Have you found the error? A Formal Framework for Learning Game Verification

A. Yessad, I. Mounier, J-M. Labat, F. Kordon, and T. Carron

LIP6, CNRS UMR 7606, Université P. & M. Curie – Paris 6
4, place Jussieu, F-75252 Paris Cedex 05, France
Amel.Yessad@lip6.fr, Isabelle.Mounier@lip6.fr,Jean-Marc.Labat@lip6.fr,
Fabrice.Kordon@lip6.fr, Thibault.Carron@lip6.fr

Abstract. Specifications of of Multi-Player Learning Games (MPLG) are expressed collaboratively by designers who don't speak the same conceptual language. Often, specifications contain design errors and inconsistencies that are difficult to detect in playtests. In this paper, we present a formal framework to assist designers in modeling and automatic verification of learning games at the design stage of development process.

Keywords: Multi-Player Learning Game, Learning Game Verification, Instructional Design, Game Design, Model Checking, Symmetric Petri nets.

1 Introduction

Context Learning Games can be defined 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 [...]" [1]. Multi-Player Learning Games (MPLG) are learning games involving multiple players who are competitors or collaborators.

MPLGs are software applications resulting from costly and complex engineering processes, involving multiple stakeholders (domain experts, game designers, learning designers, programmers, testers, etc.) who often do not speak the same language. Usually, at the end of the development process of MPLGs, testing activities are conducted by humans testers who explore the possible executions of the game to detect both design errors and programming bugs. However, the design errors are particularly hard to find and so specifications and design properties become hard to verify. This observation is more true for MPLGs implying multiple players which are concurrent and dynamic systems implementing complex interactions between game universe and players.

Problem First, the difference between the language used by game designers and the language of learning designers can lead to inconsistency and design errors in specifications. In this context, the construction of a rigorous specification clarifies several aspects of system behaviour that are not obvious in an informal specification. Then, the complexity and dynamic nature of MPLGs makes more difficult the verification of properties on specifications, only by playtests. Indeed, it is widely acknowledged that it is cost-effective to spend more efforts at the specification stage, otherwise, many flaws

would go unnoticed only to be detected at the later stages of software development that would lead to iterative changes to occur in the development life cycle [2]. In addition, the exploration of all possible execution paths of MPLG scenarios is impossible manually due to their huge number. For example, it is difficult for learning designers to ensure, in game specifications, properties such as the winner of the game is always a player that has acquired all domain skills or to ensure that it is proposed systematically a reinforcement activity to a player who fails in a game level.

Contribution We advocate to check the properties of MPLGs (such as those mentioned above) at the design stage before starting the programming stage in order to reduce cost of testing activities and verify these properties automatically. We propose a framework in order to formalize and verify game scenarios, at the design stage. Our objective is to ensure that a MPLG satisfies properties which are extremely difficult to assess only by means of playtests. Thus, once the verification is performed on an abstract design, development starts from a validated design. After this formal verification, the test of MPLGs would be less costly.

This paper presents a formal approach enabling automatic verification of MPLG properties. Among the available techniques, we chose the *Petri nets* to formally specify the MPLG 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 provides automatically complete proof of correctness, or explains, via a counter-example, why a system is not correct [4]. This counter-example can be used to pinpoint the source of the error [5].

Content The following section draws a parallel with the related work. Section 3 presents classification of properties for MPLGs. Section 4 details our verification framework. Then, we apply it to a case study in section 5. Section 6 presents a discussion of our approach.

2 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 Multi-Player Learning Games.

Moreover, in most cases, Petri nets are used to analyze game scenarios in order to adapt them to the player. In [6], the authors discuss the applicability of Petri Nets to model game systems and game flows compared with other languages such as UML. The work presented in [7] 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 [8] uses causal graphs to model game scenarios. The work presented in [9] 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 characteristics

of learning process. In particular, Hierarchical Petri nets are used in [10] to model good properties. The objective is to help designers to 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 between learning designers and game designers who do not speak the same conceptual language. Designers may have difficulties to describe consistent and non-ambiguous specifications of MPLG scenarios;
- the formal framework that we propose addresses the learning and the game properties at the same time;
- the model of Petri Nets that we chose and its optimized model checking techniques allow us to verify specifications automatically in a finite time.

3 Classification of properties for MPLGs

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

- **Learning** properties are related to the learning characteristics like the skills, the business process or the quizzes .
- **Gaming** properties are related to the game universe 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 to verify it):

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

Table 1 provides examples of properties that show intersection between the two axes. The problem is thus to be able to verify such types of properties. For this purpose, we need (1) a formalization for expressing constraints and properties of a MPLG and (2) an operational framework for automatic verification of these properties.

4 Verification Framework

Today, the MPLG companies and even the video game companies use human testers to detect errors 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 MPLG scenarios become more reliable and the development less costly if specifications

	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", " "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)

are verified prior to programming stage. This early verification is particularly adapted to verify properties such as the ones presented in section 3.

Before presenting our verification approach, we propose a generic pattern describing a wide range of MPLGs that fits our approach. The verification method is presented in [11].

4.1 Generic Pattern for MPLGs

We are interested in MPLGs where game scenarios are composed of independent activities, often presented to players as challenges. An activity can have inputs and outputs. The inputs require that a player has some skills and is in a specific virtual state (a state of the player in the game, ex. the player has the key or he finds the exit of the labyrinth, etc.). Thus, only players with the required skills and virtual state may perform the activity. At the end of the activity, players can acquire new skills and be in a new virtual state, depending on their performances. The new skills and virtual state represent the outputs of the activity.

Figure 1 shows an activity diagram of a MPLG 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).

4.2 Symmetric Petri Nets with Bags (SNB)

Among the multiple variants of Petri Nets, we chose the 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 (SNB) [12] where tokens can hold bags of colors. They support optimized model checking techniques [13] allowing to verify properties of MPLG. Moreover, the notion

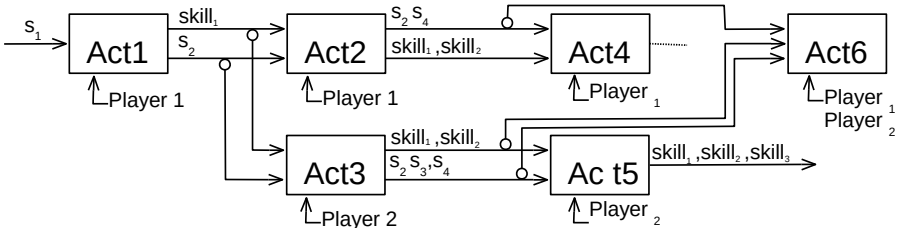

Fig. 1. Scenario of a learning game

of bags is relevant to model some dynamic aspects (game objects can appear or disappear, knowledge of players can increase or decrease) that are typical of MPLGs in a much simpler way than with most other colored Petri nets.

We provide here a short and informal presentation of Symmetric Nets and use them to model the generic pattern of MPLG 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.

The SNB of Figure 2 models activities of MPLG. Place `beforeActivity` holds players and their context: skills and virtual state (stored in bags). The initial marking `M` in place `beforeActivity` contains one token per player (identified by `p`) with his initial skills and virtual state (sets `S` and `V` respectively). Place `activityDesc` holds the required skills and virtual states for each activity. The initial marking `M'` in place `activityDesc` contains a token per activity (identified by `a`) 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 their virtual states include the prerequisite of the activity. Then, the activity may end in failure (transition `looseA`) or successfully (transition `winA`). Functions `fwin` and `floose` represent the evolution of skills and virtual states of players at the end of the activity (dropped in place `beforeActivity`). A player wins when `winCond` is true, it expresses conditions on skills or/and virtual state.

The SNB shown in figure 2 allows us to model with very abstract and concise manner MPLG scenario. This powerful expressiveness allows 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 SNB for each activity.

SNB preserve the use of symmetry-based techniques allowing efficient state space analysis [13] that is of particular interest for the formal analysis of MPLGs. The model in Figure 2 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 [13] to verify the expected properties of a MPLG.

5 Case Study: multi-players and concurrent scenarios

We have applied our automatic verification approach to multi-players, parallel, concurrent or collaborative scenarios of MPLGs in order to test the feasibility of our formal framework. We present the results obtained when studying the "Wonderland" MPLG.

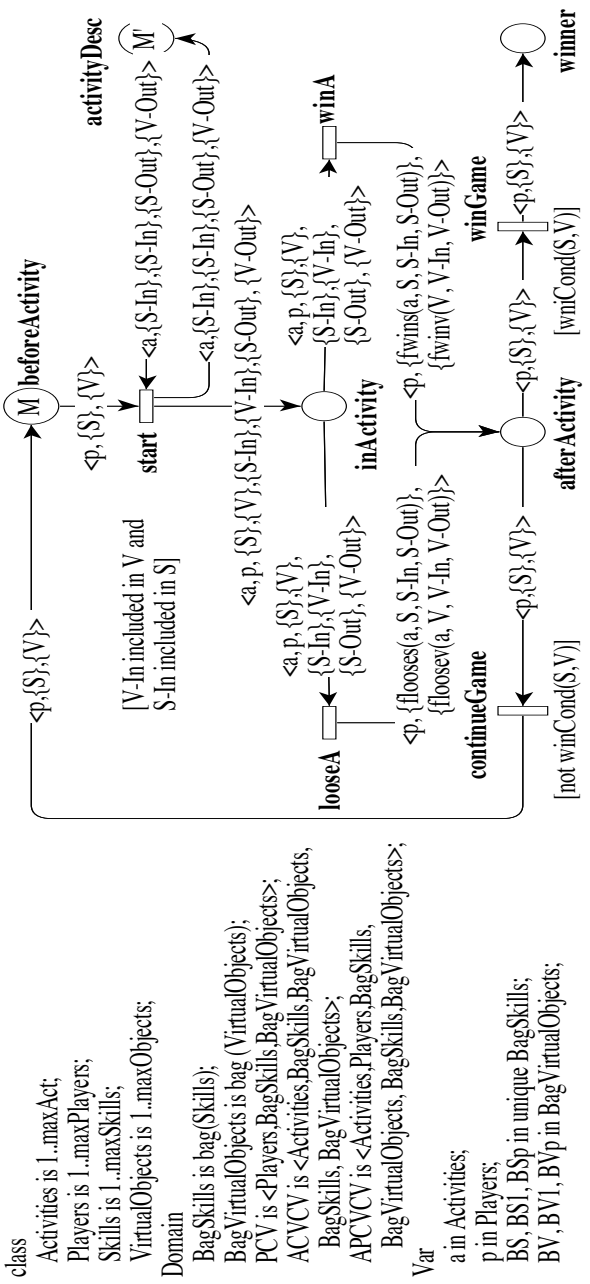

Fig. 2. Modeling game activities with SNB.

5.1 Brief Presentation of the Wonderland Scenario

The "Wonderland" MPLG aims to train pupils of the primary school to solve arithmetic problems. Knights (the players) have to find and liberate a princess kidnapped by a dragon. To succeed, the knights have to resolve arithmetic problems. In term of skills, a player wins when he holds all the skills, therefore `winCond` is `card(S) = maxSkills`.

Designers have described activities that allow players to progress in the scenario. The activity specifications were as always described by text which makes them ambiguous and non-exhaustive. In this context, the use of Petri net models allows designers to describe the MPLG behaviour formally and to verify it automatically in order to detect design errors.

First, the textual specifications of Wonderland scenario are analyzed manually in order to identify for each activity (1) its prerequisite skills and input virtual states and (2) the acquired skills and the output virtual states, at the end of the activity. Then, designers gave a list of properties that the scenario has to verify. Thanks to our properties classification (cf. section 3), we had classified these properties. For clarity reasons, we present only subset of activities in table 2 and subset of properties in the table 3, useful for explaining our approach. In the following, we will verify automatically if properties of the table 3 are satisfied given the specifications of the table 2. Design errors and inconsistencies were detected, have you found these errors?

Activities	Inputs	Outputs
level on hard additions and hard subtractions (a_8)	*"simple additions" (sk_2), "simple subtractions" (sk_3), "player meets a wizard" (s_7)*	*"complicated additions" (sk_4), "complicated subtractions" (sk_5), "player wins telescopic ladders" (s_8), "the player fights against the dragon" (s_{14})*
level on simple multiplications (a_9)	*"simple additions" (sk_2), "simple subtractions" (sk_3), "player on one side of the gorges" (s_9)*	*"simple multiplications" (sk_6), "player on the other side of the gorges" (s_{10})*
level on simple multiplications (a_{10})	*"simple additions" (sk_2), "simple subtractions" (sk_3),"player wins telescopic ladders"(s_8)*	*"simple multiplications" (sk_6), "player on one side of the gorges" (s_9)*
level on multiplication tables (a_{11})	*"simple additions" (sk_2), "simple subtractions" (sk_3),"player on the other side of the gorges"(s_{10})*	*"multiplication tables" (sk_7), "player reaches the dragon tower" (s_{11})*
level about finding errors on simple multiplications (a_{12})	*"simple additions" (sk_2), "simple subtractions" (sk_3), "player helps the tower guard to finish his homework " (s_{12})*	*"simple multiplications" (sk_6), "tower guard opens the tower door for rewarding the player" (s_{13}), "the player fights against the dragon"(s_{14})*
reinforcement level on simple multiplications (a_{13})	*"player on one side of the gorges" (s_9)*	*"simple multiplications" (sk_6), "player on the other side of the gorges"(s_{10})*
reinforcement level on simple multiplications (a_{14})	*"player reaches the dragon tower" (s_{11})*	*"simple multiplications" (sk_6), "player helps the tower guard to finish his homework " (s_{12})*
level on complicated multiplications (a_{15})	*"the player fights against the dragon"(s_{14})*	*"complicated multiplications" (sk_8), "player kills the dragon and frees Princess" (s_{15})*
reinforcement level on complicated multiplications (a_{16})	*"simple multiplications" (sk_6), "the player fights against the dragon"(s_{14})*	*"complicated multiplications" (sk_8), "player captures the dragon" (s_{16})*

Table 2. Inputs and outputs of a subset of Wonderland activities (levels)

	Learning-dependent	Game-dependent
Inv.	*"A player who acquires all skills wins the game", "No player may liberate the princess without all the skills"*	*"Only one player can win the game and all others lose it", "The player wins only when he liberates the princess"*
Reach.	*"Each skill can be acquired", "A player can win the game"*	*"A player can win telescopic ladders", "At least one player can kill the dragon"*
Temp.	*"The learner can always improve his skills or at least maintain them", "The player can not acquire the skill "complicated multiplication" (sk_8) as long as he does not have the skill "simple multiplication" (sk_6)"*	*"The player has to meet the wizard before winning the telescopic ladders"*

Table 3. Classification of Wonderland properties (invariant, reachability and temporal)

Modeling the Wonderland scenario We instantiated the generic pattern of figure 2 into the model of figure 3 where $Activities = \{a_8, ..., a_{16}\}$, $Skills = \{sk_2, ..., sk_8\}$ and $PlayerStates = \{s_7, ..., s_{16}\}$ in order to verify the part of Wonderland scenario corresponding to the activities of the table 2,. The initial marking `M'` of place `activityDesc` contains tokens corresponding to the different activities and their inputs and outputs. For example we have for the activity a_8 the token $\langle a_8, \{sk_2, sk_3\}, \{s_7\}, \{sk_4, sk_5\}, \{s_8\}\rangle$ and for the activity a_{16} the token $\langle a_{16}, \{sk_6\}, \{s_{14}\}, \{sk_8\}, \{s_{16}\}\rangle$.

The Wonderland SNB models how players acquire skills and virtual states. For instance, the activity description $\langle a_8, \{sk_2, sk_3\}, \{s_7\}, \{sk_4, sk_5\}, \{s_8\}\rangle$ means that the player who is in the state $\{s_7\}$ can perform the activity a_8 if he has already acquired the skills $\{sk_2, sk_3\}$. If he performs this activity successfully he acquires the skills $\{sk_4, sk_5\}$ and reaches the new state $\{s_8\}$. On the other hand, when a player loses an activity, we assume, in the case of Wonderland, that its skills and virtual state are not changed. But other strategies are possible (change his virtual state, call into question his skills , etc.) The Wonderland scenario allowed us to simplify the generic model by merging places `beforeActivity`, `afterActivity` and `winner`. In this SNB, the game scenario continues as long as at least one player has not acquired all the skills. This property is guaranteed by the guard $card(S) < 7$, associated to the transition start and means that the player has not yet all skills (in our case 7). We consider that a player wins when he has all the skills.

Verification Once the properties are specified for a MPLG scenario, we formalize them for the automatic verification. In our case, we use (1) the temporal logic to formalize the properties and (2) both the CPN-AMI [14] and the Crocodile tool [13] as model checkers, able to process efficiently Symmetric Nets. These model checkers terminate (if we

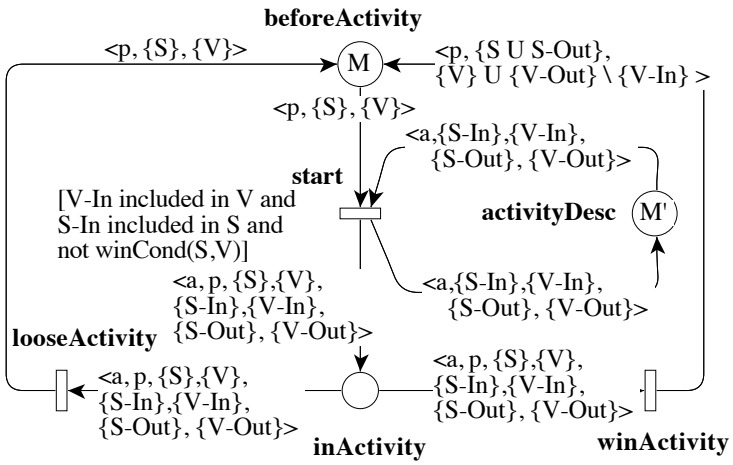

Fig. 3. Part of the Wonderland game model

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 counter-example. This counter-example is useful to pinpoint the source of the error and to correct the specifications. Since the formal representation of the MPLG allows the construction of the reachability/quotient graph, we can verify automatically invariant, reachability or temporal properties. The properties presented in the table 3 are informally specified but they were described by temporal logic formulas [4] and model-checked automatically with Crocodile tool. It provided us either a complete proof of correctness of properties or a counter-example.

Ending and winning properties Let us first define the property $winner(p)$ stating that a player won the game. The learning-dependent associated property provided by the learning designer is *"The player who acquires all skills wins the game"* (cf. the table 3). Therefore, $winner(p)$ is true if: $\exists \langle p, \{S\}, \{V\} \rangle \in beforeActivity$ such that $card(S) = maxSkills$. In other words, there is a player token in place `beforeActivity` such that the set of the player's skills contains all the possible skills (in our case 7).

We call $WinningProperty$ the property that ensures that a player can win the game: $WinningProperty \Leftrightarrow \exists p \in$ `Players`,$winner(p)$ (corresponding to the property *"A player who acquires all skills wins the game"*). We call $EndingProperty$ the property identifying the end of the game. The game designer specified it as *"Only one player can win the game and all others lose it"* (cf. the table 3), it was formalized as follows: $\exists ! p \in$ `Players`,$winner(p)$.

The Crocodile tool has proved the first property and gave us counter-examples for the second one. It allowed us to detect an inconsistency in specifications between the game designers and the learning designers. Once the inconsistency detected, the designers must agree on the solution to implement. If they want to model a game where only one player can win (*i.e.* once a player wins, the others cannot begin a new activity) they 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 $winner(p)$.

The learning-dependent property *"No player may liberate princess without all the skills"* and the game-dependent one *"The player wins only when he liberates the princess"* associate the winning condition to the liberation of the princess. Crocodile tool allows us to detect inconsistencies between the two winning views (acquisition of all the skills and liberation of the princess) and the specification of the game given in table 2.

A player could liberate the princess without acquiring the skills $\{sk_6, sk_7\}$. The counter-example is the activity sequence: $\langle a_8, a_{15}\rangle$. The inputs of action a_{15} have to be modified. The skills *simple multiplications* $\{sk_6\}$ and *multiplication tables* $\{sk_7\}$ seem to be necessary to perfom the action a_{15} that gives the skill *complicated multiplications*.

A player could acquire all the skills without liberating the princess. It occurs with the following activities sequence : $\langle a_8, a_{10}, a_{13}, a_{11}, a_{14}, a_{12}, a_{16}\rangle$. The players has all the skills, therefore he wins, he has captured the dragon but he didn't liberate the princess. Thus, the model checker allowed us to detect this problem and to correct it in the learning game specifications by modifying the output of the activity a_{16} by specifying the state s_{16} as *"players captures the dragon and frees the Princess"*.

Scheduling properties Among the temporal properties we have verified, the Crocodile tools allowed us to detect a problem in specifications about the property *"The player can not acquire the skill "complicated multiplication" (sk_8) as long as he does not have the skill "simple multiplication" (sk_6)"*. Indeed, the Wonderland SNB allows a sequence$\langle a_8, a_{15}\rangle$ where a player performs the activity a_{15} that allows to acquire the skill sk_8 before performing one of the activities a_9, a_{10}, a_{12}, a_{13} or a_{14} that allow to acquire the skill sk_6. Thus, it is possible for a player to acquire the skill sk_8 before the skill sk_6. We corrected this error by adding the skill sk_6 as input for the activity a_{15}.

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 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`, $\forall s$, a reachable state, the set of skills is equal to `Skills` or there is a path leading to s' such that $increaseStrictly(s, s', p)$.

This important learning-dependent property would have been very difficult to be verified manually. The use of Crocodile tool allowed us to validate it in the Wonderland scenario.

6 Discussion

We had the opportunity to use our framework with learning game designers. We got some interesting feedback that we have structured in four sections.

Petri net construction and abstraction level The construction of Petri net models by designers was the most big obstacle in their appropriation of the formal framework. We proposed the generic pattern of learning games in order to resolve even partially this problem. Indeed, it is less complicated to ask designers to describe a set of game activities in terms of inputs and outputs than to construct by themselves the Petri net. After that, transformation of the designer's description towards a SNB is obtained automatically. The most frequent feedback of designers were the issue of granularity of the activities. It was so difficult for designers to find the good abstraction level to describe the activities. Some designers considered that an activity is a game level and they were not focusing inside levels whereas others had more detailed abstraction and considered that an activity is a puzzle inside the level.

Complete and formal specifications In the video game industry, development teams use agile methods where the specifications are rarely described completly and formally at the beginning of the development. One reason to this is that playtest is the only method to detect bugs and so it is important to start production very fastly. Our approach is different because it requires complete description of the specifications before starting the production stage. In this sens, our approach seems constraining but in the context of MPLGs, it allows to discover errors and inconsistencies that playtests can not discover or discover too late. Other research have to be conducted to adapt or propose platforms that offer designers of MPLGs a high level of abstraction for their specification work, which is independent of any programming language, while rich enough to express their needs.

Property-oriented specifications In the property-oriented approach, the system's behaviour is defined indirectly by stating its properties. These properties are verified on SNB models automatically. During our experiences with designers, we observed that the property-oriented approach is not natural for designers. Currently, we are working on some property patterns for helping designers to express properties about their MPLG. These patterns are extracted from most frequent properties expressed on specifications of some MPLG. Ultimately, it would be interesting to build a base of reusable property patterns that designers can query according to their abstract needs.

Common language The proposed framework offers a common language to designers in order to describe their learning game activities and properties of MPLG. This language can be constraining but designers explained us that it is a good mean to give learning designers and game designers common and consensual language. The use of the model checking tool as third component at design stage made more easier the communication and had reduced conflict between designers.

Other experiments are underway and would allow us to assess more the framework.

7 Conclusion

We presented a verification framework allowing the formal modeling and the automatic verification of MPLG scenarios. Our objective is to reduce cost of MPLGs development by enabling detection of specification errors and inconsistencies at the design stage.

One interesting point of our approach is to provide tools helping both learning designers and game designers to verify the MPLG specifications. In particular, we propose a classification of properties that designers may need to verify in scenario. The class of a property determines the most efficient algorithm for verifying the property.

Another important point is the use of Symmetric Petri net with bags that better tackle the combinatorial explosion problem intrinsic to the model checking of MPLG specifications. We have focused this paper on the verification of the learning-dependent properties but our model checkers verify also the game-dependent properties.

We applied our approach to specifications of a MPLG "Wonderland" and we were able to detect inconsistencies and errors in the properties specified by learning and game designers.

Future Work Other research are conducted to extend the framework in order to support the verification of properties related to collaboration and to concurrency in MPLGs. We are also working on an editor to assist designers in building SNB models. We have considered MPLGs composed of independent activities, we now have to put our framework to the test of other types of MPLGs and adapt it if necessary.

References

1. C Fabricatore. Learning and Videogames: an Unexploited Synergy. In *2000 AECT National Convention*. Secaucus, NJ : Springer Science + Business Media, 2000.
2. Rajib Mall. *Fundamentals of Software Engineering*. Prentice-Hall of India Private Limited. Rajkamal Electric of India Private Limited.
3. K. Jensen and L. Kristensen. *Coloured Petri Nets : Modelling and Validation of Concurrent Systems*. Springer Verlag - ISBN: ISBN 978-3-642-00283-0, 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. E.M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.
6. Manuel Araújo and Licínio Roque. Modeling Games with Petri Nets. In *2009 Digital Games Research Association Conference*. Brunel Univ., September 2009.
7. 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.
8. Craig A. Lindley. The gameplay gestalt, narrative, and interactive storytelling. In *Proceedings of the Computer Games and Digital Cultures Conference*, pages 6–8, 2002.
9. R. Champagnat, A. Prigent, and P. Estraillier. Scenario building based on formal methods and adaptative execution. In *International Simulation and gaming association*, Georgia Tech, USA, 2005.
10. Feng He and J. Le. Hierarchical Petri-nets model for the design of e-learning system. In *2nd international conference on Technologies for e-learning and digital entertainment*, pages 283–292. Springer, 2007.
11. Amel Yessad, Isabelle Mounier, Thibault Carron, Fabrice Kordon, and Jean-Marc Labat. Formal Framework to improve the reliability of concurrent and collaborative learning games. *EAI Endorsed Transactions on Serious Games Journal*, page to appear, 2014.
12. 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)*, pages 5018–5025. Omnipress IEEE, 2009.

13. 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, June 2011.
14. 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)*, pages 273–275, Turku, Finland, June 2006. IEEE Computer Society.