included in V and S-In included in S]
start
<a,{S-In},{V-In}, {S-Out}, {V-Out}>
M'
<a,{S-In},{V-In}, {S-Out}, {V-Out}>
<a, p, {S},{V},{S-In},{V-In},{S-Out}, {V-Out}>
looseA
<a, p, {S},{V}, {S-In},{V-In}, {S-Out}, {V-Out}>
inActivity
<a, p, {S},{V}, {S-In},{V-In}, {S-Out}, {V-Out}>
winA
<p, {flooses(a, S, S-In, S-Out)}, {floosev(a, V, V-In, V-Out)}>
<p, {fwins(a, S, S-In, S-Out)}, {fwinv(a, V, V-In, V-Out)}>
continueGame
<p,{S},{V}>
winGame
<p,{S},{V}>
<p,{S},{V}>
[card(S) < maxSkills]
afterActivity
[card(S) = maxSkills]
winner

Figure 3. Modeling a game activity in SNB.

place `activityDesc` contains a token per activity (identified by a) associated with its prerequisite (`S-In` and `V-In`) and the information needed to compute the consequence of the activity on the player (in terms of `S-Out` and `V-Out`).

Each activity begins (firing of transition `start`) only when players' skills and virtual objects (game state) include the prerequisite ones. Then, the activity may end in failure (transition `looseA`) or successfully (transition `winA`). Functions `fwin` and `floose` represent the evolution of skills and virtual objects at the end of the activity (dropped in place `beforeActivity`).

The SNB shown in figure 3 allows us to model with a very abstract and concise manner a learning game scenario. This powerful expressiveness permits us to have the whole scenario on a "small" graph (useful for automatic execution) but for a better understanding, it is possible to imagine it "deployed": one for each activity as we will illustrate in the fourth section.

Interest of SNB. The SNB are appropriate to model learning games for three main reasons.

First, their capacity to structure data in tokens with sets and multisets (bags) allows to capture the dynamic part of learning games well: here, the number of skills and virtual objects that varies during a game (and can be empty).

Second, they provide an easy modeling of operations such as union or inclusion tests in transition guards. This allows for a more compact specification.

Third, SNB preserve the use of symmetry-based techniques allowing efficient state space analysis [8] that is of particular interest for the formal analysis of learning games. The model in Figure 3 is exactly the one that is verified (once max values defined), it is not mandatory to instantiate it per activity and player. We have thus a powerful formalism and some tools to verify the expected properties of a serious game.

4. Application against two Case Studies

As a proof of concept, we apply our automatic verification approach to several learning game scenarios. These latter are constructed on different versions of a Game Based Learning Management System called *"Learning Adventure"* (LA). It is a 3D multi-player environment.

Activities	Activity input	Activity output
chmod (a_1)	*"file permissions" area* (vo_1), *chest closed* (vo_2)	*"basic commands" area* (vo_3), *chest opened* (vo_4), *file permissions* (sk_1)
copy (a_2)	*"basic commands" area* (vo_3)	*"advanced commands" area* (vo_5), *file commands* (sk_2)
documentation (a_3)	*chest opened* (vo_4), *file commands* (sk_2)	*linux architecture* (sk_3)

Table 2. Description of activities in "Nuxil"

Players can move within this environment, performing activities in order to acquire skills.

All these scenarios have been ecologically experi-mented in an institute of technology by 60 students : these experiments were carried out in our university with co-located settings. During the experiment, four groups of around fifteen students with their teacher were present successively in the classroom equipped with 15 computers. Each student accessed the virtual environment through his/her workstation, and had a personal (adapted) view on the world. These students used the environment for approximately two hours and half.

In the first scenario, named Nuxil, the players had the same role and could perform same activities in the same order. They had a parallel evolution without synchronization or shared objects. The second scenario for its part was devoted to collaborative learning: during a same session, 3 groups of players followed the same idea of quest but with different order of activities. Thus, in the second scenario, we had more problems related to scheduling activities: three concurrent scenarios with a shared resource and synchronization's problems, inside the groups players.

4.1. First Case Study: multi-player and parallel scenarios

In the Nuxil scenario, the players explore the environment and have to use linux commands (*e.g.* copy (cp), move (mv), edit file permissions (chmod), ...) in the activities proposed. The objective is to offer to players a concrete metaphor of linux commands through a visual and interactive environment. For instance, in the scenario, the command "mv" serves to move objects between the game's areas and "chmod" to provide permissions for opening a chest.

Nuxil activities allow players to acquire skills (*e.g.* training on file commands). In this scenario, the players evolve separately and don't interact directly.

In this paper, we selected the specification of three activities to illustrate our approach; their inputs and outputs are presented in table 2. We deliberately distinguish virtual objects (vo_i) and skills (sk_i) in the description of an activity (a_i). This allows us to update them separately, based on the failure or success in a activity .

Modeling the Case Study. To verify Nuxil scenario, we instantiated the generic pattern of figure 3 into the model of figure 4 where $Activities = \{a_1, a_2, a_3\}$, $Skills = \{sk_1, sk_2, sk_3\}$ and $VirtualObjects = \{vo_1, vo_2, vo_3, vo_4, vo_5\}$.

In order to be able to start the game scenario with the activities of table 2, a player must have at least the virtual objects vo_1 and vo_2. Therefore we consider that initial marking M of place beforeActivity is such that

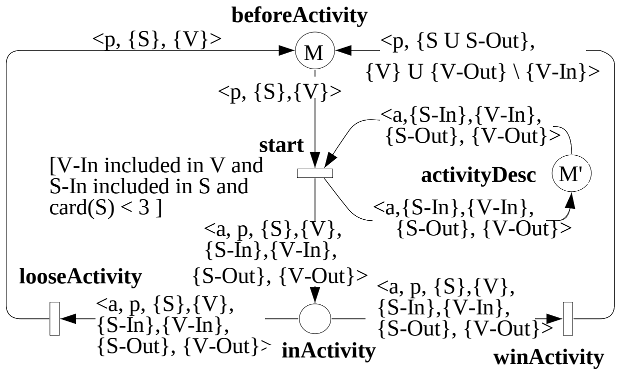


Figure 4. Part of the Nuxil game model

for each player p, M contains the token $\{\langle p, \emptyset, \{vo_1, vo_2\}\}$. The initial marking M′ of place activityDesc is $\{\langle a_1, \emptyset, \{vo_1, vo_2\}, \{sk_1\}, \{vo_3, vo_4\}\rangle, \langle a_2, \emptyset, \{vo_3\}, \{sk_2\}, \{vo_5\}\rangle\}, \{\langle a_3, \{sk_2\}, \{vo_4\}, \{sk_3\}, \emptyset\rangle\}$.

This SNB models how players acquire skills and virtual objects. For instance, the activity description $\{\langle a_1, \emptyset, \{vo_1, vo_2\}, \{sk_1\}, \{vo_3, vo_4\}\rangle$ means that the player who has the state $\{vo_1, vo_2\}$ and without skills (the empty set $\emptyset$) can perform the activity a_1 and if he performs successfully this activity he acquires the skill $\{sk_1\}$ and reaches the state $\{vo_3, vo_4\}$. Thus, when a player loses an activity, its skills and virtual objects set (game state) are not modified. When he wins one, he loses the virtual objects needed to perform the activity and wins the ones produced by the activity. New skills increase its skills set.

The Nuxil scenario allowed us to simplify the generic model by merging places beforeActivity, afterActivity and winner. We assume the game ends once all skills are obtained by a player. At this stage, no new activity should start.

For example (see Figure 4), let's imagine that the "chmod activity (a_1)" has been achieved. The player has in his SB (Skill's Bag) : sk_1 (file permissions) and in his VB (Virtual objects' Bag): 2 virtual objects/states (e.g. he obtains a key letting him access to the "basic commands area" (vo_3) and "the chest is opened" (vo_4)). At this point, the player can perform only the "copy activity" (a_2). In case of a success of this "copy activity", the player acquires a new skill "file commands" (sk_2) and one virtual object: he is moved to "advanced commands area" (vo_5) as rewards as explained on the table 2 (vo_3 is removed but not vo_4). In case of failure, we are back in the initial state but it is possible to get specific virtual objects in that case for remediation purpose (this possibility is not modeled in the Figure 4).

Verification. The model of Figure 4 allows unfair executions (*e.g.* a player will never play even when he can), that invalidate properties in unrealistic scenarios. To avoid this, we only consider fair executions. Temporal logic verification algorithms are well suited to deal with such fairness constraints.

Since the formal representation of the learning game allows the construction of the reachability/quotient graph, we can automatically verify invariant, reachability or temporal properties. We present two groups of properties. The first one concerns game deadlocks that are linked to the identification of the wining states. The second one concerns the success of learning process, *i.e.* a player may always increases his skills. The properties are informally expressed but they all can be stated as temporal logic formulas [4] that can be automatically model-checked with Crocodile tool [8].

WinningProperty Let us first define the property $winner(p)$ stating that a player won the game. We call $WinningProperty$ the property identifying the end of the game.

A player wins the game if he holds all the required skills. Therefore, $winner(p)$ is true if: $\exists\langle p, \{S\}, \{V\}\rangle \in beforeActivity$ such that $card(S) = maxSkills$. In other words, a token in place `beforeActivity` is such that the skills set holds all possible skills.

The game can end when all players (or one) win(s) all the activities. In the first case, $WinningProperty \Leftrightarrow \forall p \in$ `Players`, $winner(p)$. In the second case, $WinningProperty \Leftrightarrow \exists p \in$ `Players`, $winner(p)$.

If we want to model a game where only a player can win (*i.e.* once a player wins, the others cannot begin a new activity) we must change the guard of transition `start`. This transition can be fired only if the marking of place `beforeActivity` does not contain a token $\langle p, \{S\}, \{V\}\rangle$ such that $winner(p)$.

DeadlockProperty Unexpected deadlocks do not satisfy $WinningProperty$, therefore some executions where no player can win are possible.

If we want to verify that an ending state is always reachable (*i.e.* it is always possible to finish the game with a winner), we have to verify that a state $WinningProperty$ is always reachable from the initial configuration of the game. Such a property is a temporal logic property since it has to be verified within each execution.

Learning process property We want to verify that a player always has the possibility to increase his skills (until he wins the game). Such a property is a temporal logic property since it is necessary to compare the states along each execution. We call $increaseSkillsProperty$ the associated property.

We define first the $increaseStrictly(s, s', p)$ property where s and s' are two symbolic states and p a player. $increaseStrictly(s, s', p)$ is true if the set of skills of player p at state s is strictly included in the set of skills of player p at state s'.

Then, $increaseSkillsProperty = \forall p \in$ `Players`, $\forall s$, reachable state, set of skills is equal to `Skills` or there is a execution that leads to a state s' such that $increaseStrictly(s, s', p)$.

These properties have been checked thanks to a specific tool and thus to validate these aspects of the Nuxil scenario. Nevertheless, always more complex scenarios are imagined and with this complexity, new requirements in terms of properties verification appear (collaborative aspects, concurrency, etc.).

4.2. Second Case Study: Concurrent and Collaborative Scenario

Context and description of the experiments. The second experiment concerns a mixed reality learning game. This type of games concerns domains that present the particularity of exhibiting both theoretical knowledge and practical know-how (operations in manufacturing or medicine, for example) [9]. For such practical domains, the full digital learning games are not efficient and especially cannot guarantee both effective learning and assessment of the techniques [10]: it is so important to come back to the real world and thus develop a mixed-reality learning game.

Thus, we implemented for the second experiment a mixed-reality scenario where the course was dedicated to the Electronics field. The learning content dealt with "Electronic wiring diagrams". The aim of the session (role playing game) was to assess the knowledge and know-how of the students about these latter. More precisely, they had to create a new "very accurate" controller: a 3-axis wiimote-like controller thanks to an accelerometer. The teacher wanted to know whether the students were able to identify when a wiring diagram is valid, whether they knew how to realize a technical object from a wiring diagram, using Arduino boards, and whether they could test this technical object. In the scenario implemented, the students were divided up into three groups where the students of the same group collaborated to perform activities in predefined order. The scenario was **(1)** concurrent because some resources were shared between the three groups acting in parallel, and **(2)** collaborative because the students of a same group have to perform together the activities in order to achieve several collective goals.

The students were explicitly allowed to communicate through the chat tool provided with the system (integrated within the Game) and were warned that

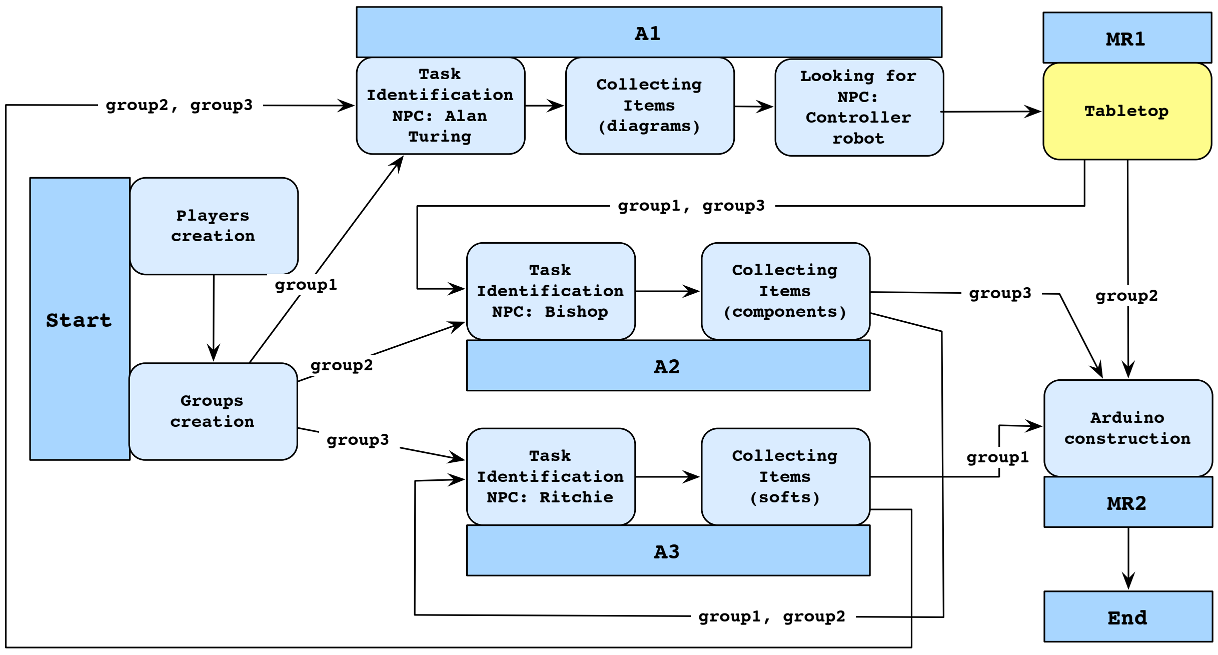

A1
Task Identification NPC: Alan Turing
Collecting Items (diagrams)
Looking for NPC: Controller robot
MR1
Tabletop
group2, group3
Start
Players creation
Groups creation
group1
group1, group3
Task Identification NPC: Bishop
Collecting Items (components)
group3
group2
group2
A2
group3
Task Identification NPC: Ritchie
Collecting Items (softs)
group1
Arduino construction
MR2
A3
group1, group2
End

Figure 5. Non formal scheme representing the multiple scenarios

they would be observed concerning the use of the system.

Concurrent aspects. The second scenario is shown schematically in the Figure 5. It is based on seven main activities (Start, A_1, MR_1, A_2, A_3, MR_2, End) that constitute the main quest (the activities Ai are performed in the virtual world and the activities MR_i, stands for MixedReality, are performed in the real life). For synchronization constraints and motivation purposes, three activity sequencing are available for the set of activities $\{A_1, MR_1, A_2, A_3\}$ and which correspond to the three students' groups: the sequence A_1-MR_1-A_2-A_3, the sequence A_2-A_3-A_1-MR_1 and the sequence A_3-A_1-MR_1-A_2. The MR_1 activity is achieved on a multitouch tabletop, it is collaborative between the students of the same group and is performed in mutual exclusion between groups. Some benefits of learning game are due to immersion and motivation aspects: if a group is obliged to wait, the immersion feeling will be broken so several concurrent scenarios must be imagined in order to avoid such loss of motivation.

To verify such scenarios thanks to our Petri net approach, we first model the game with the SNB of Figures 6 and 7. The model of Figure 6 represents de sequencing of activities without detailing them. A group is identified by an identity and the bag holding the players belonging to the group. The initial marking of place beforeGame contains all the groups, it is <1,$\{p_1,p_2,p_3,p_4\}$>+ <2,$\{p_1,p_2,p_3,p_4\}$>+ <3,$\{p_1,p_2,p_3,p_4\}$> when we have 3 groups with 4 players each. We can easily model a game with more or less groups and a number of players not the same in each group.

Place ActivitiesOrder contains the pre-defined sequencing. The tokens are structured as follows <idG,idA,idANext> where idG is the identity of the group performing the activity idA and IdANext is the activity the group has to perform after idA. Therefore we can define one sequence per group. The activity 0 is the initialization of the game (the creation of the group, we haven't modeled it), the first to be performed by each group. The activity 4 allows to know when a group has finished the game's scenario. Therefore the initial marking of place ActivitiesOrder is <1,0,1>+ <1,1,2>+ <1,2,3>+ <1,3,4>+ <2,0,2>+ <2,2,3>+ <2,3,1>+ <2,1,4>+ <3,0,3>+ <3,3,1>+ <3,1,2>+ <3,2,4>. With respect to this marking, all the groups can perform all the proposed activities in the pre-defined order.

To verify that access to tabletop is performed in mutual exclusion, we have to refined the model of activity MR1, which is done by the model of Figure 7. The places beforeActivity, beforeMR1 and ActivitiesOrder are the same as in Figure 6. It was necessary to duplicate them on this second model to show how arcs related to transition MR1 in Figure 6 are dispatched between new transitions of Figure 7. The place tableTop with 1 as initial marking ensures that the groups can access to the tabletop one at a time.

Verifying that two groups cannot arrive in front of the tabletop at the same time needs to consider only fair executions satisfying "All groups progress at a similar speed", therefore we consider only executions such that a group can progress again only if the other ones have progressed. These assumption is realistic since in the real life, if the first group to perform activity 1 progresses very slowly, it can be rejoined in front of the tabletop by the group that has to perform activity 1 in second. As this property is verified on a model that gives an abstraction of the activities, the verification results may be invalidated regarding the specification of each activity. At this stage, the performed verification ensures that the pre-defined activities order is coherent with the expected results, it will not be responsible of unexpected waiting time. Indeed, it is important to notice that these verifications enhance the global reliability in addition to beta testing but can not avoid unexpected, non formalized (or formalizable) properties.

The formal property to verify is “there is at most one token in place `beforeMR1`”. It is an invariant property automatically verified.

Collaborative aspects. Concerning this scenario, a lower analysis level can be envisaged: the collaborative aspect inside the groups. Inside a group, some collaborative tasks have to be done: for example, the players of the same group find and collect all the right electronic components dispatched over the virtual world. In a previous version of this learning game scenario, it was possible that three students achieved the collect of items whereas the last one has not yet identified this task. When urgently called by the others for getting access to the (next) $MR1$ Zone, the game did not validate the quest and blocked all the group for “unexpected” reason.

To verify such scenarios thanks to our Petri net approach we have to detail concerned activities. We model activities A2 and A3 with the SNB of Figure 8. The collaborative part of activity A1 can obviously be modeled the same way. As for the refinement of activity `MR1`, the places `ActivitiesOrder` and `beforeActivity` of Figure 8 are the same as in Figure 6.

In this part of the game, the group does not progress as a single entity but all the players that compose it progress individually. Therefore, we first have to split the group into its players. The `split` function, on the output arc of transition A23_1, performs this operation. This function has two parameters, a n-uplet and a number that identifies the component of the n-uplet that is split. In our example, the split n-uplets are `<idG,BP,idA>` where `idG` is the identity of the group, `BP` is the bag of the players belonging to the group and `idA` is the identity of the performed activity. We split the token regarding the second component, therefore we obtain a token `<IdG,idPlayer,IdA>` for each player `IdPlayer` of `BG`. The players can then play independently from each other during the activity.

The function `nbElem` is applied to an activity (respectively, a group identifier). It returns the number of items to find in order to finish the activity (respectively, the number of players that compose the group). The place `itemsToCollect` contains the items needed to complete each activity, the place `collectedItems` contains for each group and each activity the set of already collected items. The transition `lookingForMission` models the fact that the identification of the task is not immediate. Each player has to look for the non-playable character (NPC) that will gives him the information on the task.

When a player has identified the task, he begins to search and collect the item. Each time he finds one of them, his group set of collected items is updated. Once a group has collected all the needed items, all the group players have to finish the activity. They can not continue to collect new items, they all have to finish the activity. The guards of transitions `collect` and `end2` ensure that once all the expected items have been collected by the group, no member of the group continue to collect. Transitions `end_1` and its associated guard and `end_2` ensure that once all the items have been collected, all the players finish the current activity. The place `synchronization` contains, for each group, the set of players that are aware that all items have been collected by the group. Once this set holds all the players of a group, the group can go to the next activity. The initial marking of place `synchronization` associates the empty set with all groups.

Once again, to verify that a group must finish the activity when all the needed items have been collected, we have not to consider unfair executions such that a player will never play even when he can. If we consider the model without the transition `end_1` and the guard of transition `lookingForMission`, the property may not be verified even with fairness constraint. If a player has not identified the task while the others have collected all the needed items, the first player may continue to look for the task, therefore the group can no more progress, transition A23_2 can not be fired and the activity can not be finished.

The formal property to verify is “once there is a token `<IdA, IdG, {BI}>` with `card(BI) = nbElem(IdA)` in place `collectedItems`, all the executions will reach a state such that `<IdG,BP,i>` $\in$ `beforeActivity` and `IdG, IdA, i>` $\in$ `activitiesOrder`” (once a group has all the expected items, it will eventually finish the activity and go to the next activity). It is a temporal property that we can verify automatically.

5. Automatic Verification of Properties

Once the properties are specified for a learning game scenario, we formalize them for the automatic verification. In our case, we use (1) the temporal logic

```
Class
  Activities is 0..4;
  IdGroups is 1..MaxGroups;
  Players is 1..MaxPlayers;
  Items is 1..MaxItems;

Domain
  BagPlayers is bag(Players);
  BagItems is bag(Items);
  Group is <IdGroupe,BagPlayers>;
  GroupActivity is <IdGroupBagPlayers,Activities>;
  ActivitiesSequence is <idGroup, Activities,Activities>;
  ActivityItems is <Activity,Items>;
  GroupItems is <Activity,idGroup,BagItems>;

Variable
  idG in IdGroups;
  idA,idANext in Activities;
  idPlayer in Players;
  idItem in Items;
  BP in unique BagPlayers;
  BI in unique BagItems;
```

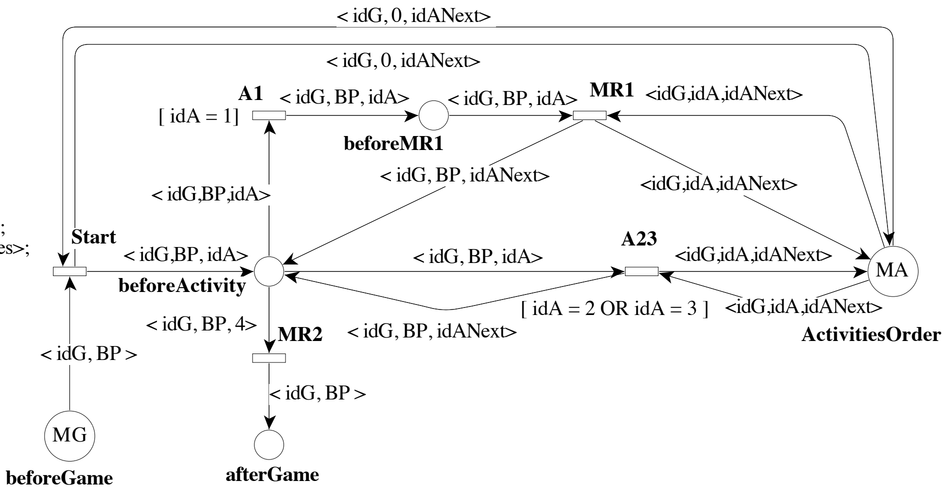


Figure 6. Formal model of the concurrent scenario, the MR_1 activity is performed in mutual exclusion

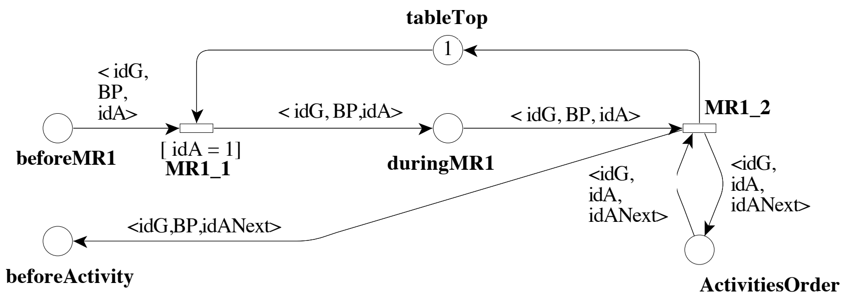


Figure 7. MR_1 activity

as a formalism for describing the properties and (2) both the CPN-AMI [11] and the Crocodile tool [8] as model checkers, able to process efficiently Symmetric Nets. These model checkers terminate (when we have sufficient memory resources and consider finite systems) with a positive answer or a negative one. If one property is not verified (negative answer), the model checker provides a useful counterexample. This counter example is useful to correct the specifications.

For instance, in the second case study, despite having found all the items, the players of one group were blocked in the collaborative activity A1. The reason was that one player was blocked and could not progress in this activity. The model checker allowed as to detect this design problem and correct it by adding the transitions Transitions end_1 and its associated guard and end_2 (see Figure 8). These two transitions ensure that once all the items have been collected, all the players finish the current activity.

Another example is that the model checkers allowed us, in the first case study, to detect that there is not an activity sequencing where all skills $\{S\}$ of the domain are reachable. In this case, it was sufficient to *(1)* specify the corresponding property: $reachable(\{S\})$ is true if: $\exists \langle p, \{S\}, \{V\} \rangle$ such that $card(S) = maxSkills$, and to *(2)* use the model checker in order to detect that this property is not valid.

6. Related Work

Petri nets are widely used in both academia and industry to model concurrent systems since they are well adapted to this class of problem. However, only a few studies address the use of Petri nets and model checkers for games.

Moreover, in most cases, Petri nets are just used to analyze game scenarios in order to adapt them to the player. In [12], the authors discuss the applicability of Petri Nets to model game systems and game flows compared with other languages such as UML. Brom and al. used this specification technique to prototype a story manager for the game *Europe 2045*[13]. The work presented in [14] uses place/transition Petri nets to assess the progression of players in games once they are developed.

Other studies focus on the analysis of game scenarios at the design stage. For instance, the "Zero game studio" group [15] uses causal graphs to model game scenarios.

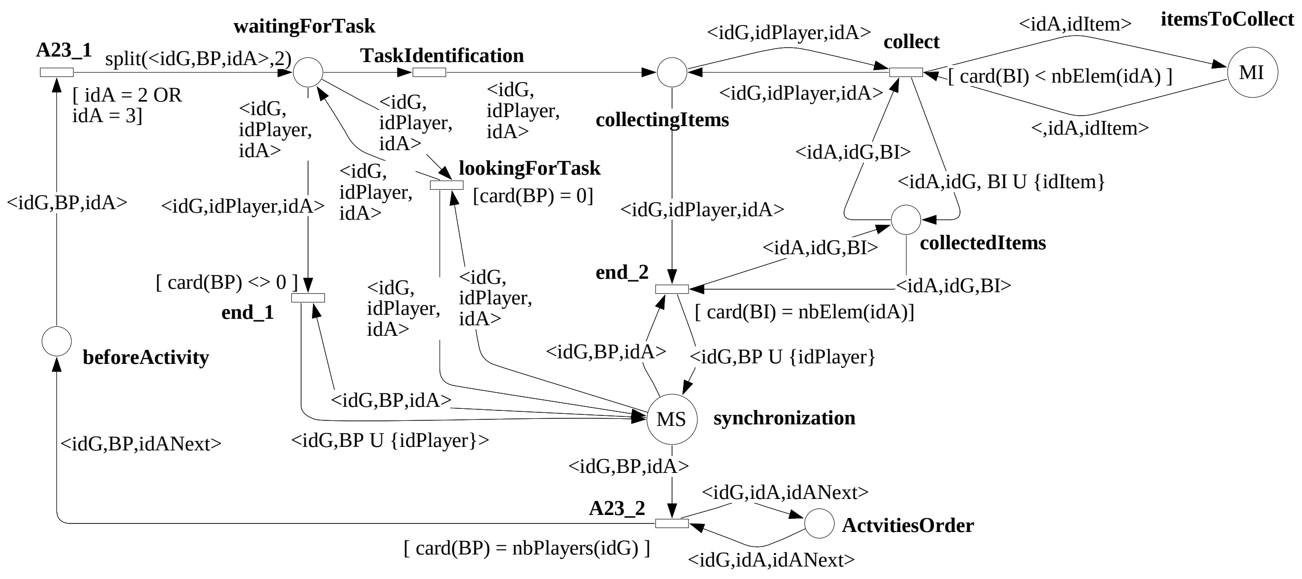


Figure 8. Collaborative activities A_2 and A_3

The work presented in [16] defines a set of safety and liveness properties of games that should be verified in the game scenarios before their implementation. In the domain of Technology-Enhanced Learning, Petri nets are used to capture the semantics of the learning process and its specificities. In particular, Hierarchical Petri nets are used in [17] to model desirable properties. The objective is to help designers to design and optimize e-learning processes.

We consider these research studies to be close to ours. The originality of our work can be summarized in three points:

- our work aims to detect inconsistencies and design errors in the learning game specifications.
- the formal framework that we propose addresses in the same way both the learning and game properties
- the SNB model that we choose and its optimized model checking techniques allow us to verify specifications automatically and efficiently.

7. Conclusion

We presented a formal verification-based approach for the design of learning games. It relies on Symmetric nets with Bags and the use of model checking to verify automatically behavioral properties of learning games. Our objective is to reduce cost and complexity of learning games elaboration by enabling early error detection (at design stage).

One interesting point of our approach is to provide a procedure helping engineers to elaborate the design of their learning game. In particular, we propose a classification of properties that are relevant in that domain. It is then possible to infer from these patterns an efficient verification procedure involving the appropriate model checkers (i.e. the one that implements the most efficient algorithms for a given property pattern).

Another important point is the use of Symmetric Petri nets with bags that better tackle the combinatorial explosion problem intrinsic to the model checking of complex systems.

We applied our approach to two real case studies for assessment purposes. Even if these case studies remain small, they show different contexts and encouraging results. This work is part of a project aiming at designing efficient formal verification based procedures for the design of learning games.

The formal model, once it is verified, could be a basis for an automated implementation of a learning game execution engine. In the long term, this could decrease the time of implementation as well as cut a large part of its costs.

Future Work A trend is to exploit the formal specification to extract relevant scenarios for testing purposes. Subsequently, human tester would have directives to follow during their testing work.

The formal specification can also be exploited to compute the pre-defined aspects of games that are modeled through initial marking of some places. The second example illustrated this need. Before the game can be played, it is necessary to define, for each group, an activities order that must satisfy some conditions. We have verified that a given pre-defined order is relevant, it would be interesting to study how scheduling and control theory results allow us to compute, among all the possible orders, the “good ones” [18, 19].

Another trend is to define transformation rules in order to construct semi-automatically Petri nets from some more user friendly models of scenarios such as eAdventure [20], LEGADEE [21].

Acknowledgements

This work was supported in part by the Region Ile de France and by the french Ministry for the Economy, Industry and Employment (FUI). We would like to thank them for their support in the "PlaySerious" project.

References

[1] Tarja Susi, Mikael Johannesson, and Per Backlund. *Serious Games: An Overview*. Institutionen för kommunikation och information, 2007.

[2] Carlo Fabricatore. Learning and videogames: an unexploited synergy. In *AECT National Convention - a recap*. Secaucus, NJ : Springer Science + Business Media, 2000.

[3] K. Jensen and L. Kristensen. *Coloured Petri Nets : Modelling and Validation of Concurrent Systems*. Springer, 2009.

[4] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001.

[5] Edmund M. Clarke. The birth of model checking. In *25 Years of Model Checking*, volume 5000 of *LNCS*, pages 1–26. Springer, 2008.

[6] E.M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.

[7] S. Haddad, F. Kordon, L. Petrucci, J-F. Pradat-Peyre, and N. Trèves. Efficient State-Based Analysis by Introducing Bags in Petri Net Color Domains. In *28th American Control Conference (ACC'09)*, pages 5018–5025. IEEE Press, 2009.

[8] M. Colange, S. Baarir, F. Kordon, and Y. Thierry-Mieg. Crocodile: a Symbolic/Symbolic tool for the analysis of Symmetric Nets with Bag. In *32nd International Conference on Petri Nets and Other Models of Concurrency*, volume 6709 of *LNCS*, pages 338–347. Springer, 2011.

[9] Sébastien George and Audrey Serna. Introducing mobility in serious games: Enhancing situated and collaborative learning. In Julie A. Jacko, editor, *Human-Computer Interaction*, volume 6764 of *Lecture Notes in Computer Science*, pages 12–20. Springer, 2011.

[10] Karen Schrier. Using augmented reality games to teach 21st century skills. In *ACM SIGGRAPH 2006 Educators Program*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[11] A. Hamez, L. Hillah, F. Kordon, A. Linard, E. Paviot-Adet, X. Renault, and Y. Thierry-Mieg. New features in CPN-AMI 3 : focusing on the analysis of complex distributed systems. In 6^{th} *International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 273–275, Turku, Finland, 2006. IEEE Computer Society.

[12] Manuel Araújo and Licínio Roque. Modeling Games with Petri Nets. In *Breaking New Ground: Innovation in Games, Play, Practice and Theory: Proceedings of the 2009 Digital Games Research Association Conference*, London, 2009.

[13] C. Brom, V. Sisler, and T. Holan. Story manager in 'Europe 2045' uses petri nets. In *ICVS 2007*, volume 4871 of *Lecture Notes in Computer Science*, pages 38–50. Springer, 2007.

[14] Amel Yessad, Pradeepa Thomas, Bruno Capdevila Ibáñez, and Jean-Marc Labat. Using the petri nets for the learner assessment in serious games. In *ICWL*, pages 339–348, 2010.

[15] Craig A. Lindley. The gameplay gestalt, narrative, and interactive storytelling. In *Proceedings of the Computer Games and Digital Cultures Conference*, pages 6–8, 2002.

[16] R. Champagnat, A. Prigent, and Estraillier P. Scenario building based on formal methods and adaptative execution. In *International Simulation and gaming association*, 2005.

[17] Feng He and J. Le. Hierarchical Petri-nets model for the design of e-learning system. In *Proceedings of the 2nd international conference on Technologies for e-learning and digital entertainment*, Edutainment'07, pages 283–292. Springer, 2007.

[18] Peter Brucker. *Scheduling algorithms (4. ed.)*. Springer, 2004.

[19] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[20] Universidad Complutense Madrid. http://e-adventure.e-ucm.es.

[21] LIRIS, Lyon. http://liris.cnrs.fr/legadee/.