Toward Improvement of Serious Game Reliability

Thibault Carron, Fabrice Kordon, Jean-Marc Labat, Isabelle Mounier and Amel Yessad
LIP6, CNRS UMR 1606, Université Pierre & Marie Curie, Paris, France
Thibault.Carron@lip6.fr
Fabrice.Kordon@lip6.ff
Jean-Marc.Labat@lip6.fr
Isabelle.Mounier@lip6.fr
Amel.Yessad@lip6.fr

Abstract: Serious games are complex software applications resulting from a costly and complex engineering process, involving multiple stakeholders (domain experts, teachers, game designers, designers, programmers, testers, etc.). In addition, the serious games implying multiple learners-players are dynamic systems that evolve over time and implement complex interactions between objects and players. Traditionally, once a serious game is developed, testing activities are conducted by humans who explore the possible executions of the game's scenario to detect bugs. The non-deterministic and dynamic nature of multi-player serious games enforces the complexity of testing activities. Indeed, exploring all possible execution paths manually is impracticable humanly due to their large number. Moreover, the test can detect some bugs, but cannot verify some properties of serious games such as the scenario allows a learner to acquire all the knowledge, that the winner is necessarily one who has achieved all the learning objectives or the scenario does not lead to deadlock situations between learners. This type of properties has to be verified at the design stage of serious games' development. We propose a framework enabling a formal modelling and an automatic verification of serious game's scenario at the design stage of development process. We use Symmetric Petri nets as a modelling language and choose to verify properties by means of model checking. Petri nets are a mathematical notation suitable for the modelling of concurrent and dynamic systems. Due to the dynamic nature of serious game's scenario, we selected a particular Petri net model: Symmetric Petri net. Model checking is a powerful way to verify systems; it provides automatically a complete proof of correctness, or explains, via a counter-example, why a system's property is not correct. This paper discusses how this framework is used to verify the serious game properties before the programming stage begins. In order to concretise our discourse, we apply our approach on a scenario of a serious game and present how design's properties are expressed and verified thanks to the formal framework.

Keywords: Serious Game, software engineering, Serious Game verification, model checking, petri nets

1. Introduction

Context

Serious games can be defined as ``(digital) games used for purposes other than mere entertainment'' (Susi et al., 2007). They are a way to help people to acquire domain knowledge and develop skills. Particularly, we share the definition of Fabricatore (2000): ``a serious game is 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 [...]''.

Serious 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 serious games implying multiple players are dynamic systems that evolve over time and implement complex interactions between objects and players. Once a serious game is developed, testing activities are conducted by humans who explore the possible executions of the game to detect bugs.

Problem

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

Moreover, testing activities do not allow verifying 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 complex testing procedures and preserve serious game reliability, we propose to perform formal verification of serious games at the design stage. Our objective is to ensure that a serious game satisfies properties that are extremely difficult to assess by means of tests only. Once the verification has been performed on an abstract specification, development starts from a validated design.

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

Petri nets are a mathematical notation suitable for the modelling of concurrent and dynamic systems (Jensen et al., 2009). Due to the dynamic nature of serious 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 (Bérard et al., 2001).

The paper presents our methodological approach that is illustrated on a case study based on a real serious game as proof of the concept. Section 2 presents relevant properties for serious games. Section 3 details our approach. Then, we apply it to the case study in section 4 before section 5 presents some related work before concluding and presenting some perspectives.

2. Relevant properties for Serious Games

Our work aims at automatically verifying (using model checking) stated properties on serious games at the design stage. We classify expected properties along two axes (see table 1). The first one deals with the relationship between a property and a game that can be:

- Game-independent: it is relevant for any serious game,
- Game-dependent: it is specific to a given serious game and not applicable to others.

The second axis only involves the type of property (and later, the algorithms to be used for verification):

- Invariant properties are always verified in the game,
- Reachability properties must be verified in a game state that can be reached from the initial state,
- Temporal properties, expressed using a temporal logic like CTL or LTL, define causal relations between some classes of states in the game.

Table 1 provides examples of game properties. Property patterns can be defined for game-independent properties, and then filled with conditions describing specificities of a given game. This is the case for the game-independent reachability properties where only lose or win conditions need to be specified. Game-dependent properties need to be defined according to the rules and the gameplay of the game modelled.

Table 1: Classification of properties (invariants, reachability and temporal)

	Game-independent	Game-dependent
Invariant	It is always possible to perform an action before the game ends	The player can always call for help
Reachability	The player can win (respectively lose) the game	"The player can reach the virtual lab" or "The player can get skills to kill the monster"
Temporal	The player must perform at least one action before winning or loosing	The player can not complete the level as long as he does not have the competence C

According to the type of formula, a verification scheme can be elaborated. Temporal properties require the designer to write a temporal logic formula concerning the causal relation between the identified states. Other properties only require the definition of a logic formula without any temporal connector. Usually, temporal properties are more likely to be game-dependent than others.

3. Verification framework

Today, the video game industry uses human testers to detect bugs in games. Obviously, this method is costly and unreliable. In order to cut development costs and increase serious game reliability, serious game specifications have to be verified prior to implementation. Indeed, testing is used to verify properties of serious games, such as the ones presented in section 2. However, testing is not sufficient to verify properties

of serious games. In this section, we advocate the use of formal verification as a better way to proceed. To do so, we propose a generic pattern model describing a wide range of serious games.

3.1 Generic pattern for Serious Games

Our research focuses on multi-player serious 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/or virtual objects, depending on his performance. Thus, only players having the required skills and virtual objects (vo) may perform an activity.

Figure 1 shows a diagram relating activities in a game. Activities can be performed in sequence (*e.g.* Act1, then Act2), in parallel (*e.g.* Act2 and Act3) or with some exclusion (*e.g.* Act6 requires players 1 and 2, thus, if it is completed, player 1 cannot perform Act4 anymore).

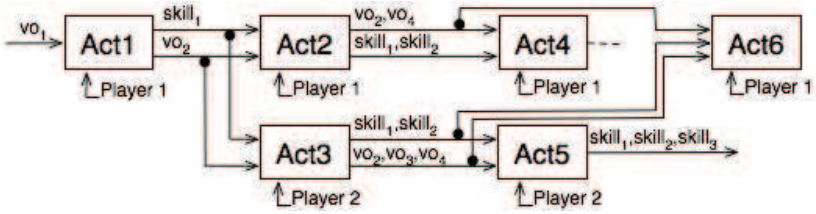

Figure 1: Example of a scenario of a serious game

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 behaviour 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 (Clarke et al., 1999), 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 serious 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 Coloured nets that are necessary to get a reasonable sized specification, thanks to the use of colours to model data. Next, within the large variety of coloured Petri Nets, we selected Symmetric Nets with Bags (Haddad et al., 2009) where tokens can hold bags of colours. They support optimized model checking techniques (Colange et al., 2011). Moreover, the notion of bags is relevant to modelling some dynamic aspects that are typical of serious games in a much simpler way than with most other coloured Petri nets.

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

3.3.1 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 a serious game activity. Place *beforeActivity* holds players and their context: skills and virtual objects (stored in bags). Here, only a set (not a bag) is required for skills, which is denoted by the keyword *unique* in the declaration of variables *BS*, *BS1* and *BSp*. The initial marking *M* in place *beforeActivity* contains one token per player (identified by *p*) associated with its skills and virtual objects (sets *BS* and *BV* respectively). Place *activityDesc* holds the required skills and virtual objects for each activity. The initial marking *M'* in place *activityDesc* contains a token per activity (identified by *a*) associated with its prerequisite (*BS1* and *BV1*) and the information needed to compute the consequence of the activity on the player (in terms of *BSp* and *BVp*).

```
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;
  BS, BS1, BSp in unique BagSkills;
  BV, BV1, BVp in BagVirtualObjects;
```

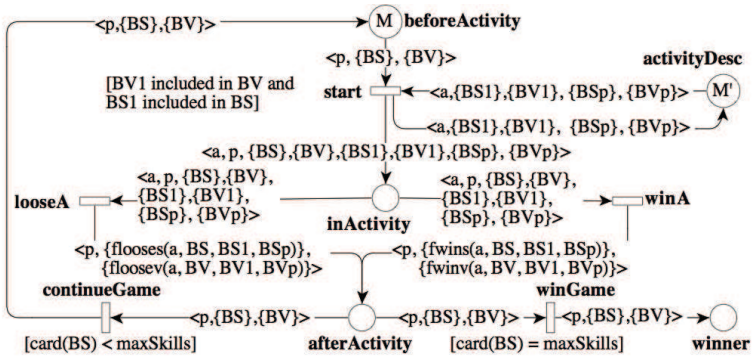

Figure 2: Modelling a game activity in SNB

Each activity begins (firing of transition *start*) only when players' skills and virtual objects include the one of its prerequisite. Then, the activity may end in failure (transition *looseA*) or in 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 2 allows us to model with a very abstract and concise manner a serious 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.

3.3.2 Interest of SNB

The SNB are appropriate to model this type of system for three main reasons.

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

Second, they provide an easy modelling 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 (Colange et al., 2011) that is of particular interest for the formal analysis of serious games. 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.

4. Application to a case study

As a proof of concept, we apply our verification framework to the serious game Nuxil constructed on a Game Based Learning Management System called *Learning Adventure* (LA). LA is a 3D multi-player environment with lakes, mountains and hills where activities can be performed. Players can move within this environment, performing activities in order to acquire skills and virtual objects. We present here the Nuxil’s automatic

verification during the design stage. At the end of the development, the Nuxil's scenario has been ecologically tested in an institute of technology by 60 students (four groups of 15).

In the game Nuxil, players explore the environment and have to use linux commands (*e.g.* copy (*cp*), move (*mv*), edit file permissions (*chmod*), etc.). The objective is to illustrate linux commands through both visual and interactive environment. For instance, the command *mv* is used by players in order to move virtual objects between game areas and the command *chmod* provides permissions for opening a chest that is represent a locked computer file.

4.1 The case study

Nuxill activities allow players to acquire skills (e.g. mastering the file management commands) and win virtual objects (e.g. the key of a chest). Activities are proposed to players, depending on their skills and their virtual objects. In this article, we selected 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. The former are related to the gaming and the second to the learning.

4.2 Modeling the case study

To verify Nuxil, we first instantiated the generic pattern of figure 2 into the model of figure 3 with *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, vo_6}*. For example, a player must have at least the virtual objects vo_2, vo_3, *and* vo_6 in order to perform the activity a_2.

We consider that initial marking *M* of place *beforeActivity* is such that for each player *p*, *M* contains the token *{<p, {}, {vo_1, vo_2}}*. The initial marking *M'* of place *activityDesc* is

{<a_1, {}, {vo_1}, {sk_1}, {vo_1}>, {<a_2, {}, {vo_2,vo_3,vo_6}, {sk_2}, {vo_4, vo_5}>}, {<a_3, {sk_1}, {vo_4}, {sk_3}, {}>}.

This SNB models how players acquire skills and virtual objects. When a player loses an activity, its skills and virtual objects sets 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.

Table 2: Description of some activities in Nuxil

Activities	Activity input (preconditions)	Activity output (postconditions)
copy (a_1)	*"basic commands" area (vo_1)*	*"basic commands" area (vo_1), file commands (sk_1)*
chmod (a_2)	*"file permissions" area (vo_2), chest closed (vo_3) "PNJ wizard (vo_6)*	*"advanced commands" area (vo_5), chest opened (vo_4), file permissions (sk_2)*
documentation (a_3)	*chest opened (vo_4), file commands (sk_1)*	*linux architecture (sk_3)*

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

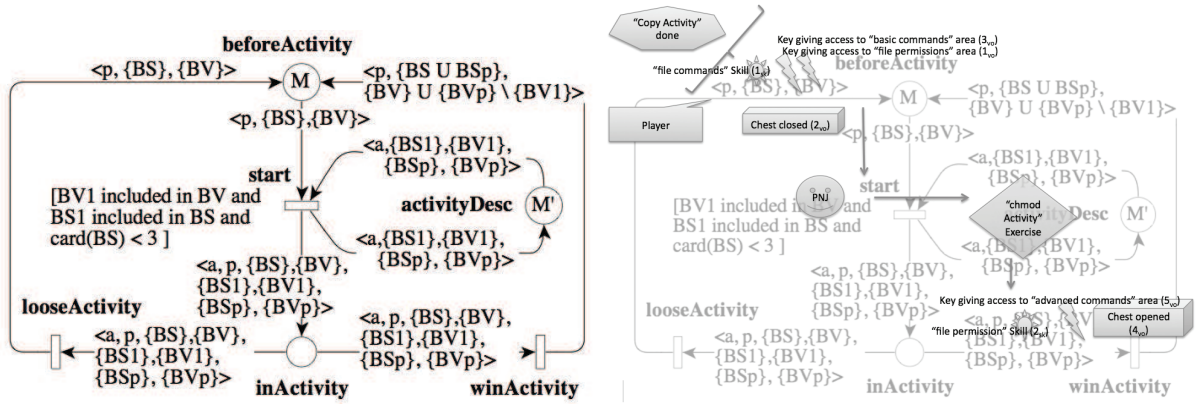

Figure 3: Part of the Nuxil game formal model and an instantiation of it on the right

For example (see Figure 3), let's imagine that the "copy" activity has been achieved. The player has in Skill Bag : sk_1 (file command skill: the upper star on the figure 3) and in his BV (virtual objects bag): 3 virtual objects (two access keys and a closed chest). The PNJ will then propose the "chmod" activity and in case of a success, the player will get in addition a new skill (sk_2) and two virtual objects: vo_5 ("*advanced commands*" *area) and* vo_4 (*chest opened)* as rewards as explained on the table 2 (vo_2 will be removed). 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.

4.3 Verification

Since the formal representation of the serious game allows the construction of the reachability/quotient graph, i.e., the state space of the game, we can 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 can be stated as temporal logic formulas (Bérard et al., 2001) that can be model-checked.

4.3.1 WinningProperty

Let us first define the property *winner(p)* stating that a player *p* 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: there is a token *<p,{BS},{BV}>* in the place *beforeActivity* such that *cardinality(BS)=maxSkills*. In other words, a token in place *beforeActivity* is such that the bag of skills *BS* contains all skills.

The game can end when all players (or one) win(s) all the activities. In the first case:
WinningProperty = for all p in Players, winner(p).
In the second case:
WinningProperty = there is at least one 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 have to change the guard of transition *start in the SNB*. This transition can be fired only if the marking of place *beforeActivity* does not contain a token *<p, {BS}, {BV}>* such that *winner(p) is true*.

4.3.2 DeadlockProperty

When deadlocks appear, the property *WinningProperty* becomes not satisfied. *In these case*, 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 state of the game. Such a property is a temporal logic property since it has to be verified for each execution.

4.3.3 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 of the game and *p* a player. *increaseStrictly(s,s',p)* is true if the bag of skills of player *p* at state *s* is strictly included in the bag of skills of player *p* at state *s'*.

Then, *increaseSkillsProperty = for all p in Players, for all reachable state s, set of skills is equal to Skills or all the possible executions lead to a state s' such that increaseStrictly(s,s',p).* This formula allows verifying by model checking that the serious game always improves the player's skills.

5. Related work

Petri nets are 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 used to analyse game scenarios in order to adapt them to the player. (Araujo et al., 2009) discuss the applicability of Petri Nets to model game systems and game flows compared with other languages such as UML. The work presented by (Yessad et al., 2010) uses place/transition Petri nets to assess the progress 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 (Lindley 2002) uses causal graphs to model game scenarios. The work presented in (Champagnat, 2005) defines a set 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 by (He et al., 2007) to model desirable properties. The objective is to help designers to design and optimize e-learning processes.

We consider these researches to be close to ours except that our research allows verifying patterns of properties related to both the learning aspects and the behaviour of the game. This is a possible thanks to a generic model of serious games, capturing most games and modelling learning skills as well as learners and game objects. In addition, SNB provides an efficient way to model games and together with efficient model-checking-based analysis. We claim our solution is more adapted to the dynamic and complex nature of serious games.

6. Conclusion

We presented a formal verification of serious games at the design stage. It relies on Symmetric nets with Bags and the use of model checking to verify automatically behavioural properties of serious games. Our objective is to reduce cost and complexity of serious 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 serious 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 a real case study for assessment purposes. Even if this case study remains small, it shows encouraging results. This work is part of a project aiming at designing efficient formal verification based procedures for the design of serious games.

The formal model, once it is verified, could be a basis for an automated implementation of a serious game execution engine. In the long term, this could decrease the time to implementation as well as to 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.

Another trend is to define transformation rules in order to construct semi-automatically Petri nets from some models of scenarios more user friendly such as eAdventure (http://e-adventure.e-ucm.es), LEGADEE (http://liris.cnrs.fr/legadee/), etc.

References

Araujo, M. and Roque, L. (2009) *Modeling Games with Petri Nets*, Breaking New Ground: Innovation in Games, Play, Practice and Theory: Proceedings of the 2009 Digital Games Research Association Conference, London.

Bérard B. et al (2001) *Systems and Software Verification. Model-Checking Techniques and Tools.* Springer, 2001.

Champagnat, R., Prigent, A. and Estraillier, P. (2005) *Scenario building based on formal methods and adaptative execution*, International Simulation and gaming association.

Clarke, E., Grumberg, O. and Peled, D. (1999) *Model checking.* MIT Press.

Colange, M. et al. (2011), Crocodile: a Symbolic/Symbolic tool for the analysis of Symmetric Nets with Bag, 32nd International Conference on Petri Nets and Other Models of Concurrency, LNCS, vol. 6709. Springer, pp. 338–347.

Fabricatore, C. (2000) *Learning and videogames: an unexploited synergy*, *AECT National Convention - a recap.* Secaucus, NJ : Springer Science + Business Media.

He, F. and Le, J. (2007) *Hierarchical Petri-nets model for the design of e-learning system*, Proceedings of the 2nd international conference on Technologies for e-learning and digital entertainment.

Lindley, C. A. (2002) *The gameplay gestalt, narrative, and interactive storytelling*, Proceedings of the Computer Games and Digital Cultures Conference, pp. 6–8.

Haddad, S. et al. (2009) *Efficient State-Based Analysis by Introducing Bags in Petri Net Color Domains*, 28th American Control Conference (ACC'09). IEEE Press, pp. 5018–5025.

Jensen, K. and Kristensen, L. (2009) *Coloured Petri Nets : Modelling and Validation of Concurrent Systems,* Springer.

Susi, T., Johannesson, M. and Backlund, P. (2007) *SeriousGames: An Overview* (technical report)., Skövde, Sweden: University of Skövde.

Yessad, A. et al. (2010) *Using the Petri nets for the learner assessment in serious games*, *ICWL,* pp. 339–348.