



TECHNISCHE
UNIVERSITÄT
DARMSTADT

AUTHORING COLLABORATIVE MULTIPLAYER GAMES –
GAME DESIGN PATTERNS, STRUCTURAL VERIFICATION,
COLLABORATIVE BALANCING *and* RAPID PROTOTYPING

Vom Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
genehmigte Dissertation

von

CHRISTIAN REUTER, M.Sc.

Geboren am 06. Februar 1987 in Bad Soden

Vorsitz: Prof. Dr.-Ing. Rolf Jakoby
Referent: Prof. Dr.-Ing. Ralf Steinmetz
Korreferent: Prof. Dr.-Ing. Wolfgang Effelsberg

Tag der Einreichung: 31. Mai 2016
Tag der Disputation: 25. Juli 2016

Hochschulkennziffer D17
Darmstadt 2016

Dieses Dokument wird bereitgestellt von
tuprints, E-Publishing-Service der Technischen Universität Darmstadt.
<http://tuprints.ulb.tu-darmstadt.de>
tuprints@ulb.tu-darmstadt.de

Bitte zitieren Sie dieses Dokument als:

URN: <urn:nbn:de:tuda-tuprints-56167>

URL: <http://tuprints.ulb.tu-darmstadt.de/id/eprint/5616>

Die Veröffentlichung steht unter folgender Creative Commons Lizenz:
International 4.0 – Namensnennung, nicht kommerziell, keine Bearbeitung
<https://creativecommons.org/licenses/by-nc-nd/4.0/>



ABSTRACT

VIDEO games are not only a growing business field, but also provide interesting research opportunities. In particular, collaborative multiplayer games have become increasingly popular with players, developers and researchers alike. These games offer players the opportunity to tackle difficult challenges together instead of competing against each other. Collaboration within a game not only increases player interest and developer revenue, research has also shown that it can have positive effects on the players' social skills.

The rise of collaborative games, however, also has a downside. As more developers create this kind of game, instances in which players complain about the collaborative elements being implemented halfheartedly are becoming more and more frequent. At least in part, this can be attributed to the fact that developing collaborative multiplayer games introduces unique challenges to the game development process. The players must not only get enough opportunities to interact with each other in a meaningful way, but it is also essential that their contributions are similar. Moreover, testing the game becomes more difficult due to a higher number of testers required. But even when there are enough testers, the complexity of the state space grows exponentially with each new player. This means that human testers are often unable to test every combination of events. These aspects make it much harder to develop collaborative games, especially for small and inexperienced teams.

Although these issues are well known, the current state of the art is only partially able to solve them. For example, there are guidelines on how to develop collaborative games – but these only give general ideas. Therefore, they are not always directly implementable, especially for developers that are new to multiplayer development.

This thesis aims at supporting those developers by conceptualizing an authoring environment that addresses these issues. Its overall concept consists of three steps or four modules: Game design patterns as player interaction templates (1), a formal analysis concerning structural errors (2a) and collaborative balancing (2b) as well as a rapid prototyping environment (3).

To help developers with the initial design, a number of well-received player interactions, which can be seen as the central element of a collaborative game, is gathered. These interactions are described using the well-known format of game design patterns. In order to make the patterns more user-friendly, the format is extended with properties specific to collaborative interaction. For example, one property describes whether the players must be close to each other in order to trigger the interaction. Following this, a representative selection of the patterns is used to develop a game, in which collaboration can be switched off. This game is evaluated in a user study later. Here, the outcomes have shown that the interactions are well received by the players.

If combined in the wrong way, however, these interactions can cause structural issues such as deadlock situations, i.e. a state from which the game can no longer be completed. In order to detect such problems, verification algorithms can be used. Those require the transformation of the game into a formal model. As such, colored

petri nets are chosen, since they can model concurrent actions by multiple players. Since developers cannot usually be expected to create such a model by themselves, an automated transformation approach is devised. For that, rules between each element of the underlying game model and the elements of a colored petri net are defined. Additionally, optimization rules are developed to mitigate the state space explosion when verifying complex games. These rules use knowledge about the game model in order to reduce the complexity of the resulting petri net. Evaluations with both existing games and synthetic examples confirm that such a verification approach is viable for smaller games.

Whether the game involves all players on an equal basis can be checked in tandem to this. However, a novel definition for balancing in collaborative games needs to be created first, as the related work only examines balancing from a competitive perspective. Based on this definition, concrete measurements to indicate disparities between the players are devised. As the games' complexity and the players' influence on the course of the game prevent exact measurements, an approximation approach is instead developed. Similar to the structural verification, an evaluation shows that this approach works for smaller games.

Finally, a rapid prototyping environment is designed, which allows a single developer to test games designed for up to four players. For this, visual and audio information for separate players is made both observable and, at the same time, clearly attributable. Additionally, simultaneous player actions can be simulated by using a record and replay approach.

By implementing and evaluating these modules, this work is able to show how development issues that are specific to collaborative multiplayer games can be addressed. Using these modules can therefore reduce the effort involved in developing collaborative multiplayer games.

KURZFASSUNG

VIDEO SPIELE repräsentieren nicht nur einen kommerziell wachsenden Markt, sondern stellen auch ein interessantes Forschungsfeld dar. Insbesondere kollaborative Mehrspieler-Spiele werden immer beliebter, sowohl bei Spielern und Entwicklern als auch in der Forschung. Diese Art von Spielen ermöglicht es den Spielern, Herausforderungen zusammen mit ihren Freunden zu bewältigen anstatt sich mit diesen im Duell zu messen. Kollaboration in Spielen kann dabei nicht nur das Interesse der Spieler und die Absatzzahlen der Entwickler erhöhen. Wissenschaftliche Studien haben außerdem gezeigt, dass diese Zusammenarbeit positive Auswirkungen auf die sozialen Fähigkeiten der Spieler haben kann.

Die wachsende Beliebtheit von kollaborativen Spielen hat aber auch eine Kehrseite. Während immer mehr Entwickler diese Art von Spielen erstellen, mehren sich gleichzeitig die Fälle, in denen sich Spieler über nur halbherzig umgesetzte Kollaborationselemente beschweren. Zumindest partiell kann dies damit erklärt werden, dass die Entwicklung dieser Art von Spielen besondere Herausforderungen beinhaltet. Beispielsweise sollten die Spieler nicht nur sinnvoll miteinander interagieren können, sie sollten auch vergleichbare Beiträge zur Lösung des Spiels liefern. Auch das Testen des Spieles wird alleine dadurch schwieriger, dass eine größere Anzahl von Spielern benötigt wird. Selbst wenn genug Testspieler zur Verfügung stehen, steigt jedoch die Komplexität des Zustandsraumes durch jeden weiteren Spieler. Daher ist es menschlichen Testspielern oft nicht möglich alle denkbaren Varianten des Spiels zu durchlaufen. All diese Aspekte erschweren die Entwicklung von kollaborativen Spielen, insbesondere für kleine und unerfahrene Entwicklerstudios.

Obwohl diese Herausforderungen bekannt sind, gibt es aktuell nur teilweise Lösungsansätze. Beispielsweise gibt es Empfehlungen für die Entwicklung von kollaborativen Spielen – jedoch beinhalten diese meist nur grundsätzliche Aspekte. Sie sind daher oft nicht direkt umsetzbar, insbesondere wenn die Entwickler vorher noch keine Erfahrungen mit Mehrspieler-Spielen gesammelt haben.

Um genau diese Entwickler zu unterstützen, wird im Rahmen dieser Dissertation eine Autoren-Umgebung konzipiert, die diese Probleme gezielt adressiert. Das Konzept hierfür gliedert sich in drei Stufen beziehungsweise vier Module: Game Design Patterns als Vorlagen für Spieler-Interaktionen (1), eine formale Analyse bezüglich struktureller Fehler (2a) und kollaborativem Balancing (2b) sowie eine Umgebung die das Testen des Spiels vereinfacht (3).

Um den Entwicklern einen Einstieg zu bieten, wird eine Menge von beliebten Spielerinteraktionen identifiziert, die als das zentrale Element eines kollaborativen Spieles gesehen werden können. Diese werden anschließend im bekannten Format der Game Design Patterns beschrieben. Um deren Nutzung zu vereinfachen, wird das Format vorher um zusätzliche, Interaktions-spezifische Eigenschaften erweitert. Beispielsweise wird so beschrieben, ob sich die Spieler am gleichen Ort befinden müssen, um die Interaktion ausführen zu können. Danach wird eine repräsentative Auswahl der Interaktionsmuster zur Entwicklung eines Beispiel-Spiels eingesetzt, in dem die Kollaboration auch ausgeschaltet werden kann. Dieses Spiel wird anschlie-

ßend mit Hilfe einer Benutzerstudie evaluiert. Diese zeigt, dass die Interaktionen von den Spielern positiv bewertet werden.

Bei ungünstiger Kombination dieser Interaktionen können strukturelle Probleme wie Zustände, in denen ein Spiel nicht mehr regulär beendet werden kann, entstehen. Zur Erkennung derartiger Probleme können Verifikationsalgorithmen eingesetzt werden. Für diese muss das Spiel in ein formales Modell überführt werden. Hierfür werden farbige Petri Netze gewählt, da diese parallele Aktionen von mehreren Spielern abbilden können. Da man von einem Entwickler nicht erwarten kann, dass dieser selbst ein entsprechendes Modell erstellt, wird zudem eine automatische Übersetzung entworfen. Diese beinhaltet Übersetzungsregeln für jedes Element des zugrundeliegenden Spiel-Modells in Petri Netz Elemente. Um der Zustandsexplosion bei der Verifikation von komplexen Spielen entgegenzuwirken, werden zusätzliche Optimierungsregeln entwickelt. Die Regeln nutzen dabei Wissen über das Spielmodell um die Größe des entstehenden Petri Netzes von vornherein zu reduzieren. Eine Evaluation mit realen und synthetischen Beispielen zeigt, dass eine derartige Verifikation für kleinere Spiele möglich ist.

Gleichzeitig sollte geprüft werden, ob das Spiel alle Spieler gleichermaßen fordert. Hierfür muss zunächst eine neue Definition für Balancing in kollaborativen Spielen entwickelt werden, da die verwandten Arbeiten lediglich eine kompetitive Perspektive einnehmen. Basierend auf dieser Definition werden konkrete Metriken entwickelt, die Ungleichheiten zwischen den Spielern aufzeigen können. Da die Komplexität der Spiele und der Einfluss der Spieler auf den Spielablauf jedoch eine exakte Berechnung unmöglich machen, wird stattdessen eine Abschätzung entwickelt. Analog zur strukturellen Verifikation kann per Evaluation gezeigt werden, dass dieser Ansatz für kleinere Spiele funktioniert.

Zuletzt wird eine Umgebung für schnelle Tests entwickelt, die es sogar einzelnen Autoren, ein Spiel für bis zu vier Spieler zu testen. Dazu werden optische und akustische Informationen für verschiedene Spieler so vermittelt, dass sie gleichzeitig verfügbar und trotzdem klar zuordenbar sind. Simultane Aktionen mehrerer Spieler können zudem per Aufzeichnung von Eingaben simuliert werden.

Durch die Implementierung und Evaluation dieser Module kann diese Arbeit zeigen, dass dadurch Probleme angegangen werden können, die für kollaborative Mehrspieler-Spiele spezifisch sind. Die Nutzung der Module bietet daher das Potential den Erstellungsaufwand für kollaborative Mehrspieler-Spiele zu reduzieren.

CONTENTS

1	INTRODUCTION	1
1.1	Motivation.....	1
1.2	Challenge	2
1.3	Contribution.....	4
1.4	Outline	4
2	RELATED WORK	5
2.1	Authoring Multiplayer Games	5
2.2	Designing Multiplayer Games	6
2.2.1	Collaborative and Cooperative Multiplayer Games	7
2.2.2	Game Mechanics for Collaborative Interactions	8
2.3	Game Design Patterns	9
2.3.1	Formalization	10
2.3.2	Pattern Generation	12
2.3.3	Patterns for Collaborative Multiplayer Games	12
2.4	Formal Verification of Games	14
2.4.1	General verification approaches	14
2.4.2	Verification using Petri Nets	15
2.5	Game Balancing	17
2.5.1	Balancing Calculation and Adaptation.....	19
2.6	Rapid Prototyping	20
2.7	Identified gap	20
3	OVERALL CONCEPT AND APPROACH	23
3.1	Application Scenario	23
3.2	Research Questions	24
3.3	Overall Concept for Authoring Collaborative Multiplayer Games	24
4	TECHNICAL FOUNDATIONS	27
4.1	Underlying game model	27
4.2	Multiplayer Authoring	29
4.3	Multiplayer Runtime Environment	30
4.4	Multiplayer Adventure Game	32
5	GAME DESIGN PATTERNS FOR PLAYER INTERACTIONS	33
5.1	Collaborative Player Interactions	33
5.2	Collecting Interaction Instances	35
5.3	Interaction Classification and Post-Processing	35
5.4	Design Pattern Formalization	38
5.5	Procedural Content Generation Using Patterns	39
5.6	Design of an Evaluation Game	41
5.7	Summary	44

6	STRUCTURAL VERIFICATION OF COLLABORATIVE MULTIPLAYER GAMES	45
6.1	Problem Definition	45
6.2	Formal Game Model	46
6.3	Petri Net Model	47
6.3.1	Game State	48
6.3.2	Game Events	49
6.3.3	Extensions	52
6.3.4	Alternate Model	57
6.4	Properties and Complexity	58
6.5	Optimization Strategies	60
6.6	Game Generator	63
6.7	Summary	65
7	COLLABORATIVE BALANCING IN MULTIPLAYER GAMES	67
7.1	Collaborative Balancing Definition	67
7.2	Approximating Balance	68
7.2.1	Path sampling	69
7.2.2	Path assessment	73
7.2.3	Value Aggregation	79
7.2.4	Value Distribution	82
7.2.5	Extension: Multiple shortest paths	90
7.2.6	Extension: Culling End States	91
7.3	Game Generator	92
7.4	Summary	93
8	RAPID PROTOTYPING OF COLLABORATIVE MULTIPLAYER GAMES	95
8.1	Requirements	95
8.2	Visualization	96
8.3	Internal Adaptation Models	97
8.4	Sound	99
8.5	Input	100
8.6	Record and Replay	101
8.7	Game Interface	102
8.8	Summary	102
9	PRACTICAL IMPLEMENTATION	105
9.1	Game Design Patterns	105
9.2	Structural Verification	106
9.3	Collaborative Balancing	109
9.4	Rapid Prototyping	113
10	EVALUATION	115
10.1	Game Design Patterns	115
10.2	Structural Verification	121
10.3	Collaborative Balancing	130
10.4	Rapid Prototyping	141
10.5	Discussion	141

11 CONCLUSION	143
11.1 Main Contributions.....	143
11.2 Outlook	145
BIBLIOGRAPHY	147
LIST OF FIGURES	159
LIST OF TABLES	161
A PATTERN LANGUAGE COMPARISON	163
B ANALYZED GAMES	165
C GAME DESIGN PATTERN COLLECTION.....	167
D DETAILED BALANCING CALCULATIONS.....	181
E GAME QUESTIONNAIRE.....	185
F FULL EVALUATION RESULTS.....	195
G SUPERVISED STUDENT THESES	203
H AUTHOR'S PUBLICATIONS	205
I CURRICULUM VITÆ.....	207
J ERKLÄRUNG LAUT §9 DER PROMOTIONSORDNUNG.....	209

INTRODUCTION

OVER the years the games industry has become an important economical factor, with digital sales alone growing to \$61 billion worldwide in 2015 [35]. Additionally, video games are also an important research field in multiple disciplines, as they touch technical as well as social aspects.

Multiplayer games, in which players play together or against each other, are an important subgroup. Repeated samples drawn from Steam¹, the largest digital distribution platform for PC games, show the popularity of such games. Each time, between 40 and 50% of the top selling games were marked as multiplayer games. Developing a multiplayer game also includes some interesting technical (multiple active players) and design (involvement of differently skilled players) challenges to be researched.

1.1 MOTIVATION

»In only half a decade we've gone from famine to feast
in terms of co-op offerings in games.«

— Nick Puleo [88]

Collaborative/cooperative games² are multiplayer games in which the players are working together instead of competing. Such games have become more and more popular among both players and the research community, both as video- as well as boardgames [52]. Additionally, we analyzed data provided by Co-Optimus [31], a large database focusing on cooperative games, and MobyGames [40], a website cataloging as many games as possible. This analysis did not only confirm that the absolute number of cooperative games increase over the recent years, but their percentage in comparison to the overall releases also grew (Figure 1).

Aside from being relevant due to their popularity, cooperative and collaborative games can provide additional benefits. On the one hand, playing with friends is generally more fun for most players, on the other, they can use their assistance to overcome difficult challenges they could not solve on their own. In addition, they can even use this as an opportunity to teach the game to their peers. Doing so can consequently help them to work on their social skills, practice teamwork and enable collaborative learning [122], which is more effective than learning alone. It has also

¹ <http://www.steampowered.com>

² Some authors differentiate between collaborative and cooperative games, others only use the term “cooperative” when players are working together. Those who use both terms usually differentiate them based on the players’ goals: The goals of all players align in collaborative games, whereas they differ slightly in cooperative games [132]. This means that in collaborative games, players work together all the time, while in cooperative games this might happen for only a limited time or in certain aspects. However, this goal-based distinction is irrelevant in the context of this work since it focuses on individual instances of players working together. Therefore, the term “collaborative” will be used – but most findings are also applicable to cooperative games as well.

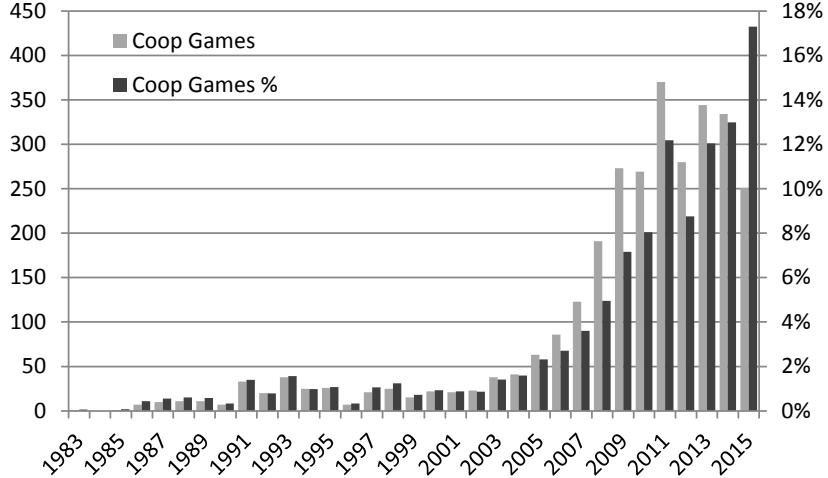


Figure 1: Cooperative games released per year (dark, left scale) and in comparison to overall releases (light, right scale). Data taken from Co-Optimus [31] and MobyGames [40].

been shown that playing collaboratively increases cooperation [13, 45] and decreases violent behavior [120], even after play.

1.2 CHALLENGE

»While making co-op [...] was difficult with a small team, I have to say there was nothing nearly as rewarding as watching a group of players taking on the challenge [...].«

— Tim Keenan [56]

Developing multiplayer games is often more complex than developing singleplayer games [130]. The fact that multiple players interact not only with the game world, but also with each other, increases the complexity of the implementation. It also affects the uncertainty during gameplay due to the influence of other players and complicates organizational aspects such as the number of required testers. Addressing such issues requires specific knowledge and experience, which makes it harder to develop multiplayer games – even for seasoned singleplayer game developers³.

One common pitfall during multiplayer development is that the multiplayer aspects are added too late during development. In such cases, this mode is usually inferior to the singleplayer part [99]. This pitfall could still be observed during the recent influx of collaborative games, where players and professional reviewers complained about games in which the collaboration feels “tacked-on”. In some cases, this impression is caused by a game being designed for a single player initially, with additional players having been added without adapting the overall design. Such feedback makes it clear that collaborative games must be engaging in their own right and that the maxim “everything is better with friends” does not hold for games lacking a certain quality.

³ There is a multitude of roles necessary to develop a game, including programmers, game designers and story authors. This thesis, however, focuses on small teams or even single persons that develop a game, which means that multiple roles are fulfilled by the same person. We therefore use the more general term “developer” to describe any person who is working on a game. Furthermore, only male pronouns are used to improve readability – although developers and players of both genders are addressed.

One crucial aspect that has to be considered is the number and type of collaborative interactions. Many games offer what has been described as simultaneous play, which means that the players play alongside each other [29]. This implies that they only interact with the game world as they would do in a singleplayer game and cannot interact with each other in any meaningful way. This has already been investigated in a study which found that closely coupled interactions are more appreciated than loosely coupled ones [17].

Another aspect in which collaborative games can fail is game balancing. If a singleplayer game design is not extended correctly the game can feel too easy, as there are now more players working on the same tasks. Moreover, there is also the issue of balancing contributions between group members. If one player is faster than the others, it is possible that this player solves the majority of the challenges on his or her own. Not only could this make the other players feel unnecessary and demotivated, it could also cause them to miss some of the game's content when a task is solved before they could even take a look at it. The "better" player might, in turn, complain that the others are not helping – therefore, such a scenario has the potential to leave everyone involved dissatisfied.

Once the game is implemented testing becomes another challenge. In multiplayer games, there are more possibilities for how they can play out due to simultaneous and sometimes unexpected player interactions. Having multiple players also increases the number of potential game states. For example, in a game with n discrete locations and m players there are n^m possible ways the players could inhabit those locations. This often renders it impossible for human playtesters, the current state-of-the-art approach in the industry, to check all of those possibilities.

Additionally, user testing is more difficult for multiplayer games as, by definition, there must be multiple players available. This increases organizational overhead as well as the associated costs. Finally, user testing in general requires a game to have a certain level of fidelity, for example, there must be an understandable representation of the game state as a game without graphics cannot be tested by people. Even using simple placeholders could cause inexperienced testers to get distracted in such a way that they focus purely on this lack of fidelity. However, as in general software development, it becomes more costly to fix problems the later they are discovered. This implies that it is highly important to validate games as early as possible.

The combination of these issues renders it hard to develop multiplayer games, especially for smaller studios with few employees and a low budget. While larger studios can use their workforce to tackle the additional complexity, smaller ones need to neglect other aspects in order to work on the multiplayer elements. Most players, however, will still judge these studios by the same standards as their larger competitors, resulting in them gaining a bad reputation. Although the increased complexity is most problematic for so called "indie developers" consisting of a few people, improving the current state of the art could help larger studios as well. This makes research on improving the quality and decreasing the development effort for (collaborative) multiplayer games generally applicable.

1.3 CONTRIBUTION

Due to these challenges it is the goal of this thesis to support the development of multiplayer games, especially for smaller development teams. The proposed concept is intended to provide design templates focusing on player interactions as the central element, and to give feedback on how balanced a game's collaboration is. Additionally, it should be able to verify that a game has no critical errors on any possible path and to enable quick user tests. However, closing the gap in development effort between single- and multiplayer games completely is not part of the scope of this work, as this also involves many other aspects.

The contribution of this work consists of a development process that combines four modules:

- Collaborative interaction patterns that can serve as building blocks or inspire inexperienced developers to include enough opportunities for interaction.
- Formalized verification of the game's structure to detect problems which cause the game to end unexpectedly ("game-breaking bugs").
- Collaborative balancing assessment that gives feedback on the involvement of each player.
- Rapid prototyping to quickly test multiplayer games without the overhead of recruiting additional players.

1.4 OUTLINE

In Chapter 2, the current state of the art, in relation to multiplayer game development in general and more specific aspects linked to the contribution of this work, is analyzed. This is followed by Chapter 3, which defines the research questions based on the application scenario of this work. It also explains the overall approach linking the individual module concepts. Following this, the conceptual and technical foundations that are integral for understanding this thesis, are described in Chapter 4. The next four chapters describe the concept of the four modules that represents the main contribution in detail: Interaction patterns (Chapter 5), structural verification (Chapter 6), collaborative balancing (Chapter 7) and rapid prototyping environment (Chapter 8). Chapter 9 describes the technical implementation that is provided in order to evaluate this concept and chapter Chapter 10 discusses the results of the subsequent studies. Lastly, a conclusion on the main contributions of this work is given in Chapter 11.

RELATED WORK

THIS thesis tackles the overall topic of authoring collaborative multiplayer games.

Therefore, the topics of authoring (Section 2.1) and multiplayer game design (Section 2.2) serve as the foundations of this work. After that, more specific aspects related to the challenges identified in Section 1.2 are discussed. These include game design patterns (Section 2.3), formal verification (Section 2.4), game balancing (Section 2.5) and rapid prototyping (Section 2.6). However, there are still challenges which have not been addressed yet (Section 2.7).

2.1 AUTHORING MULTIPLAYER GAMES

Authoring tools like *<e-Adventure>* [115], *SeGAE* [127] and *StoryTec* [74, 73] are well-known instruments to support video game development. The main approach of these tools is to facilitate collaboration between game developers and people from other disciplines (e.g. pedagogy) by guiding them into a common workflow. To assist non-programmers, they also provide predefined templates encapsulating complex gameplay functionality and which can be filled with arbitrary content. Lastly, they validate the user input in order to prevent errors, both simple (is a variable defined?) and complex (is the game solvable?).

Authoring tools are used in e-learning as well, *COLLAGE* [50] and *docendo* [51] for example provide configurable learning templates. The main difference between these two application domains is that learning templates are mostly static, while game templates are highly dynamic.

Aside from *COLLAGE*, which contains e-learning templates for the interaction between learners, these tools are used to create applications for individuals. Therefore, to the best of our knowledge, there are no game authoring tools available that explicitly address the challenges that arise when creating multiplayer games (Section 1.2).

Another option for creating multiplayer games is to use general purpose game engines [84], which are very flexible. For example, *Unity3D*¹ and *Unreal*² also support collaborative working processes and provide predefined functionality like physics calculations. But, in contrast to authoring tools, they require programming skills and do not offer design templates or advanced error checking. In regard to networked multiplayer games, for example, they provide basic technology like network interfaces, but the user still has to decide which information is sent over the network. Therefore, they require expert knowledge in order to be used.

An easier but less flexible alternative is to take an existing multiplayer game which has a publicly available game editor. This kind of editors is less complex than a game engine, as the user can only modify certain aspects of the game. Depending on the configuration provided by the game developer, this can include missions, locations,

¹ <http://www.unity3d.com>

² <http://www.unrealengine.com>

stories or characters. Since the underlying technology is already provided by the game, the users can focus on design and content when using such an editor. The drawback of using editors is that everything created is limited by the bounds of the game and the expressiveness of the editor. For example, the editor of *Portal 2*³ allows the creation of puzzles based on gravity and distances, but it is impossible to create tasks that include magnetism.

2.2 DESIGNING MULTIPLAYER GAMES

Most relevant literature on game design tackles the design and development of multiplayer games. Adams [2] for example describes two basic types of multiplayer modes: competitive (players have mutually exclusive goals) and cooperative (they have the same or at least related goals). Team-based modes represent a middle ground, in which players on the same team share a goal that is in conflict with the other teams' goals. While a single game can support multiple modes, each of them has to be designed separately and they can therefore vary in quality as perceived by the players. Adams also mentions different settings: playing on the same device, playing on different devices located in the same room and playing on different devices which are connected over a network. These settings can influence a game's design. For example, communication is trivial when playing on the same device but requires effort when playing in different rooms. Adams also urges developers to take social (misbehavior, cheating), technical (connection problems) and organizational aspects (players having to leave early) into account.

Contrary, Zagal et al. [130] do not view multiplayer as one of several design aspects. They argue that there are so many differences between single- and multiplayer games that they require separate design processes. To support these processes, they propose a multiplayer design methodology. This methodology focuses on the important characteristics of multiplayer games that influence the design of the game rules. Similar to Adams, one of these characteristics is the trade-off between competition and cooperation (in between: temporary alliances). The existence of meta games (getting information outside of the game to gain an advantage inside) is also unique to multiplayer games. According to Zagal et al., these aspects are influenced by the players themselves. Other aspects like whether the players can act synchronously or who coordinates the game are influenced by the tools, i.e. the soft- and hardware in digital games. Finally, social interactions between the players, which can be natural or stimulated by the game, are influenced by both the players and the tools that are available to them.

Harteveld et al. [48] also argue that single- and multiplayer games require different design approaches. They justify this claim by stating that both types of games differ significantly along four dimensions. For example, singleplayer games rely on pre-configured elements (data intensive) and strictly defined rules. Multiplayer games in contrast are process intensive (i.e. they can be greatly influenced by the players) and usually have unwritten, socially established rules as well. This means that singleplayer games can be better controlled by the developers, but multiplayer games offer the players more freedom. Giving the players freedom to choose between different possibilities implies that for each decision multiple outcomes must be defined. This

³ <http://www.thinkwithportals.com/>

requires more development effort, which – again – indicates that developing multiplayer games can be very complex. Since this also implies that both game types have distinct features as well as different strengths, they suggest to focus development resources on either type in order to create a good game. The decision on which of the types is chosen should be based on the goals that the developers have for the game. In order to support their claims, Hartevald et al. compared the properties of a single- and a multiplayer game which they have created, trying to point out these general differences in the concrete examples as well. They are able to do so for the first two dimensions (process vs. data intensive and formal vs. social rules), but they also have to admit that it was a difficult comparison because the games also differ in many other aspects unrelated to the distinction between single- and multiplayer.

If games are played on different devices over a network, additional technical and design challenges arise. Pinelle et al. [86] noted ten heuristics for this kind of games, which include having session management, matchmaking and communication tools. They also suggested reducing temporal dependencies between the players and trying to prevent waiting times caused by connection problems. However, there are also approaches that can keep temporal dependent games in a consistent state, even when limited delays between transmission arise. This is done by retroactively manipulating the execution order of events [71].

2.2.1 Collaborative and Cooperative Multiplayer Games

Developing for a scenario in which players work together instead of fighting each other greatly influences the game's design. Zagal et al. [132] investigate this by analyzing collaborative board games. They note that for creating an interesting collaborative game there must be some tension between collaboration and selfish play. Although the players ultimately share the same goal and always win or loose as a group, this tension can facilitate discussions about how to reach this goal. Their findings also lead to the formulation of several lessons and pitfalls, which are relevant for computer games as well:

- Introduce tension between perceived individual and team value of decisions
- Allow players to make decisions without the consent of the group
- Enable players to understand the results of their decisions
- Give players different roles and abilities
- Prevent dominant players from making decisions for the team,
for example by distributing information among all players
- Make players care about the outcome of the game
- Vary the game between play session,
for example by making it harder for experienced players

Zea et al. [134] transformed collaborative learning requirements into game design guidelines:

- Give players a common goal and shared rewards
- Require a minimal score of each player before the group can progress,
but also give the players enough information to enable helping
- Make players accountable for their actions, for example by showing their individual results to the group

- Guide group members towards social interactions, for example require consensus to foster discussions
- Establish a rotating leader role

During the development of the game “Jamestown”, the developers gathered some practical advice based on player observations [29]:

- Prevent waiting times
- Avoid differentiating statistics like individual scores (contradicts Zea et al.)
- Take into account that the players’ skill can vary and that negative contributions could result in blaming
- Make sure that teams only fail as a collective and that each player is able to contribute something tangible
- Facilitate interactions among the players

The developers of the game “Together” followed similar rules to establish a relationship between the players [30], including:

- Avoid levels that could be solved without all players contributing
- Add game mechanics that allow helping and coordination
- Have no abilities unique to each player so that each player knows exactly what the others can do (contradicts Zagal et al.)
- Let players choose their responsibilities at any given time, for example to help when a player has difficulties using a certain ability

Furthermore, Corrigan et al. [32] conducted a user study regarding a collaborative board game. Their main finding is that collaboration has to be required by the game, otherwise the players tended to play solitary.

These suggestions differ significantly and even contradict each other in some aspects. This highlights the fact that in game design there is no single right answer for most questions. Instead, decisions have to be made for each game individually and based on the intended target audience.

2.2.2 Game Mechanics for Collaborative Interactions

The interactions between players are not only a central aspect of multiplayer games in general, but also of collaborative games in particular. They usually take the form of game mechanics, which can be defined as player activities that result in some kind of response from the game [38]. Following this definition, game mechanics can be sorted into different categories depending on whether they are required to solve the game (core mechanics) or not (satellite mechanics). The non-essential satellite mechanics provide choices to the players and enhance, substitute or diminish core mechanics. A similar definition by Sicart [105] describes game mechanics as methods invoked by agents that interact with the game state. Agents cannot only be players, but also entities controlled by the game itself. Since other players are a part of the game state, his definition also covers interactions among the players.

Koster [58] listed a large number of game mechanics which are regularly found in multiplayer games. These include collaborative player interactions like helping, mentoring new players, trading items, voting and specializing for a certain role inside of a group. Manninen and Korva [69] described eight collaborative puzzles, which were based on the following interaction mechanics:

- Coordinating different perspectives and actions (under time constraints)
- Sharing information
- Negotiating multiple possible solutions and plans

During their evaluation they found that they had to guide their players towards collaboration, even though their mechanics already required them to work together. Manninen and Korva realized this by constraining the environment, which made the players stay together, and by introducing a threat of punishment, which forced the players to coordinate their actions. Kim [57] also gave some ideas on how multiplayer jigsaw puzzles could work in general. His suggestions use the cooperative mechanics of trading puzzle pieces, acting simultaneously at different puzzle parts and exchanging information.

In that context, it has to be noted that although multiple sources mention information exchange as a mechanic, communicating with another player does not directly change the game state. This means that it is not a mechanic itself when following the definitions provided at the start of this section. However, one can easily construct mechanics that require the players to exchange information in order to change the game state. For example, one player could be able to determine a monster's weakness that another player then has to exploit in combat. Therefore, exchanging information has to be seen as a special case in relation to the concept of game mechanics.

Parker [83] noted that team-based mechanics can result in frustration, if not implemented correctly. Arguing that even bad teammates must always be better than an incomplete team, he suggested that player interactions should always result in something positive. He also advised against total dependencies between the players and that it should be hard to criticize other players. Such criticism can be impeded by providing multiple solutions to each problem and by giving imperfect information about these options as well as the current game state to the players. This means that no option can objectively determined to be the best one.

2.3 GAME DESIGN PATTERNS

The idea of design patterns first appeared in architecture [3], where Alexander et al. collected recurring problem-solution pairs using a fixed structure. Their goal was to support the design and construction of buildings by preserving existing solutions. They also aimed at providing a common language for experts in the field and therefore called their results a pattern language. Gamma et al. [41] adapted this approach for software design, in which design patterns are meant to provide reusable building blocks for common problems.

Kreimeier [59] proposes to use the same approach for game design aspects. However, Björk et al. [19] argues that game design patterns, unlike architecture and software design patterns, should not be defined as strict problem-solution pairs. They justify this by pointing out that game design is a creative process with no objectively "right" solutions, as the intended outcome can often be achieved with multiple patterns. Additionally, each game design decision imposes different constraints on design aspects other than their intended outcome. Therefore, pattern definition by Björk et al. emphasizes the relationships between different patterns as an important aspect. The *Game Ontology Project* [131] is very similar to the game design pattern approach. It also identifies important game design elements and the hierarchical relationships

between them. The ontology's goal, however, is to describe, to analyze and to study existing games. Game design patterns, in contrast, also aim at supporting the design of new games. Because of this, the ontology format focuses on general descriptions and the relationships between elements, while game design patterns also contain practical advise on how to use them.

The approach by Björk et al. is criticized by McGee [72] for being too hard to use, especially by non-experts in game design. He argues that their descriptions lack prescriptive information about the context in which the patterns can be used. For that, McGee proposes an alternative design pattern definition which views patterns as trade-offs between conflicting forces. Contrarily, Olsson et al. [81] argue that design patterns should only describe abstract concepts in a neutral way. On a lower and more concrete level, contextualization takes actual design goals into account in order to provide general design directions. In their opinion, mechanics (Section 2.2.2) are even more specific, providing actionable advise for the developer.

Design patterns have further applications other than preserving recurring elements for other experts. Game design patterns are also algorithmically combined in order to procedurally generate game content [108, 33]. Patterns are used in e-learning as well, for example as templates that can be filled with content [50] or to model the process of creating e-learning material itself [135].

2.3.1 *Formalization*

A comparison between multiple software and game design pattern sources [41, 76, 20, 59, 19, 72] indicates that the underlying pattern description formats share many similarities (the complete comparison can be found in Appendix A, Table 21).

All of them split their description into several properties, most of which are textually described (one paragraph or more). There are, however, exceptions that are shorter (identifiers like name) or follow a different format (source code, diagrams). Every analyzed pattern language requires a name as a unique identifier for each pattern, with some of them allowing aliases as well. The problem each pattern addresses and its solution are very common elements as well. Only Björk et al. do not describe patterns as problem-solution-pairs and instead focus on common design elements and their effects. Another common property are the consequences of using the patterns (all but McGee). Finally, usage examples and the relationship to other patterns can be found in Björk et al. and all software design sources.

While the software design patterns provide practical implementation guidelines or example implementations, only the design patterns of Björk et al. feature something similar (although more abstract). They describe the decisions which follow when a pattern is used. This lack of implementation details can be explained by game design patterns being more about abstract design concepts. Therefore, there are many valid ways to integrate them into a game design, let alone implement them into the game's code. A related aspect is that software design patterns feature constraints, which have to be met for the pattern to be applicable. But there is nothing similar required for game design patterns. Due to their higher abstraction level and the overall creativity involved in game design, game design patterns can usually be adapted to fit a specific context.

However, formally describing individual patterns is only one part of a pattern collection. The potentially large number of patterns must also be organized in such a way that a user can find specific patterns as fast as possible. This is trivial when knowing its name, for example after having it suggested by another person, but harder when having only a problem description in mind. Gamma et al. [41] therefore provide a list with the pattern names and a very short description, which can be skimmed and searched. They also drew a diagram visualizing the relations between patterns, helping the users to find similar patterns. Björk et al. [18] grouped their patterns into hierarchical categories based on the topic they describe, which also allows users to find similar patterns. Furthermore, they employ a wiki, which allows them to provide navigational links between patterns at any point of their description. They also classified the relations between patterns as:

CAN INSTANTIATE / CAN BE INSTANTIATED BY denotes a more concrete implementation of the same concept (superior / sub-patterns)
 CAN MODULATE / CAN BE MODULATED BY when patterns influence each other in some way with no implications about abstraction level,
 POTENTIALLY CONFLICTING WITH when patterns cannot be used together
 POSSIBLE CLOSURE EFFECTS for timed relations

An example pattern including its relations is given in Table 1.

Name	Multiplayer Games
Description	Games that have more than one player. Most games let several players participate in the gameplay, either against each other or working together towards a common goal. [...]
Consequences	[...] The gameplay in Multiplayer Games naturally provide <i>Mutual Experiences</i> as long as the players have some direct interaction with each other. [...] It can be difficult to have <i>Extra Chances</i> in <i>Multiplayer Games</i> since letting one player undo an effect in a game easily disrupts the other players gameplay experience [...]
Using the Pattern	[...] A basic decision regarding Multiplayer Games is if the primary gameplay resolves around <i>PvE</i> (Players vs. Environments) as in Space Alert and Left 4 Dead series or <i>PvP</i> (Players vs. Players) as in Chess, Go, and Quake series. [...]
Examples	Chess has two players competing against one another by taking turns. [...] Computer and console games such as the Quake series or the Need for Speed series allow players to compete against each other in combat or races. [...]
Relations	Can Instantiate: [...] <i>Mutual Enemies</i> , <i>Mutual Experiences</i> [...] Can Be Modulated By: [...] <i>PvE</i> , <i>PvP</i> [...] Potentially Conflicting With: <i>Extra Chances</i> [...]

Table 1: Game design pattern “Multiplayer Games” by Björk et al. [18] (abbreviated).

2.3.2 *Pattern Generation*

There are multiple ways in which design patterns can be generated. Gamma et al. [41] envisioned experienced practitioners writing down a software design pattern each time they observe a recurring problem and its solutions. To this end Wellhausen and Fiesser [123] suggested a writing process, starting with a critical evaluation of the novelty and relevance of the pattern. Then, one should write the solution first and after that the problem, the consequences including drawbacks, the forces that make the problem difficult and the context (which is not modified by the solution). After that, the authors should give the pattern a short name that is easy to remember and refine it, especially comparing the consistency between each of these sections.

For game design patterns, Björk et al. [19] suggested three sources: Converting descriptions of game mechanics by abstraction and merging, analyzing existing games and interviewing experienced developers. Later, Lankoski and Björk [63] discussed generating patterns purely from theory as well. Using this approach there are no existing games using the patterns. Therefore, special care must be taken to ensure that the patterns result in interesting gameplay mechanics.

Once a pattern has been created, its quality needs to be assessed. The dominant approach for evaluating software design patterns are expert discussions, written or in workshops, with the goal of refining the pattern. This is often called “shepherding” [76].

Björk et al. [19] suggest workshop discussions for validating game design patterns as well, but are also in favor of testing their applicability by using them in practice (analysis and design). But when evaluating their practical applicability, there is a key difference between software and game design patterns. Software design patterns are primarily relevant for the developers, as they focus on the technical implementation details. Game design patterns in contrast have two target groups, with the purpose of helping developers to create an engaging experience for the players. This means that game design pattern evaluations need to be done in separate user studies for each group. One study would focus on developers working with the patterns [72], the other one would expose players to games implementing the patterns [97, 104].

2.3.3 *Patterns for Collaborative Multiplayer Games*

Due to the focus of this work it is important to analyze existing game design patterns which describe player interactions in collaborative multiplayer games.

Bergström et al. [14] analyzed existing collaborative games, finding several new game design patterns. Most importantly, the pattern “Team Strategy Identification” describes players interacting with each other in order to decide on a strategy. The other patterns describe more general concepts like accomplishing something together, but no concrete interactions. Additionally, they found new instances of patterns already existing in the database by Björk [18], which at the time of writing consists of 536 game design patterns describing a wide variety of topics. The most relevant ones for this thesis describe general collaborative multiplayer concepts:

- Altruistic Actions (benefit another player)
- Collaborative Actions (multiple players take part)

- Asynchronous Collaborative Actions (multiple players take part, but not at the same time)
- Team Combos (additional benefits when multiple players synchronize their actions).

Rocha et al. [97] informally described six more design patterns for cooperative games, providing only a name and a general description. Two of them describe concrete player interactions, namely making another players actions more effective and using abilities on another player. The other ones are more general, like giving players shared goals and different character abilities. They also collected how certain challenge archetypes are used in cooperative games, which could be translated into design patterns as well: Physical challenges (carrying a heavy object together or being at different locations at the same time), combining different character abilities for exploration challenges, sharing resources, as well as solving tougher challenges that could not be overcome by a single player.

Seif El-Nasr et al. [104] extended this work by finding additional patterns based on existing games, also using only the name and description. Among more general patterns, their interaction patterns describe manipulating the same object and exchanging resources. Then they evaluated several games using those patterns, collecting so-called “cooperative performance metrics” among the participating players. These include positive (expressing enjoyment, discussion strategies, helping each other) as well as negative (waiting or blocking each other) events. As a result they found that the patterns “complementarity”, “shared goals”, “shared puzzles” and “interaction with the same object” mostly triggered positive reactions. Camera issues on the other hand could lead to problems. Furthermore, players helping each other was more common in more difficult games.

Beznosyk et al. [16] took the same patterns and created a small game for each pattern. They classified each game as either closely- or loosely-coupled depending on how much direct influence players have on each other. Afterwards a user study was conducted, during which the players were not able to communicate in any form. It was found that there was more collaboration in the closely-coupled games, but also longer waiting times and the individual enjoyment depends on the other players’ actions. Specifically the patterns “complementary” and “interaction with the same object” showed the highest level of collaboration and were rated as most enjoyable. “Limited resources” in contrast resulted in the lowest level of collaboration and enjoyment, but was also less influenced by the lack of communication. Of the loosely-coupled patterns “abilities that can be used on other players” was enjoyed the most, despite the fact that it did not trigger much teamwork. “Shared goals” in contrast did not trigger any collaboration or enjoyment at all.

Additionally, Reichart and Bruegge [89] collected informal patterns for cooperation and competition through game analysis and literature research: “Complementary roles”, “scarce resources”, “halting points” (for letting slower players catch up) and “team formation”. Azadegan and Harteveld [8] suggested inverting existing collaborative engineering patterns in order to create challenging activities. As examples they provided information obfuscation by distributing it among the players, or forcefully separating players based on the divide and conquer approach.

2.4 FORMAL VERIFICATION OF GAMES

The current industry standard approach for detecting errors in games are best-effort user tests [101]. Doing so, however, is cumbersome, especially in multiplayer games, and happens relatively late during development when the game is already playable. Human testers are also unable to test every possible path through the game, which means that some errors remain undetected. [21]

Verification algorithms, in contrast, can be used to conclusively prove that a game has certain properties, for example that its ending can always be reached. This requires a formal model of the game, which translates all relevant aspects into a precise and machine readable format. Such models usually ignore properties which are not relevant for the task they are specified for, for example the graphical representation of the game.

For describing a game in a precise way, several formal methods have been proposed. Smith and Mateas [107] used first order logic, which consists of basic facts including contradictions, to describe the design spaces for a game. They could find solutions to this specification with answer set programming, which in this case resulted in the procedural generation of game levels. Marchiori et al. [70] proposed to model games as finite state machines consisting of states, inputs and outputs to describe the game state and how it can be changed. This model was then used as input for the authoring tool *<e-Adventure>* [115], which translates it into a playable game.

Other modeling approaches use the visual UML language to precisely communicate certain game aspects to other people working in the development team. Taylor et al. [113] used UML use case diagrams in order to describe the game flow between static and dynamic objects inside a level. Reyno and Cubel [96] proposed a more implementation-oriented combination of social context diagrams for the relationships between players, a structure diagram for game elements as classes and a rule set diagram mapping game events to effects. Pleuss [87] developed a similar model, describing the game objects and their relationships as classes, the game's scenes including menus as a state machine and the interactions in each scene with an extended UML activity diagram including an abstract representation of the user interface.

2.4.1 General verification approaches

Video games are software systems, so one approach for verifying them is to use general software verification tools like the ZING model checker [5]. The drawback of this method is that it works on the game's code and therefore can only be used at a late development state. General methods are also unable to exploit specific game properties to reduce verification complexity, for example by automatically ignoring unimportant details. Design level questions like "are players always able to reach an ending?" can also not be answered directly using such general software tools.

Haufe et al. [49] proposed a description language intended for finite n-player games. This language consists of logical clauses (a set of literals implying another literal) and additional functions to model randomness and visibility for each player. Interpreting the game as a state transition system, they used temporal logic and an-

swer set programming for verification using an existing solver. They also developed an extension for complex games in which the state space is too large for an exhaustive search. The approach was evaluated on a set of games like “Connect 4”, most of which could be verified in a few seconds. For a few configurations, however, the solver was stopped without a result. A verification approach for another game description language was defined by Ruan et al. [100]. The language they used consists of game states, players, possible player actions, a function for defining the successor states based on the current state and the action taken as well as an interpretation function that assigns properties to states. Based on the *Datalog* programming language, it can also be understood as a set of literals implying another literal. It is possible to verify game properties described with alternating-time temporal logic on this model, but this process is EXPTIME-complete.

The drawback of these approaches is that the game under test has to be specified in a formal language, which requires expert knowledge. Therefore, Osborn et al. [82] designed the game definition language *Gamelan* based on board game rules. They argued that such a language can easily be understood by game developers, as they often use this kind of rules for specifying prototypes. It allows the developer to answer questions like “is every available function useful” in non-deterministic games of incomplete information over discrete domains, e.g. card, puzzle or strategy games.

An even more accessible approach is to automate the model creation by specifying transformation rules that are integrated into the game development environment itself. This was done by Moreno-Ger et al. [77] for *<e-Adventure>*. The platform is able to translate games into transition systems with finite variables, which can be verified using the external model checking tool *NuSMV*[28]. This tool allows the verification of temporal properties, for example whether a game scene can be reached by the players. When a state violating such a property is found, the chain of actions leading to this state is automatically replayed to make the interpretation of the result easier.

2.4.2 Verification using Petri Nets

Petri nets are a well established model for describing concurrent systems. They are bipartite graphs, which means that there are two types of nodes that are used in an alternating manner. Because of that, two nodes of the same type cannot be connected directly. One of these types are places, usually described as circles (“A”, “B”, “C” and “D” in Figure 2). These places can hold any number of tokens (black dots) and the state of a petri net at any given time is defined by this token placement. Transitions (boxes) define the possible changes that can happen to this state and are connected to places via directed arcs (arrows). A transition is called “enabled” when a token can be assigned to each incoming arc, i.e. there is at least one token per arc in each associated place (Figure 2a). Enabled transitions can be fired, in which case the tokens assigned to the incoming arcs are consumed and a token for each outgoing arc is produced in the corresponding places (Figure 2b). If multiple transitions are enabled at the same time, the order in which they are fired is unspecified.

Jensen [53] defined an extension called “colored petri nets”, which adds additional elements to the model. Colors can be seen as variable types with which places and tokens are annotated (“Boolean” and “Objects” in Figure 2). Tokens can then only exist in places which share their color. The colors also define value ranges and to each

token of that color a value from this range is assigned. This value can be changed by a transition (from “true” to “false” in the example). Lastly, boolean guard expressions can be added to transitions (“ $x = \text{true}$ ”), which constrain the token values that enable the transition.

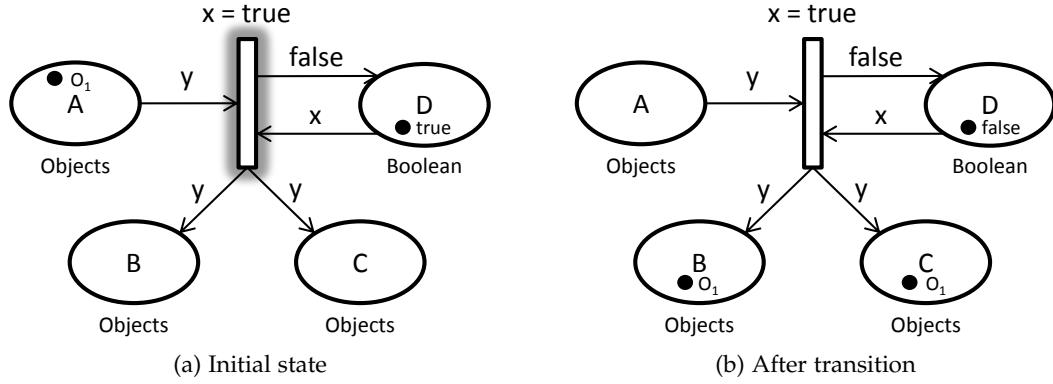


Figure 2: Colored petri net example.

There are tools available for verifying colored Petri nets [54]. However, doing so becomes problematic for bigger nets due to the size of their state space. This can only be mitigated in part by reducing the state space or by making state space exploration more efficient [26]. Only calculating which transitions are enabled at any given time is already NP-hard, although there are heuristics to accelerate that process [67].

Using Petri nets to model certain aspects of games has been proposed by several authors. Natkin and Vega [78] developed a Petri net model for task structures in a game. This model contains templates for individual tasks including pre- and post-conditions as well as the options to cancel a task or to limit its number of executions. Relations between the individual tasks, like the need to complete another task before (once or a certain number of times), and exclusions between tasks can also be described. Multiple relations can be linked using “and” as well as “or” operators, like “(A before C) or (B before C)”.

Using this model Natkin et al. [79] modeled the behavior and relationships of items the players can pick up and use in a game. They also modeled the game environment as a network of rooms using a hypergraph, which allows the assignment of items to rooms. With that they calculated a temporal order in which the items have to be acquired based on their logical dependencies, and analyzed the spatial topology of the rooms (reachability). Based on that, Natkin et al. could verify whether the game contained deadlocks (i.e. states from which players are unable to reach an ending).

Araújo and Roque [6] also argued that Petri nets are a useful tool for simulating the game’s behavior even before it is fully implemented. They mentioned multiplayer games as a use case in which this is especially valuable, since players often find ways to interact in unexpected ways. In their opinion, Petri nets are well suited for verification because they are mathematically well-founded, can be verified by existing tools and are less complicated in terms of visualization compared to UML. However, they also mentioned that Petri nets can grow fast in complexity, which can make their interpretation difficult. To mitigate this problem they suggest using colored Petri nets for complexer games, offloading some information onto the tokens themselves.

Carron et al. [21] also used colored petri nets, more specifically symmetric petri nets with bags, to model a game's abstract specification. With this model they were able to calculate the reachability of learning objectives and detect deadlocks in which the game stops unexpectedly. Invariant properties, which must always be true, and temporal ones concerning the order of certain events could also be verified. This includes game-independent (the game being solvable) and game-dependent (a certain element must always be used) properties.

Champagnat et al. [23] stated that the state space explosion of petri nets usually prohibits complete proofs. Instead they proposed checking games iteratively at runtime and, when a problem could arise during the next few steps, adapt the game in such a way that it is avoided. They also differentiated between two types of properties that should be verified: First, playability properties include making sure that the game is playable at all. This can imply that there are no deadlocks, that certain states can be reached (reachability) and whether events will (liveness) or will not (safety) occur at all or infinitely often (fairness). Second, relevance properties are supposed to assure a game is interesting. They can include not ending too quickly (complexity), every player having a fair chance at winning (impartiality) and having a direct competition between them (concurrence).

There are also approaches which extend petri nets with game-specific elements. Verbrugge [121] developed the model of "narrative flow graphs" based on petri nets, extending them with labeling functions for user input and feedback as well as winning and losing states. Pickett [85] translated this model into input for the NuSMV model checking tool, which allowed him to verify a game's temporal properties. These properties included the reachability of states, the distance between two states and pointlessness. This last property describes the number of steps which, after reaching a state where winning the game is no longer possible, a player has to take before he actually loses.

However, Dormans [36] argued that petri nets are still too difficult to read and proposed his own description language loosely based on petri nets. This language is based on resources being produced and consumed, although an example is given how they can also be used to model other aspects like the jumping height of a player character. It also uses the concept of tokens and places, but adds additional elements encapsulating complex functionality, for example actions that can only be executed if a player is skilled enough. The drawback of this language is that the new elements are not mathematically founded. Therefore, the language can only be used for discussions and manual analysis, but not for formal verification.

Aside from verification tasks, petri nets have also been used to generate a game's code [126] or as a player model [111, 128].

2.5 GAME BALANCING

The term "balance" is often used in the context of games, usually when criticizing that a game "is not balanced". People using this term usually refer to some kind of inequality between things that should be equal in their opinion, for example the abilities of different player characters. This is also tied to the concept of fairness, as unbalanced games feel unfair due to this inequality. However, there are multiple

levels on which the term is used and even within such a level there are multiple definitions of its meaning.

The first balancing level is the game design. Looking at multiplayer games, Sirlin [106] stated that a balanced design provides similar chances of winning to equally skilled players (balance between players). Additionally, during play there should be a large number of options the players can choose between in order to make the game more interesting and unpredictable. All of these options should be viable in at least some situations (balancing between individual options) and there should be no strategy that is always right or wrong (balance between overall strategies). To achieve a balanced game design he suggested running user tests, especially with expert players, but also following the developer's intuition.

Newheiser [80] noted that there are contradicting expectations when it comes to balancing a multiplayer game. Novice players feel like they should have a chance of winning, even when playing against experts. Loosing all the time feels unfair to them. Having random advantages and disadvantages will sometimes tip the odds in favor of weaker players and can therefore improve their chances. In contrast, experienced players want to be rewarded for their effort in improving themselves and describe it as unfair if their experience does not count. Since both views are understandable Newheiser suggested to mix skill- and chance-based elements, with the weight between them depending on which audience the game is for. But his general understanding of balancing is not only focused on the concept of winning. He also stated that the means of play should be roughly the same for every player and that there should always be counters for any given strategy. This means that differences between the players (for example different player characters) do not automatically break the game balance as long as they are equally useful.

Berry [15] also proposed using random elements in order to ensure short distances between winners and losers, for example in terms of the score. This leads to the outcome of the game being unclear for most of the time, resulting in a more interesting experience. Similarly, Adams [1] stated that the leading player should change regularly during gameplay. In his opinion, however, one should assure that the better player always wins.

Balancing is also important for singleplayer games. Here, Schreiber [103] stated that the general challenge level of the game should be appropriate for the intended target audience (balance between player skill and challenge). Like in multiplayer games, he also suggested having multiple balanced options and strategies in a singleplayer setting. Aside from user tests and instinct, he proposed using math to achieve this, for example by having all purchasable items share the same cost to benefit ratio. According to Schreiber, however, it is also good to have a few incomparable elements and that imbalance can be fun, too.

The common aspect of these views on balancing is that they are competitive, concentrating on the player's struggle against the game or other players. But it is obvious that in collaborative games there is also a relationship between the players who are working together, which must be balanced. After all, a game for two people in which one is doing all the work with the other one passively watching would not be considered fair or interesting by most players. In collaborative learning there is a related problem called "free-riding" [110], which describes participants trying to profit from the work of others. But, to the best of our knowledge, there is only one work by Goh et al. [43] that addresses the contribution balance between team members in games.

Looking at a scenario with differently skilled players they investigated design elements that cater to the needs of stronger and weaker players simultaneously. This included giving a small reward even if there is no collaboration to mitigate frustration caused by other players' mistakes, creating incentives for the stronger players to help, removing or weakening elements like individual scores that could be used competitively as well as giving an advantage to weaker players.

It is also important to note that there are other sources of unfairness in multiplayer games, which are usually not tied to the concept of balancing: Using an internet connection with high delays can be a disadvantage in most games [133], even though the issue of connection quality in general is well researched [109]. Besides, there is also the issue of other players using illegitimate methods to gain an advantage ("cheating") [129, 64].

2.5.1 *Balancing Calculation and Adaptation*

Aside from these informal definitions and general guidelines, there are also formal models and algorithms for balancing. Leigh et al. [66] used a coevolutionary algorithm that is able to generate and test strategies as well as counter strategies. If the algorithm found a dominating strategy for which no counter exists, they changed the rules and restarted the algorithm. However, in most games testing the complete strategy spaces is impossible due to their complexity. Chen et al. [24] even argued that finding an optimal solution for balancing design parameters is NP-hard.

Additionally, games are played by a variety of different players possessing different skill levels. Hence, even if a game was perfectly balanced during the design phase, it can feel unbalanced once the individual player's skills are considered. A singleplayer game for example can be too hard for one player and too easy for another. Runtime adaptation can help in such cases by tweaking the balancing, either by making the game easier or by introducing additional challenges. Andrade et al. [4] realized that using a reinforcement learning approach.

In multiplayer games, adaptation can also help weaker players against stronger opponents. Examples for such approaches include changing the performance of cars in racing games [22] or correcting the player's input in shooter games [11]. Whether or not to expose the existence of such manipulations to the players has different effects. Adjusting the game in secret can cause good players to feel less competent, while openness can cause the weaker ones to feel dependent [9]. In another study, players who noticed that the game was "lying" to them even showed decreased interest in the game [119].

It is obvious that players and developers believing in skill as the only factor deciding who is winning, oppose such approaches and view them as unfair advantages. Therefore, it could be better to match players against opponents with similar skill levels [34], sidestepping the problem of uneven matches without manipulation.

For collaborative games, Wendel [124] developed an automated adaptation approach that can recognize game situations at runtime and select an appropriate balancing reaction from a pool. The resulting framework is very flexible, allowing developers to define arbitrary situations to be detected, adaptations to be triggered and actions to be selected. Therefore, that approach cannot only be used to optimize the game's difficulty level, but also to improve teamwork.

2.6 RAPID PROTOTYPING

Even when the verification has shown that there are no critical errors and that the game is balanced, some questions remain. This includes highly subjective questions like whether the game's graphics look good. Other questions depend on the target audience of the game, for example whether the players can understand what they are supposed to do. Since these questions cannot be answered algorithmically, it remains important to continuously test the game's playability and player experience during development [44]. Testing early and often is also called rapid prototyping, which involves testing changes quickly and without fully implementing other aspects. In the best case, this is directly supported by the development tools [75].

For multiplayer games that are played by a large number of players simultaneously, internal tests are insufficient. In such cases, developers often host public beta tests [99] to increase their chances for finding rare problems. These large-scale tests, however, result in a great amount of organizational overhead and consequently are unsuited for rapid prototyping. Especially single developers working on a multi-player game might not have enough players at hand for testing their game, both from internal and external sources. To remove the need for additional testers the *Portal 2*-Editor⁴ supports asynchronously switching control between two player characters. The screen is separated into two parts (Figure 3). Thus it is always visible what each player would see. By pressing a key, the control is transferred to the other character, which allows one developer to act as two players. The drawback of this approach is that situations in which both players act simultaneously cannot happen and that sounds cannot be attributed to one of the two players.

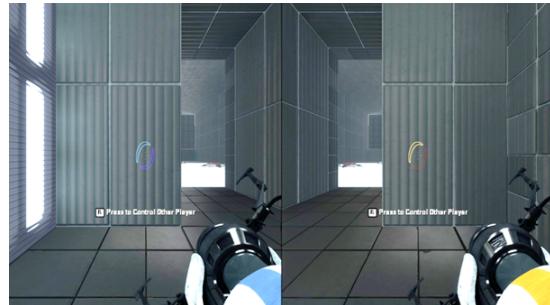


Figure 3: Screen layout when testing a level for two players in the *Portal 2*-Editor.

Recording user input is a common task when studying human-computer interaction [60]. In a multiplayer setting, such techniques could be used to pre-record the input for one or more players independently. Afterwards, the developer can control one player in real-time while the others are controlled based on the recorded data. Furthermore, the additional players could be simulated by an artificial intelligence, which has already been done while testing a singleplayer game [61].

2.7 IDENTIFIED GAP

Reviewing the related work, it becomes clear that there are still additional challenges that developers need to face when creating collaborative multiplayer games. On one

⁴ <http://www.thinkwithportals.com/>

hand, technical foundations like the support for establishing network connections are widely available and directly integrated into game engines or middlewares. But on the other hand, these tools are not supporting game creation on a content level, which does not prevent developers from building boring or even unplayable games.

The concept of authoring tools (Section 2.1) could aid developers in that regard. But this potential is not yet realized by the existing tools, which do not address the challenges that arise when creating collaborative multiplayer games. Most of them even lack multiplayer functionality completely.

There are several development phases during which specific support is needed. First of all the game must be designed. Here, the player interactions, which are a central aspect of multiplayer games, must be defined among other design aspects like graphic style or controls. But finding engaging (collaborative) interactions is hard and all existing design guidelines are relatively abstract (Section 2.2). Having concrete examples for well received player interactions, cataloged and grouped based on their properties, would be very helpful for developers. If these examples are transformed into predefined building blocks that encapsulate common functionality and can be immediately (re-)used in a game, they can also help experts to save time and reduce costs.

A good method for formalizing such examples is the format of game design patterns (Section 2.3). But although this concept is very popular and there are pattern collections for many types of games, there are only few patterns focusing on collaborative games. Furthermore, most of these patterns are located on a much higher abstraction level than concrete player interactions. This implies that although expert developers could transform them into such interactions, non-experts might not be able to use them. Some patterns are also not formalized, giving only general descriptions without specifying when they can be used or which side effects they might have. This also makes it more difficult to use them.

Once a game is created, it is usually played by dedicated testers first, whose task is to ensure that there are no critical problems left. A typical problem would be the game reaching a state in which it cannot be finished anymore, for example after the players destroyed a crucial item by accident. But with more players the complexity of testing grows exponentially, as multiple players can interact with objects or other players at the same time. Therefore, in most cases human testers are unable to test every possible sequence and combination of these events. Testing is also more costly for multiplayer games, as multiple players must be present.

Automated verification (Section 2.4) can take over the aspects of testing that can be formalized, for example ensuring the reachability of an ending. But existing verification approaches only operate on high-level models of the game, for example the dependencies between missions. With such abstract models not every aspect can be verified, for instance whether all individual mission are solvable. Another common drawback of the approaches found in the related work is that almost every model must be created manually. This is not only time-consuming, but also requires knowledge in formal modeling, which is normally unrelated to game development. Besides, there is also the potential of the model not exactly matching the actual game, either by human error or because it was not updated to match a change made to the game. All in all there is only one approach that generates the verification model automatically, but only for singleplayer games.

But even when a game does not contain critical errors, it can still feel broken if it is not balanced (Section 2.5). Existing balancing definitions, however, are based on a competitive viewpoint and do not take the collaboration between team members into account. Therefore, a game in which one player does all the work would still be balanced according to most definitions as long as he is able to do so. Again, only one example exists which takes collaboration into account when providing abstract balancing guidelines. In order to objectively verify that a game is balanced, a clear definition and concrete measurements for collaborative balancing are required.

Still, some aspects can only be tested and not verified, as they involve the subjective game experience of players. Here, the concept of rapid prototyping (Section 2.6) is a promising approach. But using the current state of the art, multiple players must be present to test a multiplayer game. This means that rapid prototyping becomes very cumbersome, if not impossible, when not enough players are present.

All of these aspects constitute gaps in the related work that should be addressed when supporting the development of collaborative multiplayer games.

OVERALL CONCEPT AND APPROACH

THE gap identified in the related work (Section 2.7) makes it clear that a new concept for supporting the development of collaborative multiplayer games is required. We therefore propose such a concept, which supports the creation of interesting player interactions, the detection of critical design errors, the balancing between team members and rapid prototyping.

3.1 APPLICATION SCENARIO

This work focuses on collaborative multiplayer games, in particular on the collaborative interactions between players, as the most central aspect. In general, there are two basic types of player interactions available in games, namely synchronous and asynchronous play. Whereas in synchronous games the players are playing at the same time and directly interact with each other, in asynchronous games a session of a player can impact upon a later session of another player. For this work we assume a synchronous setting, as supporting continuous direct interactions is more challenging from a research perspective. Building the concept based on collaborative interactions means that it is also applicable for most cooperative games, especially for sections or aspects in which the players' goals are not conflicting. Aside from including collaboration, there are no further assumptions made about the game in regards to its content, gameplay mechanics or genre.

The target audience that is to be supported are small game development studios or single hobbyists, which are inexperienced at making collaborative multiplayer games. Those developers usually start with smaller games in regards to playtime (a few hours), group size (up to four players) and game world (a limited number of discrete locations). This defines the scope of this work, although some of the concepts can also work for certain parts of larger games. An example for this are massive multiplayer online games, which support hundreds of concurrent players on the same server. Direct collaboration in such games, however, only takes place in substantially smaller groups – which makes these interactions fit the application scenario. Additionally, singleplayer games can be seen as a special case of a multiplayer game with a player count of one. Hence, in theory, the approach can also be applied to singleplayer games, although most of its benefits are lost in this case.

From a process-oriented perspective, the approach is concerned with games that are currently in development. This means that the actual players are not yet known and are only viewed as abstract entities. Differences in individual player's skills, for example, can therefore not be covered. Real-time requirements are also more relaxed during development. Although it is desirable to get results as quickly as possible, running an analysis overnight and incorporating the results on the next day is also a viable option at this stage.

3.2 RESEARCH QUESTIONS

Combining the most important challenges (Section 1.2) with the gap identified in the related work (Section 2.7) yields four aspects in which authoring support is needed:

- Inexperienced developers do not know where to start when creating a collaborative multiplayer game, as only general advice exists. Concrete guidelines or building blocks for collaborative player interaction as the central aspect of such games would provide a good foundation for the development.
- Games consist of a large number of game elements with which the players interact until an ending has been reached. Critical errors that prevent players from completing a game must therefore be rooted in the game's structure – for example when individual elements are missing or when connections between them are wrongly configured. Detecting such errors with traditional playtests (by human players) can be hard, especially if they only arise when multiple players interact in unexpected ways. Formal verification could solve this issue, but the current approaches require a great deal of additional effort and expert knowledge. Therefore, fully automated tests that detect such errors would be beneficial.
- Players could become displeased if they are not involved in the same way as their teammates, for example when they are doing all the work for the group. However, there are only concepts available for balancing games from a competitive point of view, i.e. the difficulty for the whole group. An objective measure for collaborative balancing, especially when it can be acquired automatically, is therefore necessary.
- Testing aspects such as aesthetics, which are not automatically verifiable, requires a group of testers and in some cases also setting up a network connection. This overhead makes rapid prototyping quite hard, especially when a single developer is working on the game. In this case, automating recurring tasks and restructuring information targeted at multiple players could be highly valuable.

The overall goal of this thesis is to demonstrate the feasibility of supporting the development of collaborative multiplayer games by addressing these issues. This can be broken down into the following more focused research questions:

- RQ 1: How to aid the creation of collaborative multiplayer games using player interactions as building blocks?
- RQ 2: How to detect structural problems caused by unexpected interactions?
- RQ 3: How to measure game balance in a collaborative game, specifically considering inter-player differences?
- RQ 4: How to support single developers when testing a game that requires multiple players to interact with each other?

3.3 OVERALL CONCEPT FOR AUTHORIZING COLLABORATIVE MULTIPLAYER GAMES

The overall concept consists of three steps that specifically support the challenges which arise during the development of multiplayer games. These steps constitute a process that can be integrated into an authoring tool and are designed to be used sequentially in form of a pipeline (Figure 4). Game development, however, is usually

iterative, so in reality each step, as well as the overall pipeline is repeated multiple times. It is also possible to skip any of these steps if their support is not required. For example, a developer could decide to ignore the prototyping environment if lots of testers are available.

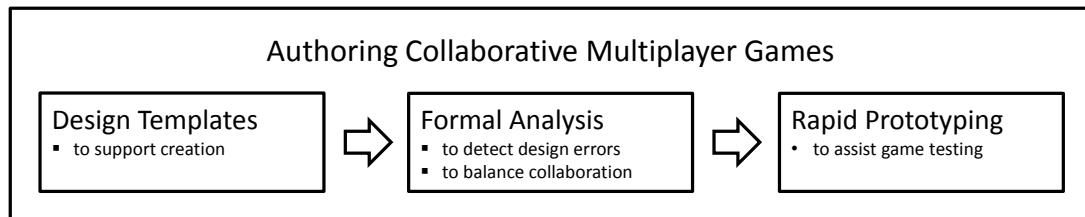


Figure 4: Concept for authoring collaborative multiplayer games that consists of three sequential steps and four modules.

The first step is based on a collection of design templates focusing on player interactions that have been extracted from well-received games. These templates can be used directly as an inspiration, or they can be implemented as ready-to-use templates. In the form of templates, they can reduce the effort required to build the game, even for experts. Since only a finite number of patterns can be defined using this approach, their definitions must allow variations. That way patterns can then be implemented differently, so that the interaction does not become monotonous when repeatedly used. This is also a requirement for being adaptable to different game types and settings. With this step, the first research question (**RQ 1**) can be answered.

As soon as these building blocks have been connected, the resulting structure can be analyzed through the next step. This analysis is broken down into two modules. On the one hand, the structure itself is verified against strict rules. These include that there are no interactions after which the game cannot be finished anymore (e.g. deadlocks). On the other hand, the involvement of different players of the group must be balanced. To achieve this, a definition of collaborative balancing has to be developed first, as previous approaches consider balancing only from a competitive perspective. By using this definition, metrics can be developed, which allows the balancing of a given game to be measured. In contrast to structural problems, the balancing only has to “feel” right and having limited imbalances might even be a conscious design decision. Therefore, the balancing measurements are only displayed as an information, with the developer deciding on how to react to the findings. Both analyses are completely automated, requiring no additional effort or expert knowledge in order to be usable. This also means that there is no room for errors when building the game models to be analyzed. Additionally, they do not require actual players to play the game and can be used at a very early stage during the development process. For example, a game lacking graphics completely would not be playable at all, but an automated verification could already be conducted. The two modules used in this step can provide the answers to the second and third research question (**RQ 2** and **RQ 3**).

Some aspects cannot be verified, especially aspects related to the “feel” of the game – this includes the graphical style or the mood conveyed by a background music track. The third module is therefore meant to make rapid user tests possible, even when there are no additional players available. It does so by providing the developer with

the information available to each player as well as the ability to control all of them at the same time. This step can answer the last research question (**RQ 4**).

After an explanation of the technical foundations, the following four chapters each describe one of these modules in detail.

TECHNICAL FOUNDATIONS

THE authoring tool *StoryTec* [74] and its *StoryPlay* runtime environment [75], developed by the Serious Games group at the Multimedia Communications Lab of TU Darmstadt, are used as a foundation for this work. Supporting the development of singleplayer adventure games, the authoring tool already allows the creation of game worlds consisting of an arbitrary number of discrete locations. As the main gameplay mechanic, aside from navigating the world, it provides players with the ability to manipulate objects and solve complex puzzles. The tool has been the main contribution of Florian Mehm's PhD thesis [73] and has since been used successfully in several research projects, including inDAgo [55] and NeuroCare [47, 37].

By using this tool, functionality needed in both single- and multiplayer games – like adding rooms and objects to the game world – can be re-used. Since there are no comparable tools available, that are both actively developed and for which the source code could be accessed, the only alternative would have been to build this functionality from scratch. But because the tool was originally developed for singleplayer games, basic multiplayer functionality [91, 90] had to be added before the concepts described in this thesis can be implemented. This extension includes the underlying concept as well as the practical implementation, concerning both the authoring tool and the runtime environment.

4.1 UNDERLYING GAME MODEL

The basic game model on which the tool is based consists of three parts: The static structure, the current state and all actions that are able to change this state.

The general structure of each game is defined by a number of discrete locations called *scenes*. Depending on the developer's intention these locations can represent literal story scenes, spatial rooms which compose the game world or a combination of both. How the players can move between these locations is defined by the developer via *transitions*. These transitions allow non-linear game structures in which the players can chose the way they want to proceed. Inside these locations there can be an arbitrary number of objects for the players to interact with. These objects can range from simple text labels to complex multimedia objects like videos. Each object type possesses a number of different properties that can be changed, for example the content of a text label.

Obviously the current location of the player, or all players when using the multiplayer extension, is an important part of the current game state. The state of each object's properties, for example if it is enabled, is another part of the overall game state. Additionally, many games use helper variables to track higher order information, for example counting how many tasks have been solved. These variables have a type, for example integer, which defines its the value range.

This game state can be changed by events, or *ActionSets* as they are named in the tool. Most events are triggered by a player's action, for example by clicking on an enabled game object. Additionally timers are supported, which allow the game to change its state even if the players are not acting themselves. Each event can contain any number of game reactions, including variable changes or transferring the player to another location. There are also reactions which do not impact the game state directly, for example playing a sound effect. To make reactions more flexible the game model allows the organization of events as branching trees. The execution of such complex event trees will be assumed to be atomic, which means that events triggered by multiple players simultaneously cannot interrupt each other. In such a case the events are executed one after the other, even if each of them consists of multiple game reactions.

At each branch a condition must be defined, which is used to decide on the branch that is to be executed during runtime. This way the game can react differently to the same player action, for example depending on the current value of a helper variable. The most simple conditions are a comparison between a variable and a constant (e.g. *sounds = true* or *player <> 1*). However, more complex boolean expressions are also possible, using the *and* and *or* operators as well as brackets.

For multiplayer it was also necessary to allow different reactions depending on the player that triggered the action (Figure 5). The developer can use this to model player roles, which means that different players are able to execute different actions. Such roles can be fixed when chosen at the start of the game or flexible to allow the ad-hoc distribution of responsibilities. Making an action available for one player only creates clear responsibilities and makes sure that this player is involved. But it also limits the players' freedom, as the group cannot decide who has to take this action.

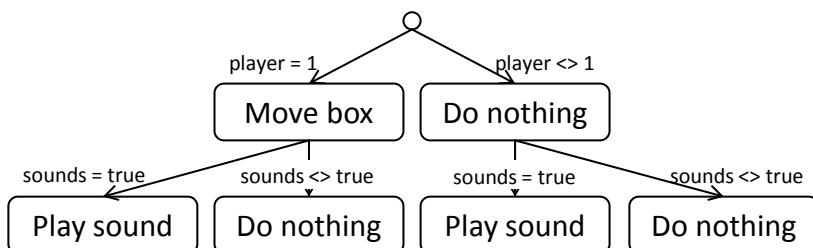


Figure 5: Example event with exclusive conditions. Only the first player (*player = 1*) is strong enough to move the box. If the sound is enabled (*sounds = true*), a scraping sound is played regardless of which player triggers the event.

Numerous convenience features exist as shortcuts for the developer, which do not increase the expressiveness of the game model. Locations can be organized hierarchically, with the child locations inheriting the objects and events placed in their parents. This way objects can be re-used in several locations, for example a sky being defined once for multiple outside locations (Figure 6). Events can be re-used in a similar fashion by chaining them, for example tying a follow-up event to the result of a player being moved to a certain location. Doing so is especially useful when there are multiple ways to trigger such a transition, for example when a room has multiple entry points. Without chaining, the task of playing a sound effect on entering would have to be added to each entry point separately.

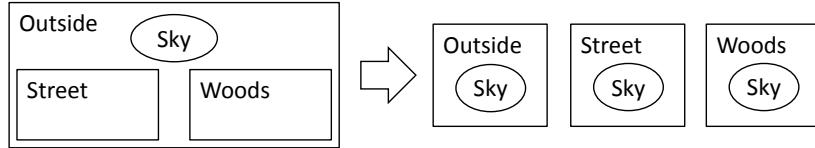


Figure 6: Location hierarchy example. Both models describe the same game, but in the left one only one “Sky” object is needed for three locations.

There is a shortcut for decreasing the complexity of events as well. Conditions in hierarchical trees do not have to be exclusive, which means that the conditions of multiple paths can be true at the same time (or none at all). The reactions on these paths are then executed in a depth-first manner. This allows the developer to compress events with several independent conditions. As an example, Figure 7 describes the same event as Figure 5 using non-exclusive conditions. The drawback of non-exclusivity is, however, that it is less intuitive for most developers and more error-prone.

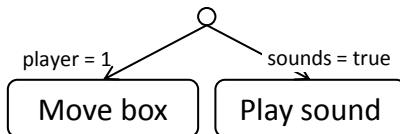


Figure 7: The same example event as in Figure 5, but with non-exclusive conditions.

By allowing arbitrary variable changes, this model is flexible enough to model the internal logic of any game found in the application scenario (Section 3.1) – even though the authoring tool implementation itself is focused on adventure games.

4.2 MULTIPLAYER AUTHORIZING

Some additional features had to be added to the existing authoring tool in order to support multiplayer games. First of all when creating a game (or *story* in the context of the tool) the number of players has to be specified. If this number is higher than one, the game is built for multiplayer and a few additional functions are unlocked. The first one is the object type *PlayerCharacter*, of which there is exactly one for each player of the game. This type is an extension of the existing *Character* type that makes each object instance identifiable by adding a unique ID property. The ID property then can be checked in conditions using the *TriggeredByPlayerID* variable. That way the developer can specify different reactions for each player using the item in order to model roles and character abilities. An example can be seen in Figure 5 in which the box only moves if the first player interacts with it (*TriggeredByPlayerID* has been shortened to *player*). The *IsPlayerIDAtLocation* is used in a similar way, allowing the developer to specify that another player must be present at the same location in order to support an interaction. It is essentially a shortcut for manually setting a helper variable when the other player is entering or leaving the location.

In addition to that an audibility for media elements like sounds, videos and text-to-speech acts can be defined. For example, this property governs which players can hear a sound that one of them has triggered to play. It supports internal thoughts (*triggering player only*), spatial limited noises (*all players at the same location*) and very

loud sounds (*every player* regardless of his or her location). As audibility is associated with the act of playing the object and not encoded in the object's specification itself, the same media file can be re-used with different audibilities. The same scoping model is used for the visibility of pop-up messages and for updating the internal player and learner model.

The modified authoring environment was used by the developer to create the game which is described in Section 4.4. A preliminary evaluation also showed that the tool could be used by other experts as well [91, 90]. However, later (informal) tests with inexperienced developers showed there is still room for improvements, which influenced the research questions tackled in this work.

4.3 MULTIPLAYER RUNTIME ENVIRONMENT

The *StoryPlay* player [75], which loads the game definition created with the authoring environment and controls the runtime behavior of the game, was also extended (Figure 8). First of all, it needed basic networking functionality in order to synchronize multiple game instances running on different computers. Compared to an approach in which the players need to share the same device, this approach allows the players to play together over a network while being physically separated. It also gives the developer the ability to display different information to each player. If a game for multiple players is loaded the user can select whether to start a new server or to connect to an existing one via the IP address, which initiates a direct TCP connection. Some stories include large multimedia asset files, for example videos. These would have to be initially transmitted over the network if a client connects to a server with an unknown game file. Therefore it was decided to load the assets locally from each machine. The drawback of this method is that the same files must be present on both machines, which is verified by comparing their hash values.

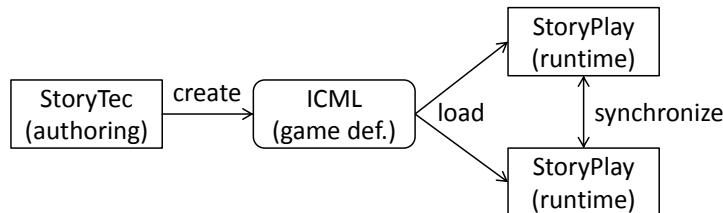


Figure 8: Interactions between the authoring and runtime environment for a two-player game.

During gameplay a client-server architecture is used, which means that the clients send their user input to the server (Figure 9). Hereby, the player who hosts the server runs both a server and a client process, which initiates a local connection. The server checks whether this input is triggering an event and distributes the resulting event IDs to each client. Realizing the input-event mapping on the server side prevents manipulated clients from triggering events that are actually inaccessible for them at that time. The clients then use the IDs to execute the associated events. Sending only the IDs also reduces network load by a huge factor compared to sending each game reaction associated with the events. Since the clients are working with the same game specification and receive events in the same order, consistency is ensured nonetheless. This approach could however be extended by not only sending the events, but also

the complete input. For example, this would allow the clients to display the mouse movement of the other players. While this would increase the network traffic, it would visualize the other players' actions in more detail. Timed events are triggered only by the server, which distributes the associated event IDs together with the ones triggered by the players.

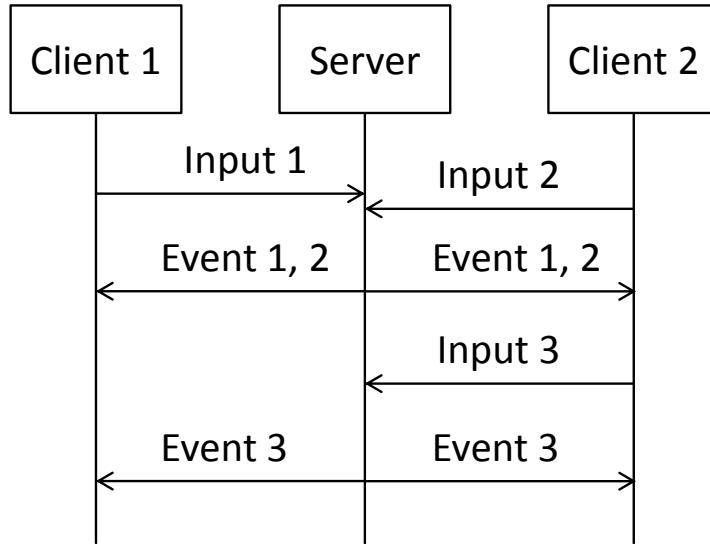


Figure 9: Multiplayer communication in the runtime environment.

The drawback of this approach is that the players cannot immediately see the results of their actions. Instead, the effects are only visible after the server has received the input and has sent back the calculated results. This problem could be solved by using client prediction. In this case, preliminary results of the player actions are shown immediately as predicted by the client. If this prediction is wrong, for example due to another player interfering, the preliminary result is rolled back after the actual effects have been received from the server. It was, however, decided to not implement such predictions, as the input delay is not problematic for the use case of adventure games. These games are relatively slow-paced and favor thoughtful actions over fast reactions, so a slight delay does not decrease playability.

The game state had to be adapted for multiple players as well. Some elements like variable values, the properties of the game objects and the overarching story model can be shared between players. Other data has to be duplicated for each player, for example his or her current location. Sharing this as well would prevent players from moving into different directions. The same is true for the state of the individual player and learner models.

As already mentioned, the *ActionSet* events allow the definition of player actions and game reactions. The scope of these events and their changes is normally defined by the location the event is attached to. With multiple players this gets more complicated. Especially when the game's locations are structured hierarchically, one player's action could change an object that is also available to another player in a different scope. An example for this is given in Figure 10: Both players P1 and P2 are in different child locations of the same parent, in which a sound object is located. If an event triggers a property change for this shared sound object, the effect is relevant for both players. However, if the sound is played for *players at the same location*, the

scope of the *ActionSet* must be considered. If this action is triggered by the *ActionSet* "A1", which has the location "Child 1" as a scope, the sound is only played for "P1". If the same action is triggered by "A2", the scope is the parent location and therefore the sound is heard by both players.

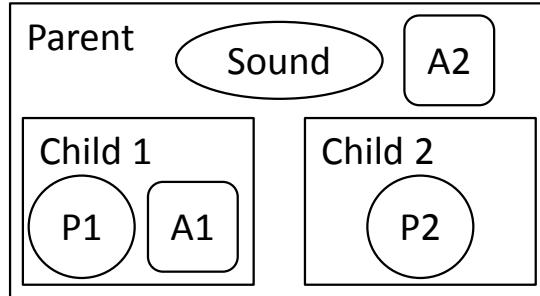


Figure 10: Example scenario for explaining the event scope in a multiplayer game.

To enable communication between the players, a text chat with which the players can exchange free-text messages at any time, has also been implemented. Additionally a simple avatar system, which uses icons to display which players are at the same location, has been added.

4.4 MULTIPLAYER ADVENTURE GAME

As a test case, a collaborative adventure game for two players has been developed using the enhanced version of the authoring tool. In the game the players have to repair a damaged research station, which is located deep in the jungle. To do so they have to solve several independent puzzles, which include restoring the energy supply and removing fallen trees.

The game has been designed with the following requirements in mind, extracted from the related work (Section 2.2):

- having puzzles that require communication and coordination
- requiring all players to contribute equally to the solutions (fairness)
- minimizing the time periods during which one player has to wait for the other
- providing comprehensible and realistic reasons for collaboration.

The game has been evaluated [91, 90] in combination with the runtime environment described in Section 4.3. This has proven that the extended authoring environment can be used for creating multiplayer games, which qualifies it as a valid basis for this thesis. Additionally, the game can now serve as the first test case for evaluating the concepts described in this thesis (Chapter 10).

GAME DESIGN PATTERNS FOR PLAYER INTERACTIONS

PLAYER interactions are one of the central elements in a collaborative game. It is therefore beneficial to provide a library of best-practice interactions. Here, our approach is to extract well-received interaction types from existing games, classify them and preserve them in the form of game design patterns [93].

5.1 COLLABORATIVE PLAYER INTERACTIONS

The first step in collecting instances of collaborative player interactions is to define which game mechanics are covered by the term. In order to observe instances of collaborative player interactions in existing games, there must be a precise definition of what to look for first. As a basis, the individual terms “collaboration” and “player interactions” have already been defined by related work. Collaborative gameplay is defined by Zagal et al. [132] as follows:

“In a collaborative game, all the participants work together as a team, sharing the pay-offs and outcomes; if the team wins or loses, everyone wins or loses.”

This means that in collaborative games the result of each player action affects everyone in the same way due to their goals being identical. Another definition for collaboration in the context of learning is given by Roschelle and Teasley [98]:

“Collaboration is a coordinated, synchronous activity that is the result of a continued attempt to construct and maintain a shared conception of a problem.”

Here, collaborative actions are also directed towards the same goal, in this case the understanding of a shared problem. However, they also state that these actions have to be coordinated and synchronous. Additionally, the term “attempt” implies that actions can be collaborative even if they fail to contribute to the common goal, as long as they are intended to do so. Player interactions are in turn defined by Manninen [68]:

“Interaction forms are perceivable actions that act as manifestations of the user-user and user-environment interactions. They enable awareness of actions by offering mutually perceivable visualizations [...].”

According to this definition, every action visible to another player is an interaction between them. In contrast to the other sources, this includes purely social interactions which are not directed towards an overarching goal.

We combine these definitions in order to define collaborative player interactions¹:

“Collaborative player interactions are a chain of coordinated actions by multiple players, which are intended to benefit their shared goals. Each individual action may be directed upon another player or the game world and the number of actions might vary between the participating players.”

This definition yields several important implications. First of all, players have to be aware of each other’s actions in order to coordinate them. This can be achieved by the players communicating them explicitly or by the game showing them implicitly. The need for coordination transforms these isolated actions into actual interactions. Collaborative player interactions can be modulated by varying the amount of coordination required to execute them successfully. Requiring more coordination increases the dependency between the players as well as the overall difficulty of the task, but also forces them to work together more closely.

However, interactions unrelated to the shared goal are not seen as collaborative. This distinction is not always easy, for example when players are talking to each other. Here, it depends on the content of the conversation whether the interaction is collaborative or not. Whereas exchanging critical information about the game is collaborative, talking about events outside of the game is not.

The aspect of collaborative interactions potentially failing to contribute to the common goal is integrated as well. Most games employ a certain difficulty in order to be challenging and engaging. This implies a possibility for failure, for example when there are enemies that can interrupt player interactions.

A single interaction can consist of several smaller actions, like walking or picking something up. Most often these individual actions alone do not constitute a collaborative player interaction. But in combination they can be a collaborative interaction, for example when another player is grabbed up and moved out of harm’s way. Keeping everyone alive will help the group to reach their overall goal.

There are not only interactions in which the players interact with each other directly, but also others in which they interact indirectly through manipulation of the game world. For example, holding a trapdoor open for another player requires coordination and may serve the common goal, even though the players never touch each other.

Furthermore, the individual actions of each player are allowed to vary in difficulty, effort and kind. A common example in action games is one player healing another, which usually requires the players to be close to each other. In this case, the player getting healed only has to stand still while the other one has a more active part. These differences can be based on fixed roles, for example when only players that have selected a “medic” class are able to heal others. Roles can be situational as well, for example when everybody is able to heal others they depend on which player is currently injured.

¹ In contrast to our previous work [93], the requirement of actions being synchronous has been removed. For example, a player leaving an item for another one constitutes an asynchronous interaction. In our opinion there is no reason for not describing such interactions as collaborative, too. Nevertheless, we will still focus on synchronous interactions due to the nature of the application scenario described in Section 3.1.

5.2 COLLECTING INTERACTION INSTANCES

To find proven examples of collaborative player interactions, an analysis of several cooperative video games is conducted. The selection of games (full list in Appendix B) is based on online searches for well-received games, both in professional reviews as well as in player feedback. Afterwards, the initial list is supplemented with recommendations by fellow researchers and students. Aside from looking for popular games, we try to include as many different game genres as possible in order to be representative. Additionally, collaborative player interactions can also take place between players cooperating in a team, so games with team based multiplayer modes are included as well.

The analysis itself consists of playing the game and taking notes whenever a synchronous (Section 3.1), collaborative interaction instance is encountered. These notes are mostly informal, containing a short description of the interaction as well as a reference to the game and situation (for example a level) it appears in. Since the goal is to find common patterns, it is deemed sufficient to play each game for three hours. After that, a few random samples of game reviews and “let’s play”-videos are used to check whether there are noteworthy changes later in the game. This way, there is a high chance of noticing all common interactions in each game. Getting a complete list of all interactions would be unrealistic anyhow, especially due to the fact that some games have hidden sections. Some of the games are also still in development, therefore new interactions could be added at any point.

5.3 INTERACTION CLASSIFICATION AND POST-PROCESSING

Based on the unstructured notes, the interactions are classified along four dimensions to better understand their common aspects and differences:

- Where does the interaction happen (**Space**)?
- When does it happen and how long does it take (**Time**)?
- How does it relate to the player as a person (**Player**)?
- Which types of games can use this interaction (**Game context**)?

Each dimension is split further into several properties with associated value ranges and scales, which are refined until all notable aspects of the interactions could be covered. By design, all of these properties can be observed for every interaction instance that is extracted. If necessary, these dimensions and properties can be extended in order to accommodate new interaction instances that are identified in the future. For example, early experiences with virtual reality (VR) devices indicate that many aspects of traditional game design may not be applicable to this new setting. Should this be the case in regards to collaborative interactions, a new property could be added to the game context dimension to specify whether an interaction is possible in VR.

The **spacial** dimension consists of two properties, “spatial relation” and “spatial location”. The relation describes the position of the players in relation to each other while executing the interaction. They can either be at the same position (interaction is “collecting”) or be split into smaller subgroups (“separating”). This allows developers to set up further tasks, e.g. guaranteeing that the players are collected at the same location for witnessing an important event. The spatial location describes

whether the interaction is available at a “specific” location in the game world or is “pervasive” for longer sections or the whole game. Specifying that two players are at the same location requires a clear understanding on what a location is. Some games already define fine-grained locations, for example when the game is split into small rooms. Most games however consist of huge open spaces and require a definition of proximity in order to decide if two elements are at the same location. In our opinion, proximity should always consider the speed of the players in addition to the distance: If the movement speed of the players is very high, even a large game world feels small, as distant locations can be reached quickly. Considering this, we count two objects as being at the same location if a player can overcome their distance in less than a second. In games based on realistic speed and size measurements this translates to a few meters, which corresponds to what most people would say in reality.

The most important **temporal** aspect is the duration of the interaction. We divide this property into “short” (up to five seconds), “long” (more than one minute) and “medium” (everything in between). Here, only the time an interaction takes when executed faultlessly is counted – pauses or failed attempts are not considered. Additionally, if a short interaction is repeated multiple times it is still counted as short. Whether an interaction is repeated and how often is also not added as a separate property. Repeating the same interaction multiple times might become boring, hence this should be avoided in general. Whether the players’ actions happen at the same time (“synchronous”) or not (“asynchronous”) also represents a temporal property. But this property is omitted for this work because only synchronous interactions are collected. The interaction mechanics themselves do not limit the time during which a specific interaction can happen, neither relative to another one nor based on the absolute playtime. Therefore, no temporal or causal availability property is defined. If required, the availability of any interaction can be controlled by the developer via additional scripting, though.

The first aspect of the **player** dimension is difficulty, which is broken down into the properties “communication” (sharing information) and “timing” (synchronizing individual contributions). Both properties can take the values “low” (not necessary or impossible to fail), “medium” (could fail for novice players, but no obstacle for experienced players) or “high” (essential and will regularly fail). While being impossible to measure objectively on a more detailed scale, these three categories offer a reasonable amount of distinction between interactions. Keeping these requirements in mind is especially important when the game only offers limited communication tools or the target audience is not familiar with teamwork. Another property of the player dimension describes whether the players are forced to take part in an interaction (“mandatory”) or not (“voluntary”). Voluntary interactions often offer benefits for taking part in order to discourage players from ignoring them. However, there is also a middle ground for interactions that can be used both as mandatory and voluntary. If the bonus provided by an optional interaction is large and the game itself is difficult, an interaction that is voluntary in theory can become mandatory in practice. As a last property in regards to the player the subjective experiences the testers have during these interactions are noted, for example if they get frustrated because they have to wait for another player. However, due to the small sample size and the absence of a formal user study, these results should only be seen as pointers for future work and not as conclusive results.

In the **game context** dimension, the role flexibility describes whether an interaction can be used in games with “fixed” (players choose classes or characters providing them with different options) or “free” roles (abilities can be reassigned during play or there are no differences at all). It is also important how many players take part in an interaction. Obviously, this cannot exceed the number of players in the game. However, it can be lower if only a subset of players is required, which allows them to choose who participates. Lastly, the game genres [7] in which the interaction typically appears is described. Examples for this include action, strategy or role-playing games. Although experienced developers might be able to adapt the interaction for other kinds of games, novices should keep to this suggestion. This way they are able to use the interaction in a way similar to existing implementation examples.

In general, there are interdependencies between the four dimensions. The larger a space is, the longer is the time it takes to (physically) move through it. Certain game genre conventions favor different types of interactions in regards to space and time. For example, role playing games often contain larger worlds, whereas faster action games usually contain shorter interactions. And obviously, every property could have an impact on the player experience.

The individual properties of each dimension, however, are intentionally defined to be independent, so that there is no redundant information. For example, one could think that an interaction being mandatory during a larger section of the game takes a longer time. However, this could also describe the repeated use of a short interaction. Longer interactions in turn can also happen in a small location if the players stay immobile. While some game genres might favor certain interaction properties, genre definitions themselves are generally broad. This means that there is no strict dependency between the genre classification and other individual aspects of the interaction. The assumption that difficult interactions take more time is also not true when perfect executions are assumed. There is also no clear connection between role flexibility and player requirements. On the one hand, fixed roles make it clear what a player has to do and thereby decrease the need for discussions. On the other hand, free roles decrease the dependencies between players, which could also reduce communication. Only for the subjective player experience one cannot prevent dependencies to other properties. But since the subjective experience is different for each player anyhow and serves only as additional information, this is not problematic.

After classification the interactions are post-processed. First of all, the notes are brought to the same abstraction level. Based on the intended use as templates, the interactions have to be clearly defined while having adaptable variables that allow variations. Interactions that are functionally identical on this abstraction level are merged. As an example, the merged “Restore” interaction is based on the interactions of players healing others in some instances and giving ammunition to them in others. A side effect from this generalization is that certain actions in a game can instantiate more than one interaction. Pushing another player away from danger is mechanically an instance of the “Transport” interaction, but its result also happens to fit the descriptions of the “Savior” interaction (Appendix C).

After merging, it became clear that some of the classification properties define an interaction. In these cases the property has the same value for all instances. Other properties show multiple values and thereby constitute different contexts in which the interaction can be used. Which properties define an interaction and which ones are variable varies between the interactions.

5.4 DESIGN PATTERN FORMALIZATION

As a formalized description of the abstracted interaction instances the well-known concept of game design patterns (Section 2.3.1) is selected. Specifically, the format proposed by Björk et al. [19] is chosen, as it is designed for describing game design aspects with the goal of inspiring future games. This fully matches the intended use of the interactions in this work. Another benefit of the format is that it is widely accepted by researchers in the field, which results in a large number of potential sources for related design patterns while making our results more relevant to others.

However, the format by Björk et al. is extended based on the classification properties (Section 5.3), as they describe the most important aspects of the collected interactions in a more precise way. This is supposed to make them easier to use. For example, the number of players participating in an interaction allows a developer to immediately see which patterns can be used with the player number they are aiming for in their game. To keep the formats compatible, the new properties are sorted into the existing ones already supported by the format. As already mentioned, these new properties can either define an interaction or provide a means for variation. If a property defines the interaction, it is clearly a part of the “consequences” in the pattern format. But if there is a range of possible values, the property is related to “using the pattern”.

After the interaction instances have been converted into game design patterns using this format, they are grouped into categories based on the central element of their description. This makes it easier to find a pattern that fits the game being currently worked on. These categories are:

GENERAL Universally applicable patterns that describe high-level concepts and that can be realized using a wide variety of concrete mechanics.

PROGRESSION GATES Universally applicable patterns that control player progression by imposing requirements before allowing them to continue.

INFORMATION EXCHANGE Patterns that require the players to exchange information and therefore need the game to provide some means of communication.

PLAYER SUPPORT Patterns describing direct interactions in which one player provides an immediate benefit to another, requiring a (visible) avatar to be targeted. These patterns are mostly used in team-based action games and are known by lots of players.

NPCS Patterns that describe interactions with Non-Player-Characters, which need a visual representation that can be interacted with.

MOVEMENT Patterns that concern the players’ movements in the game world, requiring the movement to be visible and reasonably nuanced.

RESOURCES Patterns that need multiple resources or items which the players can use or trade.

COMPETITION Patterns that relax collaboration by introducing competitive elements.

Despite being no concrete player interactions, two additional categories are deemed useful nonetheless and therefore added as well:

MODIFIERS These patterns modify other interactions in a fundamental way, but are no complete interactions themselves.

ADDITIONS Patterns that already exist in other collections, but to which novel aspects can be contributed.

These classification properties and categories allow a workflow in which developers search for patterns matching their requirements, for example a collaborative adventure game for two players with fixed roles. Then the matching patterns can be applied to the game, with their variable properties being adapted based on the developer's design goals. After that there is still some creative work remaining, like the integration of the interactions into the game's narrative. This is necessary due to the patterns being intentionally abstract enough to be applicable to a wide variety of games and settings.

The complete collection of patterns that are created in the course of this thesis can be found in Appendix C. This collection is intended to be a proof of concept showing the viability of the player interaction pattern concept. It is not meant to be complete, as there is lots of creativity involved in creating games and it is very likely that experienced developers will continue to come up with new collaborative player interaction mechanics. Those new mechanics could however easily be classified using the proposed properties and then transformed into the given pattern format.

5.5 PROCEDURAL CONTENT GENERATION USING PATTERNS

The interaction patterns, when provided as configurable building blocks, can be linked in order to build the game's structure. This structure can be visualized by exploiting some of the patterns' properties.

First of all, the patterns' logical dependencies, i.e. the order in which the interactions have to be executed, can be visualized. Figure 11 shows an example with three players, in which the *Patterns A*, *B* and *C* have to be executed in a strictly linear order. This is symbolized by directed edges connecting the patterns. The description on the edges defines which players take part in each interaction instance. For the first two patterns every player has to take part. The interaction used in *Pattern C*, however, requires the first player (constant P_1) to take part, together with any other player (variable P_y). The remaining player (variable P_z) is not involved and has an edge avoiding the pattern.

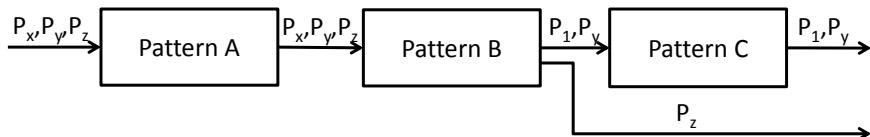


Figure 11: Example for visualizing logical relations between interaction patterns.

This logical structure can be combined with a visualization of the game world in order to display the spatial properties of the patterns (Figure 12). Each pattern is placed in the location where the interaction takes place (if it is specific) or in an abstract "meta-location" encompassing all locations in which it is available (if pervasive). When the players are separated for an interaction this can be visualized by the pattern spanning multiple locations and the players' edges are split up accordingly. In the example, *Pattern A* is located in *Location 1*. Player separation can be displayed by drawing the pattern over multiple locations, as can be seen with *Pattern B*. This interaction is conducted by one player in *Location 1* and two in *Location 2*. However, the overlay visualization can become quite confusing if the underlying game is complex, especially when players have to revisit locations.

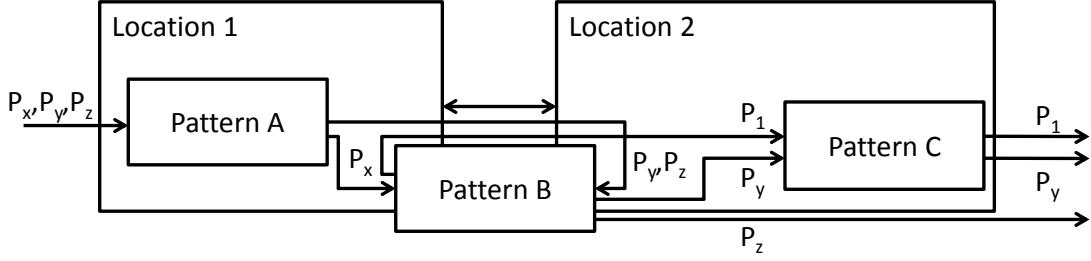


Figure 12: Example for visualizing logical and spatial relations between interaction patterns.

More complex concepts like branching and non-linearity can be modeled, too (Figure 13). If the players can decide between multiple paths or challenges, the alternatives are added to a *1 out of X* container. In the example the players can choose between interactions based on the patterns *A* or *B*. It is also possible to do several challenges in any given order (e.g. in form of a *2 out of 2* container). Partial orders are supported as well – in this case *Pattern D* must be done before *E*. By definition, optional interactions have to be disconnected in the dependency graph as no other interactions depend on them. Instead, they are added in dashed containers signaling the game section during which they are available. Dependencies can exist between voluntary interactions (*Pattern G* only after *F*, but only *F* or neither of them is also possible). Sometimes patterns can (when optional) or have to (when mandatory) be repeated. This is indicated by a number in the upper right corner of the pattern. In the example, *Pattern A* has to be done twice and the optional *Pattern G* can be repeated infinitely. If no number is given, one-time usage is assumed as the default case.

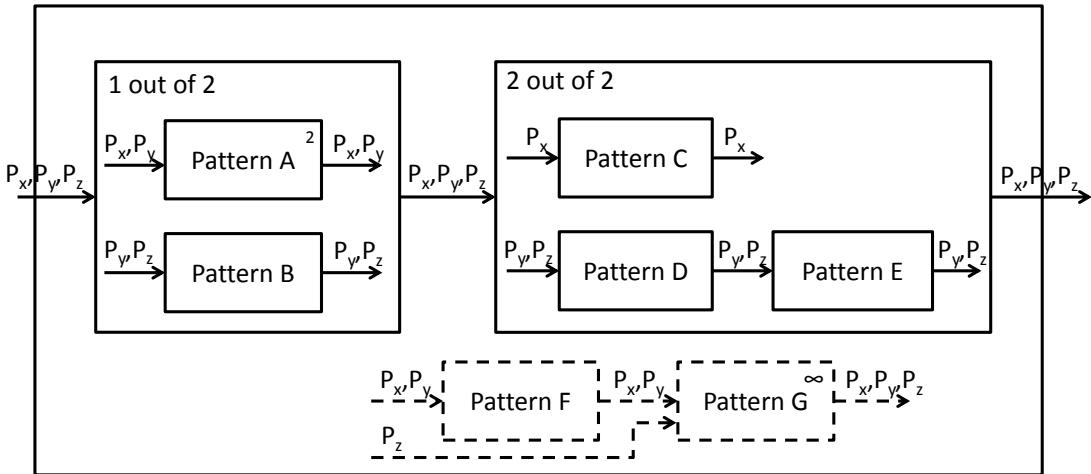


Figure 13: Example for visualizing complex relations between interaction patterns.

The patterns' properties do not only enable visualization, but also algorithmic usage. This allows the procedural generation of a complete game structure, which has been explored together with Tregel [117]. For this, all properties with descriptive values like "low", "medium" and "high" are mapped to numeric values first. Afterwards the patterns and their properties are added to a database. Next, an algorithm is designed that iteratively constructs the game's structure in a breadth-first manner. In each of these expansion steps either a branch, a merge (inverted branch),

an optional pattern or a linear section is generated depending on whether they fit the already existing structures. Then, all patterns matching the requirements of the current step and the user defined parameters are selected from a database using a fitness function. This function sums up the differences between the pattern value and the required value for each property. After the list of pattern candidates has been narrowed, some additional rules are applied, for example to prevent pattern repetition. Finally, the pattern instances are configured with appropriate values for their variable properties. This process is repeated until the intended game length has been reached. As a concluding step the whole structure is fed into a post-processing step which balances the playtime on alternative paths. It also finalizes pattern parameters which are dependent on the overall structure, for example to raise the difficulty over the course of the game.

However, the result is a structure of named pattern instances and parameters, not a concrete implementation using actual game objects. Therefore, we have explored together with Schölei [102] how these pattern instances can be implemented as concrete puzzles for an adventure game. In combination, this can allow the generation of a complete adventure game structure for which only the content, for example graphical assets and story texts, must be added. Additionally, a mixed approach in which the system suggests appropriate patterns for the developer to use (guided process) is also possible using this algorithm.

5.6 DESIGN OF AN EVALUATION GAME

Since abstract patterns cannot be evaluated directly, an evaluation game implementing a representative selection of them as concrete challenges is designed. The main idea is to compare two versions of the same game, one implementing the patterns and one without these collaborative interactions. Aside from this difference both version should be as similar as possible in regards to the game mechanics, visual style and overall feel to make them comparable. An example design for this can be found in Figure 14. Comparing the players' reactions to both versions can yield important information on how the interaction patterns are received.

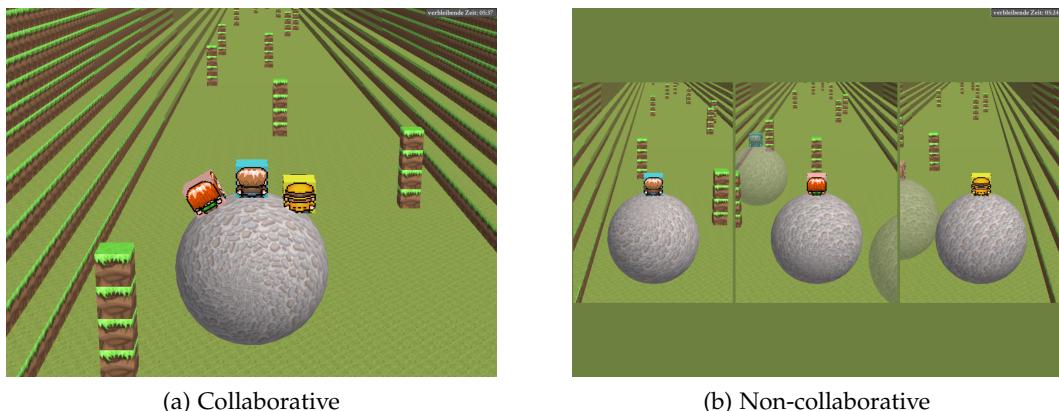


Figure 14: Visual comparison between the two versions of the “Boulder Ride” level in the pattern evaluation game.

The initial design for this game has been developed together with Uhlig [118], to whom only the pattern collection (Appendix C) has been given at first. This way some additional feedback, from the perspective of a developer using the patterns, could be gathered. For example, the pattern descriptions are deemed generally understandable. The resulting game prototype has been refined later for this thesis, especially by adding new levels covering further interaction patterns.

On the one hand, it has been decided to design the game for as few players as possible to make evaluations easier. On the other hand, it should also contain collaborative interactions between the players which do involve some, but not all of them. This resulted in a player count of three. The game is played on the same computer using a single screen (using both shared screen and split screen [114] concepts depending on the level). This makes evaluations easier as it requires less hardware, but also prevents the distribution of visual information between the players because everybody has the same view. The game is controlled with one gamepad for each player, as a single keyboard is too small for multiple players. Gamepads offer another benefit as well: Most models possess rumble motors for vibrational feedback, which can be used as an information channel hidden to the other players.

In order to address as many players as possible, the game should not require any experience in playing games and should be solvable by everyone. Therefore the final version is designed to be easy to understand and to play, with simple controls (only movement as well as jumping) and no harsh penalties for mistakes. Additionally, destructive play that could hinder the group's progression is limited as much as possible. Some types of malicious behavior could be removed completely, for example by preventing players from pushing others into traps. Other kinds of behaviors, like repeatedly jumping in a trap themselves, could not be prevented without actively interfering with the players. Such an influence would compromise the validity of the evaluation's results. Instead, these actions are designed to be more boring than playing the game normally. For example, there are no interesting animations or sound effects when jumping into a pit.

To gather feedback on individual patterns, each level of the game implements one interaction pattern as the general theme. The main gameplay requires the players to reach the end of each level by navigating the environment and solving puzzles based on the central interaction patterns. This classifies the game as an *Action-Adventure*. At the end of each level there is an additional "Gathering Gate" pattern to make sure that every player experiences the complete level, even if its main pattern does not guarantee this. All additional elements aside from the interaction puzzles are kept minimal in order to focus the evaluation results on the patterns.

Another goal is that, for practical concerns regarding the evaluation, one play session should fit into a 30 minute time slot. Since each level should be long enough for the players to understand and to apply the interactions required to solve it, the overall game is limited to five levels. The patterns used in these levels are selected to be as representative as possible, which means covering as many different property values as possible. For example, they include optional and mandatory as well as collecting and separating ones:

LITTLE WALK The first level serves as a tutorial by explaining the basic movement controls. To ease the players into collaboration, only one optional interaction pattern is used in a short section of the level. There is one boulder that must be moved,

which is faster when multiple players push from different attachment points (“Parallelization” pattern). For the non-collaborative version of this level even a single player alone can push the boulder at full speed.

BOULDER RIDE In the second level, all players are standing on a rolling boulder and must avoid obstacles. Since the movement inputs of all players are combined, they need to coordinate themselves in order to move into the right direction (“Transport” pattern). Because this is the first time the players have to work together, there is no big penalty for failure at first: the boulder is only slowed down. It also features a simplified movement at first, which allows the players to only move left or right. The later section of the level introduces pits over which the players must jump. Similar to the sideways movements, players must synchronize their jumps in order to succeed. This does not only introduce another movement option, but also increases the difficulty by forcing the players to repeat a failed jump until they succeed. Consequently, the players must collaborate in order to finish this level. For the non-collaborative version of this level each player is placed on an individual boulder, so they can avoid obstacles on their own. There is no collision between the players so they cannot interfere with each other (Figure 14).

MAZES The level is designed to guide the players towards voluntary collaboration by making it obvious that sharing work can save them time. This is done by having two consecutive tasks that are triplicated (finding a path through a maze and finding out which chest contains a key). The level layout suggests that the players work on these copies in parallel, but it is also possible for one player to do all of them consecutively (“Parallelization” pattern). For the non-collaborative version of this level, each task consists of only one component.

CAVE At this stage, the players should know that they are required to collaborate and have also been familiarized with each other. Therefore, the level is build in such a way that it requires constant communication and coordination, with the players being highly dependent on each other. This is done via the “Perspective” pattern, which gives every player different information about the cave they are exploring. One player can detect the metal vein they are supposed to follow via a continuous high frequency rumble of the gamepad. Another one can sense dangerous ground with a low frequency rumble. The last player can detect arrow traps in a small radius around him and mark them on the screen. This also forces the players to stay together if they do not want to run into traps. For the non-collaborative version of this level all types of information are available to every player.

SEPARATED After the players have learned how to collaborate, the last level forces them to split up again by using the “Separation Gate” pattern. Players must climb onto each other and then press buttons at different locations, after which there is no way back for them. They can only reunite after each of them has solved a challenging section on their own. This means, that the other players need to wait if one of them has difficulties with his or her section. For the non-collaborative version of this level there is only a single path for all players.

Some ideas for later extensions are also developed, but have been left out in the final version to keep the playtime from exceeding the planned evaluation time:

FIGHTING SYSTEM A complex fighting system could instantiate several patterns at once. One of the players could have a limited number of arrows, with supplies being available from one of his or her colleagues (“Resupply” interaction). Enemies could only be vulnerable when pinned down by one player (“Weakening”), during which this player has to remain stationary and must be protected from other enemies (“Protector”). This could be done by luring the other enemies away (“Distraction”), if combat is not viable against a large number of targets. If a player is hit by an enemy, the others could help him up again (“Savior”) and the game is only reset if all players are incapacitated at the same time. For the non-collaborative version all players could have infinite arrows, enemies can be attacked directly and there is an automated recovery after being hit.

RIVER As a more literal, but also asymmetric implementation of the “Transport” pattern, one player could control a raft to avoid rocks. At the same time, the other players have to fend off enemies that try to attack the raft (“Protector”). For the non-collaborative version there could be multiple rafts and the players could stop to attack the enemies on their own.

5.7 SUMMARY

In this chapter, an extendable concept for using well-established collaborative player interactions as design patterns is described. These patterns can be used as an inspiration or as predefined building blocks for collaborative multiplayer games.

Since the patterns describe interactions in an abstract way, they can be adapted to the individual developer’s needs and can be varied for repeated use. The pattern’s usefulness has already been validated by integrating them into a demonstration game. The expectation is that this game is more engaging because of these interactions, which is later evaluated in a user study (Section 10.1).

Aside from the pattern concept itself, this thesis also contributes to the quasi-standard of game design patterns introduced by Björk et al. [19] by showing that the format can be augmented with specific information for a certain domain, in this case player interactions. Moreover, it also provides a list of 24 new patterns that are compatible with the standard (Appendix C).

In the context of the overall authoring process, this component can help with the creation of the game. However, ensuring the quality of the resulting game is the task of the following verification step, which is described in the following chapter.

STRUCTURAL VERIFICATION OF COLLABORATIVE MULTIPLAYER GAMES

GAMES are supposed to be free of structural errors, especially of ones that prevent the players from reaching a valid ending. This should be regularly checked during development, as fixing structural problems gets more costly later on. However, it cannot be guaranteed that human testers will try every possible path through the game. In contrast, formal verification is able to conclusively decide whether such problems exist – but only for formal models. Therefore, our approach is to transform games from our application scenario (Section 3.1) into such a formal description [94], which can then be analyzed with existing verification methods.

6.1 PROBLEM DEFINITION

Syntactic problems, like missing start points and unconnected locations, are relatively easy to detect by simple searches. Smaller semantic mistakes, like conditions containing logical contradictions, also can be found relatively easily by looking at individual game elements. Complex player tasks, however, can involve several locations, events, variables and players. Detecting errors in such a task definition requires the verification of all combinations in which these elements can interact. To give an example:

There are multiple connected rooms between which the players can freely move. In one of them is a key located that any player can pick up. This key can be used in another room to open a storage closet containing a ladder. A third room has a large hole in the floor, where a player has to jump in order to retrieve an important item. There is no way to leave the hole without the ladder, which can be carried while jumping or dropped in from the outside by another player.

Each of these elements seems fine when viewed individually and there are even multiple ways to solve the puzzle. For example, one player can either take the ladder into the hole or his friends can put the ladder in from the outside. Even if the players forgot the ladder or the key at first, they can still be retrieved after one player jumped into the hole. However, there is one fatal flaw. If one player picks up the key and then jumps into the hole there is no way to open the closet afterwards. Therefore, the ladder cannot be retrieved and the player is trapped in the hole forever, preventing everyone from finishing the game. This kind of problem is hard to detect manually, especially when a puzzle spans a greater number of elements or larger sections of a game. Even having players testing the game might not always help to find such problems. Cautious players would delay jumping into the pit as long as possible, so they will never actually run into this problem.

Therefore, the game should be verified against these kinds of critical problems (based on Tai[112], but applied to games):

Deadlocks are situations in which the players cannot change the game state anymore, for example when being trapped.

Livelocks are situations in which the players can change the game state, but cannot reach a well-defined ending. An example for this is a trap in which the players can toggle a light switch, but are unable to leave.

Errors that do not prevent the players from reaching an ending are less critical. However, having designed a section of a game that can never be reached by the players is hardly the intention of the developer. Since detecting dead- and livelocks requires a full exploration of the game's state space, two additional error classes can be detected in parallel:

Unreachable locations cannot be entered under any circumstances. This can happen if there is no connection to them or when these connections are guarded by unsatisfiable conditions.

Impossible events can never be triggered, for example due to unsatisfiable conditions. Another example would be an event being tied to an object that is always out of reach for the players.

A secondary, but nevertheless relevant concern is that these errors should be detected in such a way that the result is easily interpretable for game developers.

6.2 FORMAL GAME MODEL

In order to verify that such problems cannot occur, a formalization of the general game model (Section 4.1) is required. First of all a game is made for multiple **players** $P_l = \{p_{l_0}, \dots, p_{l_{|P_l|-1}}\}$, of which there must be at least one ($P_l \neq \emptyset$). The constants p_{l_i} are used to access the i -th player of P_l and $p_{l_x} \in P_l$ denotes a variable for an arbitrary player.

Each game world consists of several **locations** $L = \{l_0, \dots, l_{|L|-1}\}$, $L \neq \emptyset$. Similar to the players, l_i is defined to be the i -th location of L . One of these locations is marked as the starting point $l^I \in L$.

Variables are defined as $V = \{v_0, \dots, v_{|V|-1}\}$ and are used to represent both helper variables and the state properties of game objects. Variables are optional, which means that V could be empty. Again, v_i is defined to be the i -th variable of V .

Variables have **types**, with multiple variables being able to share the same type:

$$V_t = \{v_{t_0}, \dots, v_{t_{|V_t|-1}}\} = \bigcup_{v \in V} \text{typeOf}(v)$$

The most commonly used types are *boolean*, *integer*, *float* and *string* ($V_t = \{\text{Boolean}, \text{Integer}, \text{Float}, \text{String}\}$). Since every variable needs a type there has to be at least one type if the game uses variables ($V \neq \emptyset \implies V_t \neq \emptyset$). The type also defines the value range of each variable ($\text{values}(v_t)$), for example *true* and *false* for Booleans.

The game can assume several **states** $S = \{s_0, \dots, s_{|S|-1}\}$, $S \neq \emptyset$. A state contains the current location of each player ($\text{location}(p_l, s) \in L$) as well as the value of each variable ($\text{value}(v, s) \in \text{values}(\text{typeOf}(v))$). Due to this similarity, the player location can also be interpreted as a “player-variable” whose value is a reference to a location. From this perspective querying the current location constitutes a reading operation and a player moving means writing this variable. One state is designated to be the initial one ($s^I \in S$), in which all players are at the starting location ($\forall p_l \in P_l : s^I.l = l^I$).

$\text{location}(\text{pl}, s^I) = l^I$). Other states can be marked as end states $S^E = \{s_0^E, \dots, s_{|S^E|-1}^E\}$, $S^E \subseteq S$. In these states the game ends intentionally after the players have finally reached their goal or ultimately failed to do so. There must always be at least one well-defined ending ($S^E \neq \emptyset$) and unintentional game endings like deadlocks are not part of S^E .

The game state can be changed by **events** $Ev = \{ev_0, \dots, ev_{|Ev|-1}\}$ and ev_i is defined to be the i -th event of Ev . These events are most commonly triggered by player actions, but could also be executed based on a timer. There must be multiple events for the game to be interactive ($Ev \neq \emptyset$). Each event can yield a number of **reactions** r_i by the game, which can have additional conditions cn_i attached. Therefore, the reactions that are actually executed depend on the current state of the game and which player triggers them ($\text{reactions}(ev, s, pl)$). After executing the event and applying all associated reactions, the next game state is $\text{nextState}(ev, s, pl) \in S$. It is also assumed that each event is only available at a certain location $\text{location}(ev) \in L$, which means that a player must be at this location in order to trigger it. But this does not limit the expressiveness of the model, since an exact copy of the event could be available at another location, too.

6.3 PETRI NET MODEL

This general game model is translated into a petri net model (to which a short introduction can be found in Section 2.4.2) for verification. The main reason for this decision is that petri nets have been developed to model concurrent executions, which is important when multiple players interact with the game simultaneously. Additionally, the verification of petri net properties like liveness (whether the state of the model can be changed) or the reachability of certain states, has already been well researched. There are also tools available that provide best practice implementations for verifying petri nets.

The usage of formal languages and symbolic model checkers as an alternative has been explored together with supervised student [12]. Although this approach works as well, the result is much harder to interpret compared to the visual model of petri nets, in which the token movement can easily be observed. This token movement is also relatively similar to the movement of players in a game, further facilitating the interpretation of the result. Developers are hereby enabled to map a state of the petri net, for example a deadlock, to a situation in their game and then act accordingly.

Class-based games differentiate between various player roles or even individual players in order to allow them to execute different actions. To support such games, this kind of differentiation has to be possible in the verification model, too. Therefore, the subclass of colored petri nets is used because it allows the differentiation between multiple tokens.

The petri net model of a game should be created automatically in order to prevent inconsistencies between the model and the actual game. If the model was created manually instead, transference mistakes would become more likely. In order to automate the model creation, clear transformation rules must be specified. These rules must contain a mapping from each game element to one or more colored petri net elements. These elements include colors Σ , places P , transitions T , arcs A , a function N for mapping arcs to place/transition pairs, a function C for assigning colors to

places, a function G that assigns guard expressions to transitions, a function E that maps annotations to the arcs, and a definition of the initial token placement I . A more detailed explanation of these elements can be found in Section 2.4.2.

6.3.1 Game State

As already mentioned in Section 6.2, the current game state is defined by the location of each player as well as the state of all variables. The current state of a colored petri net is defined by the tokens that are present at each place as well as the values of these tokens. Therefore it is natural to map the game state to these elements.

The best way to map the player locations to a petri net element is to have one place for each location and exactly one token for each player. The place that a player's token is in then corresponds to the player's location in the game. This makes it easy to follow the players' movements visually when their tokens move through the petri net. In order to differentiate between the tokens of different players their ID should be stored in their token's value.

Variables are mapped differently, with exactly one place for each variable in which there is always one token. The value of the variable is encoded in the value of the token, which can be changed by petri net transitions.

Alternatively, the player locations and variables could be mapped the same way. Although theoretically possible, this would yield some critical disadvantages. Encoding the player location as a token value would make the petri net harder to read, while having the variable tokens move would result in a model which is too complex (Section 6.3.4 for a detailed discussion of this alternate model). The result is the following definition for the places of the petri net:

$$P_L = \bigcup_{l \in L} p_l \quad P_V = \bigcup_{v \in V} p_v \quad P = P_L \cup P_V$$

In the following, these two types of places are called "location-places" (P_L) and "variable-places" (P_V). The auxiliary functions $\text{location}(p) \Rightarrow L$ (location-places) and $\text{variable}(p) \Rightarrow V$ (variable-places) are used to address the game elements encoded in this place.

During the initial state the player tokens are at the place associated with the starting location l^I , while all other location-places are empty. In each variable-place there is one token which holds the initial value of this variable, resulting in the following initialization function:

$$I(p) = \begin{cases} P_l & \text{if } p \in P_L \wedge \text{location}(p) = l^I; \\ \emptyset & \text{if } p \in P_L \wedge \text{location}(p) \neq l^I; \\ \text{value}(\text{variable}(p), s^I) & \text{if } p \in P_V. \end{cases}$$

Having tokens symbolize both players and variables, we suggest differentiating them by the use of colors. To provide more clarity each variable type should also have its own color:

$$\Sigma = \{c_{\text{player}}\} \cup \bigcup_{vt \in Vt} c_{vt}$$

For most games this will lead to the colors $\Sigma = \{c_{\text{Player}}, c_{\text{Boolean}}, c_{\text{Integer}}, c_{\text{Float}}, c_{\text{String}}\}$.

Tokens can only enter places which are annotated with their color. Therefore the player color is assigned to all location-places and the variable-places are colored with the type of the variable:

$$C(p) = \begin{cases} c_{\text{Player}} & \text{if } p \in P_L; \\ c_{\text{typeOf(variable}(p))} & \text{if } p \in P_V. \end{cases}$$

Since tokens can only exist in places that share their color, they implicitly inherit the color of their initial place.

A partial example net is provided in Figure 15. It represents a game for two players (pl_1 and pl_2) that consists of two rooms, the starting location *Hall* and *Storage*. The *Player* color is assigned to both rooms. Aside from that one additional variable (*Open*, Boolean color and initially *false*) is used. All other elements related to the game events are left out to improve the clarity of the example.

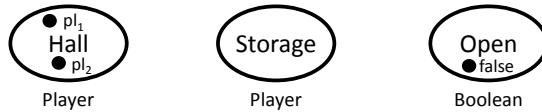


Figure 15: A partial example net encoding only the current game state.

6.3.2 Game Events

Events change the game state and transitions change the state of the petri net. Therefore, a direct mapping between these elements seems promising. Additionally, game events and transitions are both atomic in their respective domains. But it is important to note that each game event can consist of multiple game reactions, of which only a subset is fired depending on their conditions and the current variable values. Since the effect of petri net transitions is always the same, it is not possible to model this varying behavior directly. Instead, the events must be linearized, i.e. split up into all possible linear paths that can be taken through them:

$$LEv = \bigcup_{ev \in Ev} \{\text{linearizations}(ev)\} = \{lev_0, \dots, lev_{|LEv|-1}\}$$

lev_i is defined to be the i -th event of LEv . These linearized events are available at the same locations as the original event ($\forall ev \in Ev, \forall lev \in \text{linearizations}(ev) : \text{location}(lev) = \text{location}(ev)$). An alternative that maps each game reaction to a transition would not need such a linearization. It would, however, violate the atomicity of the game events because petri net simulators are allowed to fire each transition independently.

The separation into paths results in at least one linearization per original event ($|LEv| \geq |Ev|$). Each new event consists of a single combined condition and an ordered set of reactions:

$$\text{condition}(lev) : f(s, pl) \Rightarrow \text{boolean} \quad \text{reactions}(lev) = (r_0, \dots, r_n)$$

The reaction set in turn contains all reactions of the original event that would have been executed in the states for which the linearized condition is true. An example for this can be seen in Figure 16.

If the branching conditions are exclusive, then the conditions of all linearized paths are mutually exclusive, too. This means that for each game state at most one linearization per original game event is executable. In this case, the petri net simulator firing any enabled transition always matches the behavior of the original game. Subsequently, a direct mapping from the linearized events to petri net transitions is valid:

$$T = \bigcup_{lev \in LEv} t(lev)$$

Furthermore, each linearized condition is directly usable as the guard function for the corresponding transition: $G(t(lev)) = \text{condition}(lev)$.

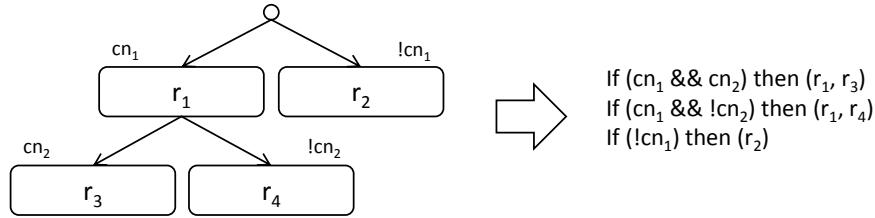


Figure 16: Event linearization example.

Having a single condition per linearized event allows the definition of two auxiliary functions. All variables that are read in the condition of a linearized event can be written as `readVars(lev)`. Since the current states of variables and player positions are modeled in different ways, the player-variables read are defined separately as `readPVars(lev)`. It is important to note that these lists can be empty for events that can be triggered under any circumstances.

In order to evaluate whether the condition of a transition is true, it must be connected to the elements holding the necessary state information. For each variable being read (Figure 17) this means that the place associated with this variable must be connected to the transition with an incoming arc (α_{VarIn}). This arc must be annotated with the variable name to make it accessible by the guard condition. In order to not consume this token, an outgoing arc (α_{VarOut}) to the same place with the same annotation is added.

To tie the event to its game location, another incoming arc ($\alpha_{PlayerIn}$) is drawn from the location-place to the transition. The arc is annotated with a unique identifier in order to differentiate it from the other arcs connecting variables (Figure 18, lower transition). If the condition requires a certain player to trigger the event, this identifier can also be referenced in the guard condition (Figure 18, upper transition). Similar to reading variables, an outgoing arc ($\alpha_{PlayerOut}$) with the same properties is added by default.

Conversely, outgoing arcs are used to write the effects of the game's reactions back to the game state. If a variable is written, an outgoing arc (α_{VarOut}) to the variable's places is created, annotated with the new value as a constant (1) or as a formula ($var + 1$). Since a variable can only have one value at a time, this production of a new token must be accompanied by the consumption of the old value. Therefore, a matching in-arc (α_{VarIn}) is needed (Figure 19, upper transition). However, if the variable is

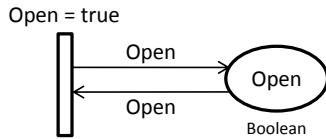


Figure 17: Petri net model of a boolean variable being read.

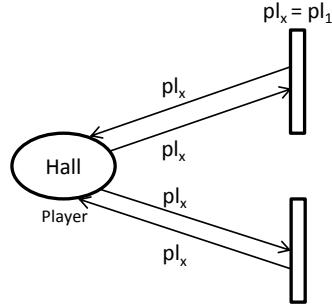


Figure 18: Petri net model of a player triggering two events.

already read in the condition then both aspects must be combined, with an incoming arc for reading and an outgoing arc for writing (Figure 19, lower transition).

Events can also move the triggering player to another location. In this case, the outgoing edge for the player ($\alpha_{PlayerOut}$) is switched to the new location's place (Figure 20, lower transition). This can also be combined with reading the player-variable (Figure 20, upper transition). Other reactions that do not affect the game state (like playing a sound effect) are not relevant for the structural verification and therefore do not need to be modeled. However, in order to verify that a certain irrelevant action happens, an additional outgoing arc to a special place could be added. This means that every time this action is executed, a token is created in this place.

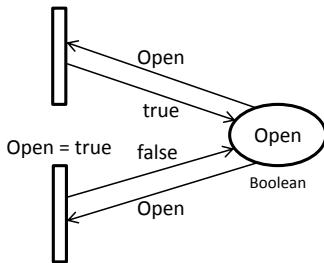


Figure 19: Petri net model of a boolean variable being written.

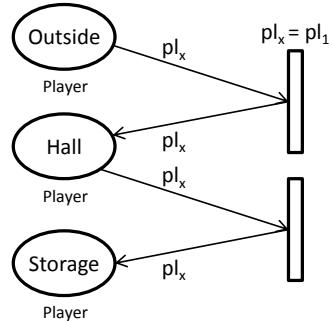


Figure 20: Petri net model of a player being moved to another location.

In order to completely formalize the petri net definition for game events, some auxiliary functions are needed. The variables written by an event are denoted with $writeVars(lev)$, which can be empty. Based on that, the used variables (read and/or written) can be defined as $usedVars(lev) = readVars(lev) \cup writeVars(lev)$. After executing the event the new value of a given variable is $newValue(lev, v)$, which can be the same as before when it is not written. And the next location of the triggering player, which might also be the same as before, is $nextLocation(lev)$.

Using these definitions the arcs can be defined:

$$\begin{aligned} A &= \bigcup_{lev \in LEv} \text{arcs}(lev) \\ \text{arcs}(lev) &= \{a_{\text{PlayerIn}}(lev), a_{\text{PlayerOut}}(lev)\} \cup \\ &\quad \bigcup_{v \in \text{usedVars}(lev)} \{a_{\text{VarIn}}(lev, v), a_{\text{VarOut}}(lev, v)\} \\ &= \{a_0(lev), \dots, a_{|\text{arcs}(lev)|-1}(lev)\} \end{aligned}$$

The node function for mapping arcs to place/transitions-pairs is as follows:

$$N = \bigcup_{a \in A} N(a) \subseteq (P \cdot T) \cup (T \cdot P)$$

$$N(a_i(lev)) = \begin{cases} (p_{Src}, t(lev)) & \text{if } a_i(lev) = a_{\text{PlayerIn}}(lev) \wedge \\ & \text{location}(p_{Src}) = \text{location}(lev); \\ (t(lev), p_{Dst}) & \text{if } a_i(lev) = a_{\text{PlayerOut}}(lev) \wedge \\ & \text{location}(p_{Dst}) = \text{nextLocation}(lev); \\ (p_{Src}, t(lev)) & \text{if } a_i(lev) = a_{\text{VarIn}}(lev, v) \wedge \text{variable}(p_{Src}) = v; \\ (t(lev), p_{Dst}) & \text{if } a_i(lev) = a_{\text{VarOut}}(lev, v) \wedge \text{variable}(p_{Dst}) = v. \end{cases}$$

And the function for mapping expressions (or annotations) to the arc is:

$$E(a_i(lev)) = \begin{cases} \text{triggeringPlayerID} & \text{if } a_i(lev) = a_{\text{PlayerIn}}(lev); \\ \text{triggeringPlayerID} & \text{if } a_i(lev) = a_{\text{PlayerOut}}(lev); \\ v & \text{if } a_i(lev) = a_{\text{VarIn}}(lev, v); \\ \text{newValue}(lev, v) & \text{if } a_i(lev) = a_{\text{VarOut}}(lev, v). \end{cases}$$

A full example with two events combining multiple primitives can be found in Figure 21. There are two rooms, *Hall* and *Storage*. The two players (pl_1, pl_2) start in the *Hall* and can move to the *Storage* via the lower transition. However, the guard expression on this transition requires the door to be open. This is modeled using the helper variable *Open*, which is initially *false*. Therefore its value must be changed first, which is done via another event encoded in the upper transition. Due to the guard condition ($pl_x = pl_1$) only the first player is able to do so.

The example net is shown in its initial state (Figure 21a), in which only the transition for opening the door is enabled. After executing this event the variable value changes and the lower transition becomes enabled (Figure 21b). Now, any of the two player is able to change its location, for example pl_2 (Figure 21c). Afterwards the transition is still enabled and pl_1 is able to follow (Figure 21d).

6.3.3 Extensions

Based on this basic model, a few extensions are defined. These include an explicit ending (Section 6.3.3.1), tying events to objects instead of locations (Section 6.3.3.2), events in which multiple players take part (Section 6.3.3.3) and the support for location hierarchies and non-exclusive conditions (Section 6.3.3.4).

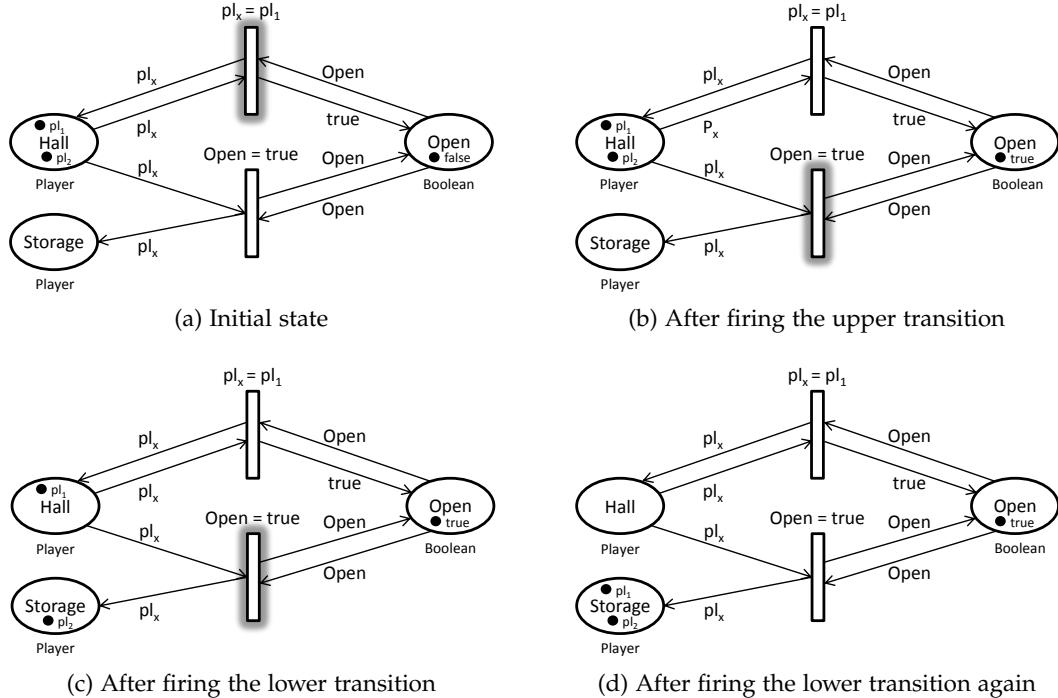


Figure 21: A full example net with two rooms, two events and one variable.

6.3.3.1 Explicit Game Ending

First of all, a special “end”-place (p_{End}) can be added as an additional location, resulting in a new definition for P : $P = P_L \cup P_V \cup \{p_{End}\}$. The place is colored for the players and initialized empty:

$$C(p) = \begin{cases} c_{Player} & \text{if } p \in P_L \vee p = p_{End}; \\ c_{typeOf(variable(p))} & \text{if } p \in P_V. \end{cases}$$

$$I(p) = \begin{cases} P_l & \text{if } p \in P_L \wedge location(p) = l^I; \\ \emptyset & \text{if } (p \in P_L \wedge location(p) \neq l^I) \vee p = p_{End}; \\ value(variable(p), s^I) & \text{if } p \in P_V. \end{cases}$$

After that, a token can be moved into this location to explicitly state when an ending has been reached. The easiest way to do so while allowing the definition of arbitrary ending conditions for the game is to introduce an “end game” reaction for events. Anytime a player triggers an event containing this reaction, its token can easily be moved to this special place instead of a normal location ($nextLocation(lev) = End$). Without this place the ending conditions would have to be coupled to existing locations by tagging them.

A designated ending place offers another benefit. Adding inhibitor arcs from this place to all transitions prevents them from firing once a token has reached the ending (Figure 22). This forces the petri net into a static state, which is not explored further during the simulation. Since the behavior of the game is irrelevant once an ending has been reached, this does not lead to invalid results also saving the efforts that

would be required for further simulation steps. This extends the definition of arcs, N and E as follows:

$$\text{arcs}(\text{lev}) = \{a_{\text{PlayerIn}}(\text{lev}), a_{\text{PlayerOut}}(\text{lev}), a_{\text{Inhib}}(\text{lev})\} \cup \\ \bigcup_{v \in \text{usedVars}(\text{lev})} \{a_{\text{VarIn}}(\text{lev}, v), a_{\text{VarOut}}(\text{lev}, v)\}$$

$$N(a_i(\text{lev})) = \begin{cases} (p_{\text{Src}}, t(\text{lev})) & \text{if } a_i(\text{lev}) = a_{\text{PlayerIn}}(\text{lev}) \wedge \\ & \text{location}(p_{\text{Src}}) = \text{location}(\text{lev}); \\ (t(\text{lev}), p_{\text{Dst}}) & \text{if } a_i(\text{lev}) = a_{\text{PlayerOut}}(\text{lev}) \wedge \\ & \text{location}(p_{\text{Dst}}) = \text{nextLocation}(\text{lev}); \\ (p_{\text{Src}}, t(\text{lev})) & \text{if } a_i(\text{lev}) = a_{\text{VarIn}}(\text{lev}, v) \wedge \\ & \text{variable}(p_{\text{Src}}) = v; \\ (t(\text{lev}), p_{\text{Dst}}) & \text{if } a_i(\text{lev}) = a_{\text{VarOut}}(\text{lev}, v) \wedge \\ & \text{variable}(p_{\text{Dst}}) = v; \\ (t(\text{lev}), p_{\text{Ending}}) & \text{if } a_i(\text{lev}) = a_{\text{Inhib}}(\text{lev}). \end{cases}$$

$$E(a(\text{lev})) = \begin{cases} \text{triggeringPlayerID} & \text{if } a(\text{lev}) = a_{\text{PlayerIn}}(\text{lev}); \\ \text{triggeringPlayerID} & \text{if } a(\text{lev}) = a_{\text{PlayerOut}}(\text{lev}); \\ v & \text{if } a(\text{lev}) = a_{\text{VarIn}}(\text{lev}, v); \\ \text{newValue}(\text{lev}, v) & \text{if } a(\text{lev}) = a_{\text{VarOut}}(\text{lev}, v); \\ \emptyset & \text{if } a(\text{lev}) = a_{\text{Inhib}}(\text{lev}). \end{cases}$$

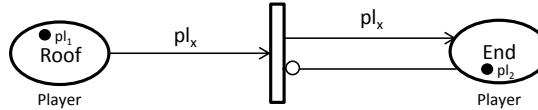


Figure 22: Partial example net with a dedicated ending and an inhibitor arc. The second player (pl_2) has just triggered the ending, so his token has been moved to the *End* place. The inhibitor arc (marked by a small circle) now prevents the transition from firing again.

6.3.3.2 Tying Events to Objects

Another extension allows the developer to tie events not to locations, but to the objects that exist in these locations. This matches their intention more closely, as one would rather say “I’m using the telephone at the office to call someone” than “I’m using the office to call someone”. Doing this has no immediate impact on the definitions provided above, as the events can still be mapped to the locations indirectly. However, an addition is needed if those objects have changeable properties that allow the game’s creator to disable them (for example if an object is spent on use). If such an object is disabled, the event attached to it must also become unavailable. Therefore, the state of the object must be modeled by adding an implicit variable signaling its availability. After that, the condition that this variable is true has to be added to all events attached to the object. Changes to the object’s availability property also need to be translated into a write operation on this variable.

6.3.3.3 Events for Multiple Players

Sometimes it can be interesting to have conditions that do not only check which player is triggering an event, but also which of the other players are present at the same location. This can make it easier to design coordinated actions, for example one player boosting another one to a higher location. Without such a check an additional helper variable would have to be updated manually whenever a player enters or exits this location. Doing so may require lots of effort, especially if there are many connections to other locations. If this check happens in a condition without “or” operators the petri net model requires only two additional arcs, from the location to the transition and vice versa. Both of these arcs are annotated with an identifier other than the one for the triggering player, which is then also used in the transition’s guard condition (Figure 23, right transition). For that, the petri net definitions need to be changed as follows:

$$\begin{aligned}
 \text{arcs}(\text{lev}) = & \left\{ \text{a}_{\text{PlayerIn}}(\text{lev}), \text{a}_{\text{PlayerOut}}(\text{lev}), \text{a}_{\text{Inhib}}(\text{lev}) \right\} \cup \\
 & \bigcup_{v \in \text{usedVars}(\text{lev})} \{\text{a}_{\text{VarIn}}(\text{lev}, v), \text{a}_{\text{VarOut}}(\text{lev}, v)\} \cup \\
 & \bigcup_{pl \in \text{readPVars}(\text{lev})} \{\text{a}_{\text{PVarIn}}(\text{lev}, pl), \text{a}_{\text{PVarOut}}(\text{lev}, pl)\} \\
 N(a_i(\text{lev})) = & \begin{cases} (\text{p}_{\text{Src}}, t(\text{lev})) & \text{if } a_i(\text{lev}) = \text{a}_{\text{PlayerIn}}(\text{lev}) \wedge \\ & \text{location}(\text{p}_{\text{Src}}) = \text{location}(\text{lev}); \\ (\text{t}(\text{lev}), \text{p}_{\text{Dst}}) & \text{if } a_i(\text{lev}) = \text{a}_{\text{PlayerOut}}(\text{lev}) \wedge \\ & \text{location}(\text{p}_{\text{Dst}}) = \text{nextLocation}(\text{lev}); \\ (\text{p}_{\text{Src}}, t(\text{lev})) & \text{if } a_i(\text{lev}) = \text{a}_{\text{VarIn}}(\text{lev}, v) \wedge \text{variable}(\text{p}_{\text{Src}}) = v; \\ (\text{t}(\text{lev}), \text{p}_{\text{Dst}}) & \text{if } a_i(\text{lev}) = \text{a}_{\text{VarOut}}(\text{lev}, v) \wedge \text{variable}(\text{p}_{\text{Dst}}) = v; \\ (\text{p}_{\text{Src}}, t(\text{lev})) & \text{if } a_i(\text{lev}) = \text{a}_{\text{PVarIn}}(\text{lev}, pl) \wedge \\ & \text{location}(\text{p}_{\text{Src}}) = \text{location}(\text{lev}); \\ (\text{t}(\text{lev}), \text{p}_{\text{Dst}}) & \text{if } a_i(\text{lev}) = \text{a}_{\text{PVarOut}}(\text{lev}, pl) \wedge \\ & \text{location}(\text{p}_{\text{Dst}}) = \text{location}(\text{lev}); \\ (\text{t}(\text{lev}), \text{p}_{\text{Ending}}) & \text{if } a_i(\text{lev}) = \text{a}_{\text{Inhib}}(\text{lev}). \end{cases} \\
 E(a_i(\text{lev})) = & \begin{cases} \text{triggeringPlayerID} & \text{if } a_i(\text{lev}) = \text{a}_{\text{PlayerIn}}(\text{lev}); \\ \text{triggeringPlayerID} & \text{if } a_i(\text{lev}) = \text{a}_{\text{PlayerOut}}(\text{lev}); \\ v & \text{if } a_i(\text{lev}) = \text{a}_{\text{VarIn}}(\text{lev}, v); \\ \text{newValue}(\text{lev}, v) & \text{if } a_i(\text{lev}) = \text{a}_{\text{VarOut}}(\text{lev}, v); \\ \text{otherPlayerID} & \text{if } a_i(\text{lev}) = \text{a}_{\text{PVarIn}}(\text{lev}, pl); \\ \text{otherPlayerID} & \text{if } a_i(\text{lev}) = \text{a}_{\text{PVarOut}}(\text{lev}, pl); \\ \emptyset & \text{if } a_i(\text{lev}) = \text{a}_{\text{Inhib}}(\text{lev}). \end{cases}
 \end{aligned}$$

Conditions containing “or” operators cannot easily check the presence of another player, though. The problem can best be described by modifying the example in Figure 23. If a third player is defined to be acrobatic enough to reach the roof on his own, the condition would need to be $pl_x = pl_3 \vee pl_y = pl_2$. Yet, in a situation during which only pl_3 is present at this location, the modified transition could not fire, as

there is no other player token available to bind pl_y ¹. There are two solutions for allowing “or” conditions with “player in scene”-variables. One way is to duplicate the transition and to split its condition at the “or”. After that, one copy has the guard $pl_x = pl_3$ and only one arc for pl_x (Figure 23, left transition). This transition could fire if only pl_3 is present. The other one has the condition $pl_y = pl_2$ and both arcs for pl_x and pl_y (Figure 23, right transition). This models the original case of one player helping the other. If pl_3 and any other player are present, either transition could fire in order to move pl_3 to the roof. But this does not constitute a problem as the result is the same in both cases.

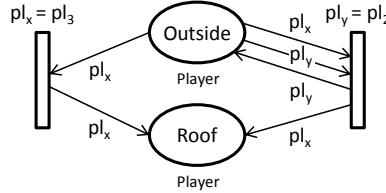


Figure 23: Petri net model checking the presence of another player.

This solution only becomes problematic, if there are more complex conditions with multiple layers of “and” and “or” conditions. The condition must then be transformed until it is in the disjunctive normal form, which is an “or”-clause consisting exclusively of “and”-clauses. After that, one duplicate is created for each “and”-clause – although clauses with the same number of player variables can be combined again. In this case, the number of parts and duplicates can rise quickly. To solve this issue, another model using dummy player tokens is developed. It introduces $|Pl| - 1$ identical dummy players in order to model events in which all players are required (one triggers it):

$$Pl_{\text{Dummy}} = \{pl_{\text{Dummy}}, \dots, pl_{\text{Dummy}}\} \quad |Pl_{\text{Dummy}}| = |Pl| - 1$$

These dummy player tokens are initially placed at each location and can then be used to bind the remaining arcs in case there are no actual players available.

$$I(p) = \begin{cases} Pl \cup Pl_{\text{Dummy}} & \text{if } p \in P_L \wedge \text{location}(p) = l^I; \\ Pl_{\text{Dummy}} & \text{if } p \in P_L \wedge \text{location}(p) \neq l^I; \\ \emptyset & \text{if } p = p_{\text{End}}; \\ \text{value(variable}(p), s^I) & \text{if } p \in P_V. \end{cases}$$

Since they are marked with the dummy value, they can never satisfy a condition requiring a specific player to be present. Additionally, it has to be ensured that these dummies never trigger any events themselves or move around the net, as they can still fulfill conditions that do not expect a specific player to trigger them. This can be prevented by extending every condition with the requirement that the triggering player is not a dummy. $G(t(\text{lev})) = \text{condition}(\text{lev}) \wedge \text{triggeringPlayerID} \neq pl_{\text{Dummy}}$.

¹ For variables, “or” conditions are not problematic since their tokens are always present in the associated place. Therefore, their arcs can always be bound, even if their value does not match the condition.

6.3.3.4 Game Model Shortcuts

Additionally, some shortcuts specific to the game model cannot be represented directly in the petri net and need to be transformed beforehand. For example, location hierarchies must be flattened by copying the objects and events from the parent into the child locations. After that, each location can be handled like a non-hierarchical one.

Non-exclusive event branches are problematic as well, aside from the fact that they contain more paths (compare Figure 24 to Figure 16). Linearizing these events would result in wrong behavior during verification, as can be seen after the first step in Figure 24. If, for example, C_1 and C_3 are true, the condition of both the first and the second line is fulfilled. A petri net simulator could therefore choose between both of these transitions, while the correct behavior as defined by the game model is to only fire the second one. Therefore all conditions must be made exclusive during the path calculation by explicitly adding the inverted conditions of the paths not taken to the conditions (second step in Figure 24).

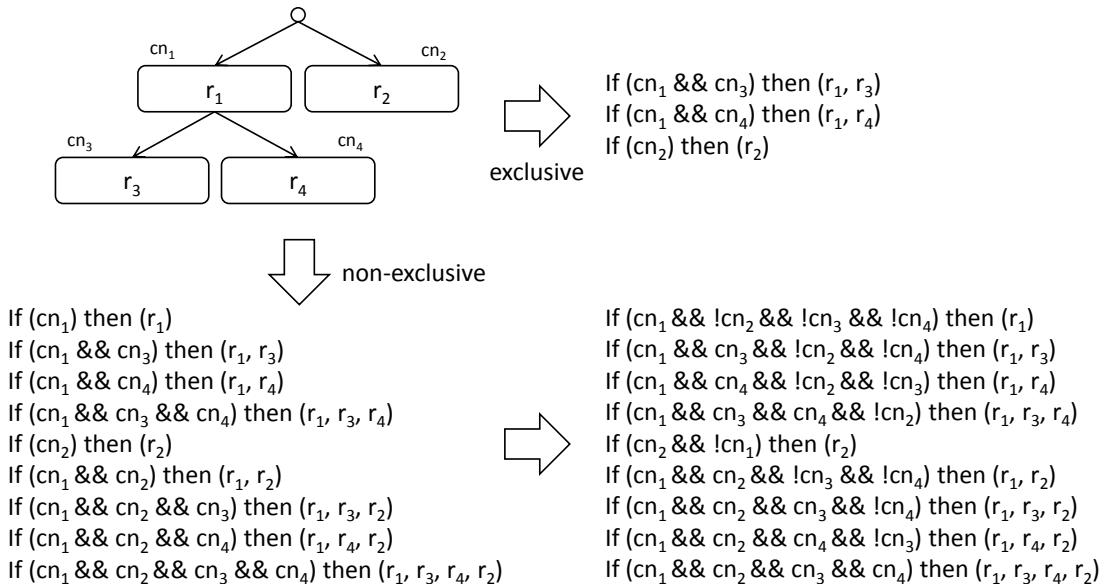


Figure 24: Linearization example with non-exclusive branches. In the game model (Section 4.1), non-exclusivity results in a depth-first execution order for the reactions. Due to the hierarchy, cn_3 and cn_4 are irrelevant if cn_1 is false.

6.3.4 Alternate Model

As mentioned above, an alternative model could treat variables as tokens that move between different places representing their current value. This would make their encoding consistent to the player locations and their value changes more obvious in the visual petri net representation. An example is given in Figure 25, which describes the same game as Figure 21. Instead of having one place for the variable, there is one for each possible value – in this case *true* and *false*. Differentiation between the variables is provided by adding their name as the token value (*Open*). This is necessary since multiple variables now have to share one place if they have the same value. Conditions reading variables are modeled by connecting an arc to the place

of the expected value, annotated with the variable name. In the example, the lower transition requires the variable token *Open* to be in the *true* place. Combined with the fact that player constants are valid arc annotations (the upper transition requires the first player, pl_1), this model does not require guard conditions at all. In the alternate model writing variables is similar to moving players. This is illustrated in Figure 21, in which the upper transition moves *Open* from *false* to *true*.

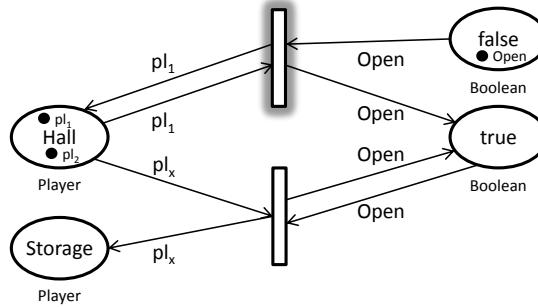


Figure 25: An example net using the alternate model for the same situation as described in Figure 21.

The alternate model, however, has a serious drawback, which is similar to the problems that arise when checking the presence of other players (Section 6.3.3). Since every arc must be enabled by a token, *or*-conditions are problematic using this representation. Again, one solution for this problem is to split the condition at every *or*, which significantly increases the net's complexity. For that, every condition must be brought into the disjunctive normal form first, which only has *or* operators on the top level. Another solution would introduce as many dummy variable tokens as there are variables, of which there are usually a lot more than players. This method also requires the re-introduction of guard conditions to handle dummies differently.

Furthermore, this approach models every variable value as a place, which is only possible when there is a finite number of potential values. This excludes float variables, for example.

Expressing range conditions like $count > 4$ and $count < 7$ becomes more difficult, too. The variable could be in different places that would satisfy such a condition. But there cannot be arcs to each place, as there is no token in most of them that could bind the arc. The only solution for this problem is to transform range conditions into an *or* combination of equality tests, for example $count = 5$ or $count = 6$. Again, this is only possible if the range contains a finite number of known values and the *or* requires the condition to be split. Just setting a variable without reading its value first is the worst case in that regard, since it can have any value in its range for the input arc. This implicates a range condition over all of its possible values. Hence, although it is theoretically possible with finite variable values, this model is not used.

6.4 PROPERTIES AND COMPLEXITY

Certain petri net properties can be exploited during verification in order to reduce the complexity. It is therefore analyzed which of these properties are present at the generated nets.

For example, there can be at most all player tokens in one location-place and exactly one value token in each variable-place. This means that the resulting net is always *n-safe* (*n* being $|Pl|$ or $|Pl| + |Pl_{\text{Dummy}}|$ respectively). It is *not acyclic*, as variable tokens always return to their associated place. Initially, there is only one token with the variable's starting value in each variable-place, which means that the initial markings are *not symmetric*. This would require every value of a color or none at all to be present at each place. Therefore, the net is *not regular* and by extension *not well-formed* – a property that could be used to speed up analysis [27].

The nets are also *not conflict free*. In this case, every place would need to have exactly one outgoing transition (or looping ones) [62]. Players, however, have usually more than one possible action available at some locations, which results in multiple outgoing transitions in this place. Similarly, variables are often used in more than one event and it does not make sense to read a variable in the event which set it. The net is also *not free-choice*, which would imply that if multiple transitions share an input place, then this input place is the only input of these transitions [25]. This is also violated by the variables being relevant for multiple events, as each event also has at least one other place as an input (the location it is attached to).

Based on a game's properties the overall size of the resulting petri net can be estimated. The number of colors is the number of variable types plus the player color.

$$|\Sigma| = 1 + |V_t|$$

It can be assumed to be constant, since there is always a fixed number of variable types supported by the technology used. A typical value for this is five with the usual variable types of *Boolean*, *Integer*, *Float* and *String* in addition to *Player*. The number of places is the sum of the locations and used variables, with the special ending place added. Therefore, this number grows linearly each time a new location or variable is added.

$$|P| = |L| + |V| + 1$$

The number of transitions is dependent on the number of events. However, since events must be linearized into each possible path, usually there is more than one transition for each event.

$$|T| = |LEv| = \sum_{ev \in Ev} |\text{linearizations}(ev)|$$

With exclusive conditions there are as many paths as there are leafs in the event tree, so the worst case is a shallow structure with each reaction at the top level. This results in an upper bound for the paths which is equal to the number of reactions.

$$|\text{linearizations}_{ex}(ev)| = |\text{paths}(ev)| = |\text{leafs}(ev)| \leq |\text{reactions}(ev)|$$

With non-exclusive conditions, all combinations of paths (condition true or false) need to be counted instead.

$$|\text{linearizations}_{nex}(ev)| = 2^{|\text{paths}(ev)|} = 2^{|\text{linearizations}_{ex}(ev)|}$$

Hence, the result of non-exclusivity is a substantially increased number of transitions.

For each of these event transitions there are multiple arcs. This includes incoming and outgoing arcs for the triggering player, for each variable read or written and for each additional player that must be present. The inhibitor arc must also be counted.

$$|\text{arcs}(\text{lev})| = (|\text{usedVars}(\text{lev})| + |\text{readPVars}(\text{lev})| + 1) \cdot 2 + 1$$

Since the number of used variables and players is limited by their overall numbers, an upper bound can be calculated by ignoring the specifics of the event.

$$|\text{arcs}(\text{lev})| \leq (|V| + |Pl| + 1) \cdot 2 + 1$$

The overall number of arcs is then the sum of the arcs for each event.

$$|A| = \sum_{\text{lev} \in LEv} |\text{arcs}(\text{lev})|$$

In order to detect the error classes described in Section 6.1, it must be checked whether at least one ending state is reachable from any other state. This requires a full state space calculation. Regarding the state of the players, there are $|S_{\text{Players}}| = |L|^{|Pl|}$ possibilities to distribute $|Pl|$ players over $|L|$ locations. Each variable can have as many states as it has values, so all variables combined lead to a number of states which is the product of their values.

$$|S_{\text{Variables}}| = \prod_{i=0}^{|V|-1} |\text{values}(\text{typeOf}(v_i))|$$

It is important to note that some variable types have an unlimited number of values (*Float* for example). But there are algorithms which can calculate the state space regardless – for example by using equivalence classes [53]. Nevertheless, the overall number of states is the combination of all player states with all variable states.

$$|S| = |S_{\text{Players}}| \cdot |S_{\text{Variables}}|$$

It is easy to see that this number increases fast when more locations or variables are added (“state space explosion”). Furthermore, finding all enabled transitions for a given state is already NP-hard [67].

6.5 OPTIMIZATION STRATEGIES

Due to this complexity, it is highly beneficial to reduce the size of the resulting net as much as possible before its verification. Doing so without loosing relevant information requires knowledge about the application scenario that cannot be found in general petri net tools.

The general idea is to iterate over all game elements and to decide quickly whether this individual element can be simplified or even completely removed. A first pass is used to gather relevant dependencies between objects, for example a list of events that change a given variable. During subsequent passes these lists can be used to efficiently check for dependencies. This approach looks at game element specifications in general and not at their concrete properties during an actual game state. Hence, it can be described as a static perspective. Doing more complex checks, for example taking the dynamics (the order of events) into account, would be as complex as

the actual verification itself. This would only shift effort from the verification to the optimization step, so this is not an appropriate strategy.

After collecting the references the first actual optimization step is an iterative simplification of the game. During this step different actions are taken, depending on which element is encountered. Events are simplified by removing all of their reactions that do not actually change the game state, for example playing a sound. This decision should be based on a black- or whitelist for reaction types that has been generated in advance, instead of calculating the effect of each individual element. Removing reactions from an event tree is also quite easy. For this, all child reactions are moved up in the hierarchy while their conditions are combined with the parent one (Figure 26).

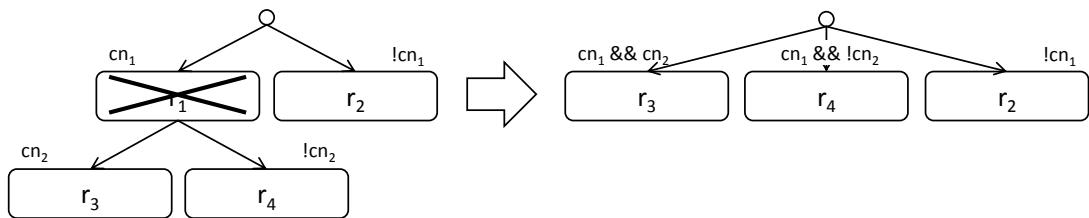


Figure 26: Event reorganization after removing a reaction.

Furthermore, reactions as well as their children can already be removed at this stage if their condition can never be true. This can be the case when their target value is never reached. But since this is one of the error types that should be detected, an error message is displayed to the user whenever this occurs. In games that only set variables to constant values ($a = 4$), instead of calculating them based on their current value ($a = a + 1$), this can easily be checked by looking at its write operations. Logical contradictions in a condition, for example $a < 5 \wedge a > 7$, can also be detected. It is important to note that having impossible conditions is not necessarily a user error. This can also happen during the event linearization when the conditions of different path combinations are merged automatically. Contrarily, conditions which are always true (like $a < 5 \vee a > 4$) can be removed altogether. Removing conditions also removes their references to the variables which they read previously. Variables with no more references are never used and can be removed as well. Similarly, variables which are never written can be replaced with constants and the conditions which reference them are simplified accordingly. Lastly, objects without an event attached to them – for example if they only provide visual effects – can be safely removed. Depending on the game, these optimization actions can yield great savings. In Figure 27 the event could be simplified substantially, which would also remove all of its references to the variable *sound*.

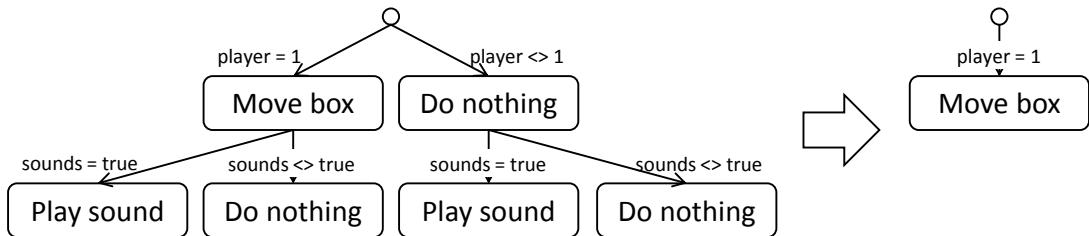


Figure 27: Event optimization example.

These optimization actions must be made iteratively in order to take care of dependent elements. If the variable *sound* is not read outside of the example event, it can safely be removed after the event was simplified. This in turn means that all reactions writing the variable are not needed anymore and can be deleted as well. If there is a button which only toggles this variable, it has now no more relevant reaction attached and can be removed, too. This in turn would render all reactions that change a property of this button irrelevant. Therefore, this step should be repeated until no more elements can be simplified. It must be noted that this approach only detects simple cases, cyclic dependencies for example can never be removed. Due to that fact, scenarios such as two variables influencing one another but having no further impact on the game, cannot be removed. As already mentioned, this would require not only a verification of the resulting petri net, but a second one during verification as well.

Other optimizations do not have dependencies with other objects and therefore only need one pass. Sometimes conditions can be simplified on their own, for example $a < 5 \wedge a < 4$ (which can also be caused by condition merging). Unconnected locations can also be discarded easily when maintaining a list of connections. Since this is an error that should be detected, a message for the user should be displayed when doing so. However, this is not always a mistake – especially when a location hierarchy is flattened and some parent locations only serve as object containers.

A more strict approach for detecting which objects and events are necessary in order to reach the ending would be able to remove substantially more elements from the game. It would, however, not work in relation to the overall verification goal, as optional actions leading to deadlocks would be ignored this way.

Separating the game into independent subsections has the potential to reduce the state space substantially. For example, there are many games that consist of a series of puzzles that must be solved in order to proceed. Often there are multiple ways to solve each puzzle, varying from slight changes like pressing buttons in a different order to completely different approaches. Typically, each of these solutions consists of a number of events that must be triggered. Because the verification has to check every combination of individual solutions between the puzzles, this results in a combinatorial explosion. For example, a small game consisting of five puzzles that must be solved in a specific order can be considered. If each of these puzzles has four solutions there are already $4^5 = 1024$ different combinations that must be tested. However, if it is only relevant that each puzzle is solved and not how the players do that, the puzzles could be verified independently first. Afterwards each one can be substituted with a trivial action (Figure 28) for a final verification of the overall structure. This removes the combinatorial explosion, in our example reducing the number of verifications to $4 \cdot 5 = 20$.

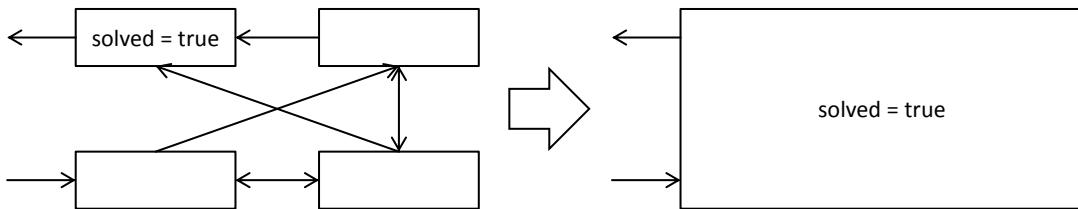


Figure 28: Game partitioning example, in which a puzzle spanning four locations is replaced with a trivial “solve puzzle”-action.

This separation could be done by a developer who is absolutely sure that the sections are truly independent. However, this could hide actual errors if this assumption is false. As such, an algorithm or heuristic for automated separation would be highly beneficial. Formally speaking, these independent sections are sequences of events. All events of a section need to be executable in an uninterrupted sequence, for example by being rooted in the same location or by containing events that trigger the necessary location changes. In the following, all variables that are read or written exclusively during this event sequence will be called local variables. Variables that are also used by other events are called global variables. Once a section has been entered by any combination of players, it must always produce the same result in order to be substitutable. This means that the player(s) must end up in the same location and that all global variables have the same value afterwards, regardless of the overall game state. A simple heuristic that could guarantee this is to forbid all internal events to read global variables. Not writing global variables would make the section truly independent, but it would also prevent other game elements from checking whether this section has been completed. Hence, writing events must be possible. However, since the sequence is to be replaced with a single (atomic) event, all global variables should only be written once and with a value that is always the same for all possible paths through the section. Overwriting the value multiple times could mask side effects and would violate atomicity, even if the end result of this section is always the same value. In Figure 29 for example, the upper events cannot be merged as the lower one is only available for a short time between them. It is obvious that calculating all possible paths through the events in each candidate section again requires a more complex analysis than fits into the scope of a simple optimization without deep verification. Therefore, this approach is not pursued any further. Instead, there are existing methods for directly separating the resulting petri net [65].

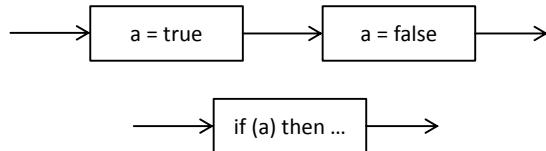


Figure 29: Example in which game partitioning could have unintended side effects.

All of these optimization steps do not change the result of the verification, as they only remove or simplify aspects unrelated to structural errors. Aside from combining conditions, they also do not impact the developer's reverse mapping of petri net elements to game elements when a problem is discovered.

As a more practical optimization, the transformation process itself can be sped up by using multithreading. Especially the linearization of events, which is exceptionally complex when the exclusivity of events must be guaranteed, and the flattening of location hierarchies can be done independently for each element.

6.6 GAME GENERATOR

In order to investigate how well the verification approach scales with different game sizes, a generator algorithm is conceptualized. Supplementing the verification of ex-

isting games, this allows the investigation of the approach's scaling properties without having to manually create a large number of games with different structures. The generation algorithm is able to produce a wide range of game sizes and complexities in regards to the number of locations, events and alternative routes. The complexity of the events should also be variable, as this impacts the number of connections in the resulting petri net. In order to fully explore the state space the resulting games must also be solvable, as deadlocks make some states inaccessible and cause the verification to terminate early.

First of all, the algorithm generates the game's static structure, which consists of the locations and the connections between them. Here, the first relevant parameter is the number of locations the players must traverse in order to reach the ending. These locations are then connected in a linear fashion, which also constitutes the shortest path to the goal. Afterwards, additional branches can be generated that contain optional locations that the players can visit apart from the main path. While the branching factor dictates how many alternative branches are added to each step of the main path, the branching length defines how many events can be executed until each branch is explored completely (Figure 30). Since these branches never loop back on the critical path they are always optional and cannot decrease the shortest path to the final goal. In order to keep the game from becoming unsolvable the players must be able to return to the main path, for which back transitions are added.

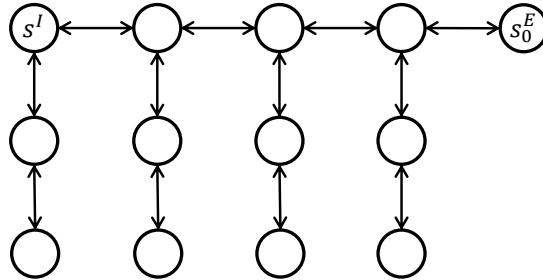


Figure 30: Generated game structure with a path length of four, a branching factor of two and a branching length of two.

The dynamics of the game, i.e. the events located at those locations, must also be generated. The structure and complexity of these events, which is mainly governed by the number of variables written and read, can also be configured. For the events three predefined templates based on typical game tasks are implemented. The simplest option allows players to move to another location without any precondition, so there are no variables involved. The second version adds doors that must be opened by switches, which requires two events and one variable per location. Lastly, the most complex option contains a complex puzzle with five buttons toggling the state of four variables (Table 2). Each button changes multiple variables and the door only opens if all variables are true, which must be achieved by pressing the correct buttons. This results in $2^5 = 32$ possible combinations of the buttons being pressed, of which only one is correct. The events associated with the buttons consist of 28 paths altogether (2 buttons toggle 3 variables: $2 \cdot 2^3$; 3 buttons with 2 variables: $3 \cdot 2^2$). After having been selected, the given task is attached to each transition towards the goal and to the end of each branch. The back transitions do not have a specific task type, which means

that there is always a way back to guarantee solvability. It is important to note that the puzzles are simply cloned and therefore share the same solution. But contrary to actual players the verification has to consider every combination regardless and is therefore unable to exploit this fact.

Switch	Light				Solution
	1	2	3	4	
1	X	-	X	X	Use once
2	X	-	-	X	Used once
3	-	-	X	X	Unused
4	-	X	X	-	Unused
5	X	X	-	X	Used once

Table 2: Structure of the complex puzzle used for game generation.

6.7 SUMMARY

This chapter presents an approach for automatically detecting structural errors in a game, especially ones that prevent the players from reaching the game's ending. To this end a formalized game model based on petri nets accompanied by clear transformation rules for all relevant elements is developed. These rules allow the verification of every single element of a game that fits this model. All error classes that are to be detected are then mapped to states of the petri net as follows:

- **Deadlocks** are states in which there are no more enabled transitions, but the game ending has not yet been reached.
- **Livelocks** are states in which there are enabled transitions, but in which no state that constitutes a game ending can be reached.
- **Unreachable locations** are location-places that never contain a player token.
- **Impossible events** are transitions that are never triggered.

Additionally, the transformation of games from the application scenario described in Section 3.1 into this model can be done algorithmically. This ensures consistency between the model and the game and requires less effort and modeling knowledge.

In order to keep a complete verification viable, the resulting nets should be as small as possible. Therefore, multiple optimization steps are introduced, which use knowledge about the game's semantic to remove or simplify some elements of the model.

Although focused on the application scenario, the model could also be used as a higher level model for more complex games. For instance, in a game with continuous movement the locations could be discretized into equivalence classes. However, then the advantage of every game element being verified would be lost.

In the context of the overall authoring process, the verification can guarantee that the game does not contain errors that prevent the game from running correctly. However, it does not further assess the quality of different paths through the game, especially in regards to balancing. For this, the component described in the next chapter is needed.

COLLABORATIVE BALANCING IN MULTIPLAYER GAMES

ASIDE from ensuring that a game contains no structural errors, developers should also balance their games. For this, we propose a novel balancing definition that focuses specifically on collaborative settings by balancing the players' contributions to their shared goals [95]. Based on this definition, we also provide concrete measurements that allow developers to assess whether their game is balanced or not.

7.1 COLLABORATIVE BALANCING DEFINITION

Providing balance between the players that play together collaboratively is a new field, as existing balancing definitions focus on a competitive view (Section 2.5). Therefore an own definition for balancing in a collaborative setting has to be provided first, which can then be used to evaluate whether a given game is balanced or not.

An informal survey among colleagues and students has revealed that players care about the effort they provide towards the group's goal. Some of them consider a game to be unbalanced when they have to "do all the work," which can mean both the amount of actions they have to execute as well as the (perceived) difficulty of these actions. Such situations match the definition of "free-riding" found in the related work (Section 2.5). In contrast to this negative (goal oriented) view on effort a positive perspective exists, too. Here, doing something is better than doing nothing. This can be explained by the fact that playing a game is generally enjoyable – not playing can therefore provide not as much fun. Additionally, in the context of story-based or learning games, players who are not participating could miss parts of the story or the learning content. Independent of whether effort is seen as positive or negative, the players agreed that the effort they contribute should be similar to their peers.

Players also do not like to wait for other players, for example when the others have to provide a prerequisite to the next action they want to take themselves. Waiting is mostly seen as negative because players want to play and act themselves in games. Although sometimes pauses might be welcome, players usually prefer to take them onto their own account. The advice to prevent unintended waiting times is also found in the related work (Section 2.2.1).

Last but not least, the related work about competitive balancing states that the players should have a number of equally viable and interesting options to choose from (Section 2.5), for example in regards to their strategy. It seems fitting to apply this to collaborative games as well, in which the options of each player should be balanced, too. In our game model (Section 4.1) options are actions the players are able to execute at any given time. These actions are relevant if they contribute to the overall goal of the group and are required to reach an ending. Choosing between multiple alternative actions is not only more interesting than repeating a single action over and over again. It also allows personalization when the players are able to play

the game the way they like, for example following their own moral compass for story-relevant decisions. In a collaborative setting this can spark discussions among the group members, which can provide additional enjoyment. Irrelevant actions, in contrast, do not contribute to the overall goal. They can be seen positively, when they allow the players to do something while waiting for others. Or they could just confuse the players, if there are too many options and it is not clear which ones are relevant.

Because effort and options can be seen positively or negatively, developers cannot generally be advised to minimize or maximize these factors. Even with waiting, which is mostly negative, there is a significant variance what different players would accept in different game types. For example, in puzzle games waiting players could think about their colleague's problems, too. This would speed up the overall progress of the group while giving them something to pass the time – which can cause them to not notice that they are in fact waiting at all. Therefore there are no optimal absolute values for provided effort, waiting time and options that should be generally pursued. Instead, the game should try to keep that factor as equal as possible among the players in order to be collaboratively balanced.

However, balancing the overall game is not enough. A two-player game in which one player has to provide all the effort in the first half, while the other one waits with this relationship being switched in the second half, would be balanced overall. But each half would feel unbalanced, therefore an equal distribution of effort, waiting times and options has to be provided in each section of a game.

We therefore propose the following definition for collaborative balancing [95]:

A collaborative or cooperative game is considered balanced among group members when the individual efforts and waiting times required to solve the game as well as the number of options available along the way are equally high for each player and uniformly distributed throughout the game.

It has to be noted that this definition is only focused on the relationship between players playing collaboratively. Additionally, the game has to be balanced "competitively" for the player group as a whole. For this, the existing definitions and guidelines can be applied by taking the group as a collective and viewing it like a single player. At this point the difficulty, for example, can be balanced as it would be done for a singleplayer game.

7.2 APPROXIMATING BALANCE

In order to balance a game the balance must be measured first. For this the same formalization as for the structural verification (Section 6.2) after event linearization (Section 6.3.2) is used. Players (P_l), game states (S), one initial game state (s^I), several ending states (S^E) and linearized events (LEv) are used in the same way.

Additionally, several auxiliary functions are defined. A linearized event is available when its condition is satisfied by the current game state with at least one player present at its location: $\text{available}(lev, s) \Leftrightarrow \exists pl \in P_l : \text{eval}(\text{condition}(lev), pl, s) = \text{true} \wedge \text{location}(pl, s) = \text{location}(lev)$. This allows the retrieval of all events which are available at a given state: $\text{avail}_{LEv}(s) = \{lev \in LEv | \text{available}(lev, s)\}$. As there could be multiple players present which could satisfy the event's condition there is

also a function for checking which players could trigger a given event at a given state: $\text{avail}_{\text{Pl}}(\text{lev}, \text{s}) = \{\text{pl} \in \text{Pl} | \text{eval}(\text{condition}(\text{lev}), \text{pl}, \text{s}) = \text{true} \wedge \text{location}(\text{pl}, \text{s}) = \text{location}(\text{lev})\}$.

It should also be modeled that executing the player actions to trigger an event at a given state can vary in difficulty. This can be modeled by a weight ($\text{weight}(\text{lev}, \text{s})$), which is 1 by default and which can be adjusted using the developer's estimation or a heuristic evaluation of the event and its location.

Based on this model, a general approach for approximating the balance can be defined. First, the quickest way to reach each ending should be calculated. This is useful because games usually contain optional actions that can be repeated, which means that the longest and therefore also the average path is infinite. Also in non-linear games there can be numerous choices for the players, which result in different paths being taken. To avoid the creation of many game elements, which are only reached by a small number of players, some of these paths usually merge again later on. This increases the number of paths to a given ending exponentially.

Whether an action is relevant for quickly reaching an ending is directly visible from the shortest path and must not be marked beforehand. For the players as a group, an event is mandatory if it is executed along this path. But from the perspective of a single player, an event is mandatory if it has to be triggered by this particular player. If the other players could also trigger this event on the respective path, then no player is forced to execute it and it is optional from everyone's personal perspective.

After that, separate balancing metrics for each player can be calculated on every path, mapping concrete game elements to the more abstract concepts of effort, waiting time and options. If the game is perfectly balanced, the metrics should have the same value for each player and on every path. But since there can be a large number of endings and the developer should not be forced to compare every combination manually, this balance must be aggregated into a single value describing the overall amount of (in-)equality in the game. Yet, perfect balance is not always possible or even necessary, so the developer could also accept slight inequalities like a deviation of 5% between the players.

Since only the shortest path to each ending is calculated, this value is only an approximation. An exact calculation would not only require the analysis of every (possibly infinite) path, but also the probabilities with which each path is chosen. This probability depends on the actual players and would need to be estimated at the design stage, too.

Lastly, the actual difficulty of each action can only be estimated and varies depending on the player's skill and prior knowledge. Because of that it is impossible to accurately calculate a general balancing value and an approximation has to be provided instead.

7.2.1 Path sampling

If a structural verification (Chapter 6) has been done beforehand, the state space of the game is known. This graph of state nodes connected by event edges can be used as an input for standard pathfinding algorithms like Dijkstra or A*. Alternatively, the events necessary to reach an end state can be calculated directly on the game

using planning algorithms, which has been explored together with a supervised student [46].

Usually there are multiple paths from the initial state to each end state: $\text{paths}(s^E) = \{p_0, \dots, p_{|\text{paths}(s^E)|-1}\}$. These paths are an ordered sequence which alternates between states and player-triggered events:

$$p = (s_0, (\text{lev}_0, \text{pl}_0), s_1, \dots, (\text{lev}_{\text{length}(p)-1}, \text{pl}_{\text{length}(p)-1}), s_{\text{length}(p)})$$

The length of the path ($\text{length}(p)$) is defined to be the number of states. In order to be valid, each path has to start with the initial state ($s_0(p) = s^I$) and has to end at an end state ($s_{\text{length}(p)}(p) = s^E$). The states in between are not allowed to be end states or the initial state ($\forall i \in [1, \text{length}(p) - 1] : s_i(p) \neq s^I \wedge s_i(p) \notin S^E$), otherwise the path would not be minimal. The path must also be possible in the game, which means that at a given state s_i the next event ev_i must be available and the next state after its execution next_S must be the next one in the path s_{i+1} : $\forall i \in [0, \text{length}(p) - 1] : \text{lev}_i(p) \in \text{avail}_{LEv}(s_i(p)) \wedge s_{i+1}(p) = \text{next}_S(\text{lev}_i(p), s_i(p), \text{pl}_i(p))$.

To simplify the formulas, a few auxiliary functions are introduced. One function filters the events from a path ($\text{events}(p) = (\text{lev}_0, \dots, \text{lev}_{\text{length}(p)-1})$), with $\text{lev}_i(p)$ being the i -th event of $\text{events}(p)$. The same is done for the states ($\text{states}(p) = (s_0, \dots, s_{\text{length}(p)})$) and $s_i(p)$ being the i -th state of $\text{states}(p)$. Sometimes multiple players are able to trigger an event, which impacts balancing. Therefore, $\text{players}(p)$ does not show which of the players executes an event on a given path, but rather which players could have done it at this state in order to reach the same result (due to linearization):

$$\text{players}(p) = (\text{avail}_{p1}(\text{lev}_0(p), s_0(p)), \dots, \text{avail}_{p1}(\text{lev}_{\text{length}(p)-1}(p), s_{\text{length}(p)-1}(p)))$$

Again $\text{pl}_i(p)$ is used to address the i -th element of $\text{players}(p)$.

As mentioned before, every event on the path has a weight that describes its difficulty:

$$\text{weights}(p) = (\text{weight}(\text{lev}_0(p), s_0(p)), \dots, \text{weight}(\text{lev}_{\text{length}(p)-1}(p), s_{\text{length}(p)-1}(p)))$$

As before, $\text{weight}_i(p)$ is used to address the i -th element of $\text{weights}(p)$. Weights are not allowed to be negative, as no event should be able to “undo” the weight of a previous one.

As already mentioned, three separate balancing metrics should be approximated: effort, waiting times and options ($M \in E, W, O$). Since these metrics look at different aspects of the game and the shortest path should always minimize the current metric, they can yield different shortest paths. For that, the general $\text{weight}(\text{lev}, s)$ of events has to be overwritten for each metric to $\text{weight}^M(\text{lev}, s)$. To shorten this term $\text{weight}^M(p, i) = \text{weight}^M(\text{lev}_i(p), s_i(p))$ can be used. Additionally, each player might weight an event differently, for example if he does not have to trigger it, for which a player-specific weight $\text{weight}^M(p, i, pl)$ is introduced. It is assumed that this individual perspective is never higher than the player-independent view:

$$\forall i : \text{weight}^M(p, i) \geq \text{weight}^M(p, i, pl) \tag{1}$$

Auxiliary functions similar to $\text{weights}(p)$ and $\text{weight}_i(p)$ can be defined for the metric-specific weights:

$$\begin{aligned}\text{weights}^M(p) &= (\text{weight}^M(p, 0), \dots, \\ &\quad \text{weight}^M(p, \text{length}(p) - 1)) \\ \text{weights}^M(p, pl) &= (\text{weight}^M(p, 0, pl), \dots, \\ &\quad \text{weight}^M(p, \text{length}(p) - 1, pl))\end{aligned}$$

Again, $\text{weight}_i^M(p)$ and $\text{weight}_i^M(p, pl)$ are used for the i -th element of the corresponding set. The minimal path to the end state s^E , based on the metric M , is then:

$$\text{minPath}^M(s^E) = \operatorname{argmin}_{p \in \text{paths}(s^E)} \sum_{i=0}^{\text{length}(p)-1} \text{weight}_i^M(p)$$

As mentioned, the minimal path from each player's perspective must also be calculated. Since pathfinding algorithms can only minimize a single metric at a time this must happen in additional calculations using the player-specific metrics:

$$\text{minPath}^M(pl, s^E) = \operatorname{argmin}_{p \in \text{paths}(s^E)} \sum_{i=0}^{\text{length}(p)-1} \text{weight}_i^M(p, pl)$$

To keep formulas short the following abbreviations for directly accessing the shortest path have been defined:

$$\begin{aligned}\overline{\text{length}}^M(s^E) &= \text{length}(\text{minPath}^M(s^E)) \\ \overline{\text{length}}^M(pl, s^E) &= \text{length}(\text{minPath}^M(pl, s^E)) \\ \overline{\text{weight}}_i^M(s^E) &= \text{weight}_i^M(\text{minPath}^M(s^E)) \\ \overline{\text{weight}}_i^M(pl, s^E) &= \text{weight}_i^M(\text{minPath}^M(pl, s^E), pl)\end{aligned}$$

However, there is a potential pitfall when implementing pathfinding naively. Intuitively a metric can be zero, for example when a player does not have to provide any effort to an event triggered by another player. This would signal the pathfinding that this event is free and it will always be taken first. Therefore, in a scenario like Figure 31a a large loop of "free" events is explored first before the player's own action (value of one) is eventually executed. It is easy to see that for a real game these unnecessary loops can get much more complex and more numerous, greatly increasing the runtime of the pathfinding. Additionally, if the A* algorithm is used, an admissible heuristic that does not underestimate the remaining distance to the goal is required. Using the number of events to reach a goal without calculating their actual weight could be a fitting heuristic, since it is easy to understand and can be calculated in a reasonable amount of time using a breath first traversal (once per goal state). This requires the weight of each event being at least 1 to keep the heuristic from overestimating the real costs. Adding 1 to each weight could fix both of these problems. But it becomes problematic in the scenario displayed in Figure 31b: Now, the cost of an event with the actual weight of one is increased to 2, so it has the same weight as two "irrelevant" events. For this reason, the right path around the action by pl_0 is aborted early after two actions. Instead of exploring this path further and reaching

the ending without any more actual costs for this player, the left path is taken and a wrong result is returned (two actions by pl_0 instead of one).

This means that an action by the focused player must cost significantly more in relation to the not important ones. The increase of the weights cannot be too large, however. In the extreme case of an infinite value, the action would only be taken if there are no other paths left – resulting in the same problem as in the first scenario. Therefore, it is suggested to multiply the original value by a penalty factor, if it is relevant for the minimized player, and then add one to all weights in general. Then, the heuristic is still admissible and the pitfalls are governed by a trade-off between runtime and accuracy. The higher the penalty factor, the more irrelevant events are explored first, increasing the runtime (Figure 31a). But on the other hand, a lower value means a higher chance that the wrong result is returned (Figure 31b). Based on our observations, we suggest a penalty factor of 20. This explores other player's events up to a depth of 20, which is feasible in regards to runtime. It also means that a wrong result is returned only if there is a way around a relevant action with 21 actions. We deem this is as acceptable, since the whole calculation is only an approximation anyhow. Naturally, after the paths have been calculated, the unmodified metric should be used for calculating the real costs of this path.

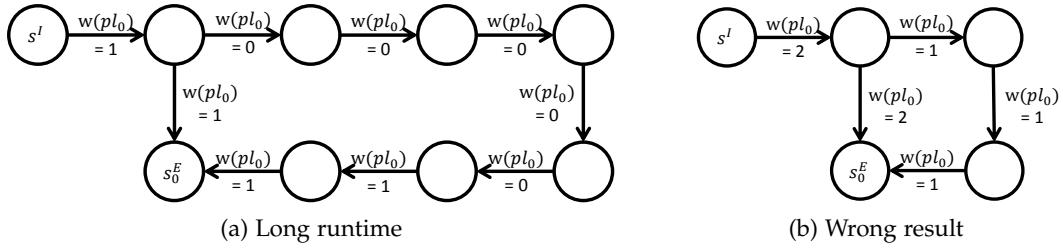


Figure 31: State space examples containing pathfinding pitfalls. The weights from the perspective of pl_0 are noted at each event transition.

The state space graph contains as many vertices as there are states ($|Vertices| = |S|$). The number of edges can be at most the product of linearized events and states, which is $|Edges| \leq |S| \cdot |LEv|$ since an event could be available in multiple states. The Dijkstra and A* algorithms have a complexity of $O(|Vertices|^2) = O(|S|^2)$ or, if implemented with a Fibonacci Heap [39], $O(|Vertices| \cdot \log|Vertices| + |Edges|)$:

$$O(|S| \cdot \log|S| + |S| \cdot |LEv|) = O(|S| \cdot (\log|S| + |LEv|)).$$

There are paths calculated for all combinations of players and end states ($|Pl| \cdot |S^E|$ runs). Its overall complexity is therefore

$$O(|Pl| \cdot |S^E| \cdot |S| \cdot (\log|S| + |LEv|)).$$

This can be simplified into $O(|S^E| \cdot |S| \cdot (\log|S| + |LEv|))$, since $|Pl|$ is a small constant in the application scenario (Section 3.1). The breath first traversal required for the A* heuristic has a complexity of $O(|Edges| + |Vertices|) = O(|S| \cdot |LEv| + |S|)$ per run, which can be simplified to $O(|S| \cdot |LEv|)$ since $|S| \cdot |LEv| > |S|$. It requires one run for each end state $|S^E|$:

$$O(|S^E| \cdot |S| \cdot |LEv|).$$

This results in an overall complexity of

$$O(|S^E| \cdot |S| \cdot (\log|S| + |LEV|)) + O(|S^E| \cdot |S| \cdot |LEV|) = O(|S^E| \cdot |S| \cdot (\log(|S| \cdot |LEV|) + |LEV|)).$$

Because the path calculations for each ending and player only share their inputs but are independent from each other's results, they can be calculated in separate threads to speed up the process in practice.

Figure 32 illustrates an example state space with three endings, based on a game for two players. The players able to trigger an event are noted at the edges, whose weight is assumed to be one for all edges to simplify the example. Furthermore, there is only one path to each ending and there are no optional events, so the shortest path is the same for every metric and perspective. It is important to note that the structure is kept simple for illustration purposes. In real games it would be much larger (longer path) and more complex (more junctions). After the path calculation the following paths are found:

$$\begin{aligned} \forall i \in [0, 2] p_i &= \text{minPath}^M(s_i^E) = \text{minPath}^M(Pl_0, s_i^E) = \text{minPath}^M(Pl_1, s_i^E) \\ \text{states}(p_0) &= (s^I, s_0, s_1, s_2, s_0^E) \\ \text{events}(p_0) &= (lev_0, lev_1, lev_2, lev_3) \\ \text{states}(p_1) &= (s^I, s_0, s_1, s_3, s_1^E) \\ \text{events}(p_1) &= (lev_0, lev_1, lev_4, lev_5) \\ \text{states}(p_2) &= (s^I, s_0, s_4, s_5, s_2^E) \\ \text{events}(p_2) &= (lev_0, lev_6, lev_7, lev_8) \end{aligned} \quad (2)$$

These paths can then be used as input for the next step, calculating the three balancing metrics for each player.

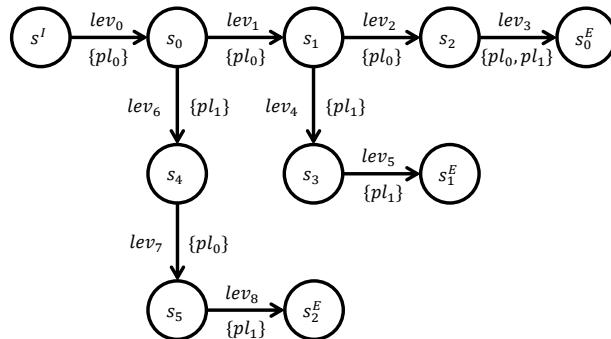


Figure 32: Example state space for balancing. For each event its ID and all players that could trigger it are noted.

7.2.2 Path assessment

Having the shortest path for player pl to an ending s^E following the metric M the overall "length", i.e. its value for the metric can be calculated by summing the metric-specific weights for each event from the perspective of this player:

$$M^{player}(pl, s^E) = \sum_{i=0}^{\text{length}^M(pl, s^E)-1} \text{weight}_i^M(pl, s^E)$$

A pairwise comparison of all players' values would result in $|Pl|^2$ values, which is relatively difficult to interpret. Instead, it is beneficial to relate all individual player values to a common anchor, resulting in only $|Pl|$ values. The minimal metric for the group as a whole, which requires an additional shortest path calculation, can be used as this anchor:

$$M^{overall}(s^E) = \sum_{i=0}^{\overline{length}^M(s^E)-1} \overline{weight}_i^M(s^E)$$

With this a percentage metric for each player can be calculated:

$$M^{player\%}(pl, s^E) = \frac{M^{player}(pl, s^E)}{M^{overall}(s^E)}$$

The shortest path for each player is always shorter or equal to the path for the whole group in regards to the metric used. This can be proven by looking at a minimal path for the whole group $p_{Grp} = \minPath^M(s^E)$. Because of Equation 1, every event on this path has at most the same weight for any player as for the group, which is translated to the overall path length as the sum of the individual weights. Therefore, the individual player values on p_{Grp} are at most the same as the value for the whole group – or there are other, even shorter, paths from the player's perspective. This means that the maximum of the percentage metric is one:

$$\begin{aligned} \forall pl \in Pl, s^E \in S^E : M^{overall}(s^E) &\geq M^{player}(pl, s^E) \\ \Rightarrow \forall pl \in Pl, s^E \in S^E : M^{player\%}(pl, s^E) &\leq 1 \end{aligned}$$

Since the individual weights are always positive, the value range of $M^{player\%}(pl, s^E)$ is $[0, 1]$. This also serves as a normalization of the values, making them comparable between paths that have a different length.

Alternatively, the sum of the individual player values could be used for comparison, saving the additional path calculation:

$$\begin{aligned} M^{sum}(s^E) &= \sum_{i=0}^{|Pl|-1} M^{player}(Pl_i, s^E) \\ M^{player\%'}(pl, s^E) &= \frac{M^{player}(pl, s^E)}{M^{sum}(s^E)} \end{aligned}$$

But this would mask cases in which most of the weights cannot be attributed to a specific player, resulting in low values for all of them. Simply comparing these values to the sum would not show the large difference between their individual values and the one actually required for solving the game.

Another alternative is to calculate the group effort on each player's path instead of a separate path:

$$\begin{aligned} M^{overall'}(pl, s^E) &= \sum_{i=0}^{\overline{length}^M(pl, s^E)-1} \overline{weight}_i^M(s^E) \\ M^{player\%''}(pl, s^E) &= \frac{M^{player}(pl, s^E)}{M^{overall'}(pl, s^E)} \end{aligned}$$

But this yields the problem that path finding algorithms can only optimize a single value. Therefore, no reliable statement about the group effort on a player path can be given. The same problem appears when using a single path, for example the one being optimal for the group, and then calculating the player's individual values on it:

$$M^{player'}(pl, s^E) = \sum_{i=0}^{\overline{length}^M(s^E)-1} weight_i^M(minPath^M(s^E), pl)$$

$$M^{player'''}(pl, s^E) = \frac{M^{player'}(pl, s^E)}{M^{overall}(s^E)}$$

For the example found in Figure 33 both paths are four events long, so their overall length with a uniform weight of one is also four. A pathfinding algorithm optimizing the overall length could therefore return any of the two paths (but always only one). Depending on which one is chosen, either pl_0 or pl_1 would have more weight than the other – which suggests an imbalance. A human looking at the graph would describe the overall construct as balanced, due to its symmetry, however.

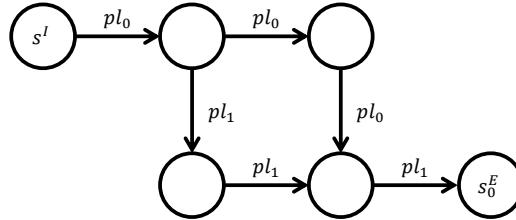


Figure 33: State space example in which different paths are required for each player. Every weight is assumed to be one and is only counted for the player noted at the event's transition. The upper right path weighs more for pl_0 , the lower left path for pl_1 .

Since all of these calculations can be done by summing up event weights during the pathfinding, they do not increase the complexity of the overall approach.

This path assessment approach works with any metric M that defines non-negative individual and overall weights for the events that follow Equation 1. Based on that, the following three sections discuss concrete metrics for effort, waiting times and options as examples.

7.2.2.1 Effort

For measuring the minimal (matching the minimizing nature of the pathfinding) effort for a specific player, only events that this player has to trigger are counted.

$$weight^E(p, i) = weight(lev_i(p), s_i(p))$$

$$weight^E(p, i, pl) = \begin{cases} weight^E(p, i) & \text{if } avail_{pl}(lev_i(p), s_i(p)) = \{pl\}; \\ 0 & \text{else.} \end{cases}$$

Therefore, events that can be triggered by multiple players are not counted for anyone, as nobody has the sole responsibility to do so. This does not violate Equation 1 as an individual player's weight is either the same as the general one or zero.

This results in the following efforts for the example in Figure 32 and the paths from Equation 2:

$$\begin{aligned}
 \text{weights}^E(p_0) &= (1, 1, 1, 1) & E^{\text{overall}}(s_0^E) &= 4 \\
 \text{weights}^E(p_0, pl_0) &= (1, 1, 1, 0) & E^{\text{player}}(pl_0, s_0^E) &= 3 \\
 \text{weights}^E(p_0, pl_1) &= (0, 0, 0, 0) & E^{\text{player}}(pl_1, s_0^E) &= 0 \\
 E^{\text{player}\%}(pl_0, s_0^E) &= 75\% & E^{\text{player}\%}(pl_1, s_0^E) &= 0\%
 \end{aligned}$$

The path above has a big difference between the player's efforts, which means that there is imbalance among the players.

$$\begin{aligned}
 \text{weights}^E(p_1) &= (1, 1, 1, 1) & E^{\text{overall}}(s_1^E) &= 4 & (3) \\
 \text{weights}^E(p_1, pl_0) &= (1, 1, 0, 0) & E^{\text{player}}(pl_0, s_1^E) &= 2 \\
 \text{weights}^E(p_1, pl_1) &= (0, 0, 1, 1) & E^{\text{player}}(pl_1, s_1^E) &= 2 \\
 E^{\text{player}\%}(pl_0, s_1^E) &= 50\% & E^{\text{player}\%}(pl_1, s_1^E) &= 50\%
 \end{aligned}$$

$$\begin{aligned}
 \text{weights}^E(p_2) &= (1, 1, 1, 1) & E^{\text{overall}}(s_2^E) &= 4 \\
 \text{weights}^E(p_2, pl_0) &= (1, 0, 1, 0) & E^{\text{player}}(pl_1, s_2^E) &= 2 \\
 \text{weights}^E(p_2, pl_1) &= (0, 1, 0, 1) & E^{\text{player}}(pl_0, s_2^E) &= 2 \\
 E^{\text{player}\%}(pl_0, s_2^E) &= 50\% & E^{\text{player}\%}(pl_1, s_2^E) &= 50\%
 \end{aligned}$$

On these paths there is no difference between the effort both players provide, the game seems balanced.

7.2.2.2 Waiting Times

Waiting times can be estimated by a weight function that is similar to the one for effort. Every event that a player cannot trigger has to be triggered by someone else while the player is waiting.

$$\begin{aligned}
 \text{weight}^W(p, i) &= \text{weight}(\text{lev}_i(p), s_i(p)) \\
 \text{weight}^W(p, i, pl) &= \begin{cases} \text{weight}^W(p, i) & \text{if } pl \notin \text{avail}_{pl}(\text{lev}_i(p), s_i(p)); \\ 0 & \text{else.} \end{cases}
 \end{aligned}$$

Analogue to the effort, if multiple players can trigger an event it is not counted as waiting time for any of them due to the goal of getting a minimal value. In contrast to effort however, for which an event is counted for at most one player, here multiple players can wait at the same time. Therefore, in a game in which only one player acts, all others are waiting 100% of the time. This difference is crucial enough to calculate waiting times separately from the effort, although there is a correlation between them. The calculation of the waiting time is also an approximation, as it assumes that events are executed as soon as they are available for a player. Decision-making and reaction times are ignored as they are unknown at this stage of development. Furthermore, they vary between different players anyhow. Calculating the actual waiting times also would need information on the logical dependencies between events, especially in regards to which events can be parallelized. If this is the case, a player could trigger

a future event early or could execute some optional actions instead of waiting. Due to the complexity of calculating such dependencies and the approximating nature of the overall approach, this is currently not considered. The weight formula for waiting also guarantees Equation 1 as an individual player's weight is either the same as the general one or zero.

In the example (Figure 32 and Equation 2) the following waiting times are approximated:

$$\begin{array}{ll} \text{weights}^W(p_0) = (1, 1, 1, 1) & W^{\text{overall}}(s_0^E) = 4 \\ \text{weights}^W(p_0, pl_0) = (0, 0, 0, 0) & W^{\text{player}}(pl_0, s_0^E) = 0 \\ \text{weights}^W(p_0, pl_1) = (1, 1, 1, 0) & W^{\text{player}}(pl_1, s_0^E) = 3 \\ W^{\text{player}\%}(pl_0, s_0^E) = 0\% & W^{\text{player}\%}(pl_1, s_0^E) = 75\% \end{array}$$

The path above is also imbalanced in regards to waiting times, as there is a big difference between the players values.

$$\begin{array}{ll} \text{weights}^W(p_1) = (1, 1, 1, 1) & W^{\text{overall}}(s_1^E) = 4 \\ \text{weights}^W(p_1, pl_0) = (0, 0, 1, 1) & W^{\text{player}}(pl_0, s_1^E) = 2 \\ \text{weights}^W(p_1, pl_1) = (1, 1, 0, 0) & W^{\text{player}}(pl_1, s_1^E) = 2 \\ W^{\text{player}\%}(pl_0, s_1^E) = 50\% & W^{\text{player}\%}(pl_1, s_1^E) = 50\% \\ \\ \text{weights}^W(p_2) = (1, 1, 1, 1) & W^{\text{overall}}(s_2^E) = 4 \\ \text{weights}^W(p_2, pl_0) = (0, 1, 0, 1) & W^{\text{player}}(pl_0, s_2^E) = 2 \\ \text{weights}^W(p_2, pl_1) = (1, 0, 1, 0) & W^{\text{player}}(pl_1, s_2^E) = 2 \\ W^{\text{player}\%}(pl_0, s_2^E) = 50\% & W^{\text{player}\%}(pl_1, s_2^E) = 50\% \end{array}$$

Both of these paths feature balanced waiting times.

7.2.2.3 Options

The options at a given state can be defined as the events that can be triggered there. Because an event is a transition between two states, there is no definitive mapping between an event and the options metric. Since the effects of the event are more relevant than its preconditions, the options are counted in its successor state. But one should also penalize if a player has the same options for a longer time, as they can get boring after a while. To recognize this, only options that are not available in the previous state (δ) are counted. The following formulas use two player variables, of which pl_f is the player for which the balancing value is currently calculated. In contrast, pl_t is used to denote the player that has to trigger an event on the path.

$$\begin{aligned} \text{weight}^O(p, i) &= \sum_{lev \in \delta(p, i)} \text{weight}(lev, \text{next}_S(\text{lev}_i(p), s_i(p), pl_i(p))) \\ \text{weight}^O(p, i, pl_f) &= \sum_{lev \in \delta(p, i, pl_f)} \text{weight}(lev, \text{next}_S(\text{lev}_i(p), s_i(p), pl_i(p))) \\ \delta(p, i) &= \delta'(s_i(p), \text{lev}_i(p), pl_i(p)) \\ \delta(p, i, pl_f) &= \delta'(s_i(p), \text{lev}_i(p), pl_i(p), pl_f) \end{aligned}$$

$$\begin{aligned}\delta'(s, lev, pl_t) &= \{\text{lev} \in \text{avail}_{LEV}(\text{next}_S(\text{lev}, s, pl_t)) | (\text{lev} \notin \text{avail}_{LEV}(s))\} \\ \delta'(s, lev, pl_t, pl_f) &= \{\text{lev} \in \text{avail}_{LEV}(\text{next}_S(\text{lev}, s, pl_t)) | \\ &\quad (pl_f \in \text{avail}_{PL}(\text{lev}, \text{next}_S(\text{lev}, s, pl_t))) \wedge \\ &\quad (\text{lev} \notin \text{avail}_{LEV}(s) \vee pl_f \notin \text{avail}_{PL}(\text{lev}, s)))\}\end{aligned}$$

Again, some events can be available to multiple players as options. And although δ is dependent on something other than the event transition instance itself, the states before and after it are always fixed and independent of the rest of the path. Therefore, it is a valid measurement during pathfinding. The drawback of this measure is that it only filters out immediate repetitions of an option. A more complex metric would also reduce its value if an event is re-encountered later on. But to keep the metric simple, and because the calculation is an approximation anyhow, this is not done. The relation specified in Equation 1 is guaranteed by this weight function because for each player only a subgroup of all options is counted.

Due to the fact that the options metric focuses on the state after an event it omits the initial state, which has no event leading to it. This means that all options from this state must be added to the path to get the overall result:

$$\begin{aligned}O^{\text{overall}}(s^E) &= \sum_{i=0}^{\text{length}^O(s^E)-1} \text{weight}_i^O(s^E) + \sum_{\text{lev} \in \text{avail}_{LEV}(s^I)} \text{weight}(\text{lev}, s^I) \\ O^{\text{player}}(pl, s^E) &= \sum_{i=0}^{\text{length}^O(pl, s^E)-1} \text{weight}_i^O(pl, s^E) + \sum_{\substack{\text{lev} \in \text{avail}_{LEV}(s^I) \\ pl \in \text{avail}_{PL}(\text{lev}, s)}} \text{weight}(\text{lev}, s^I)\end{aligned}$$

For the example in Figure 32 and Equation 2, this results in the following option values:

$$\begin{array}{ll} \text{weights}^O(p_0) = (2, 2, 1, 0) & O^{\text{overall}}(s_0^E) = 5 + 1 = 6 \\ \text{weights}^O(p_0, pl_0) = (1, 1, 1, 0) & O^{\text{player}}(pl_0, s_0^E) = 3 + 1 = 4 \\ \text{weights}^O(p_0, pl_1) = (1, 1, 1, 0) & O^{\text{player}}(pl_1, s_0^E) = 3 + 0 = 3 \\ O^{\text{player}\%}(pl_0, s_0^E) \approx 67\% & O^{\text{player}\%}(pl_1, s_0^E) = 50\% \end{array}$$

On the path above there is a difference of about 17% between the players, which is slightly unbalanced.

$$\begin{array}{ll} \text{weights}^O(p_1) = (2, 2, 1, 0) & O^{\text{overall}}(s_1^E) = 5 + 1 = 6 \\ \text{weights}^O(p_1, pl_0) = (1, 1, 0, 0) & O^{\text{player}}(pl_0, s_1^E) = 2 + 1 = 3 \\ \text{weights}^O(p_1, pl_1) = (1, 1, 1, 0) & O^{\text{player}}(pl_1, s_1^E) = 3 + 0 = 3 \\ O^{\text{player}\%}(pl_0, s_1^E) = 50\% & O^{\text{player}\%}(pl_1, s_1^E) = 50\% \end{array}$$

The second path does not show any differences and therefore is fully balanced.

$$\begin{array}{ll} \text{weights}^O(p_2) = (2, 1, 1, 0) & O^{\text{overall}}(s_2^E) = 4 + 1 = 5 \\ \text{weights}^O(p_2, pl_0) = (1, 1, 0, 0) & O^{\text{player}}(pl_0, s_2^E) = 2 + 1 = 3 \\ \text{weights}^O(p_2, pl_1) = (1, 0, 1, 0) & O^{\text{player}}(pl_1, s_2^E) = 2 + 0 = 2 \\ O^{\text{player}\%}(pl_0, s_2^E) = 60\% & O^{\text{player}\%}(pl_1, s_2^E) = 40\% \end{array}$$

Lastly, the third path also has a difference of about 17%, which is slightly unbalanced.

7.2.3 Value Aggregation

After the balancing metrics based on paths for each player pl and ending s^E have been calculated, the developer would have to look at $|Pl| \cdot |S^E|$ different values:

$$\begin{pmatrix} M^{player\%}(Pl_0, s_0^E) & \dots & M^{player\%}(Pl_{|Pl|-1}, s_0^E) \\ \dots & \dots & \dots \\ M^{player\%}(Pl_0, s_{|S^E|-1}^E) & \dots & M^{player\%}(Pl_{|Pl|-1}, s_{|S^E|-1}^E) \end{pmatrix}$$

It is obvious that it would be beneficial to aggregate these values into just a few ones describing the overall balancing of the game.

For this, we propose to first aggregate over all players towards each end state, describing the (in-)equality among the players on the paths to this ending. Then, the values for each ending can be aggregated to generalize these results for the whole game. Aggregating over all paths first would instead show how much the balancing for each player varies between paths. Afterwards, aggregating over all players would show whether these fluctuations are different between the players. In order explain why this is problematic, the following results that signify strong imbalances along both dimensions are assumed:

$$\begin{pmatrix} M^{player\%}(Pl_0, s_0^E) = 0\% & M^{player\%}(Pl_1, s_0^E) = 100\% \\ M^{player\%}(Pl_0, s_1^E) = 100\% & M^{player\%}(Pl_1, s_1^E) = 0\% \end{pmatrix}$$

Aggregating over the end states first shows that the balancing results vary greatly between the endings, then aggregating over the players shows that this is the case for both players. Intuitively this may sound balanced at first. In contrast, aggregating over the players shows that the paths to both endings are highly unbalanced and subsequently aggregating over the paths shows that this is the case for both paths, in this case the whole game. This result allows for a much clearer interpretation. Additionally, players usually play a game only once or a few times, so they are more concerned about the path they are taking than an overall picture. Therefore, having an aggregated value for each ending is a more useful intermediate result when looking at individual values for further analysis.

For assessing the (in-)equality on the path values to an ending, $|Pl|$ values must be compared: $\bar{M}^{ending}(s^E) = \{M^{player\%}(Pl_0, s^E), \dots, M^{player\%}(Pl_{|Pl|-1}, s^E)\}$. One popular metric for calculating the variance between a number of values is the standard deviation $SD(M^{ending}(s^E))$, which becomes smaller the more similar the values are. For a perfectly balanced game in which all values are the same, the standard deviation is 0. However, the maximum value of the standard deviation depends on the concrete metric, making the interpretation of values larger than zero difficult. We therefore propose to normalize the value in order to keep it in the range of $[0, 1]$ ($[0\%, 100\%]$). To do this, the highest possible standard deviation has to be known, which is different depending on whether the number of values $|M|$ is odd or even.

If $|M|$ is even it can be written as $|M| = 2 \cdot n$ with $n \in \mathbb{N}$. In this case, n values being at the lower and n values being the upper bound of the value range results in the highest possible standard deviation. If the values are in $[0, 1]$, like in $M^{player\%}(Pl, s^E)$, this means that the worst value distribution is:

$$\begin{aligned} M_{worst}^{even} = & \{m_0 = 0, \dots, m_{n-1} = 0, \\ & m_n = 1, \dots, m_{2 \cdot n - 1} = 1\} \end{aligned}$$

This results in the following standard deviation for this worst case (the full calculation can be found in Section D.1):

$$\begin{aligned} SD_{\text{worst}}^{\text{even}}(n) &= \sqrt{\frac{n \cdot \left(0 - \frac{\text{SUM}(M_{\text{worst}}^{\text{even}})}{2 \cdot n}\right)^2 + n \cdot \left(1 - \frac{\text{SUM}(M_{\text{worst}}^{\text{even}})}{2 \cdot n}\right)^2}{2 \cdot n}} \\ SD_{\text{worst}}^{\text{even}}(n) &= \sqrt{\frac{n \cdot \left(0 - \frac{n}{2 \cdot n}\right)^2 + n \cdot \left(1 - \frac{n}{2 \cdot n}\right)^2}{2 \cdot n}} \\ SD_{\text{worst}}^{\text{even}}(n) &= \sqrt{\frac{n \cdot \left(\frac{1}{4} + \frac{1}{4}\right)}{2 \cdot n}} = \frac{1}{2} \end{aligned}$$

If $|M|$ is odd it can be written as $|M| = 2 \cdot n + 1$ with $n \in \mathbb{N}$. In this case, n values being at the lower and $n + 1$ values being the upper bound (or vice versa) of the value range results in the highest possible standard deviation. For values in $[0, 1]$ the worst value distribution is then:

$$\begin{aligned} M_{\text{worst}}^{\text{odd}} &= \{m_0 = 0, \dots, m_{n-1} = 0, \\ &\quad m_n = 0, \\ &\quad m_{n+1} = 1, \dots, m_{2 \cdot n} = 1\} \\ M'_{\text{worst}}^{\text{odd}} &= \{m_0 = 0, \dots, m_{n-1} = 0, \\ &\quad m_n = 1, \\ &\quad m_{n+1} = 1, \dots, m_{2 \cdot n} = 1\} \end{aligned}$$

This results in the following standard deviation (the full calculation for $M_{\text{worst}}^{\text{odd}}$ as well as the one for $M'_{\text{worst}}^{\text{even}}$ can be found in Section D.2):

$$\begin{aligned} SD_{\text{worst}}^{\text{odd}}(n) &= \sqrt{\frac{(n+1) \cdot \left(0 - \frac{\text{SUM}(M_{\text{worst}}^{\text{odd}})}{2 \cdot n+1}\right)^2 + n \cdot \left(1 - \frac{\text{SUM}(M_{\text{worst}}^{\text{odd}})}{2 \cdot n+1}\right)^2}{2 \cdot n+1}} \\ &= \sqrt{\frac{(n+1) \cdot \left(0 - \frac{n}{2 \cdot n+1}\right)^2 + n \cdot \left(1 - \frac{n}{2 \cdot n+1}\right)^2}{2 \cdot n+1}} \\ &= \sqrt{\frac{n \cdot \left(\frac{n^2}{(2 \cdot n+1)^2} + \frac{n}{(2 \cdot n+1)^2} + 1 - \frac{2 \cdot n}{2 \cdot n+1} + \frac{n^2}{(2 \cdot n+1)^2}\right)}{2 \cdot n+1}} \\ &= \sqrt{\frac{n \cdot \left(\frac{n \cdot (2 \cdot n+1)}{2 \cdot n+1} + 1\right)}{(2 \cdot n+1)^2}} = \frac{\sqrt{n \cdot (n+1)}}{(2 \cdot n+1)} = M'_{\text{worst}}^{\text{odd}} \end{aligned}$$

The general formula for the highest possible standard deviation combines these two cases:

$$SD_{\text{worst}}(M) = \begin{cases} SD_{\text{worst}}^{\text{even}}(|M|) = \frac{1}{2} & \text{if } |M| \text{ is even;} \\ SD_{\text{worst}}^{\text{odd}}(|M|) = \frac{\sqrt{|M| \cdot (|M|+1)}}{(2 \cdot |M|+1)} & \text{else.} \end{cases}$$

Using this formula, a normalized standard deviation can be calculated, which is always in $[0, 1]$ and therefore can be interpreted more easily:

$$SD_{\text{norm}}(M) = \frac{SD(M)}{SD_{\text{worst}}(M)} \tag{4}$$

The final result for a metric M and a given ending s^E is then the normalized standard deviation over all players:

$$M^{ending}(s^E) = SD_{norm}(\bar{M}^{ending}(s^E))$$

This gives the difference (imbalance) between players for this ending in $[0, 1]$ or $[0\%, 100\%]$. Accordingly, the value should be low for a balanced game (zero would indicate a perfectly balanced game).

Since there is a value for each player at this stage, the worst case SD can be calculated once using the number of players for $|M|$. In the example (Figure 32 and Equation 2) there are two players, so the worst possible SD is $SD_{worst}(M) = \frac{1}{2} = 50\%$. Using this, the following results per ending are obtained (effort only):

$$\begin{aligned} E^{ending}(s_0^E) &= SD_{norm}(\{75\%, 0\%)\} = \frac{SD(\{75\%, 0\%)\})}{50\%} = \frac{37.5\%}{50\%} = 75\% \\ E^{ending}(s_1^E) &= SD_{norm}(\{50\%, 50\%\}) = \frac{SD(\{50\%, 50\%\})}{50\%} = \frac{0\%}{50\%} = 0\% \\ E^{ending}(s_2^E) &= SD_{norm}(\{50\%, 50\%\}) = \frac{SD(\{50\%, 50\%\})}{50\%} = \frac{0\%}{50\%} = 0\% \end{aligned}$$

For s_0^E there is a big difference between the players' values, but the other paths seem to be balanced. These aggregated values match the results obtained by looking at the individual values (Section 7.2.2.1), which is exactly what is needed.

After this step there is only one value per path, which need to be aggregated over all paths in order to get a single value for the game.

$$M_{agg} = AGG \left(\left\{ M^{ending}(s_0^E), \dots, M^{ending}(s_{|S^E|-1}^E) \right\} \right)$$

The first interesting aggregation function is the average ($agg = avg$), which represents the overall picture over all paths. Additionally, the maximum value ($agg = max$) should be checked as well in order to find the worst path. If the average (and max) is zero, the game is perfectly balanced according to the approximation. In contrast, if the average is greater than zero but still low, the game is reasonably balanced. But if the average is high, the game must be very unbalanced. Lastly, if the maximum is high, there is at least one unbalanced path in the game, even if the average is low and the game is pretty balanced overall. In this case the developer could identify the unbalanced path by looking at the individual values before they were aggregated. To improve the balance he can then try to reassign some events on this path to another player. The minimum aggregation is not interesting, since there is no use in identifying paths that are already balanced.

For the example (Figure 32 and Equation 2) the following end result is achieved (effort only):

$$E_{avg}^{ending} = avg(\{75\%, 0\%, 0\%)\} = 25\%$$

$$E_{max}^{ending} = max(\{75\%, 0\%, 0\%)\} = 75\%$$

The average effort difference is noticeably greater than zero, which means that either all paths are a bit unbalanced or that a few ones are greatly unbalanced. Since the maximum effort difference is much higher, the latter of these explanations is confirmed. To improve the game's effort balance the offending path to s_0^E should be changed, for example by assigning lev_2 and lev_3 to pl_1 only.

These aggregations represent a one-time post-processing step with a complexity of $O(|Pl| \cdot |S^E|)$, which is smaller than the pathfinding complexity and has therefore no impact on the overall complexity.

7.2.4 Value Distribution

Looking at the recurring example (Figure 32), a weakness of the approach so far becomes apparent. Although the path to s_1^E is balanced overall (Equation 3), the effort is concentrated into two blocks. For the first part only pl_0 is acting, the second part is solved exclusively by pl_1 . It is easy to imagine that such homogenous (or unbalanced) sections are viewed as boring, especially if the game is longer. Therefore, the balancing definition also requires the metrics to be uniformly distributed. Hereby, it is important to note that if the overall path is badly balanced, the distribution of values along this path is a menial issue.

Thus, the general idea is to calculate the distribution of each metric $M^{distrib}(pl, s^E)$ on all paths (i.e. for each ending and player). This value lies in $[0, 1]$ with an optimal value of zero (no difference between different parts).

This distribution should be calculated on the optimal paths which have already been found ($M^{distrib}(pl, s^E) = M^{distrib}(\text{minPath}^M(pl, s^E))$). Optimizing it on its own during the pathfinding step could result in paths on which players that are normally idle execute irrelevant actions to improve the distribution. However, this would increase the overall path length and could even lead to the algorithm moving farther away from the final goal in regards to the actual effort, which increases runtime and is ultimately counterintuitive to what a user would expect. The drawback of this method is that – since the distribution is not optimized – there could be multiple shortest paths in regards to the original metric, from which only one is returned by the pathfinding. Since these paths may have different distributions, it is therefore beneficial to investigate all shortest paths (Section 7.2.5).

Again, the values for multiple players and endings have to be aggregated. Since the optimal value is known and because taking the standard deviation would mask similar bad distributions over all paths, the average and maximum ($\text{agg} \in \{\text{avg}, \text{max}\}$) should be considered instead. This is the case for aggregating over all players and endings, therefore the aggregation can be done along both dimensions simultaneously.

$$M_{\text{agg}}^{distrib} = \text{AGG} \left(\begin{array}{ccc} M^{distrib}(Pl_0, s_0^E) & , \dots, & M^{distrib}(Pl_{|Pl|-1}, s_0^E), \\ \dots, & \dots, & \dots \\ M^{distrib}(Pl_0, s_{|S^E|-1}^E), & \dots, & M^{distrib}(Pl_{|Pl|-1}, s_{|S^E|-1}^E) \end{array} \right)$$

Like the individual values of $M^{distrib}$, the aggregated value is also in $[0, 1]$ with zero being optimal. If the value is higher, the developer can again look at the values of each individual path in order to solve the underlying problem.

7.2.4.1 Fixed Sections

The simplest approach for calculating the value distribution is to partition a given path into $|Sc|$ fixed sections (indices $Sc = 0, \dots, |Sc| - 1$). Taking each partition as a

path on its own, the desired metric can be calculated for each section. By comparing these values, the metric's distribution between the sections can be determined.

These sections can either be defined manually by the developer or generated automatically. Letting the developer define the sections allows him to match the semantic structure of the game, for example partitioning the game according to the dramatic acts of the underlying narrative. Technically this means that the developer has to provide $|Sc| - 1$ cutting points $c_0, \dots, c_{|Sc|-1}$, which are states ($c \in S$) that mark the end of a section. These constitute an ordered list, to which the initial state and the end state of the path are added: $C(s^E) = (s^I, c_0, \dots, c_{|Sc|-1}, s^E)$. In order to work correctly, the developer has to make sure that the chosen cutting points exist on the current path:

$$\text{minPath}^M(\text{pl}, s^E) = \{s^I, \dots, c_0, \dots, c_1, \dots, c_{|Sc|-1}, \dots, s^E\} \quad (5)$$

To calculate the balancing values for each section, an auxiliary function that returns the index of a state in a path is needed: $\text{index}(s, p) = i \Leftrightarrow s_i(p) = s$. The straightforward approach would then be to sum up the desired metric over all events in the subpath between two cutting points. However, those paths can be of different lengths:

$$\begin{aligned} \text{length}_{Sc}^M(\text{pl}, s^E, sc) &= \text{index}(c_{sc+1}, \text{minPath}^M(\text{pl}, s^E)) \\ &\quad - \text{index}(c_{sc}, \text{minPath}^M(\text{pl}, s^E)) \end{aligned}$$

If p_{\min}^M is a minimal path to an ending using the metric M and the path contains all cutting points as required by Equation 5 this can also be written as:

$$\begin{aligned} \text{length}_{Sc}(p_{\min}^M, sc) &= \text{index}(c_{sc+1}, p_{\min}^M) \\ &\quad - \text{index}(c_{sc}, p_{\min}^M) \end{aligned}$$

Using this length, the balancing value for each section can be normalized in order to make them comparable:

$$\begin{aligned} M^{\text{player}}(\text{pl}, s^E, sc) &= M^{\text{player}}(\text{minPath}^M(\text{pl}, s^E), sc) \\ &\quad \sum_{i=\text{index}(c_{sc}, p_{\min}^M)}^{\text{index}(c_{sc+1}, p_{\min}^M)-1} \text{weight}_i^M(p_{\min}^M) \\ M^{\text{player}}(p_{\min}^M, sc) &= \frac{\sum_{i=\text{index}(c_{sc}, p_{\min}^M)}^{\text{index}(c_{sc+1}, p_{\min}^M)-1} \text{weight}_i^M(p_{\min}^M)}{\text{length}_{Sc}(p_{\min}^M, sc)} \end{aligned}$$

If the developer did not define cutting points, for example when the game's structure is diverging in a way that many individual cutting points need to be defined for each ending, another approach is also viable. Paths can be cut automatically, either based on the desired number of sections or their intended length. As the number of sections can be calculated directly from the intended length by dividing the overall path length, only the first option will be described. Compared to defining sections manually, these automated splits require no knowledge about the game's structure and therefore work under any circumstances.

When a specific number of sections need to be generated, their average length can be calculated for each path:

$$\text{avgLength}_{Sc}^M(\text{pl}, s^E) = \frac{\text{length}^M(\text{pl}, s^E)}{|Sc|}$$

If the number of sections is a divisor of the overall path length every path has the same length and no normalization is needed:

$$\text{length}_{S_c}^M(\text{pl}, s^E, sc) = \text{avgLength}_{S_c}^M(\text{pl}, s^E) \quad \forall sc \in S_c$$

However, the average length is not always a natural number, for example if a path of ten events (ev_0, \dots, ev_9) has to be split into three sections (average section length of about 3.33). One option is to round at each multiple of the average length (which defines a section border) and assign the event on that border to one of the adjacent sections based on this rounding:

$$\text{index}_{\text{End}}(\text{pl}, s^E, sc) = \text{round}(\text{avgLength}_{S_c}^M(\text{pl}, s^E) \cdot (sc + 1)) - 1$$

$$\text{index}_{\text{Begin}}(\text{pl}, s^E, sc) = \begin{cases} \text{index}_{\text{End}}(\text{pl}, s^E, sc + 1) & \text{if } i > 0; \\ 0 & \text{else.} \end{cases}$$

$$\text{length}_{S_c}^M(\text{pl}, s^E, sc) = \text{index}_{\text{End}}(\text{pl}, s^E, sc) - \text{index}_{\text{Begin}}(\text{pl}, s^E, sc)$$

Following this, the sections in the example would be $(ev_0, ev_1, ev_2), (ev_3, ev_4, ev_5, ev_6), (ev_7, ev_8, ev_9)$. Due to the rounding, the length of each section varies slightly, similar to the developer-defined sections. Thereby normalization is needed for them, too. As an alternative, the value of the events lying “between” two sections could be partially assigned to each adjacent section based on the fractional digits remaining.

Using this, the normalized value for each section is defined as:

$$\begin{aligned} M^{\text{player}}(\text{pl}, s^E, sc) &= M^{\text{player}}(\text{minPath}^M(\text{pl}, s^E), sc) \\ &\quad \sum_{i=\text{sc} \cdot \text{length}_{S_c}^M(\text{pl}, s^E, sc)}^{(\text{sc}+1) \cdot \text{length}_{S_c}^M(\text{pl}, s^E, sc)-1} \text{weight}_i^M(p_{\text{min}}^M) \\ M^{\text{player}}(p_{\text{min}}^M, sc) &= \frac{\sum_{i=\text{sc} \cdot \text{length}_{S_c}^M(\text{pl}, s^E, sc)}^{\text{sc} \cdot \text{length}_{S_c}^M(\text{pl}, s^E, sc)} \text{weight}_i^M(p_{\text{min}}^M)}{\text{length}_{S_c}^M(\text{pl}, s^E, sc)} \end{aligned}$$

Independent of how the path is separated into sections, for each combination of player and ending there are now $|S_c|$ section values of the metric M :

$$M^{\text{playerSect}}(\text{pl}, s^E) = \{M^{\text{player}}(\text{pl}, s^E, 0), \dots, M^{\text{player}}(\text{pl}, s^E, |S_c| - 1)\}$$

These values have to be aggregated in such a way that the distribution of a balancing metric between the sections is expressed in a single value. Here, the standard deviation is useful again. In the best case, every section has the same value, which means that the value calculated for the overall path is also representative for each part. In this case the standard deviation is zero. The worst case is that the value of the overall path is concentrated in only one section while the values of every other section on the path are zero. Due to the normalization, this section must also be one of the smallest ones for its value being the largest one possible:

$$\begin{aligned} M_{\text{worst}}^{\text{playerSect}}(\text{pl}, s^E) &= \left(\frac{M^{\text{player}}(\text{pl}, s^E)}{\text{minLength}_{S_c}^M(\text{pl}, s^E)}, 0, \dots, 0 \right) \\ \text{minLength}_{S_c}^M(\text{pl}, s^E) &= \min_{sc \in S_c} \text{length}_{S_c}^M(\text{pl}, s^E, sc) \end{aligned}$$

This is different to the aggregation over all players (Section 7.2.3), for which the worst case is that half of the values have the maximum value and the other ones are zero.

Hereby, the crucial difference is that the sum of all section values combined can be at most the overall value of the path.

$$\sum_{i=0}^{|Sc|-1} M^{player}(p_{min}^M, sc) \cdot \text{length}_{Sc}^M(pl, s^E, sc) = M^{player}(pl, s^E)$$

$$\text{SUM}(M^{playerSect}(pl, s^E)) \leq M^{player}(pl, s^E)$$

Therefore, not only is every value in the range of $\left[0, \frac{M^{player}(pl, s^E)}{\minLength_{Sc}^M(pl, s^E)}\right]$, but their sum is also in the same value range. For the player percentages where theoretically every player could reach the maximum value of 100% (for example when every option is available to every player) this is not the case and the sum is in $[0\%, |Pl| \cdot 100\%]$. This results in a worst case standard deviation dependent on the number of values n (for the complete calculation see Appendix Section D.3):

$$\begin{aligned} SD_{worst\%}(M) &= \sqrt{\frac{\left(\text{SUM}(M) - \frac{\text{sum}(M)}{n}\right)^2 + (n-1) \cdot \left(0 - \frac{\text{sum}(M)}{n}\right)^2}{n}} \\ &= \sqrt{\frac{\left((\text{SUM}(M))^2 \cdot (Pl-1)\right)}{Pl^2}} = \sqrt{(n-1)} \cdot \frac{\text{sum}(M)}{n} \end{aligned}$$

To remove this dependency on the number of sections and to get a value in $[0, 1]$, the normalized standard deviation should be used (also Equation 4):

$$\begin{aligned} SD_{norm\%}(M^{playerSect}(pl, s^E)) &= \sqrt{(|Sc|-1)} \cdot \frac{\text{SUM}(M^{playerSect}(pl, s^E))}{|Sc|} \\ M^{section}(pl, s^E) &= SD_{norm\%}(M^{playerSect}(pl, s^E)) \\ &= \frac{SD(M^{playerSect}(pl, s^E))}{SD_{worst\%}(M^{playerSect}(pl, s^E))} \end{aligned}$$

This value then constitutes how much the metric M differs between the individual sections. Due to the normalization it lies in $[0, 1]$ with zero signaling that there is no difference between the sections. Therefore, $M^{section}(pl, s^E)$ fulfills the requirements defined for $M^{distrib}(pl, s^E)$ in Section 7.2.4.

Alternatively, the maximal difference between a section value and the overall path value can be used:

$$M^{sectionDiff}(pl, s^E) = \max_{sc \in Sc} \frac{(M^{player}(pl, s^E) - M^{player}(pl, s^E, sc))}{M^{player}(pl, s^E, sc)}$$

This value also fulfills the requirements, but takes only the section with the largest difference into account.

One drawback of this method is that the order of sections is not important and the result is highly dependent on the number of sections. For example, with a section length of two, a weight distribution of $((0, 0), (0, 0), (1, 1), (1, 1))$ is not worse than $((0, 0), (1, 1), (0, 0), (1, 1))$, in which four are reordered. Though, with a section length of four $((0, 0, 0, 0), (1, 1, 1, 1))$ is significantly worse than $((0, 0, 1, 1), (0, 0, 1, 1))$, on which the same reordering is applied. There is also a high dependence on the location of the section borders. $((0, 0), (1, 1), (0, 0), (1, 1))$ is completely unbalanced,

but moving a single event from the first position to the last results in the completely balanced $((0,1),(1,0),(0,1),(1,0))$. At least some of these problems could be mitigated by having a sliding window with a fixed length moving over the whole path and evaluating every combination of connected events, as it would include both advantageous as well as disadvantageous splits. However, the impact of these flaws diminishes quickly the longer the sections get, which highlights the importance of choosing the right number of sections. With a section count of one this section represents the overall path, so there is no additional information obtained. In contrast, if there are as many sections as there are events, the game can seem to be highly imbalanced. In this case there is nothing else in each “section” that could act as a counter-balance when the single event is assigned to a specific player. So, our advice is to choose a section length which represents between one or two minutes of playtime, as this is a time span in which players would start to notice imbalances. Depending on the time each individual event takes, this can mean different section lengths, but should in any case result in sections that do not fall into the pitfalls above.

In order to work correctly with this approach, the options metric defined in Section 7.2.2.3 has to be changed. In the original version options that are available in multiple states are only counted in the first state at which they appear in order to penalize repetition. This works well for calculating the overall sum of weights and is the only viable option for the pathfinding, during which the future states (and therefore how long an option will be available) are not yet known. Events that are available in multiple sections, however, are only counted in the first one. One could argue that options are only interesting when they are new and keep it that way, but alternatively a post-processing step can be executed in which each option’s weight is distributed equally over all states during which it is available. This prevents misleading results when a section seems to have no options at all because all options available are already known. A mixed approach combining both variants (with an weight factor to control their influence) is also possible.

$$\begin{aligned} \text{weight}^O(p, i) &= \sum_{\text{lev} \in \gamma(p, i)} \frac{\text{weight}(\text{lev}, \text{next}_S(\text{lev}_i(p), s_i(p), p_{l_i}(p)))}{\text{repetitions}(p, i)} \\ \gamma(p, i) &= \text{avail}_{\text{LEV}}(\text{next}_S(\text{lev}_i(p), s_i(p), p_{l_i}(p))) \\ \text{weight}^O(p, i, pl) &= \sum_{\text{lev} \in \gamma(p, i, pl)} \frac{\text{weight}(\text{lev}, \text{next}_S(\text{lev}_i(p), s_i(p), p_{l_i}(p)))}{\text{repetitions}(p, i, pl)} \\ \gamma(p, i, pl) &= \text{avail}_{\text{LEV}}(\text{next}_S(\text{lev}_i(p), s_i(p), p_{l_i}(p))) | pl \in \text{avail}_{pl}(\text{lev}, s_i(p)) \end{aligned}$$

The function $\text{repetitions}(p, i, pl)$ counts how long the event $\text{lev}_i(p)$ is available on the path p for player pl , $\text{repetitions}(p, i)$ does the same for the whole group.

When calculating the overall effort for the example (Figure 32 and Equation 2) the paths to ending s_1^E and s_2^E seem to be perfectly balanced. For simplification reasons, only the values for pl_0 are calculated and $\text{weights}_{Sc}^M(p, pl)$ is used as a variant of $\text{weights}^M(p, pl)$ in which the weights are grouped into sections.

For $|Sc| = 2$ fixed length sections, the path to end state s_1^E shows the worst possible distribution:

$$\begin{aligned} \text{weights}_{Sc}^M(p_1, pl_0) &= ((1, 1), (0, 0)) \\ E^{\text{playerSect}}(pl_0, s_1^E) &= \left\{ \frac{2}{2}, \frac{0}{2} \right\} = \{1, 0\} \\ \text{SUM}(E^{\text{playerSect}}(pl_0, s_1^E)) &= 1 \\ \text{AVG}(E^{\text{playerSect}}(pl_0, s_1^E)) &= \frac{1+0}{2} = \frac{1}{2} \\ \text{SD}(E^{\text{playerSect}}(pl_0, s_1^E)) &= \sqrt{\frac{(1-\frac{1}{2})^2 + (0-\frac{1}{2})^2}{2}} = \sqrt{\frac{2 \cdot (\frac{1}{2})^2}{2}} = \sqrt{\frac{1}{4}} = \frac{1}{2} \\ \text{SD}_{\text{worst}\%}(M) &= \sqrt{(2-1)} \cdot \frac{1}{2} = \frac{1}{2} \\ E^{\text{section}}(pl_0, s_1^E) &= \frac{\frac{1}{2}}{\frac{1}{2}} = 1 = 100\% \end{aligned}$$

The path to ending s_2^E , however, is perfectly balanced, even between sections:

$$\begin{aligned} \text{weights}_{Sc}^M(p_2, pl_0) &= ((1, 0), (1, 0)) \\ E^{\text{playerSect}}(pl_0, s_2^E) &= \left\{ \frac{1}{2}, \frac{1}{2} \right\} \\ \text{SUM}(E^{\text{playerSect}}(pl_0, s_2^E)) &= 1 \\ \text{AVG}(E^{\text{playerSect}}(pl_0, s_2^E)) &= \frac{\frac{1}{2} + \frac{1}{2}}{2} = \frac{1}{2} \\ \text{SD}(E^{\text{playerSect}}(pl_0, s_2^E)) &= \sqrt{\frac{(\frac{1}{2}-\frac{1}{2})^2 + (\frac{1}{2}-\frac{1}{2})^2}{2}} = \sqrt{\frac{2 \cdot (0)^2}{2}} = \sqrt{0} = 0 \\ \text{SD}_{\text{worst}\%}(M) &= \sqrt{(2-1)} \cdot \frac{1}{2} = \frac{1}{2} \\ E^{\text{section}}(pl_0, s_2^E) &= \frac{0}{\frac{1}{2}} = 0 = 0\% \end{aligned}$$

7.2.4.2 Variable Sections

Instead of using fixed section lengths a variable approach can be used, which solves the problems caused by inappropriate section lengths or cutting points. This approach counts how many values larger than zero appear next to each other in a path. For example, while weights of $(0, 0, 1, 1, 0, 0, 1, 1)$ are balanced with a fixed section length of four and unbalanced with a section length of two, the flexible approach would instead note that the longest series of weights larger than zero is two. Therefore, this can also be interpreted as adaptively finding a good value for the section lengths. To combine the consecutive non zero values, a function is needed which transforms a series of $(2, 2, 1, 0, 0, 1, 0, 1)$ into $(5, 1, 1)$.

First, the auxiliary function start is used to decide whether a value marks the begin of a cluster. This is the case when it is greater than zero and either the first value in a path or the value beforehand is zero.

$$\text{start}^M(p, i, pl) = \begin{cases} 1 & \text{if } (\text{weight}^M(p, i, pl) \neq 0 \wedge \\ & (i = 0 \vee \text{weight}^M(p, i - 1, pl) = 0)); \\ 0 & \text{else.} \end{cases}$$

This would transform $(2, 2, 1, 0, 0, 1, 0, 1)$ into $(1, 0, 0, 0, 0, 1, 0, 1)$.

Another function consecutive is used to sum up the weight of an event and all following ones until the first zero is encountered or the path ends.

$$\text{cons}^M(p, i, pl) = \begin{cases} 0 & \text{if } \text{weight}^M(p, i, pl) = 0; \\ \text{weight}^M(p, i, pl) + \text{cons}^M(p, i + 1, pl) & \text{if } \text{weight}^M(p, i, pl) \neq 0 \wedge \\ & i < \text{length}(p) - 1; \\ \text{weight}^M(p, i, pl) & \text{if } \text{weight}^M(p, i, pl) \neq 0 \wedge \\ & i = \text{length}(p) - 1. \end{cases}$$

This function calculates $(5, 3, 1, 0, 0, 1, 0, 1)$ from the input $(2, 2, 1, 0, 0, 1, 0, 1)$.

Since the function start is only one for the first event in a cluster and cons calculates the combined cluster value remaining, multiplying their individual values produces $(5, 0, 0, 0, 0, 1, 0, 1)$. Removing all values that are zero afterwards produces exactly the result needed.

$$\begin{aligned} c^M(p, i, pl) &= \text{start}^M(p, i, pl) \cdot \text{cons}^M(p, i, pl) \\ c_{\min}^M(i, pl, s^E) &= c^M(\text{minPath}^M(pl, s^E), i, pl) \\ M^{\text{cons}}(pl, s^E) &= \bigcup_{i=0}^{\overline{\text{length}}^M(pl, s^E)-1} \{c_{\min}^M(i, pl, s^E) | c_{\min}^M(i, pl, s^E) > 0\} \end{aligned}$$

Alternatively, all homogenous clusters (zero or non zero) could be counted, which would result in $(3, 2, 1, 1, 1)$ for the example. Counting the zero events, however, prevents the summation of weights, as this would result in zero no matter how many consecutive zeros are encountered. Not summing up the weights in turn also hides whether the weights vary between events, so this approach is not pursued any further.

After the consecutive weights have been combined, the average and maximum cluster weight can be calculated. If those values are high, then there are long sequences of uninterrupted weights larger than zero, which indicates an uneven distribution. Again, the minimum is irrelevant due to being the best case and using the standard deviation would mask similarly bad values for all clusters.

$$\begin{aligned} M^{\text{playerAvg}}(pl, s^E) &= \text{AVG}(M^{\text{cons}}(pl, s^E)) \\ M^{\text{playerMax}}(pl, s^E) &= \text{MAX}(M^{\text{cons}}(pl, s^E)) \end{aligned}$$

To normalize these values into $[0, 1]$ they can be divided by the overall value for the player on this path. This allows outputs of “at most 5% of a path’s effort is provided without pause”, which are more understandable than abstract values without a reference point. It also causes the values fulfill the requirements of $M^{\text{distrib}}(pl, s^E)$. However, it is important to note that for this value the lower threshold is not zero because at least a single event must be “consecutive”. But these thresholds get smaller the longer the overall path is and perfect values (0% difference between players) are seldom reached, even in other metrics. This means that a small but non perfect value for the distribution should not concern the user. Both $M^{\text{playerAvg}\%}$ giving the average distribution and $M^{\text{playerMax}\%}$ providing the worst case are relevant, but only one final value $M^{\text{distrib}}(pl, s^E)$ is needed. Therefore, we suggest using a combination of both, weighted by a user defined value σ . For this value we propose an equal influence of both ($\sigma = 0.5$).

$$\begin{aligned} M^{\text{playerAvg}\%}(pl, s^E) &= \frac{M^{\text{playerAvg}}(pl, s^E)}{M^{\text{player}}(pl, s^E)} \\ M^{\text{playerMax}\%}(pl, s^E) &= \frac{M^{\text{playerMax}}(pl, s^E)}{M^{\text{player}}(pl, s^E)} \\ \sigma &\in [0, 1] \\ M^{\text{distrib}}(pl, s^E) &= \sigma \cdot M^{\text{playerAvg}\%}(pl, s^E) + \\ &\quad (1 - \sigma) * M^{\text{playerMax}\%}(pl, s^E) \end{aligned}$$

But this metric has a major drawback. While its behavior is intuitive for metrics like effort and waiting, in which the individual weights are frequently zero, it does not work with metrics for which this is not the case. In regards to options, for example, interesting games provide players a few options in each state, so their value would never be zero. In this case there would be only one section spanning the whole path and no additional information is provided by the flexible sections metric. Mathematically, this could be fixed by defining a threshold larger than zero and only counting events in which more options are provided. However, there is no objective and consistent way to define this threshold.

Applied to the recurring example (Figure 32 and Equation 2), the following results for the effort of pl_0 are obtained:

$$\begin{aligned} E^{\text{cons}}(pl_0, s_1^E) &= (2) \\ E^{\text{playerAvg}}(pl_0, s_1^E) &= 2 & E^{\text{playerAvg}\%}(pl_0, s_1^E) &= \frac{2}{2} = 1 \\ E^{\text{playerMax}}(pl_0, s_1^E) &= 2 & E^{\text{playerMax}\%}(pl_0, s_1^E) &= \frac{2}{2} = 1 \\ M^{\text{distrib}}(pl_0, s_1^E) &= 0.5 \cdot 1 + (1 - 0.5) \cdot 1 = 1 = 100\% \end{aligned}$$

$$\begin{aligned} E^{\text{cons}}(pl_0, s_2^E) &= (1, 1) \\ E^{\text{playerAvg}}(pl_0, s_2^E) &= 1 & E^{\text{playerAvg}\%}(pl_0, s_1^E) &= \frac{1}{2} = 0.5 \\ E^{\text{playerMax}}(pl_0, s_2^E) &= 1 & E^{\text{playerMax}\%}(pl_0, s_1^E) &= \frac{1}{2} = 0.5 \\ M^{\text{distrib}}(pl_0, s_2^E) &= 0.5 \cdot 0.5 + (1 - 0.5) \cdot 0.5 = 0.5 = 50\% \end{aligned}$$

Again only the overall balanced endings s_1^E and s_2^E are shown. For s_1^E all of the overall effort that pl_0 has to provide is required sequentially, which is a very bad distribution. The value of 50% for s_2^E seems also pretty bad, although it is already at the actual minimum because each section consists of a single event. This is due to the overall path being unrealistically short. For example, a path length of 20 would result in a value of 10% given the same situation. However, looking at the values before normalization ($E_{\text{playerAvg}}$ and $E_{\text{playerMax}}$) a developer can easily discover this pitfall, even with a short path length.

7.2.5 Extension: Multiple shortest paths

As already mentioned, the basic balancing approximation is unable to detect if there are multiple shortest paths to an end state. This is due to the pathfinding algorithms being designed for finding one example as fast as possible. In some circumstances this can lead to problems, for example in the scenario displayed in Figure 34. Assuming a weight of 1 for each event, both possible paths have the same overall value for each player. In this case, the implementation of the pathfinding algorithm dictates which one of these paths is returned as a result. However, the order of player actions is different, which impacts the distribution (Section 7.2.4). The events in the lower path are equally distributed with the players taking turns, in the upper path both players act in one single block. It would therefore improve the quality of the approximations result if both paths could be considered. Calculating multiple shortest paths can also handle cases in which paths contain events that can be executed in different orders. In this case, all possible permutations of the events are returned as well.

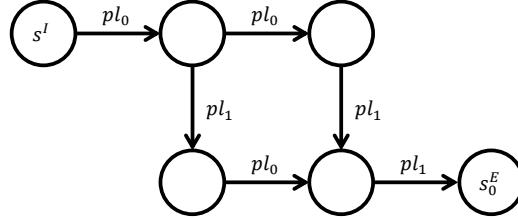


Figure 34: State space example in which multiple shortest paths should be considered for one player.

Finding all shortest paths can be achieved by modifying the Dijkstra or A* algorithm twofold. First, the algorithms must not only record a single transition with which a node has been reached, but multiple ones if the path up to this point has the same length. Additionally, the algorithms cannot terminate anymore after reaching the goal, but after the traversed path grows larger than the shortest path to the ending. These modifications do not increase the theoretical complexity of the algorithm (Big O notation), as they only cause a few more edges to be explored. Finding the goal with the last edge, for example when the states form a straight line, is already the worst case for this type of algorithms without the modification. The calculation result is then not a single path, but a subgraph in which only the nodes and edges lying on any of the shortest paths remain. As the balancing calculations require paths as an input, an intermediate step has to generate all possible paths through this subgraph. The nodes and edges of this graph are a subgroup of the original ones and there are no loops or multiple edges between the same nodes. Each minimal path

can contain at most $|S| - 1$ events. This is due to the fact that the same event could be triggered multiple times from different states, but every state can be visited at most once when the path is an optimal one. So in theory there can be $|S|!$ permutations in which the states can be visited. This is also the upper bound for the number of possible parallel paths to be extracted, if every event is relevant for the game ending and they can be triggered in any order. But in reality, most games are designed to follow a certain progression structure and story flow, which means that the order in which states are visited is much more restricted.

Afterwards, any metrics can be calculated on each of the multiple shortest paths. As the main balancing value is optimized by the pathfinding, it will be the same for every path – but the not optimized distribution will return multiple values. In order to receive a single result nonetheless, the results for all paths must be aggregated. Instead, the average and maximum (i.e. worst) distribution over all parallel paths to the given ending should be used, which is consistent with the next aggregation step over all endings. As noted before, using the standard deviation is not appropriate as a bad distribution on all paths could be masked. However, one must keep in mind that the pathfinding can only optimize a single metric, for example the effort from the perspective of the first player. In this case, the effort of all other players is ignored. When calculating the distribution, the effort provided by other players becomes relevant, as inserting other players' actions into the paths could improve the distribution by breaking up clusters. This in turn can skew the average by including paths which are not consistent with the general approach of looking at the minimal paths only. To fix this, all of the parallel paths on which the overall metric is significantly larger than the minimum of all other paths should be ignored.

7.2.6 Extension: Culling End States

Another extension has been defined that can remove end states. Figure 35 serves as a minimal example for a scenario in which this is beneficial. The (artificial) game it is based on contains two events that the players can trigger. The first one, lev_0 , allows the players to end the game. Additionally, the players can open an irrelevant door with lev_1 first and then trigger the ending. Due to the door having changed its state this results in a different end state. The pathfinding is run for both of these endings and finds two minimal paths as a result. A human, in contrast, would say that there is only one shortest path, the one on which the optional door is ignored. To meet the expectation set by the approach being based on shortest paths, the path including the optional event and the ending it is leading to, should be removed. While concerning a single ending in this example, real games contain multiple optional events which can create a large number of additional end states due to the combinatorial explosion. Also, there can be more than one event triggering the change.

However, those end states cannot be detected by looking at them. In the example above, a simplified version of the endings could be described as $s_0^E = (\text{gameEnded} = \text{true}, \text{doorOpen} = \text{false})$ and $s_1^E = (\text{gameEnded} = \text{true}, \text{doorOpen} = \text{true})$. Comparing these two endings, the only difference between them is the state of doorOpen . Knowing that the initial state of doorOpen is false , it seems intuitive to conclude that an unnecessary action happened to reach s_1^E . But in another version, where opening the door is mandatory as players can only trigger the ending from the other

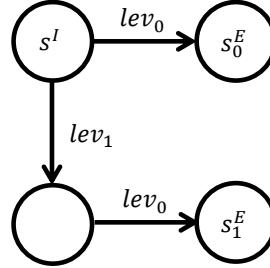


Figure 35: State space example with a superfluous ending.

side, another problem becomes obvious. Here, the optional action is to close the door again after passing through. This would result in the same end states, this time, however, the one in which the door was closed again has to be removed. It is therefore obvious that it is not sufficient to look at the endings alone.

Instead, the paths leading to the endings should be examined. If a path to one ending is a superpath to another, i.e. it contains all events of the other one as well as some additional ones, the path and the ending it is leading to should be removed. For this, the order of events is irrelevant, especially when only one shortest path is calculated per ending, meaning that other valid permutations are unknown. Since this approach checks only whether an event has been triggered or not, their weight is not important and the culling result can be reused for other metrics. However, the check only works when paired with the extension for multiple shortest paths. In games where the same result can be achieved with different events, it could happen that the path to different endings use different events for the same goal. This means that none of these paths can be a superpath of the other, even if one ending contains optional changes. Therefore, it is necessary to compare all combinations of shortest paths to find every subpath-superpath-match. In the example above, the end state s_1^E would be removed, as the path to it is a superpath of the path to s_0^E . The full superpath detection algorithm is described in Listing 1.

As already mentioned in Section 7.2.5 each minimal path contains at most $|S| - 1$ events. If the lookup `FindEquivalent` is done using a hashtable containing the event ID, its complexity is $O(1)$, so there is only an iteration over one path. Therefore, the complexity of the superpath check is $O(|S|)$. Since the check needs to be done for each combination of endings, it is run $|S^E|^2$ times, which leads to a combined complexity of $O(|S^E|^2 \cdot |S|)$. However, there can be multiple shortest paths ($|S|!$) that must be compared ($(|S|!)^2$). This increases the theoretical overall complexity up to $O(|S^E|^2 \cdot |S| \cdot (|S|!)^2)$. But in practice, the check is still viable due to a much lower number of parallel paths.

7.3 GAME GENERATOR

In the previous version of the game generator (Section 6.6) any player can trigger the events. While this would, strictly speaking, allow an investigation of how well the approach scales with different game sizes, it would not respect different scenarios of how the player actions are assigned. For example, if every action can always be done by any player, they are always weighted 0 in regards to effort, so the pathfinding is relatively unguided and could explore lots of unnecessary states (see the first

Listing 1: Superpath detection algorithm

```

IsSuperPathOf(currentpath, comparisonPath) {
    if (currentpath.eventcount < comparisonPath.eventcount) return false;

    var tempCollection = currentPath.events;
    foreach (event in comparisonPath.events) {
        match = tempCollection.FindEquivalent(event);

        // Event found which is in the comparison path but not in the current path
        if (match == null) return false;

        // If an event appears in the comparison path multiple times it must be
        // matched multiple times so the current match is removed to prevent it
        // from being matched again
        tempCollection.Remove(match);
    }
    return true;
}

```

pitfall in Section 7.2.1). Therefore, the generator has to be extended in order to better investigate how well the balancing calculations scale.

For the extension, every action that opens a door is assigned to a specific player, so that only this player is able to trigger the event. Moving through the opened door remains possible for every player. Then, there are three different assignment scenarios that can be evaluated. In the *singleplayer* scenario one player has to open all the doors and trigger the game ending. All other players can stay in the starting location and do not need to do anything, which is very unbalanced. For the *block-based* scenario the game is separated into $|pl|$ continuous parts and every player needs to open the doors in one part. In order to trigger the ending all players need to move to the last location and press one button each, which instantiates the “Gathering Gate” game design pattern (Appendix C). The resulting games are balanced overall, but the events are badly distributed. Lastly, the *alternating* scenario is similar to the block-based one, but the players need to take turns to open the doors. This is also balanced in regards to the distribution.

7.4 SUMMARY

Players only feel treated fairly if a game is balanced. However, up to this point balancing definitions only focus on competition; accordingly, this work proposes a novel balancing definition with the focus on collaborative games. This definition considers the distribution of the effort required to solve the game, the waiting times involved and the options the game provides along the way between the individual players as well as for different sections of the game.

To assess whether or not a game is mathematically balanced, an extendable approach for obtaining concrete measurements has been devised. The approach works with any game that can be modeled as sequences of events between game states and

for any metric that assigns weights to the events. This is illustrated with concrete metrics for effort, balancing and options.

Since it is purely based on the game's model, the balancing can be calculated at an early design stage and without actual players. But this also implies that due to the uncertainty of not knowing how the players will actually play the game, the balancing can only be approximated.

In regards to the overall approach for authoring collaborative multiplayer games, two formal aspects (solvability and balancing) are covered in the last chapters. Nevertheless, there are still "soft" criteria that must be manually checked, for example, the game's visual consistency. For this, dedicated multiplayer tools are also beneficial, one of which is described in the following chapter.

RAPID PROTOTYPING OF COLLABORATIVE MULTIPLAYER GAMES

In order to facilitate the testing of collaborative multiplayer games a specialized rapid prototyping environment is conceptualized [92]. This environment allows single developers to test games developed for up to four players by re-organizing visual and auditory information and by providing a record and replay queue for simulating concurrent input.

8.1 REQUIREMENTS

In order to investigate the challenges that arise when testing a multiplayer game, two scenarios have to be considered. The first scenario is “local multiplayer”, i.e. all players share the same PC, using different input devices (or different sections of the keyboard). For the second scenario “network multiplayer” the players use different PCs which are connected over a network. We noticed during the development of several multiplayer games (for example [91] and [125]) that both types of games cannot be tested quickly without having enough test players at hand.

Hereby, the second scenario is the most problematic one. Running the game in its intended scenario would require the developer to set up multiple computers connected via a network. In order to play the game the developer would have to observe all monitors at once, to listen to all audio input at the same time and to switch between the devices to simulate actions from different players. An alternative is to run multiple instances of the game on a single PC, connecting to the other ones via the `localhost` network address. The user then has to switch between these instances to assume the role of a specific player, but again, this yields some important limitations:

- Setting up multiple instances and connecting them manually requires some time.
- Monitoring what all players see at a given time requires switching all game instances into the windowed mode and placing them next to each other on the user’s screen. Some games, however, are programmed to pause immediately when their window loses focus, in which case only one of them can show up-to-date information. Although the developers could temporarily change that, it would cost time and introduce a discrepancy between the tested version and the final product, which could have unintended side effects. Additionally, setting up these individual windows further increases the testing overhead.
- While visual information can be distributed on the screen, this is not possible for audio information. Therefore, when a sound is played there is no way to recognize from which game instance it originates from. Thus, the sound could have been heard by any player or even multiple ones and the information about who would receive the information in the final game is lost.

- Only the focused instance window receives input events, even if the others do not pause. Therefore, the developer must always select the right window in order to control a player. Aside from being cumbersome, this introduces an artificial delay when switching between players, which may be problematic in games containing time-critical elements. Furthermore, simulating concurrent user input is completely impossible using this setup.

In the first scenario the game is already designed to be played on a single machine, which typically has only one monitor and speaker setup. This circumvents the first three problems, but the issue of concurrent input remains. Although it might be possible in some fringe cases to issue commands for two players at once, for example when only a single button has to be pressed for each one, in most cases more complex input is required. In these cases it is mentally challenging to simultaneously input two complex sequences, or it can even be physically impossible, if two-handed input is required for each player.

Therefore a rapid prototyping environment specifically addressing these multiplayer issues is needed. This environment should make visual information for all players available at the same time and allow the user to discriminate which player would have received an audible information. It should also provide one user with the ability to control multiple players simultaneously. Despite of these features the testing setup should be as easy and fast as possible, avoiding any extra steps like organizing multiple game instances. The initial design of such a prototyping environment has been developed together with Tregel [116]. This first version is extended for this work by adding new features and refining existing ones.

In the following sections, specific modules of this environment will be described. Hereby we assume a scenario in which a single developer tests a game for multiple players. This means that there is only a single user – the developer – and that the players are not actual people, but merely roles between which he can switch.

8.2 VISUALIZATION

Generally speaking, the prototyping environment is supposed to provide lots of (visual) information that should be associated with different players. The relationship between information and player should always be clear, which can be done by color coding. That way a color is associated to each player and every information is marked with the color of the player it is linked to. To be useful this coding has to be used consistently in every module of the prototyping environment.

The most important visual aspect of a game is the view of players, i.e. what is displayed on their monitors. Here a split-screen approach [114], which is a seamless version of having multiple windows for each player, is used. For this, the actual screen area of the prototyping environment is separated into as many parts as there are players. This is done in such a way that the relationship between screen width and height for each section is as close as possible to the overall one (Figure 36).

A drawback of this method is that the size of the individual players “screen” decreases. Decreasing it too much might lead to some small details becoming invisible, in which case the player can also not interact with them anymore. Therefore, each partition should be kept as large as possible. Additionally, games which are implemented for a local multiplayer setting have already been designed for a single screen

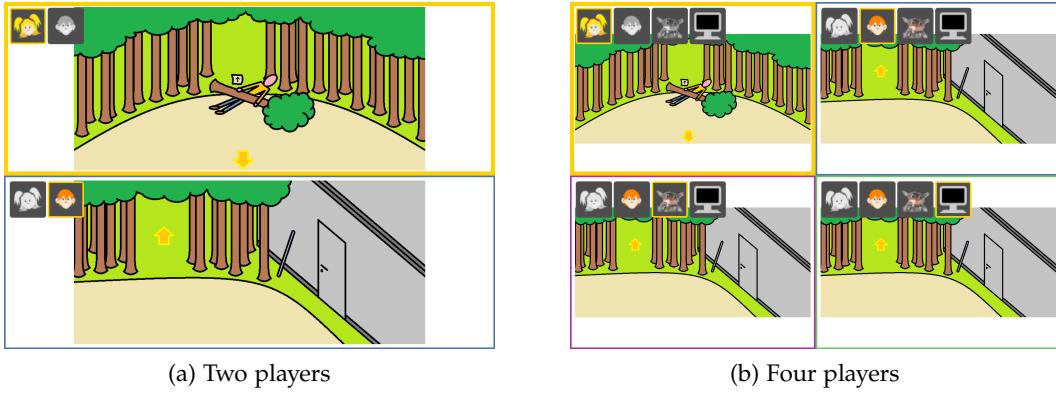


Figure 36: Different split-screen setups.

and therefore require no splitting at all. Furthermore, it is obvious that this approach only works for a comparatively small number of players – but this fits the application scenario (Section 3.1).

Aside from split-screen, one could display only one view using the full screen and give the developer the ability to switch between them. However, this would hide information, as important events could happen at one of the other screens, and would require additional switching effort by the developer.

8.3 INTERNAL ADAPTATION MODELS

In addition to the traditional player’s view the existing singleplayer prototyping environment (Section 4.3) also features a developer’s view. This view exposes the otherwise hidden internal adaptation models at runtime and is therefore especially useful for prototyping adaptive games. For example, these models include an assessment of the player’s preferred playstyle (player model). Using this information, certain games can adapt themselves to the player by selecting locations based on their appropriateness to that preference. The way these models are displayed is adapted for multiple players, too.

A naive approach would treat these models like the player’s screen, having one view for each player and arranging them in a split-screen setup. While universal, this method would make a direct comparison of several player’s values hard due to the individual values being in separate sections of the screen. It would also waste screen space by multiplying static elements like captions. Therefore, each model is handled individually, taking their specific properties into account. But although the models discussed are part of a specific adaptation approach developed by Mehm et al. [75], the way in which they are extended can easily be applied to similar models as well.

Extending bar charts like the player model (Figure 37) for multiple players is relatively straightforward. The single bar for each dimension in singleplayer can be split up into multiple smaller bars, one for each player. This facilitates comparisons between multiple players along the same dimension. Naturally, each player’s bar should follow the global color scheme.

For the knowledge space (Figure 38) a similar solution is chosen. This module displays the knowledge that the players have acquired while playing the game, which

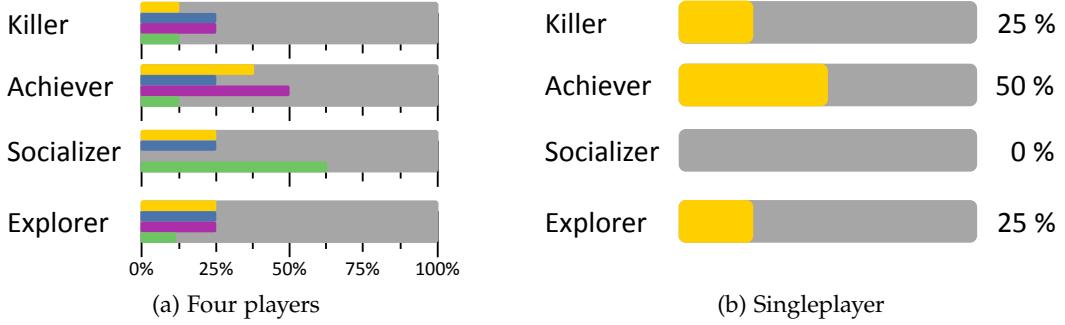


Figure 37: Player model comparison, both versions describe how well the player(s) fit the archetypes defined by Bartle [10].

is used for learning games. The knowledge is separated into individual topics whose dependencies are displayed as a graph. Originally the information on the probability with which the player has understood each topic was conveyed by the color of the node. Since the knowledge structure is static it can be kept unchanged. But the knowledge already acquired might vary between the players, so this information is split into a bar chart similar to the player model.

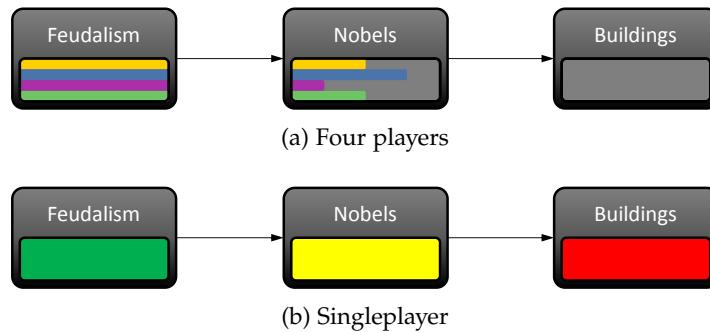


Figure 38: Knowledge space comparison, both versions describe a hierarchy of three topics that strictly follow each other.

Lastly, the location history (Figure 39), which records the player's path through the game, has to be adapted. In its singleplayer variant it not only displays the progress in a linear fashion, but also the alternative locations the player could have visited at each intersection. If the next location has not been selected by an explicit player choice but by the game adaptation algorithms [42], it is highlighted and the score that lead to this decision is displayed. Adapting this model for multiple players yields two challenges. Firstly, the players can take different paths through the game, splitting up and meeting again later. This increases the complexity of the resulting graph significantly, especially when alternative locations are displayed. Players can also change their location at different times – it could even happen that one player stays at one location while another hops between multiple locations at the same time. This temporal aspect is not included in the singleplayer model, but essential in multiplayer games. Therefore, a new model is needed which includes the temporal aspect and which does not become too complex when displaying multiple players in the same graph. It should also show when multiple players have been at the same location and keep information on the automatic adaptations at hand.

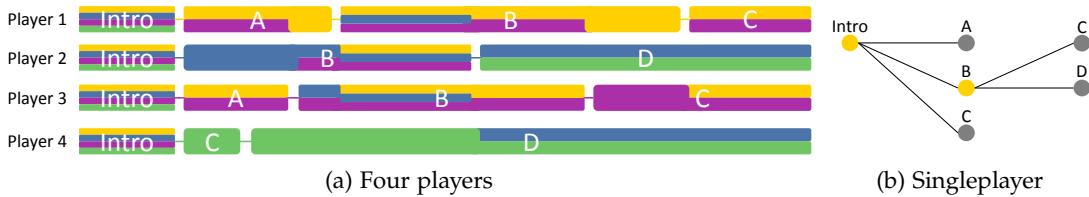


Figure 39: Location history comparison.

This new model organizes the information in a two-dimensional space. On the x-axis all players are lined up, with each row showing the game from this player's perspective. The x-axis displays the time. A player staying at one location is displayed by a box spanning the duration he occupies this location and a small gap signalizes this player moving to another place. This gap can become longer depending on the loading times of this player's PC. In the example (Figure 39) the "Intro" location is left by all players simultaneously, which can be seen by their gaps lining up. Additionally, if multiple players are at the same location the corresponding box is split into their colors, similarly to the bar chart. This way, the developer can easily find out where a player has been at a given time and who has also been there at the same time. In the example the first player has shared location "A" (second box) for some time with the third player, who leaves this location earlier.

In order to keep the graph from becoming too complex, the information on the automated adaptation is moved into an overlay. If such an automated location selection has happened the transition is marked by a dotted line. When the developer hovers the mouse over such a transition, all evaluated locations and their scores are displayed.

8.4 SOUND

In comparison to visual information which can be displayed next to each other, differentiating between multiple audio streams transmitted through the same speakers is much harder. Especially if the same sound is played for multiple, but not all players, it is impossible to know to whom it is addressed. On the one hand having a speaker setup for each player and diverting the audio to each one is not only impractical, but even then, audio would mix before reaching the developer's ear. On the other hand, playing only audio for one player at a time would hide information.

To solve this problem our concept adds an audio log (Figure 40), which displays every sound played with a timestamp and the players for whom it was audible. If available, a textual transcript (subtitle) is displayed, too. This way no information is lost, even if two sounds overlap. Additionally, every sound can be replayed by clicking on the corresponding entry if the developer wants to hear it again, for example when it has been overshadowed by another, louder sound. Aside from an overall log there are filtered versions for each player between which the developer can switch. If there are changes in a log that is currently in the background, its tab is highlighted in the corresponding player's color to prevent a loss of information. The sound output for each player can also be muted separately. This enables the developer to decide which audio information is currently most relevant, and prioritize if there are too

many different sounds being played at the same time. Due to the logging, however, no information is lost even if a player's sound output is muted.

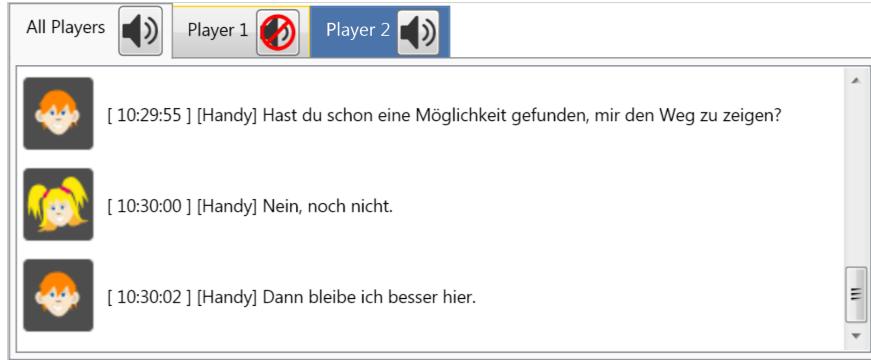


Figure 40: Sound log of a dialog (with a textual transcript). The first player is muted; the second's log has new entries.

8.5 INPUT

Another challenge for the prototyping concept concerns the fact that multiple players must be controlled by the same person. If the game is already designed for local multiplayer on the same device, there must already be a way to register the input from different players separately. But a network multiplayer game could for example assume that each player controls the game only with the mouse. Therefore, the prototyping environment must provide a way to emulate multiple inputs when started on a single PC, to which usually only one mouse and keyboard is connected.

For mouse input this is relatively easy. If the mouse hovers over one of the split-screen views, which can be calculated based on its screen coordinates, all mouse events (clicks, movement) are sent to the corresponding instance of the game. The only pitfall is that the coordinates must be converted for this, as a game normally would receive coordinates in a value range of $[(0, 0), (\text{Screen.Width}, \text{Screen.Height})]$ when running in a maximized window. Since the viewport of this player covers only a part of the prototyping environment, these coordinates must be translated and scaled based on the split-screen size and position for the game.

Pressing a key on the keyboard (or other input devices like gamepads) has no screen coordinates attached to it, which means there is no way to determine which instance should receive the event. Sending it to all instances would cause all players to execute the same action, which is usually not what the developer has intended. Therefore it is necessary that the user manually selects a game instance which is supposed to receive the keyboard events, for example by clicking on the split-screen border. The currently selected border can be highlighted to make it clear that it will receive the next key press events.

Nevertheless, the approach is still limited by the fact that it is impossible to send mouse or keyboard events to multiple game instances at the same time – there is no way to click two separate locations with one mouse. This would mean that the developer is not able to test situations in which multiple players act simultaneously. As some games use such situations in order to promote player coordination they should be testable, too, and a specific module must be designed for this use case.

8.6 RECORD AND REPLAY

Even if simulating multiple players acting simultaneously is not necessary, there is another application in which switching between multiple game instances is not sufficient. Some games contain tasks that must be solved in a given time. In this case a single developer impersonating four players would need four times the time he would need for controlling a single player. Therefore, this might be challenging, even if the time limit is generous for four players.

One solution is to manipulate the game time by allowing the tester to completely pause the game. He is then able to still perform actions, which are not sent directly to the game. Instead all input events are stored in a queue (Figure 41). This way, the developer can take as much time as necessary to switch between the players and to execute a sequence of inputs. The developer can also delete or reorder the events already recorded. All events are color coded to signal the player they are associated with. In order to keep the prototyping environment as generic as possible only abstract input events, mouse clicks and coordinates or key codes, are recorded. The environment has therefore no access to the information about which element the user clicked (if any). After resuming the game all events are performed instantly. Since the events must be processed sequentially by the game, it is sufficient to execute them one after the other as long as there is no delay (in terms of elapsed game time).



Figure 41: Record and replay queue containing mouse events for two players.

Another approach would be to simulate all but one player. For our use case, however, the developer wants to test specific actions since the game's solvability has already been verified (Chapter 6). Additionally, an efficient simulation that goes beyond randomly triggering input events has to be tailored for a specific game, which requires lots of additional effort.

But there is a pitfall when recording abstract input events: When many events are recorded together, the developer has to anticipate their results. For example, the user might execute an action as one player, which exchanges a button with another one. Due to the game being paused this does not change anything yet. After this he could then click on the first button, which is still visible and which is recorded as a click on an abstract coordinate. Once resuming, the actual result would instead be a click on the second button, as it has been exchanged directly beforehand. If the developer does not remember the effect of the first button this would result in unexpected, although correct, behavior. To combat that, the prototyping environment offers a callback with which the game can signal whether a player's possibility space has changed – in the example once the button has been exchanged. If such a signal is received, the tool immediately pauses execution again and asks the user if he wants to modify the pending executions based on the changed situation.

Games with strong interdependencies between the players causing many changes in the others' possibility space and a strict time limit are the worst case for this solution. Here, the game would assume a turn-based characteristic, which means that the developer takes turns recording inputs for each player while no game time passes due to the usage of the queue. If a large part of the game is time-based, for example in racing games, this can prevent the approach from being viable due to the frequent pausing required.

8.7 GAME INTERFACE

In order to make the prototyping environment compatible with games implemented in different programming languages, a common interface has been defined. These functions can be either implemented in a single game or directly in a game engine or authoring tool.

All functions are described in Table 3, the ones marked as "basic" being always necessary. Depending on which features should be used, only a subset of the functions has to be implemented. Optional functions like the sound transcript are not required, but can provide additional benefits to the user.

FUNCTION	FEATURE
Setting a canvas for the graphical output	Basic (required)
Triggering the main loop	Basic (required)
Triggering keyboard input events	Keyboard controls (required)
Triggering mouse input events	Mouse controls (required)
Starting multiple instances of the game	Network multiplayer (required)
Hosting and joining a game session	Network multiplayer (required)
Muting the sound	Sound (required)
Providing a sound event log	Sound (required)
Providing a sound event transcript	Sound (optional)
Pausing the game	Record and replay (required)
Signaling when a players options have changed	Record and replay (optional)
Updating a key-value-pair in a model	Internal models (optional)
Appending a key-value-pair to a model	Internal models (optional)

Table 3: Rapid prototyping API functions and the features they are related to.

8.8 SUMMARY

In this chapter, the concept for a rapid prototyping environment enabling a single user to test collaborative multiplayer games is described. This environment reorganizes visual and auditory information, originally intended for multiple players,

using a split-screen arrangement and an audio log. It also allows the simulation of simultaneous player actions via a record and replay approach.

In theory, this concept works for every game with one screen, audio output and event-based input devices. But in practice, its usage is limited to smaller games as described in the application scenario (Section 3.1). For example, when applied to larger player groups, each split-screen view would become too small. Additionally, it would take a long time to test a fast-paced game with many simultaneous interactions.

After the developer has tested the game and decided that it works as intended, the last step is to get feedback from members of the target audience. For this, actual players are needed and it is therefore not within the scope of this thesis' application scenario.

PRACTICAL IMPLEMENTATION

SEVERAL prototype modules are implemented as a proof of concept for the individual components described in the last chapters. This is largely done based on the existing authoring environment *StoryTec* (Chapter 4), which provides two benefits. Firstly, a repository of existing games created with the environment is available. Hence, existing real-world examples can be used as test cases. Secondly, basic functionality like parsing a game's structure is already implemented and can be reused.

9.1 GAME DESIGN PATTERNS

Since game design patterns are abstract descriptions, there is no definitive way to implement them. Instead, a two-step approach is necessary. First, the general concept is applied to extract a collection of patterns from existing games (Appendix C). Second, the patterns from this collection are used in several implementations ranging from abstract templates for the authoring tool to a complete game using the popular, industry-relevant game engine *Unity3D*¹.

9.1.1 Procedural Content Generation using Patterns

For the authoring tool, a procedural content generation module based on the game design pattern collection has been developed [117] (also Section 5.5). This module uses the structural properties of each pattern to generate a game structure consisting of connected pattern instances directly inside the authoring tool. Each pattern instance is also preconfigured in such a way that a concrete value is assigned for all variable properties of the pattern.

A prototypical evaluation showed that the procedural generation based on the patterns produces game structures that fulfill the developer's specification. Subsequently it was proven that the overall approach is viable (an example result can be found in Figure 42). But it was also found that some patterns appear relatively often due to the fact that they are the only ones that fit a specific combination of constraints. This means that the pattern collection should be extended to fully realize the potential of the algorithm.

Additionally, we investigated how the abstract design patterns can be applied to the genre of Point-and-Click Adventure games [102]. Almost all games in this genre are designed for a single player, which means that none of the games from which the patterns have been extracted are a part of this category. Therefore using the patterns in combination with the conventions of this genre is an interesting test case for their generalizability.

In a second step the resulting puzzle ideas are mapped to existing elements of the authoring tool. This way concrete design templates using these elements can be

¹ <http://www.unity3d.com>

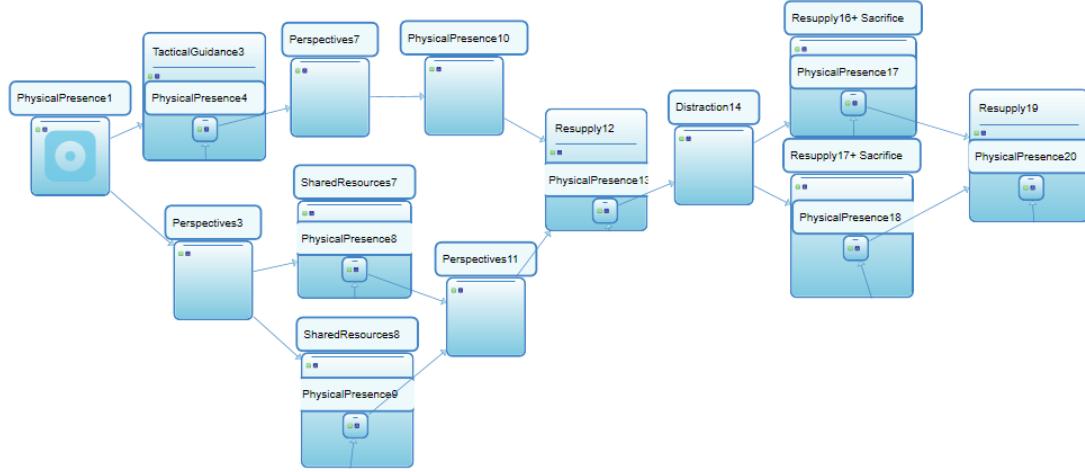


Figure 42: Procedurally generated game structure using interaction patterns. Each box constitutes the instance of a pattern, the arrows show the possible player paths through the game.

devised for each supported pattern. As a result, several patterns are found that can be implemented directly (“Parallelization”, “Separation Gate”, “Gathering Gate” and “Perspective”), with a few others requiring small extensions (“Protector”, “Trade”). However, one of the analyzed patterns could not be transformed into an interaction template without extending the underlying game model significantly (“Transport”), which may indicate that the interaction is unsuited for this type of games.

9.1.2 Evaluation Game

Additionally, a standalone game is developed with the popular game engine *Unity3D* to show that the usefulness of the patterns is not tied to a specific technology. The game is implemented as a local multiplayer game for three players. For the graphics a simple block-based style is chosen, similar to the popular game *Minecraft*². A camera control is designed that uses an isometric view and that zooms out if necessary to keep all player characters in view.

The original prototype [118] is extended with additional levels and is generally refined to improve the player experience. Its final version contains five levels that implement one collaborative interaction pattern each (Figure 43). For comparison reasons, the patterns requiring collaboration can be switched off, replacing each puzzle with a similar one that can be solved by a single player alone. To facilitate testing, this is even possible while playing the game. Additional features to support the evaluation, like logging, an enforced time limit per level as well as automated loading of the levels in the right order, are also implemented.

9.2 STRUCTURAL VERIFICATION

In order to implement the structural verification, the transformation rules for all supported game elements have to be implemented into the authoring tool first. This allows the generation of a matching colored petri net, which can be used for verifica-

² <http://minecraft.net>

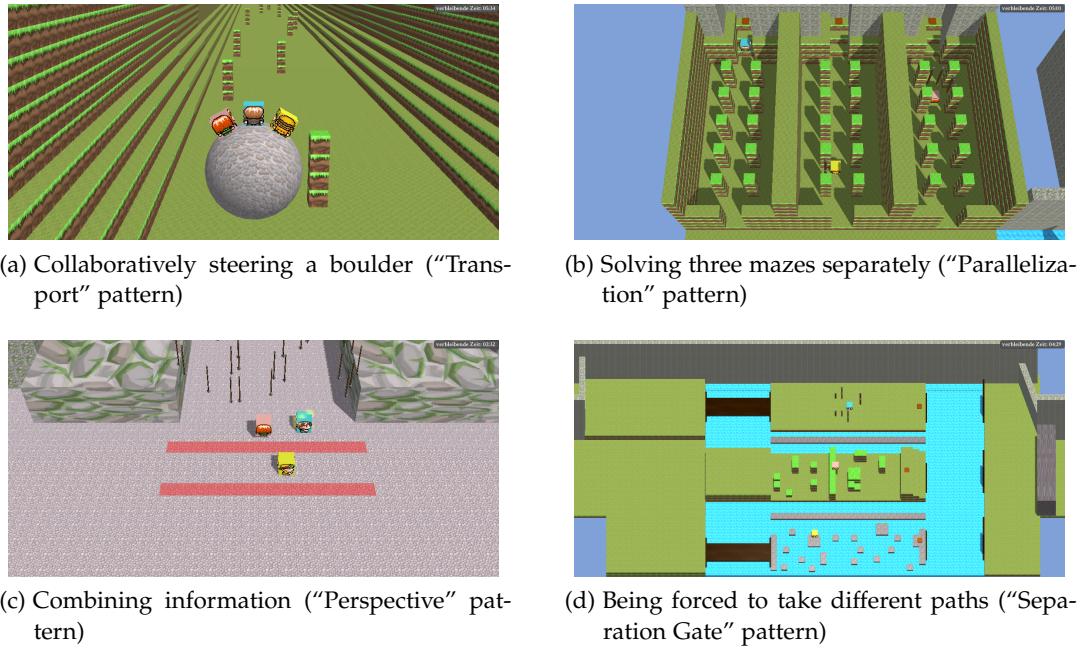


Figure 43: Pattern implementations in the evaluation game.

tion. Since there are established tools available for that, for example *CPN Tools* [54], the verification step is not re-implemented. Instead, an exporter is written which is able to export the petri net into a file format understood by the tool.

The transformation process can be influenced by several parameters. Most importantly, the developer can define whether the export should assume all event conditions to be exclusive or not. If this is not the case an additional transformation step (Section 6.3.3) for making them exclusive is triggered. The other parameters are for evaluation purposes only. One of them controls whether the resulting net should be optimized (Section 6.5). The other causes the exporter to use the alternate model described in Section 6.3.4 – if possible (finite value range of variables).

The process starts with parsing the game, during which references between game elements (for example events changing a certain variable) are collected. These relations can be used to efficiently optimize the game following the concept described in Section 6.5. Finally, the events can be linearized into single events for each path, as the branching tree structure used in the game model cannot be used in a petri net (Section 6.3.2).

9.2.1 Petri Net Generation and Verification

After optimization and linearization, the petri net is generated by applying the transformations defined in Section 6.3 to each game element. For verifying the resulting net, several existing tools have been evaluated: *CPN Tools*³, *TimeNET*⁴ and *CPN-AMI*⁵. While functionally similar, *TimeNET* requires the installation and configuration of an

³ <http://cpntools.org>

⁴ <http://www.tu-ilmenau.de/sse/timenet/>

⁵ <http://move.lip6.fr/software/CPNAMI/>

additional shell and *CPN-AMI* cannot be used as a standalone application. As this could overburden non-technical developers, *CPN Tools* is used.

In order to verify the generated petri net, it has to be exported in a way that can be understood by the tool. As it supports saving and loading petri nets in an xml-based format, this format is analyzed. Emulating the format is relatively straightforward, as every petri net element has a corresponding xml tag. All used variables and variable types are defined in a header and every element also needs a unique id. These ids which can be referenced to connect two elements, for example an arc to a place and a transition. Lastly, every element needs a position at which it is displayed in the visual model. As most elements have no dedicated position in the game world due to being abstract, there is no reference point for them. An optimal placement would result in a graph that is as planar as possible (no arc intersections), as this makes it easier to read the result. Such a placement is seldom possible though, especially in games that provide lots of options to the players. These events cause many arcs in the net that are connected to both variable-places and location-places, which in turn are often connected to multiple events. Instead a simpler approach is developed, which works for arbitrary petri nets while being clearly structured (Figure 44): On the left side all variable places are stacked vertically, in the example the boolean variable "EndGuard". The locations are arranged at the bottom, starting with the starting point, in this case "Outside". Further locations are unsorted, as the linear arrangement cannot match non-linear location structures, which are found in most games. In the example, there is only one more "Inside" location and the special "EndGame" place. All transitions are stacked above the location in which the events they model are located, e.g. "Event1" above "Outside".

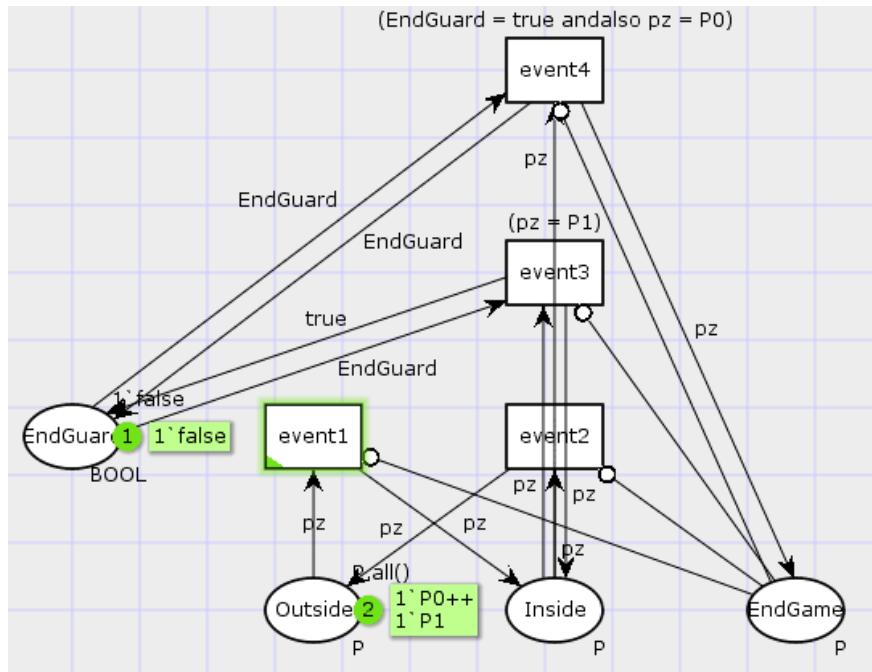


Figure 44: Exported petri net layout (as displayed in *CPN Tools*).

To verify the desired properties of the game in *CPN Tools*, the state space graph of the net must be calculated first. This graph contains all states that can be reached from the initial token placement. Then, the presence of dead markings, i.e. token

placements in which no more transitions are enabled, must be checked. Since markings symbolize game states, a dead marking corresponds to a game state in which there are no more options available to the players. Due to the use of inhibitor arcs (Section 6.3.3) this should be the case when the game has ended. This means that, if there are no dead markings at all, the ending can never be reached. If there are dead markings the token placement in these states must be checked, specifically whether a player token is present at the “end”-place. If there are only dead markings without a token at this place, the ending can never be reached and the states found are deadlocks. In contrast, the game has no deadlocks if there are only dead markings in which a player has reached the end. A mix of both means that there are some paths on which the ending can be reached, but also others that end in deadlocks.

But even if the ending can be reached without deadlocks, another check is necessary for livelocks. For this, it must be verified whether the dead markings calculated in the last step build a homespace. This means that from every other reachable marking at least one of them is reachable. Translated into the game domain, this means that at least one ending can be reached from every possible game state. If this is not the case there must be at least one livelock (or deadlock, but those are already checked before). If the dead markings are a homespace, however, the game is always solvable and there are no livelocks.

Unreachable locations can be found by looking at the bounds of each place, which describe the minimum and maximum number of tokens which have been present there. For unreachable locations the corresponding place has an upper bound of zero tokens. If a player had been there during the exhaustive search through the state space, this number would be higher. Linearized events that cannot be triggered in the game appear as dead transitions in the petri net, which have never been fired during the simulation.

To make the verification as easy as possible, the commands for each step are already prepared during export and saved alongside the petri net so that they can be executed with a single click.

9.3 COLLABORATIVE BALANCING

The collaborative balancing analysis is also implemented into the existing authoring tool. First, every shortest path through the game needs to be calculated. This is done on the state space, which has already been calculated during the structural verification (Section 9.2.1). The result of this calculation can be exported from the petri net tool with a prepared command, and imported into the authoring tool (Section 9.3.1). To find the shortest paths, multiple pathfinding algorithms (Section 9.3.2) have been implemented. An alternative path discovery method using a planning algorithm on the game itself was also implemented [46]. When relaxing the task by accepting slightly suboptimal paths, this alternative approach was also viable.

As soon as the minimal paths have been found, the balancing calculations are done. For this step all events must be weighted. In order to avoid assessing each event manually, several weight heuristics have been implemented (Section 9.3.3).

Similar to the structural verification, several parameters are added to allow the comparison of different variants. This includes choosing between multiple pathfinding algorithms as well as weighting heuristics. To trade runtime against accuracy, the

reference path for the whole group can be left out or all metrics can be calculated on a single path optimized for effort (Section 7.2.2). Removing end states with irrelevant changes (Section 7.2.6) is also optional. The number of sections when calculating the balancing distribution (Section 7.2.4) can be configured, too.

9.3.1 State Space Import

CPN Tools, used in the verification step, offers multiple ways to export the state space it has calculated (Figure 45). One possibility is to export the graph into a format used by a tool called *Graphviz*. This format is also used for the balancing approximation, as it contains all necessary information and can be parsed easily. Since the graphs are originally meant to be read by humans, they are limited in complexity and the graph can either be exported as connected nodes without details or as unconnected but detailed nodes. This means that in order to import the full state space, the results of both functions must be combined.

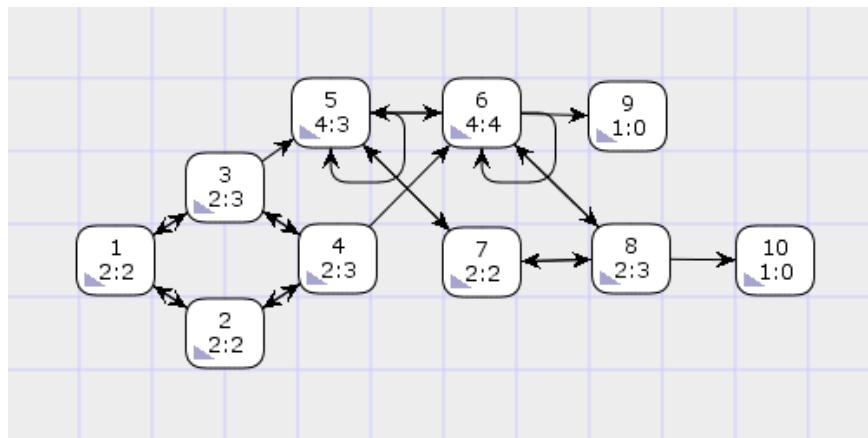


Figure 45: Example state space for the petri net in Figure 44 (as displayed in *CPN Tools*).

The detailed node descriptions are exported in the following format:

```
N<id> [label=<id>:  
<netName>'<place0Name> 1: <tokens>  
<netName>'<place1Name> 1: <tokens>  
...  
"];
```

Since the node description refers to petri net elements, a reverse mapping of petri net elements to game elements is required. This is facilitated by the authoring tool already containing the mapping from game elements to petri net elements. However, places can either correspond to a location or to a variable, which means that each game element needs a globally unique name.

As an illustration, a real node can look like this:

```
N15 [label="15:  
MultiplayerAdventure'Outside 1: empty  
MultiplayerAdventure'Hall 1: 1'P1  
MultiplayerAdventure'Roof 1: 1'P0
```

```

MultiplayerAdventure'EndGame 1: empty
MultiplayerAdventure'DoorOpen 1: 1'true
MultiplayerAdventure'GeneratorOn 1: 1>false
...
"];

```

In this state, the first player p_0 is at the location "Roof", the second player p_1 at the "Hall". No player is at the "Outside" location and since the special place "EndGame" is also empty, the game is still running. The variable "DoorOpen" is currently true, "GeneratorOn" false.

The full graph is exported as follows:

```

N<id0> -> N<id1> [ label="<ids>:<netName>'<eventName> 1:
    {<triggeringPlayer>,<variableConditions>}" ];
...

```

This symbolizes a directed arc (transition) between two states whose IDs can be matched to the detailed descriptions. The curly braces encapsulate the variable binding on the triggered petri net transition, which contains the player that triggered the event.

To give an example:

```

N1 -> N2 [ label="A1:1->2:MultiplayerAdventure'event1 1:
    {pz=P0,DoorOpen=true}" ];
N14 -> N11 [ label="A45:14->11:MultiplayerAdventure'event2 1:
    {pz=P1}" ];
N14 -> N12 [ label="A44:14->12:MultiplayerAdventure'event2 1:
    {pz=P0}" ];
...

```

The first entry shows that a transition from node 1 to 2 happens when "event1" is triggered, which can only be done by the first player p_0 while the variable "DoorOpen" is true. For "event2" there are two entries, which means that in node 14 either p_0 or p_1 can trigger it (resulting in a different game state). Which players are able to trigger an event at a given state can be directly derived from this information, in this case $\text{avail}_{P1}(\text{event2}, N14) = \{P0, P1\}$.

9.3.2 Pathfinding

After the state space has been imported the fastest way to each ending must be calculated. For comparison reasons two pathfinding algorithms are implemented, both the well-known Dijkstra's algorithm and its faster A* extension. This variant uses a heuristic to guide the search in promising directions. The heuristic has to be admissible, i.e. its value must be lower bound for the remaining distance (summed weight of events) to the goal. If the cost of each event is never lower than 1, the remaining number of transitions is such a valid heuristic. This information can be calculated once per goal by incrementing a step counter while doing a breadth-first search on the graph, starting at the end node. While this pre-calculation takes time, using the more directed A* can speed up the following path calculations, especially when the graph spreads out into multiple directions. As it only depends on the goal,

it also can be reused for each player and therefore has less impact the more player paths have to be explored.

Additionally, an extension to find all shortest paths (Section 7.2.5) has been implemented. However, there can be a large number of paths that have the same length. If this number grows, it becomes impossible to keep all paths in the main memory. In this case they must be temporarily swapped out of the memory onto the much slower harddrive, which increases runtime significantly. Therefore an upper bound on the number of parallel paths that are explored, is added, which currently is a hundred paths per ending. This value has been determined by experimenting with the most complex game available on a computer with a main memory of eight gigabyte.

9.3.3 Weight Heuristics

If the developer does not want to manually assign weights to every event of the game, an automatic fallback has to be implemented. Here, three heuristics with a different level of sophistication have been developed to investigate how much impact the heuristics can have on the result.

STIMULI can be used to count how many events each player has to trigger, therefore every event is counted with a constant value of one:

$$\text{weight}_{\text{stimuli}}(\text{lev}, s) = 1 \quad (7)$$

This makes the heuristic independent of the current game state.
REACTIONS takes the complexity of the game's reactions into account:

$$\text{weight}_{\text{reactions}}(\text{lev}, s) = |\text{reactions}(\text{lev})| \quad (8)$$

This heuristic is also independent of the current state.
RELEVANCE tries to additionally estimate how important these reactions are. It assumes that changing variables is most important, as they can influence the future actions of another player. Moving to another location changes the game state, but usually does not impact other players. Actions not changing the game state are ignored completely. To reflect the importance of variable changes they are counted with a higher factor. For example, this factor could consider how often a changed variable is read in other events. In turn, when playing a sound that other players are able to hear its relevance could be weighted with more than 0, even though it does not change the game state. However, there is no "right" or objective weighting between different actions and it also cannot be optimized as there is no known target value due to the overall process being an approximation anyhow. Therefore, getting a quick estimate based on a constant factor of ten for variable changes and zero for actions not changing the game state is prioritized instead.

$$\begin{aligned} \text{weight}_{\text{Relevance}}(\text{lev}, s) &= \sum_{r \in \text{reactions}(\text{lev})} \text{weight}^R \text{elevance}(r) \\ \text{weight}_{\text{Relevance}}(r) &= \begin{cases} 10 & \text{if } r \text{ changes a variable value;} \\ 1 & \text{if } r \text{ moves the player;} \\ 0 & \text{else.} \end{cases} \end{aligned} \quad (9)$$

9.4 RAPID PROTOTYPING

Similar to the verification and balancing being implemented as modules for the authoring tool, the rapid prototyping environment is built as an extension of the existing singleplayer runtime environment. The overall arrangement of the different modules can be seen in Figure 46. On the upper left side is a split-screen view showing the different players' perspectives (Section 8.2). To save space, the views are implemented in a flexible manner, which means that there are only as many parts as absolutely necessary for the current game. Hence, each view can be kept as large as possible. The record and replay queue (Section 8.6) is located directly below, which minimizes the distance between the views on which mouse input is executed and the pause button of the queue. The sound transcript is located at the bottom and – like in the singleplayer version – the multiplayer visualizations of the internal models are shown on the right side.

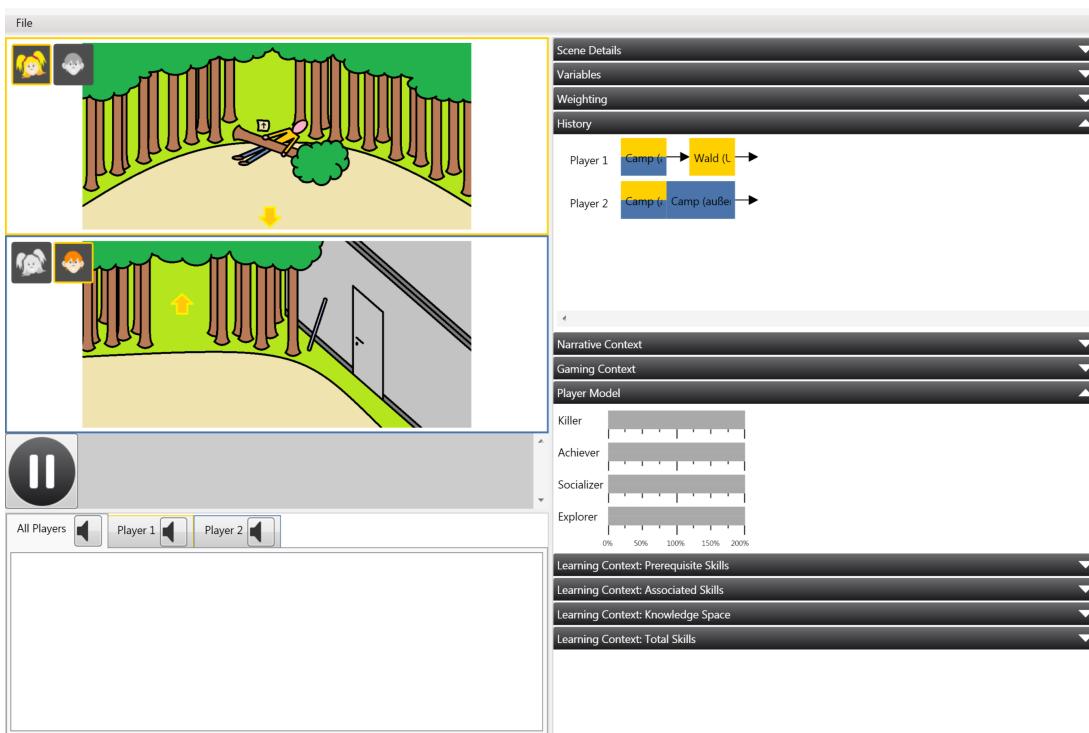


Figure 46: Multiplayer rapid prototyping environment.

In order to reduce the testing overhead, the environment itself starts the game multiple times and establishes network connections among these instances. Alternatively, since each instance is running on the same PC anyhow, the network setup could be skipped completely. If all game instances were running in the same executable, local function calls could be used instead. However, in most cases this would require a substantial modification of the game code itself. Therefore, setting up network connections to the same machine makes the environment compatible to more games.

Finally, the API definition (Section 8.7) is implemented in such a way that it can be accessed via C# or C++ libraries, which are the most commonly used languages in PC and console game development.

We evaluate all four modules of the multiplayer authoring framework to show that our approach is viable. In order to get qualitative player feedback on the game design patterns, we conducted a comprehensive user study on an example game that implemented a selection of the interactions patterns. The structural verification and collaborative balancing modules are both evaluated in a simulative manner. This means running a large number of calculations on both real examples and synthetically generated games in order to investigate their correctness and scaling behavior. In turn, the usefulness of the rapid prototyping environment is evaluated in a small-scale, qualitative user study.

10.1 GAME DESIGN PATTERNS

The main goal of the game design pattern evaluation is to get qualitative feedback on how well the collaborative player interactions that are described in the patterns are received by players. Although they have been picked from successful games, these interactions need to work in other games as well in order to be useful as patterns. This leads to the following leading questions for this part of the evaluation:

1. Are the collaborative player interactions described in the patterns still engaging after being applied to another game?
2. Do players prefer having these collaborative interactions or do they prefer to play alongside each other without interacting directly?

Aside from the overall enjoyment, other effects of the collaborative interactions are investigated as well:

3. How does collaboration impact the perceived difficulty?
4. Does the need for coordination increase play time?
5. Do collaborative interactions increase the amount of game-related player communication?

The patterns' adequacy for developers has already been evaluated indirectly, as the original game design has been developed by a third party and based solely on the pattern list. Afterwards, the developer of the patterns has checked whether the game design still reflected the intention of the patterns. This is true for all levels.

10.1.1 *Data collection*

The data required to answer these questions has been collected on three levels: a questionnaire for the players' self-assessment, a logging module to record game events and a manual protocol for data not collected by the other methods, for example verbal player communication.

The questionnaire is based loosely on the interaction and teamwork questionnaire developed by Wendel [124], but is tailored towards the goal of getting feedback on individual interaction patterns. It contains identical question blocks for each individual level, which ask the players about their enjoyment, contribution (self-assessment and others), helpfulness (self-assessment and others) and perceived difficulty. Additionally, players are asked whether they have enjoyed the game in general, are satisfied with their fellow players, would have liked to play the game alone and wanted to be first to finish each level. Lastly, there are some questions about the players themselves, which ask about their experience and preferences in regards to playing and collaboration as well as some demographic data. Almost every question is formulated as a statement on a 5-point Likert scale (agree - disagree). For difficulty, a 5-point scale is also employed, but since a game can be either too hard or too easy, it is two-sided (too easy - right - too hard). Age and gender are free-text fields. Introducing multiple variants of each question has been considered in order to assess the consistency between answers. However, since the amount of questions is already high due to the individual level blocks, we decided to ask every question only once. The full questionnaire (original German version as well as an English translation) can be found in Appendix E.

The game's logging module has collected individual events, annotated with a time stamp. These include entering and solving a level to calculate the playtime and player's deaths (and the cause), which count as mistakes. If possible, the act of interacting with certain puzzles and spatial progress (entering / exiting certain areas) has also been tracked in some levels as a contribution. Additionally, whether a player has pressed any button (made a contribution) and the players' positions have been recorded continuously in 1 s intervals. From this position data, the distance between players, and between them and the level exit can be calculated.

For the protocol, the following events have been noted per game level:

- Helping (distinction between active and verbal only)
- Encouraging other players
- Positive and negative feedback towards other players
- Positive and negative feedback about the level
- Positive and negative feedback about the game in general
- Communication about the game (on-topic)
- Communication in general (off-topic)

The previously stated questions are mapped to the gathered data as displayed in Table 4. While no further data is needed to answer these questions, it has been recorded nonetheless in order to facilitate future work.

10.1.2 *Setup*

The evaluation was conducted in part at TU Darmstadt and in part at AVM Rüsselsheim. At both locations, the same hardware setup was used – three XBox 360 Gamepads with the game being displayed on a projector screen. Each time, two researchers were present, one focusing on taking notes while the other one introduced the game and was available for questions.

At the beginning of each experiment, participants were greeted and a standardized introduction was given. This introduction included a basic explanation of the game

QUESTION	QUESTIONNAIRE	LOGGING	PROTOCOL	LEVEL	OVERALL	COLLABORATIVE	NON-COLLABORATIVE
1 Interactions engaging?	X			X X	X		
2 Collaborative preferred?	X			X X	X X		
3 Difficulty?	X			X	X X		
4 Playtime?		X		X	X X		
5 On-topic communication?			X X	X X	X X		

Table 4: Mapping of the evaluation questions to the data gathered.

and the overall evaluation process. At certain points during gameplay, short hints were given whenever a new game element was introduced.

It was decided to let the players play both the collaborative and the non-collaborative version of the game in order to increase the sample size. This required the order of these versions to be varied between groups to mitigate learning or priming effects. Additionally, all questions regarding individual levels and the overall game had to be duplicated to gather information on both versions separately. The game was played completely before switching modes, as pre-tests had shown that playing each level in both variants would have been more confusing and would have resulted in a worse player experience.

However, while the collaborative version enforces teamwork, the non-collaborative version does not enforce selfish behavior. This means for example, that the players can still wait for each other if they want to. While a valid behavior, we wanted to make sure that the players did so only by choice and not by falsely assuming that they still had to collaborate. Therefore, it was explicitly stated at the start of each level whether they had to reach the goal together (collaboration) or could play on their own (non-collaboration). Otherwise, players who played the collaborative version first could have assumed that they had to do the same in the second run.

After every level, the corresponding set of questions was answered by each participant. Overall, each session took 30-60 min (45 min on average).

To make the evaluation more efficient, the game automatically selected the next level and whether it should be collaborative or not, based on the previously defined rules. In order to have all groups play every level of the game, it was also decided to include a hard time limit of 5 min per level. If this limit was reached, the game automatically stopped the current level and marked it as failed. After a level ended, regardless of whether it was lost or won, the game stopped and only continued after it was unlocked by one of the supervisors. This way, the players could focus on the questionnaire and were not distracted by the game running.

10.1.3 Results

In total, 93 participants (31 groups) took part in the evaluation. From these, 4 participants cannot be considered for the final results, as they had played the game before and are assumed to be biased. Out of the 89 remaining participants, 74 are male and 15 female, with an age between 18 and 37 (24.12 on average).

As already mentioned, each group played either the collaborative version of the game first (16 groups), or started with the non-collaborative version (15 groups). In order to detect effects which are caused by the order of play, both of these variants are analyzed separately. To keep graphics and tables consistent, the collaborative results are mentioned first in both cases.

Each of the measured values is investigated independently, as this is sufficient to answer the previously stated questions. For example, it is only checked whether the game is fun in general, but not whether the abilities of other players have an impact on their enjoyment. First, the mean results for the collaborative and non-collaborative version of each level are analyzed individually for each question. This yields information on how the values are distributed over all players and groups. After that, a paired samples t-test is done (two-sided, confidence level 95%). This test takes into account that the same players played both versions and that, consequently, the samples are dependent. The P value that this test calculates allows the assessment of whether the observed differences in the mean rating between the collaborative and non-collaborative versions are statistically significant.

Looking at the answers to the “enjoyment” questions in the questionnaire (Figure 47; full data including P values in Appendix F, Table 22), it can be noticed that both versions of the game are perceived as fun (on average about 4 out of 5, higher means better). This is not only true for the overall game, but also for each individual level. Therefore, we can conclude that the version including the collaborative interactions is engaging for the players (Q1).

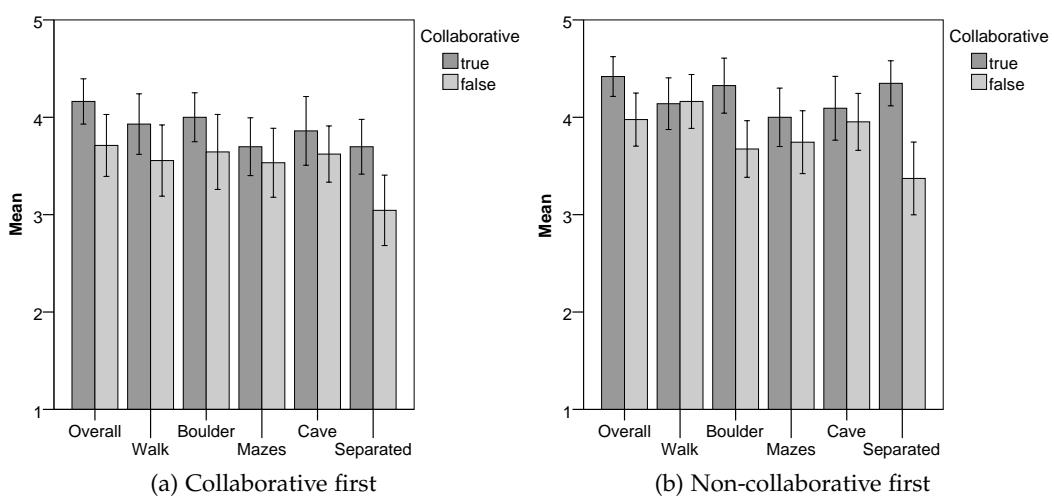


Figure 47: Questionnaire results for the “enjoyment” questions about each level and the overall game (error bars: 95% confidence interval).

However, it is also important to analyze whether the collaborative versions are rated higher than their non-collaborative counterparts (Q2). This is true for almost all

combinations of levels and playing orders, only the “non-collaborative first” players rated the first level a bit lower when playing it collaboratively (Figure 47b, “Walk”). This anomaly could be explained by the level being very similar in both versions. Therefore, when the level is played for the second time, there is nothing new to discover – which might negatively impact the players’ enjoyment. If the collaborative version of “Walk” was slightly more fun, as in the other levels, this lead could be consumed by such a familiarity effect if the collaborative version is played second. The t-test confirms that some of these findings are significant for some of the levels. Regardless of which variant is played first, the overall game is enjoyed more when being played collaboratively (highly significant, $P < 1\%$). The collaborative version of the “Separated” level is also enjoyed more in both orders (highly significant, $P < 0.1\%$). Lastly, the collaborative “Boulder” level is more popular for players who play the non-collaborative version first (highly significant, $P < 0.1\%$).

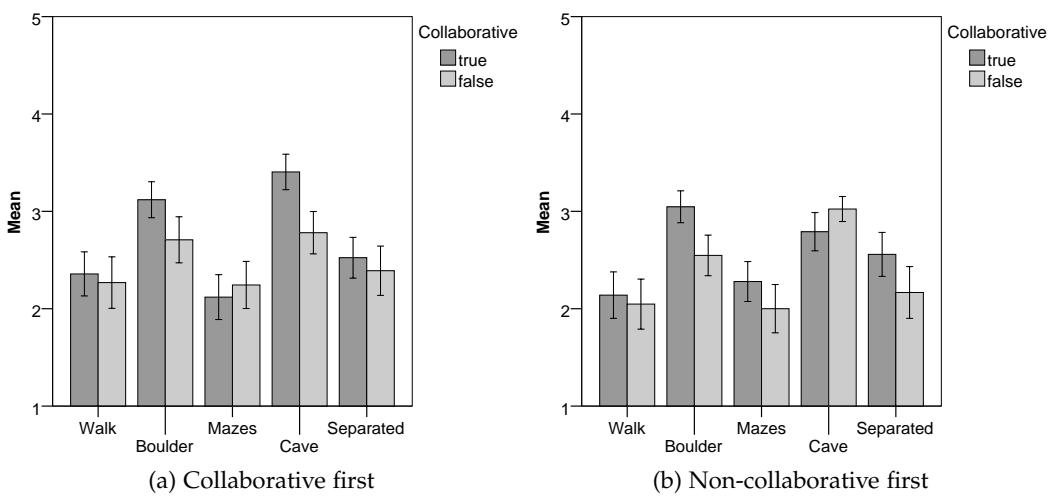


Figure 48: Questionnaire results for the “perceived difficulty” questions about each level (error bars: 95% confidence interval).

The collaborative versions of the levels are perceived to be more difficult in general (Q3; Figure 48; full data including P values in Appendix F, Table 23). This can be explained by the additional coordination effort required, which also introduces more opportunities for mistakes. But there are two exceptions for which this is not true. When playing the collaborative version first, the “Mazes” level is deemed harder when played non-collaboratively later (Figure 48a). However, when starting non-collaboratively, this level is described as more difficult in its collaborative version (significantly, $P < 5\%$). Starting in the non-collaborative version, players find the collaborative version of the “Cave” level easier afterwards (Figure 48b). The reason for this might be that the level uses the novel and complex game mechanic of detecting objects with the controller’s rumble functionality, which must be understood first. Additionally, the path that the players have to find is the same in both versions. This means that there is an initial barrier when playing the level for the first time, while it is much easier the second time. Playing non-collaboratively first, this “understanding” difficulty seems to dominate the coordination effort of the second run. This finding is also significant ($P < 5\%$). But when starting with the collaborative version, these effects amplify each other. This results in the collaborative version of the “Cave”

level being harder when played first (highly significant, $P < 0.1\%$). Lastly, the "Boulder" level requires a high amount of coordination, too. Compared to "Cave" though, it is mechanically simple and can be understood immediately. Therefore, there are no learning effects and the collaborative version of the level is significantly harder, independent of the order in which both versions are played (both times $P < 1\%$).

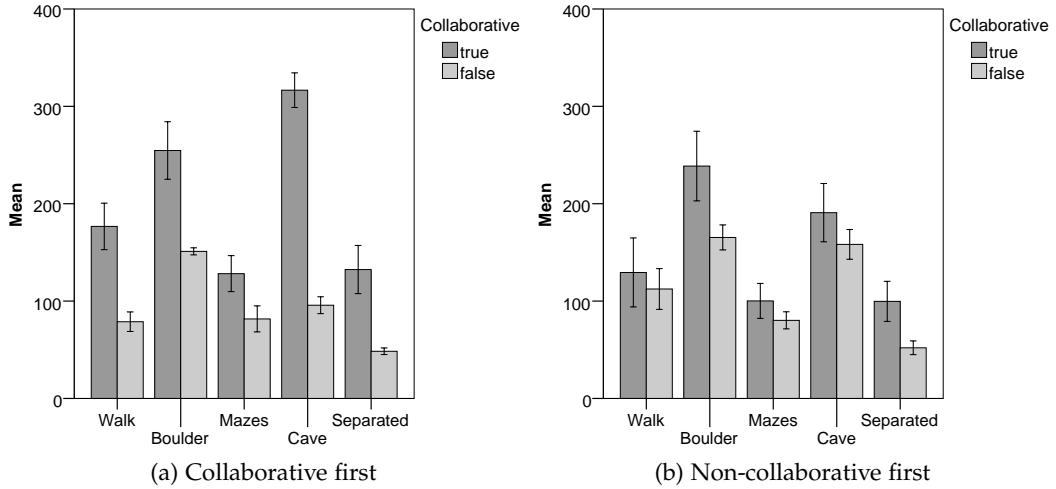


Figure 49: Logging results for the time the players took to solve each level (error bars: 95% confidence interval).

Similar to the perceived difficulty, the playtime also rises when playing collaboratively (Q4; Figure 49; full data including P values in Appendix F, Table 24). In general, this is not surprising, since in this version every player has to reach the ending. Therefore, the slowest player defines how much time the group takes – compared to the non-collaborative version, which ends once the fastest one has reached the exit. This result holds for both orders in which the levels have been played, but starting with the non-collaborative version reduces the difference (both graphs in Figure 49, especially "Walk" and "Cave"). Again, this is caused by the time that it takes to understand the level and the coordination time adding up (collaborative first) or being distributed between both runs (non-collaborative first). When the collaborative version is played first, the difference between both versions is highly significant ($P < 0.1\%$ for all levels). Starting non-collaboratively, the difference is still highly significant for "Boulder" and "Separated" ($P < 0.1\%$) and significant for "Mazes" and "Cave" ($P < 5\%$). Overall, only 6 out of 153 levels were aborted because the players took more than 6 min to solve it (2x "Boulder" and "Cave" collaborative, 1x "Walk" and "Boulder" non-collaborative). One should note that the levels end for all players simultaneously, therefore there is only one playtime measurement per group and level instead of three.

To numerically assess the amount of "on-topic" communication, the informal protocols have to be pre-processed first. This is done by quantizing them into events. For example, a communication event is noted when the players started to talk. Once they stopped for a few seconds, this event is finished and a new one is counted if they started again. In case they switched topics, a new event is counted immediately. Since there are multiple people required for communication, this value is also noted on the group level. Again, it is universally true that the collaborative

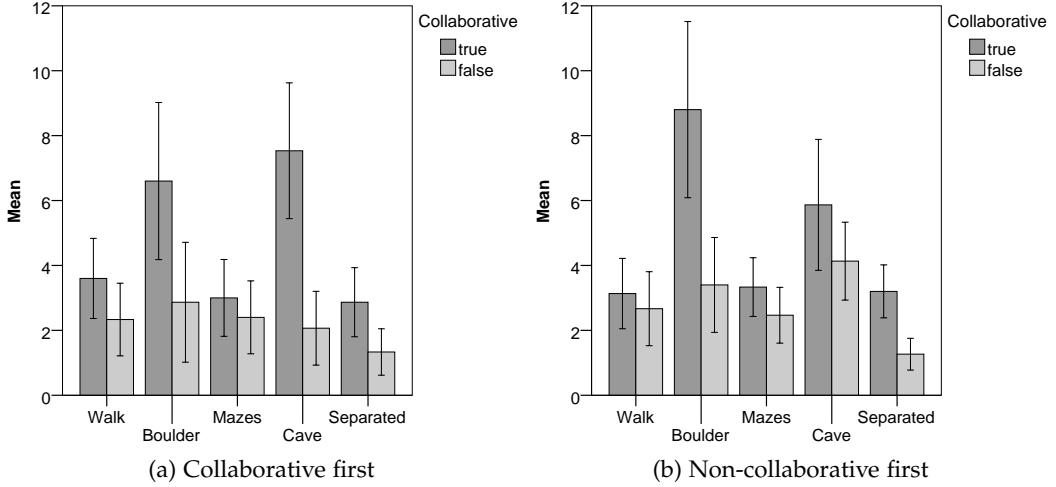


Figure 50: Protocol results for the amount of “on-topic” communication in each level (error bars: 95% confidence interval).

levels require more communication (Q5; Figure 50; full data including P values in Appendix F, Table 25). When playing collaboratively first, the “Boulder” and “Cave” level require the highest amount of communication. This has to be expected, as these levels have the highest dependency between the players. However, when playing non-collaboratively first, the difference in “Cave” is reduced greatly, while it remains relatively stable in “Boulder” (both graphs in Figure 50). This can be explained by the observed type of communication: “Cave” requires the players to share information, which is the same between both runs. If they have played non-collaboratively first, most players remember at least parts of this information and do not have to ask their peers again. “Boulder” in contrast requires the players to time their movement, which is necessary even if they knew the level already. In the collaborative-first-run the increase in communication is highly significant for “Cave” ($P < 0.1\%$), “Boulder” and “Separated” (both $P < 1\%$). It is still significant for “Walk” ($P < 5\%$), too. When the non-collaborative version is played first, it is only highly significant for “Boulder” and “Separated” (both $P < 0.1\%$).

10.2 STRUCTURAL VERIFICATION

The structural verification consists of two steps that must be evaluated separately. First, there is the transformation of a given game to a petri net. This step should be completed quickly while also exporting valid nets that are as small as possible. In the second step, the net is verified, which should also happen in a reasonable time span. The length of this time span depends on the scenario and ranges from a few seconds for rapid prototyping to overnight verification when the game is close to completion. Since the verification time depends on the size of the net exported in the first step, it is also an indirect measure for the export quality. It is also integral to the approaches from the users point of view, as the verification is necessary in order to get results. Therefore, the second step is evaluated as well, although it is handled by an external tool. If not otherwise noted, 2 min are taken as the cut-off point for both steps. While a few minutes more could yield a result in some cases, the trend

clearly shows that another slight increase in game size would multiply this value (state space explosion).

Three categories of games served as use cases:

- **Toy examples** containing intentional errors to check whether the verification is working correctly.
- **Synthetic examples** (Section 6.6) to investigate the approach's scaling properties in relation to game size.
- **Existing games** previously created at our lab as practical examples.

All calculations are conducted using a Intel Core i7-4790K CPU (4.00 GHz, 8 logical cores) and 16 GB RAM on Windows 8.1 Pro (64 bit). The complexity reductions are given as percentage values that have been rounded down. This is done due to 99.6% being displayed as 100% otherwise, which would be misleading as there are still some elements left.

10.2.1 Toy Examples

As the first test case for the structural verification, a game containing a single puzzle has been created (Figure 51). In this example one player has to boost the other one into a vent, allowing him to reach another room. From there, the second player has to retrieve a key, which can be brought back through the vent by using a ladder. After using the key in the first room, the game ends.

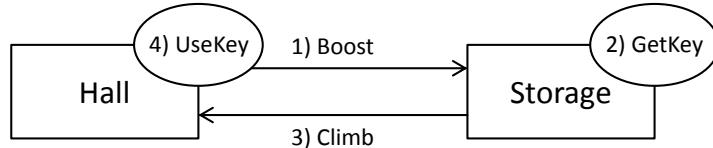


Figure 51: Example scenario: Boosting another player to retrieve an object.

While being solvable, the example can be easily modified to include each one of the error classes that need to be detected (Section 6.1). For example, removing the ladder causes a *deadlock* as both players are stuck in their rooms, unable to reach the vent without help. Splitting one of the rooms into two sections between which the players can move converts this deadlock into a *livelock* – the game ending cannot be reached but the state can always be changed by moving. Lastly, an optional location without an event for moving a player there (*unreachable location*) and an event with an unsatisfiable condition (**impossible event**) have been added.

For this minimal example both the export and the verification take less than 1 s. Independently of the chosen parameters, the indicators for each error class (Section 9.2) can be observed as expected.

10.2.2 Synthetic Examples

First of all, one specific configuration for the extended generator described in Section 7.3 is designated as a baseline. It describes a game for two players and consists of eight mandatory locations and no optional ones. The task the players must solve

in order to progress is opening a locked door. In order to have both players acting, these tasks are assigned to the players in an alternating fashion.

10.2.2.1 Export Parameters

To investigate the effect of different export parameters, four generator configurations are derived from this baseline. On the one hand, different lengths are sampled (8 and 32). On the other hand, the task type is varied between “locked door” and “puzzle”.

LEN.	EXCL.	OPT.	PLACES	TRANSITIONS	ARCS	EXP.	VER.
8	Yes	Yes	19 66%	26 0%	114 31%	0:00	0:03
		No	54	26	166	0:00	0:04
	No	Yes	19 66%	26 40%	114 58%	0:00	0:03
		No	54	44	274	0:00	0:04
32	Yes	Yes	67 64%	98 0%	426 31%	0:00	0:12
		No	198	98	622	0:00	0:20
	No	Yes	67 64%	98 40%	426 58%	0:00	0:12
		No	198	164	1018	0:00	0:28

Table 5: Verification results for the generated examples (export parameters for “locked door” task). For places, transitions and arcs the reduction that can be achieved by the optimization is given in percent.

For all samples using the “locked door” task, the optimization is able to remove some of the complexity (Table 5). These savings are between 64% and 66% of the places for all cases. The reduction in transitions and arcs depends on whether the net is exported assuming non-exclusive conditions. If this is the case, 58% of the arcs and 40% of the transitions can be removed. But when exclusivity is assumed, only 31% of the arcs and no transitions are removed. This difference is caused by the additional event paths created when restoring exclusivity after assuming non-exclusivity, which results in about 35-40% more transitions and arcs. Interestingly, every single one of these is detected as impossible by the optimization, so the non-exclusive net after optimization is exactly the same as the exclusive one. The impact of these size reductions on the verification time is negligible for a length of 8 (3 s instead of 4 s). But for a length of 32, it can save 40-57% of the verification time (12 s instead of 20-28 s). The optimization has no noticeable impact on the export time, which is less than 1 s in total for all cases.

When using the complex “puzzle” task instead, the exported petri nets contain about 9 times more transitions and about 17 times more arcs (Table 6). Assuming exclusive conditions, the optimization reduces places (68%) and transitions (0%) in a similar fashion as in the “locked door” example, but only 19% of the arcs are removed. With non-exclusivity, however, even for the small example with a length of 8 there are too many additional path combinations generated during export, and it has therefore been aborted after 5 min. This shows that exclusivity should always be preferred, especially since the same example can be exported with exclusive conditions in less than 1 s. But even though the exclusive version can be exported, the resulting

LEN.	EXCL.	OPT.	PLACES	TRANSITIONS	ARCS	EXP.	VER.
8	Yes	Yes	43	68%	242	0%	1946
		No	134		242		2430
	No	Yes	-	-	-	-	-
		No	-	-	-	-	-

Table 6: Verification results for the generated examples (export parameters for “puzzle” task). For places, transitions and arcs the reduction that can be achieved by the optimization is given in percent.

net is too complex to be verified completely. This can be explained by the state space explosion, though the optimization is able to remove elements to a similar degree as in the synthetic examples. Since the switches can be flipped after opening the door in order to close it again while the other player has moved on and is working on a different puzzle, every combination of switch positions in all locations must be verified.

LEN.	EXCL.	OPT.	PLACES	TRANSITIONS	ARCS	EXP.	VER.
8	Yes	Yes	12	0%	35	0%	159
		No	12		35		229
	No	Yes	12	0%	35	35%	159
		No	12		54		342

Table 7: Verification results for the generated examples (export parameters for “locked door” task, alternate model). For places, transitions and arcs the reduction that can be achieved by the optimization is given in percent.

The alternate model described in Section 6.3.4 is also evaluated for the small test case with a length of 8 (Table 7). Its export results for the same game structure are roughly similar, with less places (12 instead of 19 after optimization), but more transitions (35 instead of 26) and arcs (159 instead of 114). But the verification has been stopped after several minutes without a result, which indicates that the alternate structure is much harder to verify for the chosen tool – and that the size is not a good indicator for a net’s verification complexity. Therefore, there is no reason to use this model instead.

Based on these findings, assuming exclusivity and optimizing the net is used as the default setting for the following experiments.

10.2.2.2 Game Sizes

In order to investigate the scaling properties of the approach, the same baseline configuration is used. Then, multiple experiments are conducted, each of them varying only one size parameter and keeping everything else constant.

The game length varies between 2 and 32 for both the “locked door” and “puzzle” task (full data in Table F, Table 26). This causes the petri net size to grow in a linear fashion, for which the arcs (Figure 52a) are a good indicator as, by definition, each

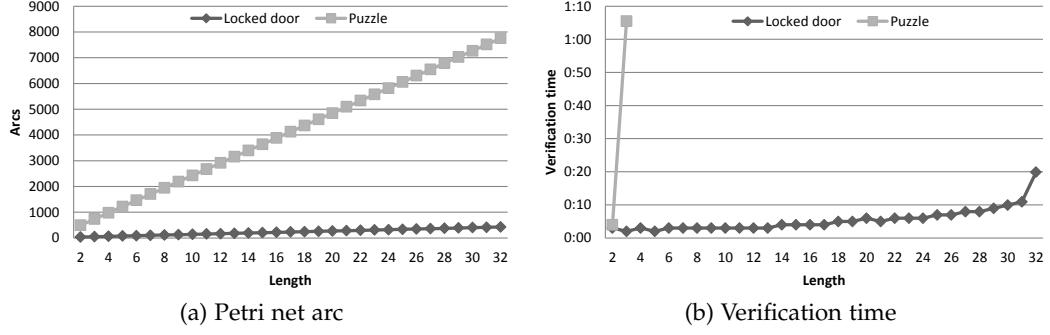


Figure 52: Verification results for different game lengths.

additional place and transition introduces at least two more arcs. But this increase in size has no noticeable impact on the export time, which stays below 1 s for all examples. The verification time on the other hand grows exponentially (Figure 52b), which can be explained by a state space explosion. For the “locked door” task, verification is still viable up to a length of 32 locations (20 s), as the state is only increased by the possible players position and a single variable per room. Nevertheless, an exponential increase is suggested by the increase being only 1 s (10 s and 11 s) between a length of 30 and 31, while being 9 s (11 s and 20 s) between 31 and 32. The “puzzle” version, however, takes more than 60 s for three locations and is already infeasible for four locations. This is due to the fact that the state of each puzzle consists of multiple variables that can be toggled back and forth, which multiplies the state space size.

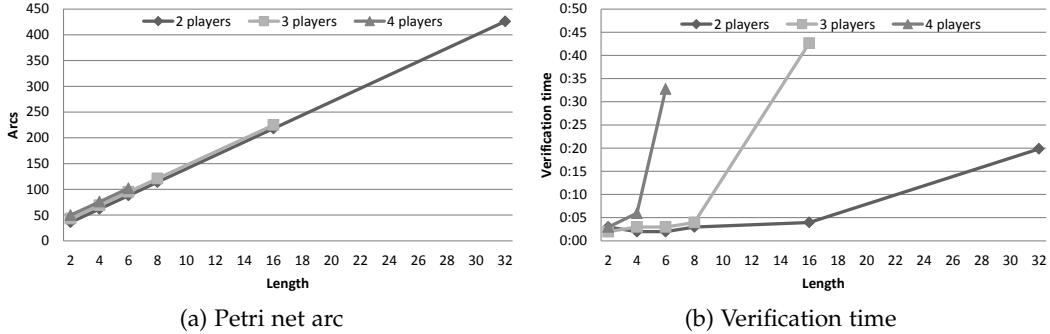


Figure 53: Verification results for different player counts.

Having more players (full data in Table F, Table 27) increases the petri net size slightly (Figure 53a). This difference is purely caused by the generator having to involve all players. But the verification time increases exponentially, as can be seen by sampling game lengths of 2, 4, 8, 16 and 32 with 2-4 players (Figure 53b). The two smallest lengths show only small differences in verification time between the different player counts (3-6 s), but when having 6 locations, there is a noticeable increase for 4 players (33 s). With 8 locations 2 and 3 players are unproblematic, but the net with 4 players cannot be verified anymore. Using a length of 16 there is a great increase when verifying a 3-player version (43 s) and with 32 locations, even the 3-player version has to be aborted. This can be explained by the fact that each new player token can be in any game location, while the additional boolean variable of

another room with a locked door only knows two states. Therefore, each additional player increases the state space by a larger margin than another game room.

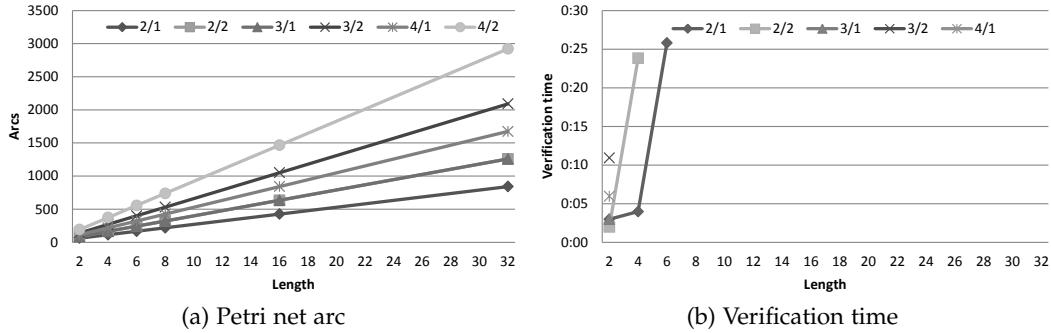


Figure 54: Verification results for different branch structures (“locked door” task).

The influence of optional locations and events is also investigated by sampling the length at 2, 4, 6, 8, 16 and 32 with branching factors of 2-4 and branch lengths of 1 and 2 (full data in Table F, Table 28). Again, the petri net size increases linearly with the game length and the branch configuration governs how steep this increase is (Figure 54a and Figure 55a). But with the “locked door” task, a length of 4 is not verifiable for all but two branching configurations, which become quickly impossible for larger lengths, too (Figure 54b). This is caused by the door state in each optional location that can be toggled, which creates exponentially more game states. The effect is even larger with branches, as these optional doors can be opened at different times. For example, a player could move to the third location and then back to open the optional door to a branch in the first location.

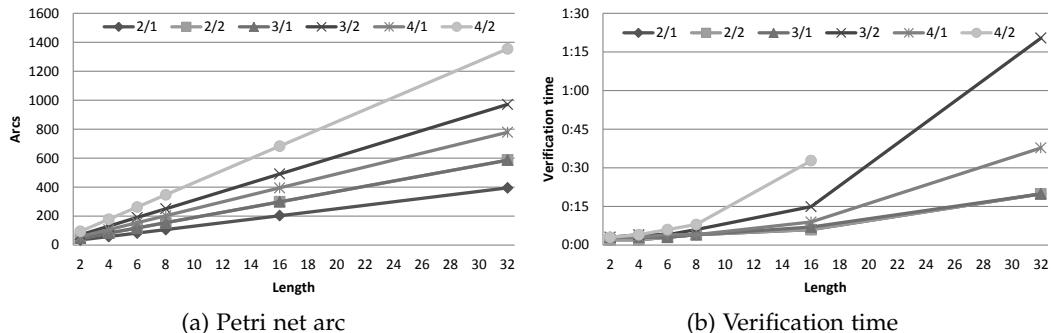


Figure 55: Verification results for different branch structures (no task).

Without any task, there is no explicit state for each room and only the value range for the player location increases when more branches are added. This causes the exponential increase to be noticeable at larger lengths (Figure 55b), with 32 being the first one for which a branching factor of 4 and a length of 2 is impossible.

10.2.3 Existing games

A few real world examples are taken from a pool of existing games made with the authoring tool. In order to be able to compare exclusive and non-exclusive event

linearization, those games have to be designed for exclusive branching. Assuming non-exclusivity does not change the results in this case, while assuming exclusivity when the game is in fact non-exclusive, can lead to wrong results. Additionally, their variables need to have a limited value range so that they can be exported using the alternate model, too.

As the primary example, a multiplayer adventure game for two players is chosen [91]. Its most recent version consists of 13 locations and features four complex tasks the players must solve, which takes about 15 min. These tasks span multiple subtasks located at different rooms and require the players to collaborate. The game is designed for fixed player roles, which means that some actions can only be executed by one of the players. For example, only the first player is small enough to fit a vent while the other one is strong enough to move a fallen tree branch. Trivial actions like opening a door can be done by either player. Whether a subtask has been solved is encoded in variables, of which there are 25 overall (24 boolean, 1 integer).

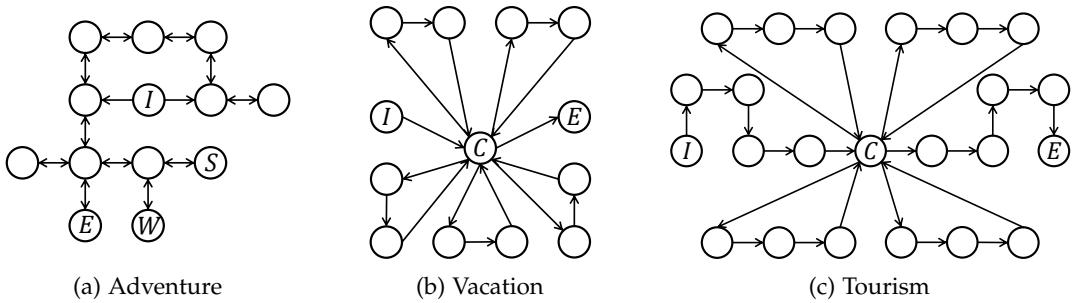


Figure 56: Game structures.

The adventure game is structured (Figure 56a) in such a way that the whole game world becomes accessible after solving the first puzzle (I). This means that players can move freely between rooms and solve the remaining tasks in any order and piece by piece. Therefore, the state space of this game is much larger in comparison to a linear game containing the same tasks. After all tasks have been solved, the players meet at a special location to trigger the ending (E).

While all tasks take roughly the same time to solve, two of them stand out from a verification perspective. One of them requires the players to align a satellite dish until it reaches 100% signal strength. This is modeled using eight discrete states encoded in the integer variable. For the other one the players need to connect four cables with the right sockets, for which there are $4! = 24$ solution candidates. Since the players can freely pick up, connect and disconnect cables at any time, there are even more intermediate states. To investigate the impact of these puzzles two additional versions of the game are created. For one version the wire puzzle (W) is replaced with a trivial action (- 3 min playtime), for the other version the satellite puzzle (S) is also removed (- 5 min). Additionally, the wire puzzle is verified on its own.

Aside from these different versions of the multiplayer adventure game, two single-player games are tested as well ("vacation", Figure 56b and "tourism", Figure 56c). Both of them have a similar structure and begin with a linear introduction (I). After that, the player arrives at a central location (C) and has to solve six minigames from there, but can choose the order in which to do so. When this is done, the game ends with a linear conclusion (E). In contrast to the adventure game, the different parts

of the game are linear, i.e. once a section has been chosen the player cannot go back – which causes their state space to be much smaller in comparison. In both games, the minigames include a jigsaw puzzle, a memory and a hidden object game. These minigames are based on predefined interaction templates and are not scripted manually in the authoring tool. Therefore, they are black boxes for the petri net export, which assumes that they work like any other action. This means that solving the memory minigame is like clicking a button, i.e. at one point the players have solved the game. From the perspective of the application scenario, this is not a problem. All templates can be verified before implementing them as black boxes and the developer using them cannot make any further mistakes. While structurally similar, the tourism game contains more story content and it therefore takes players more time to complete it (8 min for tourism, 5 min for vacation).

ALT.	EXCL.	OPT.	PLACES		TRANSITIONS		ARCS		EXP.	VER.
VACATION										
No	Yes	Yes	29	77%	28	9%	144	29%	0:00	0:03
		No	131		31		203		0:00	0:05
	No	Yes	29	77%	28	26%	144	45%	0:00	0:05
		No	131		38		262		0:00	0:04
Yes	Yes	Yes	19	24%	46	6%	270	26%	0:00	0:10
		No	25		49		365		0:00	0:15
	No	Yes	19	24%	46	24%	270	39%	0:00	0:10
		No	25		61		449		0:00	0:16
TOURISM										
No	Yes	Yes	44	74%	67	0%	241	35%	0:00	0:04
		No	175		67		375		0:00	0:07
	No	Yes	44	74%	67	20%	241	51%	0:00	0:04
		No	175		84		500		0:00	0:07
Yes	Yes	Yes	46	13%	84	0%	326	34%	0:00	0:10
		No	53		84		494		0:00	0:16
	No	Yes	46	13%	84	19%	326	46%	0:00	0:10
		No	53		104		610		0:00	0:48

Table 8: Verification results for the vacation and tourism examples. For places, transitions and arcs the reduction that can be achieved by the optimization is given in percent. The upper half of the table shows the chosen model, the lower an alternative one discussed in Section 6.3.4.

Both singleplayer examples are viable with all export parameters (Table 8). The export time is always less than 1 s, verifying the chosen model takes 3-5 s (“vacation”) and 4-7 s (“tourism”). Optimization is able to reduce the net size significantly (74-77% of the places, 0-26% of the transitions, 29-51% of the arcs), which can also reduce

the verification time by a few seconds. As in the generated examples, all additional paths generated when assuming non-exclusivity can be removed by the optimization. The examples can still be verified using the alternate model (lower halves of Table 8), although this increases verification time to 48 s in the unoptimized, non-exclusive “tourism” example. Matching its longer playtime, the model of the tourism game is about 50% bigger – both in net size as well as verification time (Table 8, lower part).

As expected, the adventure game variants are more complex than the singleplayer examples in terms of petri net size and verification time (Table 9 and Table 10). But this also means that the optimization yields greater savings and is in some cases even able to reduce the export time itself. For example, when assuming non-exclusivity for the full game (Table 9, first part), the unoptimized export time is 80 s. Here, the export times drop to 1 s overall after optimization. This means that the optimization save more time in the later transformation steps – especially during the event linearization – than what the optimization costs itself. In contrast to the (minimal) generated examples, the optimization is also able to remove many actions that have visual effects only. Some cases, like the version without the wire puzzle (Table 9, second part), can only be verified when their net has been optimized. Here, especially the non-exclusive nets explode in terms of complexity when not properly optimized (99% of both transitions and arcs are removed). As before, the exclusive and non-exclusive nets are identical after optimization.

The full example cannot be verified in any configuration, however, and the process has been aborted after 10 min (Table 9, first part). Cutting this example into pieces by separating the wires yields a different result, with both parts being verified after 5 s (wire puzzle, Table 10) and 46 s (rest of the game, second part in Table 9). This can be explained by the fact that the other part is then treated like a blackbox, which at some point produces a result. Intermediate states, i.e. players switching between unfinished tasks, are not known and therefore not combined with the own state space. It is therefore advisable to split larger games into sections if there are no dependencies between their intermediate states.

The alternate model is completely infeasible for all adventure variants. Only the wire puzzle alone can be verified in its optimal configuration (Table 10). But even the simplest version of the overall game cannot be verified using the alternate model (Table 9, third part), although this is possible when using the default model. For the larger ones already the export fails, unless the game is exclusive and the export optimized (lower halves of the first part of Table 9 and Table 10).

All of the examples discussed here do not contain any structural errors. An older version of the multiplayer adventure contains a livelock, however. This allows us to confirm that the structural errors discussed do not influence the export time or net size, as every element is transformed individually. But the verification ends earlier because the error reduces the state space to be explored. Fixing the problem is relatively easy, since the simulation also shows lots of unvisited locations. These locations contain crucial tasks required for solving the game, which leads to the ending being unreachable, too. This hints at the event for moving the players there being impossible, which can be confirmed quickly.

All in all, these real data results confirm the findings based on the generated examples.

ALT.	EXCL.	OPT.	PLACES		TRANSITIONS		ARCS		EXP.	VER.
FULL GAME										
No	Yes	Yes	54	67%	131	27%	1141	59%	0:00	-
		No	164		180		2822		0:00	-
	No	Yes	54	67%	131	99%	1141	99%	0:01	-
		No	164		37708		785650		1:20	-
Yes	Yes	Yes	25		493		5209		0:00	-
		No	-		-		-		-	-
	No	Yes	-		-		-		-	-
		No	-		-		-		-	-
WITHOUT WIRE PUZZLE										
No	Yes	Yes	41	73%	72	35%	400	49%	0:00	0:46
		No	156		111		793		0:00	-
	No	Yes	41	73%	72	99%	400	99%	0:00	0:46
		No	156		30570		329768		0:37	-
Yes	Yes	Yes	25	0%	105	38%	647	51%	0:00	-
		No	25		171		1323		0:00	-
	No	Yes	25		105		647		0:01	-
		No	-		-		-		-	-
WITHOUT WIRE AND SATELLITE PUZZLES										
No	Yes	Yes	39	75%	54	33%	292	47%	0:00	0:05
		No	156		81		557		0:00	0:23
	No	Yes	39	75%	54	79%	292	87%	0:00	0:04
		No	156		265		2367		0:00	-
Yes	Yes	Yes	19	5%	84	36%	512	49%	0:00	-
		No	20		132		1006		0:00	-
	No	Yes	19	5%	84	97%	512	98%	0:00	-
		No	20		3689		29613		0:01	-

Table 9: Verification results for different versions of the multiplayer adventure. For places, transitions and arcs the reduction that could be achieved by the optimization is given in percent. The upper half of the table shows the chosen model, the lower an alternative one discussed in Section 6.3.4.

10.3 COLLABORATIVE BALANCING

To evaluate the verification approach, the minimal functionality and larger scaling tests are separated into toy and synthetic examples. For the balancing approxima-

ALT.	EXCL.	OPT.	PLACES	TRANSITIONS		ARCS		EXP.	VER.
No	Yes	Yes	18	80%	59	13%	737	63%	0:00 0:05
		No	91		68		2016		0:00 0:06
	Yes	Yes	18	80%	59	99%	737	99%	0:01 0:04
		No	91		7132		455824		0:43 -
Yes	Yes	Yes	9		385		4539		0:04 0:25
		No	-		-		-		-
	Yes	Yes	-		-		-		-
		No	-		-		-		-

Table 10: Verification results for the wire puzzle of the multiplayer adventure. For places, transitions and arcs the reduction that can be achieved by the optimization is given in percent. The upper half of the table shows the chosen model, the lower an alternative one discussed in Section 6.3.4.

tions, in contrast, minimal toy examples cannot cover all calculations as they take the distribution over multiple subsections of the game into account. Another difference is that structural errors reduce the state space, while bad balancing does not cause the approximation to terminate early. Therefore, the generated examples are built in such a way that they also allow functionality testing, taking on the role of toy examples. As a second test case, the existing games already discussed above are used, with all singleplayer games being left out. This is due to the fact that structural errors can appear for any number of players, including a single one. In contrast, collaborative balancing by definition requires multiple players to be present.

The different methods for separating a game into sections (Section 7.2.4) are very similar in regard to their complexity. It has therefore been decided to evaluate only the fixed section method.

For all results a PC with an Intel Core i7-4790K CPU (4.00 GHz, 8 logical cores) and 16 GB RAM on Windows 8.1 Pro (64 bit) is used.

10.3.1 Synthetic Examples

In the synthetic example there are only two types of events, solving a task and moving to another room. Therefore, the event weight heuristic is less relevant and only the “stimuli” heuristic is chosen for the evaluation. Additionally, the different methods are designed to be interchangeable. To improve the clarity of the tables, only the effort metric is shown if not otherwise noted, although the waiting times and the options have been approximated as well.

To investigate the runtime behavior of the approximation process, some of the previously generated games with a verifiable state space are selected:

FOR LENGTH SCALING 2 players with no branches and the lengths of 8, 16 and 32.
 FOR BRANCH SCALING 2 players with a length of 4, a branching factor of 2 and the branch lengths of 1 and 2 (also length 4 and no branches for comparisons).
 Additionally 2 players, length of 6, branching factor of 2 and branch length of 1

(again without branches as a comparison). These are the largest verifiable state spaces with branches.

FOR PLAYER SCALING 3 players with a length of 16 and no branches (the two player version for comparison is already calculated for length scaling). This is the largest verifiable state space with more than two players.

All of these examples use the “locked door” task, since it constitutes a reasonable trade-off between complexity and verifiability. In the following, the notation of *length-/branching factor/branch length* is used to describe a game’s structure, e.g. 32/1/0 means a length of 32, a branching factor of 1 and a branch length of 0.

These examples show that although the A* requires more calculations to prepare the additional heuristic, it always saves time in the end (Table 11). Especially in two cases in which paths to many endings¹ need to be found (i.e. games with many optional branches), this algorithm can reduce the time from more than 180 s to about 40 s. Only in the example for three players, the calculation time is longer than 240 s with both algorithms. This can be explained in part by the need to calculate more paths, but also by the indirect effect of a much larger state space to explore. In the less complex examples, the balancing calculations take just a few seconds with both algorithms. Naturally, both pathfinding algorithms find the same shortest paths, therefore A* is used as the default from now on.

STRUCTURE	PLAYERS	ENDINGS	PATHS	DIJKSTRA	A*	DIFF.
8/1/0	2	9	72	0:00	0:00	0%
16/1/0	2	17	136	0:00	0:00	0%
32/1/0	2	33	264	0:06	0:05	17%
4/1/0	2	5	40	0:00	0:00	0%
4/2/1	2	112	896	0:02	0:00	100%
4/2/2	2	729	5832	3:27	0:45	78%
6/1/0	2	7	56	0:00	0:00	0%
6/2/1	2	640	5120	3:32	0:36	83%
16/1/0	3	289	3179	4:50	4:21	10%

Table 11: Balancing times for the generated examples and different pathfinding algorithms. The rows display the game structure (*overall length / branching factor / branch length*), the *number of players*, the *number of endings*, the *number of explored paths*, as well as the calculation time when using the *Dijkstra*- or *A**-algorithms and the difference between them.

Based on these findings, the three most complex two-player-games based on their calculation time (4/2/2, 6/2/1 and 23/1/0) are selected as the examples for the subsequent experiments.

To check whether the calculated balancing results are as expected, the values for the three task assignment strategies (Section 7.3) are compared (Table 12, upper part).

¹ For each ending the paths for each metric are calculated. Each metric in turn requires a path for each player and one overall path. Since the overall paths for effort and waiting are the same, this does not need to be calculated twice. Therefore, the overall path count is $|S^E| \cdot (3 \cdot |Pl| + 2)$.

When calculated over the whole game, the values for the alternating and the block based strategy are similar. In the best case (minimal standard deviation), the game is perfectly balanced (0%), but there seem to be some paths that are unbalanced as well (maximum of up to 47% difference). Analyzing these paths shows that on them, one player moves to another location after completing his tasks. In the examples with optional rooms there can also be paths on which some of the irrelevant doors have been opened. This changes the end state, because this player is now at another location, and is therefore counted as well. For the singleplayer assignment, it is possible that one player does everything and the others do nothing (maximum of 100% difference between the players). Similarly to the other assignment strategies, there are also some alternative paths on which the others move around as optional actions (minimum of 34%). But in the end, the averages already show that alternating and block based assignments are relatively balanced while singleplayer is not (average of 13-20% compared to 63-75%). Therefore, all of these values match the expected values based on the metric definitions as well as the game generation specification.

Enabling the culling extension for end states changes these results (Table 12, lower part). Although the paths to end states with superfluous movement are still discovered, they are now discarded by the subpath culling. Only the path to a single (minimal) ending remains, which is used in all 8 calculations. Because of this, the results are less ambiguous and show that the overall balancing is perfect for the alternating and block based examples (0%). On the other hand, the difference between players reaches its worst case in the singleplayer assignment (100%). This matches exactly what would be expected based on the game generation algorithm. The additional path comparisons have no noticeable effect on the runtime, which is still below 60 s for all examples.

Another extension that compares the games' sections should show a difference between the block based and alternating examples because the effort is differently distributed there. Without culling end states, the results on this are mixed (Table 13, upper part). For the short game with lots of optional paths (4/2/2), the average difference between two sections is very similar between the perfectly distributed (alternating, 21%) and worst case (block based, 28%) versions. This can be explained by the optional actions being included into the paths at different positions, which skews the clear cut distribution on the main paths. In the longest example without branches (23/1/0) this effect is smaller, as the only optional actions consist of moving backwards through the game. Here, the block based approach (30%) shows a much higher variance between the sections compared to the alternating version (13%). Having four sections instead of two increases the section differences between both task distributions (22% and 40% for the shortest example, 14% and 35% for the longest). Calculating the sections has no substantial impact on the runtime (still below 60 s), as it requires only a simple calculation on the paths that have already been calculated.

Culling should make these results less ambiguous, as it removes all paths containing optional actions (Table 13, lower part). However, for the shortest game (4/2/2) the balancing between sections is worse in the alternating approach (29% and 35% instead of 21% and 22%), which seems counter-intuitive. This is due to the fact that the game contains of only a few events, which can skew a section's effort greatly and makes differences between the sections more likely. But for the longest example (32/1/0) the results are almost as expected, with the alternating example being close to perfectly distributed (4% on average for both section counts) and the block based

STRUCT.	ASSIGNMENT	PATHS	EFFORT SD				
			MIN.	Avg.	MAX.	TIME	
No CULLING							
4/2/2	Alternating	5832	0%	13%	38%	0:45	
	BlockBased	5832	0%	20%	47%	0:51	
	SinglePlayer	5832	38%	75%	100%	0:24	
6/2/1	Alternating	5120	0%	13%	33%	0:36	
	BlockBased	5120	0%	13%	31%	0:43	
	SinglePlayer	5120	37%	69%	100%	0:20	
32/1/0	Alternating	264	0%	13%	25%	0:05	
	BlockBased	264	0%	13%	25%	0:08	
	SinglePlayer	264	34%	63%	100%	0:04	
CULLING							
4/2/2	Alternating	8	0%	0%	0%	0:50	
	BlockBased	8	0%	0%	0%	0:51	
	SinglePlayer	8	100%	100%	100%	0:24	
6/2/1	Alternating	8	0%	0%	0%	0:36	
	BlockBased	8	0%	0%	0%	0:43	
	SinglePlayer	8	100%	100%	100%	0:21	
32/1/0	Alternating	8	0%	0%	0%	0:05	
	BlockBased	8	0%	0%	0%	0:08	
	SinglePlayer	8	100%	100%	100%	0:05	

Table 12: Balancing results for the generated examples and different task assignments. The rows display the game structure (*overall length / branching factor / branch length*), the *task assignment strategy*, the *number of explored paths*, the *standard deviation between players* in regards to *effort* (*minimum, average, maximum*) and the *calculation time*.

approach being much worse (29% and 34% on average). The still not perfect distribution for the alternating example could be surprising at first. Each of the 32 steps contains of 3 actions, one player opening the door and then both players passing through. With an alternating approach and n players, every group of n sequential steps should be balanced as different players open the door in each one. Any multiple of these n steps is balanced, too. This suggests that in the 32 step game with $n = 2$ players both 2 ($32/2 = 2 \cdot 8$ steps) and 4 ($32/4 = 2 \cdot 4$) section should be balanced. However, the ending of the game is an additional step which requires 2 actions (1 by each player). Therefore, the borders of the 2 or 4 sections do not align perfectly with the 33 steps, causing these slight imbalances in the alternating example.

If the multiple shortest paths extension is enabled (Table 14, upper part), all shortest paths to the ending are calculated. Consequently, the number of paths is greatly

STRUCT.	ASSIGN.	PATHS	SECTION SD (EFFORT, 2 SECTIONS)			SECTION SD (EFFORT, 4 SECTIONS)			TIME
			MIN.	AVG.	MAX.	MIN.	AVG.	MAX.	
No Culling									
4/2/2	Altern.	5832	0%	21%	60%	0%	22%	46%	0:55
	Blocks.	5832	0%	28%	100%	20%	41%	62%	0:57
6/2/1	Altern.	5120	0%	14%	38%	8%	17%	29%	0:43
	Blocks.	5120	0%	22%	60%	26%	39%	58%	0:50
32/1/0	Altern.	264	0%	13%	31%	2%	14%	28%	0:07
	Blocks.	264	27%	30%	35%	33%	35%	38%	0:09
Culling									
4/2/2	Altern.	8	14%	29%	43%	27%	35%	43%	0:54
	Blocks.	8	14%	14%	14%	36%	42%	49%	1:01
6/2/1	Altern.	8	0%	10%	20%	12%	12%	12%	0:43
	Blocks.	8	0%	10%	20%	35%	38%	42%	0:52
32/1/0	Altern.	8	2%	4%	6%	2%	4%	5%	0:07
	Blocks.	8	27%	29%	31%	33%	34%	34%	0:10

Table 13: Balancing results for the generated examples with a varying number of sections. The rows display the game structure (*overall length / branching factor / branch length*), the *task assignment strategy*, the *number of explored paths*, the *standard deviation* between players in regards to *effort* for 2 and 4 sections (*minimum, average, maximum*) and the *calculation time*.

increased (e.g. more than 500,000 instead of 5,823 for the 4/2/2 structure). This can be explained by the fact that, after a door has been opened, the players can go through it in different orders – multiplying the number of paths with each new door. Considering each of these additional paths changes the overall section results only slightly (eg. 19% instead of 21%), because swapping the order of these movement actions seldom causes them to fall into another section. As expected, it increases runtime (by around 50% in the example, up to 90 s instead of 60 s).

When culling is also enabled in addition to the multiple path extension, no results can be obtained for the examples with branches (Table 14, lower part). This is due to the current version of the subpath detection requiring a pairwise comparison between each path, which yields too many combinations for the already high number of paths. Optimizing the culling algorithm could maybe solve this issue. For the 32/1/0 structure, culling improves the values in a similar fashion as in the previous experiments without the multiple shortest path extensions (Table 13).

A simplified calculation is also tested (Table 15). This version calculates only one path for each player and ending, which optimizes the effort but on which the other metrics are calculated as well. Additionally, it does not calculate an optimal path for the group as a reference point, and used the overall effort on the player’s paths instead. All measurements are taken without the culling extension so that more paths

STRUCT.	ASSIGN.	PATHS	SECTION SD (EFFORT, 2 SECTIONS)			SECTION SD (EFFORT, 4 SECTIONS)			TIME
			MIN.	AVG.	MAX.	MIN.	AVG.	MAX.	
No CULLING									
4/2/2	Altern.	521943	0%	19%	60%	0%	21%	49%	1:26
	Blocks.	513032	0%	27%	100%	0%	38%	64%	1:29
6/2/1	Altern.	456098	0%	13%	38%	0%	16%	38%	1:16
	Blocks.	452928	0%	22%	60%	12%	37%	59%	1:23
32/1/0	Altern.	23133	0%	13%	31%	2%	14%	28%	0:12
	Blocks.	23133	27%	30%	35%	31%	34%	38%	0:15
CULLING									
4/2/2	Altern.	-	-	-	-	-	-	-	-
	Blocks.	-	-	-	-	-	-	-	-
6/2/1	Altern.	-	-	-	-	-	-	-	-
	Blocks.	-	-	-	-	-	-	-	-
32/1/0	Altern.	701	2%	4%	6%	2%	4%	5%	0:09
	Blocks.	701	2%	27%	29%	31%	33%	34%	0:11

Table 14: Balancing results for the generated examples with a varying number of sections and multiple shortest paths. The rows display the game structure (*overall length / branching factor / branch length*), the *task assignment strategy*, the *number of explored paths*, the *standard deviation between players in regards to effort* for 2 and 4 sections (*minimum, average, maximum*) and the *calculation time*.

are evaluated. These simplifications cause only 2 instead of 8 paths to be calculated per ending, one for each player, which results in 75% less paths being explored. This saves between half and two thirds of the time (e.g. 18 s instead of 51 s). On the generated examples they also produce exactly the same results – even on the options metric, which is independent of the (optimized) effort. But upon close inspection, this is not surprising. The generated examples are linear and the only difference between the ending states is which optional tasks are solved along the way. In turn, this means that there is only one path to each ending without backtracking and that this path is the same for all metrics.

10.3.2 Existing Games

The balancing calculations are also tested using the multiplayer adventure game variants. This game has already been evaluated in a user study during which all players said that they felt involved throughout the game [91]. As this can be correlated to our balancing definition, the approximation is expected to show the game to be relatively balanced, too (low deviation between players). However, due to the full version not being verifiable in the previous step, only the separated parts can be analyzed.

STRUCT.	ASS.	SIM.	PATHS	EFFORT SD			OPTIONS SD			TIME
				MIN.	AVG.	MAX.	MIN.	Avg.	MAX.	
4/2/2	Alt.	No	1458	0%	13%	38%	0%	5%	17%	0:17
4/2/2	Alt.	Yes	5832	0%	13%	38%	0%	5%	17%	0:45
4/2/2	Blo.	No	1458	0%	20%	47%	0%	7%	21%	0:18
4/2/2	Blo.	Yes	5832	0%	20%	47%	0%	7%	21%	0:51
4/2/2	Sin.	No	1458	38%	75%	100%	0%	32%	78%	0:09
4/2/2	Sin.	Yes	5832	38%	75%	100%	0%	32%	78%	0:24
6/2/1	Alt.	No	1280	0%	13%	33%	0%	0%	2%	0:14
6/2/1	Alt.	Yes	5120	0%	13%	33%	0%	0%	2%	0:36
6/2/1	Blo.	No	1280	0%	13%	31%	0%	0%	2%	0:16
6/2/1	Blo.	Yes	5120	0%	13%	31%	0%	0%	2%	0:43
6/2/1	Sin.	No	1280	37%	69%	100%	0%	31%	77%	0:08
6/2/1	Sin.	Yes	5120	37%	69%	100%	0%	31%	77%	0:20
32/1/0	Alt.	No	66	0%	13%	25%	0%	0%	0%	0:02
32/1/0	Alt.	Yes	264	0%	13%	25%	0%	0%	0%	0:05
32/1/0	Blo.	No	66	0%	13%	25%	0%	0%	0%	0:02
32/1/0	Blo.	Yes	264	0%	13%	25%	0%	0%	0%	0:08
32/1/0	Sin.	No	66	34%	63%	100%	0%	37%	96%	0:01
32/1/0	Sin.	Yes	264	34%	63%	100%	0%	37%	96%	0:04

Table 15: Balancing results for the generated examples with a simplified calculation. The rows display the game structure (*overall length / branching factor / branch length*), the *task assignment strategy*, whether the *simplified calculations* are used, the *number of explored paths*, the *standard deviation* between players in regards to *effort* and *options* (*minimum, average, maximum*) and the *calculation time*.

The other two examples can not be used as they have been designed as singleplayer games, which means that there is no collaborative balance by definition.

Similar to the synthetic examples, using the A* algorithm for pathfinding can lead to a significant runtime reduction of up to 52% (Table 16).

VERSION	ENDINGS	PATHS	DIJKSTRA	A*	DIFFERENCE
W/o wires	88	704	1:05	0:31	52%
W/o wires+sat.	44	352	0:02	0:01	50%
Wires only	18	144	0:00	0:00	0%

Table 16: Balancing times for the multiplayer adventure with for different pathfinding algorithms.

But here, the path cost heuristic has a noticeable impact on the calculated values (Table 17). For example, while the average difference between players in the version

without the wire puzzle is 10% when using the stimuli heuristic, it changes to 21% for the other heuristics. Removing the satellite puzzle as well changes these values to 8%, 17% and 24%. This can be explained by the game's events greatly varying in complexity. Depending on how much influence the differences in complexity have on the weighting heuristics, their estimated costs can vary as well. The resulting balancing value changes, when the most complex events are distributed in a different manner than the ones with an average weight. For the wire puzzle the impact is even more substantial. While the *stimuli* and *relevance* heuristics are slightly unbalanced on average (30% and 44%), there are also perfectly balanced paths (minimum of 0%). Using the *actions* heuristic however, even the best case is really unbalanced (62%). This can be explained by one event consisting of many actions, which are ultimately irrelevant. Therefore, this event has a high weight in this metric only. Additionally, the puzzle consists of relatively few events compared to the overall game, so each one has a greater impact on the balancing.

VERSION	WEIGHT HEUR.	PATHS	EFFORT SD			
			MIN.	AVG.	MAX.	TIME
W/o wires	Stimuli	704	0%	10%	22%	0:31
	Reactions	704	3%	21%	48%	0:41
	Relevance	704	0%	21%	43%	0:40
W/o wires+sat.	Stimuli	352	0%	8%	22%	0:01
	Reactions	352	0%	17%	39%	0:01
	Relevance	352	1%	24%	43%	0:01
Wires only	Stimuli	144	0%	32%	100%	0:00
	Reactions	144	62%	70%	100%	0:00
	Relevance	144	0%	44%	100%	0:00

Table 17: Balancing results for the multiplayer adventure with different weight heuristics.

It is also interesting that there are perfectly balanced paths (0%) in different combinations of both the game versions and the heuristics. For example, the game without the wire puzzle has perfect paths using the *stimuli* and *relevance* heuristics. But when the satellite is also removed, only the *stimuli* and *action* heuristics can produce perfect paths. In the biggest example, the heuristic has also an influence on the calculation time (31 s instead of about 40 s). Such an effect can be caused by the pathfinding having to explore more suboptimal paths first. However, there is no “right” heuristic, as the event weighting is left to the developer’s discretion.

Calculating the deviation between 2 and 4 sections does show significant imbalances between these parts of the game (Table 18, first part). For the largest version there is 33% variance between two sections and 29% between four, which means that the effort is distributed unevenly between these parts of the game. In the wires puzzle the differences are even more extreme (63% and 80% on average), although there are paths on which there is perfect balance between each section (0%). This is due to the short overall game length, which increases the impact that a single event can have on a section’s value. Enabling end state culling does not change this result sig-

VERSION	PATHS	EFFORT SD (BETW. 2 SECTIONS)			EFFORT SD (BETW. 4 SECTIONS)			TIME
		MIN.	AVG.	MAX.	MIN.	AVG.	MAX.	
ONE SHORTEST PATH, NO CULLING								
W/o wires	704	0%	33%	73%	16%	29%	52%	0:32
W/o wires+sat.	352	0%	19%	42%	8%	25%	53%	0:01
Wires only	144	0%	63%	100%	0%	80%	100%	0:00
ONE SHORTEST PATH, CULLING								
W/o wires	145	0%	30%	70%	18%	31%	52%	0:36
W/o wires+sat.	102	0%	17%	36%	14%	28%	53%	0:01
Wires only	144	0%	63%	100%	0%	80%	100%	0:00
MULTIPLE SHORTEST PATHS, NO CULLING								
W/o wires	67804	0%	33%	73%	13%	28%	52%	0:45
W/o wires+sat.	34528	0%	19%	42%	8%	24%	53%	0:06
Wires only	11120	0%	50%	100%	0%	61%	100%	0:03
MULTIPLE SHORTEST PATHS, CULLING								
W/o wires	13956	0%	30%	70%	14%	30%	52%	2:27
W/o wires+sat.	9976	0%	17%	36%	14%	27%	53%	0:35
Wires only	11120	0%	50%	100%	0%	61%	100%	0:04

Table 18: Balancing results for the multiplayer adventure with different section lengths.

nificantly (Table 18, second part), even though it removes many paths for two of the examples (145 instead of 704 and 102 instead of 352). For the wire puzzle no paths are removed, which means that that there are no end states that imply irrelevant actions. Calculating all shortest paths to the same end state does not change the average section differences significantly for the first two examples (Table 18, third part). But in the wire puzzle, it improves the calculate average deviation from 63-50% (2 sections) and 80-61% (4 sections). The greatest increase in runtime caused by this extension is observed for the largest example (45 s instead of 36 s). In contrast to the synthetic examples, enabling both the culling as well as the multiple shortest paths extension is always viable for these examples (Table 18, fourth part). Their combination changes the average result by only 3% while also increasing the runtime substantially (150 s instead of 45 s with multiple shortest paths only and 32 s without extensions).

The adventure game is also analyzed using the simplified method calculating less paths (Table 19). Again, this results in a greatly reduced analysis time (12 s instead of 31 s maximum). But more surprisingly, it produces exactly the same results as the more complex version. This indicates that the shortest paths are very similar for each metric and player in this game.

VERSION	SIM.	PATHS	EFFORT SD			OPTIONS SD			TIME
			MIN.	AVG.	MAX.	MIN.	Avg.	MAX.	
W/o wires	No	704	0%	10%	22%	0%	3%	5%	0:31
	Yes	176	0%	10%	22%	0%	3%	5%	0:12
W/o wires+sat.	No	352	0%	8%	22%	0%	4%	6%	0:01
	Yes	88	0%	8%	22%	0%	4%	6%	0:00
Wires only	No	144	0%	32%	100%	0%	0%	0%	0:00
	Yes	36	0%	32%	100%	0%	0%	0%	0:00

Table 19: Balancing results for the multiplayer adventure with a simplified calculation.

To get a final verdict for the balancing of the real example game, all relevant metrics are calculated for the largest viable version without the wire puzzle (Table 20). These results are calculated with 4 sections and with both extensions (culling and multiple paths) enabled. Counting only the actions that need to be executed (stimuli weight heuristic), the game is quite balanced in effort and waiting times (11% difference on average) and really balanced in the options metric (5% max difference). There are some noticeable distribution differences between the four sections though (about 30% on average for effort and waiting time as well as 27% for options). Using the other weight heuristics, the game is deemed unbalanced for effort and waiting (about 20% on average and 45% maximum). The options in contrast are well balanced according to the game reaction count heuristic (7% average difference, 23%

WEIGHT HEURISTIC			
METRIC	STIMULI	REACTIONS	RELEVANCE
Effort SD (Avg)	11%	21%	22%
Effort SD (Max)	22%	48%	43%
Effort Sections SD (Avg)	30%	40%	36%
Effort Sections SD (Max)	52%	71%	57%
Waiting SD (Avg)	11%	21%	22%
Waiting SD (Max)	22%	48%	43%
Waiting Sections SD (Avg)	30%	40%	36%
Waiting Sections SD (Max)	52%	71%	57%
Options SD (Avg)	2%	7%	29%
Options SD (Max)	5%	23%	55%
Options Sections SD (Avg)	27%	35%	38%
Options Sections SD (Max)	44%	61%	54%
Overall time	02:20	02:27	02:27

Table 20: Overall balancing results for the multiplayer adventure.

maximum), but not according to relevance (29% average, 55% maximum). Again, there are noticeable differences between the sections (up to 40% on average). It is also interesting that the effort and waiting time metrics produce the same results for all weight heuristics. This can be explained by the game assigning most actions to one of the two players, in which case the effort for one player is always counted as waiting time for the other. With all weight heuristics, the calculations took about 150 s, which is perfectly acceptable during development.

In an earlier user study [91, 90], players perceived the game as balanced. But this does not necessarily mean that the balancing approximations produce wrong results. It is possible, that the subjective balancing perception allows limited inequalities and does not need all actions to be distributed perfectly. Maybe there are other factors that influence the perceived balancing as well. Additionally, the user study was conducted with a relatively small sample size, which means that its results should not be seen as fully conclusive. Either way, a developer who wants to improve the game could now look at the individual balancing values of each of the four sections first, in order to narrow down which ones are unbalanced. He could then trace back the calculation results to individual puzzles or events and reassign them to the other player in order to improve the game's balancing.

10.4 RAPID PROTOTYPING

Originally, the rapid prototyping environment was meant to be evaluated in a large scale user study, similar to the evaluation of the pattern game. However, a first qualitative pre-study with 5 users at TU Darmstadt showed that a comparison study between the tool and the naive approach of using multiple instances is trivial. As expected, the dedicated tool makes testing for a single user easier and faster. Especially the automated network setup and the ability to test in a single application on a single device were very well received. Additionally, no substantial usability issues were detected during these tests as the tool was already used regularly at this point and all known problems were removed before. It was therefore decided to not conduct a full user study for the prototyping environment, but rather to focus on evaluating the other modules more thoroughly.

10.5 DISCUSSION

The goal of this chapter is to provide a comprehensive evaluation of all four components of the collaborative multiplayer authoring framework: the player interaction patterns, the structural verification, the collaborative balancing approximation and the rapid prototyping environment.

The player interaction patterns are evaluated in a qualitative user study with 93 participants. For this study, a game that was designed using the player interaction patterns is compared to a version that omits these interactions, but is otherwise as similar as possible. By combining questionnaires, log files and manually created protocols we can show that the collaborative interactions are generally more entertaining for the players. This is true for both the overall game (on average 4.29 compared to 3.85 out of 5) as well as for each individual pattern. It is also shown that collaboration is perceived as more difficult and requires both more time and more communication.

Although these trends have been observed for almost all patterns used, only some of the findings are statistically significant with a sample size of 93.

The verification, in contrast, is evaluated in a simulative manner. For this, a large number of verifications are run on both existing games as well as synthetically generated examples. During these experiments, both the verification and generation parameters for the synthetic examples are varied. This way, we are able to verify the correctness of this approach as well as its scaling behavior. Our results show that the approach is able to detect structural problems such as deadlocks, livelocks, unreachable locations as well as impossible game events. For the given application scenario, i.e. games with a limited state space, such a verification is still viable: When using the proposed parameters, most examples can be verified in a timely manner (i.e. in less than 2 min). Our optimization algorithm is especially crucial for this, as some examples cannot be processed without optimizing them first. On average, the optimization is able to remove 30-60%, and in some extreme cases up to 99%, of the petri net elements without changing the verification result. The runtime of the export process itself, including optimization, is hereby found to be insignificant in comparison to the externally provided verification. However, the overall approach does not scale infinitely due to the state space growing exponentially with a game's size. This means that a full verification becomes infeasible once a game is too complex. In our case, a 15 min game for two players can be exported in less than 1 s, but is still not verified after 10 min. Cutting this game into two mostly independent parts, however, brings their combined verification time down to 51 s – which is still viable.

The balancing approximation is evaluated in a similar manner by using the same examples (real and synthetic). Again, a large number of calculations are run while varying the balancing as well as the game generator parameters. Since the concrete implementation of the approach relies on the structural verification being run first, it inherits the limitations of this approach. This means that the balancing can only be calculated in games for which the verification is viable as well. On all games for which this is the case, the balancing approximation is able to produce its results in a timely fashion as well (i.e. less than 5 min). When using the suggested parameters, the balancing for all two-player examples can even be calculated in less than a minute. For the generated examples, which are intentionally built to reflect certain types of (in-)equality between players, the approximated balancing values match our expectations. For the real games, the balancing approximations are slightly worse than what the players had subjectively reported. But overall, the approximations seem to provide a good indication on whether a game is collaboratively balanced or not.

Lastly, the rapid prototyping environment is evaluated in a small-scale, qualitative user study. Here, it is shown that using this environment constitutes a large improvement when compared to a manual approach.

Overall, these findings indicate that the framework's components work as intended. Namely, using the collaborative interaction patterns causes players to enjoy a game more than a similar, non-collaborative version. The structural verification and collaborative balancing in turn are viable for small games, as described in our application scenario, or even parts of larger ones. Their combination means that it is not necessary anymore to manually test this type of games in order to find critical errors or to assess their balancing. Aspects for which manual tests are still required, for example aesthetic ones, are in turn facilitated by the rapid prototyping environment – even for single developers that want to test a collaborative multiplayer game.

CONCLUSION

THIS work proposes a concept for authoring collaborative multiplayer games.

More specifically, it aims to remove some of the additional complexity that is intrinsic to the development of multiplayer games. This is especially valuable for smaller or inexperienced development teams.

11.1 MAIN CONTRIBUTIONS

The overall concept is represented through four generally independent modules. Each of these modules is meant to support a specific task during the development process and is directly related to one of the research questions presented in Section 3.2. Thereby, the main contribution of this work lies in the individual concepts underlying these modules.

GAME DESIGN PATTERNS FOR COLLABORATIVE PLAYER INTERACTIONS

Collaborative interactions between the players are a central aspect of a collaborative game. Therefore, it is crucial that developers implement them correctly if they want players to feel as though they are genuinely working together. As such, a collection of proven interactions can be a useful inspiration, especially for novice developers. These interactions could even lead to the development of “ready to use” building blocks, which can also help experienced developers to save time and effort. The concept of game design patterns has been used in the past to describe game design elements, with a widely used description language being available. This language was extended for this work by adding sub-properties that cover aspects specific to collaborative player interactions. For example, these properties include whether two players have to be physically close to each other in order to execute this type of interaction. This makes the patterns more user-friendly and allows developers to find appropriate patterns more quickly. The extended format is put to the test by extracting exemplary interactions from a list of well-received games and transforming them into design patterns. Some of these interaction patterns are then implemented in a game for which two versions are designed. One includes the collaborative interaction patterns as specified, while the other one is built for comparison reasons only and contains largely similar tasks, but does not require any collaboration. This game was evaluated in a user study, which showed that the game was generally well received. Therefore, using game design patterns in an adapted description format as building blocks is a viable option when creating collaborative multiplayer games (**RQ 1**).

STRUCTURAL VERIFICATION OF COLLABORATIVE MULTIPLAYER GAMES

Due to the complexity of multiple players acting concurrently, it is usually impossible to test every possible path that exists in a multiplayer game. However, dead- or livelock situations in which the game cannot continue as planned, are serious is-

sues that should be avoided on any path. A structural verification could conclusively prove that there are no such issues, but existing methods require the user to manually transform the game into a formal model. This does not only require expert knowledge in formal modeling, but is also error-prone and costs time. As a trade-off, only the high-level structure of a game, for example its mission structure, is verified. Our approach, in contrast, is intended to verify each individual game element based on a model that is created automatically. Therefore, rules for algorithmically translating every game element into a verifiable model – in this case colored petri nets – is defined. To reduce the complexity of the resulting net, custom optimization strategies based on specific properties of the game model have been developed. The correctness and scaling properties are evaluated using a large number of synthetic examples as well as some existing games. This showed that, as long as the suggested parameters are chosen, the approach produces the correct results and is a viable option for smaller games. Larger games, in turn can only be verified after they have been separated into independent parts due to the state space explosion. However, for smaller games this means that an automated petri net export and the subsequent verification can be used to detect structural problems in games (**RQ 2**).

COLLABORATIVE BALANCING IN MULTIPLAYER GAMES

If a game does not contain any critical errors, there are other quality characteristics that must be considered next. One aspect is balancing, which is often linked to fairness, but for which only competitive perspectives have been defined. In collaborative games, the contributions of the players must also be balanced, which is covered by a novel balancing definition proposed in this work. From this definition, concrete metrics have been derived that allow the user to measure a game's balance. However, in practice, calculating these metrics on the complete game is often infeasible. For example, the number of paths through a game is infinite if the players are able to return to past states. Instead, an approximation process sampling the paths through the game has been suggested. Again, this approach was evaluated using synthetic and existing games. In these simulations, it produced estimations that matched the values that were expected based on the generation algorithm (for synthetic examples) or previous user studies (for the actual games). With A* as the pathfinding algorithm, its runtime is mostly below one minute, which makes it usable in practice. This shows that the proposed method is a viable approach for quantifying the balance between players playing collaboratively (**RQ 3**).

RAPID PROTOTYPING OF COLLABORATIVE MULTIPLAYER GAMES

Lastly, a rapid prototyping environment has been conceptualized, which allows a single user to easily test a game created for up to four players. In the environment, visual and audio information is given in such a way that the user can notice everything, while being still able to discern which of the players would have normally received this information. This is achieved by a split-screen arrangement and an audio transcript. Additionally, a record and replay queue for simulating concurrent input and several multiplayer visualizations for internal game models have been designed. Using the resulting prototyping environment it became clear that it does indeed allow single developers to test slow-paced games in which multiple players interact with each other (**RQ 4**).

The successful implementation of each module shows that the complexity of some multiplayer game development tasks can be reduced by using specialized tools.

11.2 OUTLOOK

By showing that our concept works, this work builds the ground for further research on applying and refining its individual modules. First, more collaborative games should be analyzed in order to extract a greater variety of player interaction patterns using the same approach. The existing patterns could be organized in a searchable database, integrated into a recommender system or implemented as “ready to use” templates to improve their usability. Additionally, the large data base that was assembled during the pattern game evaluation could be analyzed further. Especially the influence of other factors contributing to the enjoyment of the players, for example how well their fellow players play, could be investigated more deeply. This could yield interesting results from the perspective of the social sciences – not only about the patterns, but also on collaborative games in general.

For both the structural verification and the balancing approximation, this work has focused on the general viability of the results. Further work could also investigate the usability of the results, i.e. whether they can be easily interpreted. The verification could also be made fully automatic, integrating the export implementation and one of the existing petri net verification tools more closely. An in-depth comparison between multiple verification tools could also be interesting, for example to investigate whether they differ in runtime.

In regards to balancing, there are multiple ways in which the current approach could be extended in order to make it easier to use. One could extend the balancing approximation in such a way that it does not only report measurements, but also suggests changes to improve the metrics. These changes could include changing an event’s weight or assigning it to another player. However, this is not trivial, since events are usually encountered on multiple paths. Improving one path could therefore make another one worse. Incorporating suggestions could also lead to completely automated changes, which can then also happen during runtime with concrete players (game adaption). This could be realized by defining a weight for each player, depending on his or her abilities. The weight can then be used in the balancing calculation to model the fact that an action requires more effort by less skilled players. Taking these modified weights into account, the remaining parts of the game could be re-balanced. Similar to the interaction patterns, a user study on the comparison between mathematically calculated and player reported balancing could reveal interesting results from the social sciences’ perspective.

Lastly, although an API for using the rapid prototyping environment with other games has been defined, there is no other game implementing it yet. Using this API for another game or even a game engine could therefore be both beneficial for the game’s developers and a new testcase for the prototyping environment.

BIBLIOGRAPHY

- [1] Ernest Adams. Balancing Games with Positive Feedback, 2002. URL http://www.designersnotebook.com/Columns/043_Positive_Feedback/043_positive_feedback.htm. Last accessed: 2016-05-18.
- [2] Ernest Adams. *Fundamentals of Game Design*. Pearson Education, 2010. ISBN 9780132104753.
- [3] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*, volume 2. Oxford University Press, 1977.
- [4] Gustavo Andrade, Geber Ramalho, Hugo Santana, and Vincent Corruble. Automatic Computer Game Balancing: A Reinforcement Learning Approach. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS, pages 1111–1112, New York, NY, USA, 2005. ACM. ISBN 1-59593-093-0. doi: 10.1145/1082473.1082648.
- [5] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: Exploiting Program Structure for Model Checking Concurrent Software. In *In Concurrency Theory (CONCUR)*, pages 1–15. Springer, 2004.
- [6] Manuel Araújo and Licínio Roque. Modeling Games with Petri Nets. In *Proceedings of DiGRA 2009: Breaking New Ground: Innovation in Games, Play, Practice and Theory*. DiGRA, September 2009.
- [7] Dominic Arsenault. Video Game Genre, Evolution and Innovation. *Journal for Computer Games Culture*, 3(2):149–176, 2009.
- [8] Aida Azadegan and Casper Harteveld. Work for or Against Players: On the Use of Collaboration Engineering for Collaborative Games. In *Foundations of Digital Games*, Fort Lauderdale, USA, 2014.
- [9] Alexander Baldwin, Daniel Johnson, and Peta A. Wyeth. The Effect of Multiplayer Dynamic Difficulty Adjustment on the Player Experience of Video Games. In *CHI ’14 Extended Abstracts on Human Factors in Computing Systems*, CHI EA ’14, pages 1489–1494, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2474-8. doi: 10.1145/2559206.2581285.
- [10] Richard Bartle. Hearts, Clubs, Diamonds, Spades: Players Who Suit MUDS. *Journal of Virtual Environments*, 1(1):19, 1996. URL <http://www.mud.co.uk/richard/hcds.htm>. Last accessed: 2016-05-18.
- [11] Scott Bateman, Regan L. Mandryk, Tadeusz Stach, and Carl Gutwin. Target Assistance for Subtly Balancing Competitive Play. *Proceedings of the 2011 Annual Conference on Human Factors in Computing Systems – CHI ’11*, page 2355, 2011. doi: 10.1145/1978942.1979287.

- [12] Philipp P. Battenberg. Detection of deadlocks in multiplayer games. Bachelor's thesis, TU Darmstadt, 2014.
- [13] April K. Bay-Hinitz, Robert F. Peterson, and H. Robert Quilitch. Cooperative Games: A Way to Modify Aggressive and Cooperative Behaviors in Young Children. *Journal of Applied Behavior Analysis*, 27(3):435–446, 1994. ISSN 1938-3703.
- [14] Karl Bergström, Staffan Björk, and Sus Lundgren. Exploring Aesthetical Game-play Design Patterns: Camaraderie in Four Games. In *Proceedings of the 14th International Academic MindTrek Conference: Envisioning Future Media Environments*, MindTrek '10, pages 17–24, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0011-7. doi: 10.1145/1930488.1930493.
- [15] Dani B. Berry. Imaginary Playmates in Real-time or Why Online Games Suck. In *Computer Game Developers Conference*, 1997.
- [16] Anastasiia Beznosyk, Peter Quax, Karin Coninx, and Wim Lamotte. The Influence of Cooperative Game Design Patterns for Remote Play on Player Experience. In *Proceedings of the 10th Asia Pacific Conference on Computer Human Interaction*, APCHI '12, pages 11–20, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1496-1. doi: 10.1145/2350046.2350051.
- [17] Anastasiia Beznosyk, Peter Quax, Wim Lamotte, and Karin Coninx. The Effect of Closely-Coupled Interaction on Player Experience in Casual Games. In Marc Herrlich, Rainer Malaka, and Maic Masuch, editors, *Entertainment Computing – ICEC 2012 SE – 21*, volume 7522 of *Lecture Notes in Computer Science*, pages 243–255. Springer, 2012. ISBN 978-3-642-33541-9. doi: 10.1007/978-3-642-33542-6_21.
- [18] Staffan Björk. Game Design Patterns Database, 2009. URL <http://129.16.157.67:1337/mediawiki-1.22.0/index.php/Category:Patterns>. Last accessed: 2016-05-19.
- [19] Staffan Björk, Sus Lundgren, and Jussi Holopainen. Game Design Patterns. In Marinka Copier and Joost Raessens, editors, *Level Up – Proceedings of Digital Games Research Conference 2003*, Utrecht, Netherlands, 2003.
- [20] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, volume 1. Wiley, 1996. ISBN 0471958697.
- [21] Thibault Carron, Fabrice Kordon, Jean-Marc Labat, Isabelle Mounier, and Amel Yessad. Toward Improvement of Serious Game Reliability. In *7th European Conference on Games-Based Learning*, pages 80–87. Academic Conferences and Publishing International, 2013.
- [22] Jared E. Cechanowicz, Carl Gutwin, Scott Bateman, Regan Mandryk, and Ian Stavness. Improving Player Balancing in Racing Games. In *Proceedings of the First ACM SIGCHI Annual Symposium on Computer-human Interaction in Play*, CHI PLAY '14, pages 47–56, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3014-5. doi: 10.1145/2658537.2658701.

- [23] Ronan Champagnat, Armelle Prigent, and Pascal Estrailleur. Scenario Building Based on Formal Methods and Adaptative Execution. *ISAGA, Atlanta (USA)*, 6, 2005.
- [24] Haoyang Chen, Yasukuni Mori, and Ikuo Matsuba. Solving the Balance Problem of Massively Multiplayer Online Role-Playing Games Using Coevolutionary Programming. *Applied Soft Computing*, 18(0):1–11, May 2014. ISSN 1568-4946. doi: 10.1016/j.asoc.2014.01.011.
- [25] Allan Cheng, Javier Esparza, and Jens Palsberg. Complexity Results for 1-Safe Nets. In Rudrapatna K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science SE – 28*, volume 761 of *Lecture Notes in Computer Science*, pages 326–337. Springer, 1993. ISBN 978-3-540-57529-0. doi: 10.1007/3-540-57529-4_66.
- [26] Allan Cheng, Søren Christensen, and Kjeld Høyer Mortensen. Model Checking Coloured Petri Nets – Exploiting Strongly Connected Components. *DAIMI Report Series*, 26(519), 1997.
- [27] Giovanni Chiola, Claude Dutheillet, Giuliana Franceschinis, and Serge Hadad. On Well-Formed Coloured Nets and Their Symbolic Reachability Graph. In Kurt Jensen and Grzegorz Rozenberg, editors, *High-level Petri Nets SE – 13*, pages 373–396. Springer, 1991. ISBN 978-3-540-54125-7. doi: 10.1007/978-3-642-84524-6_13.
- [28] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.
- [29] Co-Optimus Staff. Indie-Ana Co-Op and The Dev Stories – You’re All In This Together, 2012. URL <http://www.co-optimus.com/editorial/976/page/1/indie-ana-co-op-and-the-dev-stories-you-re-all-in-this-together.html>. Last accessed: 2016-05-19.
- [30] Co-Optimus Staff. Indie-Ana Co-Op and the Dev Stories – Fostering Gaming Relationships, 2014. URL <http://www.co-optimus.com/editorial/1376/page/1/indie-ana-co-op-and-the-dev-stories-fostering-gaming-relationships.html>. Last accessed: 2016-05-19.
- [31] Co-Optimus Staff. Cooperative Games Database, 2016. URL <http://www.co-optimus.com/games.php>. Last accessed: 2016-01-13.
- [32] Siobhán Corrigan, Rolf Zon, A. Maij, Nick McDonald, and Lena K. Mårtensson. An Approach to Collaborative Learning and the Serious Game Development. *Cognition, Technology & Work*, 17(2):269–278, 2015. ISSN 1435-5558. doi: 10.1007/s10111-014-0289-8.
- [33] Steve Dahlskog and Julian Togelius. Patterns and Procedural Content Generation: Revisiting Mario in World 1 Level 1. In *Proceedings of the First Workshop on Design Patterns in Games*, DPG ’12, pages 1–8, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1854-9. doi: 10.1145/2427116.2427117.

- [34] Olivier Delalleau, Emile Contal, Eric Thibodeau-Laufer, Raul Chandias Ferrari, Yoshua Bengio, and Frank Zhang. Beyond Skill Rating: Advanced Matchmaking in Ghost Recon Online. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(3):167–177, September 2012. ISSN 1943-068X. doi: 10.1109/TCIAIG.2012.2188833.
- [35] Tom DiChristopher. Digital Gaming Sales Hit Record \$61 Billion in 2015: Report, 2016. URL <http://www.cnbc.com/2016/01/26/digital-gaming-sales-hit-record-61-billion-in-2015-report.html>. Last accessed: 2016-05-19.
- [36] Joris Dormans. Machinations: Elemental Feedback Patterns for Game Design. In Joseph Saur and Margaret Loper, editors, *GAME-ON-NA 2009: 5th International North American Conference on Intelligent Games and Simulation*, pages 33–40, 2009.
- [37] Abdelkarim El Moussaoui, Christian Reuter, Maurice Wiegel, Sven Unkauf, Tanja Wiesmann, Mareike Dornhöfer, Madjid Fathi, and Sara Nasiri. NeuroCare: Digitalisierte Früherkennung leichter kognitiver Einschränkungen. In *AAL-Kongress 2015*. VDE Verlag, 2015.
- [38] Carlo Fabricatore. Gameplay and Game Mechanics: A Key to Quality in Videogames. In *Proceedings of the OECD-CERI Expert Meeting on Videogames and Education*, Santiago de Chile, Chile, 2007.
- [39] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987. ISSN 0004-5411. doi: 10.1145/28869.28874.
- [40] Moby Games. Games Database, 2016. URL <http://www.mobygames.com/browse/games>. Last accessed: 2016-01-13.
- [41] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In Oscar M. Nierstrasz, editor, *ECOOP'93 – Object-Oriented Programming SE – 21*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431. Springer, 1993. ISBN 978-3-540-57120-9. doi: 10.1007/3-540-47910-4_21.
- [42] Stefan Göbel, Viktor Wendel, Christopher Ritter, and Ralf Steinmetz. Personalized, Adaptive Digital Educational Games Using Narrative, Game-Based Learning Objects. In *Proceedings of Edutainment 2010*, pages 438–445, aug 2010. ISBN 9783642145322.
- [43] Wooi-boon Goh, Fitriani, Lum Guan Ting, Wei Shou, Chun-Fan Goh, Monica Menon, Jacquelyn Tan, and Libby Gordon Cohen. Potential Challenges in Collaborative Game Design for Inclusive Settings. In *ACM CHI Conference on Human Factors in Computing Systems*, pages 1–4, 2011. ISBN 9781450302685.
- [44] Jose Luis González Sánchez, Natalia Padilla Zea, and Francisco L. Gutiérrez. From Usability to Playability: Introduction to Player-Centred Video Game Development Process. In Masaaki Kurosu, editor, *Human Centered Design SE – 9*, volume 5619 of *Lecture Notes in Computer Science*, pages 65–74. Springer, 2009. ISBN 978-3-642-02805-2. doi: 10.1007/978-3-642-02806-9\9.

- [45] Tobias Greitemeyer and Christopher Cox. There's no "I" in Team: Effects of Cooperative Video Games on Cooperative Behavior. *European Journal of Social Psychology*, 43(3):224–228, 2013. ISSN 1099-0992. doi: 10.1002/ejsp.1940.
- [46] Fabian Groh. Calculating Solutions for Collaborative Multiplayer Adventure Games. Bachelor's thesis, TU Darmstadt, 2014.
- [47] Sandro Hardy, Christian Reuter, Stefan Göbel, Ralf Steinmetz, Gisa Baller, Elke Kalbe, Abdelkarim El Moussaoui, Sven Abels, Susanne Dienst, Mareike Dornhöfer, and Madjid Fathi. NeuroCare – Personalization and Adaptation of Digital Training Programs for Mild Cognitive Impairments. In *AAL-Kongress 2014, Tagungsband*. VDE Verlag, 2014.
- [48] Casper Harteveld and Geertje Bekebrede. Learning in Single-Versus Multiplayer Games: The More the Merrier? *Simulation & Gaming*, 42(1):43–63, August 2011. ISSN 1046-8781. doi: 10.1177/1046878110378706.
- [49] Sebastian Haufe, Stephan Schiffel, and Michael Thielscher. Automated Verification of State Sequence Invariants in General Game Playing. *Artificial Intelligence*, 187–188:1–30, August 2012. ISSN 0004-3702. doi: 10.1016/j.artint.2012.04.003.
- [50] Davinia Hernández-Leo, Eloy D. Villasclaras-Fernández, Juan I. Asensio-Pérez, Iván M. Jorrín-Abellán, Inés Ruiz-Requies, and Bartolomé Rubia-Avi. COLAGE: A Collaborative Learning Design Editor Based on Patterns. *Educational Technology and Society*, 9(1):58–71, 2006.
- [51] Stefan Hoermann, Tomas Hildebrandt, Christoph Rensing, and Ralf Steinmetz. ResourceCenter – A Digital Learning Object Repository with an Integrated Authoring Tool Set. In Piet Kommers and Griff Richards, editors, *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications ED-MEDIA 2005*, pages 3453–3460, June 2005.
- [52] Carolina Islas Sedano, Maira B. Carvalho, Nicola Secco, and C. Shaun Longstreet. Collaborative and Cooperative Games: Facts and Assumptions, 2013.
- [53] Kurt Jensen. Coloured Petri nets. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets: Central Models and Their Properties SE – 10*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer, 1987. ISBN 978-3-540-17905-4. doi: 10.1007/BFb0046842.
- [54] Kurt Jensen, Lars Michael Kristensen, and Lisa 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. ISSN 1433-2779. doi: 10.1007/s10009-007-0038-x.
- [55] Felix Kamieth, Tim Dutz, Pia Weiss, Stefanie Müller, Christian Reuter, Reiner Wichert, Peter Klein, and Stefan Göbel. Navigationsassistenz für ältere Menschen im öffentlichen Nahverkehr. In *Lebensqualität im Wandel von Demografie und Technik – 6. Deutscher AAL-Kongress*, Berlin, jan 2013. VDE Verlag.

- [56] Tim Keenan. Effective Co-Op Design, 2012. URL http://www.gamasutra.com/view/feature/176453/effective_coop_design.php. Last accessed: 2016-05-19.
- [57] Scott Kim. Multiplayer Puzzles, 2005. URL <http://www.scottkim.com.previewc40.carrierzone.com/thinkinggames/multiplayerpuzzles/index.html>. Last accessed: 2016-05-19.
- [58] Raph Koster. Social Mechanics – The Engines Behind Everything Multiplayer. In *GDC 2011*, 2011. doi: [10.1177/019263654102510024](https://doi.org/10.1177/019263654102510024).
- [59] Bernd Kreimeier. The Case For Game Design Patterns, 2002. URL http://www.gamasutra.com/view/feature/132649/the_case_for_game_design_patterns.php. Last accessed: 2016-01-21.
- [60] Urmila Kukreja, William E. Stevenson, and Frank E. Ritter. RUI: Recording User Input from Interfaces under Windows and Mac OS X. *Behavior Research Methods*, 38(4):656–659, 2006. ISSN 1554-351X. doi: [10.3758/BF03193898](https://doi.org/10.3758/BF03193898).
- [61] Alen Ladavac. Fast Iteration Tools in the Production of the Talos Principle. In *GDC Europe*, 2015.
- [62] L. H. Landweber and E. L. Robertson. Properties of Conflict-Free and Persistent Petri Nets. *Journal of the ACM*, 25(3):352–364, 1978. ISSN 00045411. doi: [10.1145/322077.322079](https://doi.org/10.1145/322077.322079).
- [63] Petri Lankoski and Staffan Björk. Theory Lenses: Deriving Gameplay Design Patterns from Theories. In *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments*, pages 16–21. ACM, 2011. ISBN 9781450308168.
- [64] Peter Laurens, Richard F. Paige, Phillip J. Brooke, and Howard Chivers. A Novel Approach to the Detection of Cheating in Multiplayer Online Games. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 97–106. Ieee, 2007. ISBN 0-7695-2895-3. doi: [10.1109/ICECCS.2007.11](https://doi.org/10.1109/ICECCS.2007.11).
- [65] K.-H. Lee and Joël Favrel. Hierarchical Reduction Method for Analysis and Decomposition of Petri Nets. *IEEE Transactions on Systems, Man and Cybernetics, SMC-15*(2):272–280, 1985. doi: [10.1109/TSMC.1985.6313357](https://doi.org/10.1109/TSMC.1985.6313357).
- [66] Ryan Leigh, Justin Schonfeld, and Sushil J. Louis. Using Coevolution to Understand and Validate Game Balance in Continuous Games. *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation – GECCO '08*, page 1563, 2008. doi: [10.1145/1389095.1389394](https://doi.org/10.1145/1389095.1389394).
- [67] Fei Liu and Monika Heiner. Computation of Enabled Transition Instances for Colored Petri Nets. In *Proc. AWPN*, volume 643, pages 51–65, 2010.
- [68] Tony Manninen. Interaction Forms in Multiplayer Desktop Virtual Reality Games. In Simon Richir, Paul Richard, and Bernard Tarave, editors, *Proceedings of VRIC 2002 Conference*, pages 223–232, 2002.

- [69] Tony Manninen and Tuomo Korva. Designing Puzzles for Collaborative Gaming Experience – CASE: eScape. In *Selected Papers of the Digital Interactive Games Research Association's Second International Conference (DiGRA 2005)*, pages 233–247, Vancouver, Canada, 2005. Digital Interactive Games Research Association.
- [70] Eugenio J. Marchiori, Ángel del Blanco, Javier Torrente, Iván Martínez-Ortiz, and Baltasar Fernández-Manjón. A Visual Language for the Creation of Narrative Educational Games. *Journal of Visual Languages & Computing*, 22(6):443–452, December 2011. ISSN 1045926X. doi: 10.1016/j.jvlc.2011.09.001.
- [71] Martin Mauve, Jürgen Vogel, Volker Hilt, and Wolfgang Effelsberg. Local-Lag and Timewarp: Providing Consistency for Replicated Continuous Applications. *IEEE Transactions on Multimedia*, 6(1):47–57, feb 2004. ISSN 1520-9210. doi: 10.1109/TMM.2003.819751.
- [72] Kevin McGee. Patterns and Computer Game Design Innovation. In *Proceedings of the 4th Australasian Conference on Interactive Entertainment*, IE '07, pages 1–16, Melbourne, Australia, Australia, 2007. RMIT University. ISBN 978-1-921166-87-7.
- [73] Florian Mehm. *Authoring of Adaptive Single-Player Educational Games*. PhD thesis, TU Darmstadt, 2013.
- [74] Florian Mehm, Stefan Göbel, Sabrina Radke, and Ralf Steinmetz. Authoring Environment for Story-Based Digital Educational Games. In Yiwei Cao, Anna Hannemann, Baltasar Fernández-Manjón, Stefan Göbel, Cord Hockemeyer, and Emmanuel Stefanakis, editors, *Proceedings of the 1st International Open Workshop on Intelligent Personalization and Adaptation in Digital Educational Games*, pages 113–124. CEUR Workshop, 2009.
- [75] Florian Mehm, Viktor Wendel, Stefan Göbel, and Ralf Steinmetz. Bat Cave: A Testing and Evaluation Platform for Digital Educational Games. In *Proceedings of the 4th European Conference on Games Based Learning*, pages 251–260, 2010. ISBN 9781906638795.
- [76] Gerard Meszaros and Jim Doble. A Pattern Language for Pattern Writing. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, chapter A Pattern, pages 529–574. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0-201-31011-2.
- [77] Pablo Moreno-Ger, Rubén Fuentes-Fernández, José-Luis Sierra-Rodríguez, and Baltasar Fernández-Manjón. Model-Checking for Adventure Videogames. *Information and Software Technology*, 51(3):564–580, March 2009. ISSN 09505849. doi: 10.1016/j.infsof.2008.08.003.
- [78] Stéphane Natkin and Liliana Vega. A Petri Net Model for the Analysis of the Ordering of Actions in Computer Games. In *GAME ON*, pages 82–89, 2003.
- [79] Stéphane Natkin, Liliana Vega, and Stefan M. Grünvogel. A New Methodology for Spatiotemporal Game Design. In *Proceedings of CGAIDE*, pages 109–113, 2004.

- [80] Mark Newheiser. Playing Fair: A Look at Competition in Gaming, 2009. URL <http://www.strangehorizons.com/2009/20090309/newheiser-a.shtml>. Last accessed: 2016-05-19.
- [81] Carl Magnus Olsson, Staffan Björk, and Steve Dahlskog. The Conceptual Relationship Model: Understanding Patterns and Mechanics in Game Design. In *Proceedings of DiGRA 2014: <Verb that ends in 'ing'> the <noun> of Game <plural noun>*, page 16, 2014.
- [82] Joseph Carter Osborn, April Grow, and Michael Mateas. Modular Computational Critics for Games. In *AIIDE*, 2013.
- [83] Michael Parker. Designing Multiplayer Team-Based Mechanics without Adding Frustration, 2012. URL <http://www.gamasutra.com/blogs/MichaelParker/20120921/91039/>. Last accessed: 2016-05-19.
- [84] Panagiotis Petridis, Ian Dunwell, Sara de Freitas, and David Panzoli. An Engine Selection Methodology for High Fidelity Serious Games. *2010 Second International Conference on Games and Virtual Worlds for Serious Applications*, pages 27–34, March 2010. doi: 10.1109/VS-GAMES.2010.26.
- [85] Christopher J. F. Pickett. Formal Verification of Computer Narratives. Technical report, McGill University, 2005.
- [86] David Pinelle, Nelson Wong, Tadeusz Stach, and Carl Gutwin. Usability Heuristics for Networked Multiplayer Games. In *GROUP'09*, pages 169–178, Sanibel Island, Florida, USA, 2009. ACM. ISBN 9781605585000.
- [87] Andreas Pleuss. MML: A Language for Modeling Interactive Multimedia Applications. In *Seventh IEEE International Symposium on Multimedia*, page 9, 2005. doi: 10.1109/ISM.2005.80.
- [88] Nick Puleo. When Co-Op Becomes a Necessity – The Changing Face of Single Player, 2010. URL <http://www.co-optimus.com/editorial/479/page/1/when-co-op-becomes-a-necessity-the-changing-face-of-single-player.html>. Last accessed: 2016-05-19.
- [89] Barbara Reichart and Bernd Bruegge. Social Interaction Patterns for Learning in Serious Games. In *Proceedings of the 19th European Conference on Pattern Languages of Programs*, EuroPLoP '14, pages 1–7, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3416-7. doi: 10.1145/2721956.2721985.
- [90] Christian Reuter. Development and Realization of Methods and Concepts for Multiplayer Adventures. Master's thesis, TU Darmstadt, 2011.
- [91] Christian Reuter, Viktor Wendel, Stefan Göbel, and Ralf Steinmetz. Multiplayer Adventures for Collaborative Learning With Serious Games. In Patrick Felicia, editor, *6th European Conference on Games Based Learning*, pages 416–423. Academic Conferences Limited, 2012.
- [92] Christian Reuter, Thomas Tregel, Florian Mehm, Stefan Göbel, and Ralf Steinmetz. Rapid Prototyping for Multiplayer Serious Games. In Carsten Busch,

- editor, *Proceedings of the 8th European Conference on Games Based Learning*, pages 478–486. Academic Conferences Limited, 2014. ISBN 978-1-910309-55-1.
- [93] Christian Reuter, Viktor Wendel, Stefan Göbel, and Ralf Steinmetz. Game Design Patterns for Collaborative Player Interactions. In *Proceedings of DiGRA 2014: <Verb that ends in 'ing'> the <noun> of Game <plural noun>*, Salt Lake City, USA, 2014.
 - [94] Christian Reuter, Stefan Göbel, and Ralf Steinmetz. Detecting Structural Errors in Scene-Based Multiplayer Games Using Automatically Generated Petri Nets. In *Foundations of Digital Games*, Pacific Grove, USA, 2015.
 - [95] Christian Reuter, Stefan Göbel, and Ralf Steinmetz. Approximating Balance in Collaborative Multiplayer Games. In Robin Munkvold and Line Kolas, editors, *Proceedings of the 9th European Conference on Games Based Learning*, pages 449–455. Academic Conferences Limited, 2015. ISBN 978-1-910810-58-3.
 - [96] Emanuel Montero Reyno and José Á. Carsí Cubel. A Platform-Independent Model for Videogame Gameplay Specification. In *Proceedings of DiGRA 2009: Breaking New Ground: Innovation in Games, Play, Practice and Theory*. DiGRA, September 2009.
 - [97] José Bernardo Rocha, Samuel Mascarenhas, and Rui Prada. *Game Mechanics for Cooperative Games*, pages 73–80. Universidade do Minho, 2008. ISBN 978-989-95500-2-5.
 - [98] Jeremy Roschelle and Stephanie D. Teasley. The Construction of Shared Knowledge in Collaborative Problem Solving. In *Computer Supported Collaborative Learning*, pages 418–420. Springer, 1995. doi: 10.1177/004057368303900411.
 - [99] Richard Rouse III. *Game Design: Theory & Practice*. Wordware Publishing, Inc., 2nd edition, 2004. ISBN 1556229127.
 - [100] Ji Ruan, Wiebe van der Hoek, and Michael Wooldridge. Verification of Games in the Game Description Language. *Journal of Logic and Computation*, 19(6): 1127–1156, December 2009. doi: 10.1093/logcom/exp039.
 - [101] Vikram Saralaya, J. K. Kishore, Sateesh Reddy, Radhika M. Pai, and Sanjay Singh. Modeling and Verification of Chess Game Using NuSMV. In Ajith Abraham, Jaime Lloret Mauri, John F. Buford, Junichi Suzuki, and Sabu M. Thampi, editors, *Advances in Computing and Communications SE – 47*, volume 191 of *Communications in Computer and Information Science*, pages 460–470. Springer, 2011. ISBN 978-3-642-22713-4. doi: 10.1007/978-3-642-22714-1_47.
 - [102] Kathrin Schölei. Interaction Templates for Collaborative Adventure Games (German). Bachelor’s thesis, TU Darmstadt, 2015.
 - [103] Ian Schreiber. Level 16: Game Balance, 2009. URL <https://gamedesignconcepts.wordpress.com/2009/08/20/level-16-game-balance/>. Last accessed: 2016-05-19.

- [104] Magy Seif El-Nasr, Bardia Aghabeigi, David Milam, Mona Erfani, Beth Lame-man, Hamid Maygoli, and Sang Mah. Understanding and Evaluating Cooperative Games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 253–262, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-929-9. doi: 10.1145/1753326.1753363.
- [105] Miguel Sicart. Defining Game Mechanics. *The International Journal of Computer Game Research*, 8(2):1, 2008.
- [106] David Sirlin. Balancing Multiplayer Games, 2002. URL <http://www.sirlin.net/articles/balancing-multiplayer-games-part-1-definitions>. Last accessed: 2016-05-19.
- [107] Adam M. Smith and Michael Mateas. Answer Set Programming for Procedural Content Generation: A Design Space Approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, 2011. doi: 10.1109/TCIAIG.2011.2158545.
- [108] Gillian Smith, Mee Cha, and Jim Whitehead. A Framework for Analysis of 2D Platformer Levels. In *Proceedings of the 2008 ACM SIGGRAPH Symposium on Video Games*, Sandbox '08, pages 75–80, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-173-6. doi: 10.1145/1401843.1401858.
- [109] Ralf Steinmetz and Klara Nahrstedt. *Multimedia Systems*. X.media.publishing. Springer, 2013. ISBN 9783662088784.
- [110] Jan-Willem Strijbos and Maarten F. De Laat. Developing the Role Concept for Computer-Supported Collaborative Learning: An Explorative Synthesis. *Computers in Human Behavior*, 26(4):495–505, July 2010. ISSN 0747-5632. doi: 10.1016/j.chb.2009.08.014.
- [111] Moh. Aries Syufagi, Mochamad Hariadi, and Mauridhi Hery Purnomo. Petri Net Model for Serious Games Based on Motivation Behavior Classification. *International Journal of Computer Games Technology*, 2013:1–12, 2013. ISSN 1687-7047. doi: 10.1155/2013/851287.
- [112] Kuo-Chung Tai. Definitions and detection of deadlock, livelock, and starvation in concurrent programs. In *International Conference on Parallel Processing*, volume 2, pages 69–72, Aug 1994. doi: 10.1109/ICPP.1994.84.
- [113] Mark John Taylor, David Gresty, and Michael Baskett. Computer Game-Flow Design. *Computers in Entertainment*, 4(1):5, January 2006. ISSN 1544-3574. doi: 10.1145/1111293.1111300.
- [114] Richard Terrell. Shared-Multi-Split Screen Design, 2011. URL http://gamasutra.com/blogs/RichardTerrell/20110617/88846/SharedMultiSplit_{_}Screen_{_}Design.php. Last accessed: 2016-05-19.
- [115] Javier Torrente, Pablo Moreno-Ger, Iván Martínez-Ortiz, and Baltasar Fernandez-Manjon. Integration and Deployment of Educational Games in E-Learning Environments: The Learning Object Model Meets Educational Gaming E-Learning and Videogames. *Educational Technology & Society*, 12(4):359–371, 2009.

- [116] Thomas Tregel. Design and implementation of a rapid prototyping-environment for multiplayer adventures. Bachelor's thesis, TU Darmstadt, 2013.
- [117] Thomas Tregel. Procedural Generation of Multiplayer Games Based on Game Design Patterns for Collaborative Player Interactions. Master's thesis, TU Darmstadt, 2015.
- [118] Roman Uhlig. Design and Development of a Collaborative Multiplayer Game Using Design Patterns. Bachelor's thesis, TU Darmstadt, 2014.
- [119] Marynel Vázquez, Alexander May, Wei-hsuan Chen, and Aaron Steinfeld. A Deceptive Robot Referee in a Multiplayer Gaming Environment. In *International Conference on Collaboration Technologies and Systems (CTS)*, pages 204–211, Philadelphia, PA, 2011. IEEE. ISBN 978-1-61284-638-5.
- [120] John A. Velez, Tobias Greitemeyer, Jodi L. Whitaker, David R. Ewoldsen, and Brad J. Bushman. Violent Video Games and Reciprocity: The Attenuating Effects of Cooperative Game Play on Subsequent Aggression. *Communication Research*, September 2014. doi: 10.1177/0093650214552519.
- [121] Clark Verbrugge. A Structure for Modern Computer Narratives. In Jonathan Schaeffer, Martin Müller, and Yngvi Björnsson, editors, *Computers and Games SE – 21*, volume 2883 of *Lecture Notes in Computer Science*, pages 308–325. Springer, 2003. ISBN 978-3-540-20545-6. doi: 10.1007/978-3-540-40031-8_21.
- [122] Iro Voulgari and Vassilis Komis. Antecedents of Collaborative Learning: Insights from Massively Multiplayer Online Games. *2010 International Conference on Intelligent Networking and Collaborative Systems*, pages 32–39, November 2010. doi: 10.1109/INCOS.2010.79.
- [123] Tim Wellhausen and Andreas Fießer. How to Write a Pattern? A Rough Guide for First-Time Pattern Authors. In *EuroPLoP '11 – 16th European Conference on Pattern Languages of Programs*, page 13, 2011. ISBN 9781450313025. doi: 10.1145/2396716.2396721.
- [124] Viktor Wendel. *Collaborative Game-Based Learning – Automatized Adaptation Mechanics for Game-Based Collaborative Learning Using Game Mastering Concepts*. PhD thesis, TU Darmstadt, 2015.
- [125] Viktor Wendel, Michael Gutjahr, Stefan Göbel, and Ralf Steinmetz. Designing Collaborative Multiplayer Serious Games. *Education and Information Technologies*, 2(18):287–308, June 2013. ISSN 1360-2357.
- [126] Dung-Chiuan Wu, Huan-Yu Lin, Shian-Shyong Tseng, Jui-Feng Weng, and Jun-Ming Su. Constructing a Role Playing Interactive Learning Content Model. In *ISMW '07. Ninth IEEE International Symposium on Multimedia Workshops*, pages 15–22, 2007. ISBN VO -. doi: 10.1109/ISM.Workshops.2007.9.
- [127] Amel Yessad, Jean-Marc Labat, and François Kermorvant. SeGAE: A Serious Game Authoring Environment. *2010 10th IEEE International Conference on Advanced Learning Technologies*, pages 538–540, July 2010. doi: 10.1109/ICALT.2010.153.

- [128] Amel Yessad, Pradeepa Thomas, Bruno Capdevila, and Jean-Marc Labat. Using the Petri Nets for the Learner Assessment in Serious Games. In Xiangfeng Luo, Marc Spaniol, Lizhe Wang, Qing Li, Wolfgang Nejdl, and Wu Zhang, editors, *Advances in Web-Based Learning – ICWL 2010*, volume 6483 of *Lecture Notes in Computer Science*, pages 339–348. Springer, 2010. ISBN 978-3-642-17406-3. doi: [10.1007/978-3-642-17407-0_35](https://doi.org/10.1007/978-3-642-17407-0_35).
- [129] Siu Fung Yeung, John C. S. Lui, Jiangchuan Liu, and Jeff Yan. Detecting Cheaters for Multiplayer Games: Theory, Design and Implementation. In *CCNC 2006. 2006 3rd IEEE Consumer Communications and Networking Conference*, 2006., volume 2, pages 1178–1182. Ieee, 2006. ISBN 1-4244-0085-6. doi: [10.1109/CCNC.2006.1593224](https://doi.org/10.1109/CCNC.2006.1593224).
- [130] José Pablo Zagal, Miguel Nussbaum, and Ricardo Rosas. A Model to Support the Design of Multiplayer Games. *Presence: Teleoper. Virtual Environ.*, 9(5):448–462, 2000. ISSN 1054-7460.
- [131] José Pablo Zagal, Michael Mateas, Clara Fernández-Vara, Brian Hochhalter, and Nolan Lichti. Towards an Ontological Language for Game Analysis. In *Proceedings of the 2005 DiGRA International Conference: Changing Views: Worlds in Play*, 2005.
- [132] José Pablo Zagal, Jochen Rick, and Idris Hsi. Collaborative Games: Lessons Learned from Board Games. *Simulation & Gaming*, 37(1):24–40, March 2006. ISSN 1046-8781. doi: [10.1177/1046878105282279](https://doi.org/10.1177/1046878105282279).
- [133] Sebastian Zander, Ian Leeder, and Grenville Armitage. Achieving Fairness in Multiplayer Network Games through Automated Latency Balancing. *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology – ACE ’05*, pages 117–124, 2005. doi: [10.1145/1178477.1178493](https://doi.org/10.1145/1178477.1178493).
- [134] Natalia Padilla Zea, José Luís González Sánchez, Francisco L. Gutiérrez, Marcelino J. Cabrera, and P. Paderewski. Design of Educational Multiplayer Videogames: A Vision From Collaborative Learning. *Advances in Engineering Software*, 40(12):1251–1260, December 2009. ISSN 09659978. doi: [10.1016/j.advengsoft.2009.01.023](https://doi.org/10.1016/j.advengsoft.2009.01.023).
- [135] Birgit Zimmermann. *Pattern-basierte Prozessbeschreibung und -unterstützung: Ein Werkzeug zur Unterstützung von Prozessen zur Anpassung von E-Learning-Materialien*. PhD thesis, TU Darmstadt, 2008.

LIST OF FIGURES

Figure 1	Cooperative games released per year	2
Figure 2	Colored petri net example	16
Figure 3	Screen layout when testing in the <i>Portal 2</i> -Editor	20
Figure 4	Concept for authoring collaborative multiplayer games	25
Figure 5	Game model: event with exclusive conditions	28
Figure 6	Game model: location hierarchy	29
Figure 7	Game model: event with non-exclusive conditions	29
Figure 8	Interactions between authoring and runtime environment	30
Figure 9	Multiplayer communication in the runtime environment	31
Figure 10	Event scope in a multiplayer scenario	32
Figure 11	Pattern visualization: logical relations	39
Figure 12	Pattern visualization: logical and spatial relations	40
Figure 13	Pattern visualization: complex relations	40
Figure 14	Collaborative and non-collaborative version of the same level ..	41
Figure 15	Petri net model: game state	49
Figure 16	Event linearization	50
Figure 17	Petri net model: reading a boolean variable	51
Figure 18	Petri net model: player triggering events	51
Figure 19	Petri net model: writing a boolean variable	51
Figure 20	Petri net model: player moving	51
Figure 21	Petri net model: complete example	53
Figure 22	Petri net model: dedicated ending	54
Figure 23	Petri net model: checking the presence of another player	56
Figure 24	Event linearization with non-exclusive branches	57
Figure 25	Petri net model: complete example (alternate model)	58
Figure 26	Event reorganization after reaction removal	61
Figure 27	Event optimization	61
Figure 28	Game partitioning	62
Figure 29	Game partitioning with side effects	63
Figure 30	Generated game structure	64
Figure 31	Balancing: pathfinding pitfalls	72
Figure 32	Balancing: state space example	73
Figure 33	Balancing: different paths for each player	75
Figure 34	Balancing: multiple shortest paths for one player	90
Figure 35	Balancing: superfluous endings	92
Figure 36	Prototyping: split-screen setup	97
Figure 37	Prototyping: player model comparison	98
Figure 38	Prototyping: knowledge space comparison	98
Figure 39	Prototyping: location history comparison	99
Figure 40	Prototyping: sound log	100
Figure 41	Prototyping: record and replay queue	101
Figure 42	Procedural generated pattern structure	106

Figure 43	Pattern implementations in the evaluation game	107
Figure 44	Exported petri net layout.....	108
Figure 45	Example state space.....	110
Figure 46	Multiplayer prototyping environment.....	113
Figure 47	Pattern game results: enjoyment questions	118
Figure 48	Pattern game results: difficulty questions.....	119
Figure 49	Pattern game results: playtime logging.....	120
Figure 50	Pattern game results: communication protocol	121
Figure 51	Example scenario: boosting another player	122
Figure 52	Verification results: game lengths.....	125
Figure 53	Verification results: player counts	125
Figure 54	Verification results: branches (“locked door” task).....	126
Figure 55	Verification results: branches (no task).....	126
Figure 56	Game structures.....	127

LIST OF TABLES

Table 1	Game design pattern “Multiplayer Games”.....	11
Table 2	Generated puzzle structure.....	65
Table 3	Rapid prototyping API functions.....	102
Table 4	Data mapping for pattern evaluation questions.....	117
Table 5	Verification results: export (“locked door” task).....	123
Table 6	Verification results: export (“puzzle” task).....	124
Table 7	Verification results: alternate export (“locked door” task).....	124
Table 8	Verification results: vacation and tourism.....	128
Table 9	Verification results: multiplayer adventure	130
Table 10	Verification results: wire puzzle	131
Table 11	Balancing results: pathfinding (generated).....	132
Table 12	Balancing results: task assignments (generated)	134
Table 13	Balancing results: sections (generated)	135
Table 14	Balancing results: sections, multiple paths (generated).....	136
Table 15	Balancing results: simplified calculation (generated).....	137
Table 16	Balancing results: pathfinding (adventure)	137
Table 17	Balancing results: weight heuristics (adventure)	138
Table 18	Balancing results: sections (adventure)	139
Table 19	Balancing results: simplified calculation (adventure)	140
Table 20	Balancing results: final assessment “Multiplayer Adventure” ...	140
Table 21	Pattern language comparison.....	163
Table 22	Full pattern game results: enjoyment questions.....	195
Table 23	Full pattern game results: difficulty questions	196
Table 24	Full pattern game results: playtime logging	197
Table 25	Full pattern game results: communication protocol.....	198
Table 26	Full verification results: game lengths	199
Table 27	Full verification results: player counts	200
Table 28	Full verification results: branches.....	201

A

PATTERN LANGUAGE COMPARISON

SOFTWARE DESIGN PATTERNS				GAME DESIGN PATTERNS		
	[41]	[76]	[20]	[59]	[19, 18]	[72]
IDENTIFIER	Name	Name, Aliases*	Name, Also Known As	Name	Name	Name
PROBLEM	Intent, Motivation	Problem, Indications*	Problem	Problem	Description	Forces
CONSTRAINTS	Applicability	Context	Context	-	-	-
SOLUTION	Participants, Collaborations	Solution	Solution, Structure	Solution	-	Features
CONSEQUENCES	Conseq.	Forces, Resulting Context*	Conseq., Dynamics	Conseq.	Conseq.	-
IMPLEMENTATION	Diagram, Implement.	CodeSamples*	Implement.	-	Using the Pattern (Diegetic, Narrative)	-
EXAMPLES	Examples	Examples*	Example, Example Resolved	-	Examples	-
RELATIONS	See Also	Related Patterns*	See Also, Variants	-	Relations	-
META	-	Rationale*, Acknowled- gments*	Known Uses	-	-	-

Table 21: Comparison of pattern description languages (* marks optional parameters).

B

ANALYZED GAMES

COOPERATIVE GAMES:

- *Army of Two* (Electronic Arts Montreal, 2008)
- *Artemis Spaceship Bridge Simulator 2.0* (Thom Robertson, 2013)
- *Borderlands 2* (Gearbox Software, 2012)
- *Castle Crashers* (The Behemoth, 2008)
- *Eve Online* (CCP Games, 2003)
- *Far Cry 3* (Ubisoft Montreal, 2012)
- *F.E.A.R. 3* (Day 1 Studios, 2011)
- *Forced* (BetaDwarf, 2013)
- *ibb & obb* (Sparpweed, 2014)
- *Kane & Lynch 2: Dog Days* (IO Interactive, 2010)
- *Lara Croft and the Guardian of Light* (Crystal Dynamics, 2010)
- *Left 4 Dead* (Turtle Rock Studios, 2008)
- *Left 4 Dead 2* (Turtle Rock Studios, 2009)
- *Lego Indiana Jones: The Original Adventures* (Traveller's Tales, 2008)
- *Portal 2* (Valve Corporation, 2011)
- *Resident Evil 5* (Capcom, 2009)

STRATEGY GAME WITH TEAM BASED MODES:

- *Age of Empires II: The Age of Kings* (Ensemble Studios, 1999)
- *Age of Empires III* (Ensemble Studios, 2005)
- *StarCraft II* (Blizzard Entertainment, 2008)

TEAM-BASED SHOOTER GAMES:

- *Battlefield 4* (DICE, 2013)
- *Brink* (Splash Damage, 2011)
- *Counter-Strike* (Valve Corporation, 1999)
- *Dirty Bomb* (Splash Damage, 2015)
- *Natural Selection 2* (Unknown Worlds Entertainment, 2012)
- *Unreal Tournament 2004* (Epic Games, 2004)

MOBA (MULTIPLAYER ONLINE BATTLE ARENA) GAMES:

- *League of Legends* (Riot Games, 2009)
- *Dota 2* (Valve Corporation, 2013)

SANDBOX GAMES:

- *Minecraft* (Mojang, 2011)

MMORPGS (MASSIVE MULTIPLAYER ONLINE ROLEPLAYING GAMES):

- *World of Warcraft* (Blizzard Entertainment, 2004)

HERE the complete pattern collection described in [93] is reproduced for completeness. The related patterns are classified as “Superior” (more abstract), “Subpatterns” (possible implementations) or “Conflicting” and reference entries from collections mentioned in the related work [18, 97, 104].

C.1 GENERAL

Name	Concurrency
Description	Operating one or more objects simultaneously that could not be operated by a single player alone.
Examples	Genre independent: <ul style="list-style-type: none"> • Moving a heavy object (<i>Resident Evil 5</i>) • Enemies that must be flanked (<i>Army of Two</i>)
Consequences	Always mandatory at a specific location. Medium communication and high timing requirements, may break immersion if not justified properly.
Using the pattern	Can be used for collecting or separating players (position of interaction elements), length of interaction can be varied between short and long (depending on the animation), compatible with fixed or free roles (availability of interaction elements). Can be designed for an arbitrary player number (number of interaction elements).
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Synergies Between Abilities, Interacting With The Same Object; Subpatterns: Team Combos, Vulnerabilities

Name	Parallelization
Description	Splitting work that could be done alone in order to speed it up or to make it easier.
Examples	Genre independent: <ul style="list-style-type: none"> • Players splitting up to search for materials (<i>Minecraft</i>)
Consequences	Always voluntary separation with free roles, often for a longer time. Low communication and timing requirements, benefit must be noticeable by players in order to occur.
Using the pattern	Can be used for collecting or separating players (position of subtask elements) and in specific or pervasive locations, works for an arbitrary number of players (but participation is voluntary).
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment; Subpatterns: Races

Name	Joint Decision Making
Description	Letting players decide between multiple options with the result impacting the whole team.
Examples	Genre independent: <ul style="list-style-type: none"> • Choosing between different quest resolutions and rewards (initial version of <i>Borderlands 2</i>, later the decision could be made by each player individually)
Consequences	Deciding should always take a short time mechanically (although the discussion between players might take longer). High communication and low timing requirements, works for an arbitrary number of players. Players whose opinion was ignored might feel bad, depending on the importance of the decision.
Using the pattern	Can be used in specific (e.g. virtual character asking the players) or pervasive (as part of the user interface) locations and for collecting or separating players (voting location), with fixed (decision leader) or free (voting) roles and mandatory or voluntary, works for an arbitrary number of players. Can be combined with Contest-Pattern for voting rights.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Freedom of Choice; Subpatterns: Actions Have Diegetically Social Consequences

C.2 PROGRESSION GATES

Name	Separation Gate
Description	Forcing the players to split up by allowing only a certain number of players to continue while the others need to stay behind.
Examples	Mainly for Action and Adventure games: <ul style="list-style-type: none"> • Remotely opening a door by holding down a switch (<i>Portal 2</i>) • Boosting another player onto a ledge (<i>Resident Evil 5</i>)
Consequences	Always obligatorily, separating players and at a specific location. Short time (not including the duration they stay separated), low communication and timing requirements. The time spent alone might increase tension and risk.
Using the pattern	Can use fixed or free roles, works for an arbitrary number of players.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Conditional Passageways, Functional Roles, Abilities That Can Only Be Used On Another Player; Subpatterns: Inaccessible Areas; Conflicting: Gathering Gate (only simultaneously, they can be used after each other)

Name	Gathering Gate
Description	Forcing the players to wait for each other by allowing them only to continue together.
Examples	Mainly for Action and Adventure games: <ul style="list-style-type: none"> • Camera checking the presence of all players (<i>Portal 2</i>) • Doors with two switches (<i>Resident Evil 5</i>)
Consequences	Always obligatorily, collecting players and at a specific location. Short time (not including waiting times), low communication and timing requirements. Might cause faster players to get annoyed while waiting.
Using the pattern	Can use fixed or free roles, works for an arbitrary number of players.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Conditional Passageways, Functional Roles, Player-Player Proximity, Interacting With The Same Object; Subpatterns: Inaccessible Areas; Conflicting: Separation Gate (only simultaneously, they can be used after each other)

C.3 INFORMATION EXCHANGE

Name	Perspectives
Description	Giving the players different information (different goggles, viewing something from different sides simultaneously, extending view range).
Examples	Genre independent: <ul style="list-style-type: none"> • “Fog of war” with shared lines of sight (<i>Age of Empires II: The Age of Kings</i>)
Consequences	Usually requires high communication (exception: requires low communication in splitscreen-modes or with shared lines of sight).
Using the pattern	Can be implemented collecting or separating (depending on what must be observed simultaneously), specific or pervasive location, any interaction time (depends on the amount of information to be shared and whether it updates continuously), voluntary or mandatory, fixed (limited availability of view) or free roles, works for an arbitrary number of players. Timing requirements low, medium or high (depending on the time span between the information becoming available and the need to act upon it).
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Functional Roles, Information Passing; Subpattern: Fog of War, Reconnaissance; Conflicting: Split-Screen Views (removes need for communication)

Name	Guide
Description	One player acting directly on another player's information.
Examples	Mainly Action games: <ul style="list-style-type: none"> • Player with a torch providing sight for another using a tool (<i>Minecraft</i>)
Consequences	Longer interaction, high communication and high to medium timing requirements.
Using the pattern	Can be used for collecting or separating (remote observation), specific (using monitors) or pervasive location, voluntary or mandatory (depending on risk of acting blind), fixed or free roles, works for an arbitrary number of players.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Functional Roles, Altruistic Actions, Perspective; Conflicting: Split-Screen Views (removes need for communication)

C.4 PLAYER SUPPORT

Name	Strengthening
Description	Adding or increasing a positive effect on other players.
Examples	Mainly for Action and Role-Playing games: <ul style="list-style-type: none"> • Bonus damage or speed “buffs” (<i>World of Warcraft, League of Legends</i>) • Kevlar vests (<i>Brink</i>)
Consequences	Always collecting, voluntary (though importance can be increased through higher difficulty), pervasive location, low communication and timing (because it constitutes a bonus). Usually for two players only, although one could construct an interaction for which multiple players need to work together in order to strengthen another one.
Using the pattern	Can be implemented with fixed or free roles, can be combined with Sacrifice and Team-based Rewards. Interaction time is typically short, although in some cases the interaction can be maintained longer in order to maintain or increase its effect (stacking), resulting in a longer interaction time.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Altruistic Actions, Player-Player Proximity, Functional Roles, Team Combos, Complementarity, Abilities That Can Only Be Used On Another Player; Subpatterns: Invulnerabilities

Name	Resupply
Description	Restoring a positive capability for another player without getting something in return. The capability is then consumed by the player, decreased by enemies or evaporates over time.
Examples	Mainly for Action and Role-Playing games: <ul style="list-style-type: none"> • Healing others (<i>World of Warcraft</i>) • Supplying ammunition (<i>Battlefield 4</i>) • Paying tributes (<i>Age of Empires II: The Age of Kings</i>)
Consequences	Always collecting, voluntary (though importance can be increased through higher difficulty), pervasive location, medium communication and timing. Usually for two players only, although one could construct an interaction for which multiple players need to work together in order to resupply another one.
Using the pattern	Can be implemented with fixed or free roles, can be combined with Sacrifice and Team-based Rewards. Interaction time is typically short, although in some cases the interaction can be maintained longer in order to restore more of the capability, resulting in a longer interaction time.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Altruistic Actions, Player-Player Proximity, Functional Roles, Complementarity, Abilities That Can Only Be Used On Another Player

Name	Protector
Description	Preventing a negative effect on other players.
Examples	Mainly for Action and Role-Playing games: <ul style="list-style-type: none"> • Protecting players carrying objects (<i>Left 4 Dead 2</i>) • Conjuring shield effects (<i>League of Legends</i>)
Consequences	Always collecting, voluntary (though importance can be increased through higher difficulty), pervasive location, low communication (must be anticipated by protecting player) and high timing (usually immediately before negative effect). Usually for two players only, although one could construct an interaction for which multiple players need to work together in order to protect another one.
Using the pattern	Can be implemented with fixed or free roles, can be combined with Sacrifice and Team-based Rewards. Interaction time is typically short, although in some cases the interaction can be maintained longer in order to maintain or increase its effect (stacking), resulting in a longer interaction time.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Altruistic Actions, Player-Player Proximity, Functional Roles, Complementarity, Abilities That Can Only Be Used On Another Player; Subpattern: Ability Losses

Name	Savior
Description	Removing an undesired effect from another player.
Examples	Mainly for Action and Role-Playing games: <ul style="list-style-type: none"> • Pulling others up from a ledge (<i>Left 4 Dead</i>) • Reviving players (<i>Battlefield 4</i>) • Removing sticky bombs (<i>Brink</i>)
Consequences	Always collecting, voluntary (though importance can be increased through higher difficulty), pervasive location, medium communication (call for help only) and high timing (usually proceeds a more severe punishment when not removed after a short time). Usually for two players only, although one could construct an interaction for which multiple players need to work together in order to save another one.
Using the pattern	Can be implemented with fixed or free roles, can be combined with Sacrifice and Team-based Rewards. Interaction time is typically short, although in some cases the interaction can be maintained longer in order to remove multiple (stacked) effects, resulting in a longer interaction time.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Altruistic Actions, Player-Player Proximity, Functional Roles, Helplessness, Complementarity, Abilities That Can Only Be Used On Another Player; Subpattern: Ability Losses, Rescue

C.5 NPCS

Name	Distraction
Description	Capturing an enemy's attention and / or luring them away.
Examples	Mainly for Action and Role-Playing games: <ul style="list-style-type: none"> Players can get enemies to target them instead of others ("Aggro"-mechanic in <i>Army of Two</i> or <i>World of Warcraft</i>)
Consequences	Typically medium interaction time, low communication (visually indicated by the game) and medium timing requirements.
Using the pattern	Can be implemented collecting (all players keep interacting with the enemy) or separating (when the enemy is lured away), in specific or pervasive locations (depending on NPC location), mandatory or voluntary, with fixed or free roles, works for an arbitrary number of players.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Functional Roles; Sub-pattern: Evade, Flanking Routes

Name	Weakening
Description	Making enemies more vulnerable against other players' actions (debuff).
Examples	Mainly for Action and Role-Playing games: <ul style="list-style-type: none"> Weapon-effects ("Slag" in <i>Borderlands 2</i>) Marks placed on enemies that make the next strong attack more powerful (<i>Forced</i>)
Consequences	Always collecting, short interaction time, low communication (visually indicated by the game) and high timing requirements (typically lasts for a short time only).
Using the pattern	Can be implemented in specific or pervasive locations (depending on NPC location), mandatory or voluntary, with fixed or free roles, works for an arbitrary number of players.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Functional Roles, Synergies Between Abilities

C.6 MOVEMENT

Name	Transport
Description	Influencing the movement of another player.
Examples	Mainly Action games: <ul style="list-style-type: none"> • Driving transporters (<i>Battlefield 4</i>) • Manipulating conveyer-belt like elements ("Excursion Funnels" in <i>Portal 2</i>) • Placing portals to alter another player's trajectory (<i>Portal 2</i>)
Consequences	Typically lasts for a longer time and has low communication and timing requirements (exception: combination with the "Perspective" pattern in which the controlling player does not see the results of his actions).
Using the pattern	Can be implemented collecting or separating (depending on from where the movement is controlled), in (larger) specific or pervasive locations, voluntary or mandatory and with fixed or free roles. Works for an arbitrary number of players.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Functional Roles, Movement, Maneuvering

Name	Moving In Formation
Description	Moving while keeping a specific distance or formation (everyone in control of themselves).
Examples	Mainly Action games: <ul style="list-style-type: none"> • Moving behind a player holding a tactical shield (<i>Counter-Strike</i>)
Consequences	Always collecting, medium time and medium to high communication and timing requirements (depending on how large the margin for deviations is).
Using the pattern	Can be implemented in (larger) specific or pervasive locations, voluntary or mandatory and with fixed or free roles. Works for an arbitrary number of players.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Functional Roles, Player-Player Proximity, Maneuvering

Name	Physical Presence
Description	Using the physical properties of a player characters body.
Examples	Mainly Action games: <ul style="list-style-type: none"> • Using another player as a platform for jumping higher (<i>ibb & obb</i>) • Weighting down a button (<i>Portal 2</i>)
Consequences	Medium time and variable communication and timing requirements.
Using the pattern	Can be used to collect (boosting) or separate (weighting down remote switches) players, in specific or pervasive locations, voluntary or mandatory and with fixed or free roles. Works for an arbitrary number of players.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Functional Roles, Player Characters; Subpatterns: Player-Player Proximity, Player-Location Proximity

C.7 RESOURCES

Name	Trade
Description	Giving (dispensable) resources to another player, receiving something in return.
Examples	Mainly Strategy and Simulation games: <ul style="list-style-type: none"> • Trading equipment items dropped by enemies (<i>Borderlands 2</i>)
Consequences	Always collecting, in pervasive locations and with free roles. Short interaction of two players with medium communication and low timing requirements. Can be used by the players to (voluntarily) bridge skill differences by giving to weaker players.
Using the pattern	Can be implemented as mandatory (when players are unable to gather all resources) or voluntary (to enable specialization).
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Altruistic Actions, Functional Roles, Inventories, Limited Resources; Conflicting: Shared Resources (which is an easier way to provide the same function)

Name	Hot Potato
Description	Having an object which should be regularly passed between players to prevent negative or enable positive effects.
Examples	Mainly Action games: <ul style="list-style-type: none"> • Passing a ball which disables weapon usage ("Bombing Run"-Mode in <i>Unreal Tournament 2004</i>)
Consequences	Always collecting and with free roles. Takes a comparatively long time (a certain number of exchanges is needed in order to be interesting) and works for an arbitrary number of players. High communication and timing requirements, especially when the object is thrown. If the required frequency of exchanges is too high players might feel pushed.
Using the pattern	Can be implemented as voluntary (as a bonus) or mandatory (with severe negative effects) and on a specific (route) or pervasive location. Exchange might be reliable (the object is always with one player) or unreliable (it is thrown and might fall to the ground).
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Functional Roles, Limited Resources

C.8 COMPETITION

Name	Contest
Description	Players gaining individual scores with which they compete (on an insignificant level in relation to the goal) while working towards a common goal.
Examples	Genre independent: <ul style="list-style-type: none"> • Score-contests (<i>Far Cry 3</i>) • Rankings deciding the game's ending (<i>F.E.A.R. 3</i>) • Fighting over a kiss (<i>Castle Crashers</i>)
Consequences	Long interaction with low communication and timing requirements (in relation to the collaborative goal). Works for an arbitrary number of players, but may impact future teamwork negatively.
Using the pattern	Can be implemented in a specific or pervasive location, collecting or separating (chances for winning should be equal for each location), mandatory (but players may loose intentionally) or voluntary and with fixed (but designed for equal chances) or free roles.
Relations	Superior: Multiplayer Games

C.9 ADDITIONAL PATTERNS

C.9.1 *Modifiers*

Name	Dependency
Description	Harming the team with mistakes made by a single player.
Examples	Genre independent: <ul style="list-style-type: none"> • Overall balancing in MOBAs (<i>League of Legends</i>, <i>Dota 2</i>) • Sharing a civilization with all its units, resources and upgrades (<i>Age of Empires II: The Age of Kings</i>)
Consequences	Always pervasive and mandatory. May increase the bond between players, but can also cause frustration or bad feelings.
Using the pattern	Can be used with fixed or free roles, works for an arbitrary number of players. The degree of punishment can vary greatly.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Functional Roles

Name	Sacrifice
Description	Benefiting others while suffering some kind of penalty in the process.
Examples	Mainly for Action, Role-Playing and Strategy games: <ul style="list-style-type: none"> • One player must sacrifice himself to let the team advance ("The Sacrifice"-mission in <i>Left 4 Dead</i>) • Giving some resources to another player (<i>Minecraft</i>, <i>Age of Empires 2</i>) • Carrying an item that prevents the player from fighting back (<i>Dirty Bomb</i>) • Using skills on another player which are then unavailable during their cooldown duration (<i>World of Warcraft</i>)
Consequences	Modification of another support pattern (inherits most of its consequences), but requiring free roles and adding a trade-off between the interests of multiple players involved and requiring selflessness.
Using the pattern	Ratio of benefit-to-penalty dictates attractiveness of the sacrifice, often combined with Strengthening, Resupply, Protector and Savior to add a trade-off to supportive behavior.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Altruistic Actions, Social Dilemmas

Name	Team-based Rewards
Description	Benefiting others benefits the player him-/herself.
Examples	Mainly Action games: <ul style="list-style-type: none"> • Team-based actions yield more experience rewards (<i>Brink</i>)
Consequences	Modification of another support pattern (inherits most of its consequences), increasing the probability of its occurrence.
Using the pattern	Relevance of benefit impacts the probability of its occurrence further, often combined with Strengthening, Resupply, Protector or Savior to encourage supportive behavior.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Altruistic Actions

Name	Shared Resources
Description	Having a pool of objects which are accessible for all players simultaneously, sharing them implicitly and therefore removing the need for trading interactions.
Examples	Mainly Strategy, Simulation and Adventure games: <ul style="list-style-type: none"> • Guild bank (<i>World of Warcraft</i>) • Sharing a civilization and its resources (<i>Age of Empires 2</i>) • Respawn tickets (<i>Battlefield 4</i>) • Marks placed on enemies that make the next strong attack more powerful (<i>Forced</i>) • Energy orb that can be called to any player's position (<i>Forced</i>)
Consequences	Always pervasive. Low communication and timing requirements for accessing the resources, depending on their importance stronger requirements might arise (for example discussing the usage of rare objects). Works for an arbitrary number of players. May cause arguments about usage of sparse resources.
Using the pattern	Can be implemented with fixed (rights management) or free roles.
Relations	Superior: Multiplayer Games, Cooperation, Collaborative Actions, Symbiotic Player Relations, Team Accomplishment, Functional Roles; Conflicting: Trade (which is a harder way to provide the same function)

Name	Traitor
Description	Giving the players the ability to break their collaboration and pursue individual (competing) goals.
Examples	Genre independent: <ul style="list-style-type: none"> • Killing other players to take their loot (<i>Kane & Lynch 2: Dog Days</i>)
Consequences	Works for an arbitrary number of players, but introduces an atmosphere of mistrust and may impact future teamwork negatively. Interaction itself is short, but with long repercussions. No communication or timing requirement at all, since the action itself is not collaborative. Collaboration will not continue afterwards in most cases.
Using the pattern	Can happen at a specific or pervasive location and voluntary or mandatory (by allowing only a certain number of players to proceed) – although a pervasive and voluntary implementation is more unpredictable and therefore more interesting.
Relations	Superior: Multiplayer Games, Betrayal; Subpattern: Player Killing

C.9.2 Additions to existing patterns

Name	Privileged Abilities [18] / Complementarity [97]
Examples	Genre independent: <ul style="list-style-type: none"> • Actions for specific characters (<i>Lara Croft and the Guardian of Light</i>) • Actions bound to tools with limited availability (<i>Lego Indiana Jones: The Original Adventures</i>)
Consequences	Different player characters or roles are always implemented in a pervasive, location independent manner.
Using the pattern	Can be used with fixed or free (e.g. exchangeable items) roles and mandatory or voluntary (when specialization constitutes a bonus). Communication as well as timing requirements depend on the concrete application of the actions, works for an arbitrary number of players.

Name	Game Masters [18]
Examples	Genre independent: <ul style="list-style-type: none"> • Commanders organizing their team and building infrastructure (<i>Natural Selection 2</i>)
Consequences	Always pervasive, game experience heavily influenced by the game master's skills and knowledge.
Using the pattern	Can be used with fixed or free (switching allowed) roles and mandatory or voluntary (role can be taken by an AI or constitutes an optional bonus), works for an arbitrary number of players.

Name	Transferable Items [18]
Examples	Mainly Action and Adventure games: <ul style="list-style-type: none">• Tools enabling certain actions, limited to one per player (<i>Lego Indiana Jones: The Original Adventures</i>)
Consequences	Always collecting, in pervasive locations and with free roles. Short interaction of two players with medium communication and low timing requirements.
Using the pattern	Can be implemented mandatory (when multiple items must be used concurrently) or voluntary (to parallelize actions).

D

DETAILED BALANCING CALCULATIONS

D.1 WORST CASE STANDARD DEVIATION (EVEN NUMBER OF VALUES)

$$\begin{aligned} SD_{worst}^{even}(n) &= \sqrt{\frac{n \cdot \left(0 - \frac{\text{SUM}(M_{worst}^{even})}{2 \cdot n}\right)^2 + n \cdot \left(1 - \frac{\text{SUM}(M_{worst}^{even})}{2 \cdot n}\right)^2}{2 \cdot n}} \\ SD_{worst}^{even}(n) &= \sqrt{\frac{n \cdot \left(0 - \frac{n}{2 \cdot n}\right)^2 + n \cdot \left(1 - \frac{n}{2 \cdot n}\right)^2}{2 \cdot n}} \\ SD_{worst}^{even}(n) &= \sqrt{\frac{n \cdot \left(0 - \frac{1}{2}\right)^2 + n \cdot \left(1 - \frac{1}{2}\right)^2}{2 \cdot n}} \\ SD_{worst}^{even}(n) &= \sqrt{\frac{n \cdot \left(-\frac{1}{2}\right)^2 + n \cdot \left(\frac{1}{2}\right)^2}{2 \cdot n}} \\ SD_{worst}^{even}(n) &= \sqrt{\frac{n \cdot \left(\frac{1}{4} + \frac{1}{4}\right)}{2 \cdot n}} \\ SD_{worst}^{even}(n) &= \sqrt{\frac{\frac{1}{2}}{2}} = \sqrt{\frac{1}{4}} = \frac{1}{2} \end{aligned}$$

D.2 WORST CASE STANDARD DEVIATION (ODD NUMBER OF VALUES)

$$\begin{aligned} M_{\text{worst}}^{\text{odd}} = & \{m_0 = 0, \dots, m_{n-1} = 0, \\ & m_n = 0, \\ & m_{n+1} = 1, \dots, m_{2n} = 1\} \end{aligned}$$

$$\begin{aligned} SD_{\text{worst}}^{\text{odd}}(n) &= \sqrt{\frac{(n+1) \cdot \left(0 - \frac{\text{SUM}(M_{\text{worst}}^{\text{odd}})}{2 \cdot n + 1}\right)^2 + n \cdot \left(1 - \frac{\text{SUM}(M_{\text{worst}}^{\text{odd}})}{2 \cdot n + 1}\right)^2}{2 \cdot n + 1}} \\ &= \sqrt{\frac{(n+1) \cdot \left(0 - \frac{n}{2 \cdot n + 1}\right)^2 + n \cdot \left(1 - \frac{n}{2 \cdot n + 1}\right)^2}{2 \cdot n + 1}} \\ &= \sqrt{\frac{(n+1) \cdot \frac{n^2}{(2 \cdot n + 1)^2} + n \cdot \left(1 - \frac{2 \cdot n}{2 \cdot n + 1} + \frac{n^2}{(2 \cdot n + 1)^2}\right)}{2 \cdot n + 1}} \\ &= \sqrt{\frac{\left(n \cdot \frac{n^2}{(2 \cdot n + 1)^2}\right) + \left(1 \cdot \frac{n^2}{(2 \cdot n + 1)^2}\right) + n \cdot \left(1 - \frac{2 \cdot n}{2 \cdot n + 1} + \frac{n^2}{(2 \cdot n + 1)^2}\right)}{2 \cdot n + 1}} \\ &= \sqrt{\frac{n \cdot \left(\frac{n^2}{(2 \cdot n + 1)^2} + \frac{n}{(2 \cdot n + 1)^2} + 1 - \frac{2 \cdot n}{2 \cdot n + 1} + \frac{n^2}{(2 \cdot n + 1)^2}\right)}{2 \cdot n + 1}} \\ &= \sqrt{\frac{n \cdot \left(\frac{2 \cdot n^2 + n}{(2 \cdot n + 1)^2} + 1 - \frac{2 \cdot n}{2 \cdot n + 1}\right)}{2 \cdot n + 1}} \\ &= \sqrt{\frac{n \cdot \left(\frac{2 \cdot n^2 + n}{(2 \cdot n + 1)^2} + \frac{2 \cdot n + 1}{2 \cdot n + 1} - \frac{2 \cdot n}{2 \cdot n + 1}\right)}{2 \cdot n + 1}} \\ &= \sqrt{\frac{n \cdot \left(\frac{2 \cdot n^2 + n}{2 \cdot n + 1} + 2 \cdot n + 1 - 2 \cdot n\right)}{(2 \cdot n + 1)^2}} \\ &= \sqrt{\frac{n \cdot \left(\frac{n \cdot (2 \cdot n + 1)}{2 \cdot n + 1} + 1\right)}{(2 \cdot n + 1)^2}} \\ &= \sqrt{\frac{n \cdot (n+1)}{(2 \cdot n + 1)^2}} = \frac{\sqrt{n \cdot (n+1)}}{(2 \cdot n + 1)} \end{aligned}$$

$$\begin{aligned} M'_{\text{worst}}^{\text{odd}} = & \{m_0 = 0, \dots, m_{n-1} = 0, \\ & m_n = 1, \\ & m_{n+1} = 1, \dots, m_{2 \cdot n} = 1\} \end{aligned}$$

$$\begin{aligned} SD'_{\text{worst}}(n) &= \sqrt{\frac{n \cdot \left(0 - \frac{\text{SUM}(M'_{\text{worst}}^{\text{odd}})}{2 \cdot n + 1}\right)^2 + (n + 1) \cdot \left(1 - \frac{\text{SUM}(M'_{\text{worst}}^{\text{odd}})}{2 \cdot n + 1}\right)^2}{2 \cdot n + 1}} \\ &= \sqrt{\frac{n \cdot \left(0 - \frac{n+1}{2 \cdot n + 1}\right)^2 + (n + 1) \cdot \left(1 - \frac{n+1}{2 \cdot n + 1}\right)^2}{2 \cdot n + 1}} \\ &= \sqrt{\frac{n \cdot \frac{(n+1)^2}{(2 \cdot n + 1)^2} + (n + 1) \cdot \left(1 - \frac{2 \cdot (n+1)}{2 \cdot n + 1} + \frac{(n+1)^2}{(2 \cdot n + 1)^2}\right)}{2 \cdot n + 1}} \\ &= \sqrt{\frac{\frac{n \cdot (n+1)^2}{(2 \cdot n + 1)^2} + (n + 1) \cdot \left(1 - \frac{2 \cdot (n+1)}{2 \cdot n + 1} + \frac{(n+1)^2}{(2 \cdot n + 1)^2}\right)}{2 \cdot n + 1}} \\ &= \sqrt{\frac{(n + 1) \cdot \left(\frac{n \cdot (n+1)}{(2 \cdot n + 1)^2} + \left(1 - \frac{2 \cdot (n+1)}{2 \cdot n + 1} + \frac{(n+1)^2}{(2 \cdot n + 1)^2}\right)\right)}{2 \cdot n + 1}} \\ &= \sqrt{\frac{(n + 1) \cdot \left(\frac{n \cdot (n+1)}{2 \cdot n + 1} + \left(2 \cdot n + 1 - 2 \cdot (n + 1) + \frac{(n+1)^2}{2 \cdot n + 1}\right)\right)}{(2 \cdot n + 1)^2}} \\ &= \sqrt{\frac{(n + 1) \cdot \left(\frac{n \cdot (n+1)}{2 \cdot n + 1} + \left(2 \cdot n + 1 - 2 \cdot n - 2 + \frac{(n+1)^2}{2 \cdot n + 1}\right)\right)}{(2 \cdot n + 1)^2}} \\ &= \sqrt{\frac{(n + 1) \cdot \left(\frac{n \cdot (n+1)}{2 \cdot n + 1} + \left(\frac{(n+1)^2}{2 \cdot n + 1} - 1\right)\right)}{(2 \cdot n + 1)^2}} \\ &= \sqrt{\frac{(n + 1) \cdot \left(\frac{n \cdot (n+1)}{2 \cdot n + 1} + \frac{(n+1)^2}{2 \cdot n + 1} - \frac{2 \cdot n + 1}{2 \cdot n + 1}\right)}{(2 \cdot n + 1)^2}} \\ &= \sqrt{\frac{(n + 1) \cdot \left(\frac{(n \cdot (n+1)) + (n+1)^2 - (2 \cdot n + 1)}{2 \cdot n + 1}\right)}{(2 \cdot n + 1)^2}} \\ &= \sqrt{\frac{(n + 1) \cdot \left(\frac{(n^2+n) + (n^2+2 \cdot n+1) - (2 \cdot n + 1)}{2 \cdot n + 1}\right)}{(2 \cdot n + 1)^2}} \\ &= \sqrt{\frac{(n + 1) \cdot \left(\frac{n^2+n+n^2+2 \cdot n+1-2 \cdot n-1}{2 \cdot n + 1}\right)}{(2 \cdot n + 1)^2}} \\ &= \sqrt{\frac{(n + 1) \cdot \left(\frac{2 \cdot n^2+n}{2 \cdot n + 1}\right)}{(2 \cdot n + 1)^2}} = \sqrt{\frac{(n + 1) \cdot \left(\frac{n \cdot (2 \cdot n + 1)}{2 \cdot n + 1}\right)}{(2 \cdot n + 1)^2}} \\ &= \sqrt{\frac{(n + 1) \cdot n}{(2 \cdot n + 1)^2}} = \frac{\sqrt{(n + 1) \cdot n}}{(2 \cdot n + 1)} = SD'_{\text{worst}}(n) \end{aligned}$$

D.3 WORST POSSIBLE STANDARD DEVIATION USING PERCENTAGES

$$\begin{aligned}
 SD_{worst\%}(M) &= \sqrt{\frac{\left(\text{SUM}(M) - \frac{\text{SUM}(M)}{n}\right)^2 + (n-1) \cdot \left(0 - \frac{\text{SUM}(M)}{n}\right)^2}{n}} \\
 &= \sqrt{\frac{\left(\frac{n \cdot \text{SUM}(M)}{n} - \frac{\text{SUM}(M)}{n}\right)^2 + (n-1) \cdot \left(-\frac{\text{SUM}(M)}{n}\right)^2}{n}} \\
 &= \sqrt{\frac{\left(\frac{n \cdot \text{SUM}(M) - \text{SUM}(M)}{n}\right)^2 + (n-1) \cdot \left(\frac{(\text{SUM}(M))^2}{n^2}\right)}{n}} \\
 &= \sqrt{\frac{\left(\frac{\text{SUM}(M) \cdot (n-1)}{n}\right)^2 + \left(\frac{(n-1) \cdot (\text{SUM}(M))^2}{n^2}\right)}{n}} \\
 &= \sqrt{\frac{\left(\frac{(\text{SUM}(M))^2 \cdot (n-1)^2}{n^2}\right) + \left(\frac{(n-1) \cdot (\text{SUM}(M))^2}{n^2}\right)}{n}} \\
 &= \sqrt{\frac{\left(\frac{(\text{SUM}(M))^2 \cdot (n-1)^2 + (n-1) \cdot (\text{SUM}(M))^2}{n^2}\right)}{n}} \\
 &= \sqrt{\frac{\left(\frac{(\text{SUM}(M))^2 \cdot (n-1) \cdot ((n-1)+1)}{n^2}\right)}{n}} \\
 &= \sqrt{\frac{\left(\frac{(\text{SUM}(M))^2 \cdot (n-1) \cdot n}{n^2}\right)}{n}} \\
 &= \sqrt{\frac{\left(\frac{(\text{SUM}(M))^2 \cdot (n-1)}{n}\right)}{n}} \\
 &= \sqrt{\frac{\left((\text{SUM}(M))^2 \cdot (n-1)\right)}{n^2}} \\
 &= \frac{\left(\text{SUM}(M) \cdot \sqrt{(n-1)}\right)}{n} \\
 &= \frac{\sqrt{(n-1)} \cdot (\text{SUM}(M))}{n} \\
 &= \sqrt{(n-1)} \cdot \frac{\text{SUM}(M)}{n}
 \end{aligned}$$

GAME QUESTIONNAIRE

E

E.1 ENGLISH TRANSLATION

Player No.: _____

Date: _____

Time: _____

Please rate whether you agree with the following statements regarding the respective level. Rate on a scale from 1 to 5 (1 = disagree, 5 = agree).

Walk (1st run)

	disagree	1	2	3	4	5	agree
I had fun playing the level							
I contributed to solving the level							
The others contributed to solving the level							
I helped other players							
I got help from other players							
	too easy			right			too difficult
The difficulty of the level was							

Boulder (1st run)

	disagree	1	2	3	4	5	agree
I had fun playing the level							
I contributed to solving the level							
The others contributed to solving the level							
I helped other players							
I got help from other players							
	too easy			right			too difficult
The difficulty of the level was							

Mazes (1st run)

	disagree	1	2	3	4	5	agree
I had fun playing the level							
I contributed to solving the level							
The others contributed to solving the level							
I helped other players							
I got help from other players							
	too easy			right			too difficult
The difficulty of the level was							

Player No.: _____

Date: _____

Time: _____

Please rate whether you agree with the following statements regarding the respective level. Rate on a scale from 1 to 5 (1 = disagree, 5 = agree).

Cave (1st run)

	disagree	1	2	3	4	5	agree
I had fun playing the level	too easy	1	2	3	4	5	too difficult
I contributed to solving the level	1	2	3	4	5	5	
The others contributed to solving the level	1	2	3	4	5	5	
I helped other players	1	2	3	4	5	5	
I got help from other players	1	2	3	4	5	5	
The difficulty of the level was							
	1	2	3	4	5		

Separated (1st run)

	disagree	1	2	3	4	5	agree
I had fun playing the level	too easy	1	2	3	4	5	5
I contributed to solving the level	1	2	3	4	5	5	
The others contributed to solving the level	1	2	3	4	5	5	
I helped other players	1	2	3	4	5	5	
I got help from other players	1	2	3	4	5	5	
The difficulty of the level was							
	1	2	3	4	5		

Overall Rating (1st run)

	disagree	1	2	3	4	5	agree
I had fun playing the game	too easy	1	2	3	4	5	5
I was content with my teammates	1	2	3	4	5	5	
I would have preferred to play the game alone	1	2	3	4	5	5	
I wanted to be faster than the others	1	2	3	4	5	5	

Player No.: _____

Date: _____

Time: _____

Please rate whether you agree with the following statements regarding the respective level. Rate on a scale from 1 to 5 (1 = disagree, 5 = agree).

Walk (2nd run)

	disagree					agree
	1	2	3	4	5	
I had fun playing the level						
I contributed to solving the level	1	2	3	4	5	
The others contributed to solving the level	1	2	3	4	5	
I helped other players	1	2	3	4	5	
I got help from other players	1	2	3	4	5	
The difficulty of the level was	too easy		right	too difficult		
	1	2	3	4	5	

Boulder (2nd run)

	disagree					agree
	1	2	3	4	5	
I had fun playing the level						
I contributed to solving the level	1	2	3	4	5	
The others contributed to solving the level	1	2	3	4	5	
I helped other players	1	2	3	4	5	
I got help from other players	1	2	3	4	5	
The difficulty of the level was	too easy		right	too difficult		
	1	2	3	4	5	

Mazes (2nd run)

	disagree					agree
	1	2	3	4	5	
I had fun playing the level						
I contributed to solving the level	1	2	3	4	5	
The others contributed to solving the level	1	2	3	4	5	
I helped other players	1	2	3	4	5	
I got help from other players	1	2	3	4	5	
The difficulty of the level was	too easy		right	too difficult		
	1	2	3	4	5	

Player No.: _____

Date: _____

Time: _____

Please rate whether you agree with the following statements regarding the respective level. Rate on a scale from 1 to 5 (1 = disagree, 5 = agree).

Cave (2nd run)

	disagree	1	2	3	4	5	agree
I had fun playing the level	1	2	3	4	5		
I contributed to solving the level	1	2	3	4	5		
The others contributed to solving the level	1	2	3	4	5		
I helped other players	1	2	3	4	5		
I got help from other players	1	2	3	4	5		
	too easy	1	2	3	4	5	too difficult
The difficulty of the level was	1	2	3	4	5		

Separated (2nd run)

	disagree	1	2	3	4	5	agree
I had fun playing the level	1	2	3	4	5		
I contributed to solving the level	1	2	3	4	5		
The others contributed to solving the level	1	2	3	4	5		
I helped other players	1	2	3	4	5		
I got help from other players	1	2	3	4	5		
	too easy	1	2	3	4	5	too difficult
The difficulty of the level was	1	2	3	4	5		

Overall Rating (2nd run)

	disagree	1	2	3	4	5	agree
I had fun playing the game	1	2	3	4	5		
I was content with my teammates	1	2	3	4	5		
I would have preferred to play the game alone	1	2	3	4	5		
I wanted to be faster than the others	1	2	3	4	5		
	too easy	1	2	3	4	5	too difficult

Player No.: _____

Date: _____

Time: _____

Please rate whether you agree with the following statements regarding the respective level. Rate on a scale from 1 to 5 (1 = disagree, 5 = agree).

Personal Information

	disagree					agree
	1	2	3	4	5	
I have engaged in an activity with my teammates before						
I often play video games on my own	1	2	3	4	5	
I often play video games against others	1	2	3	4	5	
I often play video games together with others	1	2	3	4	5	
I like playing video games on my own	1	2	3	4	5	
I like playing video games against others	1	2	3	4	5	
I like playing video games together with others	1	2	3	4	5	
I often work together with others (at school, work, ...)	1	2	3	4	5	
I play team sports (soccer, ...)	1	2	3	4	5	
I often play board games in which the players have to work together	1	2	3	4	5	
Age						
Gender						

Thank you for participating!

E.2 ORIGINAL VERSION

Spieler-Nr.: _____

Datum: _____

Uhrzeit: _____

**Bitte beurteilen Sie in wie weit Sie die folgenden Aussagen für die jeweilige Aufgabe für zutreffend halten.
Urteilen Sie auf einer Skala von 1 bis 5 (mit 1 = stimmt nicht, 5 = stimmt).**

Ein kleiner Spaziergang (1. Durchlauf)

	stimmt nicht	stimmt			
	1	2	3	4	5
Das Level hat mir Spaß gemacht					
Ich habe zur Lösung des Levels beigetragen					
Die anderen haben zur Lösung des Levels beigetragen					
Ich habe den anderen Spielern geholfen					
Die anderen Spieler haben mir geholfen					
Die Schwierigkeit des Levels war	zu leicht	gut	zu schwer		
	1	2	3	4	5

Ritt auf der Kugel (1. Durchlauf)

	stimmt nicht	stimmt			
	1	2	3	4	5
Das Level hat mir Spaß gemacht					
Ich habe zur Lösung des Levels beigetragen					
Die anderen haben zur Lösung des Levels beigetragen					
Ich habe den anderen Spielern geholfen					
Die anderen Spieler haben mir geholfen					
Die Schwierigkeit des Levels war	zu leicht	gut	zu schwer		
	1	2	3	4	5

Im Labyrinth (1. Durchlauf)

	stimmt nicht	stimmt			
	1	2	3	4	5
Das Level hat mir Spaß gemacht					
Ich habe zur Lösung des Levels beigetragen					
Die anderen haben zur Lösung des Levels beigetragen					
Ich habe den anderen Spielern geholfen					
Die anderen Spieler haben mir geholfen					
Die Schwierigkeit des Levels war	zu leicht	gut	zu schwer		
	1	2	3	4	5

Spieler-Nr.: _____

Datum: _____

Uhrzeit: _____

**Bitte beurteilen Sie in wie weit Sie die folgenden Aussagen für die jeweilige Aufgabe für zutreffend halten.
Urteilen Sie auf einer Skala von 1 bis 5 (mit 1 = stimmt nicht, 5 = stimmt).**

Höhlenforscher (1. Durchlauf)

	stimmt nicht	stimmt			
	1	2	3	4	5
Das Level hat mir Spaß gemacht					
Ich habe zur Lösung des Levels beigetragen	1	2	3	4	5
Die anderen haben zur Lösung des Levels beigetragen	1	2	3	4	5
Ich habe den anderen Spielern geholfen	1	2	3	4	5
Die anderen Spieler haben mir geholfen	1	2	3	4	5
Die Schwierigkeit des Levels war	zu leicht	gut	zu schwer		
	1	2	3	4	5

Alleine (1. Durchlauf)

	stimmt nicht	stimmt			
	1	2	3	4	5
Das Level hat mir Spaß gemacht					
Ich habe zur Lösung des Levels beigetragen	1	2	3	4	5
Die anderen haben zur Lösung des Levels beigetragen	1	2	3	4	5
Ich habe den anderen Spielern geholfen	1	2	3	4	5
Die anderen Spieler haben mir geholfen	1	2	3	4	5
Die Schwierigkeit des Levels war	zu leicht	gut	zu schwer		
	1	2	3	4	5

Gesamtbewertung (1. Durchlauf)

	stimmt nicht	stimmt			
	1	2	3	4	5
Das Spiel hat mir insgesamt Spaß gemacht					
Ich war mit meinen Mitspielern zufrieden	1	2	3	4	5
Ich hätte das Spiel lieber alleine gespielt	1	2	3	4	5
Ich wollte schneller als die anderen sein	1	2	3	4	5

Spieler-Nr.: _____

Datum: _____

Uhrzeit: _____

**Bitte beurteilen Sie in wie weit Sie die folgenden Aussagen für die jeweilige Aufgabe für zutreffend halten.
Urteilen Sie auf einer Skala von 1 bis 5 (mit 1 = stimmt nicht, 5 = stimmt).**

Ein kleiner Spaziergang (2. Durchlauf)

	stimmt nicht					stimmt					
	1	2	3	4	5		1	2	3	4	5
Das Level hat mir Spaß gemacht											
Ich habe zur Lösung des Levels beigetragen											
Die anderen haben zur Lösung des Levels beigetragen											
Ich habe den anderen Spielern geholfen											
Die anderen Spieler haben mir geholfen											
Die Schwierigkeit des Levels war											
	zu leicht		gut		zu schwer		zu leicht		gut		zu schwer
	1	2	3	4	5		1	2	3	4	5

Ritt auf der Kugel (2. Durchlauf)

	stimmt nicht					stimmt					
	1	2	3	4	5		1	2	3	4	5
Das Level hat mir Spaß gemacht											
Ich habe zur Lösung des Levels beigetragen											
Die anderen haben zur Lösung des Levels beigetragen											
Ich habe den anderen Spielern geholfen											
Die anderen Spieler haben mir geholfen											
Die Schwierigkeit des Levels war											
	zu leicht		gut		zu schwer		zu leicht		gut		zu schwer
	1	2	3	4	5		1	2	3	4	5

Im Labyrinth (2. Durchlauf)

	stimmt nicht					stimmt					
	1	2	3	4	5		1	2	3	4	5
Das Level hat mir Spaß gemacht											
Ich habe zur Lösung des Levels beigetragen											
Die anderen haben zur Lösung des Levels beigetragen											
Ich habe den anderen Spielern geholfen											
Die anderen Spieler haben mir geholfen											
Die Schwierigkeit des Levels war											
	zu leicht		gut		zu schwer		zu leicht		gut		zu schwer
	1	2	3	4	5		1	2	3	4	5

Spieler-Nr.: _____

Datum: _____

Uhrzeit: _____

**Bitte beurteilen Sie in wie weit Sie die folgenden Aussagen für die jeweilige Aufgabe für zutreffend halten.
Urteilen Sie auf einer Skala von 1 bis 5 (mit 1 = stimmt nicht, 5 = stimmt).**

Höhlenforscher (2. Durchlauf)

	stimmt nicht	stimmt			
	1	2	3	4	5
Das Level hat mir Spaß gemacht					
Ich habe zur Lösung des Levels beigetragen	1	2	3	4	5
Die anderen haben zur Lösung des Levels beigetragen	1	2	3	4	5
Ich habe den anderen Spielern geholfen	1	2	3	4	5
Die anderen Spieler haben mir geholfen	1	2	3	4	5
Die Schwierigkeit des Levels war	zu leicht	gut	zu schwer		
	1	2	3	4	5

Alleine (2. Durchlauf)

	stimmt nicht	stimmt			
	1	2	3	4	5
Das Level hat mir Spaß gemacht					
Ich habe zur Lösung des Levels beigetragen	1	2	3	4	5
Die anderen haben zur Lösung des Levels beigetragen	1	2	3	4	5
Ich habe den anderen Spielern geholfen	1	2	3	4	5
Die anderen Spieler haben mir geholfen	1	2	3	4	5
Die Schwierigkeit des Levels war	zu leicht	gut	zu schwer		
	1	2	3	4	5

Gesamtbewertung (2. Durchlauf)

	stimmt nicht	stimmt			
	1	2	3	4	5
Das Spiel hat mir insgesamt Spaß gemacht					
Ich war mit meinen Mitspielern zufrieden	1	2	3	4	5
Ich hätte das Spiel lieber alleine gespielt	1	2	3	4	5
Ich wollte schneller als die anderen sein	1	2	3	4	5

Spieler-Nr.: _____

Datum: _____

Uhrzeit: _____

**Bitte beurteilen Sie in wie weit Sie die folgenden Aussagen für die jeweilige Aufgabe für zutreffend halten.
Urteilen Sie auf einer Skala von 1 bis 5 (mit 1 = stimmt nicht, 5 = stimmt).**

Persönliche Infos

	stimmt nicht	stimmt			
Ich habe mit meinen Mitspielern schon einmal etwas zusammen gemacht	1 2 3 4 5	1 2 3 4 5			
Ich spiele Videospiele oft alleine	1 2 3 4 5	1 2 3 4 5			
Ich spiele Videospiele oft gegen andere	1 2 3 4 5	1 2 3 4 5			
Ich spiele Videospiele oft mit anderen zusammen	1 2 3 4 5	1 2 3 4 5			
Ich spiele Videospiele gerne alleine	1 2 3 4 5	1 2 3 4 5			
Ich spiele Videospiele gerne gegen andere	1 2 3 4 5	1 2 3 4 5			
Ich spiele Videospiele gerne mit anderen zusammen	1 2 3 4 5	1 2 3 4 5			
Ich arbeite oft mit anderen zusammen (in der Schule, auf der Arbeit, ...)	1 2 3 4 5	1 2 3 4 5			
Ich mach oft Mannschaftssport (Fußball, ...)	1 2 3 4 5	1 2 3 4 5			
Ich spiele oft Brettspiele bei denen man zusammenarbeiten muss	1 2 3 4 5	1 2 3 4 5			
Alter					
Geschlecht					

Vielen Dank für Ihre Teilnahme!

F

FULL EVALUATION RESULTS

COLLAB.	INDEPENDENT			DIFFERENCE					
	MEAN	SD	N	MEAN	SD	95% CI	P	N	
COLLABORATIVE FIRST									
Overall	Yes (1)	4.16	0.75	44	0.36	0.84	[0.11; 0.62]	0.006	44
	No (2)	3.72	1.05	46					
Walk	Yes (1)	3.87	1.02	46	0.30	1.31	[-0.09; 0.70]	0.123	46
	No (2)	3.57	1.21	46					
Boulder	Yes (1)	3.98	0.80	46	0.33	1.35	[-0.08; 0.73]	0.109	46
	No (2)	3.65	1.27	46					
Mazes	Yes (1)	3.70	1.05	46	0.17	1.24	[-0.19; 0.54]	0.345	46
	No (2)	3.52	1.17	46					
Cave	Yes (1)	3.78	1.19	46	0.17	1.20	[-0.18; 0.53]	0.330	46
	No (2)	3.61	0.95	46					
Separated	Yes (1)	3.70	0.90	44	0.63	0.95	[0.34; 0.92]	<0.001	43
	No (2)	3.04	1.21	45					
NON-COLLABORATIVE FIRST									
Overall	Yes (2)	4.42	0.66	43	0.44	0.85	[0.18; 0.71]	0.002	43
	No (1)	3.98	0.89	43					
Walk	Yes (2)	4.14	0.86	43	-0.02	0.67	[-0.23; 0.18]	0.822	43
	No (1)	4.16	0.90	43					
Boulder	Yes (2)	4.33	0.92	43	0.65	1.02	[0.34; 0.97]	<0.001	43
	No (1)	3.67	0.94	43					
Mazes	Yes (2)	4.00	0.98	43	0.26	0.93	[-0.03; 0.54]	0.078	43
	No (1)	3.74	1.05	43					
Cave	Yes (2)	4.09	1.07	43	0.14	0.92	[-0.14; 0.42]	0.323	43
	No (1)	3.95	0.95	43					
Separated	Yes (2)	4.35	0.75	43	0.98	1.12	[0.63; 1.32]	<0.001	43
	No (1)	3.37	1.22	43					

Table 22: Questionnaire results for the “enjoyment” questions about each level and the overall pattern evaluation game. The *independent* rows show the individual results for each level (mean, standard deviation and sample size), the *difference* rows show the mean differences between the individual answers and the results of paired samples t-tests (standard deviation, 95% confidence interval, P value and sample size). If the P value is less than 0.05, the difference is considered to be significant, if it is less than 0.01 the difference is highly significant.

	COLLAB.	INDEPENDENT			DIFFERENCE				P	N
		MEAN	SD	N	MEAN	SD	95% CI			
COLLABORATIVE FIRST										
Walk	Yes (1)	2.36	0.72	44	0.09	0.91	[-0.19; 0.37]	0.511	44	
	No (2)	2.27	0.82	44						
Boulder	Yes (1)	3.11	0.58	44	0.43	0.79	[0.19; 0.67]	0.001	44	
	No (2)	2.67	0.77	45						
Mazes	Yes (1)	2.16	0.74	45	-0.07	0.78	[-0.30; 0.17]	0.570	45	
	No (2)	2.22	0.77	45						
Cave	Yes (1)	3.38	0.61	45	0.59	0.97	[0.30; 0.89]	<0.001	44	
	No (2)	2.75	0.69	44						
Separated	Yes (1)	2.52	0.67	42	0.10	0.84	[-0.17; 0.37]	0.457	40	
	No (2)	2.42	0.91	43						
NON-COLLABORATIVE FIRST										
Walk	Yes (2)	2.14	0.77	43	0.12	0.91	[-0.16; 0.40]	0.404	43	
	No (1)	2.02	0.83	43						
Boulder	Yes (2)	3.05	0.53	43	0.49	0.86	[0.23; 0.75]	0.001	43	
	No (1)	2.56	0.67	43						
Mazes	Yes (2)	2.28	0.67	43	0.28	0.80	[0.03; 0.52]	0.027	43	
	No (1)	2.00	0.79	43						
Cave	Yes (2)	2.79	0.64	43	-0.24	0.73	[-0.46; -0.01]	0.040	42	
	No (1)	3.02	0.41	42						
Separated	Yes (2)	2.56	0.73	43	0.35	1.15	[-0.01; 0.70]	0.054	43	
	No (1)	2.21	0.89	43						

Table 23: Questionnaire results for the “perceived difficulty” questions about each level of the pattern evaluation game. The *independent* rows show the individual results for each level (mean, standard deviation and sample size), the *difference* rows show the mean differences between the individual answers and the results of paired samples t-tests (standard deviation, 95% confidence interval, P value and sample size). If the P value is less than 0.05, the difference is considered to be significant, if it is less than 0.01 the difference is highly significant.

	COLLAB.	INDEPENDENT			DIFFERENCE				
		MEAN	SD	N	MEAN	SD	95% CI	P	N
COLLABORATIVE FIRST									
Walk	Yes (1)	176.7	44.9	16	97.9	35.8	[78.8; 117.0]	<0.001	16
	No (2)	78.8	18.8	16					
Boulder	Yes (1)	254.7	55.5	16	103.6	52.4	[75.7; 131.5]	<0.001	16
	No (2)	151.1	6.8	16					
Mazes	Yes (1)	128.2	34.7	16	46.5	25.0	[33.2; 59.8]	<0.001	16
	No (2)	81.7	25.1	16					
Cave	Yes (1)	316.7	33.3	16	220.9	35.3	[202.1; 239.7]	<0.001	16
	No (2)	95.8	16.3	16					
Separated	Yes (1)	132.4	46.5	16	83.9	46.3	[59.3; 108.6]	<0.001	16
	No (2)	48.4	6.4	16					
NON-COLLABORATIVE FIRST									
Walk	Yes (2)	129.4	64.0	15	17.0	49.9	[-10.7; 44.6]	0.209	15
	No (1)	112.4	37.8	15					
Boulder	Yes (2)	238.7	64.7	15	73.3	51.8	[44.7; 102.0]	<0.001	15
	No (1)	165.3	23.2	15					
Mazes	Yes (2)	100.2	32.4	15	20.0	28.0	[4.5; 35.5]	0.015	15
	No (1)	80.2	15.9	15					
Cave	Yes (2)	190.8	54.1	15	32.6	56.4	[1.4; 63.8]	0.042	15
	No (1)	158.2	27.5	15					
Separated	Yes (2)	99.7	37.2	15	47.7	38.3	[26.5; 68.9]	<0.001	15
	No (1)	52.0	12.7	15					

Table 24: Logging results for the time the players took to solve each level of the pattern evaluation game. The *independent* rows show the individual results for each level (mean, standard deviation and sample size), the *difference* rows show the mean differences between the individual answers and the results of paired samples t-tests (standard deviation, 95% confidence interval, P value and sample size). If the P value is less than 0.05, the difference is considered to be significant, if it is less than 0.01 the difference is highly significant.

	COLLAB.	INDEPENDENT			DIFFERENCE				
		MEAN	SD	N	MEAN	SD	95% CI	P	N
COLLABORATIVE FIRST									
Walk	Yes (1)	3.60	2.23	15	1.27	1.75	[0.30; 2.24]	0.014	15
	No (2)	2.33	2.02	15					
Boulder	Yes (1)	6.60	4.37	15	3.73	3.41	[1.84; 5.62]	0.001	15
	No (2)	2.87	3.34	15					
Mazes	Yes (1)	3.00	2.14	15	0.60	1.35	[-0.15; 1.35]	0.108	15
	No (2)	2.40	2.03	15					
Cave	Yes (1)	7.53	3.78	15	5.47	3.16	[3.72; 7.22]	<0.001	15
	No (2)	2.07	2.05	15					
Separated	Yes (1)	2.87	1.92	15	1.53	1.81	[0.53; 2.53]	0.005	15
	No (2)	1.33	1.29	15					
NON-COLLABORATIVE FIRST									
Walk	Yes (2)	3.13	1.96	15	0.47	2.07	[-0.68; 1.61]	0.396	15
	No (1)	2.67	2.06	15					
Boulder	Yes (2)	8.80	4.90	15	5.40	5.19	[2.52; 8.28]	0.001	15
	No (1)	3.40	2.64	15					
Mazes	Yes (2)	3.33	1.63	15	0.87	1.60	[-0.02; 1.75]	0.054	15
	No (1)	2.47	1.55	15					
Cave	Yes (2)	5.87	3.64	15	1.73	3.56	[-0.24; 3.70]	0.080	15
	No (1)	4.13	2.17	15					
Separated	Yes (2)	3.20	1.47	15	1.93	1.67	[1.01; 2.86]	0.001	15
	No (1)	1.27	0.88	15					

Table 25: Protocol results for the amount of “on-topic” communication in each level of the pattern evaluation game. The *independent* rows show the individual results for each level (mean, standard deviation and sample size), the *difference* rows show the mean differences between the individual answers and the results of paired samples t-tests (standard deviation, 95% confidence interval, P value and sample size). If the P value is less than 0.05, the difference is considered to be significant, if it is less than 0.01 the difference is highly significant.

LEN.	“LOCKED DOOR” TASK					“PUZZLE” TASK				
	PL.	TR.	ARCS	EXP.	VER.	PL.	TR.	ARCS	EXP.	VER.
2	7	8	36	0:00	0:03	13	62	494	0:00	0:04
3	9	11	49	0:00	0:02	18	92	736	0:00	1:06
4	11	14	62	0:00	0:03	23	122	978	0:00	-
5	13	17	75	0:00	0:02	28	152	1220	0:00	-
6	15	20	88	0:00	0:03	33	182	1462	0:00	-
7	17	23	101	0:00	0:03	38	212	1704	0:00	-
8	19	26	114	0:00	0:03	43	242	1946	0:00	-
9	21	29	127	0:00	0:03	48	272	2188	0:00	-
10	23	32	140	0:00	0:03	53	302	2430	0:00	-
11	25	35	153	0:00	0:03	58	332	2672	0:00	-
12	27	38	166	0:00	0:03	63	362	2914	0:00	-
13	29	41	179	0:00	0:03	68	392	3156	0:00	-
14	31	44	192	0:00	0:04	73	422	3398	0:00	-
15	33	47	205	0:00	0:04	78	452	3640	0:00	-
16	35	50	218	0:00	0:04	83	482	3882	0:00	-
17	37	53	231	0:00	0:04	88	512	4124	0:00	-
18	39	56	244	0:00	0:05	93	542	4366	0:00	-
19	41	59	257	0:00	0:05	98	572	4608	0:00	-
20	43	62	270	0:00	0:06	103	602	4850	0:00	-
21	45	65	283	0:00	0:05	108	632	5092	0:00	-
22	47	68	296	0:00	0:06	113	662	5334	0:00	-
23	49	71	309	0:00	0:06	118	692	5576	0:00	-
24	51	74	322	0:00	0:06	123	722	5818	0:00	-
25	53	77	335	0:00	0:07	128	752	6060	0:00	-
26	55	80	348	0:00	0:07	133	782	6302	0:00	-
27	57	83	361	0:00	0:08	138	812	6544	0:00	-
28	59	86	374	0:00	0:08	143	842	6786	0:00	-
29	61	89	387	0:00	0:09	148	872	7028	0:00	-
30	63	92	400	0:00	0:10	153	902	7270	0:00	-
31	65	95	413	0:00	0:11	158	932	7512	0:00	-
32	67	98	426	0:00	0:20	163	962	7754	0:00	-

Table 26: Verification results for the generated examples (length scaling). The rows display the *game length*, the petri net size (*places*, *transitions*, and *arcs*), the *export time* and the *verification time*.

LEN.	PLAYERS	PLACES	TRANSITIONS	ARCS	EXP.	VER.
2	2	7	8	36	0:00	0:03
	3	8	9	43	0:00	0:02
	4	9	10	50	0:00	0:03
4	2	11	14	62	0:00	0:02
	3	12	15	69	0:00	0:03
	4	13	16	76	0:00	0:06
6	2	15	20	88	0:00	0:02
	3	16	21	95	0:00	0:03
	4	17	22	102	0:00	0:33
8	2	19	26	114	0:00	0:03
	3	20	27	121	0:00	0:04
	4	21	28	128	0:00	—
16	2	35	50	218	0:00	0:04
	3	36	51	225	0:00	0:43
	4	37	52	232	0:00	—
32	2	67	98	426	0:00	0:20
	3	68	99	433	0:00	—
	4	69	100	440	0:00	—

Table 27: Verification results for the generated examples (player scaling). The rows display the *game length*, the *player count*, the petri net size (*places*, *transitions*, and *arcs*), the *export time* and the *verification time*.

"LOCKED DOOR" TASK							NO TASK				
LEN.	BR.	PL.	TR.	ARCS	EXP.	VER.	PL.	TR.	ARCS	EXP.	VER.
2	2/1	11	14	62	0:00	0:03	7	10	34	0:00	0:02
	2/2	15	20	88	0:00	0:02	9	14	46	0:00	0:02
	3/1	15	20	88	0:00	0:03	9	14	46	0:00	0:03
	3/2	23	32	140	0:00	0:11	13	22	70	0:00	0:03
	4/1	19	26	114	0:00	0:06	11	18	58	0:00	0:03
	4/2	31	44	192	0:00	-	17	30	94	0:00	0:03
4	2/1	19	26	114	0:00	0:04	11	18	58	0:00	0:03
	2/2	27	38	166	0:00	0:24	15	26	82	0:00	0:02
	3/1	27	38	166	0:00	-	15	26	82	0:00	0:03
	3/2	43	62	270	0:00	-	23	42	130	0:00	0:04
	4/1	35	50	218	0:00	-	19	34	106	0:00	0:03
	4/2	59	86	374	0:00	-	31	58	178	0:00	0:04
6	2/1	27	38	166	0:00	0:26	15	26	82	0:00	0:03
	2/2	39	56	244	0:00	-	21	38	118	0:00	0:03
	3/1	39	56	244	0:00	-	21	38	118	0:00	0:03
	3/2	63	92	400	0:00	-	33	62	190	0:00	0:04
	4/1	51	74	322	0:00	-	27	50	154	0:00	0:04
	4/2	87	128	556	0:00	-	45	86	262	0:00	0:06
8	2/1	35	50	218	0:00	-	19	34	106	0:00	0:04
	2/2	51	74	322	0:00	-	27	50	154	0:00	0:04
	3/1	51	74	322	0:00	-	27	50	154	0:00	0:04
	3/2	83	122	530	0:00	-	43	82	250	0:00	0:06
	4/1	67	98	426	0:00	-	35	66	202	0:00	0:04
	4/2	115	170	738	0:00	-	59	114	346	0:00	0:08
16	2/1	67	98	426	0:00	-	35	66	202	0:00	0:04
	2/2	99	146	634	0:00	-	51	98	298	0:00	0:06
	3/1	99	146	634	0:00	-	51	98	298	0:00	0:07
	3/2	163	242	1050	0:00	-	83	162	490	0:00	0:15
	4/1	131	194	842	0:00	-	67	130	394	0:00	0:09
	4/2	227	338	1466	0:00	-	115	226	682	0:00	0:33
32	2/1	131	194	842	0:00	-	67	130	394	0:00	0:09
	2/2	195	290	1258	0:00	-	99	194	586	0:00	0:20
	3/1	195	290	1258	0:00	-	99	194	586	0:00	0:20
	3/2	323	482	2090	0:00	-	163	322	970	0:00	1:21
	4/1	259	386	1674	0:00	-	131	258	778	0:00	0:38
	4/2	451	674	2922	0:00	-	227	450	1354	0:00	-

Table 28: Verification results for the generated examples (optional rooms). The rows display the *game length*, the branching configuration (*branching factor / branch length*), the petri net size (*places, transitions, and arcs*), the *export time* and the *verification time*.

G

SUPERVISED STUDENT THESES

G.1 DIPLOMA AND MASTER THESES

- KOM-M-0539 Cristian Greciano. *Dynamic Difficulty Adaptation for Heterogeneously Skilled Player Groups in Collaborative Multiplayer Games*. Master Thesis. TU Darmstadt, Germany, May 2016.
- KOM-M-0522 Thomas Tregel. *Procedural Generation of Multiplayer Games Based on Game Design Patterns for Collaborative Player Interactions*. Master Thesis. TU Darmstadt, Germany, October 2015.
- KOM-D-0477 Benedetto de Lauso. *Design and Development of Concepts for Authoring an ERP-Business Game with Flexible Roles*. Diploma Thesis. TU Darmstadt, Germany, May 2013.
- KOM-D-0455 Shen Chenbin. *Applicationoriented Fusion and Aggregation of Sensordata*. Master Thesis. TU Darmstadt, Germany, May 2012.

G.2 BACHELOR THESES

- KOM-B-0528 David Baran. *Design and Implementation of an Extensible AI Framework for Simulating Players and Enemies in an Action-Oriented Collaborative Learning Game*. Master Thesis. TU Darmstadt, Germany, November 2015.
- KOM-B-0525 Kathrin Schölei. *Interaction Templates for Collaborative Adventure Games*. Master Thesis. TU Darmstadt, Germany, October 2015.
- KOM-B-0499 Rebecca Schieren. *Design and Implementation of a Framework for Multiplayer Games with an Adaptive Number of Players*. Master Thesis. TU Darmstadt, Germany, October 2014.
- KOM-B-0498 Roman Uhlig. *Design and Development of a Collaborative Multiplayer Game Using Design Patterns*. Master Thesis. TU Darmstadt, Germany, October 2014.
- KOM-B-0497 Fabian Groh. *Calculating Solutions for Collaborative Multiplayer Adventure Games*. Master Thesis. TU Darmstadt, Germany, September 2014.
- KOM-B-0496 Philipp Battenberg. *Detection of Deadlocks in Collaborative Multiplayer Game*. Master Thesis. TU Darmstadt, Germany, September 2014.
- KOM-B-0482 Stefan Hübecker. *Design and Implementation of Balancing-Mechanisms for Collaborative Multiplayer Adventure*. Master Thesis. TU Darmstadt, Germany, March 2014.
- KOM-B-0466 Thomas Tregel. *Design and Implementation of a Rapid Prototyping-Environment for Multiplayer Adventures*. Master Thesis. TU Darmstadt, Germany, May 2013.
- KOM-S-0450 Nikita Khakham. *Design and Implementation of Concepts for Collaborative Serious Games with Varying Group Sizes*. Bachelor Thesis. TU Darmstadt, Germany, November 2012.

AUTHOR'S PUBLICATIONS

H.1 MAIN PUBLICATIONS

1. Christian Reuter, Stefan Göbel, and Ralf Steinmetz. *Detecting Structural Errors in Scene-Based Multiplayer Games Using Automatically Generated Petri Nets*. In: *Foundations of Digital Games*. Pacific Grove, USA, 2015.
2. Christian Reuter, Stefan Göbel, and Ralf Steinmetz. *Approximating Balance in Collaborative Multiplayer Games*. In: Robin Munkvold and Line Kolas, editors, *Proceedings of the 9th European Conference on Games Based Learning*, pages 449–455. Academic Conferences Limited, 2015. ISBN 978-1-910810-58-3.
3. Christian Reuter, Stefan Göbel, and Ralf Steinmetz. *A Collection of Collaborative Player Interaction Patterns*. Technical Report KOM-TR-2014-01, KOM – Multimedia Communications Lab, 2014.
4. Christian Reuter, Thomas Tregel, Florian Mehm, Stefan Göbel, and Ralf Steinmetz. *Rapid Prototyping for Multiplayer Serious Games*. In: Carsten Busch, editor, *Proceedings of the 8th European Conference on Games Based Learning*, pages 478–486. Academic Conferences Limited, 2014. ISBN 978-1-910309-55-1.
5. Christian Reuter, Viktor Wendel, Stefan Göbel, and Ralf Steinmetz. *Game Design Patterns for Collaborative Player Interactions*. In: *Proceedings of DiGRA 2014: <Verb that ends in 'ing'> the <noun> of Game <plural noun>*. Salt Lake City, USA, 2014.
6. Christian Reuter. *Authoring Multiplayer Serious Games*. In: *8th International Conference on the Foundations of Digital Games*, pages 490–492. Chania, Greece, 2013.
7. Christian Reuter, Florian Mehm, Stefan Göbel, and Ralf Steinmetz. *Evaluation of Adaptive Serious Games Using Playtraces and Aggregated Play Data*. In: Carlos Vaz de Carvalho and Paula Escudeiro, editors, *Proceedings of the 7th European Conference on Games Based Learning*, pages 504–511. Academic Conferences Limited, Porto, Portugal, 2013. ISBN 978-1-909507-63-0.
8. Christian Reuter, Viktor Wendel, Stefan Göbel, and Ralf Steinmetz. *Towards Puzzle Templates for Multiplayer Adventures*. In: Stefan Göbel, Wolfgang Müller, Bodo Urban, and Josef Wiemeyer, editors, *3th International Conference on Serious Games for Training, Education and Health*, pages 161–163. Springer, Darmstadt, Germany, 2012.
9. Christian Reuter, Viktor Wendel, Stefan Göbel, and Ralf Steinmetz. *Multiplayer Adventures for Collaborative Learning With Serious Games*. In: Patrick Felicia, editor, *6th European Conference on Games Based Learning*, pages 416–423. Academic Conferences Limited, 2012.

10. Christian Reuter. *Development and Realization of Methods and Concepts for Multi-player Adventures*. Master's thesis, TU Darmstadt, 2011.

H.2 CO-AUTHORED PUBLICATIONS

11. Abdelkarim El Moussaoui, Christian Reuter, Maurice Wiegel, Sven Unkauf, Tanja Wiesmann, Mareike Dornhöfer, Madjid Fathi, and Sara Nasiri. *NeuroCare: Digitalisierte Früherkennung leichter kognitiver Einschränkungen*. In: *AAL-Kongress 2015*. VDE Verlag, 2015.
12. Stefan Göbel, Florian Mehm, Viktor Wendel, Johannes Konert, Sandro Hardy, Christian Reuter, Michael Gutjahr, and Tim Dutz. *Erstellung, Steuerung und Evaluation von Serious Games*. *Informatik Spektrum*, pages 547–557, 2014.
13. Sandro Hardy, Christian Reuter, Stefan Göbel, Ralf Steinmetz, Gisa Baller, Elke Kalbe, Abdelkarim El Moussaoui, Sven Abels, Susanne Dienst, Mareike Dornhöfer, and Madjid Fathi. *NeuroCare – Personalization and Adaptation of Digital Training Programs for Mild Cognitive Impairments*. In: *AAL-Kongress 2014, Tagungsband*. VDE Verlag, 2014.
14. Laila Shoukry, Christian Reuter, and Florian Mehm. *StoryTec and StoryPlay as Tools for Adaptive Game-Based Learning Research*. In: Stefan Göbel and Josef Wiemeyer, editors, *Games for Training, Education, Health and Sports*. Springer International Publishing, Darmstadt, Germany, 2014.
15. Felix Kamieth, Tim Dutz, Pia Weiss, Stefanie Müller, Christian Reuter, Reiner Wichert, Peter Klein, and Stefan Göbel. *Navigationsassistenz für ältere Menschen im öffentlichen Nahverkehr*. In: *Lebensqualität im Wandel von Demografie und Technik - 6. Deutscher AAL-Kongress*. VDE Verlag, Berlin, 2013.
16. Florian Mehm, Christian Reuter, and Stefan Göbel. *Authoring of Serious Games for Education*. In: Wolfgang Bösche and Klaus Bredl, editors, *Serious Games and Virtual Worlds in Education, Professional Development, and Healthcare*, *Serious Games and Virtual Worlds in Education, Professional Development, and Healthcare*. IGI Global, 2012.
17. Florian Mehm, Christian Reuter, Stefan Göbel, and Ralf Steinmetz. *Future Trends in Game Authoring Tools*. In: Marc Herrlich, Rainer Malaka, and Maic Masuch, editors, *Entertainment Computing - ICEC 2012*, LNCS 7522, pages 536–541. Springer, 2012. ISBN 978-3-642-33541-9.

I

CURRICULUM VITÆ

PERSONAL INFORMATION

Name	Christian Reuter
Date of Birth	February 6, 1987
Place of Birth	Bad Soden
Nationality	German

EDUCATION

11/2011–present	Technische Universität Darmstadt Department of Electrical Engineering and Information Technology Associate researcher at Multimedia Communications Lab (KOM)
10/2009–09/2011	Technische Universität Darmstadt Department of Computer Science Master of Science
10/2006–09/2009	Technische Universität Darmstadt Department of Computer Science Bachelor of Science
06/2006	Taunusgymnasium, Königstein Abitur

AWARDS AND HONORS

09/2012	Nominated for Best Paper Award at the FDG 2015 for: Christian Reuter, Stefan Göbel, and Ralf Steinmetz. <i>Detecting structural errors in scene-based Multiplayer Games using automatically generated Petri Nets</i> . In: Foundations of Digital Games, Pacific Grove, CA, USA, 2015.
---------	--

TEACHING ACTIVITIES

2013–present	Organizer of 6 Game Jams Gamedevelopment with 130 participants in total
2012–present	Supervisor of 13 Bachelor's and Master's theses (including "Diplomarbeiten")

- 2012–present Serious Games Lecture,
Assistant lecturer, exercise coordination and exam preparation
- 2012–present Serious Games Seminar,
Several topics assigned to 17 students
- 2012–present Serious Games Lab,
Several topics assigned to 17 student Groups

SCIENTIFIC ACTIVITIES

- Session Chair European Conference on Game-based Learning (ECGBL)
(2015)
- Workshop Chair Multiple Authoring-Workshops at GameDays, Darmstadt,
Germany
(2012–2015)
- Organization GameDays, Darmstadt, Germany
(2012–present)

J

ERKLÄRUNG LAUT §9 DER PROMOTIONSORDNUNG

I
CH versichere hiermit, dass ich die vorliegende Dissertation allein und nur unter Verwendung der angegebenen Literatur verfasst habe.

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 31. Mai 2016

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both `LATEX` and `LyX`:

<http://code.google.com/p/classicthesis/>

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

<http://postcards.miede.de/>