HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG

Fakultät Informatik

Logische Oder-Zusammenführung

Prof. Dr. Ulrich Hedtstück
Sommersemester 2009

von

Max Nagl
nagl@fh-konstanz.de

Inhaltsverzeichnis

1 Einleitung

Prozesse werden in der Wirtschaft immer häufiger grafisch modelliert. Dazu werden verschiedene Spezifikationssprachen verwendet. Eine dieser Sprachen zur grafischen Modellierung von Geschäftsprozessen ist die Business Process Modeling Notation (kurz: BPMN).

In diesen Sprachen stehen verschiedene Gateways zur Verfügung um den Kontrollfluss zu steuern. Diese Gateways sind mit verschiedenen Problemen verbunden, mit denen sich diese Ausarbeitung beschäftigt. Im Besonderen wird hier auf das Problem der sogenannten logischen Oder-Zusammenführung eingegangen.

Im Rahmen dieser Arbeit ist ein Java-Programm zur grafischen Simulation dieser Gateways entstanden. Mit Hilfe dieses Programmes lassen sich die verschiedenen Gates mit verschiedenen Implementierungsstrategien simulieren. Mehrere in dieser Ausarbeitung abgedruckte Bildschirmfotos wurden mit diesem Programm erstellt. Der vollständige Quellcode ist auf der beiliegenden CD enthalten.

Grundlagen in der Modellierung von Geschäftsprozessen und der Graphentheorie werden in dieser Ausarbeitung vorausgesetzt.

2 Grundlagen

In diesem Kapitel wird auf die verschiedenen Gateways, auf Aktivitäten und auf wohlgeformte Graphen eingegangen.

2.1 Die Gateways

Gateways dienen zur Steuerung des Kontrollflusses. Ein Gateway besteht aus zwei Teilen:

- einem Split, auch Fork genannt
- einem Join, auch Merge genannt

Beim Split wird der Kontrollfluss auf verschiedene Pfade aufgeteilt. Ein Split besitzt einen Eingang und einen oder mehrere Ausgänge.

Der Join führt mehrere Kontrollflüsse zu einem Einzelnen zusammen. Ein Join besitzt einen oder mehrere Eingänge und einen Ausgang.

In der BPMN-Notation werden Gateways als Raute dargestellt.

Es existieren drei verschiedene Gateways, welche in den folgenden Abschnitten erklärt werden.

2.1.1 Das AND-Gateway

Das AND-Split teilt den Kontrollfluss immer auf alle möglichen Pfade auf. Der AND-Join wartet immer darauf, dass an allen Eingängen Kontrollflüsse angekommen sind und führt anschließend den Kontrollfluss fort.

Das AND-Gateway wird mit einer Raute und einem + gekennzeichnet.

2.1.2 Das XOR-Gateway

Der XOR-Split leitet den Kontrollfluss immer auf genau einen möglichen Pfad um. Der XOR-Join wartet auf genau einen Kontrollfluss an einem beliebigen Eingang. Kommt dieser Kontrollfluss am Join an, so wird die Ausführung sofort fortgesetzt.

Das XOR-Gateway wird mit einer Raute und einem X gekennzeichnet.

2.1.3 Das OR-Gateway

Beim OR-Split wird der Kontrollfluss auf mehrere mögliche Pfade aufgeteilt. Ob es möglich ist, dass der OR-Split gar keinen ausgehenden Pfad wählt, und somit die Ausführung zum Erliegen bringt, ist von Modell zu Modell unterschiedlich.

Der Or-Join hat die Aufgabe, die Pfade, die im OR-Split gestartet wurden, wieder zu einem gemeinsamen Kontrollfluss zusammenzuführen. Dazu muss er allerdings wissen, in welche Pfade der Split den Kontrollfluss aufgeteilt hat. Mit diesem Problem beschäftigt sich der Abschnitt „Implementierung des OR-Gateways“.

Das OR-Gateway wird mit einer Raute und einem O gekennzeichnet.

2.2 Aktivitäten

Eine Aktivität stellt eine zu erledigende Aufgabe in einem Geschäftsprozess dar. Eine elementare Aktivität wird auch als Task bezeichnet.

Aktivitäten werden in der BPMN-Notation als Rechteck mit abgerundeten Ecken dargestellt.

2.3 Graphen

Die Darstellung von Geschäftsprozessen in der BPMN kann als gerichteter Graph interpretiert werden. Dabei werden Aktivitäten und Gateways als Knoten interpretiert. Die möglichen Kontrollflüsse von Knoten zu Knoten werden als Kanten dargestellt.

Die Interpretierung als Graph erleichtert die Beschreibung einiger Probleme. Des Weiteren wird für die Implementierung viel Wissen über die Graphentheorie benötigt.

2.3.1 Wohlgeformtheit

Damit ein BPMN-Graph als wohlgeformt gilt, darf er ausschließlich aus wohlgeformten Konstrukten bestehen. Diese wohlgeformten Konstrukte müssen die folgenden Anforderungen erfüllen:

1. Auf jeden Split muss ein Join des gleichen Typs folgen.
2. Zwischen dem Split und dem Join können sich beliebig viele Aktivitäten befinden.
3. Ein XOR-Join darf vor dem XOR-Split kommen, wenn der XOR-Split auf einem Weg den Kontrollfluss fortführt und auf allen anderen Wegen zu einem direkt vor ihm liegenden XOR-Join führt.
4. Wird ein wohlgeformtes Konstrukt in eine Kante eines weiteren wohlgeformten Konstruktes eingesetzt, so ist das Ergebniss ebenfalls ein wohlgeformtes Konstrukt.

Im Folgenden sind Beispiele zu wohlgeformten Graphen zu sehen:

Abbildung 2.1 zeigt einfache, nach den Regeln 1 und 2 mögliche Konstrukte.

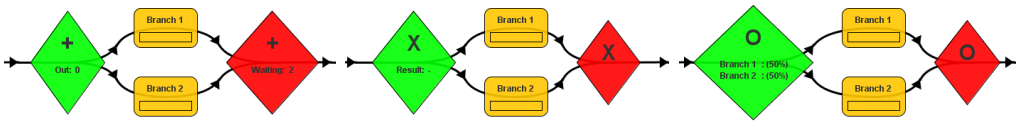

Abbildung 2.1: Einfache wohlgeformte Konstrukte

Abbildung 2.2 zeigt ein einfaches, nach den Regeln 2 und 3 mögliches Konstrukt.

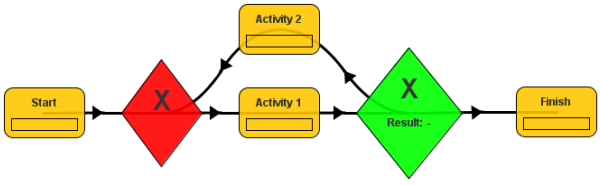

Abbildung 2.2: Ein weiteres wohlgeformtes Konstrukt

Abbildung 2.2 zeigt ein nach Regel 4 verschachteltes Konstrukt.

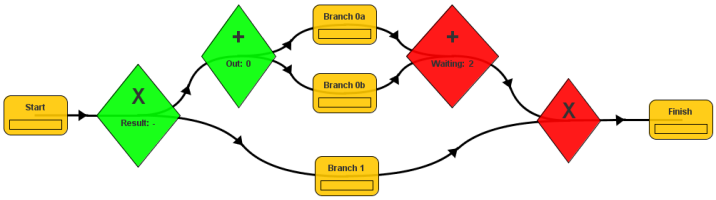

Abbildung 2.3: Ein verschachteltes wohlgeformtes Konstrukt

Abbildung 2.4 zeigt nicht wohlgeformtes Konstrukt.

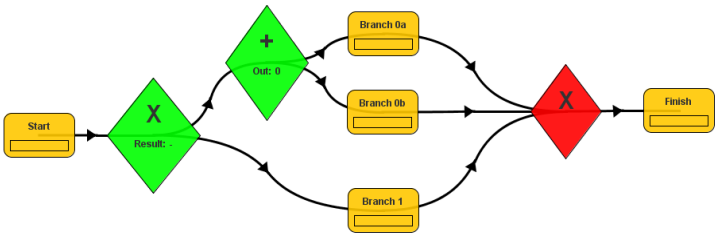

Abbildung 2.4: Ein nicht wohlgeformtes Konstrukt

Nicht wohlgeformte Graphen können zu Problemen bei den Joins führen. Darauf wird in den folgenden Kapiteln noch genauer eingegangen.

Weitere Informationen zu wohlgeformten Graphen sind in Gruhn u. Laue [2005] zu finden.

3 Die Anwendung LogiGate

Wie in der Einleitung bereits erwähnt ist im Rahmen dieser Arbeit eine Anwendung zur Simulation von Gateways mit dem Namen LogiGate entstanden. Die Anwendung wird in diesem Kapitel kurz beschrieben.

3.1 Die Darstellung

Die Darstellung in LogiGate ist stark an die Business Process Modeling Notation angelehnt. Einige Besonderheiten werden nun kurz erläutert:

- Gateways werden mit Hilfe von Rauten dargestellt. Bei manchen Splits und Joins werden innerhalb der Raute weitere Informationen angezeigt.
- Aktivitäten werden durch ein Rechteck mit abgerundeten Ecken symbolisiert. In den Rechtecken befindet sich ein Fortschritsbalken, welcher den aktuellen Fortschritt innerhalb der Aktivität verdeutlicht.
- Es existieren keine Events. Die Simulation beginnt in der Aktivität „Start“.
- Aktivitäten werden orange, Splits grün und Joins rot dargestellt.
- Bei durchlaufenen Objekten verblasst die Farbe.
- Die möglichen Kontrollflüsse werden durch Pfeile dargestellt. Bei durchlaufenen Kontrollflüssen werden die Linien rot eingefärbt. Auf diese Weise kann gut erkannt werden, auf welchen Pfaden der Kontrollfluss aktiv war.

3.2 Die Anwendungsoberfläche

In Abbildung 3.1 wurde ein Bildschirmfoto der Anwendungsoberfläche abgedruckt. Am oberen Rand des Fensters ist die Kontrollleiste zu sehen, auf welcher sich von links nach rechts die folgenden Bedienelemente befinden:

Graph-Auswahl Hier kann einer von mehr als 20 vordefinierten Beispielgraphen zur Simulation ausgewählt werden.

Run Mit dieser Schaltfläche wird die Simulation gestartet.

Reset Mit dieser Schaltfläche wird die Simulation zurückgesetzt.

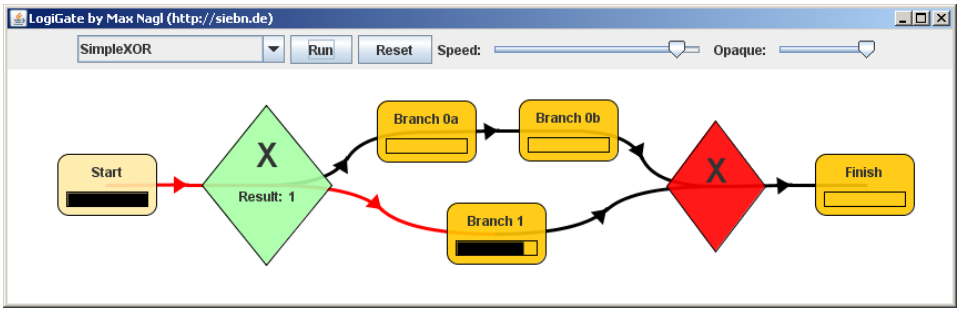

Abbildung 3.1: Die Anwendungsoberfläche von LogiGate

Speed Mit diesem Schieberegler kann die Geschwindigkeit der Simulation beeinflusst werden.

Opaque Mit diesem Schieberegler kann die Transparenz der Aktivitäten und Gateways beeinflusst werden.

Darunter befindet sich die Simulationsfläche. Die verschieden Aktivitäten und Gateways können mit Hilfe der Maus verschoben werden. Ein Doppelklick auf ein Element startet die Simulation in diesem Element.

4 Implementierungen

Bei den Implementierungen treten einige Probleme auf. Dabei unterscheidet man Probleme bei Splits und bei Joins. Die beiden Elemente werden deshalb jeweils getrennt von einander betrachtet.

Bei Splits muss entschieden werden, in welche Wege der Kontrollfluss aufgeteilt wird. Beim Join muss hingegen entschieden werden, wann der Kontrollfluss fortgesetzt wird.

Für alle Splits gilt Folgendes:

- Die Liste aller möglichen ausgehenden Pfade ist in der Variablen `branches` abgespeichert.

Für alle Joins gilt Folgendes:

- Die Liste aller möglichen eingehenden Pfade ist in der Variablen `branches` abgespeichert.
- Der nachfolgende Knoten ist in der Variablen `next` abgespeichert.

Die folgenden Quelltexte sind im Pseudocode abgedruckt. Der Syntax ist an der Sprache Java angelehnt.

4.1 Das AND-Gateway

Das AND-Gateway ist recht einfach zu implementieren und stellt, wie in den folgenden zwei kurzen Abschnitten zu sehen, kaum Probleme dar.

4.1.1 Der AND-Split

Beim AND-Split wird durch die Liste der ausgehenden Pfade iteriert. Dabei wird jeder Pfad gestartet. Eine mögliche Implementierung könnte wie folgt aussehen:

```
1 public void run() {
2   for (Branch b:branches)
3       b.run();
4 }
```

Damit die nachfolgenden Pfade parallel ablaufen, wird hierbei für jeden ausgehenden Pfad wird neuer Thread erzeugt.

4.1.2 Der AND-Join

Beim AND-Join wird jedes Mal, wenn der Kontrollfluss an einem Pfad ankommt, dieser Pfad aus der Liste der möglichen Pfade gelöscht. Sobald sich in dieser Liste kein Pfad mehr befindet, kann die Ausführung fortgesetzt werden. Eine mögliche Implementierung könnte wie folgt aussehen:

```
1 public void run(Branch b) {
2   if (branches.contains(b))
3     branches.remove(b);
4   if (branches.size() == 0)
5     next.run();
6 }
```

4.2 Das XOR-Gateway

Das XOR-Gateway ist bei wohlgeformten Graphen recht einfach zu implementieren. Ist der Graph allerdings nicht wohlgeformt, so kann es zu Problemen kommen. Diese werden im Folgenden beschrieben.

4.2.1 Der XOR-Split

Beim XOR-Split muss durch ein geeignetes Verfahren entschieden werden, auf welchem Weg der Kontrollfluss weitergeleitet wird. Dabei muss sicher gestellt werden, dass immer genau ein Weg gewählt wird. Dies kann auf verschiedene Arten realisiert werden:

- Man ruft für jeden Weg eine boolsche Entscheidefunktion auf. Sobald die Entscheidefunktion `true` zurückliefert, wird dieser Weg gewählt. Die restlichen Entscheidefunktionen werden nicht aufgerufen. Liefert keine Entscheidefunktion `true`, so wird ein Default-Weg gewählt.
- Man ruft ein Entscheidefunktion auf, welche einen ganzzahligen Wert zwischen 1 und der Anzahl der Wege zurück gibt. Der Weg mit dieser Nummer wird dann gewählt.

Eine mögliche Implementierung des zweiten Verfahrens könnte wie folgt aussehen:

```
1 public void run() {
2   int i = decision.decide();
3   if ((i > 0) && (i <= branches.size()))
4     branches.get(i).run();
5 }
```

4.2.2 Der XOR-Join

Der XOR-Join ist sehr einfach zu implementieren. So bald ein Kontrollfluss am Join ankommt, wird der Nachfolger ausgeführt. Hier eine mögliche Implementierung:

```
1 public void run() {
2   next.run();
3 }
```

Ein Problem bei der XOR-Zusammenführung bei nicht wohlgeformten Graphen besteht darin, dass sich der vom XOR-Split ausgehende Kontrollfluss durch einen weiteren Split in mehrere Kontrollflüsse aufteilen kann. Abbildung 4.1 verdeutlicht dieses Problem an einem Beispiel. Da der Kontrollfluss zwischen XOR-Split und XOR-Join durch einen AND-Split aufgeteilt wird, kommt hier der Kontrollfluss zweimal am XOR-Join an. Bei der oben beschriebenen Implementierung würde dann die Aktivität „Finish“ zwei mal ausgeführt werden. Um dies zu vermeiden, sollten die Graphen innerhalb der XOR-Gateways wenn möglich wohlgeformt sein.

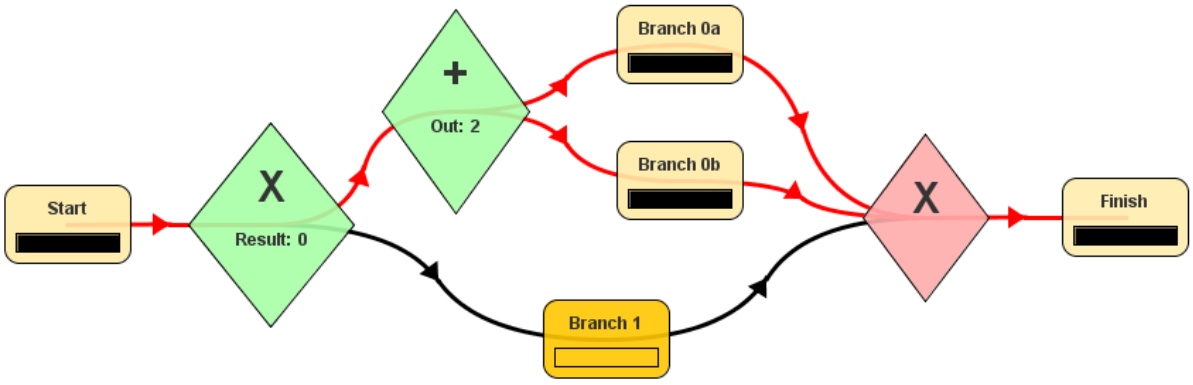

Abbildung 4.1: Ein nicht wohlgeformtes XOR-Konstrukt

Auf ein weiteres Problem mit nicht wohlgeformten Graphen und XOR-Splits wird beim OR-Join eingegangen.

4.3 Das OR-Gateway

Das OR-Gateway ist das mit Abstand am schwierigsten zu implementierende Gateway. Zuerst wird auf die allgemeinen Probleme des OR-Gateways hingewiesen. Anschließend werden verschiedene Implementierungsverfahren mit ihren Vor- und Nachteilen vorgestellt.

4.3.1 Der OR-Split

Beim Split muss als erstes durch ein geeignetes Verfahren entschieden werden, wie der Kontrollfluss weitergeleitet wird. Im Gegensatz zum XOR-Split können dabei beliebig viele Wege gewählt werden.

Bei dem hier verwendeten Verfahren ruft man für jeden Weg eine boolsche Entscheidefunktion auf. Wird bei einer Entscheidefunktion `true` zurückliefert, so wird dieser Weg

gestartet. Hierdurch ist es auch möglich, dass kein einziger Weg gestartet wird. Um dies zu vermeiden, kann einen Default-Weg definiert werden, der ausgeführt wird, wenn keine andere Entscheidefunktion `true` zurückgeliefert hat.

Damit auch hier, wie beim AND-Split, die nachfolgenden Pfade parallel ablaufen, wird hierbei für jeden ausgehenden Pfad ein neuer Thread erzeugt.

Hier ist eine einfache Implementierung des OR-Splits zu sehen:

```
1 public void run() {
2   for (int i = 1; i <= branches.size(); i++)
3     if (decisions.get(i).decide())
4       branches.get(i).run();
5 }
```

Damit der OR-Join funktioniert, muss diese Implementierung später noch erweitert werden.

4.3.2 Der OR-Join

Der OR-Join hat die Aufgabe, auf die vom OR-Split gestarteten Pfade zu warten. Erst wenn der Kontrollfluss auf all diesen Wegen am Join ankommt, darf der Nachfolger ausgeführt werden. Dies kann der Join allerdings nicht alleine entscheiden, da er lokal nicht wissen kann, auf welche Pfade er warten muss. Er benötigt dazu die Hilfe des Splits und entscheidet somit nicht lokal.

Wie diese Kommunikation zwischen Split und Join verlaufen kann, wird in den folgenden Abschnitten erklärt.

4.3.3 Ersetzen durch AND- und XOR-Gates

Die erste Möglichkeit das OR-Join-Problem zu lösen ist gar keine OR-Gateways zu verwenden. Statt dessen kann man sie durch AND- und XOR-Gateways ersetzen. Zwei Möglchkeiten dies zu realisieren werden im Folgenden erklärt.

Als Beispiel zur Umstrukturierung wird der in Abbildung 4.2 dargestellte Graph verwendet.

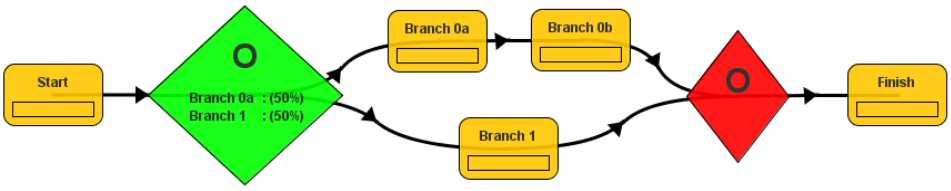

Abbildung 4.2: Der Beispielgraph für die Umstrukturierung

Ersetzen durch ein AND- und anschließend mehrere XOR-Gates

Bei diesem Verfahren wird anstatt des OR-Splits zuerst ein AND-Split erstellt. Dieser AND-Split besitzt genau so viele Ausgänge wie es mögliche Pfade gibt. Bei jedem dieser Pfade wird im Anschluss ein XOR-Split eingebaut. Dieser XOR-Split besitzt exakt zwei Ausgänge. Einer dieser Ausgänge führt zu einem möglichen Pfad. Am Ende dieses Pfades ist ein XOR-Join angebracht. Der andere Ausgang des XOR-Splits führt direkt zu dem gleichen XOR-Join. Alle XOR-Joins, welche sich am Ende der Pfade befinden, werden zum Schluss in einem AND-Join zusammengeführt. An diesen AND-Join wird der ursprüngliche Nachfolger des OR-Joins angehängt. In Abbildung 4.3 ist der oben abgedruckte Beispielgraph nach dieser Umstrukturierung zu sehen.

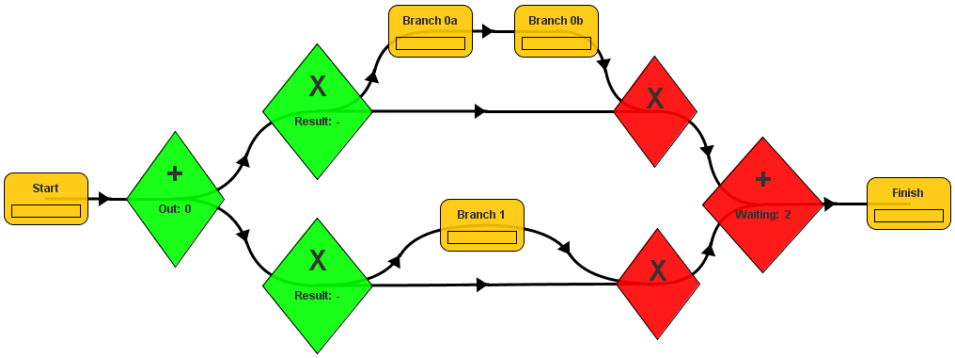

Abbildung 4.3: Der Beispielgraph nach der Umstrukturierung

Das Ersetzen der Gates auf diese Art und Weise bringt jedoch folgende Nachteile mit sich:

- Die Graphen sind nach der Umstrukturierung für den Menschen nicht mehr so einfach zu lesen.
- Man benötigt ungefähr doppelt so viele Gates. Dies führt zu einer langsameren Ausführung, sowie mehr Speicherbedarf.
- Es ist möglich, dass kein einziger Weg gewählt wird. Dennoch würde in diesem Fall der Nachfolgers des XOR-Joins ausgeführt werden. Dies ist in den meisten Fällen nicht erwünscht.

Das erste Problem lässt sich dadurch lösen, dass der Mensch bei der Modellieung weiterhin OR-Gateways verwendet und diese Gateways anschließend durch einen entsprechenden Algorithmus umgeformt werden. Hierzu muss der Mensch dem Algorithmus allerdings angeben, welcher OR-Join zu welchen OR-Split gehört. Dies löst allerdings nicht die beiden anderen Probleme.

Ersetzen durch ein XOR- und anschließend mehrere AND-Gates

Das nächste Verfahren zur Umstrukturierung ist etwas komplizierter. Hier wird der OR-Split als erstes durch einen XOR-Split ersetzt. Dieser XOR-Split besitzt für jede mögliche Pfadkombination einen Ausgang. An jedem dieser Ausgänge wird anschließend ein AND-Split angeschlossen. Entsprechend der Pfadkombination werden nun die Pfade und je ein AND-Join an die AND-Splits angeschlossen. Jeder Pfad muss am Ende ebenfalls auf den AND-Split führen. Da jeder Pfad somit auf mehrere AND-Joins führen muss, wird ein weiterer AND-Split an Ende jedes Pfades benötigt. Die AND-Joins werden am Schluss alle wieder in einem XOR-Join zusammengeführt. An diesen XOR-Join wird der ursprüngliche Nachfolger des OR-Joins angehängt. In Abbildung 4.4 ist der oben abgedruckte Beispielgraph nach dieser Umstrukturierung zu sehen.

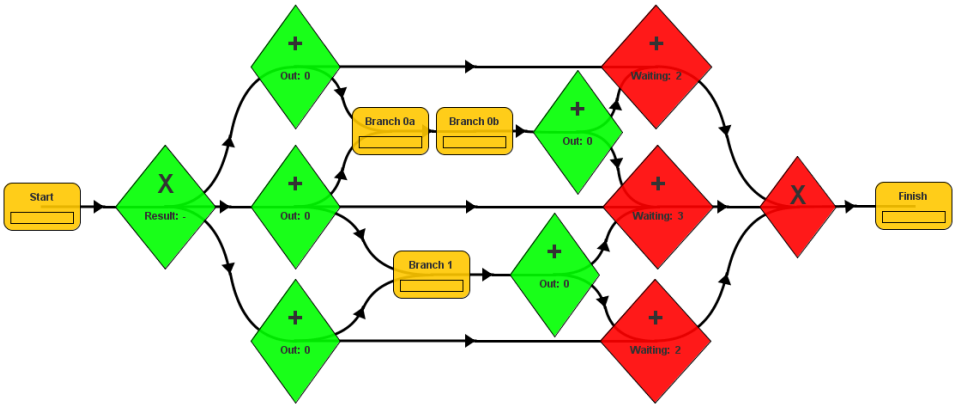

Abbildung 4.4: Der Beispielgraph nach der anderen Umstrukturierung

Bei diesem Verfahren ist es nun nicht mehr möglich, dass gar kein Pfad ausgeführt wird. Aber auch dieses Verfahren bringt Nachteile mit sich:

- Auch hier sind die Graphen nach der Umstrukturierung für den Menschen nicht mehr so einfach zu lesen.
- Die Anzahl der benötigten Gates steigt sehr schnell an. Die Anzahl der möglichen Pfadkombinationen kann man durch die Formel $2^n - 1$ (n = Anzahl der möglichen Pfade) ermitteln. Für jede Pfadkombination benötigt man ein AND-Split und ein AND-Join. Des Weiteren benötigt man am Anfang ein XOR-Split, am Ende ein XOR-Join und für jeden Pfad noch ein AND-Split. Die Gesamtanzahl der benötigten Gates berechnet sich dem zufolge mit der Formel $2 * (2^n - 1) + 2 + n \approx n^{n+1}$ und steigt somit exponentiell mit der Anzahl der Pfade an. Dies führt zu einem erheblichen Speicherbedarf. Bei 29 Pfaden werden bereits über eine Milliarde Gates benötigt. Dies ist von heutigen Rechnern kaum zu bewältigen.

Auch hier könnte ein entsprechender Algorithmus die Umstrukturierung automatisch bewerkstelligen, wodurch das zweite Problem allerdings keineswegs verschwindet.

Beide Verfahren zur Umstrukturierung bringen schwerwiegende Probleme mit sich und sind somit nur bedingt verwendbar. Aus diesem Grund werden nun Verfahren, welche ohne eine Umstrukturierung auskommen, vorgestellt.

Weitere Informationen zum Entfernen von OR-Joins aus Graphen sind in Mendling u. a. [2007] zu finden.

4.3.4 Zählen der Pfade

Ein Verfahren, bei dem keine Umstrukturierung nötig ist, sondern direkt mit den OR-Gateways gearbeitet wird, ist das Zählen der ausgehenden Pfade am OR-Split. Hierbei wird bei der Ausführung des OR-Splits gezählt, wie viele der möglichen Pfade aktiviert wurden. Diese Zahl wird dann dem OR-Join übermittelt. Dieser speichert die Zahl und dekrementiert sie jedes Mal, wenn ein Kontrollfluss am OR-Join ankommt. Sobald der Zähler Null erreicht, wird der Nachfolger des Joins ausgeführt. Eine mögliche Implementierung des Splits und des Joins könnten so aussehen:

```
1  class ORSplit {
2    ...
3    public void run() {
4      int c = 0;
5      for (int i = 0; i < branches.size(); i++)
6        if (decisions.get(i).decide()) {
7          branches.get(i).run();
8          c++;
9        }
10     join.setWaiting(c);
11   }
12   ...
13 }
14 class ORJoin {
15   ...
16   private int waiting;
17
18   public void setWaiting(int c) {
19     waiting = c;
20   }
21
22   public void run() {
23     waiting--;
24     if (waiting == 0)
25       next.run();
26   }
27   ...
28 }
```

Dieses Verfahren bringt einige Nachteile mit sich:

- Der OR-Split muss den OR-Join, mit dem die Pfade wieder zusammengeführt werden, kennen. Dies muss der Designer des Modells kennzeichnen.
- Jeder OR-Split muss zu genau einem OR-Join gehören.
- Der Graph muss wohlstrukturiert sein. Befindet sich ein AND-Split ohne einen dazugehörigen AND-Join zwischen den Gates, so können mehr Kontrollflüsse am OR-Join ankommen, als beim OR-Split erzeugt wurden. Dies ist in Abbildung 4.5

zu sehen. Der AND-Split erzeugt hier einen ausgehenden Kontrollfluss und übermittelt die Zahl Eins an den Split. Am Split kommen allerdings zwei Kontrollflüsse an. Bereits bei der Ankunft des Ersten wird der Nachfolger ausgeführt. Bei der Ankunft des zweiten Kontrollflusses wird der Zähler, wie in der Grafik zu sehen, auf -1 gesetzt.

Auch bei Aussprüngen, wie in Abbildung 4.5 zu sehen, hat dieses Verfahren Probleme. Hier werden zwei Kontrollflüsse am OR-Split erzeugt, allerdings kommt nur ein einziger am Join an. Der Zähler bleibt für immer auf Eins stehen und der Nachfolger wird niemals ausgeführt.

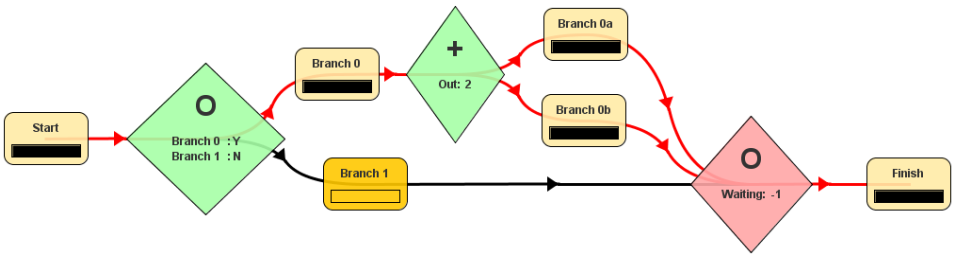

Abbildung 4.5: Zu viele Kontrollflüsse kommen am Join an.

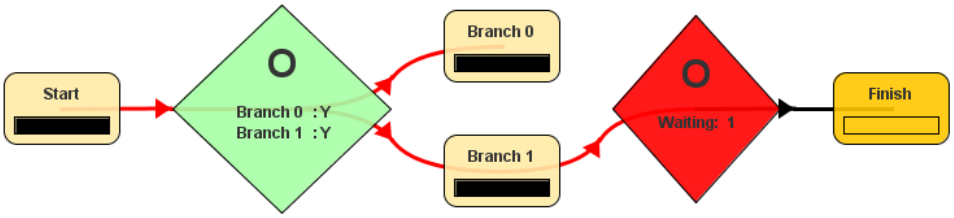

Abbildung 4.6: Zu wenig Kontrollflüsse kommen am Join an.

Das erste Problem bringt zwar zusätzliche Arbeit für den Designer mit sich, ist aber durchaus lösbar. Die beiden anderen Probleme lösen sich von selbst, wenn man nur wohlstrukturierte Graphen erlaubt.

4.3.5 Warten auf alle relevanten Pfade

Beim Warten auf alle relevanten Pfade wird der Graph analysiert. Dabei wird festgestellt, an welchen Eingängen des Joins ein Kontrollfluss ankommen kann. Die möglichen Knoten, von denen der Kontrollfluss am Join ankommen kann, werden in einer Liste der relevanten Pfade gespeichert. Jedes Mal wenn der Kontrollfluss am Join ankommt, wird der Vorgängerknoten aus der Liste der relevanten Pfade gestrichen. Sobal die Liste leer ist, wird der Nachfolger ausgeführt.

Im Folgenden werden zwei mögliche Algorithmen zur Erstellung dieser Liste vorgestellt.

Analysieren beim Durchlaufen des OR-Splits

Bei diesem Algorithmus werden beim Durchlaufen des OR-Splits alle aktiven Ausgänge analysiert. Dabei wird der Graph ausgehend von jedem Ausgang traversiert. Sobald bei der Traversierung ein Knoten von Typ OR-Join gefunden wird, wird der letzte traversierte Knoten in die Liste der relevanten Pfade beim Join eingetragen und die Traversierung abgebrochen. Eine mögliche Implementierung könnte so aussehen:

```
1  class ORSplit {
2    ...
3    public void run(Knoten k) {
4      for (int i = 0; i < branches.size(); i++)
5        if (decisions.get(i).decide()) {
6          traverse(branches.get(i), 0);
7          branches.get(i).run();
8        }
9    }
10
11   void traverse(Knoten knoten) {
12     List<Knoten> s = knoten.getNachfolger();
13     for (Knoten k:s) {
14       if (k instanceof PathOrJoin)
15         ((PathOrJoin)k).addRelevantPath(knoten);
16       else
17         traverse(k);
18     }
19   }
20   ...
21 }
22 class ORJoin {
23   ...
24   private List<Knoten> waiting;
25   private boolean active = false;
26
27   public void addRelevantPath(Knoten k) {
28     waiting.add(k);
29   }
30
31   public void removeRelevantPath(Knoten k) {
32     waiting.remove(k);
33     if (active && (waiting.size() == 0))
34       next.run();
35   }
36
37   public void run(Knoten k) {
38     waiting.remove(k);
39     active = true;
40     if (waiting.size() == 0)
41       next.run();
42   }
43   ...
44 }
```

Bei dieser Implementierung darf der Graph allerdings keine Zyklen aufweisen, da die Methode `traverse` sonst unbegrenzt weiterlaufen würde.

Die Abbildungen 4.7, 4.8 und 4.9 verdeutlichen das Verfahren. Das erste Bild zeigt den Zustand des Graphen direkt nach dem Durchlaufen des OR-Splits. Der Graph wurde an jedem Ausgang traversiert. Dabei wurden die relevanten Knoten „Branch 0b“ und „Branch 1“ im Join, wie in der roten Raute zu sehen, eingetragen. Das zweite Bild zeigt den Zustand nachdem der Kontrollfluss von „Branch 1“ am Join angekommen ist. Der

entsprechende Eintrag wurde beim Join entfernt. Dieser wartet nun nur noch auf den „Branch 0b“. Nachdem dieser auch am Join angekommen ist, wird, wie in der letzten Darstellung zu erkennen, die Nachfolgeaktion Finish ausgeführt.

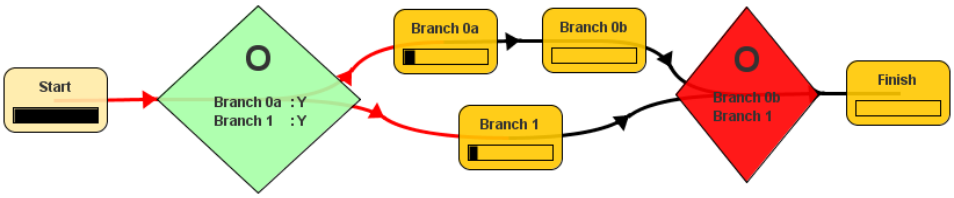

Abbildung 4.7: Der Zustand des Graphen direkt nach Durchlaufen des OR-Splits.

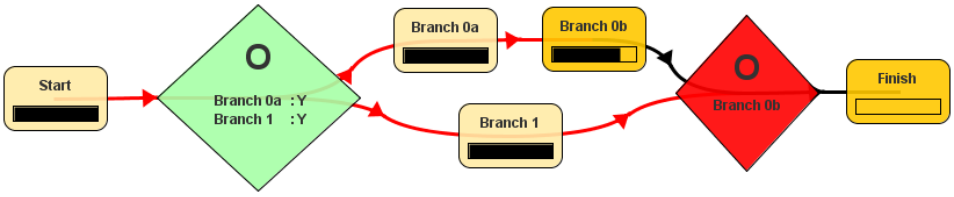

Abbildung 4.8: Der Zustand des Graphen nach Eintreffen des ersten Kontrollflusses am OR-Join.

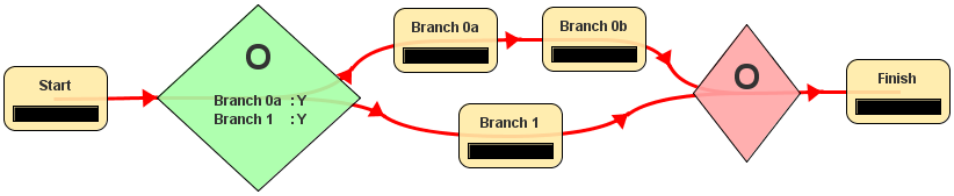

Abbildung 4.9: Der Zustand des Graphen nach Eintreffen des zweiten Kontrollflusses am OR-Join.

Dieses Verfahren bringt einige Vorteile gegenüber dem Zählen der ausgehenden Pfade mit sich. Der erste Vorteil ist, dass der Designer nicht mehr den zum Split gehörenden Join kennzeichnen muss. Der Join wird bei der Traversierung automatisch gefunden. Des Weiteren ist es durchaus möglich, dass ein Split auf mehrere Joins führen kann.

Allerdings ist das Problem der nicht wohlgeformten Graphen hierdurch nicht gelöst. Es ist zwar nun möglich, Aussprünge und nicht geschlossene AND-Splits zwischen den OR-Gates zu haben, dies gilt aber nicht für XOR-Splits. Das Problem wird in Abbildungen 4.10 verdeutlicht. Nach dem Durchlaufen des OR-Splits werden die Knoten „Branch 1-0“, „Branch 1-1“, „Branch 2-0“ und „Branch 2-1“ beim Join eingetragen. Es kommen aber niemals alle vier Kontrollflüsse am Join an, da der XOR-Split sich immer für einen Pfad entscheiden muss. Da der XOR-Split sich allerdings erst entscheidet, wenn er durchlaufen wird, kann der OR-Split nicht im Voraus entscheiden, welchen der beiden Knoten „Branch 2-0“ und „Branch 2-1“ er beim Join als relevant eintragen soll. Bei

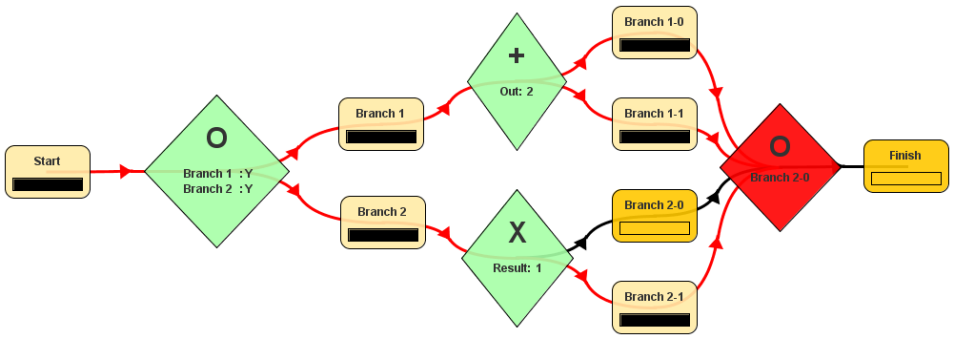

Abbildung 4.10: Das Verfahren hat Probleme mit nicht geschlossenen XOR-Gates.

nicht geschlossenen OR-Splits innerhalb eines OR-Gateways kann das Problem ebenfalls auftreten.

Schließt man das XOR-Gate jedoch ordnungsgemäß, so funktioniert der OR-Join. Dies ist in Abbildung 4.11 zu sehen.

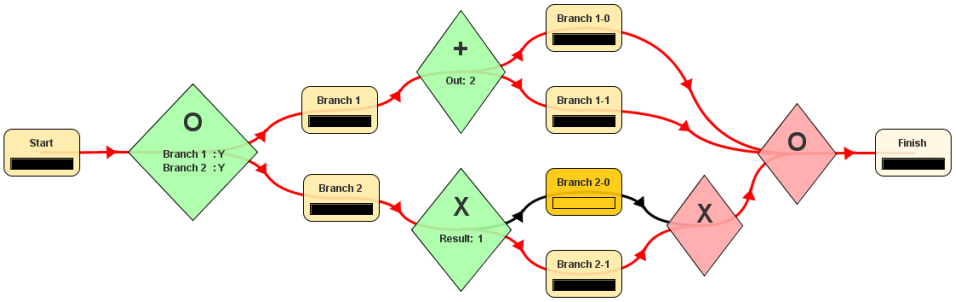

Abbildung 4.11: Mit geschlossenem XOR-Gate funktioniert das Verfahren.

Es gibt aber durchaus auch Verfahren, bei denen es erlaubt ist, XOR-Splits innerhalb des OR-Gateways zu öffnen und nicht mehr zu schließen. Dazu muss der XOR-Split beim Durchlaufen den nicht mehr relevanten Pfad vom OR-Join wieder entfernen. Dies wird im nächsten Abschnitt beschrieben.

Analysieren vor dem Start der Simulation

Beim Analysieren vor dem Start der Simulation wird der Graph vor dem Start der Simulation nach OR-Splits durchsucht. Wie beim vorherigen Verfahren werden die Nachfolger jedes gefundenen OR-Splits traversiert. Auch hier werden alle relevanten Pfade beim OR-Join eingetragen. Somit weiss jeder OR-Join bereits vor dem Start der Simulation auf welchen Pfaden der Kontrollfluss ankommen kann.

Wird nun ein OR-Split oder ein XOR-Split durchlaufen, so werden alle nicht aktiven ausgehenden Pfade traversiert. Bei jedem hierbei gefundenem OR-Join wird dann der

Vorgägner aus der Liste der relevanten Pfade ausgetragen. Auf diese Weise werden nicht mehr relevante Pfade beim OR-Join ausgetragen. Dies ermöglicht es zwischen OR-Split und OR-Join nicht geschlossene OR- und XOR-Splits einzubauen. Die Implementierung ähnelt sehr der Methode `traverse` aus dem vorherigen Abschnitt. Der Unterschied liegt darin, dass nicht die Methode `addRelevantPath`, sondern `removeRelevantPath` beim gefundenen OR-Join aufgerufen wird. Wichtig ist ebenfalls, dass der XOR-Split hierzu modifiziert werden muss, da dieser ebenfalls den Graphen traversieren muss.

Die Abbildung 4.12 zeigt das Problem aus dem vorherigen Abschnitt vor dem Eintreffen des ersten Kontrollflusses an OR-Join. Vor dem Beginn der Simulation wurden die Knoten „Branch 1-0“, „Branch 1-1“, „Branch 2-0“ und „Branch 2-1“ beim Join eingetragen. Beim Durchlaufen des XOR-Splits wurde der Knoten „Branch 2-0“ aus der Liste der relevanten Pfade beim OR-Join entfernt. Der OR-Join wartet somit nur noch auf die drei wirklich ankommenden Pfade und funktioniert somit einwandfei in dieser Situation.

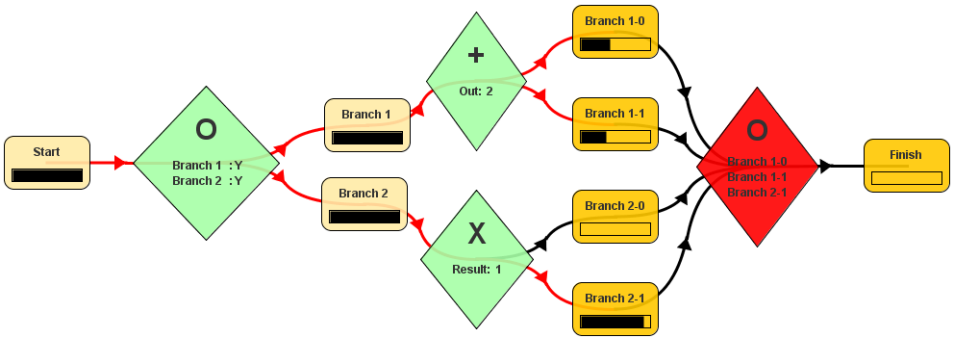

Abbildung 4.12: Dieses Verfahren hat keine Probleme mit nicht geschlossenen XOR-Gates.

Wird bei der Ausführung eins OR- oder XOR-Splits der letzte relevante Pfad entfernt, so muss der Nachfolger des OR-Joins ausgeführt werden, vorrausgesetzt es kam bereits ein Kontrollfluss am OR-Join an. Ein Beispiel hierfür wird in Abbildung 4.13 gezeigt. Im Unterschied zum vorherigen Beispiel führt der „Branch 2-1“ nun ins Leere. Entscheidet sich der XOR-Split nun für dieses Pfad, so wird der letzte Eintrag aus dem OR-Join entfernt. Da dieser OR-Join bereits durch „Branch 1-0“ und „Branch 1-1“ aktiviert wurde, muss nun der Nachfolgeknoten „Finish“ ausgeführt werden. Dies geschieht, indem sich der OR-Join beim Ankommen eines Kontrollflusses merkt, dass er soeben aktiviert wurde. Wird ein Pfad entfernt, überprüft der Join nun, ob er bereits aktiviert wurde und ob die Liste leer ist. Ist beides der Fall, so wird der Nachfolger gestartet.

Aber auch dieses Verfahren bringt Nachteile mit sich. Ein solches Problem wurde in Abbildung 4.14 abgebildet. Vor dem Start der Simulation wurde der Graph analysiert und beim OR-Join wurden die Knoten „Branch 1-0“, „Branch 2-0“ und „Branch 3“ eingetragen. Beim Durchlaufen des XOR-Splits wurde der aus Sicht des XOR-Splits nicht aktive Pfad traversiert. Somit wurde der Knoten „Branch 3“ aus der Liste der relevanten Pfade gelöscht, obwohl dieser Pfad durchaus vom AND-Split gestartet wurde. Der OR-

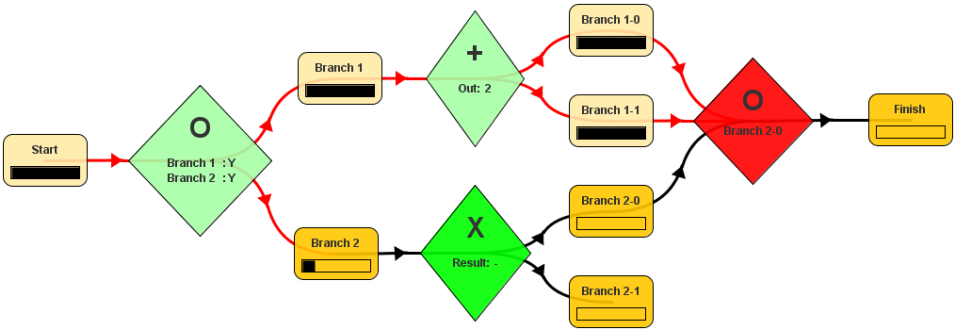

Abbildung 4.13: Dieses Verfahren hat keine Probleme mit ins Nichts führenden Pfaden.

Join wartet somit nur noch auf „Branch 1-0“ und „Branch 2-0“. Sobald diese Beiden angekommen sind wird ohne Rücksicht auf „Branch 3“ der Nachfolger ausgeführt.

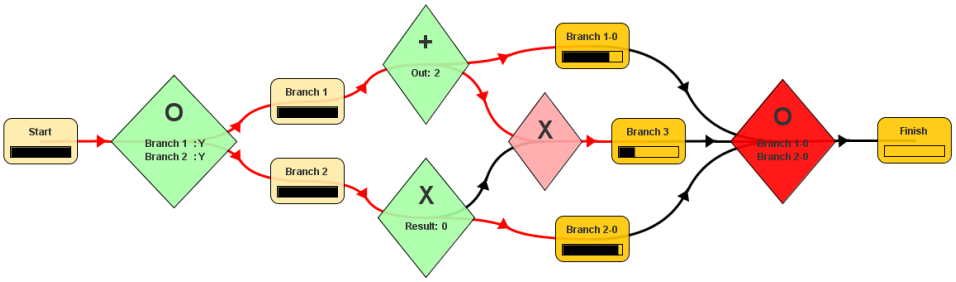

Abbildung 4.14: Dieses Verfahren bringt neue Probleme mit sich.

Dieses Problem könnte gelöst werden, indem ein Knoten beim Analysieren der relevanten Pfade mehrmals in die Liste eingetragen wird, wenn der Kontrollfluss auf verschiedenen Wegen am Join ankommen kann. So würde hier der „Branch 3“ zweimal in der Liste enthalten sein. Auf diese Art wäre es kein Fehler, sondern das richtige Verhalten, den Knoten beim Durchlaufen des untern XOR-Splits einmal zu entfernen.

4.3.6 Warten auf Beendigung der Threads

Das letzte hier vorgestellte Verfahren wird als „Warten auf Beendigung der Threads“ bezeichnet. Hierbei wird ein vollständig neuer Ansatz verwendet. Die gesamte Arbeit wird auf den OR-Split übertragen. Der OR-Join hat keine Aufgabe und kann sogar komplett entfallen.

Der OR-Split erzeugt bei der Ausführung für jeden aktiven ausgehenden Pfad einen neuen Thread und startet in diesem Thread den entsprechenden Nachfolger. Anschließend wartet der OR-Split bis alle Threads beendet wurden. Danach startet er den Nachfolgerknoten des früheren OR-Joins.

Dazu muss der Graph umgestellt werden. Die Abbildungen 4.15 und 4.16 zeigen das schon mehrmals verwendete Beispiel aus Abbildung 4.2 mit dem hier beschriebenen Verfahren. In der ersten Abbildung ist der Zustand des Graphen kurz nach der Ausführung des OR-Joins zu sehen. Dabei wurden zwei Threads gestartet. Im ersten Thread läuft der obere, in anderen Thread der mittlere Kontrollfluss. Beide Kontrollflüsse werden in absehbarer Zeit am Ende ankommen und somit terminieren. Nachdem die beiden Threads somit beendet wurden, führt der OR-Split den Nachfolgeknoten „Finish“ aus.

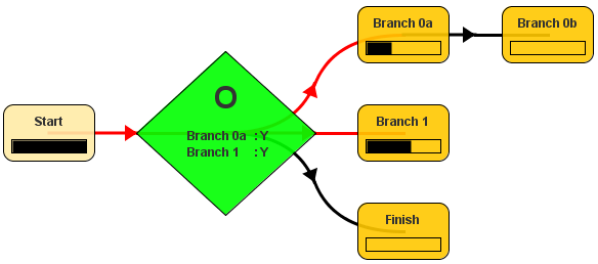

Abbildung 4.15: Der Zustand des Graphen kurz nach der Ausführung des OR-Joins.

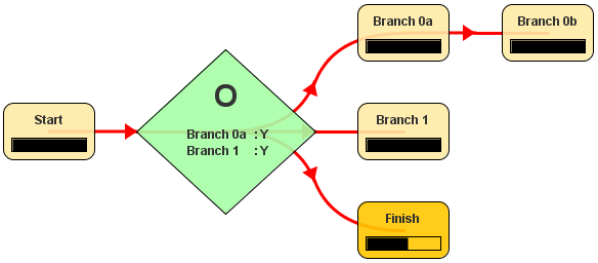

Abbildung 4.16: Der Zustand des Graphen kurz nach Beendigung der beiden Threads.

Dieses Verfahren kann besonders einfach mit der Hilfe des Executor-Frameworks realisiert werden. Eine mögliche Implementierung könnte wie folgt aussehen:

```
1 public void run() {
2   ExecutorService e = Executors.newCachedThreadPool();
3   for (int i = 0; i < branches.size(); i++)
4     if (decisions.get(i).decide())
5       e.execute( branches.get(i));
6   e.shutdown();
7   e.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
8   next.run();
9 }
```

Da die Darstellung des OR-Gates ohne OR-Join bei der Entwicklung des Modells eher irreführend ist, kann für die Darstellung zusätzlich ein OR-Join eingebaut werden. An diesen Join werden dann die Enden der Pfade angeschlossen. Kommt der Kontrollfluss allerdings an diesen Enden an, so wird nicht der OR-Join ausgeführt, sondern der ausführende Thread beendet. Nach der Beendigung aller Threads startet der OR-Split nun den Nachfolger des OR-Joins. Der OR-Join übernimmt weiterhin keine Funktion und dient nur der Darstellung. Der Graph würde nach dieser Umformung wieder exakt wie

in Abbildung 4.2 aussehen. Der Designer der Modells müsste hierbei allerdings wieder den zum OR-Split gehörenden OR-Join auf irgendeine Weise markieren. Weiterhin kann immer nur exakt ein Join zu einem Split gehören.

Dieses Verfahren ist sehr robust gegen nicht wohlgeformte Graphen. Es können beliebige AND- und XOR-Gates zwischen OR-Split und OR-Joins vorkommen. Die einzige Bedingung ist, dass die Kontrollflüsse terminieren. Dies ist allerdings bei Graphen mit Zyklen nicht immer gegeben.

Weiterhin kann es zu einer verzögerten Ausführung des Nachfolgeknoten des Joins kommen, wie in Abbildung 4.17 gezeigt wird. Hier sind bereits alle nötigen Kontrollflüsse am OR-Join angekommen, der Knoten „Finish“ könnte also bereits ausgeführt werden. Allerdings wartet der OR-Split immer noch auf die Beendigung des obersten Kontrollflusses. Da dies allerdings noch einige Zeit in Anspruch nehmen kann, wird eventuell viel Zeit vergeudet.

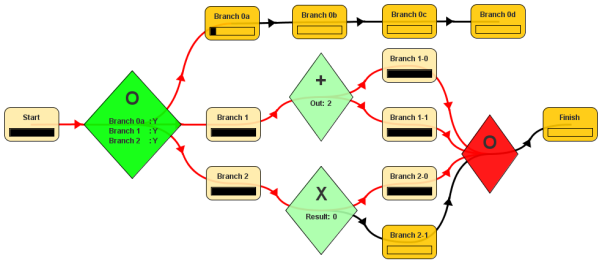

Abbildung 4.17: Die Ausführung von „Finish“ kann sich lange verzögern.

4.3.7 Verbinden eines AND-Splits mit einem OR-Join

Ein weiteres, bis jetzt vernachlässigtes Problem, ist, dass in den OR-Join Kontrollflüsse eingehen können, die nicht von einem OR-Split erzeugt wurden. Die Abbildung 4.18 zeigt einen OR-Join, in den drei Kontrollflüsse eingehen. Zwei davon stammen von einem OR-Split, der Unterste hingegen wurde von einem AND-Split erzeugt. Der OR-Join sollte somit immer auf den unteren Kontrollfluss, sowie auf jeden vom OR-Split gestarteten Kontrollfluss warten. Hierbei scheitern allerdings alle hier vorgestellten Verfahren. Hier ist eine kurze Beschreibung der Probleme und mögliche Lösungsansätze.

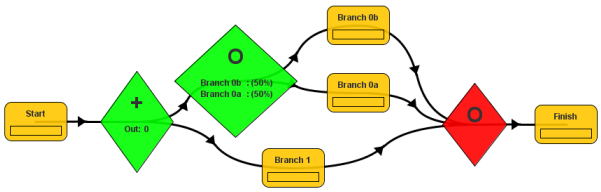

Abbildung 4.18: Ein AND-Splits wurde mit einem OR-Join verbunden.

- Beim Zählen der Pfade übergibt der Split dem Join die Anzahl der ausgehenden Pfade. Es kommt aber immer einer mehr am Join an. Hierfür ist keine Lösung bekannt.
- Beim Analysieren beim Durchlaufen des OR-Splits erfährt der Join von Split nur Informationen über die oberen beiden Kontrollflüsse. Über den Unteren erhält er keinerlei Informationen. Um dies zu vermeiden, müssten auch alle AND-Splits und XOR-Splits die ausgehenden Pfade traversieren und relevante Pfade bei OR-Joins eintragen.
- Beim Analysieren vor dem Start der Simulation existiert das gleiche Problem. Auch hier müssten alle AND-Splits und XOR-Splits die ausgehenden Pfade traversieren und relevante Pfade bei OR-Joins eintragen.
- Da beim Warten auf Beendigung der Threads der OR-Join keine Funktion übernimmt wird der untere Kontrollfluss einfach ignoriert. Hierfür ist ebenfalls keine Lösung bekannt.

Wie man hier sieht, ist das Problem mit den OR-Joins durchaus nicht auf OR-Splits begrenzt. Auch alle anderen Splits müssen entsprechend angepasst werden.

5 A Vicious Circle

Bei allen bisherigen Problemen bei der Implementierung gab es theoretisch eine richtige Lösung. Dies ist aber nicht immer der Fall. Im Folgenden wird eine Situation beschrieben, in der es keine vernünftige Möglichkeit für die Ausführung des Graphen gibt. Der entsprechende Graph ist in Abbildung 5.1 abgedruckt.

Bei diesem Graphen wird zu Beginn der Kontrollfluss auf zwei Pfade aufgeteit. Beide Pfade besitzten einen OR-Join und einen OR-Split. Im Unterschied zu allen vorherigen Graphen befindet sich der Join hierbei allerdings vor dem Split. Der jeweilige Join muss vor dem Split ausgeführt werden. Des Weiteren ist der untere Split mit dem oberen Join verbunden und der oberere Split mit dem unteren Join. Hierdurch wartet der obere Join auf den unteren Split. Dieser wartet auf den unteren Join, welcher auf den oberen Split wartet, welcher wiederum auf den oberen Join wartet. Oder kurz gesagt: Der obere Join wartet auf sich selbst. Diese Situation könnte man auch als Deadlock bezeichnen.

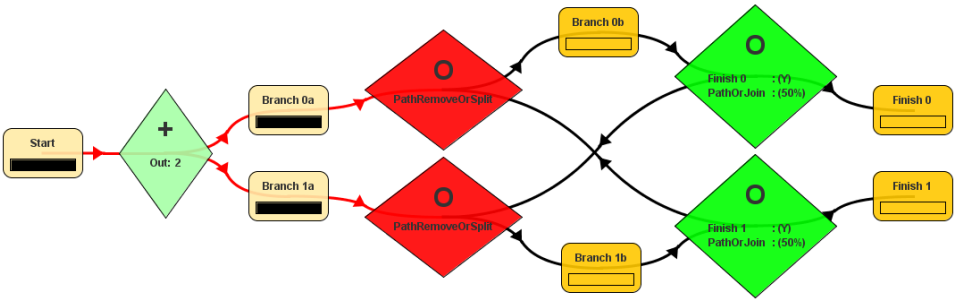

Abbildung 5.1: A Vicious Circle.

Es gibt keine ideale Möglichkeit diesen Deadlock aufzulösen. Eine Lösung wäre, einen der beiden Joins entgegen den Regeln auszuführen, obwohl nicht alle Kontrollflüsse angekommen sind. Diese Möglichkeit steht allerdings nicht zur Verfüfung, falls der Join zur Ausführung Informationen von dem Split benötigt.

Eine andere Lösung wäre, einen der beiden Kontrollflüsse abzubrechen. Würde beispielsweise der untere Kontrollfluss abgebrochen, so kann der untere Split nicht mehr ausgeführt werden, wodurch der obere Join ausgeführt werden kann. In diesem Fall müsste man einen eventuellen Datenverlust hinnehmen.

Weitere Informationen zum Vicious-Circle-Problem sind in van der Aalst u. a. [2002] und Kindler [2006] zu finden.

6 Fazit

Wie dem Leser der letzten Abschnitte vermutlich bewusst wurde, handelt es sich bei der logischen Oder-Zusammenführung um ein nicht triviales Problem. Es wurden mehrere Verfahren zur Implementierung dieser Zusammenführung vorgestellt. Jedoch war keines dieser Verfahren ohne Nachteile.

Das Ersetzten der OR-Gates durch AND- und XOR-Gates ermöglicht es, Graphen komplett ohne OR-Gates zu entwerfen. Diese sind allerdings unübersichtlich und benötigen bei vielen Pfaden schnell sehr viele Gates.

Beim „Zählen der Pfade“ hat man Probleme mit nicht geschlossenen AND-Splits, aber nicht mit OR-Splits. Im nächsten Verfahren, dem „Analysieren beim Durchlaufen des OR-Splits“, wird das Problem der AND-Splits gelöst. Allerdings bestehen nun Schwierigkeiten mit nicht geschlossenen OR-Splits. Dies wurde beim „Analysieren vor dem Start der Simulation“ gelöst. Dieses Verfahren ist sehr robust und kann in den meisten Situationen eingesetzt werden.

Das letzte Verfahren, „Warten auf Beendigung der Threads“, verwendet einen ganz neuen Ansatz und hat mit den Poblemen der vorherigen Techniken nicht zu kämpfen. Hierdurch entstehen allerdings erneut ganz neue Schwierigkeiten.

Des Weiteren wurde im Abschnitt „A Vicious Circle“ gezeigt, dass unter bestimmten Umständen gar keine vernünftige Lösung existiert.

Dies alles macht deutlich, dass die logische Oder-Zusammenführung nicht unterschätzt werden sollte. Die meisten Probleme lösen sich von selbst, wenn man Regeln zur Wohlgeformtheit des Graphen aufstellt. Dies macht nicht nur die Implementierung des OR-Joins einfacher, sondern das Modell wird auch leichter verständlich.

Literaturverzeichnis

[van der Aalst u. a. 2002] Aalst, W.M.P. van d. ; Desel, J. ; Kindler, E.: On the semantics of EPCs: A vicious circle. In: *Proceedings of the EPK*, 2002, S. 71–80

[Gruhn u. Laue 2005] Gruhn, V. ; Laue, R.: Einfache EPK-Semantik durch praxistaugliche Stilregeln. In: *EPK 2005* (2005), S. 176

[Kindler 2006] Kindler, E.: On the semantics of EPCs: Resolving the vicious circle. In: *Data & Knowledge Engineering* 56 (2006), Nr. 1, S. 23–40

[Mendling u. a. 2007] Mendling, J. ; Dongen, BF van ; Aalst, WMP van d.: Getting rid of the or-join in business process models. In: *11th IEEE International Enterprise Distributed Object Computing Conference, 2007. EDOC 2007*, 2007, S. 3–3

[Russell u. a. 2006] Russell, N. ; Ter Hofstede, A.H.M. ; Aalst, W.M.P. van d. ; Mulyar, N.: Workflow control-flow patterns: A revised view. In: *BPM Center Report BPM-06-22, BPMcenter. org* (2006), S. 06–22

[Van Der Aalst u. a. 2003] Van Der Aalst, W.M. ; Ter Hofstede, A.H. ; Kiepuszewski, B. ; Barros, A.P.: Workflow Patterns. (2003)