Erweiterung Web-basierter Prozesslernspiele zur Abbildung von allgemeinen Prozessabläufen

Bachelorarbeit

KIT – Karlsruher Institut für Technologie
Fraunhofer IOSB – Fraunhofer-Institut für Optronik, Systemtechnik und Bildauswertung

Simon Schwarz

1. März 2019

Verantwortliche Betreuer: Prof. Dr.-Ing. J. Beyerer
Prof. Dr. rer. nat. H. Steusloff
Betreuender Mitarbeiter: Dipl.-Inf. Alexander Streicher

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der gültigen Fassung beachtet habe.

Karlsruhe, den 1. März 2019

(Simon Schwarz)

Kurzfassung

Trotz der zentralen Rolle, die Prozesse in vielen Sektoren von Industrie und Regierung spielen, ist die Bereitschaft der beteiligten Akteure ihre Abläufe zu lernen gering. Hier können Serious Games wie das Prozeslernspiel Excercise Trainer (EXTRA) Abhilfe schaffen. In diesem ist es bisher allerdings nicht möglich Prozesse abzubilden, die optionale oder verpflichtende Pfade enthalten. Ebenso erfolgt die Erstellung entsprechender Level komplett manuell und ist somit fehleranfällig. Um diese Limitierungen von EXTRA zu eliminieren, entwickelt diese Arbeit ein theoretisches Modell, dass es ermöglicht, Prozesse unabhängig ihrer Modellierung als Graph zu spezifiziern. Diese generische (Graph-)Repräsentation erlaubt die Definition von Algorithmen für die Transformation eines Graphen in EXTRA-Level verschiedener Schwierigkeit sowie für die Verifikation eines solchen Levels. Darüber hinaus wird die Programmlogik von EXTRA erweitert und eine Web-Applikation implementiert, die es erlaubt, EXTRA-Level zu erstellen und zu modifizieren. Sie unterstützt zudem das (semi-)automatische Übersetzen eines in den Standards BPMN oder UML modellierten Prozesses in ein EXTRA-Level.

Abstract

Processes play a central role in many sectors of industry and government. Despite this fact, participants are usually not willing to learn about them. Serious Games like Exercise Trainer (EXTRA) can help to bridge this motivation gap. However, EXTRA's functioniality is currently limited: Processes containing optional or mandatory paths cannot be displayed. Moreover, all levels have to be created manually which can cause a multitude of errors. This thesis develops a theoretical model to represent processes as graphs, irrespective of their original format, to eliminate these limitations. This generic graph format allows for the definition of algorithms to transform such graphs into EXTRA levels (of varying difficulty) and to verify such instances. Additionally, the EXTRA game logic is extended to support these new features and a web application is developed which allows for the creation and modification of levels. This app is also able to directly translate a process (specified in BPMN or UML) into an EXTRA level.

Inhaltsverzeichnis

1 Einleitung

Unter Prozessen versteht man die formale oder informale Spezifikation von Abläufen um ein gegebenes Ziel zu erreichen [DDO08]. Sie sind von essenzieller Bedeutung in vielen Bereichen von Organisationen in der Industrie, im Bildungswesen, in Regierungen und dem Militär. Wie Arbeiten wie [San11] gezeigt haben, ist jedoch die Bereitschaft von Mitarbeitern sich mit den Prozessen auseinanderzusetzen sehr gering und die Vorteile potenzieller ausführlicher und detaillierter Planung gehen verloren. Hier können sog. *Serious Games* Abhilfe schaffen. Dabei handelt es sich um Spiele deren charakterisierendes Ziel nicht die Unterhaltung der Spieler ist. Sie verfolgen einen anderen Zweck. Dieser kann das Erlernen von Prozessabläufen sein, in solchen Fällen spricht man auch von Prozesslernspielen. Dieser Arbeit liegt ein solches Prozesslernspiel (Excercise Trainer - EXTRA) [Gun16] (Abbildung 1.1) zugrunde. In EXTRA muss der Spieler unter Zeitdruck Produkte herstellen und sie an Kunden verkaufen. Dies erreicht er durch das Platzieren und korrekte Verbinden von Gebäuden auf einem isometrischen Gatter.

1.1 Motivation

Bisher ist EXTRA jedoch limitiert: Es können nur Fabriken mit einem einzigen, unveränderlichen Eingang gebaut werden. Dies schränkt die Menge der abbildbaren Prozesse stark ein. Es können keine Prozesse mit optionalen oder verpflichtenden Pfaden abgebildet werden. Zudem geschieht die Levelerstellung manuell ohne jegliche Hilfestellungen mittels des Editieren von JSON-Level-Dateien. Dies führt zu Fehlern. Syntaktische bzw. logische Fehler fallen erst während des Spielens auf. Fehler in der Prozessabbildung fallen eventuell aber gar nicht auf: Es tritt nicht nur kein Lerneffekt auf, es wird sogar *falsch* gelernt. Bei der Levelerstellung soll nun ein Editor helfen, der die oben genannten Fehler erkennt, sie mittels eines grafischen Interfaces vermeidet und, soweit möglich, automatisch Level aus einem gegebenen Prozess erstellt.

1.2 Zielsetzung

Im Rahmen dieser Arbeit soll nun zunächst EXTRA so erweitert werden, dass allgemeine Prozesse abgebildet, als Level gespielt und somit schlussendlich erlernt werden können. Dies

impliziert insbesondere die Möglichkeit verpflichtende (und) sowie optionale (or) Pfade darzustellen. Erreicht wird dieses Ziel durch die Erweiterung der Spiellogik um entsprechende und- bzw. oder-Fabriken.

Das zweite Ziel der Arbeit ist die Konzeptionierung und Implementierung einer semiautomatischen Übersetzung eines gegeben Prozesses in ein EXTRA-Level. Dabei unterläuft der Prozess selbst einige Schritte: Zunächst muss er manuell in digitaler Weise spezifiziert werden, dazu steht eine Reihe von Sprachen zur Verfügung. Diese Arbeit betrachtet die zwei wichtigsten: Die Business Process Model Notation (BPMN) sowie die Aktivitätsdiagramme der Unified Modeling Language (UML2). Dabei handelt es sich um grafische Repräsentationen von Prozessen mit Knoten und Kanten. In den nächsten Schritten kommt das hier entwickelte Assistenzsystem zum Einsatz. Es gilt den spezifizierten Prozess in ein EXTRA-Level zu übersetzen. Dies erfordert den folgenden Ablauf: Zunächst wird der UML- oder BPMN-Prozess in eine hier entwickelte generische Graphrepräsentation übersetzt. Auf diesen Graphen können nun Übersetzungsalgorithmen spezifiziert werden, die die gegebenen Knoten und Kanten auf Level-Elemente abbilden. Von zentraler Bedeutung ist hier die Konsistenz mit der Metapher von EXTRA: Verschiedene Spielelemente haben (abhängig von ihrem) Typ verschiedene Anforderungen an ihre ein- und ausgehenden Kanten. Diese unterscheiden sich von ihren ursprünglichen Prozesselementen. Hierbei die Prozess-Semantik nicht zu verändern ist die größte Herausforderung bei dieser Übersetzung.

Da die Prozessspezifikation unabhängig vom Ziel (namentlich eine spielbare, „gamifizierte“ Variante des Prozesses als Level) geschieht, exisitiert eine Lücke zwischen der Prozessmodellierung und EXTRA. Der Prozess enthält die für das Level nötigen Meta-Informationen wie Zeitlimit, Gegner, Icons für Produkte etc. nicht. Darüber hinaus gibt es keine Garantien, dass selbst in der Ausgangsrepräsentation definierbare Informationen (wie Knoten- oder Kantenbeschriftungen) stets vollständig gegeben sind. Zu diesem Zweck werden in dieser Arbeit eine Reihe von Heuristiken für das automatische Generieren dieser Informationen beschrieben. Darüber hinaus wird auch ein Randomisierungssystem entwickelt, dass die zufällige Generierung von fehlenden Namen u.ä. ermöglicht. Weitere Untersuchungsaspekte betreffen die Verifikation von Levels bzw. einer aktuellen Levelinstanz: Bildet sie den unterliegenden Prozess korrekt ab? In diesem Zusammenhang kommt auch der Forschungsaspekt des Leveling zum Tragen. Es wird untersucht wie sich ein gegebener Prozess automatisch in mehrere EXTRA-Level unterschiedlichen Schwierigkeitsgrades unterteilen lässt.

1.3 Projektumgebung

Diese Arbeit ist am Fraunhofer Institut für Optronik, Systemtechnik und Bildauswertung (IOSB) entstanden. Die dort ansässige Abteilung Interoperabilität und Assistenzsysteme (IAS) untersucht und entwickelt Konzepte und Verfahren für intelligente Assistenzsysteme. Das Serious Game EXTRA, auf das diese Arbeit aufbaut, ist ein solches Assistenzsystem. Es dient dem *gamifzierten* Lernen von Prozessabläufen. Ursprünglich wurde es entwickelt um Operatoren bei multinationalen militärischen Manövern zu helfen, die Datenflüsse, die Interaktion diverser Computersysteme und die eigene Rolle zu verstehen.

Abbildung 1.1: Ein EXTRA-Level: Der Nutzer muss die Gebäude korrekt platzieren und verbinden. Quelle: [Gun16]

1.4 Gliederung

Zunächst wird in der zweiten Sektion der aktuellen Stand der Technik im Forschungsfeld des Digital Game Based Learning (DGBL) dargestellt. Der Oberbergriff der Serious Games wird hierbei in drei Kategorien unterteilt: Prozesslernspiele, Simulationen und Modellierungsumgebungen. Die dritte Sektion erläutert die für die folgenden Sektionen wichtigen Konzepte und Begriffe. Dazu zählen der Aufbau von EXTRA, die unterliegende Forschung von Serious Games, die Syntax und Semantik der betrachteten Prozessmodellierungsstandards sowie die

für die Implementierung verwendeten Web-Technologien und Bibliotheken. Die vierte Sektion detailliert die theoretischen Grundlagen der Arbeit. Hier wird die graphtheoretische Modellierung von Prozessen entwickelt sowie Algorithmen definiert, die auf diesem Modell arbeiten. Dazu zählen der Übersetzungs-, der Verifikations- und der Leveling-Algorithmus. Die fünfte Sektion beschreibt anschließend die Implementierung der erweiterten EXTRA-Spiellogik und des Level-Editors. Die folgende sechste Sektion beschreibt detailliert einen Anwendungsfall vom Hochladen der Datei bis zum Spielen des Levels. Die siebte und letzte Sektion fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf zukünftige Arbeiten.

2 Stand der Technik

Das Forschungsgebiet der Serious Games beschäftigt sich mit der Erhöhung der Lernmotivation durch Verwendung von Techniken und Erkenntnissen von Disziplinen wie Game Design, Linguistik, Psychologie oder Lerntheorie. Die nachfolgende Sektion stellt eine Reihe von in der Literatur vorgestellten Ansätzen im Gebiet des Digital Game Based Learning (DGBL) vor. Diese lassen sich grob in Spiele, Simulationen und Lernumgebungen unterteilen.

2.1 Lernspiele für Prozessabläufe

Process Modeling as a Serious Game Strecker und Rosenthal stellen ein Serious Game für das Prozesslernen in einer Fabrik vor [SR16]: Ein Industriepartner modelliert einen dortigen Ausschreibungsprozess als BPMN-Prozess, dieser soll der Belegschaft in Form eines Serious Game gelehrt werden. Dazu wird das vorhandene Prozessmodell auf den Kontrollfluss sowie Platzhalter für Aktivitäten, Ereignisse, Rollen, Dokumente und Informationssysteme reduziert. Bei dem hier entwickelten Serious Game handelt es sich um ein Rollenspiel in dem Gruppen, bestehend aus vier Mitgliedern unterschiedlicher Sektoren des Unternehmens, unter Zeitdruck versuchen die Platzhalter korrekt anzuordnen. Dieses Format hat mehrere Ziele: Die Teilnehmer sollen durch die Heterogenität der Gruppen nicht nur ein Verständnis für die eigene Funktion, sondern auch für die von anderen erwerben und Vorschläge einbringen. Des Weiteren ist der Anspruch des Spiels nicht den Prozess selbst zu konstruieren, sondern sich anhand von Beispiel-Dokumenten mit Aufbau und Terminologie desselben vertraut zu machen. Zu diesem Zweck wird im Rahmen des Spiels auch nicht der Originalprozess, sondern eine vereinfachte Darstellung (Abbildung 2.1) verwendet. [SR16]

Spielablauf Jeder Spieldurchlauf geht ungefähr 3-4 Stunden und unterteilt sich in drei Phasen. In der ersten Phase stellt ein Moderator die Regeln, die Notation und die zur Verfügung stehenden Werkzeuge vor. In der zweiten Phase füllen die Gruppen ausgehändigte Vorlagen aus. Ziel ist es eine bijektive Abbildung von Elementen auf Platzhalter herzustellen. Alle 10 Minuten wird (öffentlich) gezeigt wie viele Elemente (aber nicht welche) korrekt zugeordnet sind. Für Vorschläge an den Moderator wie der Prozess verbessert werden könnte gibt es einen

Preis: Die Validierung eines selbst gewählten Teil des Prozesses. Die dritte Phase besteht aus einem Multiple-Choice-Test, den jeder Teilnehmer alleine ausfüllt.

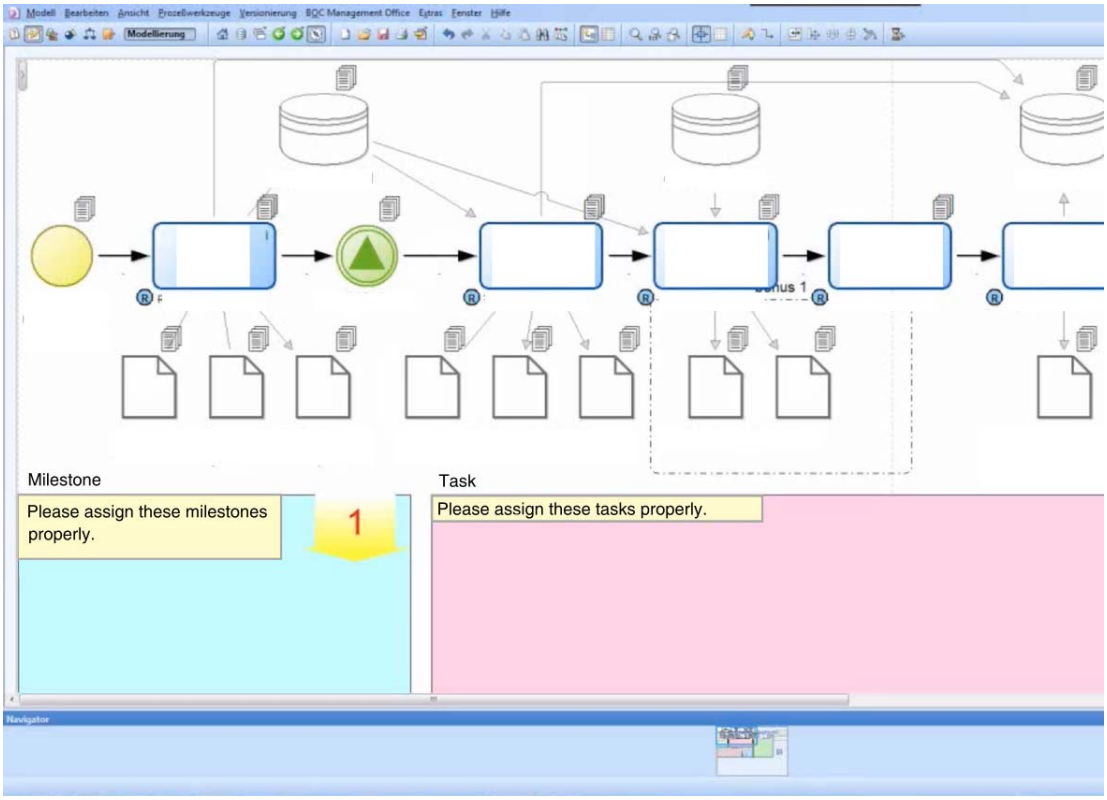


Abbildung 2.1: In diese vereinfachte Prozess-Darstellung müssen Element-Platzhalter korrekt zugeordnet werden. Quelle: [SR16]

DESIGMPS Chaves et al. stellen das Serious Game *DESIGMPS* vor. Ziel des Spiels ist es den Spieler mit Software Process Modelling (SPM) vertraut zu machen. SPM bezeichnet die Repräsentation eines Softwareprozesses, die verwendet werden kann um die Kommunikation zwischen Entwicklern und Managern zu erleichtern und das Nutzen von prozesstheoretischen Werkzeugen ermöglicht. Aufgrund seiner theoretischen Natur ist dieses Thema mit konventionellen Mitteln jedoch schwer zu lehren. Hier setzt *DESIGMP* an. Es handelt sich um ein Solo-Spiel, das den Nutzer mit Aufgaben konfrontiert, die durch das Erstellen von Softwareprozessmodellen gelöst werden müssen. Jede Aufgabe hat 4 Stufen, die unterschiedliche Schwierigkeitsstufen repräsentieren. Je höher die Stufe, desto weniger Hinweise präsentiert das Spiel. Die Aufgaben haben die folgende Form: A priori haben Experten eine Liste aus einem Referenzmodell zusammengestellt. Der Spieler muss nun Elemente und Beziehungen

aus dieser Liste korrekt in einer grafischen Repräsentation des Prozesses eintragen (Abbildung 2.2). [Cha+15]

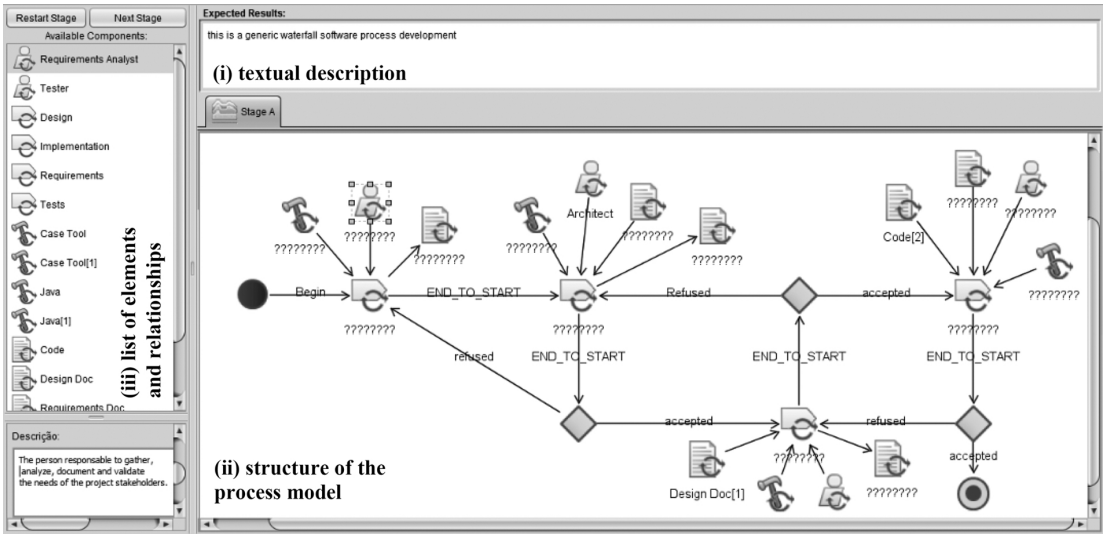


Abbildung 2.2: In diesem Skelett eines Software-Prozesses müssen Element-Platzhalter korrekt platziert werden. Quelle: [Cha+15]

2.2 Simulationen für Prozessabläufe

Using a Simulation Game Approach to Teach ERP Concepts Leger verfolgt einen *Learning by Doing*-Ansatz für das Erlernen von ERP (Enterprise Resource Planning) durch ein Simulationsspiel. Gruppen von 4-6 Mitgliedern führen jeweils eine von 5 Firmen (insgesamt also 20 - 30 Spieler pro Partie) und führen Supply Chain Management durch um Bestellungen zu erfüllen. Das Spielszenario ist das folgende: Die Firmen produzieren bis zu 6 Müsli-Produkte, die selbst wiederum aus bis zu 6 Zutaten bestehen. Alle Produkte werden vom Spiel ver- bzw. gekauft. Zum Führen der Firmen wird echte Software verwendet. (Abbildung 2.3) Das Spiel ist rundenbasiert und kompetitiv. [Leg07]

INNOV8 Für den Bereich des Business Process Managements (BPM) entwickelte IBM 2007 das Spiel *INNOV8*, 2009 erschien es in einer zweiten Version [TLB16]. Dabei handelt es sich um ein 3D-Spiel für das Modellieren und Optimieren von Business-Prozessen. Als Avatar trifft sich der Spieler mit den Stakeholders der Firma After Inc. In der Rolle eines Consultants muss er Informationen sammeln um die Prozesse eines Call-Centers zu modellieren, zu rekonfigurieren und zu optimieren. Dies beinhaltet u.a. die folgenden Aufgaben: Das Auswählen einer Strategie,

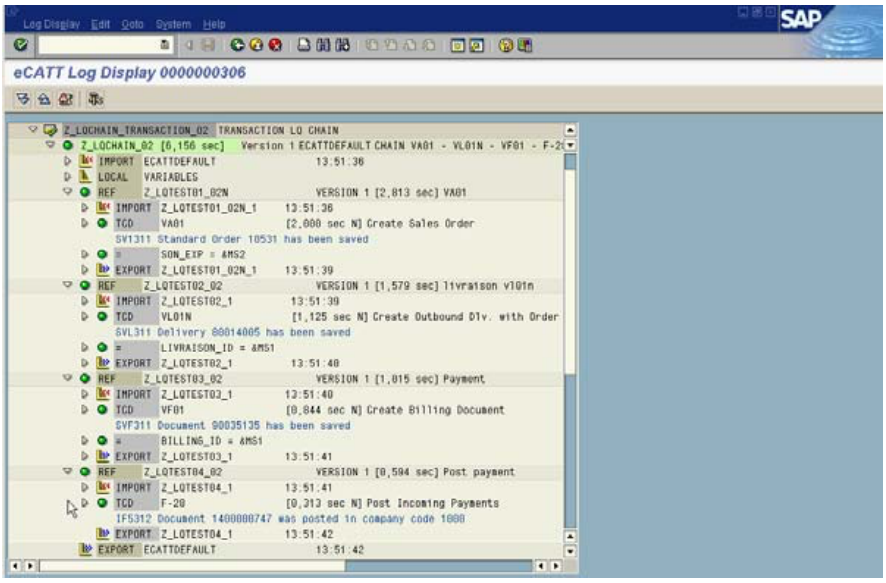


Abbildung 2.3: Im Rahmen der Simulation wird echte Software wie diese verwendet. Quelle: [Lego7]

das Zuweisen eines Budgets, das Erreichen der Ziele mittels der Prozesse und das Führen von Interviews. Am Ende des Spiels werden die gefundenen Lösungen mittels drei unterschiedlichen Simulationen getestet.

ImPROVE Ribeiro et al. stellen das Serious Game *ImPROVE* vor. Es ist darauf ausgelegt die Modellierung von Geschäftsprozessen im Manchester Triage System zu lehren. Es handelt sich um ein 3D-Spiel, dass die drei *triage healthcare values* modelliert und simuliert.

- Schütze Leben
- Effizientes Nutzen von Ressourcen
- Fairness

Das Spiel stellt BPMN-*swim lanes* dar, die Sub-Prozesse repräsentieren (Abbildung 2.5). Der Spieler muss nun Geschäftsprozesse simulieren (mittels Verzweigungen, Aktionen, Start und Ende). Nach Beendigung des Modellbaus wird der Prozess an einem Patienten getestet. Während der gesamten Simulation gibt das Spiel in Echtzeit Feedback. Dieses Feedbacksystem nutzt das Konzept *TDABC* (Time Driven Activity Based Costing). Dabei handelt es sich um ein System, Prozesskosten zu beschreiben: Jeder Prozess ist aus Aktivitäten aufgebaut. Die Kosten einer Aktivität sind das Produkt von zwei Faktoren (Kosten pro Einheit sowie Zeit). [Rib+12]

Abbildung 2.4: Innov8 erlaubt das Führen von Interviews über dieses GUI. Quelle: [TLB16]

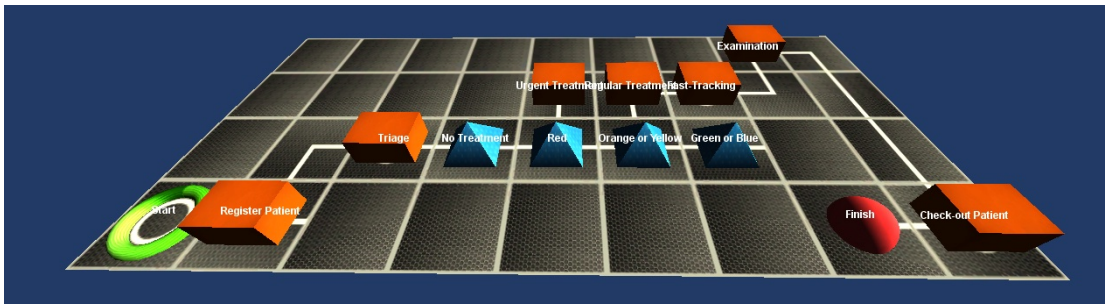


Abbildung 2.5: In ImPROVE wird ein Modell stets im Kontext eines 2.5D-Grid dargestellt. Quelle: [San11]

ISEASimuLator Ähnlich wie in den oben beschriebenen Arbeiten beschäftigt sich Santorum auch mit einem Serious Game für Business Process Management [San11]. Oft führt mangelndes Interesse der Mitarbeiter bzw. Stakeholder einer Firma die vorhandenen, bereits definierten Prozesse zu lernen oder zu verbessern zu Kommunikations- und Ablaufproblemen zwischen den verschiedenen Sektoren der Firma. Um dieses Interesse zu steigern wird das Prozessmodellierungssimulationsspiel *ISEASimulator* entwickelt. Wie bei West, Brown und Recker [WBR10] wird das Spiel in Second Life implementiert, hier allerdings besteht das Prozessmodell nicht aus Objekten in der Spielwelt, sondern wird komplett auf virtuellen Displays innerhalb der Spielewelt dargestellt. Die Teilnehmer nehmen eine von zwei Rollen an: Facilitator oder Participant. Darüber hinaus ist das gesamte Spiel in drei Pakete unterteilt:

- Rollenmanagement-Paket
- Simulations-Paket
- Evaluations-Paket

Diese können selektiv aktiviert werden und fügen dem Spiel entsprechend Elemente hinzu.

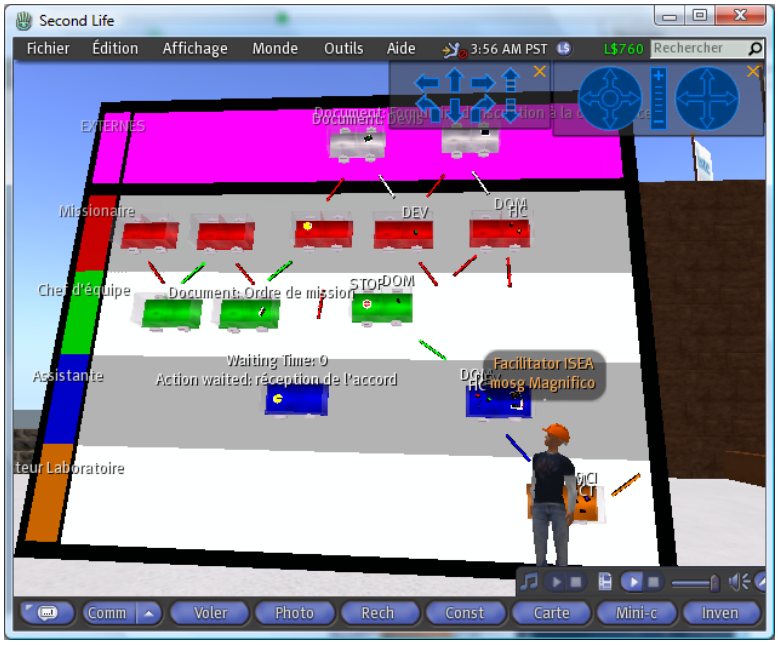


Abbildung 2.6: Ein Prozessvorgang in Second Life. Quelle: [San11]

2.3 Umgebungen für Prozessmodellierungen

BPM in Second Life West, Brown und Recker beschäftigen sich nicht mit dem Erlernen, sondern dem Modellieren eines Prozesses. Dies soll online und kollaborativ geschehen, das Endprodukt soll ein BPM (Business Process Management)-Modell sein. Zu diesem Zweck wird auf Second Life, ein Online-Rollenspiel zurückgegriffen. Wie im Spiel vorgesehen hat jeder Teilnehmer seinen eigenen Avatar, der als 3D-Person in der ebenfalls dreidimensionalen Welt repräsentiert wird (Abbildung 2.7). Als Alternative zu konventionellen 2D-GUIs laufen die Avatare nun buchstäblich auf den Elementen des Modells (die als Objekte in der Spielwelt dargestellt sind) herum. Spezielle Orte in der Spielewelt wie der *Mind Mapper* (Ein eingebautes Tool zum gemeinsamen Planen und Überlegen) oder das *Information Holodeck* (Ein virtuelles Display, welches das Speichern und Laden von Szenarien erlaubt) ersetzen konventionelle Menüs. [WBR10]

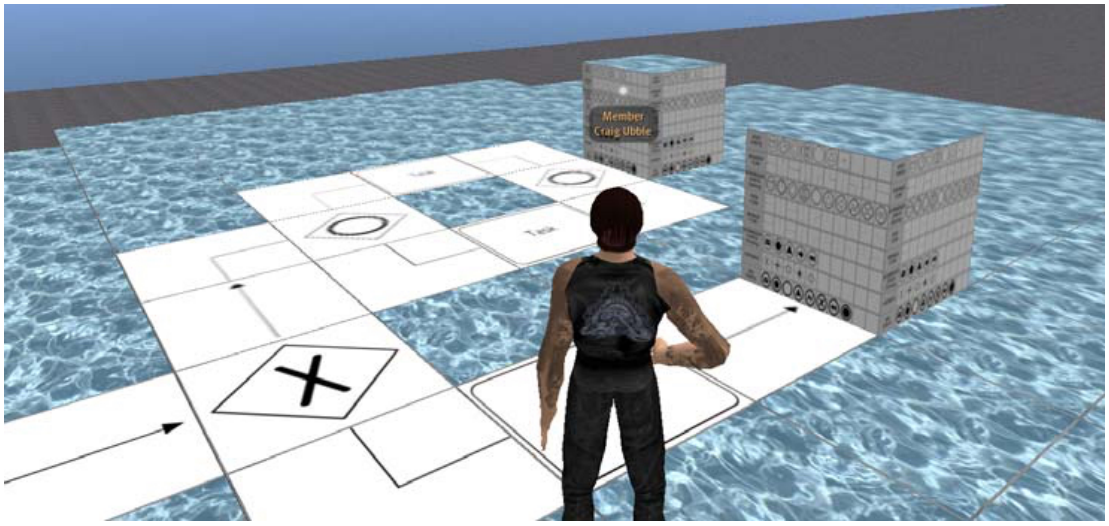


Abbildung 2.7: Der Nutzer interagiert als 3D-Avatar mit dem 2D-Prozessmodell in Second Life. Quelle: [WBR10]

Scenario Assistant (SCENAS) Streicher, Szentes und Roller entwickeln den *Mobile Scenario Assistant*, der Nutzern mobiler Endgeräte beim Setup und Verwenden ebendieser unterstützen soll [SSR14]. Dieses System ist kein Serious Game, verwendet aber ähnliche Techniken. Allen voran das Verstecken der Komplexität des Konfigurationsprozesses. Dies wird durch das Bereitstellen einer abstrakten, nicht-technischen Ansicht erreicht, die durch zusätzliche Hilfestellungen und Informationen ergänzt wird. Als konkreten Anwendungsfall wird das Experimental System for Image Exploitation (ExBA) beschrieben, ein heterogenes System zur

Bildauswertung. Dessen inhärente Komplexität wird versteckt durch eine Kombination von Assistenz und E-Learning. Das System wird durch eine sog. modulare Szenario-Beschreibung unterteilt. Die assistierte Nutzung der einzelnen Module wird dann durch eine Kombination von Agenten (Controller-Modulen) und Klienten (einem grafischen GUI) ermöglicht. [SSR14]

Abbildung 2.8: Das GUI von SCENAS auf einem Smartphone. Quelle: [SSR14]

3 Hintergrund

Diese Sektion beschriebt zunächst die Ziele, Funktionsweise und Limitierungen des dieser Arbeit zugrundeliegenden Spiels EXTRA. Danach werden die relevanten Aspekte des Forschungsgebiets der Serious Games im Kontext von Prozessmodellierung diskutiert. Abschließend werden die für die Implementierung der konzeptionierten Erweiterung wichtigen Technologien beschrieben.

3.1 Exercise Trainer - EXTRA

Die Grundlage für diese Arbeit bietet das 2016 von A. Gundermann entwickelte Serious Game Exercise Trainer (EXTRA) [SSG16; Gun16]. Dabei handelt es sich um ein webbasiertes Spiel zum Erlernen von Prozessabläufen.

Abbildung 3.1: Ein EXTRA-Level: Der Nutzer muss die Gebäude korrekt platzieren und verbinden.

Ziele Das Ziel von EXTRA ist das Erhöhen der Lernmotivation durch das Verwenden von Techniken des DGBL (Digital Game Based Learning). Die Ziele des Spiels wurden in Kooperation mit NATO erarbeitet: Als militärische Organisation hat diese die spezielle Herausforderung die komplexen Prozessabläufe für große Manöver einer sehr heterogenen Nutzergruppe zu lehren. Zentrale Lernziele sind hierbei das Verstehen von Rollen der verschiedenen Individuen, von Aktivitäten und Datenflüssen sowie das Verständnis der Folgen von Fehlern. Um diese inhärente Komplexität nicht noch unnötig zu erhöhen beschränkt sich EXTRA auf sehr einfache Mechaniken.

Spielkonzept EXTRA-Level finden in einer 2.5D-Welt (d.h. aus einer isometrischen Perspektive, Abbildung 1.1) statt. In dieser Welt können verschiedene Spielelemente gebaut werden:

- **Fabrik**: Fabriken entsprechen den Rollen, Einheiten oder datenverarbeitenden Systemen im unterliegenden Prozess. Sie haben ggf. Input und Output-Funktionalität, „produzieren“ als Produkte. Die Rate dieser Produktion ist von der Anzahl des zugewiesenen Personals abhängig. Fabriken sind entweder bereits platziert oder vom Spieler baubar.
- **Marktplatz**: Märkte entsprechen Zielprozessen. Sie akzeptieren und lagern bestimmte Produkte. Als Ende der Produktionskette ermöglichen Sie es dem Spieler Produkte zu verkaufen.
- *Route*: Routen erlauben es dem Nutzer Fabriken und Märkte zu verbinden um den Transport von Produkten zu ermöglichen. Es gibt drei verschiedene Arten von Routen: Straßen, Wasserwege und Schienen. Diese unterscheiden sich in ihrer Transportgeschwindigkeit und Kapazität.
- **Logistikzentrum**: Diese repräsentieren *Shared Databases* und erlauben es verschiedene Route-Typen zu verbinden.

Darüber hinaus gibt es noch die folgenden (konzeptionellen) Spielelemente:

- **Verbindungen**: Implementiert durch die oben beschriebene Routen, Verbindungen definieren den Fluss von Produkten und Informationen mit einer Richtung und einer Kapazität.
- **Produkte**: Diese repräsentieren verschiedene Informationen die transportiert werden können. Im Spiel sind sie farblich unterschieden.
- **Kunden**: Der Gegner des Spielers. Zeitlich bedingt stellen Kunden Anforderungen von Produkten, die der Spieler ihnen liefern muss. Geschieht dies nicht rechtzeitig, zieht das einen Punkteverlust nach sich, der proportional zur *Purchasing Power* des Kunden ist.

Implementierung EXTRA ist als NPM-Modul implementiert und besteht aus einer Server- und einer Client-Komponente da langfristig die Umsetzung diverser Online-Aspekte (wie ein High Score System oder Ingame-Events) möglich sein soll. Dies ermöglicht auch ein zentrales Datenmanagement und serverseitige Verifizierung. Aus Gründen der Modularität basiert EXTRA auf der bekannten MVC (Model View Controller)-Architektur.

Welt-Status EXTRA basiert auf der sog. Redux-Idee [Gun16; SA]. Der Status der Welt zu einem gegeben Zeitpunkt n $state_n$ ist ein JavaScript-Objekt, das zu JSON serialisiert werden kann. Das Redux-Prinzip besagt, dass dieser Status stets mittels der folgenden Gleichung aus dem vorherigen Status berechnet werden kann:

$$state_n = reducer(state_{n-1}, action)$$

In diesem Zusammenhang gilt es die drei Redux-Prinzipien zu beachten:

- **Single source of truth**: Alle Informationen sind im Status gekapselt.
- **State is read only**: Der Status kann nicht verändert, nur ersetzt werden.
- **Changes are made with pure functions**: Es treten keine Seiteneffekte auf.

Die Kombination der obigen Gleichung und den Prinzipien ermöglicht die Serialisierung von Status und Sequenz der Aktionen und somit die einfache Implementierung von *undo*-Funktionalität, einfaches Unit-Testen und die Kombination bzw. Integration von Learning Management Systemen. Es existieren die folgenden Aktionen. Für Details siehe [SSG16].

```
tick()
placeStructure(x,y,index)
removeStructure(x,y)
sellProduct(id)
changeFactory(x,y,index)
addFactoryWorkforce(x,y,n)
removeFactoryWorkforce(x,y,n)
connect(path, index)
removeConnection(id)
incrementLoad(id)
decrementLoad(id)
```

Szenarien EXTRA erlaubt die Spezifikation von Szenarien. Dabei handelt es sich um eine Mengen von Dateien, die folgende Informationen enthalten:

- JSON-Levels
- Übersetzungsdateien
- Bilder
- Prozess-Dateien (UML/BPMN)
- Startlevelspezifikation $state_0$

Limitierungen Bei Beginn der Arbeit unterstützt EXTRA nur lineare Fabriken, d.h. Fabriken mit einem Input und einem Output. Konkret haben alle Fabriken die folgende JSON-Definition:

```
{
    "type": "factory",
    "name": "A",
    "image": "icon.png",
    "workforce": 4,
    "configurations": [
        {
            "input": "one",
            "name": "default",
            "output": "two"
        }
    ]
}
```

Sie haben Informationen über Namen, Typ der Struktur, ihre grafische Repräsentation, die Belegschaft sowie eine Liste von Konfigurationen. Jede dieser Konfigurationen hat einen Namen sowie zwei Zeichenketten: Die Namen des eingehenden und des ausgehenden Produktes. Es ist also nicht möglich hier mehrere einzutragen. Darüber hinaus ist der Typ der Verbindung implizit, wird also nicht abgespeichert. Für die hier vorliegende 1:1-Abbildung ist das möglich, eine hier gewünschte $n : 1$-Abbildung benötigt allerdings einen Typ.

3.2 Serious Games

Unter Serious Games versteht man Spiele deren charakteristisches Ziel nicht die Unterhaltung, sondern ein anderer Zweck ist. Dabei kann es sich um Lehre, Training oder ähnliches handeln. Psychologische Gründe für das Nutzen von Spielen zum Lernen werden in [Fit18; CL87; ST03] beschrieben.

Ribeiro et. al. führen im Speziellen die folgenden drei Gründe an [Rib+12]:

- Visuelle Repräsentation von abstrakten Sachverhalten
- Einfachheit der Darstellung (im Vergleich zu konventioneller Software)
- Attraktivität und intrinsische Motivation (insb. für jüngere Generationen)

Serious Games können als Antwort der Bildungsinstitute an die *Digital Natives* gesehen werden [De 17]. Das heißt solchen Generationen, die ohnehin mit Computern oder mobilen Geräten aufgewachsen sind und diese Technologien auch in ihrer (Aus-)Bildung erwarten.

Im Business-Kontext werden Serious Games primär für drei Zwecke eingesetzt:

- Orientierung neuer Angestellter
- Auswahl von Managern
- Management-Training

Eine hier entscheidende Beobachtung ist, dass Business Process Modelling ohnehin computerabhängig geschieht, die Einführung neuer bzw. die Erweiterung bestehender Software ist deswegen naheliegend. Unabhängig vom konkreten Kontext erlauben es Serious Games den Spielern/Lernern Dinge zu erleben, die sonst unmöglich werden (wegen Kosten, Sicherheit etc.). In anderen Worten: Sie erlauben *learning by doing* auch für sicherheitskritische Disziplinen. So entstanden die ersten Serious Games 1956 in den beiden Sektoren Nuklearenergie und Luftfahrt (Top Management Decision Simulation [Rib+12]). Weitere Vorteile sind [Con+12; GEM13]:

- Echtzeit-Feedback
- Aktive Teilnahme der Nutzer
- Lernen aus Erfahrung
- Motivation durch Wettbewerb
- Beobachtung von Schlüsselkonzepten

Simulationen In solchen Umgebungen wird das Simulationsparadigma verwendet (Entwerfe ein Modell und führe darauf Experimente durch). Für geschäftliche Simulationen gilt es hierbei noch die komplexen Interaktionen zwischen Parteien und Rollen zu beachten. Eine Besonderheit von simulationsbasierten Serious Games ist ihre Eigenschaft, dass sie (bedingt durch die aktiven Entscheidungen des Spielers) keine statischen Regeln haben. Dies beeinflusst jedoch auch die Anwendbarkeit ihrer Ergebnisse in der echten Welt.

Klassifikation Das Feld der Serious Games kann anhand verschiedener Charakteristiken klassifiziert werden. In [SR16] stellen Strecker und Rosenthal die folgenden vor:

- Klassifikation anhand des Zielmarktes: Serious Games werden in Kontexten wie denen der Industrie, des Militärs, der Bildungsinstitutionen oder des Gesundheitswesens eingesetzt. Die unterschiedlichen Anforderungen machen eine Klassifikation möglich
- Alternativ können Serious Games anhand der vier folgenden Dimensionen kategorisiert werden
    1. Primärer Bildungsinhalt
    2. Primäres Lernprinzip
    3. Ziel-Altersgruppe
    4. Plattform

Assistenzsysteme Assistenzsysteme dienen dazu die Lücke zwischen technologischem Potential und natürlichen Möglichkeiten zu schließen. Dies ist insbesondere bei komplexen Systemen sinnvoll (Autos, Fabriken, HCI). Zum Lernen sind v.a. Kritik bzw. Feedback sowie *Gamification* geeignet. Dies beschreibt das Anwenden von Game-Design in neuen Bereichen, z.B. durch Visualisierung, Erfolge, Punktevergabe, Quizze. Genauer lassen sich diese Systeme in die folgenden Kategorien unterteilen [Gun16]:

- Gamified Assistance
- Computer Based Assistance
- Pre-Training-Tools

Sheridan schlägt eine Klassifikation nach Grad der Automatisierung vor (Stufen 0 - 5), Wandke nach einer Reihe von Kategorien (Entscheidungshilfe, Feedback, Adaption, Adaptierung). [She88; Wan05]

Digital Game Based Learning DGBL kann als Antwort der Bildungssysteme auf den Übergang der Zielgruppe von den *Digital Immigrants* (d.h. jenen Generationen, die noch weitestgehend ohne digitale Medien bzw. Geräte aufgewachsen sind) zu den *Digital Natives* (d.h. jenen Generationen die bereits in ihrer Kindheit mit den neuen Technologien vertraut waren). Motivation der Lernenden ist von essenzieller Bedeutung: Abhängig vom Kontext gilt es psychologischen Modelle wie *Flow* (Erhaltung absoluten Fokus durch richtigen Schwierigkeitsgrad, d.h. nicht zu schwer, nicht zu leicht) oder *Foggs' s Behaviour Model* (Lernbereitschaft setzt sich aus den Faktoren Motivation, Fähigkeit und Trigger zusammen) zu beachten [Fog09].

In diesem Kontext sind Serious Games also eine Variante von DGBL die versucht Spiel und Didaktik zu kombinieren. Allerdings ist dabei nur eine optimale Kombination nützlich, es gilt den „Shavian Reversal“ [Ecko3] zu vermeiden, die Eigenschaft eines Serious Games, nicht zur Unterhaltung und nicht zur Bildung zu dienen. Eine Möglichkeit ist es die Erstellung von Nuter-Inhalten bzw. Mods zu erlauben.

3.3 Prozessmodellierung

Im Rahmen dieser Arbeit wird die Prozess-Definition von Schoknecht [Sch17] verwendet: Ein Business-Prozess kann als eine Folge von Aktivitäten in einem Business-Kontext, die der Herstellung von Gütern und Services dienen. Diese Definition kann in zwei Dimensionen angewendet werden:

1. Wird ein Prozess in der Realität oder in einem Modell beschrieben?
2. Wird eine konkrete Instanz betrachtet?

Auf dieser Terminologie aufbauend wird nachfolgend die *Repräsentation* von Prozessen betrachtet. Diese lässt sich in drei Kategorien teilen: Informal, Semi-formal und formal. Für die semi-formelle oder formelle Repräsentation von Prozessen existieren eine Reihe von Sprachen, in dieser Arbeit werden zwei betrachtet: BPMN und UML. Für einen ausführlichen Vergleich siehe [Peroo].

3.3.1 BPMN

Bei *Business Process Modelling Notation* (BPMN) handelt es sich um einen Standard für die Definition von Geschäftsprozessen, insb. auf Stufen höherer Abstraktion (*Domain Analysis, System Design*) [DDOo8]. BPMN kann als Weiterentwicklung vorheriger Sprachen wie *XML Process Definition Language* (XPDL) oder auch der hier betrachteten UML-Aktivitätsdiagramme angesehen werden indem sie einige dort bereits vorhandener Elemente kombiniert oder weiterverwendet. BPMN-Prozessmodelle bestehen im wesentlichen aus drei Arten von Komponenten:

- Eventknoten: Ereignisse oder von Menschen oder Maschinen/Software verrichtete Arbeit
- Kontrollknoten: Stellen, die den Kontrollfluss zwischen Events sicherstellen
- Flussbeziehung (d.h. Kanten)

Sprachen, die diesem Paradigma entsprechen werden auch *Graphorientierte Prozessdefinitionssprachen* genannt. Die Semantik dieser Sprachen formal (bzw. statisch/automatisiert) zu untersuchen ist eine Herausforderung. Arbeiten wie die von Dijkman [DDOo8] demonstrieren, dass BPMN-Modelle oft Fehler enthalten.

Dazu zählen:

- Deadlocks
- Livelocks
- Semantische Fehler
- Unkontrollierte, parallele Subprozesse
- Inkonsistente Ergebnisse im Fehlerfall

Grund dafür ist die inexakte (teils auch inkonsistente) Definition von Semantik im Standard. Dieser besteht zwar aus vielen Tabellen und Beispielen, die die Syntax dokumentieren, die konkreten semantischen Definitionen bestehen aber nur aus Fließtext [DDO08]. Diese Arbeit behandelt dies wie folgt:

- Es wird nur eine Teilmenge von BPMN betrachtet, dies vermeidet Fehler bzgl. Parallelität, da diese nicht ausdruckbar ist
- Um mit semantischer Inkonsistenz umzugehen werden die Konzepte von *wohlgeformtem* bzw. *nicht wohlgeformtem* BPMN definiert
- Die folgende Tabelle 3.1 definiert Syntax und Semantik für *wohlgeformtes* BPMN exakt.

Die hier verwendete Übersetzung auf Graphen ist nicht der einzige Ansatz BPMN-Modelle auf eine andere Domäne abzubilden: Dijkman et. al [DDO08] stellen einen Ansatz vor, BPMN auf die Theorie und Formalismen von Petri-Netzen zu übersetzen, Tsagkani und Tsalgatidou [TT15] übersetzen BPMN-Elemente auf eine eigens definierte Prozessmodellierung und Chinosi und Trombetta [CT12] geben eine Zusammenfassung über den Umgang mit BPMN in (anderen) Softwareumgebungen.

Wohlgeformtes BPMN Formell haben Kontrollknoten eine eröffnende Komponenten und eine schließende, dies definiert ihren Wirkungsbereich eindeutig. Oft gibt es allerdings auch BPMN-Modelle die nur einen eröffnenden Knoten haben: Diese werden im Rahmen dieser Arbeit als *nicht wohlgeformtes BPMN* bezeichnet. Diese sind im Allgemeinen formell nicht äquivalent zum oben definierten BPMN und werden deshalb von der hier vorgestellten Implementierung nicht unterstützt. Ein Ansatz mit ihnen umzugehen wird aber in Sektion 4.1.1 genauer beschrieben, dieser kann eventuell in Zukunft als Grundlage für einen Fehlerkorrekturalgorithmus dienen.

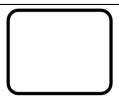
	**Start-Event**: Jeder Prozess hat genau ein solches Event, es repräsentiert den Beginn des Prozesses als initiale Aktion. Es hat keine eingehenden Kanten und beliebig viele ausgehende Kanten.
	**End-Event**: Jeder Prozess hat genau ein solches Event, es repräsentiert das Ende des Prozesses als finale Aktion. Es hat beliebig viele eingehende Kanten und keine ausgehenden Kanten.
	**Aktion**: Jeder Prozess hat beliebig viele solcher Events, sie liegen stets zwischen Start- und Endevent und haben genau eine eingehende und eine ausgehende Kante.
	**Und-Event**: Jeder Prozess hat eine gerade Anzahl dieser Event, sie liegen stets zwischen Start- und Endevent. Es gibt zwei Typen, die aber die gleiche Notation haben: Eröffnende Events haben eine eingehende Kante und mehrere (o.B.d.A. n) ausgehende Kanten, schliessende Events haben n eingehende Kanten und eine ausgehende Kanten. Die Semantik ist die folgende: das schliessende Event wird nur ausgeführt wenn alle n adjazenten Events ausgeführt werden.
	**Oder-Event**: Jeder Prozess hat eine gerade Anzahl dieser Event, sie liegen stets zwischen Start- und Endevent. Es gibt zwei Typen, die aber die gleiche Notation haben: Eröffnende Events haben eine eingehende Kante und mehrere (o.B.d.A. n) ausgehende Kanten, schliessende Events haben n eingehende Kanten und eine ausgehende Kanten. Die Semantik ist die folgende: das schliessende Event wird nur ausgeführt wenn mindestens eines der n adjazenten Events ausgeführt werden.

Tabelle 3.1: Die in dieser Arbeit betrachteten BPMN-Elemente.

3.3.2 UML

Die *Unified Modeling Language* (UML) ist eine grafische Sprache bzw. ein Standard der das Visualisieren, Spezifizieren und Dokumentieren von Artefakten softwareintensiver Systeme erlaubt [Hau03]. Dieser Standard definiert zum einen mehre Diagrammtypen (wie Klassen-, Komponenten-, Paket- und Sequenzdiagramme) und zum anderen eine theoretische/formelle Grundlage die das Erweitern dieser Typen erlaubt. Eine solche Erweiterung ist die Sprache *Systems Modeling Language* SysML, die es erlaubt komplexe Systeme zu spezifizieren, zu analysieren und zu verifizieren und zu validieren . Dazu zählen unter anderem:

- Hardware
- Software
- Daten
- Prozesse
- Fabriken/Einrichtungen

Eine Erweiterung des hier implementierten Editors um diese Sprache könnte sinnvoll sein.

Aktuell bzw. in dieser Arbeit liegt der Fokus aber auf UML im Business-Kontext. Hier gibt Heumann [Heu03] einen Überblick über die relevanten Diagrammtypen:

- Use Case-Diagramme
- Aktivitätsdiagramme
- Klassendiagramme
- Sequenzendiagramme

In dieser Arbeit Fokus auf Aktivitätsdiagramme, da diese am nächsten den BPMN-Prozessen entsprechen und somit das einfache Übersetzen der jeweiligen Repräsentation in ein generisches Format erlaubt. Ähnlich wie bei BPMN wird in Tabelle 3.2 die genaue Syntax und Semantik der unterstützten Teilmenge definiert um Fehler bzw. Mehrdeutigkeiten zu vermeiden und somit das automatische Übersetzen in eine generische Repräsentation zu ermöglichen. Ein alternativer Ansatz die Semantik von in UML modellierten Prozessen zu beschreiben, basierend auf der Theorie der *Prozessalgebra* wird von [Xu11] beschrieben. [Yi08] stellt eine Erweiterung von Aktivitätsdiagrammen vor, die es erlaubt diese in *General Purpose System Simulation* (GPSS) zu verwenden.

Da in UML-Aktivitätsdiagrammen die Kontrollflusselemente bereits syntaktisch symmetrisch (d.h. mit öffnenden und schließenden Komponenten) definiert sind ist es nicht üblich die schließenden Elemente in der Modellierung wegzulassen, analog zu BPMN ist allerdings auch hier eine theoretische Behandlung möglich (Sektion 4.1.1).

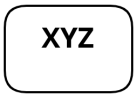
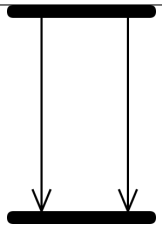
	**Start-Aktivität**: Jeder Prozess hat genau eine solche Aktivität, sie repräsentiert den Beginn des Prozesses als initiale Aktion. Sie hat keine eingehenden Kanten und beliebig viele ausgehende Kanten.
	**End-Aktivität**: Jeder Prozess hat genau eine solche Aktivität, sie repräsentiert das Ende des Prozesses als finale Aktion. Sie hat beliebig viele eingehende Kanten und keine ausgehenden Kanten.
	**Aktivität**: Jeder Prozess hat beliebig viele solcher Aktivitäten, sie liegen stets zwischen Start- und Endaktivität und haben genau eine eingehende und eine ausgehende Kante.
	**Und-Aktivität**: Jeder Prozess hat beliebig viele solcher Aktivitäten, sie liegen stets zwischen Start- und Endevent. Sie bestehen aus zwei Komponenten: Die eröffnende Aktivität hat eine eingehende Kante und mehrere (o.B.d.A. n) ausgehende Kanten, schliessende Aktivitäten haben n eingehende Kanten und eine ausgehende Kanten. Die Semantik ist die folgende: das schliessende Event wird nur ausgeführt wenn alle n adjazenten Aktivitäten ausgeführt werden.
	**Oder-Aktivität**: Jeder Prozess hat beliebig viele solcher Aktivitäten, sie liegen stets zwischen Start- und Endevent. Sie bestehen aus zwei Komponenten: Die eröffnende Aktivität hat eine eingehende Kante und mehrere (o.B.d.A. n) ausgehende Kanten, schliessende Aktivitäten haben n eingehende Kanten und eine ausgehende Kanten. Die Semantik ist die folgende: das schliessende Event wird nur ausgeführt wenn mindestens eine der n adjazenten Aktivitäten ausgeführt werden.

Tabelle 3.2: Die in dieser Arbeit betrachteten UML-Elemente.

3.4 Web-Technologien

Diese Sektion beschreibt die in EXTRA und im implementierten Editor verwendeten Technologien, besonderer Fokus wird hierbei auf die Verwendungsweise verschiedener Bibliotheken gelegt.

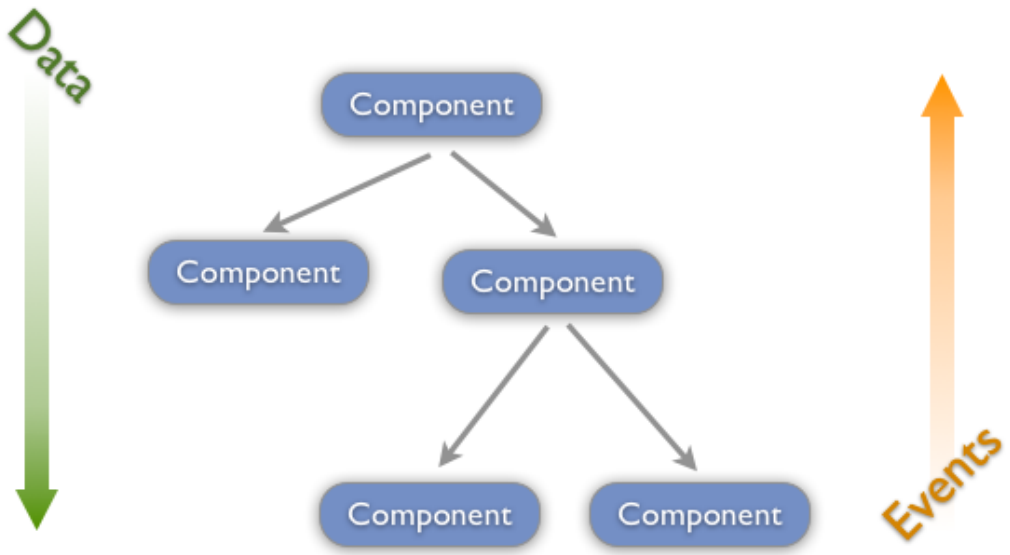


Abbildung 3.2: Der hierarchische Aufbau einer React-Applikation, Daten werden nach unten weitergegeben, Events werden nach oben delegiert. Quelle: [Gun16]

React Das dieser Arbeit zugrunde liegende Hauptspiel EXTRA ist in React implementiert, um auf einer Vielzahl von Plattformen ausführbar zu sein. Aus den selben Gründen wird der hier vorgestellte Editor auch mittels React umgesetzt. React ist ein von Facebook entwickeltes Framework zur Entwicklung von webbasierten User-Interfaces und Applikationen. Zu den besonderen Eigenschaften gehört die Tatsache, dass HTML allein mittels JavaScript erzeugt wird, es gibt kein Template-System oder Einbindung statischer HTML-Dateien (im Gegensatz zu Konkurrenten wie Angular oder Vue). Da es (konzeptionell) kein klassische Webseiten gibt, existiert auch kein Document Object Model (DOM). Für die von dieser Schnittstelle bereitgestellten Funktionalitäten bietet React als Alternative die React-DOM an. Der grobe Aufbau einer React-Applikation ist in Abbildung 3.2 dargestellt. Die *gesamte* Anwendung besteht aus JavaScript. Eines der Grundprinzipien von React ist: *Everything is JavaScript.* Genauer lässt sich die Anwendung in (unabhängige) Klassen, die Komponenten unterteilen. Es

lassen sich zwei Arten von Komponenten unterscheiden, sog. *stateless* und *stateful* Components. Die nachfolgenden Ausschnitte verdeutlichen die zentralen Unterschiede [Gac15a; Gac15b] :

Stateless Components Komponenten des Editors, die nur der Darstellung von Level-Informationen dienen können als solche simpleren Komponenten implementiert werden. JavaScript (genauer: ES5+) hat Unterstützung für viele Aspekte, die aus dem Paradigma des funktionalen Programmierens stammen. Dazu zählt insbesondere die Möglichkeit Funktionen als Variablen zu behandeln (sog. *First class citizens*). In Kombination mit der Syntax der *Arrow-Functions* ermöglicht dies die Definition von React-Komponenten als Funktion in einer Variable wie nachfolgend dargestellt.

```
1 const Modal = props => {...}
```

Diese Funktion hat ein Parameter-Objekt *props*. Aus diesem Objekt wird sog. JSX-Code und schlussendlich HTML-Code generiert. Details zu diesem Prozess werden später gegeben. Der Begriff *stateless* bezieht sich auf die Eigenschaft dieser Komponenten selbst keine persistenten Information zu speichern. In anderen Worten: sie sind frei von Seiteneffekten („pure“), es gilt also stets: $p_1 = p_2 \Rightarrow Modal(p_1) = Modal(p_2)$.

Stateful Components Editor-Komponenten, die einen (aktualisierbaren) Status haben (wie die Hauptkomponente) werden objektorientiert implementiert. Im Gegensatz zu *stateless Components* wird hier explizit von der React-Komponentenklasse geerbt wie im nachfolgenden Code-Ausschnitt verdeutlicht.

```
1 import React, {Component} from "react"
2
3 export default class TabComponent extends Component {
4   constructor(props) {
5     super(props);
6    }
7    render() {...}
8 }
```

Dabei muss die render()-Methode überschrieben bzw. implementiert werden. In der somit induzierten objektorientierten Semantik wird der Klasse auch ein spezielle Variable namens *state* zur Verfügung gestellt. Diese wird nach ihrer Initialisierung stets über eine spezielle Wrapper-Methode **asynchron** aktualisiert.

Dies hat folgende Eigenschaften zur Folge:

- Die Komponente ist nicht mehr „pure“ d.h. seiteneffektfrei
- Ändern sich Daten in der Applikation wird *state* stets durch ein gänzlich neues Objekt ersetzt. (Die Effizienz dieser Operation wird intern durch Diff-Algorithmen sichergestellt.)
- Die Asynchronität der Änderungen ermöglicht ihre Gruppierung und minimiert die Anzahl der Re-Render-Operationen

Lifecycle Hooks Wie in Abb. 3.3 dargestellt, durchläuft jede React-Komponente vier Phasen: Initialisierung, Mounting, Update, Unmounting.

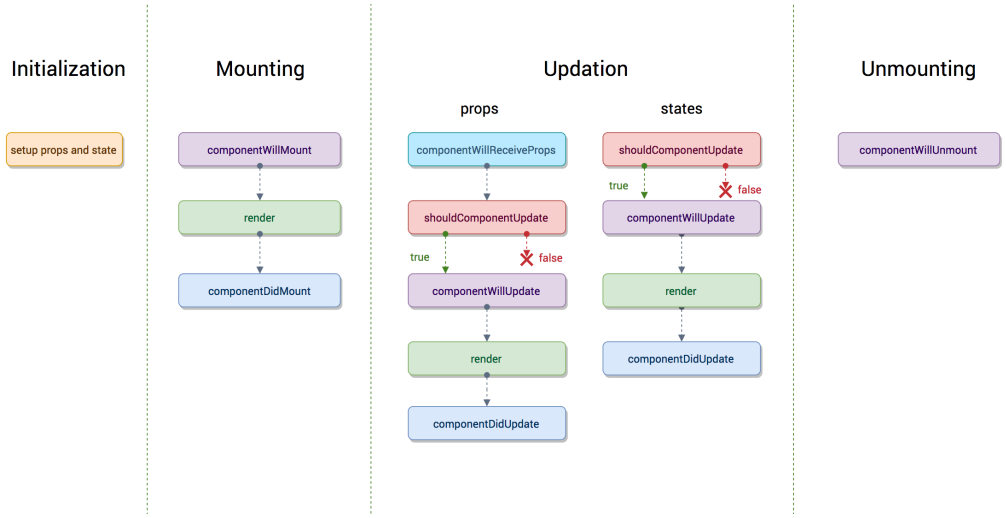


Abbildung 3.3: Die vier Phasen, die eine React-Komponente durchläuft erlauben das Nutzen von bestimmten Methoden (sog. *Hooks*). Quelle: [M17]

Jede dieser Phasen ermöglicht es Komponenten die angegebenen Methoden zu implementieren. Dies kann mehre Gründe haben:

- **Performance**: shouldComponentUpdate() berechnet stets einen booleschen Wert und gibt ihn zurück. Dieser signalisiert React ob die Komponente neu gerendert werden soll. Eine Minimierung der render-Operationen erhöht die Performance der Applikation. So ist es z.B. sinnvoll manuell zu bestimmen ob die Änderung der Daten der Komponente sichtbare Auswirkungen hat. Falls nicht muss auch nicht neu gerendert werden.
- **Ausführungssynchronisation**: Die Beziehung zwischen React-Javascript und klassischem JavaScript ist komplex, da React den Zustand und Lebenszyklus aller Komponenten kontrolliert und diese unregelmäßig (als Reaktion auf Datenänderungen) und

asynchron aktuallisiert. Dies ist insbesondere dann relevant wenn Komponenten andere Bibliotheken wie Dagre, Cytoscape oder Phaser verwenden. Falls jene nämlich Referenzen auf HTML-Elemente benötigen („gebunden“ werden) existieren diese zu Render-Zeitpunkt eventuell noch gar nicht. Deswegen werden solche Operationen stets in der componentDidUpdate()-Methode ausgeführt, dort ist deren Existenz nämlich garantiert.

JavaScript, Node.js und Webpack Neben der deklarativen Sprache HTML ist JavaScript die wichtigste Programmiersprache im Web-Kontext (Anfang 2018 wurden wöchentlich über 4 Milliarden NPM-Pakete heruntergeladen [NPM18], 500 der Fortune 500 verwenden Node.js [NPM19]), sie sorgt für die eigentliche (d.h. interaktive und imperative) Funktionalität von Webseiten. Dafür benötigt sie eine Engine, ein Programm das JavaScript-Code ausführt. Zu den wichtigsten Engines zählen hierbei *V8* von Chrome, *SpiderMonkey* von Firefox und *JavaScriptCore* von Apple. Normalerweise läuft eine JS-Engine also im Browser. Die in dieser Arbeit verwendete Laufzeitumgebung *Node.js* nutzt allerdings die *V8* Engine um JavaScript-Code auch außerhalb des Browsers auszuführen. Zusammen mit einem Paketmanager (NPM oder alternativ Yarn) ermöglicht diese Umgebung die Entwicklung von JavaScript-Applikationen, die Pakete dritter als Abhängigkeiten verwenden können. Nahezu alle in dieser Arbeit verwendeten Bibliotheken (und React) werden auf diese Art eingebunden. Um diese Applikation aber schlussendlich auch außerhalb dieser Umgebung (insb. im Browser) lauffähig zu machen sind zwei weitere Schritte nötig: Konfiguration und *Bundling*. Hierfür wird Webpack verwendet. Eine Software, die die Node-Applikation so nachbearbeitet (d.h. relative Referenzen auflöst, Assets bereitstellt, Code minifiziert etc.), dass sie ausgeliefert und ausgeführt werden kann. 3.4 stellt diesen Vorgang konzeptionell dar.

Phaser Phaser[1] ist eine Bibliothek (genauer: Spiel-Engine), die Wrapperfunktionen für bereits vorhandene HTML5-Funktionalitäten, die das native Implementieren von Spielen erlauben, bereitstellt. Aktuell befindet sich Phaser in Version 3, da EXTRA aber mithilfe von Version 2 entwickelt wurde, verwendet auch der in dieser Arbeit implementierte Editor Phaser 2. Phaser erlaubt das Einbinden von Plugins, d.h. Sub-Bibliotheken von unabhängigen Dritten. Ein solches Plugin, das sowohl in EXTRA als auch im Editor von zentraler Bedeutung ist, ist das *Phaser Isometric plug-in*. Dies erlaubt es Level in isometrischer Perspektive darzustellen ohne manuell die dafür nötigen Transformationen berechnen zu müssen.

1 `https://phaser.io`

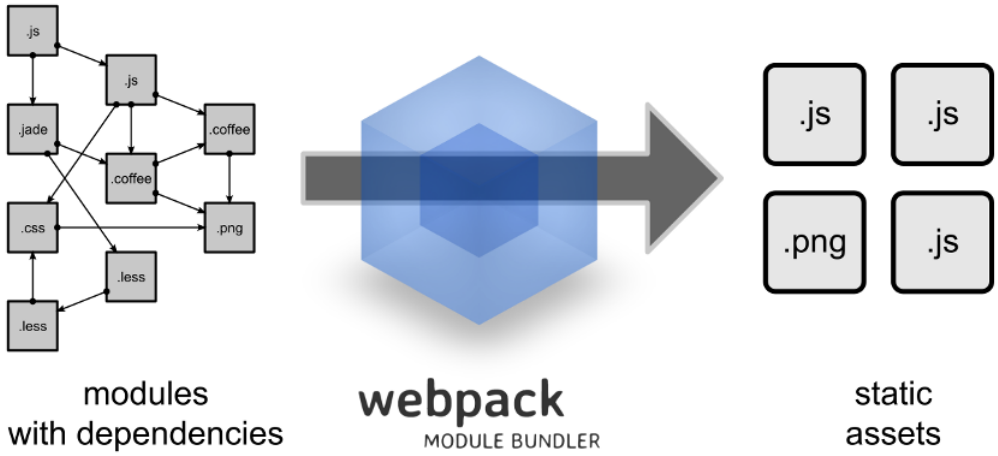


Abbildung 3.4: Ein vereinfachter, systematischer Überblick über den *bundling*-Prozess mit Webpack. Quelle: [Gun16]

Cytoscape.js Cytoscape ist eine Software-Umgebung für die Definition, das Layouting und das Rendern von Graphen, das hier verwendete Cytoscape.JS [2] ist eine Implementierung dieser Software in JavaScript. Sie wird verwendet um die aktuell im Editor geöffnete Levelinstanz als Graph darzustellen. Neben der nativen Darstellung von Graphen als HTML-Code (dessen Aussehen dementsprechend auch mit CSS editierbar ist) erlaubt Cytoscape auch partielles Re-Rendering (wichtig für die Graph-Darstellung im Editor) sowie die Interaktion mittels des Maus-Cursors (d.h. Scrollen und verschieben von Knoten und Kanten). Im Hintergrund lässt Cytoscape auch die Definition von (eigenen) Layout-Algorithmen zu, eine der Optionen ist auch das hier verwendete Dagre.

BPMN.JS Bei BPMN.JS [3] handelt es sich um die Implementierung eines grafischen Editors für BPMN-Diagramme, die nativ im Browser lauffähig ist. Sie wird im Editor für die folgende Funktionalität verwendet:

- Darstellung von hochgeladenen .bpmn-Dateien
- Hinzufügen von Events und Kanten (Topologie des Levels/ der generischen Repräsentation)
- Beschriften von Events und Kanten (Fabrik- und Produktnamen)
- Download von so erstellten Dateien

2 `http://js.cytoscape.org`
3 `https://bpmn.io/toolkit/bpmn-js/`

Codemirror Codemirror[4] ist eine JavaScript-Bibliothek, die es ermöglicht einen Text-Editor nativ im Browser darzustellen. In den hier implementierten Editor wird er verwendet um den konkreten JSON-Code des aktuellen Levels anzuzeigen. Änderungen hier werden im GUI angezeigt, Änderungen im GUI spiegeln sich dort in Echtzeit wieder. Diese Level-Datei ist alles was EXTRA für ein spielbares Level benötigt, sie kapselt somit implizit den aktuellen Zustand des Editors. Die Bibliothek ist mittels eines zweiten Pakets (*React-Codemirror2*) eingebunden. Dabei handelt es sich um eine React-Komponente, die die JavaScript-basierte Funktionalität von CodeMirror kapselt. Zu dieser Funktionalität gehören unter anderem:

- Unterstützung für Syntax-Highlighting für verschiedene Sprachen durch sog. Sprachpakete (in dieser Arbeit wird das JS/JSON package verwendet).
- Schnittstelle für zahlreiche Events wie (do/undo, bei Änderungen im Code etc.)
- Möglichkeit für das manuelle Markieren von Zeilen/Worten. Dies wird für das Highlighting von Fehlern und dem aktuell markierten Element verwendet.
- CSS-Styling (u.a. Schriftarten, Themes etc.)

Dagre und Graphlib

Dagre[5] ist eine Open Source-Bibliothek unter der MIT-Lizenz, die das Layout von ungerichteten Graphen erlaubt. Sie wird hier verwendet um die eigens entwickelte generische Prozess-Repräsentation darzustellen. Diese Graphen werden durch ein Graph-Objekt repräsentiert, welches von der Bibliothek Graphlib bereitgestellt wird. Ein solcher Graph besteht aus den folgenden Komponenten:

- Eine Liste von Knoten und Kanten, diese haben jeweils einen Namen und ein Label-Objekt (ein JavaScript-Objekt beliebiger Art)
- Eingebaute Methoden für eingehende und ausgehende Kanten, Vorgänger und Nachfolger, Eingangs- und Ausgangsgrad etc.
- Implizite Sanity Checks bzw. Convenience-Methoden: Zum Beispiel: Werden sowohl End- als auch Starknoten einer Kante entfernt, wird diese auch automatisch gelöscht. Es gibt keine klare Trennung zwischen Update und Erstellen von Knoten/Kanten, dies minimiert die Anzahl der nötigen Schritte

Layouting Für gerichtete, azyklische Graphen (DAGs) bietet die Bibliothek Dagre einen nach Brandes und Köpke [BK02] modifizierten Layout-Algorithmus an. Dieser ordnet jedem

4 `https://codemirror.net`
5 `https://github.com/dagrejs/dagre`

Knoten eine zweidimensionale Koordinate zu. Für die hier auftretenden Prozess-Graphen wird so die Topologie eines Baumes definiert. Dies kann (und wird) für den Leveling-Algorithmus ausgenutzt werden (Sektion 4.1.3).

Darüber hinaus verfügt Graphlib auch über die eingebaute Unterstützung für eine Reihe von Graphalgorithmen. Die in dieser Arbeit verwendeten Algorithmen werden im folgenden kurz beschrieben. Gegeben sei ein (Graphlib-)Graph $G = (V,E)$.

Dijkstra-Algorithmus In einer Subroutine des hier entwickelte Übersetzungsalgorithmus Split and Create müssen die Distanzen von Knoten zur Quelle berechnet werden. Der klassische Ansatz ist hierbei der Algorithmus von Dijkstra: Sei v ein gegebener Startnkoten. Berechne die Distanz zu allen (oder nur einem) anderen Knoten $v \in V$. Für die Berechnung werden die manuell definierten Kantengewichte $\omega(e),\ e \in E$ verwendet. Sind diese nicht vorhanden (was hier der Fall ist) ist das Gewicht jeder Kante gleich:

$$\forall e \in E.\ \omega(e) = \Omega,\ \Omega \in \mathbb{R} \tag{3.1}$$

Allgemein beschriebt das zu lösende kürzeste-Wege-Problem für zwei Knoten $v, u \in V$ das Finden eines Pfades $p = (v, \ldots, u)$ mit minimalem Gewicht $\omega(p)$.

$$\omega(p) \coloneqq \sum_{v \in p} \omega(v) \tag{3.2}$$

Augrund von 3.1 ist 3.2 nun äquivalent zum Finden eines Pfades p mit minimaler Pfadlänge $|p|$ Es ist also nur die Anzahl der Knoten (bzw. Kanten) relevant, nicht deren konkretes Gewicht Ω.

Graphtraversierung Im hier entwickelten Lösbarkeitsalgorithmus muss ein gegebener Graph traversiert werden. Das Traversieren eines Graphen bezeichnet das (sequenzielle) Besuchen aller über Kanten erreichbare Knoten ausgehend von einem Startknoten, ist also eine Instantiierung des Algorithmenschemas der Tiefensuche. Für die hier auftretenden DAGs gibt es verschiedene Arten, nachfolgend werden zwei betrachtet: Pre- und Post-Order.

Der Graph aus 3.5 wird vom Startknoten A aus traversiert. Das Ergebnis ist ein Pfad p. Die Reihenfolge der Knoten ist nichtdeterministisch, da bei Verzweigungen im Graph (d.h. mehreren nachfolgenden Knoten) der nächste Knoten frei gewählt werden kann. Erfolgt diese Pfadkonstruktion in Pre-Order beginnt der Pfad stets mit dem Starknoten: Der Graph wird traversiert und sobald ein unbesuchter Knoten betreten wird, wird dieser dem Pfad angehängt. Ein Ergbnis für den obigen Graph ist also: $p = (A, B, C, D, E)$. Erfolg die Traversierung dagegen in Post-Order endet der Pfad stets mit dem Starknoten: Wird hier ein unbesuchter

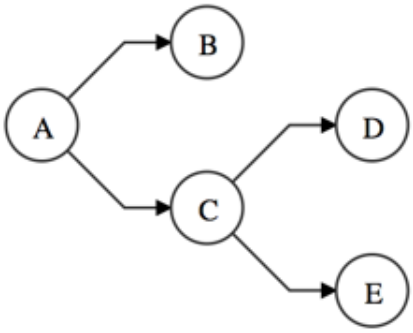


Abbildung 3.5: Ein einfacher Beispielgraph für die Traversierung. Quelle: [Dag18]

Knoten betreten wird dieser gespeichert, bis es zu einem Backtracking kommt. Erst dann wird dieser gespeicherte Knoten dem Pfad angehängt,also: $p = (B, D, E, C, A)$. Während Post-Order-Traversierung in Bäumen einige nützliche Eigenschaften hat (wie z.B. das implizite Sortieren von Zahlen) wird in dieser Arbeit nur auf Pre-Order zurückgegriffen.

Zusammenhangskomponenten Der in dieser Arbeit entwickelte Lösbarkeitsalgorithmus verwendet das im folgenden erklärte Konzept der Zusammenhangskomponenten: Die Menge der Knoten V lässt sich in sog. Zusammenhangskomponenten $V_1, \ldots, V_k \subseteq V$ unterteilen. Diese sind nach der folgenden Vorschrift definiert: Zwei Knoten $u, v \in V$ sind in der selben Zusammenhangskomponente falls es einen Pfad p von u nach v gibt:

$$u, v \in V_k \iff \exists p = (u, \ldots, v) \tag{3.3}$$

Als Konsequenz dieser Vorschrift gilt, dass jeder Knoten mit sich selbst in einer Zusammenhangskomponente ist. Das heißt:

$$\bigcup_{i=1}^{k} V_i = V \tag{3.4}$$

Weiter sind diese Komponenten disjunkt:

$$\forall 1 \leq i \leq k, 1 \leq j \leq k.\ i \neq j \Rightarrow V_i \cap V_j = \emptyset \tag{3.5}$$

Dies ist eine leicht einzusehende Konsequenz aus 3.3. Angenommen ein $v \in V_i$ und $v \in V_j$ mit $i \neq j$. Dann würde ein trivialer Pfad $p = (v, v)$ zwischen den beiden Zusammenhangskomponenten existieren und 3.3 wäre verletzt. Also muss $i = j$ gelten und 3.5 ist erfüllt.

Weitere Bibliotheken

- Lodash: An vielen Stellen im Code werden Grundfunktionen wie deep copies, Objekt-Gleichheit oder ähnliches benötigt. Lodash ist eine Bibliothek, die diese (und viele weitere) *Utility*-Funktionen bereitstellt.
- Electron: Für die native Version des Level-Editors wird Electron verwendet: Ein Wrapper um die Browser-Umgebung, der es ermöglicht Web-Applikationen nativ auszuführen.
- JSON-Source Map: Für das Anzeigen von Fehlern im JSON-Editor wird diese Bibliothek verwendet: Gegeben ein JSON-Objekt, wird für jedes Key-Value-Paar die exakte Zeile in dessen Serialiserung angegeben. Diese Informationen können dann in Zeilen im Editor umgerechnet werden.

4 Konzeptionierung der Prozessrepräsentation

Die folgenden Abschnitte beschreiben das dieser Arbeit zu Grunde liegende theoretische Modell von Prozessen. Aufbauend auf diesen Definitionen und der vorgestellten Terminologie werden dann Algorithmen entworfen und spezifiziert, die für die EXTRA-Erweiterung geforderte Funktionalität umsetzen. Namentlich:

- Die Übersetzung eines Prozesses in eine geeignete Repräsentation
- Das Unterteilen eines übersetzten Prozesses in Level (Leveling)
- Die Verifikation eines gegebenen Prozesses als Lösung eines solchen Levels

Abschließend werden diese dann mithilfe der im vorhergehenden Kapitel beschriebenen Technologien implementiert.

4.1 Übersetzung von Prozessen nach EXTRA

In dieser Sektion wird der in Abbildung 4.1 dargestellte BPMN-Prozess als durchgehendes Beispiel verwendet.

Notation Gegeben sei ein Prozess P. Dieser hat eine (graphische) Repräsentation. Dabei handelt es sich um eine der folgenden:

- P_B, die Repräsentation als BPMN-Diagramm
- P_U, die Repräsentation als UML-Diagramm
- P_G, die Repräsentation als generischen Graph

Im Rahmen dieser Arbeit werden Prozesse P_B und P_U stets in ihre generische Repräsentation P_G übersetzt und alle definierten Algorithmen darauf ausgeführt. An dieser Stelle ist zu beachten, dass die Repräsentationen für die betrachteten Elemente stets äquivalent sind. Eine Untersuchung ob dies auch im Allgemeinen (d.h für alle UML/BPMN - Elemente) gilt, könnte die Grundlage für weitere Forschung sein, geht aber im Rahmen dieser Arbeit zu weit. Stets gilt für einen generischen Graphen $P_G = G = (V,E)$.

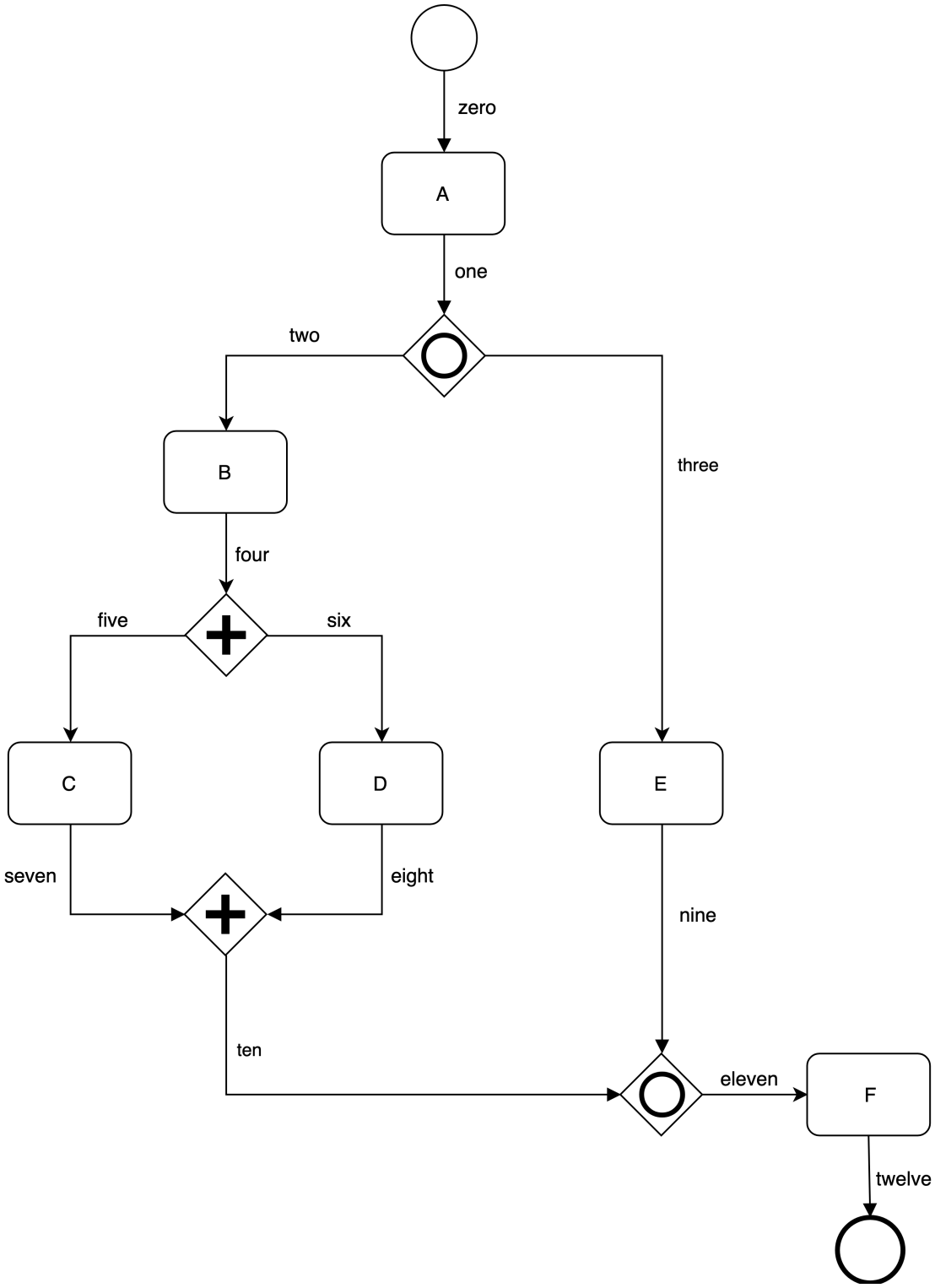


Abbildung 4.1: Ein Beispielprozess in BPMN.

Um während der Übersetzung des Prozesses nach EXTRA genau zwischen den Domänen des Prozesses und des EXTRA-Levels zu unterscheiden wird die folgende Notation für die Knoten der Graphen verwendet:

Hierbei gilt folgendes: Ein runder Knoten entspricht

- einer Aktivität, falls P_G aus P_U entstanden ist
- einer Aktion, falls P_G aus P_B enstanden ist

Ein eckiger Knoten entspricht dagegen einer Fabrik bzw. einem Marktplatz in einem EXTRA-Level.

Im generischen Prozess-Modellierungsformat entspricht er dem in 4.2 dargestellten Graphen.

Um Graphen bzw. Prozesse dieser Art als EXTRA-Level darstellen zu können, müssen sie (genauer: ihre Repräsentation) also übersetzt werden: Von P_B bzw. P_U nach P_G. Dieser Vorgang wird im Folgenden genauer beschrieben. Definiere die folgenden Funktionen:

Definition 4.1 *Ein gerichteter Graph G ist ein Tupel $G = (V,E)$, wobei V die Menge der Knoten bezeichnet und $E \subseteq V \times V$ die Menge der Kanten. In dieser Arbeit werden außerdem stets die folgenden Funktionen definiert: $type : V \to T$, wobei T die Menge der Knotentypen beschreibt: $T = \{linear, and, or, source, drain\}$ $product : E \to P$, wobei P die Menge aller Produkte beschreibt. Diese ist für jeden Prozess individuell definiert. Des Weiteren bezeichnen die Funktionen $d_+ : V \to \mathbb{N}_0, d_- : V \to \mathbb{N}_0$ den Eingangs- und Ausgangsgrad eines Knotens.*

Definition 4.2 *Die Funktion $d^p_- : V \to \mathbb{N}_0$ bezeichnet den **effektiven** Ausgangsgrad eines Knotens. Er bezeichnet die Anzahl der ausgehenden Produkte. Formell:*
$\forall v \in V.\ d^p_-(v) := |\{(v,w) \in E \mid \forall (v,u) \in E.\ w \neq u \Rightarrow product((v,w)) \neq product((v,u))\}|$

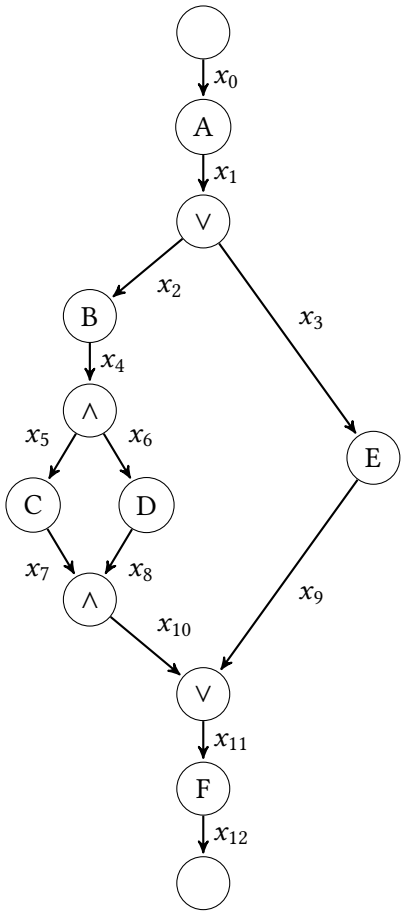


Abbildung 4.2: Der Beispielprozess in generischer Notation.

Die folgenden Beobachtungen bezüglich der Eigenschaften von EXTRA-Levels sind für die Konzeption der Übersetzung hilfreich. Jeder Knoten ist einer der beiden folgenden Kategorien zuzuordnen:

- **Fabrik** Eine Fabrik hat entweder keine eingehenden Kanten (sie ist eine *Quelle*), genau eine (sie ist eine *lineare* Fabrik) oder beliebig viele (Diese werden logisch verknüpft, das Ergebnis ist eine *and-* oder eine *or-* Fabrik). Formell:

$$d_+(v) = \begin{cases} 0 & , type(v) = source \\ 1 & , type(v) = linear \\ n & , \ sonst \end{cases}$$

  Sowohl Quellen als auch Fabriken haben beliebig viele ausgehende Kanten, jedoch **immer** einen effektiven Ausgangsgrad von 1.
- **Markt** Alle Prozesse, die in dieser Arbeit betrachtet werden haben genau einen Endknoten (Senke) und genau einen Startknoten (Quelle). Ein Endknoten hat einen beliebigen Eingangsgrad größer 0 und einen Ausgangsgrad gleich 0.

Für den effektiven Ausgangsgrad gilt folglich:

$$\forall v \in V : d_-^p(v) \leq d_-(v) \tag{4.1}$$

Wie oben beschrieben impliziert die EXTRA-Metapher von Fabriken, die Produkte produzieren oder weiterverarbeiten um sie schließlich an einem Markt an Kunden zu verkaufen, eine andere Semantik als von der ursprünglichen Repräsentation vorgesehen. Die Übersetzung des Graphen in ein entsprechendes EXTRA-Level wird nun ausgedrückt über eine Erweiterung der Wertemenge der *type*-Funktion um die Fabrik-Typen *linearfactory,andfactory* und *orfactory* sowie eine Transformation des Graphen mittels des nachfolgend beschriebenen Algorithmus.

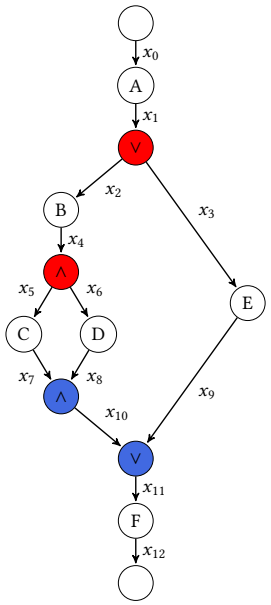


Abbildung 4.3: Die Knotenmengen R und C im Beispielsgraphen.

Zunächst wird der zu übersetzende Graph in die zwei Mengen R und C (sowie die Restmenge K) nach den folgenden Vorschriften unterteilt.

$$R := \{v \in V \mid d^p_-(v) > 1\} \tag{4.2}$$

$$C := \{v \in V \mid d_+ > 1\} \tag{4.3}$$

Für den Beispielsgraph ist dies dargestellt in Abbildung 4.3. Nun gilt es zwei Schritte durchzuführen: Zunächst, im **create**-Schritt, werden die Knoten der Menge C in Fabriken des entsprechenden Typs transformiert. Dieser ist sinnvollerweise definiert via:

$$\forall v \in C.\ type(v) := \begin{cases} andfactory & , type(v) = and \\ orfactory & , type(v) = or \end{cases} \tag{4.4}$$

Im selben Schritt wird allen Knoten im Komplement $K := V \setminus (R \cup C)$ ein neuer Typ zugeordnet:

$$\forall v \in C.\ type(v) := \begin{cases} market & , type(v) = drain \\ linearfactory & , type(v) = linear \vee type(v) = source \end{cases} \tag{4.5}$$

Der aktuelle Stand ist in Abbildung 4.4 dargestellt. Anschließend wird der zweite, der **split**-Schritt durchgeführt. Diese Routine läuft in $|R|$ Iterationen ab. In jeder Iteration wird jeweils der Knoten gewählt, der am weitesten von der Quelle entfernt ist: $r^\star := dijkstra(G)$. Dieser wird nun wie folgt bearbeitet. Zunächst werden sämtliche eingehenden und ausgehenden Kanten aus E entfernt (aber separat abgespeichert). Nun wird für jeden Nachfolger ein neuer Knoten v mit $type(v) := linearfactory$ dem Graphen hinzugefügt und mit dem entsprechenden Nachfolger sowie mit dem (eindeutigen) Vorgänger von $r^\star$ verbunden. Die Kantenbeschriftungen *product* sind eindeutig mithilfe der gespeicherten Kanten definiert (werden also übernommen). Entferne nun $r^\star$ endgültig aus V und R und beginne die nächste Iteration (sofern $R \neq \emptyset$). Das Ergebnis der ersten Iteration für den Beispielgraphen ist in Abbildung 4.5 dargestellt.

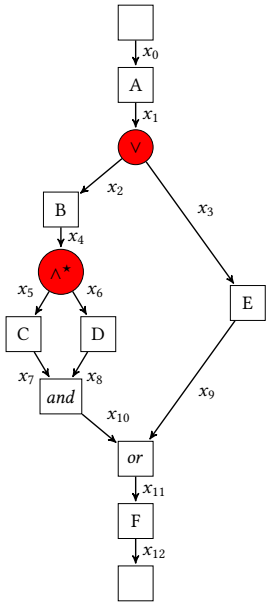


Abbildung 4.4: Der Beispielsgraph nach dem create-Schritt.

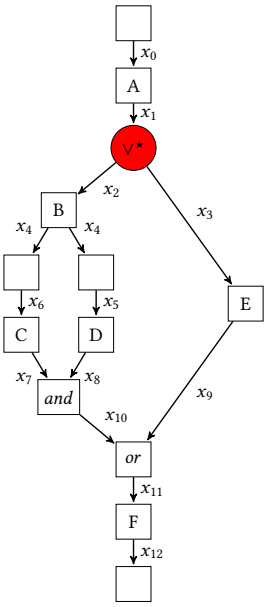


Abbildung 4.5: Der Beispielsgraph nach dem ersten split-Schritt.

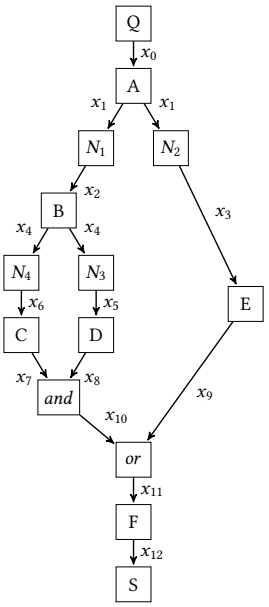


Abbildung 4.6: Der vollständige Graph des Beispielgraphen.

Beim dem in Abbildung 4.6 dargestellten Ergebnis des Split and Create-Dijkstra-Algorithmus handelt es sich um den **vollständigen** oder **ausgerollten** Graphen des Prozesses bzw. EXTRA-Levels:

Definition 4.3 *Gegeben sei ein Prozess (in beliebiger Repräsentation) P. Wird auf seiner generischen Repräsentation $P_G = (G,V)$ Algorithmus 1 ausgeführt, erhält man den zugehörigen vollständigen Graphen $G^\star = (V^\star,E^\star)$. Dieser enthält alle möglichen (optimalen) Pfade durch den Graphen. Er ist stets eine gültige Lösung für das zum Prozess gehörige EXTRA-Level.*

Die folgenden Aspekte gilt es hierbei zu beachten.

Optimalität Der vollständige Graph ist genau dann nicht optimal wenn einer der beiden folgenden Fälle eintritt:

- Der ursprüngliche Prozess ist degeneriert: Falls die ursprüngliche Prozessdefinition bereits „unnötige“ Knoten enthält oder auf andere Art degeneriert („falsch“ bzw. „redundant“) ist, ist die Übersetzung gemäß der obigen Definition auch nicht optimal.
- Es kommen optionale Pfade (d.h. eine *or*-Fabrik) vor. Da der vollständige Graph alle Pfade enthält, eine solche Fabrik aber nur eine benötigt, ist der Graph nicht optimal.

Data : Generisch modellierter Prozess $G = (V,E)$
Result : Vollständiger Graph $G^\star = (V^\star,E^\star)$
Bestimme Mengen R,C gemäß Vorschrift;
for $c \in C$ **do**
  if $type(c) == and$ **then**
    Setze $type(c) \coloneqq andfactory$;
  else
    Setze $type(c) \coloneqq orfactory$;
  end
end
for $r \in R$ **do**
  Bestimme $r^\star \coloneqq dijkstra(G)$;
  Definiere eindeutigen Vorgänger $r_p \in V$ von $r^\star$ mittels eingehender Kante $(v^p,r^\star) \in E$;
  Definiere Menge der Nachfolger $S \coloneqq \{v_0,\ldots,v_n\} \subseteq V$ mittels ausgehender Kanten $(r^\star,v_0),\ldots,(r^\star,v_n) \in E$;
  for $s \in S$ **do**
    Füge v mit $type(v) \coloneqq linearfactory$ hinzu: $V \coloneqq V \cup \{v\}$;
    Verbinde r_p mit v und v mit s: $E \coloneqq E \cup \{(r_p,v),\ (v,s)\}$;
  end
  Entferne $r^\star$ aus V: $V \coloneqq V \setminus \{r^\star\}$;
end
$V^\star \coloneqq V, E^\star \coloneqq E$

Algorithmus 1 : Der Split And Create Dijkstra-Algorithmus.

Wie in den folgenden Sektionen beschrieben, ist der vollständige Graph an zwei Stellen zentral: Dem Feststellen der Lösbarkeit einer Level-Instanz (Sektion 4.1.2) und beim Leveling-Algorithmus (Sektion 4.1.3).

4.1.1 Übersetzung nicht wohlgeformter Prozesse durch Switch And Create

Im Allgemeinen sind nicht wohlgeformte Prozesse (wie in Sektion 3.3 beschrieben) nicht äquivalent zu ihren wohlgeformten Entsprechungen. Diese Sektion führt aber trotzdem einen theoretischen Ansatz ein, wie mit diesen umgegangen werden kann: Es ist nicht klar inwiefern dieses Vorgehen generalisiert bzw. erweitert werden kann, es sollte deswegen also (nur) als Grundlage für eventuelle weitergehende Arbeiten gesehen werden. Dementsprechend ist es auch nicht als Teil des Editors implementiert.

Data : $G = (V,E),\ R,C \subseteq V$
Result : $v^\star \in V$ mit maximaler Entfernung zu Quelle
for $v \in V$ **do**
  $dist[v] \coloneqq \infty, prev[v] \coloneqq \perp;$
end
$dist[source] \coloneqq 0, Q \coloneqq V;$
while $Q \neq \emptyset$ **do**
  $u \coloneqq v \in Q \wedge v = arg \min_{x \in Q} dist[x];$
  $Q \coloneqq Q \setminus \{u\};$
  for *neighbor* v *of* u **do**
    $alt \coloneqq dist[u] + distance(u,v);$
    if $alt < dist[v]$ **then**
      $dist[v] \coloneqq alt;$
      $prev[v] \coloneqq u;$
    end
  end
end
$v^\star \coloneqq arg \max_{v \in R} dist[v];$

Algorithmus 2 : Die Implementierung des Dijkstra-Algorithmus.

Das Übersetzen eines nicht wohlgeformten Prozesses in ein EXTRA-Level ist **nicht** die Definition eines Graphisomorphismus. Dies ist einzusehen, wenn man die Knoten E und $\vee$ des in Abb. 4.7 dargestellten Graphen betrachtet. Einer der beiden muss auf eine *or*-Fabrik abgebildet werden. Es gilt folgendes: $\vee$ erhält nur ein Produkt als Eingabe und produziert zwei, kann also nicht zur *or*-Fabrik werden. Folglich muss E zur *or*-Fabrik werden. Dies geschieht durch einen **switch**, ein Vertauschen von $\vee$ und E, dargestellt in Abb. 4.8.

In einem zweiten Schritt, genannt **create**, gilt es nun den Graphen so zu transformieren, dass alle Knoten (mit der Quelle als möglicher Ausnahme) nur eine ausgehende Kante haben. Im Beispiel lässt sich das durch das Aufspalten von E in zwei (neue) Knoten E_1,E_2 erreichen:

Durch das Ausführen dieser Schritte erhält der Graph die charakteristische Topologie eines hierarchischen Baumes. Diese Eigenschaft wird vom Layout First Leveling-Algorithmus ausgenutzt. Der fertig transformierte Graph ist, analog zum wohlgeformten Fall, der **vollständige** oder **ausgerollte** Graph des Prozesses bzw. des Levels. Wird dieser nachgebaut stellt er eine gültige Lösung des Levels dar. Gemäß der EXTRA-Semantik ist diese aber im Allgemeinen nicht optimal. Da beim obigen Beispiel der Knoten $E_\vee$ nur eines der Produkte x_3,x_4 benötigt um x_5 herzustellen, kann man einen der Pfade $(A,E_1,B), (A,E_2,C)$ weglassen. Wichtig ist dieser ausgerollte Graph aber beim Prüfen der Lösbarkeit (Sektion 4.1.2).

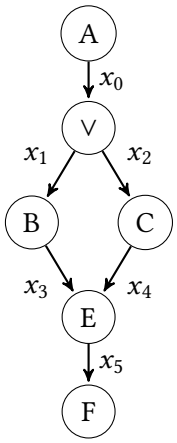


Abbildung 4.7: Ein Beispielprozess in generischer Repräsentation.

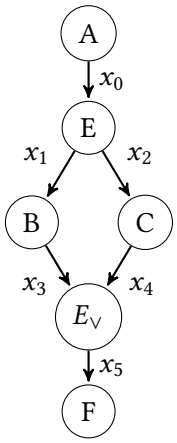


Abbildung 4.8: Der Beispielprozess nach dem switch-Schritt.

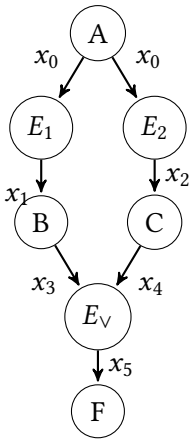


Abbildung 4.9: Der Beispielprozess nach dem create-Schritt.

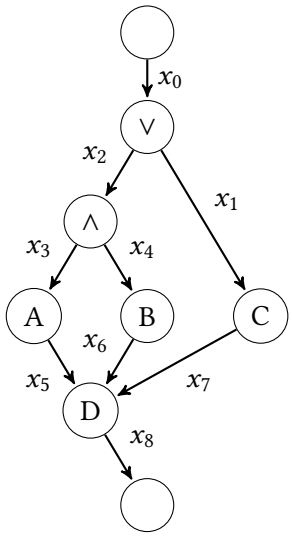


Abbildung 4.10: Der nicht wohlgeformte Beispielsgraph.

Das Ziel dieses Algorithmus ist es einen gegebenen Prozess in generischer Notation in den *vollständigen Graphen* eines EXTRA-Levels zu übersetzen. Er berechnet die Graphtransformation aus dem vorherigen Beispiel für den allgemeinen Fall. Abbildung 4.10 stellt einen solchen dar. Es gibt zwei Knoten ($\vee$, $\wedge$) die jeweils mit einem **switch** und **create** Schritt bearbeitet werden müssen. Im Gegensatz zum Beispiel aus dem vorherigen Abschnitt gilt es nun eventuelle Präzedenzen und Mehrdeutigkeiten zu behandeln. Diese werden nachfolgend betrachtet.

Definiere die Mengen R,$C \subseteq V$ gemäß folgender Vorschrift:

$$R := \{v \in V \mid d_-(v) > 1\}$$

$$C := \{v \in V \mid d_+(v) > 1\}$$

Aufgrund der Eigenschaften von G gilt:

$$R \cap C = \emptyset$$

Es gilt, eine bijektive Abbildung zwischen der Menge R der logischen Operationen und der Menge C der nichtlinearen Fabriken herzustellen. Dabei gilt folgendes:

$$|R| \geq |C|$$

Am Beispielsgraph liefern diese Definitionen: $R = \{\vee, \wedge\}$ sowie $C = \{D\}$

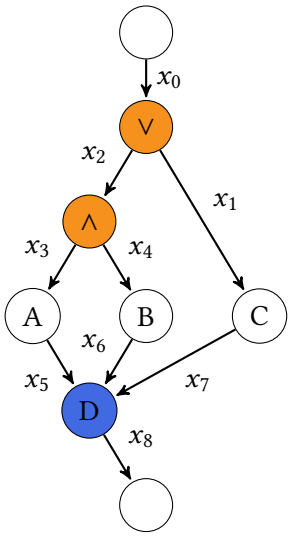


Abbildung 4.11: Die Knotenmenge R und C im nicht wohlgeformten Beispielsgraphen.

Nach Beendigung ist das Ergebnis wieder der vollständige Graph $G^\star$, dargestellt in Abbildung 4.12. Beachte, dass das Produkt zwischen der and- und der oder-Fabrik nicht programmatisch zu bestimmen ist.

4.1.2 Augment And Compress

Der Augment and Compress-Algorithmus überprüft die Lösbarkeit einer Level-Instanz. Diese besteht aus aus drei Komponenten: Dem vollständigen Graphen des unterliegenden Prozesses $G^\star$, dem aktuellen Graph G_L sowie den Meta-Informationen. G_L ist ursprünglich mittels des Leveling-Algorithmus aus Sektion 4.1.3 aus $G^\star$ entstanden und somit garantiert lösbar (im Sinne der unten stehenden Definition). Nun kann der Nutzer aber mittels des Editors die Levelinstanz verändern. Dabei bleibt $G^\star$ als Verifikationsgrundlage stets unverändert, $G_L = (V_L,E_L)$ und die Meta-Informationen können sich aber verändern. Insbesondere kann sich $aug(v)$ für ein $v \in V_L$ ändern. (Wenn dieser beispielsweise gelöscht wird). Der im folgenden beschriebene Algorithmus untersucht nun die Auswirkungen dieser Veränderung auf die Lösbarkeit.

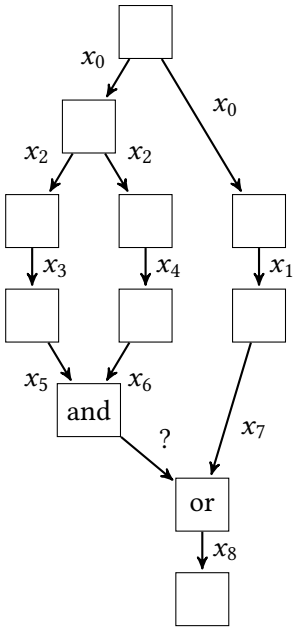


Abbildung 4.12: Der nichteindeutige vollständige Graph.

Data : Generisch modellierter Prozess $G = (V,E)$
Result : Vollständiger Graph $G^\star = (V^\star,E^\star)$
Bestimme Mengen R,C gemäß Vorschrift;
for $r \in R$ **do**
  Wähle $r^\star \in \{x \in R \mid x$ hat keine Kinder in $R\}$;
  Entferne eingehende Kanten von $r^\star$ aus E;
  Konstruiere Ebenen $0,\ldots,n-1$ mittels Breitensuche bis ein $v \in V$ in Ebene (n-1) in C liegt;
  Füge $v^\star$ vom Typ $type(r^\star)$ hinzu, verbinde mit allen Knoten aus Ebene $(n-2)$ sowie mit v;
  Entferne $r^\star$ aus V;
  if *Vorgänger von* $r^\star \in R$ **then**
    verbinde Knoten aus Ebene 0 mit Vorgänger von $r^\star$;
  else
    verbinde Knoten aus Ebene 0 mit (neuer) Quelle ;
  end
  $V^\star := V, E^\star := E$
end

Algorithmus 3 : Der Switch and Create-BFS-Algorithmus.

Da dieser Algorithmus die Meta-Informationen, also *Gamification*-Aspekte wie Zeitlimit etc., nicht betrachtet, werden im folgenden die verwendeten Begriffe und Techniken definiert.

Definition 4.4 *Ein EXTRA-Level L heißt* ***graphtheoretisch lösbar*** *wenn es im zugehörigen Graphen G_L einen Weg von einer Quelle zu der Senke gibt wobei alle eingehenden Kanten jedes* ***and****-Knoten im zugehörigen* ***vollständigen*** *Graphen $G_L^\star$ auch in G_L existieren. Für* ***or****-Knoten gibt es keine Einschränkungen.*

Der hier vorgestellte Algorithmus beruht auf zwei Konzepten. Das erste ist die **Meta-Augmentation**. Im EXTRA-Level gilt hat jeder Knoten im vollständigen Graphen eine der folgenden Eigenschaften:

- Bereits gebaut
- Baubar
- Nicht baubar

Um dies theoretisch zu modellieren wird in der nachfolgenden Sektion eine Meta-Augmentationsfunktion definiert.

Definition 4.5 *Die Funktion $aug : V \to State$ ordnet jedem Knoten im vollständigen Graphen $G^\star$ einen Status $s \in \{B_0, B_1, B_2\}$ zu. Hierbei gilt:*

$$
\begin{aligned}
aug(v) = B_0 &\iff \tau(v) \text{ gebaut} \\
aug(v) = B_1 &\iff \tau(v) \text{ baubar} \\
aug(v) = B_2 &\iff \tau(v) \text{ nicht baubar}
\end{aligned}
$$

Im Rahmen dieser Arbeit wird diese Funktion als Black Box angesehen.

Die Betrachtung als Black Box hat den Hintergrund, dass so die implementierungstechnischen Details (also die programmatische Analyse der JSON-Dateien) versteckt wird.

Der erste Schritt des Lösbarkeitsalgorithmus ist die nachfolgend erklärte **lineare Pfadkompression**. Hierbei wird (beginnend bei der Quelle) der gesamte Graph traversiert. Sei $v_0 \in V$. Es gibt drei Fälle:

1. $type(v_0) = drain$: Der Knoten ist die Senke, dies impliziert $d_-(v_0) = 0$, der Algorithmus ist fertig und terminiert.

2. $type(v_0) = linear \vee type(v_0) = source$. Der Knoten ist eine lineare Fabrik oder eine Quelle. Nun wird der Ausgangsgrad betrachtet. Ist $d_-(v_0) = 1$ ex. per Definition der eindeutige Nachfolger v_1. Nun wird ein Kompressionsschritt durchgeführt: v_0 und v_1 werden aus dem Knoten entfernt und als kompressierten Knoten $v_{0,1}$ wieder hinzugefügt. Sofern sie existieren, wird der neue Knoten mit dem Vorgänger von v_0 und dem Nachfolger von v_1 verbunden. Ist $d_-(v_0) > 1$ wird nichts durchgeführt.

3. $type(v_0) = and \vee type(v_0) = or$ Es wird nichts durchgeführt. Dies führt implizit zur Beendigung der Kompression des aktuellen Pfades.

Das Ergebnis diesen Schrittes ist der in Abbildung 4.13 dargestellte Graph. Auf diesem wird nun die Augmentation durchgeführt: Hierbei wird wieder der Graph von der Quelle aus traversiert. Jedem Knoten wird nun ein Status B_0,B_1,B_2 zugewiesen. Dies funktioniert wie folgt: Ist der Knoten nicht kompressiert, wird ihm der Wert $aug(v)$ gemäß Def. 3.5 zugewiesen. Sei der Knoten v kompressiert und hat die inneren Knoten $v_0, \ldots, v_n$. Sei $\mathbb{I}$ die Indexmenge der Menge $\mathbb{B} := \bigcup_{j=0}^{n} \{B_j\}$ wobei $B_j = aug(v_j)$. $\mathbb{B}$ ist also die Menge der Status der inneren Knoten. Somit kann $\mathbb{I}$ nur die Elemente 1,2,3 enthalten. Dann ist $aug(v) = B_k$ wobei $k := \max \mathbb{I}$. Diese Definition stellt sicher, dass falls ein Knoten auf dem Pfad nicht baubar ist, der ganze Pfad nicht baubar ist.

Ist diese Augmentierung abgeschlossen, wird der Graph wie folgt transformiert. Zunächst werden die Eingangsgrade d_+^0 aller Knoten v mit $type(v) = and$ berechnet und abgespeichert. Danach werden alle Knoten mit $aug(v) = B_2$ mitsamt ihren adjazenten Kanten aus dem Graph entfernt. Nun werden wieder die Eingangsgrade d_+^1 der and-Knoten berechnet. Gilt $d_+^0(v) > d_+^1(v)$, so ist ein Vorgänger entfernt worden und v wird mitsamt der adjazenten Kanten entfernt. Danach werden alle (verbliebenen) Kanten $e \in E$ umgedreht, also $(v,w) \mapsto (w,v)$. Es gilt zu beachten, dass der so entstandene Graph durchaus leer oder nicht mehr zusammenhängend sein kann. Nun wird der Graph noch einmal traversiert und alle besuchten Knoten als Pfad $p = (v_0, \ldots, v_n)$ abgespeichert. Nun kann entschieden werden ob das Level (graphtheoretisch)lösbar ist. Es gilt: Das Level L ist genau dann graphtheoretisch lösbar, wenn der oben konstruierte Pfad p mindestens eine Quelle enthält, also gilt: $v_i \in p \wedge type(v_i) = source$.

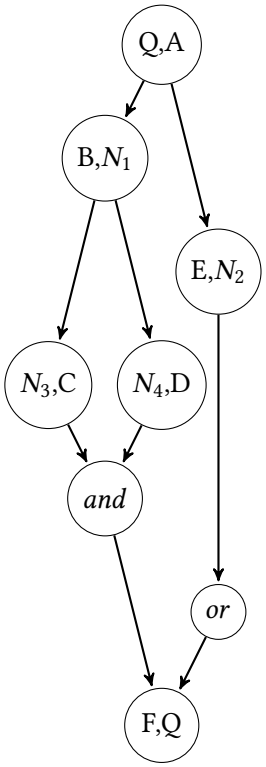


Abbildung 4.13: Der Beispielprozess nach der Pfadkompression.

Die Implementierung der oben beschriebenen Pfadkompression erfolgt über den im Folgenden beschriebenen Prozess, der mittels der Berechnung von *Zusammenhangskomponenten* basiert. Zunächst wird über die Knoten des Graphen iteriert. Für jeden betrachteten Knoten $v \in V$ wird die Menge der Kanten E wie folgt verändert:

- Ist $d_-(v) > 1$, entferne alle ausgehenden Kanten
- Ist $d_+(v) > 1$, entferne alle eingehenden Kanten
- Ist $type(v) = or \vee type(v) = and$, entferne alle eingehenden und ausgehenden Kanten

Hierbei ist zu beachten, dass vor alle Kanten vor dem Entfernen gespeichert werden müssen (um die entstehenden Zusammenhangskomponenten wieder zu verbinden). Nach Beendigung dieses Prozesses sieht der Beispielgraph aus wie in Abbildung 4.14 Fasst man nun die Knoten der einzelnen Zusammenhangskomponenten zusammen erhält man die kompressierten Pfade aus der Abbildung. Abschließend werden die entfernten (roten) Kanten wieder sinnvoll hinzu. Seien $G = (V,E)$ und $G' = (V',E')$ der Original-Graph bzw. der kompressierte Graph. Dann füge die Kanten wie folgt hinzu:

$$E' := E' \cup \{(v',w')\}, \text{ falls } (v,w) \in E \wedge ((v = v' \vee v \in v') \vee (w = w' \vee w \in w'))$$

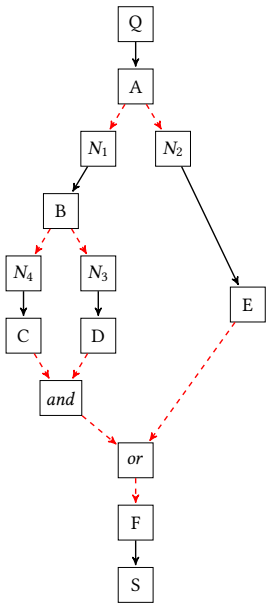


Abbildung 4.14: Der Beispielprozess nach der Entfernung der Kanten.

4.1.3 Layout First Leveling

Der in dieser Sektion vorgestellte Algorithmus wurde eigenständig entworfen um aus einem gegebenen vollständigen Graphen so Subgraphen zu extrahieren, dass diese in EXTRA-Level übersetzt werden können. Ähnliche Ansätze finden sich in der Literatur z.B. in [Bac+81], [BK02].

Definition 4.6 *Leveling bezeichnet das Aufteilen eines Prozesses in verschiedene Level. Diese unterscheiden sich v.a. in der Anzahl der zu platzierenden Fabriken. Aus graphentheoretischer Sicht kann Leveling als das Aufteilen des vollständigen Graphen $G^\star$ in (Sub-)Graphen $G_0, \ldots G_k$ beschrieben werden. Für einen solchen Subgraphen $G_i = (V_i, E_i)$ gilt stets $E_i \subseteq E^\star$. Die Teilmengenbeziehung der Knoten $V_i \subseteq V^\star$ gilt allerdings nicht. Es werden zwar **nie** Knoten hinzugefügt ($|V_i| \leq |V^\star|$), allerdings kann sich der Typ type(v)von Knoten ändern.*

Der in dieser Arbeit verwendete Leveling-Algorithmus bezieht sich auf die Topologie des vollständigen Graphen. Der Algorithmus basiert auf der Beobachtung, dass es sich bei dem vollständigen Graphen $G^\star$ stets um einen hierarchischen Baum handelt. Präziser: Der vollständige Graph $G^\star$ von Prozessen, die hier betrachtet werden, ist stets ein gerichteter, azyklischer Graph (DAG).

Im Rahmen dieser Arbeit (d.h. bezüglich zu EXTRA) lässt sich das Layout-Problem eines DAGs $G = (V, E)$ als das Einbetten von $n = |V|$ Knoten in ein Gatter B von beliebiger, aber fester Größe $X \times Y$ beschreiben (vgl. Abb. 4.15) Die Lösung dieses Problems ist im Allgemeinen nicht eindeutig.

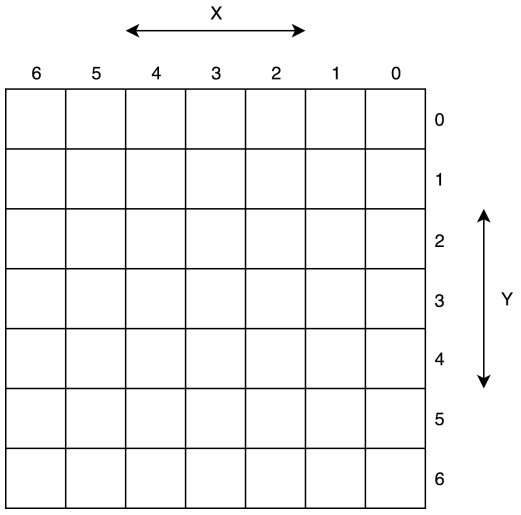


Abbildung 4.15: Die Einbettungsumgebung B: Ein (X,Y)-Gatter.

Analog zur Augmentationsfunktion in 4.1.2 wird die Einbettungsfunktion $pos : V \longrightarrow \mathbb{N}_0 \times \mathbb{N}_0$ als Black Box betrachtet. Sie erfüllt jedoch folgende Eigenschaften:

- *pos* ist injektiv, es wird also jedem Knoten eine eindeutige Position zugeordnet:

$$\forall v,w \in V.\ pos(v) = pos(w) \Longrightarrow v = w$$

- Jeder eingebettete Knoten $pos(v) = (x,y) \in \mathbb{N}^2$ hat keine direkten Nachbarn

$$\forall v,w.\ \|pos(v) - pos(w)\|_2 > 1$$

Diese beiden Eigenschaften (keine Kollisionen, Padding) ermöglichen den hier entworfenen Leveling-Algorithmus. Die Knoten des eingebetteten Graphen wird anhand der y-Komponente in Ebenen eingeteilt:

$$L_i = \{(x,y) \mid y = i\}$$

Hierbei gilt es mehrere Aspekte zu beachten:

- Die Kanten des Graphen sind zwar für die Einbettung relevant, für das Leveling werden sie aber nicht weiter verwendet.
- Theoretisch könnte die obige Vorschrift zu n verschiedenen Ebenen führen (wenn nämlich jeder Knoten eine eigene y-Koordinate bekommt) In der Implementierung (durch die *Dagre*-Bibliothek) ist dies aber nicht der Fall. In der theoretischen Modellierung könnte dies durch die folgende Vorschrift ausgedrückt werden: Die Level werden per Abstand zu der y-Koordinate (für einen konstanten Abstand c) definiert.

$$L_i = \{(x,y) \mid |y - i| < c\}$$

Mit der obigen Notation lassen sich die Subgraphen $G_0, \ldots, G_k$ nun also wie folgt definieren. Für die Menge der Knoten V_i des i-ten Subgraphen gilt:

$$V_i = \bigcup_{j=0}^{i} L_j$$

Die Kanten werden entsprechend aus dem vollständigen Graphen rekonstruiert:

$$E_i = \{(v,w) \in E^\star \mid v \in V_i \wedge w \in V_i\}$$

Die obigen Definitionen induzieren die Kette $V_0 \subseteq V_1, \subseteq \ldots \subseteq V_k$. In der hier erarbeiteten Implementierung gilt dies jedoch streng genommen nicht: Zwei Details gilt es noch zu diskutieren.

Wie in Abb. 4.16 dargestellt entspricht die Ebene i nicht dem Graphen G_i sondern G_{j+1} da ein Level stets mindestens zwei Knoten benötigt: Mindestens. einer davon muss die Quelle sein. Damit einhergehend ist folgende Vorschrift, die für die korrekte Übersetzung des Subgraphen als Level nötig ist: Setze den Typ der *obersten* Ebene (also $v \in L_i$ für einen Subgraphen G_i) auf den Quellen-Typ: $type(v) := source$. Beachte dass dies wohldefiniert ist, da sich die Vorgänger einer nichtlinearen Fabrik stets in der nächsthöheren Ebene der hierarchischen Topologie des vollständigen Graphen befinden.

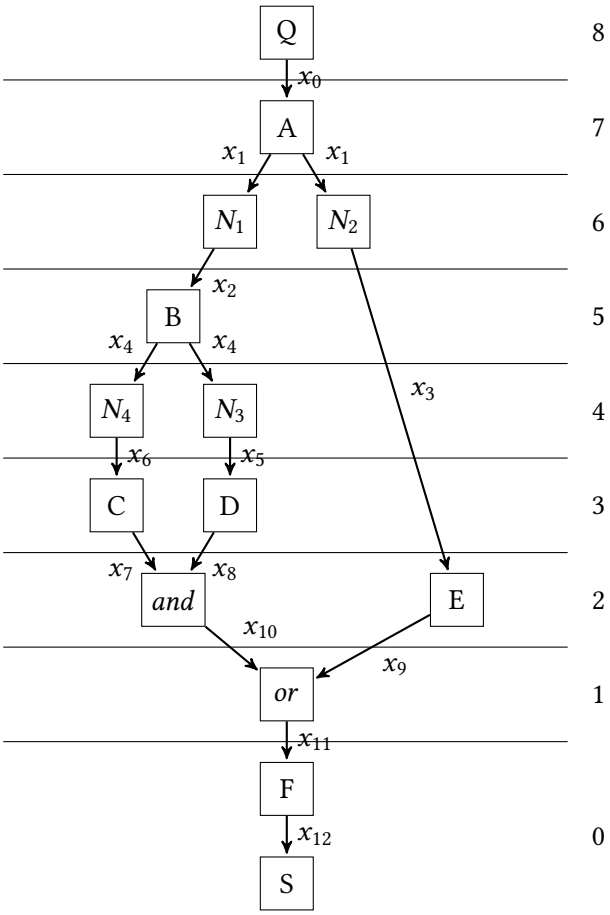


Abbildung 4.16: Die Unterteilung des Beispielgraphen in Ebenen.

4.2 Heuristiken zur Generierung von Meta-Daten

Der hier implementierte Level-Editor erhält Prozesse in der oben beschriebenen (graphischen) Form. Diese enthalten nicht alle Informationen, die EXTRA benötigt: Die Meta-Daten zur Gamification. Für das Generieren dieser Daten (soweit möglich) werden Heuristiken und ein Randomisierungs-System verwendet.

Die im Rahmen dieser Arbeit implementierten Heuristiken verwenden die folgenden Kennzahlen des vollständigen Graphen $G^\star = (V^\star, E^\star)$ als Grundlage: $n := |V|$ und $m := |E|$. Sie werden verwendet um Formeln für die folgenden Aspekte des Levels zu generieren:

- **Workforce**: Die Anzahl der Arbeitskräfte, die der Nutzer den Fabriken zuweisen kann.
- **Reputation**: Das Zeitlimit bzw. die dem Nutzer zur Verfügung stehende Energie bis das Level als verloren gilt.
- **Kunden**: Die „Gegner“, die der Nutzer zufrieden stellen muss. Sie erscheinen an festgelegten Zeitpunkten und müssen durch das Liefern bestimmter Produkte zufriedengestellt werden.
- **Routen**: Der Editor erlaubt das manuelle Platzieren von Routen um Strukturen zu verbinden. Im Regelfall ist das jedoch die Aufgabe des Nutzers, entsprechend werden ausreichend platzierbare Routen generiert.

Die konkreten Formeln für Workforce und Reputation sind:

$$\text{workforce} = 2(n + m)$$

$$\text{reputation} = 20n$$

Die Generierung von Kunden geschieht wie folgt: Die Routen-Generierung ist dagegen wieder recht simpel: Es werden stets $5n$ Routen vom Type „street“ hinzugefügt, hinzu kommen mit jeweils einer Wahrscheinlichkeit von 50% $3n$ Routen vom Typ „water“ bzw. „rail“.

Data : Vollständiger Graph $G^\star = (V^\star, E^\star)$
Result : Liste von Kunden C
Setze $C \coloneqq []$;
Bestimme $D \coloneqq \{product((v,w)) \in E^\star \mid type(w) = drain\}$;
Bestimme $r \coloneqq reputation = 20n$;
Setze Schrittweite $s \coloneqq \lceil \frac{\lfloor \frac{r}{5} \rfloor}{10} \rceil \cdot 10$;
for $d \in D$ **do**
    Sei c ein Kunde;
    Setze $c.product \coloneqq d$;
    Setze $c.quantity \coloneqq \max\{5, \lceil \frac{n}{2} \rceil\}$;
    Setze $c.power \coloneqq step$;
    $C.append(c)$;
    Setze $i \coloneqq r$;
    Setze $Cy \coloneqq []$;
    while $i > 0$ **do**
        Sei cy ein Zyklus;
        Setze $cy.reward \coloneqq i$;
        Wähle zufällig ein $x \in \{1,2\}$;
        Setze $cy.duration \coloneqq x$;
        $Cy.append(cy)$;
        Setze $i \coloneqq i - s$;
    end
    Setze $c.cycles \coloneqq Cy$;
end

Algorithmus 4 : Generierung von Kunden

Randomisierung Der Begriff der Randomisierung beschreibt im Rahmen dieser Arbeit die Vergabe von Namen für Knoten (= Fabriken/Märkte) und Kanten (= Produkte) sowie das Zuweisen von Icons für Produkte (die dann im EXTRA-Level sichtbar sind). Die möglichen Werte dieser Randomisierung werden über ein *Szenario* gesteuert. Dabei handelt es sich um ein unabhängiges, austauschbares Paket, das aus zwei Bestandteilen besteht:

- **Icons**: Ein Ordner, der eine beliebige Anzahl von Icons im svg-Format enthält. Diese werden einfach über ihren Datei-Namen identifiziert.
- **Szenario-Datei**: Eine JSON-Datei, die die Details der Wertevergabe spezifiziert. Jeder Knoten-Typ (Quelle, Senke, linear, and, or, Markt) erhält eine Liste von Namen und Input/Output-Produkten. Jedem dieser Produkte wird ein Icon (per Datei-Name) zugewiesen.

4.3 Verifikation einer EXTRA-Levelinstanz

Die Verifikation einer gegebenen Levelinstanz findet auf zwei konzeptionellen Ebenen statt: Der Lösbarkeitsprüfung im Kontext des vollständigen Graphen, beschrieben in Sektion 4.1.2 und der hier beschriebenen Untersuchung der Semantik der Leveldefinition.

Wie in Sektion 3.1 beschrieben, bestehen EXTRA-Level aus einer JSON-Datei. Alle Änderungen im Editor entsprechen der Modifikation dieser Datei. Deren Validität wird über das JSON-Schema L_E überprüft. Dieses wird nachfolgend spezifiziert. Hierbei ist zu beachten, dass dies nicht dem Sicherstellen eines korrekten (syntaktischen) Aufbaus dient, sondern die Semantik beschreibt. Für die Grammatik von JSON siehe [Int17]. Eigenschaften von Objekten, die optional sind, sind mit einem $\star$ gekennzeichnet. Mit den Definitionen in den Tabellen 4.1, 4.2 kann ein EXTRA-Level wie in Tabelle 4.3 spezifiziert werden.

Element		**Spezifikation**
Name	Datentyp	*String*
	Mindestlänge	1
Amount	Datentyp	*Integer*
	Mindestwert	0
Product	Datentyp	*Objekt*
	Eigenschaften	
		id : *Name*
		name : *Name*
		category : *Name*
		icon : *Name*
Cycle	Datentyp	*Objekt*
	Eigenschaften	
		duration : *Amount*
		reward : *Amount*
Input	Datentyp	*Objekt*
	Eigenschaften	
		type : *Name*
		input : *Array* : *Name*
Configuration	Datentyp	*Objekt*
	Eigenschaften	
		⋆ input : *Name* | *Input*
		name : *Name*
		output : *Name*
Customer	Datentyp	*Objekt*
	Eigenschaften	
		name : *Name*
		product : *Name*
		quantity : *Amount*
		arrival : *Amount*
		power : *Amount*
		cycle : *Amount*
		cycles : *Array* : *Cycle*

Tabelle 4.1: Die EXTRA-Level-Spezifikation (1).

Element		**Spezifikation**
Building	Datentyp	*Objekt*
	Eigenschaften	
		type : *Name*
		name : *Name*
		workforce : *Amount*
		products : *Array* : *Name*
		configurations : *Array* : *Configuration*
Route	Datentyp	*Objekt*
	Eigenschaften	
		type : *Name*
		routeType : *Name*
		⋆ ports : *Array* : *Name*
Meta	Datentyp	*Objekt*
	Eigenschaften	
		title : *Name*
		description : *Name*
Structure	Datentyp	*Objekt*
	Eigenschaften	
		quantity : *Amount*
		structure : *Building*

Tabelle 4.2: Die EXTRA-Level-Spezifikation (2).

Element		Spezifikation
Level	Datentyp	*Objekt*
	Eigenschaften	
		grid : *Array* : (*Array* : *Building* | *Route* | *null*)
		structures : *Array* : *Structure*
		products : *Array* : *Name*
		allProducts : *Array* : *Product*
		customers : *Array* : *Customer*
		workforce : *Amount*
		reputation : *Amount*
		meta : *Meta*

Tabelle 4.3: Die EXTRA-Level-Spezifikation (3).

5 Implementierung der EXTRA-Erweiterung

Der praktische Teil dieser Arbeit besteht aus zwei Software-Artefakten: Dem EXTRA-Level-Editor und dem erweiterten Hauptspiel. In diesem Kapitel werden diese genauer beschrieben und die unterliegenden Design-Entscheidungen beschrieben.

5.1 Der EXTRA-Leveleditor: Eine Übersicht

EXTRA ist mithilfe des React-Frameworks entwickelt worden. Aus den selben Gründen der Plattformunahängigkeit handelt es sich auch bei dem EXTRA-Leveleditor um eine React-Webapplikation. Diese kann direkt im Browser oder nativ via eines Electron-Wrappers ausgeführt werden. Nach dem Start der Applikation sieht der Nutzer das in Abbildung 5.1 dargestellte Hauptmenü. Hier kann der Nutzer drei Bereiche der Anwendung auswählen:

- **Level-Editor**: Ohne weitere Einstellungen kann der Nutzer hier grafisch oder direkt im unterliegenden JSON-Code Level erstellen
- **Upload-Portal**: Hier kann der Nutzer unterstützte Dateien hochladen. Aktuell sind dies drei: (1) *.bpmn*-Dateien, die BPMN-Prozesse definieren. (2) *.xmi*-Dateien, die UML-Prozesse definieren oder (3) *.json*-Dateien, die EXTRA-Level spezifizieren.
- **BPMN-Editor**: Hier kann der Nutzer BPMN-Prozesse erstellen, editieren und Leveling bzw. Randomisierung initiieren.

Datei-Upload Das Upload-Portal, dargsetellt in Abb. 5.2 lässt sich horizontal in zwei Bereiche gliedern: Der obere integriert die *FilePond*-Bibliothek und ist für das Initiieren und Verarbeiten des eigentlichen Datei-Uploads zuständig. Der untere Teil reagiert auf den hochgeladenen Dateityp und zeigt dem Nutzer den erkannten Dateityp sowie Buttons wie der Editor die Datei weiterverarbeiten kann. Bei BPMN zum Beispiel kann der Prozess entweder bearbeitet oder direkt „gamifiziert“ werden.

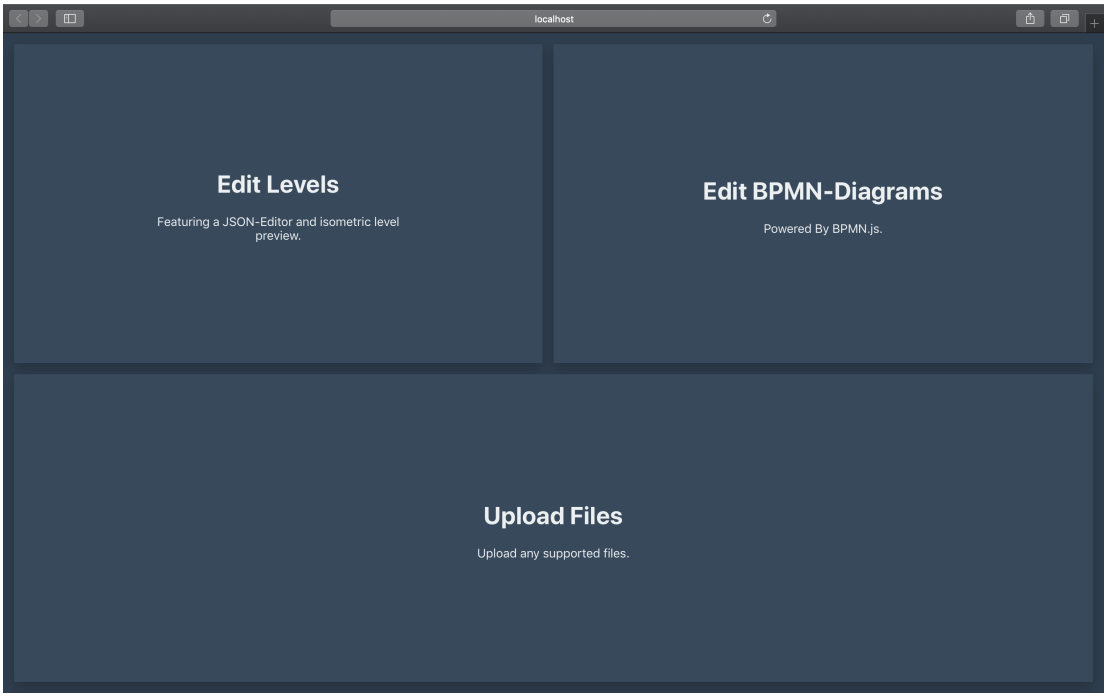


Abbildung 5.1: Das Hauptmenü des Editors.

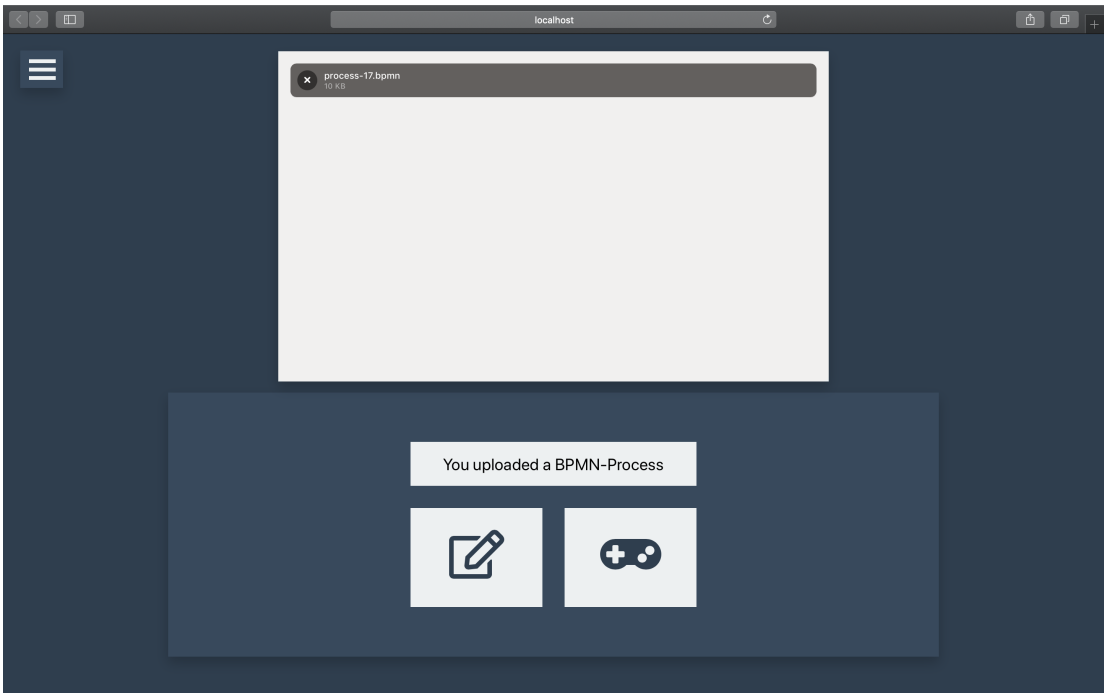


Abbildung 5.2: Das Upload-Portal des Editors.

BPMN-Editor Wieder ist das UI horizontal geteilt: Oben ist die Oberfläche von *BPMN.JS*. Links sind die Elemente, die der Nutzer dem aktuellen Diagramm hinzufügen kann. Es gilt zu beachten, dass **nicht** alle dieser Elemente von EXTRA bzw. dem Leveleditor unterstützt werden. Dieser BPMN.JS-Workspace erlaubt außerdem das Scrollen und Zoomen der Darstellung des Graphen. Der untere Teil des UIs enthält drei Buttons: Diese erlauben

- Das Herunterladen des aktuell dargestellten Prozesses als *.bpmn*-Datei
- Das Übersetzen des BPMN-Prozesses in seine generische Repräsentation sowie implizit den zugehörigen vollständigen Graphen. Nach diesem Prozess wird automatisch der Leveling-Algorithmus durchgeführt

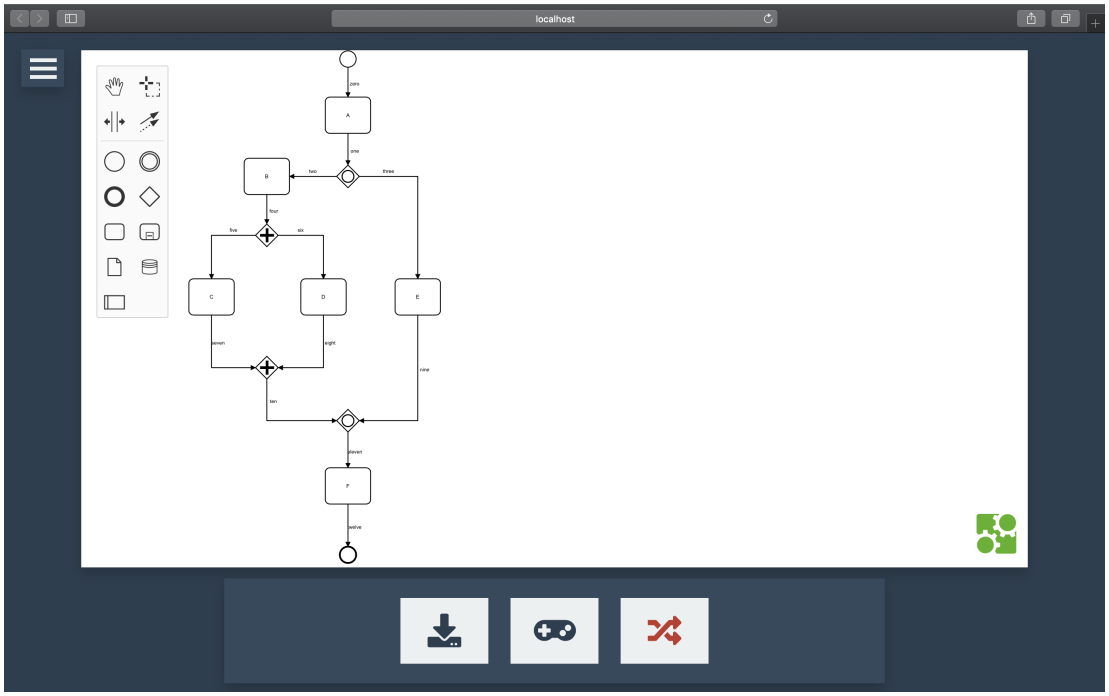


Abbildung 5.3: Der integrierte BPMN-Editor.

Level-Overview Die Ergebnisse des Leveling werden in Form eines 3-spaltigen Gitters von Karten repräsentiert, dargestellt in Abbildung 5.4. Jede dieser Karten repräsentiert ein Level (bzw. eine Schwierigkeitsstufe) und hat eine eingebaute, miniaturisierte Preview vom Level. Der Nutzer kann entweder ein Level (per Klick auf das Icon links unten) direkt herunterladen, alle Level herunterladen oder per Klick auf die Karte das Level im Editor öffnen.

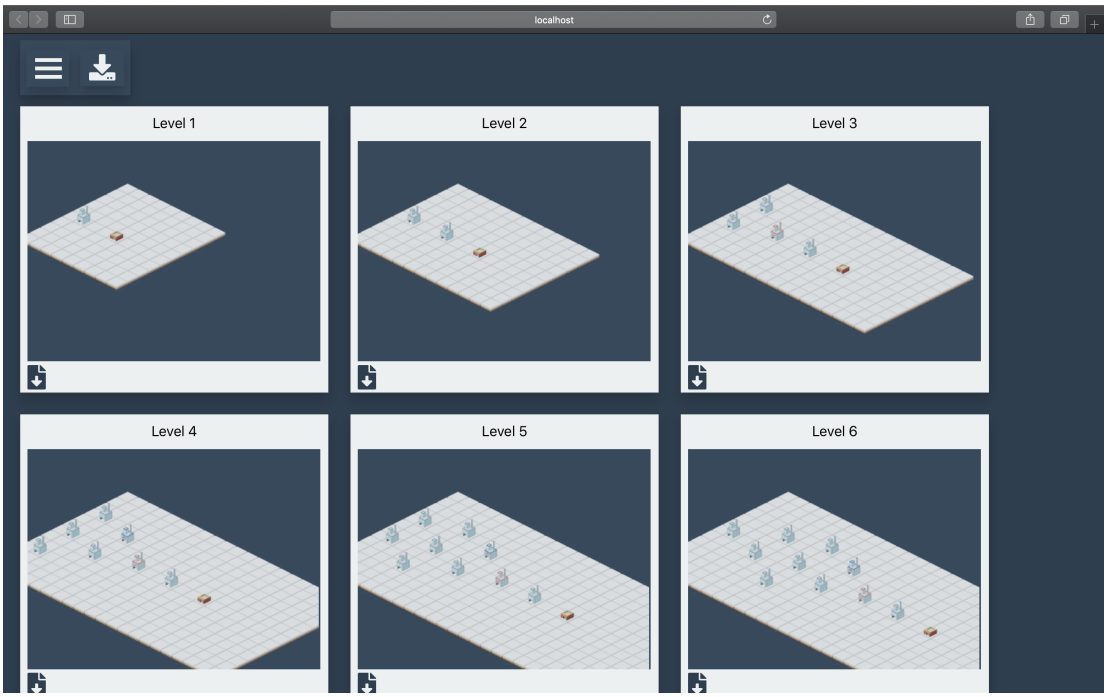


Abbildung 5.4: Die Level-Übersicht des Editors.

Der Level-Editor Der komplexeste Teil der Applikation ist das eigentliche Editor-Interface, dargestellt in Abb. 5.5. Es gibt zwei Display-Modi: Ein *split*-View, indem links der Code und rechts das GUI dargestellt wird und ein vollst. GUI-View: Verkleinert man den Code-Editor durch Ziehen des Seperators wird er ab einer bestimmten Grenze ganz ausgeblendet und das GUI größer angezeigt.

Der rechte Teil des GUIs kann wieder horizontal geteilt werden. Oben ist eine laufende Phaser-Instanz zu sehen, die gleichzeitig als Live-Preview und als Teil des Editors genutzt wird. Hier können per Drag And Drop Gebäude platziert, Straßen „gezeichnet“, die Größe des Spielfelds verändert und Objekte ausgewählt werden. Interaktionen dieser Art haben eine Änderung des unteren Teils des GUIs zur Folge: Hier werden eine Reihe von GUIs für alle möglichen Daten des Levels dargestellt. Dazu zählen u.a.

- Konfigurationen von Gebäuden
- Daten von Produkten, Gegnern
- Einstellungen für Arbeitskräfte, Zeitlimit etc.

Ist gerade kein Feld als aktiv ausgewählt stellt das untere GUI vier Buttons dar: Diese ermöglichen (von links nach rechts):

- Das Öffnen der Graph-Ansicht
- Das Öffnen der Level-Übersicht (für Meta-Daten wie Arbeitskräfte etc.)
- Das Öffnen des Bau-Interfaces
- Das Öffnen der Lösbarkeits-Übersicht

Ein weiterer Aspekt des GUIs im nachfolgenden genauer hervorgehoben werden:

Der Level-Graph Dieser Teil der Applikation kann als sekundäre Level-Ansicht betrachtet werden. Falls das Level auf einem definierten Prozess basiert (d.h. falls es aus einer hochgeladenen Datei per Leveling bzw. Gamification entstanden ist) kann der Nutzer sich per Knopfdruck den oben bereits diskutierten vollständigen Graphen anzeigen lassen. Dieser liefert nicht nur eine theoretische/konzeptionelle systematische Übersicht, an der sich der Nutzer beim Verändern des Levels orientieren kann, er wird dabei auch programmatisch unterstützt. Jeder Knoten im interaktiv dargestellten Graphen hat eine von drei Farben, entsprechend seinem Status im Sinne von Sektion 4.1.2. Platzierte Fabriken erscheinen grün, platzierbare (also im *structures*-Array definierte) Knoten erscheinen blau, nicht baubare Knoten erscheinen rot. Diese Informationen werden in Echtzeit aktualisiert. Ähnlich wie bei der Interaktion mit der Phaser-Level-Preview kann der Nutzer auch hier einen Knoten auswählen und dessen Informationen verändern. Ist dieser grün, erscheint das gleiche UI wie beim Auswählen des entsprechenden Gebäudes. Ist dieser allerdings nicht platziert erscheint ein UI, dass das automatische Platzieren eines „versteckten“ oder „gelöschten“ Knotens ermöglicht. (Hierbei sei angemerkt, dass dieser Knoten hierbei stets an seinem vom Layout-Algorithmus ursprünglich bestimmten Koordinate platziert wird, dies geschieht also nicht adaptiv.)

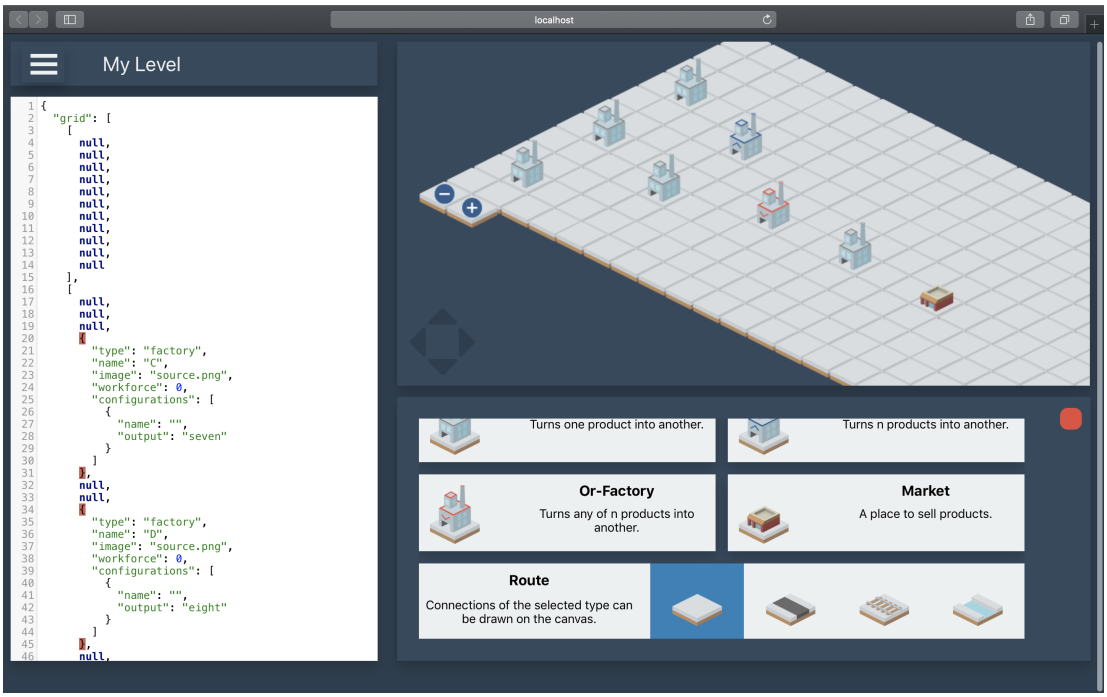


Abbildung 5.5: Das Editieren von Levels

5.2 Architektur des Editors

Während der vorherige Abschnitt die nutzerorientierten Features des Editors darstellt, erläutert diese Sektion dessen unterliegenden softwaretechnischen Aspekte sowie den konzeptionellen und objektorientierten Aufbau. Abbildung 5.6 gibt einen groben Überblick über den Aufbau des Editors, dieser ist hierarchisch. Für den Nutzer sind (hauptsächlich) drei Komponenten sichtbar:

- **BPMN-Editor**: Hier kann der Nutzer BPMN-Dateien erstellen, selbst hochgeladene Prozesse anschauen und editieren sowie die Randomisierung und Leveling starten.
- **Upload-Portal**: Hier kann der Nutzer unterstützte Dateien hochladen und kontextabhängige Aktionen durchführen. (Insb. das Öffnen des Dateiinhaltes in den anderen Komponenten).
- **Level-Editor**: Hier kann der Nutzer EXTRA-Level erstellen (direkt im Code oder in einer grafischen Vorschau).

Im Workflow sind diese durch das Hautptmenü (Mode Selector) sowie ein kontextbasiertes Dropdown-Menü verbunden. Zu den nicht bzw. nur indirekt sichtbaren Komponenten zählen:

- **BPMN.JS**: Diese Bibliothek stellt eine Umgebung für das Anzeigen und Bearbeiten von Prozessen in BPMN-Form bereit und wird in der BPMN-Editor-Komponente verwendet.
- **CodeMirror**: Diese Bibliothek stellt einen nativen Texteditor für das Bearbeiten von Code in diversen Sprachen bereit: Hier wird das JS-Paket verwendet. Weiter werden hier semantische Fehler (im Sinne der Level-Spezifikation in Sektion 4.3) angezeigt.
- **Graph**: Wie oben ausführlich diskutiert wird ein Prozess vom Editor stets als Graph betrachtet. Dies erlaubt das Ausführen von Algorithmen und das Darstellen der Musterlösung als vollständigen Graph im Editor.
- **Game**: Die Phaser-Engine stellt Level sowohl im Editor als auch im Hauptspiel dar.
- **Level-Overview**: Mithilfe eines Strategie-Musters (Abbildung 5.9) interagiert diese Vorschau mit dem Rest der Editor-Umgebung.
- **Element-View**: Per Klick auf ein Element in der Level-Vorschau markiert der Nutzer dieses als aktiv: Kontextabhängig stellt diese Komponente dann die entsprechenden Details dar.

Die Programmlogik im Hintergrund lässt sich weiter in drei Pakete teilen:

- **Heuristiken**: Der Editor hat eingebaute Heuristiken für die Bestimmung von Meta-Informationen, die im Rahmen der Gamification nützlich sind (wie Arbeitskräfte, das Zeitlimit, Details der Gegner etc.). Weiter kann der Editor auf Wunsch nicht definierte Namen eines hochgeladenen Prozesses randomisiert füllen, dazu kann der Nutzer vor dem Start der Applikation ein selbst definiertes Szenario einbinden, dies wird von diesem Paket zur Randomisierung (d.h. für Namen und Icons) verwendet.
- **Validierung**: Die Verifikation und Validierung eines gegeben Levels umfasst zwei Aspekte: Die Validierung per JSON-Schema und die Lösbarkeitsprüfung per Augment+Compress-Algorithmus.
- **Leveling**: Dieses Paket implementiert den oben beschriebenen Layout-First-Leveling-Algorithmus. Dieser teilt den vollständigen Graph in verschiedene Level auf, bettet diese in ein Level passender (per Heuristik bestimmter) Größe ein und stellt diese in einer Übersicht dar.

5.3 Erweiterung von EXTRA

Die Erweiterung von EXTRA teilt sich auf der Implementierungsebene in zwei Teile: Die Erweiterung der Spiellogik und die visuelle Darstellung (Bereitstellen der Assets, kleinere UI-Änderungen). Diese Sektion beschreibt den ersten Teil.

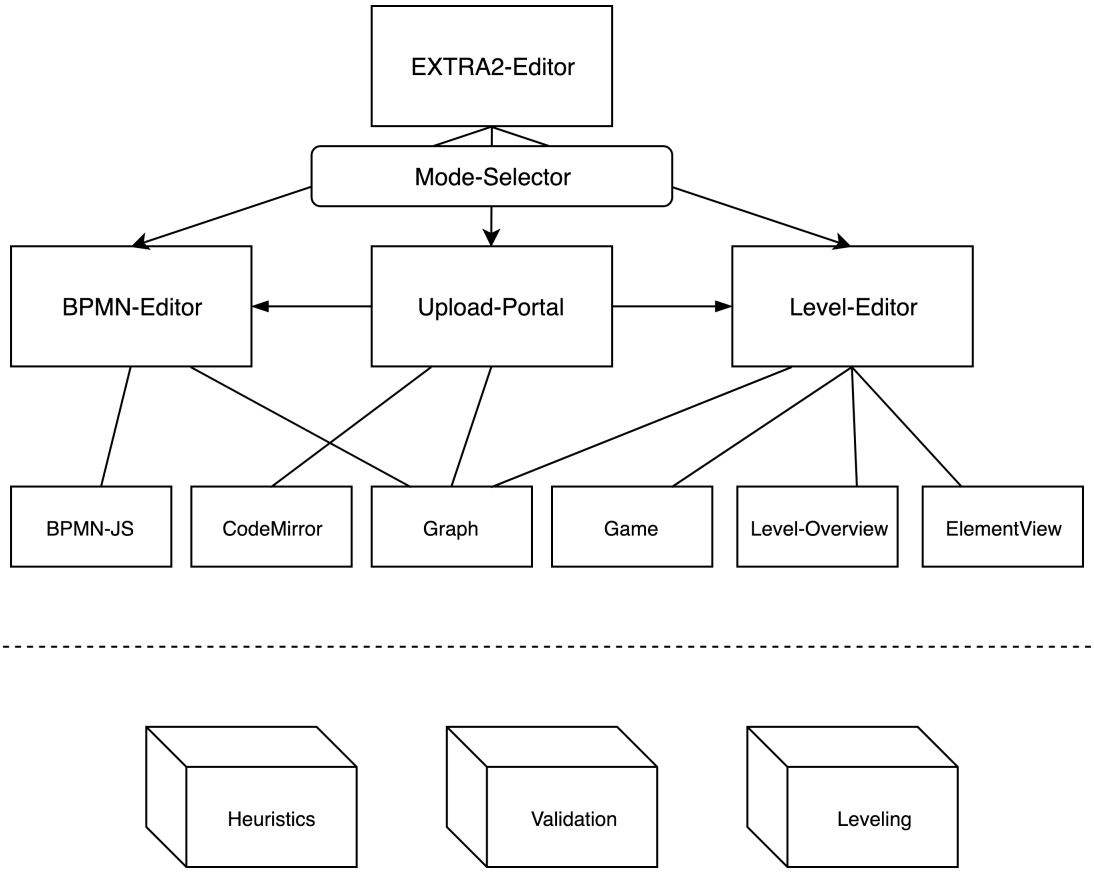


Abbildung 5.6: Die Hauptkomponenten des Editors.

EXTRA-Spiel Wie in Abb. 5.7 dargestellt, besteht der zentrale Arbeitsaufwand im Erweitern des *Structure*-Objekts: In der EXTRA-Update-Schleife hat jede Struktur (d.h. Fabrik/Markt) eine Warteschlange, die gelagerte Produkte abspeichert. Diese wird entsprechend der Transportraten bzw. des Status der verbundenen Routen regelmäßig länger wenn ein Produkt ankommt und kürzer wenn eines die Struktur verlässt („produziert“ wird.) Bei linearen Fabriken ist die Informationen, welches Produkt sich in der Warteschlange befindet irrelevant, da nur eines gelagert werden kann. Entsprechend genügt es nur die Länge der Warteschlange zu speichern. Bei den in dieser Arbeit implementierten Fabriktypen reicht diese Vereinfachung jedoch nicht: Entsprechend wird die abgespeicherte Länge ersetzt durch ein JavaScript-Map-Objekt, dass jedem der n eingehenden Produkte $p_1, \ldots, p_n$ die aktuelle Anzahl $m(p_i) \geq 0$ zuordnet. Die Semantik von eingehenden Produkten ist einfach: Kommt ein Produkt p_i an, wird der entsprechende Eintrag $m(p_i)$ um eins erhöht. Das Produzieren von Produkten muss präziser betrachtet werden: Bei einer *oder*-Fabrik kann das ausgehende Produkt p^o produziert werden, falls $\exists 1 \leq i \leq n.\ m(p_i) > 0$. Das Produzieren hat zur Folge, nun werden alle Einträge in m nach der folgenden Regel aktualisiert: $m(p_i) \coloneqq \min\{m(p_i) - 1, 0\}$. Konkret bedeutet dies, dass jedes vorhanden Produkt verwendet wird, aber nicht alle Produkte vorhanden sein müssen. Für *und*-Fabriken gilt entsprechend: p^0 kann produziert werden, falls $\forall 1 \leq i \leq n.\ m(p_i) > 0$, die Aktualisierung erfolgt mit: $m(p_i) \coloneqq m(p_i) - 1$. Also: alle Produkte müssen vorhanden sein und alle Produkte werden verwendet.

EXTRA-Editor Der für den Editor relevante Teil ist das Spezifizieren eines neuen Input-Formats für die Fabrik-Definition im JSON-Format, dargestellt im Code-Ausschnitt weiter unten. Das alte Format (was dem Format für lineare Fabriken entspricht) definiert eine Konfiguration als ein Objekt mit drei Zeichenketten als Attributen:

- **input**: Das eingehende Produkt
- **name**: Der Name der Konfiguration
- **output**: Das ausgehende Produkt

Im neuen Format entspricht eine Konfiguration einem Objekt, dass analog zu oben Name und Output als String definiert. Beim Input einer solchen Konfiguration handelt es sich nun um ein Objekt, dass folgende Attribute enthält:

- **type**: Der Typ des Inputs: (*and* oder *or*)
- **input**: Eine Liste von eingehenden Produkten

```
{
    "type": "factory",
    "name": "A",
    "image": "icon.png",
    "workforce": 4,
    "configurations": [
        {
            "input": "one",
            "name": "default",
            "output": "two"
        }
    ]
}
```

```
{
    "type": "factory",
    "name": "B",
    "image": "oricon.png",
    "workforce": 4,
    "configurations": [
        {
            "input": {
                "type": "and",
                "input": [
                "three",
                "four",
                "five"
                ]
            },
            "name": "default",
            "output": "six"
        }
    ]
}
```

5.4 Ausgewählte Aspekte und ihre Designentscheidungen

Da sowohl der Leveleditor als auch EXTRA mittels des React-Frameworks implementiert wurden, sind viele Design-Entscheidungen inhärent. Die im Folgenden beschriebenen Aspekte betreffen deswegen überwiegend das Backend bzw. die für den Nutzer nicht sichtbare Teile der Programmlogik der Applikation.

Undo/Redo Für den Leveleditor ist ein Feature sinnvoll, dass es dem Nutzer erlaubt seine Aktionen rückgängig zu machen (sollte er einen Fehler gemacht oder einfach seine Meinung geändert haben). Dazu wird das Memento-Entwurfsmuster (s. Abb. 5.8) verwendet. Die unterliegende React-Komponente hat eine Status-Variable, die sämtliche verhaltensrelevante Daten enthält. Neben diesem aktiven Status speichert sie aber auch eine Liste aller (vergangenen und „zukünftigen“) Status-Variablen ab. Was zu einem bestimmten Zeitpunkt als der aktive Status verwendet wird, wird von dem *currentState*-Pointer (einfach als Index-Wert implementiert) gesteuert. Jedes Mal, wenn sich der Status (bzw. eine seiner Member-Variablen) ändert, wird dieser mit *setState* aktualisiert und die vorherige Version der Liste *state* angehängt. Nun entspricht die *undo*/*redo*-Operation einfach dem Dekrementieren bzw. Inkrementieren dieser Zahl

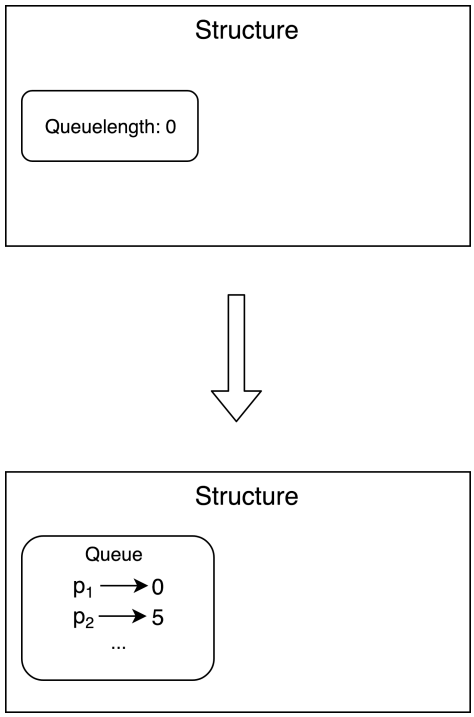


Abbildung 5.7: Die Strukturerweiterung der EXTRA-Programmlogik.

(und dem anschließenden, asynchronen Update der State-Variable mithilfe von *setState*).

Spiel-Einbindung Die Implementierung der interaktiven Level-Vorschau erfolgt, analog zum Hauptspiel, in der Phaser-Game-Engine. Da es sich dabei aber nicht um eine React-Komponente handelt ist die technische Einbindung nicht offensichtlich. Hier geschieht sie über die oben beschriebene *componentDidMount*-Methode und das in Abb. 5.9 dargestellte Strategie-Muster. Phaser interagiert stets mit einer Klasse, die von *Phaser.Game* erbt, diese Klasse wird in dieser Implementierung einfach *Game* genannt. Das konkrete Verhalten dieser Klasse in Bezug auf die Interaktion mit dem (unabhängigen) restlichen Teil der Applikation geschieht über eine *State*-Klasse. Ein solcher state implementiert eine Reihe von hier relevanter Methoden. Dazu zählen:

- Das Umgehen mit Klicks
- Das Umgehen mit unterstützten Drag and Drop-Items
- Was bei einem refresh aktualisiert werden soll

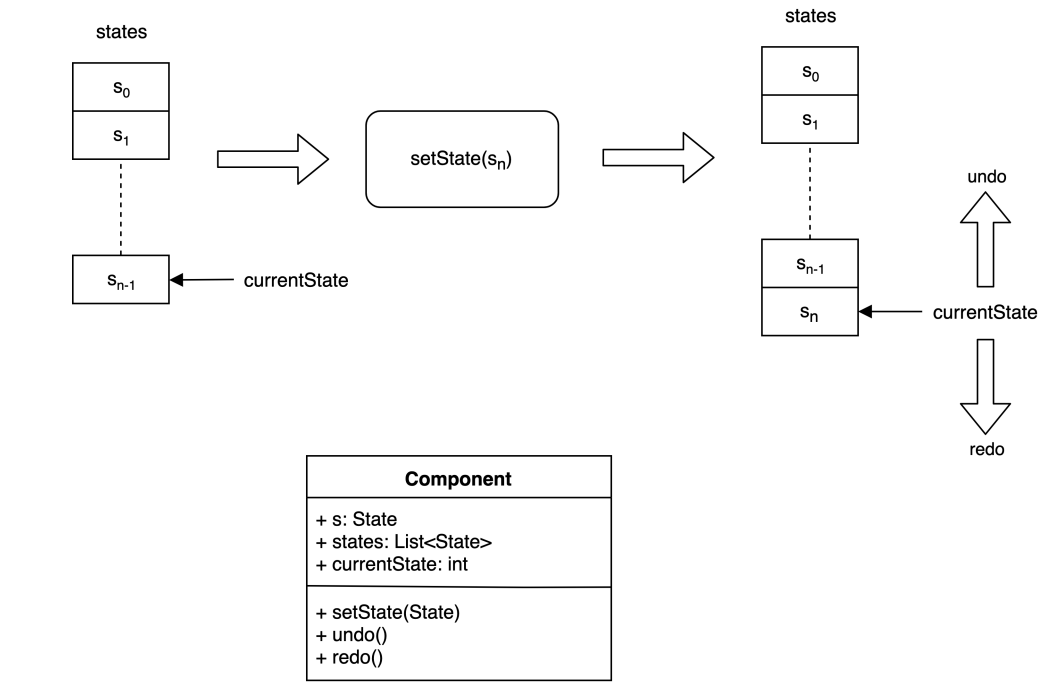


Abbildung 5.8: Die Realisierung des Memento-Musters mittels eines Pointers.

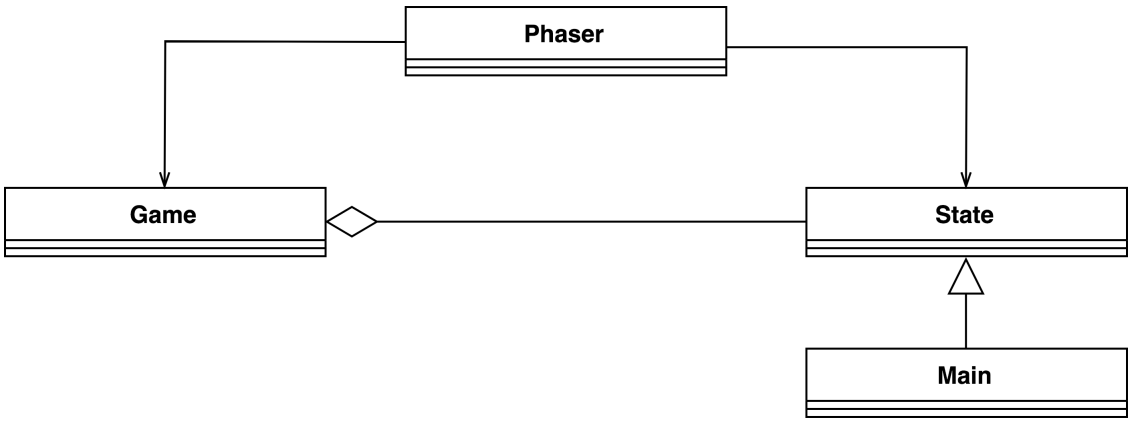


Abbildung 5.9: Die Verhaltenssteuerung des Phaser-Spiels mittels des Strategie-Musters.

Prozessübersetzung Wie oben beschrieben sind Prozesse (im Rahmen des Editors) Graphen mit Knoten- und Kantenbeschreibungen. Programmatisch sind sie JavaScript-Graph-Objekte, bereitgestellt durch die Bibliothek *Dagre*. Wie in Abbildung 5.10 gezeigt, implementiert die Parser-Klasse das Entwurfsmuster Fassade, da sie alle Übersetzungsmethoden für unterschiedliche Ursprungsdateien kapselt. Dies führt nicht nur zu einem übersichtlichen Aufbau der Applikation, sondern ermöglicht auch die einfache Erweiterung um eine weitere Prozessdefinitionssprache wie *SysML*. Der so geparste/übersetzte Graph kann nun mittels der implementierten Algorithmen bearbeitet werden, programmweit und unabhängig von seinem ursprünglichen Format.

Erstellung des vollständigen Graphen Der vorherige Abschnitt detailliert die Übersetzung einer Datei in die generische Repräsentation. Für die Übersetzung in ein EXTRA-Level muss jedoch noch eine Reihe weitere Schritte stattfinden, dargestellt in Abbildung 5.11. Nachdem nun also mit der obigen Übersetzung der Graph in der Hauptkomponente (in der Abbildung als *App* bezeichnet) vorliegt, kann nun der Leveling-Algorithmus stattfinden (*A*): Die BFSLeveling-Klasse ruft die Graphtransformator-Klasse auf (*B*). Hier wird aus dem Graphen per Split and Create-Dijkstra-Algorithmus der vollständige Graph erstellt und zurückgegeben (*C*). Dieser wird dann mithilfe der Dagre-Bibliothek um Layout-Informationen ergänzt (*D*) und an die LevelCreator-Klasse weitergegeben (*E*). Hier werden nun die Layout-Informationen verwendet um den Graph in mehrere hierarchische Subgraphen zu unterteilen. Für jeden dieser Subgraphen wird mithilfe der Layout-Klasse und der dort verwendeten (austauschbaren) Heuristiken ein Level erstellt (*F*). Diese Liste der Subgraphen wird nach Abschluss wieder an die App weitergegeben (*G*).

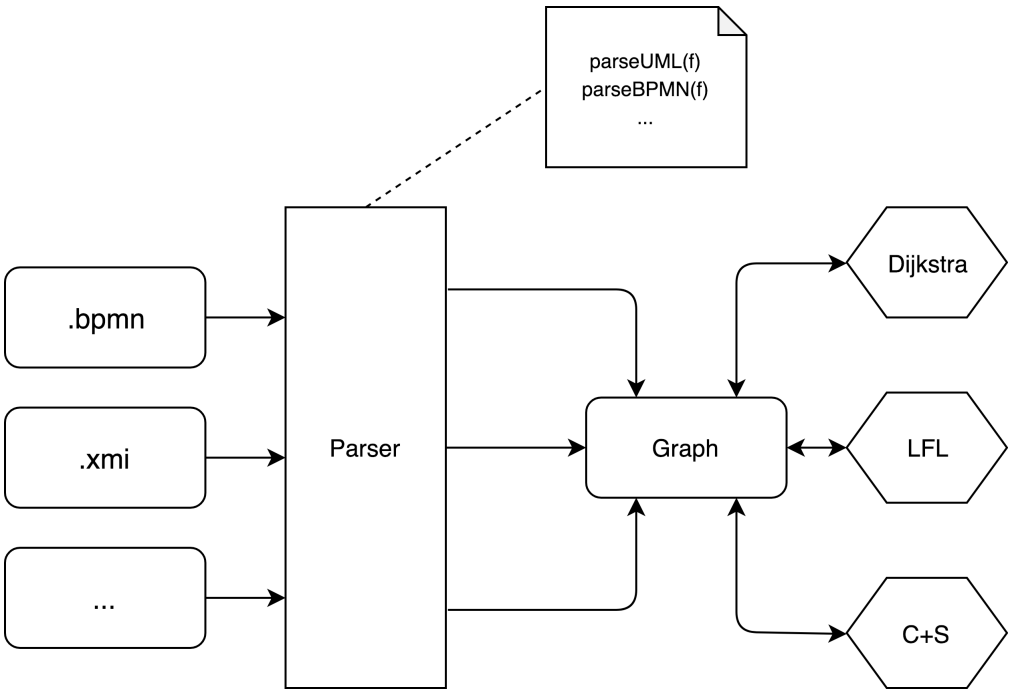


Abbildung 5.10: Die Prozessübersetzung mittels des Fassaden-Musters.

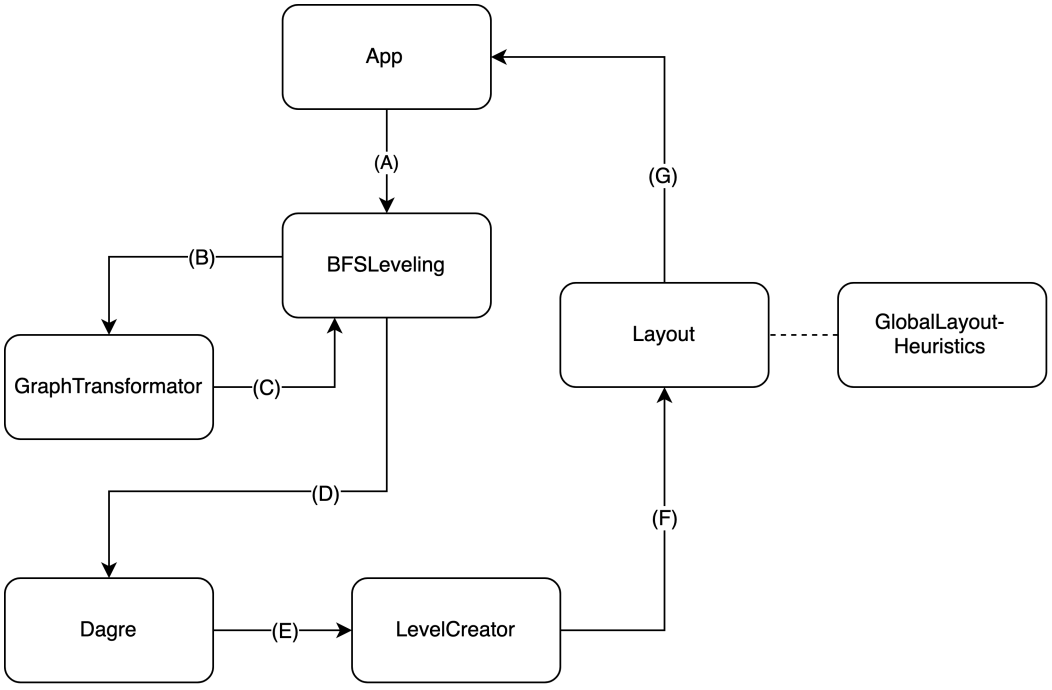


Abbildung 5.11: Der Ablauf der Prozess-Übersetzung.

6 Ein Anwendungsfall

Während die vorhergende Sektion einen Überblick über die technischen Aspekte des Editors gegeben hat, wird in diesem Kapitel ein konkreter Anwendungsfall Schritt für Schritt betrachtet. Im folgenden wird der in Abb. 6.1 dargestellt BPMN-Prozess betrachtet. Dabei handelt es sich um den Ablauf des Transfers eines Mitarbeiters des Fraunhofer IOSB von einer Organisationseinheit zu einer anderen: Der Mitarbeiter bewirbt sich für die neue Stelle. Ist diese Bewerbung erfolgreich muss ein Versetzungsdatum gefunden werden. Dann muss der Mitarbeiter entsprechende Formulare bei seiner alten und seiner neuen Arbeitsstelle unterschreiben. Nach dem anschließenden Unterzeichnen seines Vertrags ist der Prozess abgeschlossen.

In den folgenden Abschnitten werden die Aktionen des Nutzers *kursiv* beschrieben.

Die Gamifizierung mithilfe des hier entwickelten Tools läuft dann ab wie folgt:

Hochladen und Modifizieren des Prozesses Um einen Prozess in ein EXTRA-Level übersetzen zu können, muss dieser in einem unterstützten Format (aktuell *.bpmn* oder .xmi) vorliegen. Diese Dateien können mit beliebigen, externen Tools (wie StarUML für UML-Diagramme) erzeugt werden. Für BPMN kann jedoch auch der anwendungsinterne Editor verwendet werden. Dieser ist in Abbildung 6.2 dargestellt. Er kann nicht nur für das Modellieren eines neuen Prozesses verwendet werden, sondern ermöglicht auch das Modifizieren eines hochgeladenen BPMN-Prozesses. Der somit entstandene neue Prozess kann mittels der Button am unteren Rand heruntergeladen oder gamifiziert werden. (Hier lässt sich auch einstellen, ob dabei fehlende Informationen randomisiert hinzugefügt werden sollen.)

Der Nutzer öffnet die Applikation, lädt den Prozess im .bpmn-Format hoch, öffnet den BPMN-Editor und beschriftet die Knoten und Kanten des Prozesses.

Die automatische Levelgenerierung Wird der Prozess gamifiziert, findet im Hintergrund dessen Übersetzung in eine generische Repräsentation sowie deren Transformation in einen vollständigen Graphen mittels des Switch and Create-Algorithmus statt. Auf diesem wird nun der Leveling-Algorithmus ausgeführt, der ihn in Subgraphen unterteilt (un ein paar andere Schritte durchgeführt, vgl. 4.1.3) Abschließend werden diese Graphen in EXTRA-Level übersetzt

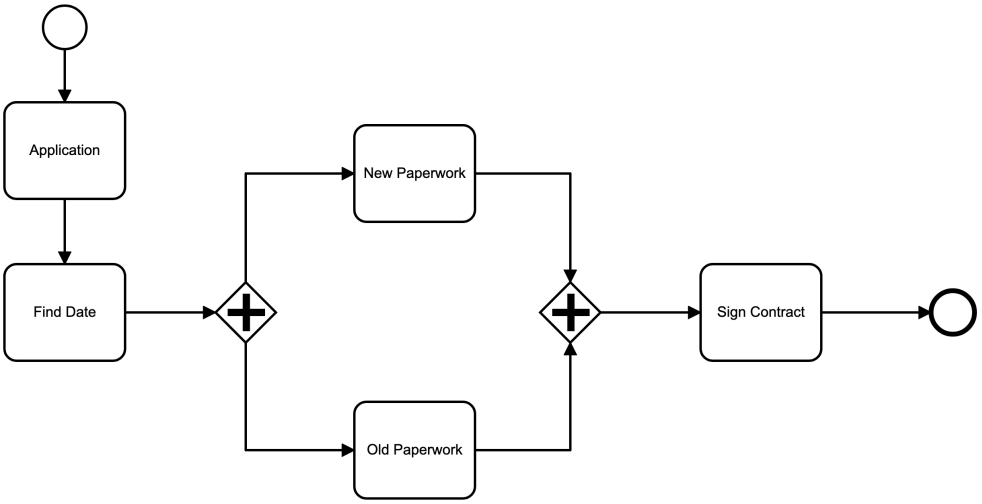


Abbildung 6.1: Der IOSB-interne Transfer-Prozess.

und in einer Übersicht dargestellt (Abbildung 6.3). Hier kann der Nutzer alle oder nur einzelne Level herunterladen sowie per Klick auf die entsprechende Vorschau das Level bearbeiten.

Der Nutzer lädt die ersten zwei Level direkt herunter, das dritte will er bearbeiten und öffnet es per Klick.

Das Bearbeiten der Level Ist ein Level geöffnet (Abbildung 6.4), hat der Nutzer die Möglichkeit alle Details manuell zu bearbeiten. Dies kann entweder über den konkreten JSON-Code geschehen oder über ein GUI. Oben rechts findet er eine laufende Level-Instanz, die auf Änderungen in Echtzeit reagiert. Diese kann er auch verwenden um Gebäude zu verschieben, zu verbinden oder sie per Klick als aktiv zu markieren. Nun öffnet sich kontextabhängig ein passendes GUI, in dem der Nutzer die jeweiligen Informationen anpassen kann.

Der Nutzer markiert die und-Fabrik und ergänzt ihren Namen.

Die Verifikation des Levels (bzw. der aktuellen Instanz desselben) im Editor hat im Wesentlichen drei Aspekte. Sie werden im Verifikationsmenü (Abbildung 6.5) angezeigt. Im Hintergrund findet bei jeder Änderung des Levels eine Analyse statt: Mittels des Augment and Compress-Algorithmus wird die graphentheoretische Lösbarkeit überprüft und mittels des in 4.3 definierten JSON-Schemas wird die Semantik der Leveldefinition überprüft. Ist diese verletzt, wird dies in diesem Menü sowie mittels roten Markierungen im Code angezeigt.

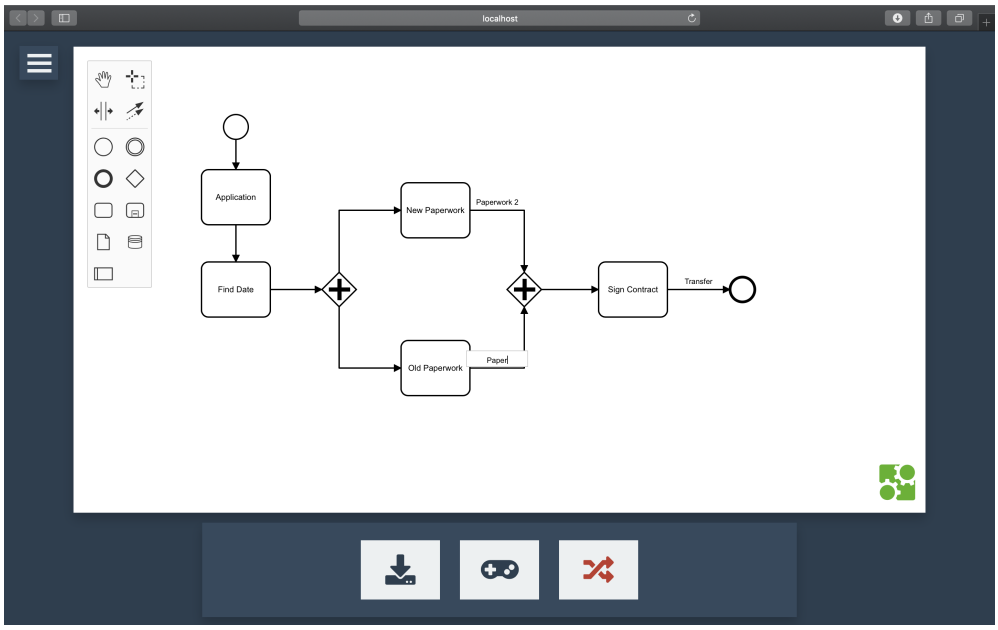


Abbildung 6.2: Der Prozess kann in der Anwendung beschriftet werden.

Der Nutzer sieht die roten Markierungen im Code und öffnet das Verifikationsmenü um mehr zu erfahren. Dort sieht er, dass das Level mehr Informationen braucht aber graphentheoretisch lösbar ist.

Es gibt zwei Hauptansichten des Levels. Neben der Level-Vorschau oben kann der Nutzer auch stets den vollständigen Graphen des Levels anzeigen lassen (Abbildung 6.6). Dieser zeigt Informationen über die Topologie des unterliegenden Prozesses an. Darüber hinaus wird auch farblich angezeigt ob das zum jeweiligen Knoten passende Gebäude baubar ist. Per Klick auf einen solchen Knoten kann der Nutzer, analog zur Levelvorschau, die Details des entsprechenden Gebäudes bearbeiten. Ist dieses jedoch versteckt (d.h. im Level als baubar eingetragen aber nicht platziert) oder gar nicht vorhanden, erscheint ein anderes Menü: Dieses gibt dem Nutzer die Möglichkeit das entsprechende Gebäude wieder im Level zu platzieren oder es zu verstecken.

Der Nutzer betrachtet abschließend die grafische Repräsentation des unterliegenden Prozesses und versteckt gezielt einige Fabriken, die der Spieler dann in EXTRA platzieren muss.

Nun kann das fertige Level in EXTRA gespielt werden (Abbildung 6.7).

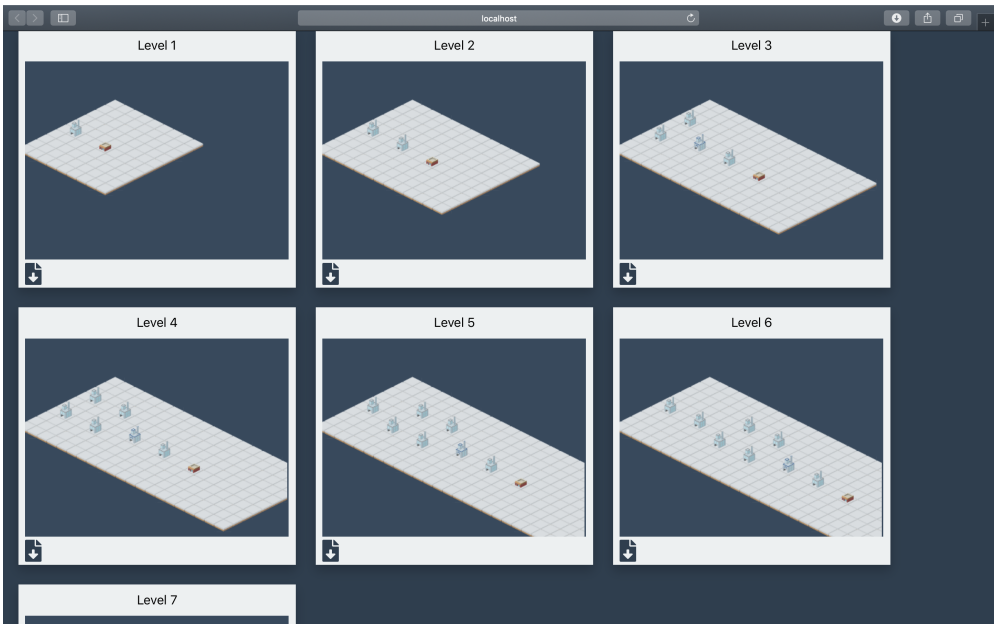


Abbildung 6.3: Die Übersicht der algorithmisch generierten Level.

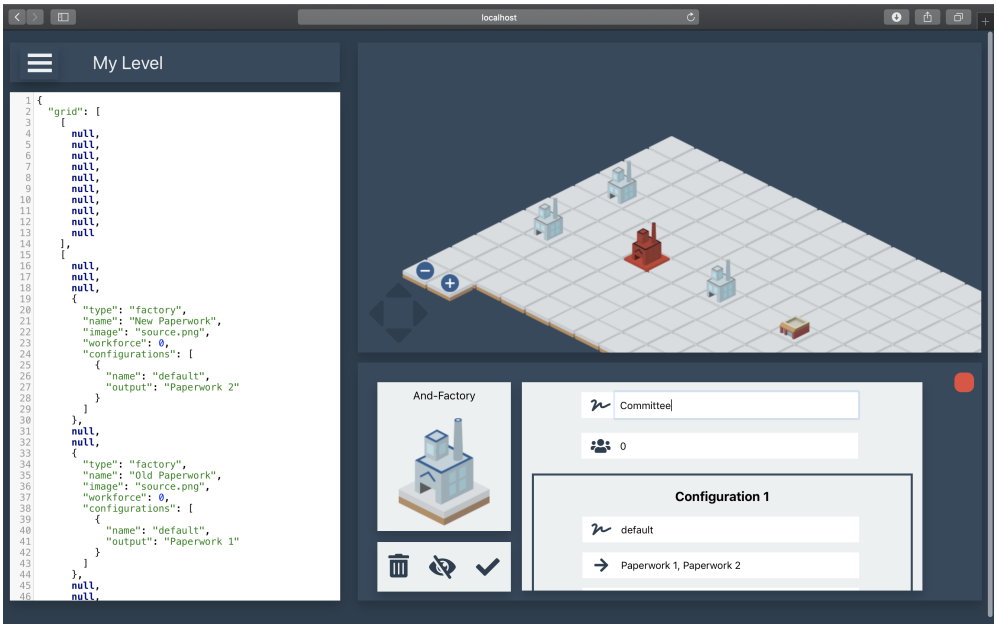


Abbildung 6.4: Im Leveleditor können Informationen ergänzt und verändert werden.

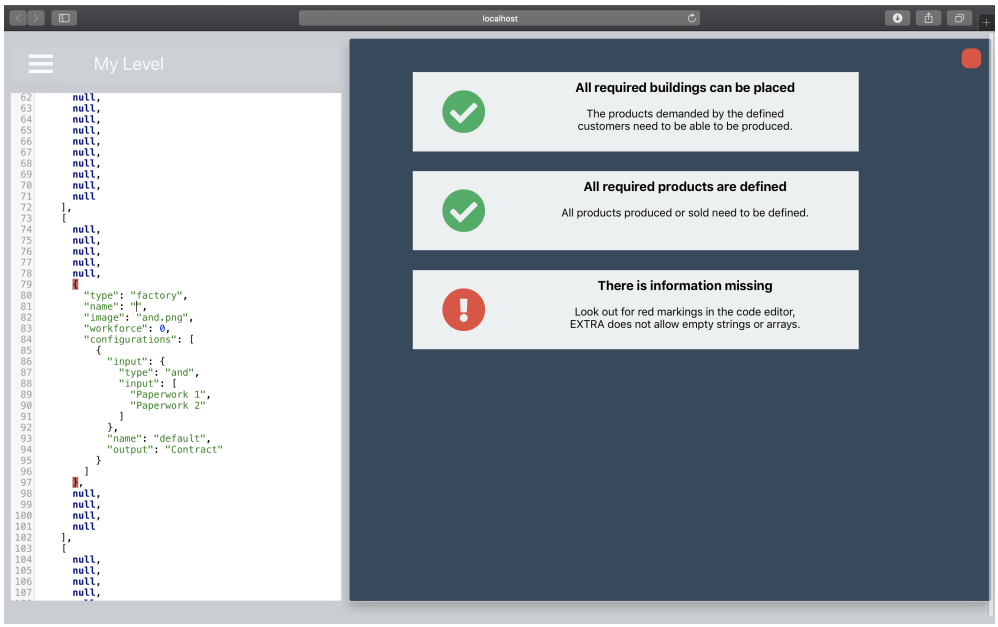


Abbildung 6.5: Dieses Fenster gibt eine Übersicht über die Lösbarkeitskriterien.

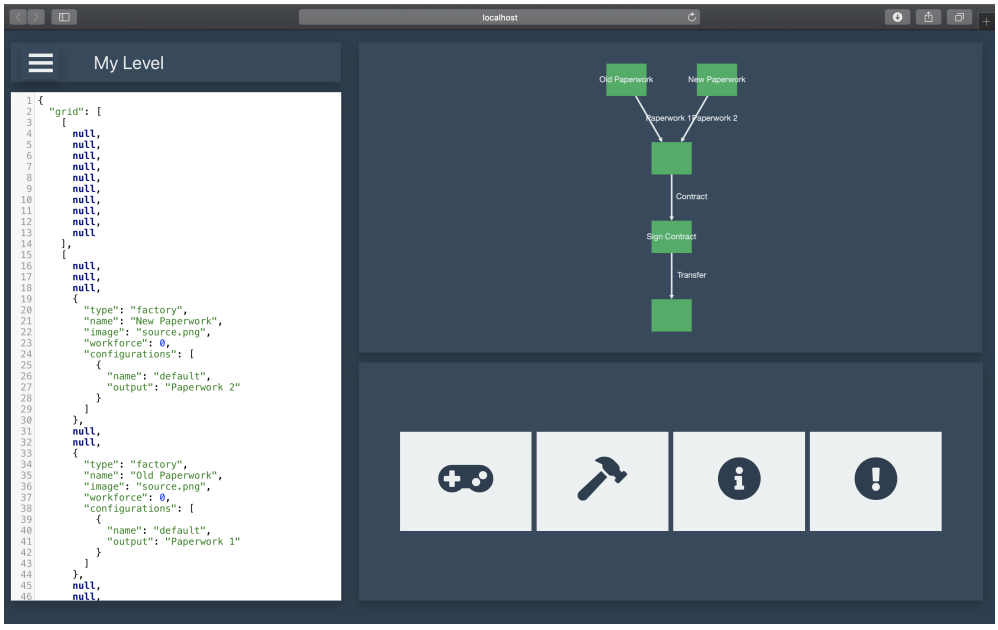


Abbildung 6.6: Die alternative Levelansicht zeigt den vollständigen Graphen.

Abbildung 6.7: Das fertige, spielbare Level in EXTRA.

7 Fazit

7.1 Zusammenfassung

Diese Arbeit detailliert die Konzeption und Implementierung der Erweiterung des Prozesslernspiels EXTRA für die Unterstützung allgemeiner Prozesse. Auf konzeptioneller Ebene wird hierzu die Übersetzung von spezifischen Prozessrepräsentationen (UML,BPMN) in ein generisches Graph-Format definiert. Prozesse in diesem Format können dann mithilfe hier entworfener Algorithmen weiter bearbeitet bzw. analysiert werden: Der *Split and Create-Dijkstra*-Algorithmus erstellt aus einem gegebenen Prozess den vollständigen Graphen, dieser enthält alle möglichen Pfade im zum Prozess gehörenden EXTRA-Level. Dabei handelt es sich um einen Graphen, der (gemäß der EXTRA-Metapher von Knoten als Fabriken) nur Knoten mit einem Ausgangsgrad von max. 1 enthält. Dieser Graph wird in den zwei anderen theoretischen Konzepten verwendet: Der *Layout-First-Leveling*-Algorithmus erweitert jeden Knoten um Layout-Informationen (hier einfach eine Koordinate $(x,y) \in \mathbb{N} \times \mathbb{N}$), teilt diese damit in Ebenen ein und bestimmt, angefangen an der letzten Ebene, iterativ die Subgraphen, die dann in konkrete EXTRA-Level übersetzt werden. Um die Lösbarkeit eines gegebenen Levels zu prüfen wird der *Augment and Compress*-Algorithmus verwendet. Dieser kombiniert drei Aspekte: Augmentierung jedes Knotens um einen Status (gebaut, baubar, nicht baubar), Pfadkompression linearer Pfade und Tiefensuche im so entstehenden Graphen. So kann die sog. *graphentheoretische Lösbarkeit* des Levels programmatisch bzw. theoretisch bestimmt werden.

Auf praktischer Ebene beschriebt die Arbeit die Implementierung eines Level-Editors, der es erlaubt Level für EXTRA zu erstellen und die oben beschriebenen Algorithmen umsetzt um aus separat definierten Prozessen spielbare Level zu generieren. Da ein solcher Prozess nicht alle (Meta-)Informationen enthält, die ein solches Level benötigt, werden in den Editor integrierte Heuristiken und Randomisierungsalgorithmen verwendet um solche Lücken, soweit möglich, zu füllen.

7.2 Weitere Arbeiten

Heuristiken und Randomisierung Die zum jetzigen Stand verwendeten Heuristiken sind komplett manuell (d.h. zur Entwicklungszeit) definiert, sind also nicht zur Laufzeit und nur von fortgeschrittenen Nutzern anpassbar. Des Weiteren ist die Funktionsweise der Heuristiken erweiterbar: Bisher werden nur Knoten- und Kanten-Anzahl als Kennzahlen in den Formeln verwendet, hier könnte man weitere, komplexere Daten als Grundlage verwenden. Ebenfalls sinnvoll wäre die Unterstützung von dynamischer Schwierigkeit. So könnte der Nutzer dann mittels eines Schiebereglers die gewünschte Schwierigkeit der generierten Level einstellen. Ein weiterer Ansatzpunkt für zukünftige Arbeiten ist das Randomisierungssystem. Die vorhandene Randomisierung ist zwar mittels des Szenario-System variabel auf die Bedürfnisse des Nutzers einstellbar, die eigentliche Generierung (d.h. die Verteilung der Namen) ist aber nicht weiter modifizierbar: Hier könnte eventuell ein kontextabhängiges System implementiert wird, dass über den Knoten-Typ als alleinige Informationsgrundlage hinausgeht: Es könnten Vorgänger bzw. Nachfolger betrachtet werden oder gar ein Natural Language Processing-basiertes System verwendet werden.

Prozessmodellierung Die im Rahmen dieser Arbeit implementierte Prozessübersetzung unterstützt die zwei wichtigsten Prozessmodellierungsstandards, je nach Anwendungsfall kann diese jedoch um die Unterstützung anderer Sprachen erweitert werden. Damit einhergehend gibt es noch zwei weitere Erweiterungsmöglichkeiten:

Weitere Diagrammtypen und -elemente Während es im BPMN-Standard kein Konzept von Diagrammtypen gibt, spezifiziert UML eine ganze Reihe solcher Typen. Aktuell wird nur das *Aktivitätsdiagramm* unterstützt. Zukünftige Versionen des Spiels (und des Editors) können um Unterstützung für andere Diagrammtypen erweitert werden. In diesem Zusammenhang können sich zukünftige Arbeiten auch mit weiteren Diagrammkomponenten beschäftigen, die bisher nicht unterstützt werden: Sowohl BPMN als auch UML erlauben die Spezifikation von anderen Elemente bzw. komplexerer Semantik wie *Parallelität* oder *Nachrichtentypen*. Die korrekte Abbildung dieser Konzepte erfordert jedoch weitere Forschung und konzeptionelle Arbeit.

Anwendungsinterner UML-Editor Durch die Einbindung der BPMN.JS-Bibliothek ist das Erstellen bzw. Bearbeiten von BPMN-Prozessen direkt im hier implementierten Editor möglich. Für UML existiert keine solche Bibliothek, weswegen auf die Open-Source- Software

StarUML[1] und das dort installierbare XMI-Export-Plugin zurückgegriffen wird. Ein natives Tool für das Erstellen von Aktivitätsdiagrammen wäre eine sinnvolle Ergänzung zum Editor.

EXTRA Die zukünftige Erweiterung von EXTRA kann zwei Aspekte betreffen: Aus der prozesstheoretischen Sicht können zukünftige Arbeiten Unterstützung für die oben beschriebenen zusätzlichen semantischen Elemente (Nachrichtentypen etc.) hinzufügen. Aus Game-Design-Sicht bieten sich zahlreiche Erweiterungen an: Ereignisse wie Naturkatastrophen, die die Geometrie des Levels während des Spielens verändern oder solche, die die Spiellogik komplexer machen: Streiks, falls sich zu viele Arbeiter in einer Fabrik befinden oder Kunden, die (zufallsbasiert) kurzfristig ihre Wünsche ändern. In diesem Zusammenhang könnte man auch verschiedene Typen von Bauplätzen entwerfen oder Fabriken nicht nur nach ihrem Input-Typ (sondern nach Kategorien wie Preis, Kapazität etc.) einteilen.

1 http://staruml.io

Literatur

[Bac+81] Christian Bachmaier u. a. *Cyclic Leveling of Directed Graphs*. 1981.

[BK02] Ulrik Brandes und Boris Köpf. *Fast and Simple Horizontal Coordinate Assignment*. 2002.

[Cha+15] Rafael Oliveira Chaves u. a. *Experimental Evaluation of a Serious Game for Teaching Software Process Modeling*. 2015.

[CL87] Mihaly Csikszentmihalyi und Reed Larson. *Flow and the Foundations of Positive Psychology*. 1987.

[Con+12] Thomas M. Connolly u. a. „A systematic literature review of empirical evidence on computer games and serious games". In: *Computers and Education* 59.2 (2012), S. 661–686. arXiv: arXiv:1011.1669v3. URL: http://dx.doi.org/10.1016/j.compedu.2012.03.004.

[CT12] Michele Chinosi und Alberto Trombetta. „BPMN: An introduction to the standard". In: *Computer Standards and Interfaces* 34.1 (2012), S. 124–134. arXiv: 9503016v1 [arXiv:quant-ph]. URL: http://dx.doi.org/10.1016/j.csi.2011.06.002.

[Dag18] Dagre. https://github.com/dagrejs/graphlib/wiki/API-Reference. 2018.

[DDO08] Remco M. Dijkman, Marlon Dumas und Chun Ouyang. „Semantics and analysis of business process models in BPMN". In: *Information and Software Technology* 50.12 (2008), S. 1281–1294. arXiv: 1304.1186.

[De 17] Olga De Troyer. „Towards effective serious games". In: *2017 9th International Conference on Virtual Worlds and Games for Serious Applications, VS-Games 2017 - Proceedings* (2017), S. 284–289.

[Eck03] Richard Van Eck. *Building Artificially Intelligent Learning Games*. 2003, S. 271–307.

[Fit18] Z Fitz-Walter. „Introduction to Gamification". In: *Introduction to Gamification* 1.1 (2018), S. 1–67. eprint: 2009/837095.

[Fog09] Bj Fogg. *A behavior model for persuasive design*. 2009.

[Gac15a] Cory Gackenheimer. *Introduction to React*. 2015.

[Gac15b] Cory Gackenheimer. *What Is React?* 2015.

[GEM13] C. Girard, J. Ecalle und A. Magnan. „Serious games as new educational tools: How effective are they? A meta-analysis of recent studies“. In: *Journal of Computer Assisted Learning* 29.3 (2013), S. 207–219. arXiv: 0710.4428v1.

[Gun16] Alexander Gundermann. *A Web-Based Serious Game for Joint Training*. 2016.

[Hau03] M Hause. *Domain specific process modelling - making UML accessible*. 2003.

[Heu03] Jim Heumann. *Introduction to business modeling using the Unified Modeling Language (UML)*. 2003.

[Int17] ECMA International. *ECMA-404 The JSON Data Interchange Syntax*. 2017. URL: https://www.ecma-international.org/publications/standards/Ecma-404.htm.

[Leg07] Pierre Leger. *Using a Simulation Game Approach to Teach ERP Concepts*. 2007.

[M17] Haldar M. *ReactJs component lifecycle methods: A deep dive*. https://hackernoon.com/reactjs-component-lifecycle-methods-a-deep-dive-38275d9d13c0. zuletzt besucht: 27.2.19. 2017.

[NPM18] NPM, Inc. https://medium.com/npm-inc/npm-weekly-133-billions-of-packages-downloaded-621a5196593. zuletzt besucht: 27.2.19. 2018.

[NPM19] NPM, Inc. https://www.npmjs.com/. zuletzt besucht: 27.2.19. 2019.

[Per00] Simon Perry. *When is a Process Model Not a Process Model - A Comparison between UML and BPMN*. 2000.

[Rib+12] Cláudia Ribeiro u. a. *Using Serious Games to Teach Business Process Modeling and Simulation*. 2012.

[SA] E Sachlos und D Auguste. *Three Principles : Redux*. URL: https://redux.js.org/introduction/threeprinciples.

[San11] Marco Santorum. „A serious game based method for business process management“. In: *Proceedings - International Conference on Research Challenges in Information Science* (2011).

[Sch17] Andreas Schoknecht. *Similarity of Business Process Models — A State-of-the-Art Analysis*. 2017.

[She88] Thomas Sheridan. „Task Analysis, Task Allocation and Supervisory Control“. In: *Handbook of human-computer interaction* (Dez. 1988).

[SR16] Stefan Strecker und Kristina Rosenthal. „Process Modelling as Serious Game: Design of a Role-Playing Game for a Corporate Training“. In: *Proceedings - CBI 2016: 18th IEEE Conference on Business Informatics* 1 (2016), S. 228–237.

[SSG16] Alexander Streicher, Daniel Szentes und Alexander Gundermann. „Game-based training for complex multi-institutional exercises of joint forces“. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9891 LNCS (2016), S. 497–502.

[SSR14] Alexander Streicher, Daniel Szentes und Wolfgang Roller. „Scenario Assistant for Complex System Configurations“. In: *IADIS International Journal on Computer Science and Information Systems* 9.2 (2014), S. 38–52.

[ST03] Valerie Shute und Brendon Towle. „Adaptive E-Learning“. In: *Educational Psychologist* 38.2 (2003), S. 105–114. URL: http://www.tandfonline.com/doi/abs/10.1207/S15326985EP3802%7B%5C_%7D5.

[TLB16] Olfa Chourabi Tantan, Daniel Lang und Imed Boughzala. „Learning Business Process Management through Serious Games: Feedbacks on the Usage of INNOV8“. In: *Proceedings - CBI 2016: 18th IEEE Conference on Business Informatics* 1 (2016), S. 248–254.

[TT15] Christina Tsagkani und Aphrodite Tsalgatidou. „Abstracting BPMN models“. In: *Proceedings of the 19th Panhellenic Conference on Informatics - PCI '15* (2015), S. 243–244. URL: http://dl.acm.org/citation.cfm?doid=2801948.2802035.

[Wan05] H. Wandke. „Assistance in human–machine interaction: a conceptual framework and a proposal for a taxonomy“. In: *Theoretical Issues in Ergonomics Science* 6.2 (2005), S. 129–155. eprint: https://doi.org/10.1080/1463922042000295669. URL: https://doi.org/10.1080/1463922042000295669.

[WBR10] Stephen West, Ross A. Brown und Jan C. Recker. „Collaborative Business Process Modeling Using 3D Virtual Environments“. In: *16th Americas Conference on Information Systems (AMCIS). 249* June (2010), S. 1–11.

[Xu11] Yingjuan Xu. *The Formal Semantics of UML Activity Diagram Based on Process Algebra*. 2011.

[Yi08] Xie Yi. „Process modeling and simulation based on extended UML activity and GPSS“. In: *Proceedings of the IEEE International Conference on Automation and Logistics, ICAL 2008* September (2008), S. 2931–2935.

Tabellenverzeichnis

Abbildungsverzeichnis