Detecting structural errors in scene-based Multiplayer Games using automatically generated Petri Nets

Christian Reuter
TU Darmstadt
Rundeturmstraße 10
64283 Darmstadt, Germany
christian.reuter@kom.tu-darmstadt.de

Stefan Göbel
TU Darmstadt
Rundeturmstraße 10
64283 Darmstadt, Germany
stefan.goebel@kom.tu-darmstadt.de

Ralf Steinmetz
TU Darmstadt
Rundeturmstraße 10
64283 Darmstadt, Germany
ralf.steinmetz@kom.tu-darmstadt.de

ABSTRACT

When building a complex game it is very easy to make small mistakes during design or implementation. These flaws can result in situations where the players cannot continue the game as planned (for example due to dead- or livelocks). In this paper we describe our approach for detecting those structural errors in scene-based single- and multiplayer games. This is done by generating a Petri Net where the individual game elements are mapped to appropriate constructs. We do so not only for high level goals, but for each possible user interaction. Combining this approach with an automatic transformation of the game yields a huge benefit: there is no need to develop a verification model in parallel to the game and there is also no margin for inconsistencies that could lead to the model not accurately representing the game.

The resulting Petri Net can then be used in an external tool in order to verify that the game has certain properties, for example that it is always possible to reach a well-defined ending. We also discuss the complexity of the nets and explain our optimization approaches that allow us to create Petri Nets for small games covering every single user input. In order to show its applicability we evaluate our approach by verifying multiple real world example games, taking complexity and time measurements.

Categories and Subject Descriptors

K.8 [**Personal Computing**]: Games

General Terms

Games

1. INTRODUCTION

Creating games nowadays is an incredibly complex task, since they often consist of multiple systems which must not only interact with each other, but also react to any possible player input in a well-defined manner. Thereby, it creates a vast number of possible game states and interactions with lots of room for errors, both on a (theoretical) design and on a (practical) implementation level. But although it is widely known in software development that it gets more costly to fix an error the later during development it is detected, game developers focus their quality assurance mostly on playtesting. While having humans test a game is certainly necessary in order to evaluate the user experience, user tests can only be done once the game has reached a certain level of fidelity (for example users need some kind of understandable representation in order to comprehend the game).

Playtesting is also a simulation approach, which means that it is able to detect errors, but cannot guarantee their absence because not every possible path through the game can be tested that way. Doing this would require the testers to try out every possible combination of interactions, often with different timings as well, which is practically impossible. This is especially obvious for multiplayer games where multiple testers would be required to act in a certain way and where there is an even larger possibility space for game states (for instance there are n^m ways m players could be located in n rooms instead of m ways for a single player).

It would be beneficial, however, to make sure that the players could not under any circumstances reach a game state where there is no way to bring the game to a meaningful ending as intended by the designers. Such an ending does not necessarily have to be a winning state. Showing a "game over" screen is a valid ending state for lots of games as well. Guaranteeing that such states are always reachable requires verification, which can conclusively prove that a certain condition will or will not be true. Another advantage of formal verification methods compared to user tests is that they can be used at an early project stage without requiring real content in the form of assets and without recruiting real players.

The drawback of verification is, however, that it is quite costly in terms of calculation time to an extent where it is not possible once the problem at hand (i.e., the game to be verified) reaches a certain size. Therefore, most existing approaches do not aim to verify every (implementation) aspect of a game. Instead they analyze a high level design, for example the game's mission structure, but not every single step required to solve each individual (sub-)goal. Doing so allows designers to check whether their ideas could work, but not if they have been implemented correctly and therefore

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.
Proceedings of the 10th International Conference on the Foundations of Digital Games (FDG 2015), June 22-25, 2015, Pacific Grove, CA, USA.
ISBN 978-0-9913982-4-9. Copyright held by author(s).

they cannot be sure that the resulting game can be solved by real players. Especially misunderstandings between the designers and programmers, but also things like programming bugs can therefore not be detected using this approach.

We therefore investigated an approach that works in an early design stage as well, but in which the verification model "grows" together with the game as it gets implemented. When it is completed, the model therefore describes the complete game with every possible player interaction, not only high level description. And since the model is generated automatically from the game there is also no margin for inconsistencies between the game and its model as it could be the case when they were created in parallel by the developers themselves.

The paper is structured as follows: First we take a look at game verification approaches existing in the related work (Section 2). After that we elaborate on the requirements our approach was designed to fulfill, i.e. the elements of scene-based multiplayer games that are modeled and the error classes that can be detected (Section 3). We then describe why Petri Nets have been chosen as a formal representation for the games for verification. It is also detailed how the game state and game events, as well as the individual elements of which they consist, are mapped to Petri Net elements (Section 4). Then the complexity of the resulting nets and optimization strategies in order to reduce their size are discussed (Section 5), followed by the results we could achieve when applying a prototypical implementation to both toy and real world example games (Section 6). We conclude by debating on how the approach does scale for larger games (Section 7).

2. RELATED WORK: VERIFICATION

Since video games are software systems by definition, general software verification approaches like the *ZING* model checker [1] work on games as well. But by being general purpose, these methods are unable to exploit specific properties of games and directly answer design level questions such as "Will players be able to reach an ending?".

A more game specific approach is to view games as transition systems with finite variables. While being relatively easy to understand, these systems can be verified using model checking tools. For this purpose Moreno-Ger et al. [10] implemented an automatic transformation process for games created with with the <*e-Adventure*> platform. Using the external tool *NuSMV*, the transformed game's temporal properties, such as whether a scene can be reached, can be verified. This is similar to the approach by Picket [14], who modeled games as a *narrative flow graph* (i.e., an extended 1-safe Petri Net defined by Verbrugge [19]) and then also checked the game's temporal properties using *NuSMV*.

Osborn et al. [13] designed the *Gamelan* language to describe games based on board game rules, which, they argued, is more understandable due to game designers often using board games for prototyping purposes. This language can also be used to define and answer questions (e.g. is every available function used) in discrete single- and multiplayer games.

Using Petri Nets, originally designed for describing distributed systems, as a model for games is not a new idea. Natkin et al. for example used Petri Nets at the design stage to detect deadlocks in adventure-like games [11]. Their model supports a game world consisting of rooms in which objects can be interacted with and items can be picked up. Champagnat et al. [4] choose a similar approach, but argued that the state space explosion prohibits a complete proof. Instead they suggested a combination of simulation and verification that analyzes only a few steps ahead of the current game state and solves problematic situations when they arise during runtime. They also differentiated between playability properties like having no situations in which the players cannot progress further and relevance properties like assuring a certain complexity of the game.

Furthermore, Araújo and Roque [2] argued that Petri Nets are mathematically well-founded and can be used for early simulations in order to detect unexpected multiplayer interactions. They also mentioned the complexity of the resulting nets as a problem. Since 1-safe Petri Nets are limited by the fact that tokens moving through the system cannot have properties on their own and are therefore indistinguishable, they suggested using colored Petri Nets to describe more complex situations. This was supported by Carron et al. [3] who used colored Petri Nets to calculate the reachability of learning objectives and to detect deadlocks in multiplayer games. Their approach allowed them to also verify invariant properties, which must always be true, and temporal ones concerning the order of certain events. These properties can be both game-independent (a game must be solvable) and game-dependent (a specific item must be used). The drawback is that they relied on the Petri Net being created in parallel to the game during the design stage.

It is also interesting to note that Petri Nets have not only been used to verify a game's properties, but also to generate stories [15], support artificial intelligence agents [18], and model the player's behavior [17] or learning progress [20].

In contrast, Dommans [5] argues that Petri Nets are difficult to read and therefore proposed his own graphical language using the same basic concepts (tokens and places). He then added additional elements that encapsulate complex functionality, for example events that can only happen if the player is skilled enough. The language is geared towards resource production / consumption and was used to describe individual feedback loops to be analyzed by the designer, not for the automatic analysis of complete games.

Aside from formal verification it is possible to simulate a game's behavior over time by also using a (simplified) model, written in a specification language [12].

A common drawback of all approaches found in the related work, aside from the work by Moreno-Ger et al. [10], is that the verification models have to be created manually. This requires the game designer to have additional knowledge while introducing the potential for human error.

3. PROBLEM DEFINITION

In this paper we will focus on detecting structural problems in scene-based single- and multiplayer games.

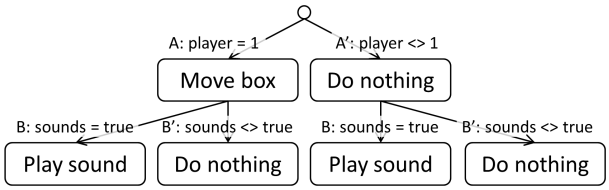

Figure 1: An example event with exclusive conditions. Only player one is strong enough to move the box. If the sound is enabled, a scraping sound is played when any player tries to do so.

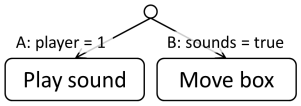

Figure 2: The same example event as in Figure 1, but with non-exclusive conditions.

3.1 Scene-based Games

Scene-based games are separated into a number of discrete scenes (or rooms) between which the players are able to move. Switching between rooms can happen in a linear or non-linear fashion and often triggers a noticeable transition like a loading screen. A well-known example for this type of games are point-and-click-adventures like *Maniac Mansion* [8]. It is obvious that the room a player is in is a vital part of the current game state in this kind of games. In a multiplayer setting it is important to notice that the players can be in the same scene or in different ones, which results in a much larger possibility space for the game state. Objects with which the players can interact are located in these rooms, providing the main gameplay mechanic. Another important part of the game state consists of additional helper variables, allowing the game to track if a puzzle has been solved or an item has been picked up, for example.

This game state can be changed by events, which are triggered by the users' input. These events can consist of an arbitrary number of game (re-)actions, for example changing variables, playing a sound or moving a player to another scene. The actions can be organized into a branching tree with conditions so that an event can yield different results depending on the current game state and the player triggering the action (Figure 1). The conditions themselves can consist of atomic boolean expressions comparing variable values to a constant (e.g. *sounds* > *true* or *player* <> *1*). These simple expressions can then be combined into more complex ones using the *and* / *or* operators as well as brackets. We will assume that events are processed atomically, which means that concurrent events are executed one after another and they cannot interrupt each other, even if they consist of multiple sub-actions.

In addition to these basic features our model allows scenes to be organized hierarchically in order to save development time, with child scenes inheriting the objects placed in their parents. As an example application for this one can imagine multiple outside locations sharing a parent in which the sky is defined. The concept also supports objects that can be enabled or disabled, also affecting the events attached to them. Another shortcut for designers is that action tree branches do not have to be exclusive (i.e., exactly one of their conditions is true at any time). This can reduce the complexity of event trees with multiple independent conditions or branches where an action is removed instead of replaced under certain conditions, but is less intuitive and more error-prone. For example, Figure 2 shows the non-exclusive version of same event as Figure 1. Lastly it should be noted that any approach designed for multiplayer games with a variable number of players will work for singleplayer games as well (as a special case with a player count of one).

3.2 Error Classes

When designing or implementing a scene-based multiplayer game there are several types of mistakes that can be made. Structural errors like unconnected rooms are relatively easy to detect when an appropriate visualization is provided. Small logical errors like individual conditions that are unsatisfiable on their own (i.e., contradictions) are a little bit harder to find, but are still detectable by humans. However, there can be more complex problems spanning multiple rooms and events, especially in a multiplayer setting. To give an example:

At the start of the game there is a key which can be picked up by any player. Later on, there is a larger area with multiple rooms between which the players can move freely. One room is a storage closet that can be opened with the key and which contains a ladder. Another room has a trapdoor from which the players must recover an item. A player can jump into the trapdoor and get out again using the ladder. If the ladder is not in his or her inventory, other players can let the ladder drop in from above.

Looking at the individual elements everything seems to be alright and there are multiple ways to solve this puzzle. However, if the player who picked up the key jumps into the pit before opening the storage door there is no way for him or her to get out again. Especially when interdependencies between puzzle elements get more complex this kind of error is very hard to see and may not be found during testing if the players choose to play the game the "right" way.

All in all, our approach is able to detect the following error classes:

Deadlocks: Situations in which the players cannot change the game state anymore, for example when sitting in a trap without any escape.

Livelocks: Situations in which the players can change the game state, but not to reach the ending. An example for this is a trap in which the players can move around or interact with objects, but are unable to leave and finish the game.

Unreachable scenes: Scenes that cannot be reached under any circumstances. This can happen when there is no connection to them at all, or when these connections are guarded by unsatisfiable conditions.

Impossible actions: Actions that can never be triggered, for example because of unsatisfiable conditions or because the object they are attached to is always disabled.

4. PETRI NET MODEL

We decided to use Petri Nets as the model in which to detect structural errors because they have been designed to model concurrent executions. This is especially important when it comes to multiplayer games where multiple players are acting independently and concurrently. Petri nets are defined as graphs where places (P) are connected with transitions (T) via arcs (A). The graph is bipartite, which means that there are no direct arcs between two places or two transitions. Each place (usually visualized by a circle) can hold one or more tokens while transitions (displayed as boxes) define the movement of tokens between the places. A transition is called "enabled" when each place connected to an incoming arc holds at least one token. Enabled transitions can then be fired, which results in one token per incoming edge being deleted and one per outgoing being created. If multiple transitions are enabled the net may choose between them at random.

Verifying properties like liveness (there is a transition that can be fired) or reachability of certain states has been well researched with lots of tools being readily available. Another benefit of Petri Nets compared to formal languages is that the place-transition-graph is more easily readable for humans.

For multiplayer games it is important to differentiate between individual players, for example when only one player character is able to execute a specific action in a class-based game design. Therefore, a colored Petri Net [6] is needed, where colors (Σ) are used to group tokens of the same type while each one has a value from the color's value range assigned to them. Transitions can then have guard conditions (G), that require the input tokens from each class to have a certain value.

In order to prevent inconsistencies that might occur if the net and the game were created manually using separate tools, the Petri Net model has to be created automatically based on the actual game. For efficiency reasons it is beneficial to implement the transformation process directly into a game engine or authoring tool, defining a corresponding model for every supported feature and therefore supporting any game created with it.

4.1 Game state

The current state of a Petri Net is defined by the tokens that are present in each place, so it is natural to map the game state to places, too. Therefore, in our model there is one place per room where the players can be and one for each helper variable. A small example net is described in Figure 3, containing two rooms (*Hall* and *Storage*) and one variable (*Open*). From this follows that there is one token for each player ($\{P_1, ..., P_n\}$, in this case P_1 and P_2) and one for each variable (*Open*). It is also natural to map each variable type used in the game (*bool*) to one color, with the players being a special case requiring their own color (*Players*).

Since the color of each token must match the color of the places (noted below each place in the figure) it is supposed to be, the player color is also assigned to each place representing a room and the places for the variables are colored with the corresponding variable type. Nevertheless, there is an important difference between players and colors. While the variable tokens always stay in the same room with the variable values being modeled by the token value (*Open* being initially *false* in the example), the player tokens move through the room places with their value unchanged unless players are able to switch roles. Keeping the player tokens in a single place would make the Petri Net / game state more difficult to read, while the opposite approach of having moving variable tokens is too complex to model (see Section 4.3).

If the game in question organizes its scenes hierarchically, the hierarchy must be flattened in order to be exported as a non-hierarchical Petri Net. This requires copying all objects provided by the parents into the child scenes and creating an individual place for each scene, both parents and children.

4.2 Game events

Similarly, changes of the game state are mapped to transitions, so there is at least one transition per game event. However, since we assume atomic execution of events with multiple actions, all of these individual actions must be merged into a single transition for the Petri Net. This in turn prevents the net from choosing different paths through a complex action-tree based on which conditions are true at that time. So there must be one transition for each possible path through a game event representing all the actions that lie on this path.

Calculating each possible path is relatively easy when the game designers made sure that each decision in the branching tree is exclusive. In this case, firing any enabled transition in the Petri Net simulator does not harm the result. However, when there are multiple branches that might be true the simulator must execute all of them simultaneously as selecting each of them separately would violate the atomicity. In order to prevent these individual executions it is necessary to generate all possible combinations of branches being true and false, inverting their conditions when they are supposed to be false. This way, if multiple conditions are true, only their combined transition can be fired. Applying this transformation to the event described in Figure 2 results in the more complex event in Figure 1.

The combined conditions for each path are then taken as a guard condition for the corresponding transition. In Figure 3 for example, the upper transition can only be triggered by the first player ($P_x = P_1$) and the lower one by any player once the door is open ($Open = true$). The places for helper variables are connected to the transition if they are read in the condition, with both an incoming and outgoing edge (*Open* in the lower transition). Both edges are annotated with the same variable name in order to not consume the token.

Regardless of whether the condition differentiates between the players, there are also arcs connected to the room the event is triggered from (e.g., the location of the object it is attached to) because the player must be there in order to trigger it (*Hall* for the upper transition, *Storage* for the lower one). And if an object can be disabled (in which case its events become unavailable) there is also an implicit boolean variable added to its conditions that represents the object's and therefore also the transition's availability.

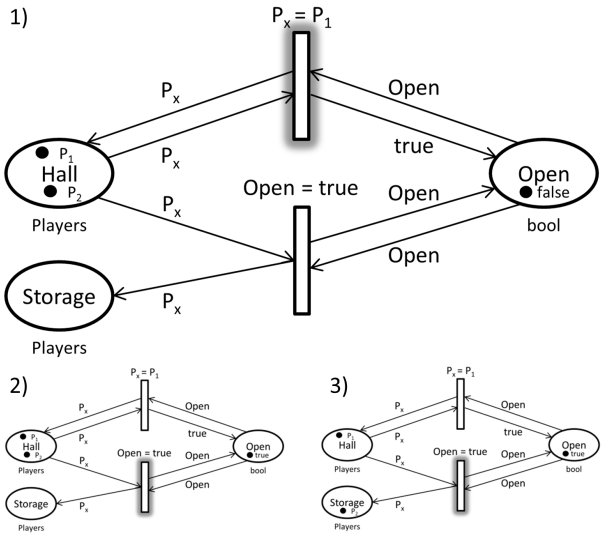

Figure 3: An example net. The door between the *Hall* and the *Storage* room can be unlocked by player one (P_1), with its state being saved in the variable *Open*. Once that is done that any player can pass through. In the initial state (1) only the action for changing the variable value is enabled. After one step (2) the value of *Open* has changed to true, and the action for moving to the other room becomes enabled. After one player token has changed its location, the second one is still able to follow (3).

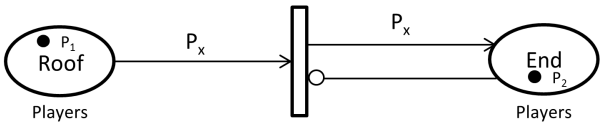

Figure 4: An example net. The second player (P_2) has triggered the ending, moving his token to the *End* place. The inhibitor arc (marked by a small circle) now prevents the transition from firing again.

Representing the effects of actions is done differently depending on the type of action. If a variable is set, the outgoing arc for this variable is annotated with the new value (*Open* is set to *true* in the upper transition). When the variable is not read in the condition, an incoming arc must also be added. In case of player transition the outgoing arc is connected to the room the player will move to instead of looping back (moving from *Hall* to *Storage* in the lower transition). If objects are enabled or disabled this action is interpreted as writing the implicit variable attached to it. Other actions that do not change the game state (like playing a sound) are not important for calculating structural errors and are therefore not modeled.

In order to check if a valid game ending has been reached a special *End* place is added. If a player triggers any ending his or her token is moved to this place. Adding so called inhibitor-arcs from this place to all transitions prevents them from firing once there is a token inside the *End*, forcing the Petri Net into a static state (Figure 4).

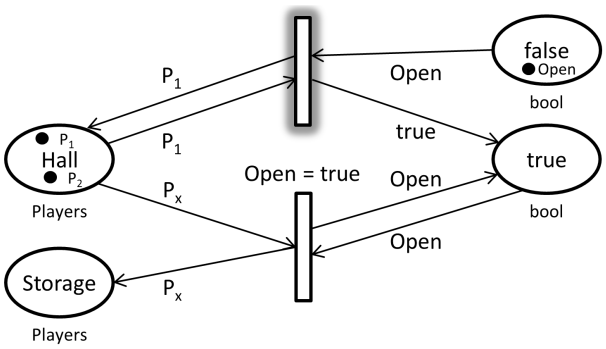

Figure 5: An example net using the alternate model for the same situation as described in Figure 3.

4.3 Alternate model

An alternative model could treat the variables the same way as the players. Instead of having one place per variable, there would be one place per variable value (*true* and *false* in Figure 5). The token value would be the variable name instead of its actual value (*Open*). This way conditions have to be modeled by connecting an arc to the expected value and annotating it with the variable name (the lower transition requires *Open* to be in the *true* place). Writing variables is handled like moving players (the upper transition moves *Open* from *false* to *true*).

While this approach would make it easier to view variable value changes, it also results in a much more complex net. Every incoming arc of a transition must be enabled by a token, so the only way to model an *or*-condition would be to duplicate the condition for each possible value. The most efficient way is then to transform the condition into the disjunctive normal form and splitting it on each remaining *or*, leaving only *and*s. Range conditions like "$count > 4$" or setting a variable from any value would also require lots of duplicates, one for each possible input value. Having multiple *or*s or variables with range conditions multiplies the number of duplicate transitions that must be created since each possible combination must be covered. Furthermore it is easy to see that this approach is only possible with variables that have discrete and strictly limited value ranges because each possible value must be modeled explicitly. Together with the fact that it results in much more complex nets for larger example games (minor decrease in places, major increase in both transitions and arcs) it was therefore decided to not use this approach.

4.4 Complexity

The complexity of the Petri Net resulting from a specific game can be estimated as follows:

The number of colors is limited by the number of supported variable types, often bool, integer, float and string. Together with the player color there are five colors, which can be assumed as constant: $\Sigma = \{c_{Player}, c_{Bool}, c_{Int}, c_{Float}, c_{String}\}$.

The number of places consist of the number of rooms and variables combined: $P = \bigcup_{r \in Rooms} p_r \cup \bigcup_{v \in Variables} p_v$. It grows linearly when more are added to the game.

The number of transitions is defined by the number of events and all paths through them: $|T| = \sum_{e \in Events} path(e)$. With exclusive branching there are exactly as much paths as there are leafs ($|path(e)| = |leafs(e)|$), but with non-exclusive branching this number increases to every combination of branches (in the worst case $|path(e)| = 2^{|branches(e)|}$).

The number of arcs per transition is dependent on the number of variables and players that are checked in its conditions or are modified by the corresponding event, with two arcs for each of them. Aside from that there is also one inhibitor arc per transition: $|A| = \sum_{t \in T}((|usedVariables(t)|+1)*2+1)$.

5. OPTIMIZATION STRATEGIES

In order to reduce the complexity of the resulting net several measures can be taken when exporting a game. First of all irrelevant elements can be removed by an iterative optimization. This includes actions that do not change the game state (e.g. playing a sound), objects with no actions attached and (implicit) variables which are not read. Since removing a variable makes actions setting it irrelevant which in turn might remove the only action associated with a certain object, these optimizations are highly dependent on each other and are therefore repeated until a static state is reached (i.e., nothing can be removed anymore). After all optimization steps, the example event in Figure 1 would only contain one action and condition: if (player = 1) then (Move box).

After that, several one-time optimizations are available. Especially when working with a scene hierarchy it may happen that there are no transitions moving a player to a parent scene if it only exists as a container for shared objects. This can already be detected during optimization, displaying a warning for the user and removing such scenes from the net. This makes it easier to notice semantically unreachable scenes (i.e., ones that have a connection for players but with a condition that will never be true). Also when combining several conditions during event transformation there might be some cases where individual checks contradict each other (e.g., $a < 5 \wedge a > 7$), are always true (e.g., $a < 5 \vee a > 4$) or where one part is completely included in the other (e.g., $a < 5 \wedge a < 4$). These can also be detected during export and are either simplified or removed completely.

Only optimization steps that can be taken by looking at static elements in the game were implemented. Taking dynamics (i.e., the order in which they are executed) into account would require verification or simulation on its own. This would greatly increase export time while ultimately just shifting effort from the actual verification step to the export phase.

6. RESULTS

In order to validate the concept, a prototypical export using the transformation steps described in Section 4, including the optimization strategies mentioned in Section 5, has been implemented. It is directly integrated into the authoring tool *StoryTec* [9], so any game created with the tool can be automatically transformed into a colored Petri Net. The resulting net is then exported using an xml-format that can be read by *CPN-Tools*[7].

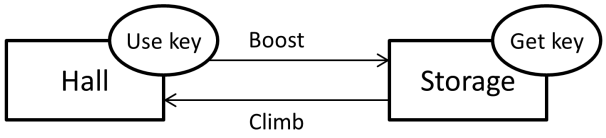

Figure 6: Example scenario: Boosting another player to retrieve an object.

Table 1: Error class indicators

Testcase	Dead markings	Home space
Alive	Ending only	Yes
Deadlock	Other than ending	Yes
Livelock	None	No
Unreachable	(Ending only)	(Yes)

In order to save export as well as verification time and to reduce the net complexity the user is asked whether the tool should assume exclusive conditions, in which case the export simplifies the condition transformation step. However, assuming exclusivity when there is none could produce false positives, i.e., the verification process finding errors in games that are in fact solvable. Therefore, the decision to simplify should only be taken when the designer is absolutely sure that all conditions are exclusive. Although it would be possible to detect exclusivity automatically, doing so is not trivial, especially when it is dependent on the interplay of multiple scenes and variables. As with a more complex optimization, this would require a verification on it's own. Assuming non-exclusivity in an exclusive net in contrast only increases export time, as additional path combinations are generated and immediately removed during optimization as their conditions are mutually exclusive.

6.1 Toy examples

To verify that each type of error is detected correctly, a toy example has been created (Figure 6). In this puzzle one player has to be boosted by the other in order to reach a second room through a vent, retrieve a key there, and come back using a ladder. While this example is perfectly solvable, it can be easily modified by removing the ladder and therefore also requiring a boost on the way back. But since the second player cannot follow the first one without a boost too, both players are stuck in their rooms (deadlock). Modifying this example again by splitting the second room into two sections allows the player to move between those, but not reach the end (livelock). Lastly, there was also an example based on the first one with a transition to an optional scene. However, the condition for this transition was impossible due to the variable never being set to true. This made the optional scene unreachable while keeping the game solvable.

When analyzing the exported Petri Nets with *CPN-Tools* there are clear indicators connected to three out of four error classes (Table 1). All dead states (markings) should be checked first by running the command `ListDeadMarkings()`. This command displays a list of numeric state identifiers. When reaching those states the net has no enabled transitions anymore and has therefore reached a static state. Since the special place for the game ending has inhibitor arcs to

every transition, any player triggering the ending (i.e., any player token moving into the game ending place) immediately results in such a state. Therefore, the list of dead markings must consist only of states in which the ending place holds one player token if the game is always solvable. This can be checked by switching the graphical representation of the net to each of these states via their identifiers. Should there be dead states where the ending has not been reached, the same tool can be used to investigate the predecessor nodes in the state space and trace what had happened beforehand.

If there are no dead markings the ending cannot be reached due to a livelock and if there are markings without a player token having reached the end (exclusively or in addition to real endings) those are deadlocks. Having only ending states however means that the game only stops if an ending has been reached – there might still be some paths resulting in livelocks. In order to get conclusive results it is therefore important to also check if those dead markings create a home space with the command `HomeSpace(ListDeadMarkings())`. If the result is *true*, then there is a path from every reachable state to at least one of the dead markings. If this is the case there are no livelocks, else one should export a graphical representation of the state space and look for cycles in order to find the livelocks.

Unreachable scenes and impossible conditions that do not impede progress through the game are not visible at first using this approach. In order to detect those the net must be analyzed further, looking for place bounds (if the upper bound of a scene place is zero it cannot be reached) and dead transitions (which can never fire). Yet, it is important to note that dead transitions may not always be a user error since they might represent additional paths generated by the export handling non-exclusivity, so users are advised to take a closer look at this part of the net. Verifying the toy examples, all indicators could be observed as expected.

6.2 Real world examples

After having shown that all relevant error classes can be detected using this approach, a number of existing games have also been tested in order to investigate how the algorithms will scale with real life examples (Table 2). In order to compare nets assuming non-exclusive conditions to others assuming exclusive ones, only games using the latter have been chosen because they produce valid results in both cases.

The first example is a multiplayer adventure game for two players with a playtime of about 15 minutes [16]. It consists of 13 rooms with four complex puzzles consisting of several sub-tasks. These span multiple rooms and require both players to collaborate. After solving the first puzzle the players are able to move freely back and forth through the game world and solve the rest of the tasks in any order, resulting in a large possibility space. From a scripting perspective most of the sub-tasks are modeled using boolean variables, for example a missing screwdriver is either available or must still be found by the players.

Two tasks stand out, however, one requiring the players to align a satellite dish. The dish has several states modeled by using an integer variable resulting in different signal strengths the players must interpret. The other one has players connecting four wires with four sockets, which results in $4! = 24$ possible solutions not counting intermediate states. During both tasks the players can move farther away from the solution by disconnecting wires or turning the dish in the wrong direction. Several versions of the game have been investigated, the full game, a version without the wire puzzle and a short one with both the wire and the satellite puzzle being replaced by a trivial action each. The wire puzzle was tested separately as well.

The other two games are singleplayer games (i.e., have a player count of one) as the approach has to works for any number of players. They both share the same basic structure, beginning with an intro and then using a central map from which the players can chose in which order to solve six minigames. Those minigames include puzzles, hidden object games and memory games and are identical in both games. It is important to note here that these minigames are based on predefined interaction templates that are not scripted using the authoring tool, so these games are a black box for the Petri Net export and it is assumed that they work like a button (i.e., the user can click / solve it if they see it). After all tasks are solved a short outro is played. While the vacation game takes about five minutes to be solved, the tourism game has a lot more (linear) story elements and therefore takes about eight minutes to be completed.

Our results show that while the generated nets are quite complex, as it is expected for fully modeled games, it is still a viable option to analyze the games at certain points during their development. Exporting the games to a Petri Net can be done almost instantly when exclusive conditions are assumed and within a few seconds when all combinations of conditions must be calculated. Furthermore when having more complex games and larger nets export time scales linearly since each element is exported individually without a particular order.

In contrast to the export time, verification time increases exponentially as the nets grow larger because the verification must take into account all possible combinations and orders of events. As such, the verification cannot always be done in real-time. For our evaluation we defined a cut-off point of 15 minutes, although depending on the application scenario developers might accept longer calculation times (similar to user tests). In this time frame the large adventure game could not be verified. Removing the most complex puzzle brings the verification time down to 44 seconds and the puzzle itself was verified after 4 seconds. Therefore, it is highly advisable to split larger games into smaller sections and analyze them independently. As doing so removes all playthroughs where players switch between those sections during intermediate solutions from the possibility space, designers have to make sure that they cut the game into parts that do not have side effects on each other.

When the game actually contains structural errors, export time and net complexity are marginally reduced. As the reachability calculation is done during verification, every element is exported even if it cannot be reached. Depending on the cause of the error, a few action branches might be

Table 2: Test results for real world examples

Game	Playtime	Players	Places	Transitions	Arcs	Export (non-excl.)	Verification
Adventure (full)	ca. 15 min	2	54	131	1141	0 s (1 s)	> 15 min
Adventure (without wires)	ca. 12 min	2	41	72	400	0 s (0 s)	44 s
Adventure (wires only)	ca. 3 min	2	18	59	737	0 s (1 s)	4 s
Adventure (simple)	ca. 10 min	2	39	54	292	0 s (0 s)	5 s
Tourism	ca. 8 min	1	44	67	241	0 s (0 s)	4 s
Vacations	ca. 5 min	1	29	28	144	0 s (0 s)	3 s

culled during optimization due to having impossible conditions, but this reduction can be neglected in comparison to the overall complexity. Verification time in contrast can vary greatly: If the error (deadlock, livelock or unreachable element) appears towards the start of the game, it can reach only few game states and the verification time is reduced to a minimum. If the error is located towards the end of the game, the state space is reduced only by a small amount. In this case the verification time is nearly as long as when the same game did not contain the error. We could verify this behavior as the adventure did actually contain a livelock at some point during development (for comparison reasons the final measurements where taken with an updated version).

Putting the Petri Net size in relation to the verification time the "Adventure (without wires)" example has almost as much places (which is directly related to the game states) and transitions (game events) as the "Tourism" example. Despite that its verification time is much higher due to its more open structure. One can easily see that as an extreme example a linear game without branches would result in a single path to be verified, so the size of the net does not provide a good estimate on how difficult verification is.

Another observation is that assuming non-exclusive trees in these examples does not increase the size of the resulting net. This was to be expected, since the games were designed to be exclusive. The increase in export time when assuming non-exclusive trees shows that it is nevertheless advisable to design the game in such a way that conditions are always exclusive. Doing so trades a little more work beforehand for time savings during verification and a Petri Net that is easier to read because the guard conditions are not modified during export.

7. CONCLUSION

In this paper we described our approach for detecting structural errors like deadlocks, livelocks or unreachable states in scene-based single- and multiplayer games using Petri Nets. For this purpose a mapping of game elements like rooms, players and variables to Petri Net concepts has been developed. After that we analyzed the theoretical complexity of the resulting nets before describing optimization techniques and the prototypical implementation as an extension for an existing authoring tool. With this tool users are now able to automatically export the game structure as a Petri Net for verification using an external tool.

Compared to creating a net manually based on the intended game design this is much less error-prone and time-consuming. Still, it must to be noted that the automated model transformation has to be implemented correctly in order to get the right mapping. We would argue however, that ensuring the functionality of a generic export for a limited number of primitives once is much more efficient than doing the same for multiple games repeatedly using these primitives.

The approach is able to include every possible user interaction into the net, not only high level elements like complete missions. Our evaluation showed that all relevant error classes can be found by this approach. Using real world examples we also found that the approach is able to provide its results in a timely manner for short games, but in order to scale for larger projects the games must be broken down into individual parts that can be verified independently. Currently this has be done by the designer, although future work could include a heuristic for suggesting promising splits. It would also be beneficial to automate as many steps of the subsequent Petri Net analysis as possible, for example eliminating the need for designers to check static states manually, making the overall process faster and easier to understand.

Although the implementation is integrated directly into a specific authoring tool, the approach itself can easily be used for similar tools, engines or individual games as it is designed to support scene-based games in general. It is also possible to use the general concept with other game types as long as a Petri Net model can be defined for each feature. Last but not least the current implementation could also be used in order to build a high-level model for other games built outside of the tool, although doing so designers would lose one of the main advantages of our concept – the automatic matching of the model with the game. Nevertheless, applying the transformation process to other existing games will also be part of our future work.

As it is intended for scene-based games, however, the approach only works for games with a limited number of discrete states. In order to verify games with continuous movement for example the player position has to be reduced to a limited number of equivalence classes (similar to having rooms in the first place), which would also destroy the direct matching between model and game. Finally is also important to note that verifying functional properties does only mean that players using a brute-force approach (i.e., systematically clicking on everything) can reach an ending. It does not mean that they have enough information to solve the game's challenges via thinking, which is dependent on the game's content and not its structure. And it also does not mean that the puzzles can only be solved as intended in the design. Therefore, the games should still be evaluated with real users after their theoretical solvability has been verified.

8. REFERENCES

[1] T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, and Y. Xie. Zing: A Model Checker for Concurrent Software. In R. Alur and D. Peled, editors, *Computer Aided Verification SE - 42*, volume 3114 of *Lecture Notes in Computer Science*, pages 484–487. Springer Berlin Heidelberg, 2004.

[2] M. Araújo and L. Roque. Modeling games with petri nets. *Breaking New Ground: Innovation in Games, Play, Practice and Theory. DIGRA2009.*, 2009.

[3] T. Carron, F. Kordon, J.-M. Labat, I. Mounier, and A. Yessad. Toward Improvement of Serious Game Reliability. In *7th European Conference on Games-Based Learning*, pages 80–87. Academic Conferences and Publishing International, 2013.

[4] R. Champagnat, A. Prigent, and P. Estraillier. Scenario building based on formal methods and adaptative execution. *ISAGA, Atlanta (USA)*, 6, 2005.

[5] J. Dormans. Machinations: Elemental Feedback Patterns for Game Design. In J. Saur and M. Loper, editors, *GAME-ON-NA 2009: 5th International North American Conference on Intelligent Games and Simulation*, pages 33–40, 2009.

[6] K. Jensen. Coloured petri nets: A high level language for system design and analysis. In G. Rozenberg, editor, *Advances in Petri Nets 1990 SE - 13*, volume 483 of *Lecture Notes in Computer Science*, pages 342–416. Springer Berlin Heidelberg, 1991.

[7] K. Jensen, L. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007.

[8] Lucasfilm Games. Maniac Mansion, 1987.

[9] F. Mehm, S. Göbel, S. Radke, and R. Steinmetz. Authoring Environment for Story-based Digital Educational Games. In Y. Cao, A. Hannemann, B. Fernández-Manjón, S. Göbel, C. Hockemeyer, and E. Stefanakis, editors, *Proceedings of the 1st International Open Workshop on Intelligent Personalization and Adaptation in Digital Educational Games*, pages 113–124. CEUR Workshop, 2009.

[10] P. Moreno-Ger, R. Fuentes-Fernández, J.-L. Sierra-Rodríguez, and B. Fernández-Manjón. Model-checking for adventure videogames. *Information and Software Technology*, 51(3):564–580, Mar. 2009.

[11] S. Natkin, L. Vega, and S. M. Grünvogel. A new methodology for spatiotemporal Game Design. In *Proceedings of CGAIDE*, pages 109–113, 2004.

[12] T. Nummenmaa, J. Kuittinen, and J. Holopainen. Simulation As a Game Design Tool. In *Proceedings of the International Conference on Advances in Computer Enterntainment Technology*, ACE '09, pages 232–239, New York, NY, USA, 2009. ACM.

[13] J. C. Osborn, A. Grow, and M. Mateas. Modular Computational Critics for Games. In *AIIDE*, 2013.

[14] C. J. F. Pickett. Formal Verification of Computer Narratives. page 13, 2005.

[15] M. Purvis. Narrative structures for multi-agent interaction. In *Intelligent Agent Technology, 2004. (IAT 2004). Proceedings. IEEE/WIC/ACM International Conference on*, pages 232–238, 2004.

[16] C. Reuter, V. Wendel, S. Göbel, and R. Steinmetz. Multiplayer Adventures for Collaborative Learning With Serious Games. In P. Felicia, editor, *6th European Conference on Games Based Learning*, pages 416–423. Academic Conferences Limited, 2012.

[17] M. A. Syufagi, M. Hariadi, and M. H. Purnomo. Petri Net Model for Serious Games Based on Motivation Behavior Classification. *International Journal of Computer Games Technology*, 2013:1–12, 2013.

[18] R. Tagiew. Multi-Agent Petri-Games. In *International Conference on Computational Intelligence for Modelling Control & Automation*, pages 130–135, 2008.

[19] C. Verbrugge. A Structure for Modern Computer Narratives. In J. Schaeffer, M. Müller, and Y. Björnsson, editors, *Computers and Games SE - 21*, volume 2883 of *Lecture Notes in Computer Science*, pages 308–325. Springer Berlin Heidelberg, 2003.

[20] A. Yessad, P. Thomas, B. Capdevila, and J.-M. Labat. Using the Petri Nets for the Learner Assessment in Serious Games. In X. Luo, M. Spaniol, L. Wang, Q. Li, W. Nejdl, and W. Zhang, editors, *Advances in Web-Based Learning - ICWL 2010*, volume 6483 of *Lecture Notes in Computer Science*, pages 339–348. Springer Berlin Heidelberg, 2010.