3. Erweiterungen des Basismodells der Diskreten-Ereignis-Simulation

Die Diskrete-Ereignis-Simulation ist in der Praxis weit verbreitet. Sie wurde in Kapitel 2 mit Hilfe eines Basismodells formalisiert, wobei das Basismodell im Wesentlichen dem Stand der Forschung bzw. der in der Praxis verwendeten Diskreten-Ereignis-Simulation entspricht.

Bei bloßer Verwendung des Basismodells ist es einem Modellierer oft erschwert, in einfacher und natürlicher Art und Weise Modelle zu erstellen, um eine Simulation ausführen zu können. So ist es vor allem für die Modellierung komplexer Systeme nur bedingt geeignet. Deshalb werden hier Erweiterungen des Basismodells der Diskreten-Ereignis-Simulation aus Kapitel 2 vorgenommen, die diese Nachteile ausgleichen sollen. Dabei werden in den Abschnitten 3.1, 3.2 und 3.3 als drei mögliche Erweiterungen der objektorientierte Systemzustand, die Unterscheidung zwischen exogenen Ereignissen und Folgeereignissen sowie die Ersetzung der Angabe von altem und neuem Zustand durch Angabe von Zustandsbedingung und Zustandseffekt vorgestellt. In Abschnitt 3.4 wird dann das Prinzip der auf der Diskreten-Ereignis-Simulation aufbauenden *Objektbasierten Simulation* vorgestellt, das diese drei Erweiterungen in sich vereinigt.

Der Name *Objektbasierte Simulation* wurde hier gewählt, da der objektorientierte Systemzustand, also die Aufteilung des Systemzustands auf mehrere Objekte eine zentrale Erweiterungen des Basismodells ist und, da der Name kurz und prägnant ist. Wenn man aktive Entitäten als Agenten und passive Entitäten als Objekte bezeichnet, wäre es exakter, hier von *Entitätsbasierter Simulation* zu sprechen, da keine Unterscheidung zwischen aktiven und passiven Entitäten vorgenommen wird. Treffend wäre auch der Name *Erweiterte Diskrete-Ereignis-Simulation* gewesen, der keine der drei Erweiterungen in den Vordergrund gestellt hätte.

3.1 Objektorientierter Systemzustand

Oftmals unterteilt sich ein System in mehrere Systembestandteile, das sind (aktive und passive) Objekte, die jeweils einen eigenen Zustand haben. Als Objekte kommen dabei in sich abgeschlossene Teile des simulierten Systems, die eine Schnittstelle zum System haben, in Betracht. Das können reale Objekte, aber auch Software-Objekte sein. Im Basismodell der Diskreten-Ereignis-Simulation ist eine hierarchische Struktur des Zustands nicht vorgesehen, sondern der Systemzustand ist eine Zusammenfassung der Zustandsinformationen (z.B. der diskreten und stetigen Größen oder der Zustandsaussagen der einzelnen Objekte).

In der hier vorgestellten Erweiterung wird der Systemzustand als Menge von Objektzuständen betrachtet. Ein Objekt verfügt dabei über

- einen Typ t
- eine innerhalb dieses Typs eindeutige ID i
- einen (vom Typ abhängigen) Zustand S.

Dabei kann der Zustand S auch komplexe Attributwerte beinhalten.

Das Objekt vom Typ *t* mit ID *i* bezeichnet man auch als *t[i]*. Den Typ eines Objekts *o* bezeichnet man ferner als *o.Typ,* seinen Zustand als *o.S* und seine ID als *o.ID*.

Bei der ID handelt es sich um eine natürliche Zahl, die einen Primärschlüssel für den Typ darstellt. Anders als die Objekt-ID in objektorientierten Programmiersprachen wie z.B. Java ist diese ID für den Anwender (Modellierer bzw. Programmierer) sichtbar. Dies ist nötig, da ein Modellierer innerhalb der Reaktionsregeln möglicherweise direkt auf ein bestimmtes Objekt zugreifen will, dessen ID er selbst innerhalb des Anfangszustandes angegeben hat.

Da in den meisten Fällen genau ein Objekt auf ein auftretendes Ereignis reagiert, wird in dieser Erweiterung das Systemverhalten in Form von Reaktionsregeln angegeben, die besagen, welches Objekt auf welches Ereignis wie reagiert und welche Folgeereignisse erzeugt werden. Bei der Angabe des alten und des neuen Zustandes wird dabei nur der Zustand dieses Objekts angegeben, so dass man, anders bei der Angabe der Transitionsfunktion f im Basismodell nicht explizit angeben muss, dass sich der Zustand der anderen Objekte nicht ändert.

3.2 Unterscheidung zwischen exogenen Ereignissen und Folgeereignissen

In der Diskreten-Ereignis-Simulation wird bei (zufalls-)periodischen Ereignissen meistens das nächste Auftreten eines Ereignisses als Folgeereignis des aktuellen Ereignisses modelliert, da die Periodizität so technisch leicht handhabbar ist. Tatsächlich sind die einzelnen Ereignisse meist unterschiedliche, voneinander unabhängige Instanzen des selben Ereignistyps, so dass eine Modellierung als Folgeereignis unnatürlich ist, weil man damit zum Ausdruck bringen würde, dass ein Auftreten eines Ereignisses das nächste logisch bedingt. Außerdem es bei dieser Art der Modellierung lästig, bei jeder Regel eines Ereignisses das nächste Auftreten des Ereignisses als Folgeereignis angeben zu müssen, was zu einer erhöhten Fehleranfälligkeit führt, da dies auch bei einzelnen Regeln vergessen werden kann.

Ein Nachteil des Basismodells der Diskreten-Ereignis-Simulation ist also, dass es keine Unterscheidung von verschiedenen Ereignistypen wie *exogene Ereignisse* und (echte) Folgeereignisse gibt.

Die vorkommenden Ereignisse lassen sich in dieser Erweiterung in zwei Gruppen unterteilen:

- exogene Ereignisse
- Folgeereignisse (die sich aus ausgeführten Aktionen von Objekten ergeben)

Bei Page [Pag91] werden Folgeereignisse auch als *endogene Ereignisse* bezeichnet.

Exogene Ereignisse sind Ereignisse, die in (zufalls-)periodischen Abständen auftreten, ohne dass ein Objekt diese auslöst. Es handelt sich hierbei um Ereignisse, deren Ursache außerhalb des simulierten Systems liegt und die von System somit nicht beeinflussbar sind. In dieser Erweiterung des Basismodells können Ereignisse als *exogene Ereignisse* deklariert werden, wobei es für einen Typ eines exogenen Ereignisses mehrere Stränge geben kann, die jeweils für sich in (zufalls-)periodischen Abständen auftreten und sich durch ihre Parameter unterscheiden können. Für jeden Strang wird eine (ggf. vom Zufall abhängige) Periodizität angegeben. In der Simulation werden diese Ereignisse dann in den angegebenen Abständen erzeugt. Um eine terminierende Simulation zu erreichen, kann man für einen Strang eines exogenen Ereignisses eine Stopbedingung angeben, bei deren Erfüllung das aktuelle Ereignis nicht mehr stattfindet und die Erzeugung weiterer Ereignisse unterdrückt wird.

In dieser Erweiterung werden exogene Ereignisse unter Angabe einer Periodizität und einer Stopbedingung spezifiziert.

3.3 Zustandsbedingung und Zustandseffekt

Im Basismodell der Diskreten-Ereignis-Simulation ist man stets gezwungen, den alten Zustand, der vor Anwendung einer Regel gelten muss sowie den neuen Zustand, der nach Anwendung der Regel gilt, exakt anzugeben. Möglich ist es jedoch, Teile des Zustands, deren Wert für die Regel ohne Bedeutung ist, durch Unterstriche oder Pünktchen zu kennzeichnen oder freie Variablen zu verwenden, die dann wiederum bei der Beschreibung des neuen Zustands verwendet werden können.

Oftmals will man aber komplexe logische Bedingungen angeben, unter denen die Regel gilt. Außerdem ist es einfacher für den Modellierer, wenn er nur die Teile des neuen Zustands angeben muss, die sich aufgrund der Regel geändert haben.

In dieser Erweiterung kann man deshalb die Angabe des alten Zustands durch Angabe einer *Zustandsbedingung* und die Angabe des neuen Zustands durch die Angabe eines *Zustandseffekts* ersetzen. Als Zustandsbedingung (alternativ als *Zustandsvorbedingung* zu bezeichnen) kann der geforderte alte Zustand oder eine Bedingung, die für den alten Zustand gelten muss, stehen. Spiegelbildlich dazu kann bei Zustandseffekt (alternativ als *Zustandsnachbedingung* zu bezeichnen) der neue Zustand oder eine Bedingung, die für den neuen Zustand gelten muss, stehen.

Eine Regel r kann dann durch eine Regeldefinition r (VB, e) = (NB, E'), bei der VB eine konkrete Zustandsbedingung (Zustandsvorbedingung), NB ein konkreter Zustandseffekt (Zustandsnachbedingung), e ein konkretes Ereignis und E' eine konkrete Ereignismenge ist dargestellt werden. Der Übersichtlichkeit halber bietet sich folgende tabellarische Form an:

Ereignis	**e**
Zustandsbedingung	**VB**
Zustandseffekt	**NB**
Folgeereignisse	**E'**

Formal kann man dieses Konzept wie folgt darstellen.

Sei *State* die Menge aller Systemzustände, $L_{StateCond}$ eine Sprache für (Systemzustands-) Bedingungsformeln sowie $>$ eine Inferenzrelation mit $> \subseteq (State \times L_{StateCond})$, wobei $S > F$ bedeutet, dass die Formel F für den Zustand S wahr ist. Dann gibt es eine Aktualisierungsfunktion *Upd* mit dem Typ Upd: $(State \times L_{StateCond}) \rightarrow State$, die einem Zustand S und einer Nachbedingung F einen Zustand S' zuweist, so dass $S' > F$.

Existiert eine Regeldefinition r (VB, e) = (NB, E'), so ergibt sich für die Transitionsfunktion f aus dem Basismodell eine Funktionsdefinition f (S, e) = (Upd (S, NB) , E'), falls S > VB.

Es ist im Allgemeinen sinnvoll, Aktualisierungsfunktionen zu definieren, die den ursprünglichen Zustand mit minimaler Veränderung in einen Zustand überführen, in dem die Nachbedingung gilt. Insbesondere soll die Aktualisierungsfunktion den Zustand unverändert lassen, wenn die Nachbedingung schon im ursprünglichen Zustand galt. Die Angabe des ursprünglichen Zustands als Argument ermöglicht solche Aktualisierungsfunktionen. Nur durch solche Aktualisierungsfunktionen wird es bei Angabe der Nachbedingung möglich, die sich nicht verändernden Teile nicht anzugeben, ohne unspezifiziert zu lassen, dass sie sich nicht ändern sollen.

Im allgemeinen Fall ist es nicht notwendig eindeutig, welcher Folgezustand der Zustand ist, bei dem sich der Zustand minimal ändert. Besteht z.B. der Zustand aus einem Tupel von natürlichen Zahlen (a, b), so ist Upd ((1, 1) , a + b > 3) durch die Forderung nach minimaler Änderung nicht hinreichend beschrieben. Ist hingegen der Zustand ein Tupel von Zustandsvariablen und sind die Nachbedingungen nur Konjunktionen von konkreten Werten einzelner Zustandsvariablen, so ist eindeutig, wie der Zustand mit minimaler Veränderung aussieht. Besteht z.B. der Zustand aus einem Quintupel von natürlichen Zahlen (a, b, c, d, e), so ist Upd ((1, 2, 3, 4, 5) , (a = 7 $\wedge$ c = 3 $\wedge$ e = 8)) = (7, 2, 3, 4, 8). Man kann hier also die Gleichungen aus der Nachbedingung auch als imperative Wertzuweisungen auffassen. Ebenfalls eindeutig ist die minimale Zustandsänderung, wenn der Zustand aus einer Menge atomarer Aussagen besteht und die Nachbedingung nur eine atomare Aussage oder eine Konjunktionen atomarer Aussagen enthält. Sei etwa S = { .. , p (a) , .. }, dann ergibt sich für die Aktualisierungsfunktion Upd (S, p (b)) = S \ { p (a) } $\cup$ { p (b) }.

Bei einer Simulationsspezifikation kann man nur dann Zustandsbedingungen und Zustandseffekte spezifizieren, wenn entweder die Aktualisierungsfunktion *Upd* explizit angegeben wird oder sich in natürlicher Weise eine Aktualisierungsfunktion ergibt.

Die Angabe eines alten Zustandes *a* und eines neuen Zustands *n* für einen Zustand *S* kann auch als Spezialfall der Angabe von Zustandsbedingungen und Zustandseffekte angesehen werden, wobei die Zustandsbedingung $S = a$ und der Zustandseffekt $S = n$ ist. Tut man dies, so spricht man auch von unechten Zustandsbedingungen und unechten Zustandseffekten, ansonsten von echten Zustandsbedingungen und echten Zustandseffekten. Auch bei unechten Zustandsbedingungen kann es mehrere Zustände geben, für die eine Regel gilt, wenn ein Ausdruck angegeben wird, der freie Variablen enthält.

Verwendet man echte Zustandsbedingungen und Zustandseffekte, so bietet sich zur Formulierung die zu UML [OMG04a] gehörende Sprache OCL an. Allerdings kann auch jede formale oder natürliche Sprache, die die Bedingungsformeln eindeutig beschreibt, verwendet werden.

3.3.1 Formulierung von Zusicherungen

Oftmals gibt es Bedingungen, die zum Zeitpunkt der Ausführung eines Ereignisses immer erfüllt sind, d.h., wenn immer ein Ereignis eines bestimmten Typs auftritt, ist aufgrund der Gesamtheit der

spezifizierten Reaktionsregeln klar, dass der Zustand des Systems die Eigenschaft hat, dass diese Bedingung gilt.

Man kann nun in einem Simulationssystem zu jedem Ereignistyp eine solche Bedingung explizit formulieren. Dies tut man in Form von Zusicherungen (Assertions[34]). Es ist klar, dass man durch die explizite Angabe von Zusicherungen die Spezifikation inhaltlich nicht verändert, denn man hat ja sowieso die Regeln schon so formuliert, dass die Bedingungen gelten. Allerdings sind durch die Angabe von Zusicherungen die Regeln für einen Leser eventuell leichter verständlich, insbesondere, wenn bei ihrer Formulierung ihre Gültigkeit implizit angenommen wurde. Man kann auf diese Art und Weise sogar nicht nur die einzelnen Regeln, sondern sogar das gesamte Simulationssystem leichter verstehen. Der Zusammenhang zwischen der Formulierung von Zusicherungen und der Formulierung von Regeln wird in Abschnitt 3.3.2 näher erläutert.

Durch die Formulierung von Zusicherungen ist es möglich, die Einhaltung der zugesicherten Bedingung beim Ablauf der Simulation explizit zu überprüfen, um mögliche Fehler in der Spezifikation des simulierten Systems (oder auch Fehler in der Implementierung des Simulationssystems) zu finden[35].

3.3.2 Formulierung von mehreren Regeln

Spezifiziert man zu einem Ereignistyp mehrere Reaktionsregeln, so wird die erste Regel ausgewählt, deren Zustandsbedingung erfüllt ist, was voraussetzt, dass eine totale Ordnung auf den zu einem Ereignistyp definierten Regeln existiert[36].

Die Bedingungen der Regeln so zu formulieren, dass sie bei mehreren Regeln gleichzeitig wahr sein können, hat Vor- und Nachteile. Vorteil ist, dass man die Regeln nicht notwendig so formulieren muss, dass alle Bedingungen der vorhergehenden Regeln ausgeschlossen werden. Nachteil ist, dass man eine Regel dann nicht mehr für sich, sondern nur im Kontext der vorhergehenden Regeln verstehen kann.

Wie im Folgenden gezeigt wird, kann man jede Folge von Regeln in eine äquivalente Folge von Regeln so umformulieren, dass die Bedingungen der einzelnen Regeln sich gegenseitig ausschließen.

Seien für einen Ereignistyp n Regeln $r_1, r_2, .., r_n$ mit Bedingungen $b_1, b_2, .., b_n$ und sei z die für diesen Ereignistyp spezifizierte Zusicherung, so wird Regel r_i genau dann ausgewählt, wenn $b_i \wedge z \wedge \bigwedge_{j=1}^{i-1} \neg b_j$ gilt, d.h., wenn ihre Bedingung erfüllt ist, wenn die Zusicherung erfüllt ist und wenn die Bedingungen aller vorhergehenden Regeln nicht erfüllt sind. Damit sichergestellt ist, dass es immer eine Regel gibt, die ausgewählt werden soll, muss immer mindestens ein b_i erfüllt sein, wenn z gilt. Anders ausgedrückt dürfen nur dann alle b_i unerfüllt sein, wenn auch z nicht eingehalten wurde, also $\neg b_1 \wedge \neg b_2 \wedge .. \wedge \neg b_n \Rightarrow \neg z$.

Oft ist eine Bedingung einfacher zu formulieren, wenn man ausnutzt, dass die Bedingung einer Regel nur dann ausgewertet wird, wenn die Bedingungen aller vorstehenden Regeln nicht zugetroffen haben. Man kann also innerhalb der Bedingung voraussetzen, dass alle anderen Bedingungen nicht zugetroffen haben. Außerdem kann man voraussetzen, dass die spezifizierte Zusicherung erfüllt ist[37].

---

[34] Die Idee der Assertions in einem Simulationssystem entspricht den Assertions in einer imperativen Programmiersprache.

[35] Hierdurch kann man zwar beweisen, dass ein Fehler vorliegt, wenn man eine zugesicherte Bedingung nicht erfüllt ist. Einen Beweis, dass eine Spezifikation fehlerfrei ist, kann man so allerdings nicht führen.

[36] Man könnte sich auch Simulationsparadigmen vorstellen, in denen keine Ordnung auf den zu einem Ereignistyp festgelegten Regeln existiert. Dann wäre man aber gezwungen, die Bedingungen in den Regeln so zu formulieren, dass nie mehrere von ihnen gleichzeitig wahr sein können. Eine Alternative wäre, festzulegen, wie das Simulationssystem damit umgehen soll, wenn mehrere Regeln gleichzeitig wahr sind, z.B. eine zufällige Regel auswählen.

[37] Dies sollte man nur tun, wenn das System die spezifizierten Zusicherungen automatisch überprüft oder wenn man davon überzeugt ist (etwa durch einen Beweis), dass das System korrekt spezifiziert ist.

Wenn man alle Bedingungen b_i durch alternativ formulierte Bedingungen b_i' ersetzt, derart dass $b_i' \equiv b_i \vee \neg z \vee \bigvee_{j=1}^{i-1} b_j$, so sind b_i und b_i' äquivalent, d.h. Regel r_i wird nach der Modifizierung der Bedingung immer noch genau in den Fällen ausgewählt, in denen sie vorher auch ausgewählt wurde.

Beweis:

Zu zeigen ist, dass b_i' und b_i in den gleichen Fällen ausgewählt werden, also dass die Äquivalenz $b_i' \wedge z \wedge \bigwedge_{j=1}^{i-1} \neg b_j \Leftrightarrow b_i \wedge z \wedge \bigwedge_{j=1}^{i-1} \neg b_j$ gilt.

Dies ist der Fall wegen:

$$b_i' \wedge z \wedge \bigwedge_{j=1}^{i-1} \neg b_j \equiv (b_i \vee \neg z \vee \bigvee_{j=1}^{i-1} b_j) \wedge z \wedge \bigwedge_{j=1}^{i-1} \neg b_j \equiv$$

$$(b_i \wedge z \wedge \bigwedge_{j=1}^{i-1} \neg b_j) \vee (\neg z \wedge z \wedge \bigwedge_{j=1}^{i-1} \neg b_j) \vee (\bigvee_{j=1}^{i-1} b_j \wedge z \wedge \bigwedge_{j=1}^{i-1} \neg b_j) \equiv$$

$$(b_i \wedge z \wedge \bigwedge_{j=1}^{i-1} \neg b_j) \vee \mathit{false} \vee \mathit{false} \equiv (b_i \wedge z \wedge \bigwedge_{j=1}^{i-1} \neg b_j) .\blacksquare$$

Es ergibt sich, dass b_n' immer *true* sein muss.

Beweis:

$$b_n' \equiv b_n \vee \neg z \vee \bigvee_{j=1}^{n-1} b_j \equiv \bigvee_{j=1}^{n} b_j \vee \neg z \equiv b_1 \vee b_2 \vee .. \vee b_n \vee \neg z \equiv \neg b_1 \wedge \neg b_2 \wedge .. \wedge \neg b_n \Rightarrow \neg z$$

Letzteres ist aber genau die Forderung, die an die b_i gestellt wurde und somit immer *true*. ■

Man kann sogar viel allgemeinere Umformulierungen von Regelbedingungen vornehmen, ohne dass sich an der Regelauswahl etwas ändert. Dies tut man, indem man jedes b_i durch eine beliebige Bedingung $b_i'' \equiv b_i \vee h_i$ ersetzt, wobei h_i eine Hilfsbedingung ist mit $h_i \Rightarrow \neg z \vee \bigvee_{j=1}^{i-1} b_j$. Setzt man $h_i \equiv \mathit{false}$, so ist $b_i'' \equiv b_i$, d.h. man lässt die Formulierung der Regelbedingung unverändert. Setzt man $h_i \equiv \neg z \vee \bigvee_{j=1}^{i-1} b_j$, so ergibt sich $b_i'' \equiv b_i'$. Der Beweis, dass auch bei dieser Modifizierung der Bedingung eine Regel immer noch genau in den Fällen ausgewählt, in denen sie vorher auch ausgewählt wurde, ist ähnlich zum obigen Beweis und wird hier ausgelassen.

In den nachfolgenden Beispielen wird klar, wie man durch geschickte Wahl von h_i einfachere Bedingungen formulieren kann.

Sei $z \equiv 0 < x \leq 8$ und seien $b_1 \equiv 6 < x \leq 8$, $b_2 \equiv 4 < x \leq 6$ und $b_3 \equiv 0 < x \leq 4$. Mittels $h_1 \equiv x > 8$, $h_2 \equiv x > 6$ und $h_3 \equiv (x > 4) \vee (x \leq 0)$ ergeben sich $b_1'' \equiv x > 6$, $b_2'' \equiv x > 4$ und $b_3'' \equiv \mathit{true}$.

Seien p_1, p_2 und p_3 Prädikate und sei $z \equiv p_1 \vee (p_2 \wedge p_3)$. Für die Bedingungen $b_1 \equiv p_1 \wedge p_2$, $b_2 \equiv (p_2 \wedge \neg p_1) \vee (p_3 \wedge (p_2 \vee p_1))$ und $b_3 \equiv p_1 \wedge \neg p_2 \wedge \neg p_3$ kann man $h_1 \equiv \mathit{false}$, $h_2 \equiv (p_1 \wedge p_2) \vee (\neg p_1 \wedge \neg p_2 \wedge p_3)$ und $h_3 \equiv \neg p_1 \vee p_2 \vee p_3$ setzen, womit sich $b_1'' \equiv p_1 \wedge p_2$, $b_2'' \equiv p_2 \vee p_3$ und $b_3'' \equiv \mathit{true}$ ergeben.

Wie man am 2. Beispiel sieht, ist es oft schwieriger, geeignete h_i explizit zu ermitteln als direkt einfachere Bedingungen zu suchen.

Wenn man die Regelbedingungen so formulieren will, dass sie sich gegenseitig ausschließen, damit jede Regel unabhängig vom Kontext der anderen Regeln verständlich ist, muss man die Transformation der Regeln in die entgegengesetzte Richtung vornehmen. Die einzelnen b_i müssen also durch $b_i \wedge z \wedge \bigwedge_{j=1}^{i-1} \neg b_j$ ersetzt werden.

3.4 Objektbasierte Simulation

In der Objektbasierten Simulation wird das Basismodell der Diskreten-Ereignis-Simulation um die drei vorgestellten Erweiterungen angereichert. Diese sind ein objektorientierter Systemzustand, die Unterscheidung zwischen exogenen Ereignissen und Folgeereignissen und die Angabe von Zustandsbedingungen und Zustandseffekten. Alle drei Erweiterungen sind logisch unabhängig voneinander, so dass man jede beliebige Teilmenge der Erweiterungen verwenden könnte, um das Basismodell zu erweitern. Die Objektbasierte Simulation kann als die maximale[38] nicht-agentenbasierte Erweiterung des Basismodells der Diskreten Ereignis-Simulation angesehen werden.

Die folgende informelle Grafik (Abbildung 6) veranschaulicht den Weg vom Basismodell der Diskreten-Ereignis-Simulation zur Objektbasierten Simulation grafisch.

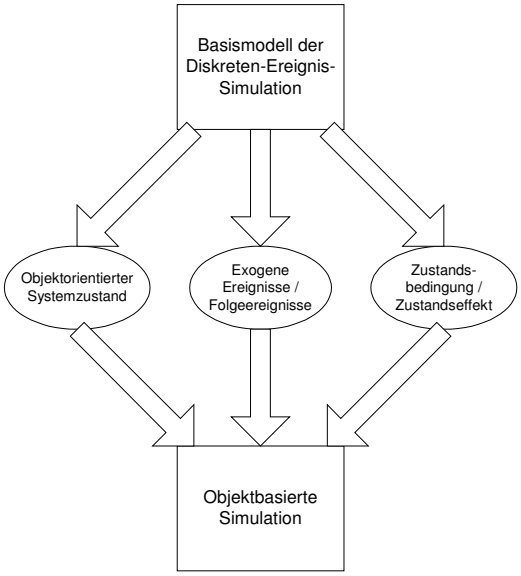

Abbildung 6: Der Weg vom Basismodell der Diskreten-Ereignis-Simulation zur Objektbasierten Simulation

Auf den ersten Blick scheinen der objektorientierte Systemzustand und die Angabe von Zustandsbedingungen und Zustandseffekten verwandt zu sein, da beide es ermöglichen, dass man sich nicht verändernde Teile des Systemzustands nicht explizit angeben muss. Verwendet man jedoch nur den objektorientierten Systemzustand, so müssen unveränderte Teile innerhalb des Objektzustands weiterhin mit angegeben werden. Verwendet man hingegen nur die Angabe von Zustandsbedingungen und Zustandseffekten, so kann man die einzelnen Zustandsvariablen nur mit einem globalen Namen bezeichnen und nicht mit einem lokalen Namen innerhalb eines Objektes. Außerdem ginge dann die Strukturierungsmöglichkeit, die ein objektorientierter Systemzustand bietet, verloren.

In Abschnitt 3.4.1 wird das Konzept der Objektbasierten Simulation präzisiert und mathematisch formalisiert. Um zu zeigen, dass die Objektbasierte Simulation tatsächlich eine Verfeinerung der Diskreten-Ereignis-Simulation ist, wird in Abschnitt 3.4.2 gezeigt, wie man ein mit Hilfe der Objektbasierten Simulation spezifiziertes Modell in ein mit Hilfe der Diskreten-Ereignis-Simulation spezifiziertes Modell transformiert. Abschnitt 3.4.3 enthält das bereits aus Abschnitt 2.2 bekannte Beispiel einer Warteschlange in einem Supermarkt diesmal mit Hilfe der Objektbasierten Simulation spezifiziert. In Abschnitt 3.4.4 wird auch das aus Abschnitt 2.3 bekannte Fahrstuhl-Beispiel mit Hilfe der Objektbasierten Simulation spezifiziert.

[38] Maximal bedeutet hier maximal bezüglich der vorgestellten Erweiterungen. Das soll nicht heißen, dass keine weiteren nicht-agentenbasierten Erweiterungen denkbar wären.

3.4.1 Formalisierung

In der Objektbasierten Simulation wird das simulierte System als Menge von Objekten angesehen, die jeweils über einen eigenen Zustand verfügen.

Eine Reaktionsregel besteht aus

- einem (Ereignis-)Typ des die Regel anstoßenden Ereignisses (Trigger)
- einer Bedingung für das Triggerereignis, unter dem die Regel gilt (Triggerbedingung)
- einem (Objekt-)Typ sowie die ID[39] des auf das Triggerereignis reagierenden Objekts (Reaktor)
- einer Bedingung für den Zustand des Reaktors, unter dem die Regel gilt (Zustandsbedingung)
- einer Menge von Reaktionen auf das Ereignis.

Als Reaktionen auf das Ereignis kommen in Betracht:

- eine Veränderung des Zustands des Reaktors, ausgedrückt in Form eines Zustandseffekts
- erzeugte Folgeereignisse (die sich z.B. aus ausgeführten Aktionen des Objekts ergeben).

Eine Reaktionsregel kann in der nachfolgenden tabellarischen Form dargestellt werden.

Trigger	
Triggerbedingung	
Reaktor	
Zustandsbedingung	
Zustandseffekt	
Folgeereignisse	

Eine Veränderung des Zustands anderer Objekte kommt nicht in Betracht. Das widerspräche der Aufteilung der Reaktionsregeln auf einzelne Objekte. Da es jedoch nicht immer vorteilhaft ist, ein System so zu formulieren, dass bei jeder Reaktionsregel nur ein Objekt seinen Zustand ändert, ist die Verwendung von globalen Regeln zulässig. Bei diesen entfällt die Angabe des Reaktors. Die Zustandsbedingung sowie der Zustandseffekt werden bei globalen Regeln aus globaler Sicht und nicht aus lokaler Sicht des Reaktors formuliert. Eine globale Regel kann in der nachfolgenden tabellarischen Form dargestellt werden.

Trigger	
Triggerbedingung	
Zustandsbedingung	
Zustandseffekt	
Folgeereignisse	

Zu jedem Ereignistyp kann eine Zusicherung angegeben werden, die bei Eintritt des Ereignisses immer erfüllt ist. Für die Zusicherung bietet sich an, die gleiche Sprache zu verwenden wie für die Zustandsbedingungen und Zustandseffekte.

[39] Im Allgemeinen wird hier keine konkrete ID verwendet, sondern eine, die sich aus den Parametern des Triggerereignisses ergibt.

Eine Zusicherung zu einem Ereignistyp kann in der nachfolgenden tabellarischen Form dargestellt werden.

Ereignis	
Zusicherung	

Stränge von exogenen Ereignissen werden unter Angabe einer Periodizität und einer Stopbedingung spezifiziert.

Ein Strang eines exogenen Ereignisses kann in der nachfolgenden tabellarischen Form dargestellt werden.

exogenes Ereignis	
Periodizität	
Stopbedingung	

3.4.2 Abbildung auf die Diskrete-Ereignis-Simulation

Um zu zeigen, dass die Objektbasierte Simulation tatsächlich eine Verfeinerung der Diskreten-Ereignis-Simulation ist, wird in diesem Abschnitt gezeigt, wie man ein mit Hilfe der Objektbasierten Simulation spezifiziertes Modell in ein mit Hilfe der Diskreten-Ereignis-Simulation spezifiziertes Modell transformiert.

Der Systemzustand in der Diskreten-Ereignis-Simulation ist ein Tupel, das aus den Zuständen der Objekte in der Objektbasierten Simulation besteht.

Eine Reaktionsregel entspricht einer Regel für die Transitionsfunktion f, bei der

- der Typ und die Bedingung für das Triggerereignis spezifiziert werden
- das Reaktorobjekt bestimmt und auf seine Zustandsbedingung abgefragt wird
- der Folgezustand die Veränderungen des Zustands des Reaktors berücksichtigt und die Zustände aller anderen Objekte unverändert lässt
- erzeugte Folgeereignisse enthalten sind.

Alle exogenen Ereignisse aus der Objektbasierten Simulation werden in die initiale Ereignismenge E_0 der Diskreten-Ereignis-Simulation aufgenommen. Alle Regeln für die Transitionsfunktion f, die auf Reaktionsregeln beruhen, die exogene Ereignisse als Triggerereignis haben, müssen modifiziert werden. In ihnen muss das nächste (zufalls-)periodische Auftreten aus dem Strang des exogenen Ereignisses als Folgeereignis hinzugefügt werden. Dies darf aber nur dann geschehen, wenn die Stopbedingung noch nicht erfüllt ist. Ist sie erfüllt, so darf kein Folgeereignis erzeugt werden und die Regel muss den Zustand unverändert lassen.

Für die Ersetzung des alten Zustands durch eine Zustandsbedingung und des neuen Zustands durch einen Zustandseffekt gibt es die folgende Rückführung auf das Basismodell der Diskreten-Ereignis-Simulation. Die Zustandsbedingung löst man auf, indem man die Regel für alle konkreten alten Zustände dupliziert[40], die diese Zustandsbedingung erfüllen. Der konkrete neue Zustand ergibt sich, indem man die Aktualisierungsfunktion Upd auf den alten Zustand und den Zustandseffekt anwendet.

3.4.3 Beispiel: Warteschlange im Supermarkt

Das bereits in Abschnitt 2.2 beschriebene Beispiel einer Warteschlange im Supermarkt wird nun mit Hilfe der Objektbasierte Simulation spezifiziert. Um das Lesen zu erleichtern, werden die Erläuterungen zu den Regeln aus Abschnitt 2.2 bei den Erläuterungen in diesem Abschnitt wiederholt.

[40] Sofern es möglich ist, Teile des Zustands, deren Wert für die Regel ohne Bedeutung ist, durch Unterstriche oder Pünktchen zu kennzeichnen oder freie Variablen zu verwenden, müssen Regeln nicht dupliziert werden.

Die Benennung der Reaktionsregeln stimmt mit der dortigen Benennung überein. Außerdem wird die Nummer der äquivalenten Regel aus Abschnitt 2.2 angegeben.

Bei den Zustandsbedingungen und den Zustandeffekten handelt es sich jeweils um den vollständigen Zustand des Objekts. Auf die Möglichkeit, echte Zustandsbedingungen und Zustandseffekte zu formulieren, wurde also verzichtet.

3.4.3.1 Ereignistypen und Systemzustand

Zu unterscheiden sind nun nur noch die zwei Ereignistypen *Kunde_kommt_an* und *Bedienung_beendet*. Der Ereignistyp *Eingang_wird_geschlossen* diente in der Diskreten-Ereignis-Simulation nur dazu, das Erzeugen weiterer Ereignisse vom Typ *Kunde_kommt_an* zu unterbinden. Dies geht in der Objektbasierten Simulation mit Hilfe der Stopbedingung exogener Ereignisse, so dass der Ereignistyp *Eingang_wird_geschlossen* nicht mehr benötigt wird.

Das System kann man als Menge von Kassen beschreiben, so dass hier die n Kassen die Objekte darstellen. Da das Simulationsende mit Hilfe der Stopbedingung exogener Ereignisse erreicht werden kann, ist im Gegensatz zur Spezifikation in der Diskreten-Ereignis-Simulation kein Eingang-Objekt erforderlich. Den Zustand einer Kasse kann man durch die Länge l der Warteschlange sowie den Belegungszustand s der Kasse (kann die Werte *frei* oder *belegt* annehmen) beschreiben.

Für alle Kassen ergibt sich für den Zustand $S = (l , s)$ in natürlicher Weise der Startzustand $S_0 = (0 , frei)$, denn am Anfang sind alle Warteschlangen leer und die Kassen frei. Gestartet wird zum Zeitpunkt $c_0 = 0$. Es gibt einen Typen exogener Ereignisse, die Ankunft eines Kunden. Dieser Typ hat einen einzelnen Strang.

exogenes Ereignis	`Kunde_kommt_an ( ) @ t`
Periodizität	`Expo_ZV(`μ`)`
Stopbedingung	`t >` t_{max}

3.4.3.2 Ereignisausführung Kunde_kommt_an

Das Ereignis *Kunde_kommt_an* stellt die Ankunft eines neuen Kunden dar.

Für den Zustand des Systems gibt es beim Auftreten des Ereignisses keine Einschränkungen, also ist die Zusicherung *true*[41].

Ereignis	`Kunde_kommt_an ( ) @ t`
Zusicherung	–

Zusicherung Kunde_kommt_an

Kommt ein neuer Kunde an, so geht er zu der freien Kasse mit der kleinsten Nummer, falls eine solche existiert. Damit gilt die nachfolgende Reaktionsregel, in der i den Index der kleinsten freien Kasse darstellt. Der Zustand ändert sich nur bei s, das auf *belegt* gesetzt wird. Erzeugt wird ein Folgeereignis, das Ereignis, dass die Bedienung an Kasse i, die der Kunde jetzt belegt, später beendet sein wird. (Äquivalent: Regel 1)

[41] Meistens in der tabellarischen Form durch einen Strich dargestellt.

Trigger	**Kunde_kommt_an ( ) @ t**
Triggerbedingung	–
Reaktor	**Kasse [i], wobei i = min { j | Kasse [j].S = (0, *frei*) }**
Zustandsbedingung	**(0, *frei*)**
Zustandseffekt	**(0, *belegt*)**
Folgeereignisse	**Bedienung_beendet (i) @ (t + Gleich_ZV(a, b)) }**

Regel 26: Kunde_kommt_an.1

Zur Verdeutlichung:

In OCL könnte man für die gleiche Regel Zustandsbedingung und Zustandseffekt wie folgt formulieren.

Zustandsbedingung	**(l = 0) AND (s = *frei*)**
Zustandseffekt	**(s = *belegt*)**

Ist keine Kasse frei, stellt sich der Kunde bei der kürzesten Schlange an, wobei bei gleicher Länge die Schlange der Kasse mit der kleinsten Nummer gewählt wird. Dabei stellt i den Index der Kasse mit der kürzesten Schlange dar. Der Zustand ändert sich nur bei l, das um 1 erhöht wird. (Äquivalent: Regel 2)

Trigger	**Kunde_kommt_an ( ) @ t**
Triggerbedingung	–
Reaktor	**Kasse [i], wobei i = min { j | ∀ k (1≤k≤n): Kasse [j].S.l ≤ Kasse [k].S.l) }**
Zustandsbedingung	**(l, *belegt*)**
Zustandseffekt	**(l+1, *belegt*)**
Folgeereignisse	**Ø**

Regel 27: Kunde_kommt_an.2

Die Regel *Kunde_kommt_an.3* (Regel 3) , die es in Abschnitt 2.2 gab, ist hier nicht mehr erforderlich, da durch die Stopbedingung die Erzeugung von Ankünften von Kunden, die nicht mehr bedient werden, unterdrückt wird.

3.4.3.3 Ereignisausführung Bedienung_beendet

Das Ereignis *Bedienung_beendet* stellt die Beendigung der Bedienung eines Kunden an einer Kasse dar. Als Parameter hat *Bedienung_beendet* die Nummer der Kasse.

Wenn immer ein Ereignis *Bedienung_beendet* stattfindet, so ist die Kasse, an der die Bedienung jetzt beendet ist, vorher belegt.

Ereignis	**Bedienung_beendet (i) @ t**
Zusicherung	**Kasse [i].S.s = *belegt***

Zusicherung Bedienung_beendet

Ist die Bedienung eines Kunden an der Kasse abgeschlossen und ist die Warteschlange nicht leer, so kommt der nächste Kunde aus der Warteschlange an die Kasse. Die Länge der Warteschlange der

Kasse verkürzt sich also um 1. Erzeugt wird ein Folgeereignis: das Ereignis, dass die Bedienung an Kasse i, die der erste Kunde aus der Warteschlange jetzt belegt, später beendet sein wird. (Äquivalent: Regel 4)

Trigger	**Bedienung_beendet (i) @ t**
Triggerbedingung	**–**
Reaktor	**Kasse [i]**
Zustandsbedingung	**(l+1, *belegt*)**
Zustandseffekt	**(l, *belegt*)**
Folgeereignisse	**{ Bedienung_beendet (i) @ (t + Gleich_ZV(a, b))}**

Regel 28: Bedienung_beendet.1

Ist die Warteschlange hingegen leer, so wird die Kasse jetzt frei. Es wird in diesem Fall kein Folgeereignis erzeugt. (Äquivalent: Regel 5)

Trigger	**Bedienung_beendet (i) @ t**
Triggerbedingung	**–**
Reaktor	**Kasse [i]**
Zustandsbedingung	**(0, *belegt*)**
Zustandseffekt	**(0, *frei*)**
Folgeereignisse	**Ø**

Regel 29: Bedienung_beendet.2

3.4.4 Beispiel: Fahrstühle

Auch das bereits in Abschnitt 2.3 beschriebene Beispiel der Fahrstühle wird nun mit Hilfe der Objektbasierten Simulation spezifiziert.

Um das Lesen zu erleichtern, werden auch hier die Erläuterungen zu den Regeln aus Abschnitt 2.3 bei den Erläuterungen in diesem Abschnitt wiederholt. Außerdem wird die Nummer der äquivalenten Regel aus Abschnitt 2.3 angegeben. Die Benennung der Regeln entspricht der dortigen Benennung, so dass sich Lücken in der Nummerierung innerhalb eines Ereignistyps ergeben können, wenn dort spezifizierte Regeln in der Objektbasierten Simulation nicht mehr benötigt werden oder wenn mehrere Regeln zu einer zusammengefasst werden können.

Bei den Zustandsbedingungen und den Zustandeffekten sowie bei den Zusicherungen wird die Sprache OCL benutzt.

3.4.4.1 Ereignistypen und Systemzustand

Die folgenden Ereignistypen kommen auch hier vor: *Kommt_an*, *Wird_angefordert*, *Will_fahren*, *Fährt_an*, *Erhält_Erlaubnisanfrage* und *Erhält_Erlaubnis*.

Der Zustand des Systems besteht aus den Zuständen der beiden Fahrstühle, den beiden einzigen Objekten. Wie schon in der Diskreten-Ereignis-Simulation beinhaltet der Zustand jedes Fahrstuhls, ob er der Bevorrechtigte ist, ob er sich vorgemerkt hat, dass der andere Fahrstuhl auf eine Erlaubnis von ihm wartet, ob er selbst auf eine Erlaubnis des anderen Fahrstuhls wartet, welche Anforderungen er erhalten hat, ob diese Anforderungen von außen kamen, ob er steht oder fährt und an welchem Ort er sich befindet (wenn er steht) bzw. zuletzt befand (wenn er fährt).

Das Attribut *bevorrechtigt* drückt aus, ob der Fahrstuhl sich selbst als bevorrechtigt ansieht, wobei *bevorrechtigt* die Werte *true*[42] und *false* annehmen kann.

Das Attribut *vorgemerkt* drückt aus, ob der Fahrstuhl sich vorgemerkt hat, dass der andere Fahrstuhl auf eine Erlaubnis von ihm wartet, wobei *vorgemerkt* die Werte *true* und *false* annehmen kann.

Das Attribut *wartend* drückt aus, ob der Fahrstuhl darauf wartet, dass der andere Fahrstuhl ihm eine Erlaubnis erteilt, wobei *wartend* die Werte *true* und *false* annehmen kann.

Das Attribut *bewZust* bestimmt den aktuellen Bewegungszustand und kann die Werte *fahrend* und *stehend* annehmen.

Das Attribut *etage* besagt, dass sich der Fahrstuhl in Etage *etage* (bei Bewegungszustand *stehend*) oder auf dem Weg von Etage *etage* weg (bei Bewegungszustand *fahrend*) befindet.

Die Attribute *etage* und *bewZust* entsprechen gemeinsam dem Prädikat Ort aus der Spezifikation mit Hilfe der Diskreten-Ereignis-Simulation.

Ein Fahrstuhl kann bis zu zwei Anforderungen erhalten haben, die jeweils als Informationen die Etage enthalten, zu der der Fahrstuhl angefordert wurde sowie das Attribut *außen*, das die Werte *true* und *false* annehmen kann. Dabei drückt *außen* aus, ob die Anforderung von außen gekommen ist. Mehrere Anforderungen eines Fahrstuhls zu einer Etage werden grundsätzlich zu einer zusammengefasst, wobei eine zusammengesetzte Anforderung genau dann als von außen erfolgt angesehen wird, wenn mindestens eine ihrer Teilanforderungen von außen erfolgte.

Abbildung 7 stellt den Zustand des Fahrstuhlsystems als UML-Klassendiagramm dar.

Abbildung 7: UML-Klassendiagramm für Fahrstuhl

Für den Startzustand ergibt sich

Fahrstuhl [1] . S_0 = { bevorrechtigt = true, vorgemerkt = false, wartend = false, etage = 1, bewZust = stehend, Anforderung = Ø }

Fahrstuhl [2] . S_0 = { bevorrechtigt = false, vorgemerkt = false, wartend = false, etage = 3, bewZust = stehend, Anforderung = Ø }

Am Anfang ist Fahrstuhl 1 bevorrechtigt und steht in Etage 1, während Fahrstuhl 2 in Etage 3 steht, Anforderungen gibt es noch keine.

Gestartet wird zum Zeitpunkt $c_0 = 0$. Es gibt vier Stränge von exogenen Ereignissen, die jeweiligen Anforderungen an die Fahrstühle. Alle Stränge gehören zum Typ *Wird_angefordert*.

exogenes Ereignis	`Wird_angefordert ( 1, 1 ) @ t`
Periodizität	`Expo_ZV(` $\mu_{1,1}$ `)`
Stopbedingung	-

[42] Hier werden *true* und *false* anstelle von *ja* und *nein* verwendet, um die Formulierung mit Hilfe von OCL zu erleichtern.

exogenes Ereignis	**Wird_angefordert (1, 2) @ t**
Periodizität	**Expo_ZV(** $\mu_{1,2}$ **)**
Stopbedingung	–

exogenes Ereignis	**Wird_angefordert (2, 2) @ t**
Periodizität	**Expo_ZV(** $\mu_{2,2}$ **)**
Stopbedingung	–

exogenes Ereignis	**Wird_angefordert (2, 3) @ t**
Periodizität	**Expo_ZV(** $\mu_{2,3}$ **)**
Stopbedingung	–

Ein explizites Simulationsende wird hier nicht erfasst, die Simulation läuft unendlich lange.

3.4.4.2 Ereignisausführung Kommt_an

Das Ereignis *Kommt_an* stellt die Ankunft eines Fahrstuhls in einer Etage dar. Parameter sind die Nummer des Fahrstuhls und die Nummer der Etage, in der der Fahrstuhl ankommt.

Als Zusicherung kann gegeben werden, dass der Fahrstuhl sich vorher auf dem Weg zu dieser Etage befunden haben muss.

Ereignis	**Kommt_an (i, j) @ t**
Zusicherung	**context Fahrstuhl [i]** **etage <> j and bewZust = fahrend**

Zusicherung Kommt_an

In allen Fällen gilt, dass, wenn der Fahrstuhl in einer Etage ankommt, alle Anforderungen an diese Etage erfüllt sind. Ferner muss in allen Fällen der Ort und der Bewegungszustand des Fahrstuhls aktualisiert werden.

Kommt der Fahrstuhl in einer Etage an, zu der er von außen angefordert wurde, so will der Fahrstuhl demnächst (in c_1 Zeiteinheiten) in die andere Etage fahren, um die eingestiegenen Personen dorthin zu befördern. Die Anforderung von innen zur anderen Etage wird ergänzt. Existiert schon eine Anforderung zur anderen Etage, so will der Fahrstuhl ebenfalls demnächst dorthin fahren. In beiden Fällen wird der Wert von *außen* beibehalten, um keine evtl. bereits existierende Anforderung zu löschen. Regel *Kommt_an.1a* deckt dabei den Fall ab, dass es bereits eine Anforderung von außen zur anderen Etage gab, während Regel *Kommt_an.1b* den Fall abdeckt, dass es noch keine Anforderung von außen zur anderen Etage gab. Gemeinsam sind sie äquivalent zu den Regeln *Kommt_an.1* (Regel 7) und *Kommt_an.2* (Regel 8) aus Abschnitt 2.3.4.

Trigger	**Kommt_an (i, j) @ t**
Triggerbedingung	–
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	**let k: Integer = self.etage** **Anforderung -> exists (außen and etage = k)**
Zustandseffekt	**etage = j and bewZust = stehend and** **not (Anforderung -> exists (etage = j))**
Folgeereignisse	**{ Will_fahren (i, j) @ (t +** c_1 **) }**

Regel 30: Kommt_an.1a

Trigger	**Kommt_an (i, j) @ t**
Triggerbedingung	–
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	**let k: Integer = self.etage** **(Anforderung -> exists ((außen and etage = j) or (etage = k)))**
Zustandseffekt	**let k: Integer = self.etage@pre** **etage = j and bewZust = stehend and** **not (Anforderung -> exists (etage = j)) and** **(Anforderung -> exists (not außen and etage = k))**
Folgeereignisse	**{ Will_fahren (i, j) @ (t +** c_1 **) }**

Regel 31: Kommt_an.1b

Kommt der Fahrstuhl in Etage 2 an und hat sich vorgemerkt, dass der andere Fahrstuhl auf eine Erlaubnis wartet, in Etage 2 fahren zu dürfen, so will der Fahrstuhl (spätestens) nach c_2 Zeiteinheiten anfahren, um Etage 2 für den anderen Fahrstuhl freizumachen. (Äquivalent: Regel 9)

Trigger	**Kommt_an (i, j) @ t**
Triggerbedingung	**j = 2**
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	**vorgemerkt**
Zustandseffekt	**etage = 2 and bewZust = stehend and** **not (Anforderung -> exists (etage = 2))**
Folgeereignisse	**{ Will_fahren (i, 2) @ (t +** c_2 **) }**

Regel 32: Kommt_an.3

Ansonsten werden nur die ggf. vorhandenen Anforderungen[43] als erfüllt markiert und der Ort und der Bewegungszustand werden aktualisiert. (Äquivalent: Regel 10)

Trigger	**Kommt_an (i, j) @ t**
Triggerbedingung	**-**
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	**-**
Zustandseffekt	**etage = j and bewZust = stehend and** **not (Anforderung -> exists (etage = j))**
Folgeereignisse	**Ø**

Regel 33: Kommt_an.4

3.4.4.3 Ereignisausführung Wird_angefordert

Das Ereignis *Wird_angefordert* stellt die Anforderung eines Fahrstuhls von außen dar. Parameter sind die Nummer des Fahrstuhls und die Nummer der Etage, von der aus der Fahrstuhl angefordert wird.

Als Zusicherung kann gegeben werden, dass, wenn der Fahrstuhl in einer Etage steht, dass dann zu dieser Etage keine Anforderungen vorhanden sein können.

Ereignis	**Wird_angefordert (i, j) @ t**
Zusicherung	**context Fahrstuhl [i]** **bewZust = stehend implies** **(not (Anforderung -> exists (etage = self.etage)))**

Zusicherung Wird_angefordert

Steht der Fahrstuhl gerade in der angeforderten Etage (dann kann gemäß Zusicherung vorher noch keine Anforderung zu dieser Etage vorhanden gewesen sein), so ist der erste Teil der Anforderung sofort erfüllt. Deshalb will der Fahrstuhl demnächst (in c_1 Zeiteinheiten) in die andere Etage fahren, um die eingestiegenen Personen dorthin zu befördern. Die Anforderung von innen zur anderen Etage wird ergänzt. Regel *Wird_angefordert.1a* deckt dabei den Fall ab, dass es bereits eine Anforderung von außen zur anderen Etage gab, während Regel *Wird_angefordert.1b* den Fall abdeckt, dass es noch keine Anforderung von außen zur anderen Etage gab. Gemeinsam sind sie äquivalent zur Regel *Wird_angefordert.1* (Regel 11) aus Abschnitt 2.3.5.

[43] Dass ein Fahrstuhl in einer Etage ankommt, ohne dass eine Anforderung in diese Etage vorlag, ist möglich, wenn ein Fahrstuhl nur zurück in seine Heimatetage gefahren ist, um Etage 2 für den anderen Fahrstuhl frei zu machen.

Trigger	**Wird_angefordert (i, j) @ t**
Triggerbedingung	–
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	**let k: Integer = andereEtage (i, j)** **etage = j and bewZust = stehend and** **Anforderung -> exists (außen and etage = k)**
Zustandseffekt	–
Folgeereignisse	**{ Will_fahren (i, j) @ (t +** c_1 **) }**

Regel 34: Wird_angefordert.1a

Trigger	**Wird_angefordert (i, j) @ t**
Triggerbedingung	–
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	**etage = j and bewZust = stehend**
Zustandseffekt	**let k: Integer = andereEtage (i, j)** **Anforderung -> exists (not außen and etage = k)**
Folgeereignisse	**{ Will_fahren (i, j) @ (t +** c_1 **) }**

Regel 35: Wird_angefordert.1b

Sollte vorher noch keine Anforderung nach Etage j vorhanden sein und steht der Fahrstuhl gerade in der anderen Etage, so wird die Anforderung in den Zustand aufgenommen und der Fahrstuhl will außerdem demnächst (in c_1 Zeiteinheiten) losfahren. (Äquivalent: Regel 12)

Trigger	**Wird_angefordert (i, j) @ t**
Triggerbedingung	–
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	**(Anforderung -> isEmpty()) and** **etage <> j and bewZust = stehend**
Zustandseffekt	**Anforderung -> exists (außen and etage = j)**
Folgeereignisse	**{ Will_fahren (i, andereEtage (i, j)) @ (t +** c_1 **) }**

Regel 36: Wird_angefordert.2

Ansonsten wird nur die Anforderung in den Zustand aufgenommen (sofern nicht bereits vorhanden). (Äquivalent: Regel 13)

Trigger	**Wird_angefordert (i, j) @ t**
Triggerbedingung	–
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	–
Zustandseffekt	**(Anforderung -> exists (außen and etage = j)**
Folgeereignisse	**Ø**

Regel 37: Wird_angefordert.3

3.4.4.4 Ereignisausführung Will_fahren

Das Ereignis *Will_fahren* stellt das Entstehen eines Anfahrwunsches eines Fahrstuhls dar. Parameter sind die Nummer des Fahrstuhls und die Nummer der Etage, von der aus der Fahrstuhl anfahren will.

Als Zusicherung kann gegeben werden, dass der Fahrstuhl zu diesem Zeitpunkt noch nicht auf eine Erlaubnis wartet.

Ereignis	**Will_fahren (i , j) @ t**
Zusicherung	**context Fahrstuhl [i]** **not wartend**

Zusicherung Will_fahren

Befindet sich der Fahrstuhl in Etage 2 und will von dort wegfahren, so kann er unmittelbar losfahren. (Äquivalent: Regel 14)

Trigger	**Will_fahren (i, j) @ t**
Triggerbedingung	**j = 2**
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	**etage = 2 and bewZust = stehend**
Zustandseffekt	–
Folgeereignisse	**{ Fährt_an (i, 2) @ t }**

Regel 38: Will_fahren.1

Will er dagegen in Etage 2 fahren, so muss er den anderen Fahrstuhl um Erlaubnis fragen. Somit erhält der andere Fahrstuhl demnächst (nach Δ Zeiteinheiten, der Übertragungsdauer einer Nachricht) eine Erlaubnisanfrage. Der Fahrstuhl wartet nun auf die Erlaubnis. (Äquivalent: Regel 15)

Trigger	**Will_fahren (i, j) @ t**
Triggerbedingung	**j <> 2**
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	**etage = j and bewZust = stehend**
Zustandseffekt	**wartend**
Folgeereignisse	**{ Erhält_Erlaubnisanfrage (andererFahrstuhl (i) @ (t + Δ)}**

Regel 39: Will_fahren.2

Befindet sich der Fahrstuhl nicht mehr in der Etage, von der aus er losfahren will, so ist dieses Ereignis irrelevant. Der Zustand ändert sich nicht, es wird kein Folgeereignis erzeugt. Dieser Fall kann eintreten, wenn sowohl gemäß einer der Regeln *Kommt_an.1a* (Regel 30) , *Kommt_an.1b* (Regel 31) , *Wird_angefordert.1a* (Regel 34) , *Wird_angefordert.1b* (Regel 35) oder *Wird_angefordert.2* (Regel 36) ein Folgeereignis *Will_fahren* erzeugt wurde, um Personen in die andere Etage zu befördern als auch gemäß einer der Regeln *Kommt_an.3* (Regel 32) oder *Erhält_Erlaubnisanfrage.2* (Regel 44) ein Folgeereignis *Will_fahren*, um Etage 2 für den anderen Fahrstuhl freizumachen. (Äquivalent: Regel 16)

Trigger	**Will_fahren (i, j) @ t)**
Triggerbedingung	–
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	–
Zustandseffekt	–
Folgeereignisse	Ø

Regel 40: Will_fahren.3

3.4.4.5 Ereignisausführung Fährt_an

Das Ereignis *Fährt_an* stellt das Anfahren eines Fahrstuhls dar. Parameter sind die Nummer des Fahrstuhls und die Nummer der Etage, von der aus der Fahrstuhl anfährt.

Als Zusicherung kann gegeben werden, dass der Fahrstuhl in der Etage steht, von der aus er anfährt.

Ereignis	**Fährt_an (i , j) @ t**
Zusicherung	**context Fahrstuhl [i]** **etage = j and bewZust = stehend**

Zusicherung Fährt_an

In allen Fällen wechselt der Fahrstuhl in den Bewegungszustand *fahrend* und es wird ein Folgeereignis erzeugt, die Ankunft des Fahrstuhls auf der anderen Etage.

Fährt der Fahrstuhl von Etage 2 weg und hat er sich vorgemerkt, dass der andere Fahrstuhl auf eine Erlaubnis für das Fahren in Etage 2 wartet, so erteilt er ihm diese in diesem Moment, so dass sie nach Δ Zeiteinheiten beim anderen Fahrstuhl ankommt. Er braucht sich nun nicht mehr vorzumerken, dass der andere Fahrstuhl auf eine Erlaubnis von ihm wartet. (Äquivalent: Regel 17)

Trigger	**Fährt_an (i, j) @ t**
Triggerbedingung	**j = 2**
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	**vorgemerkt**
Zustandseffekt	**not vorgemerkt and bewZust = fahrend**
Folgeereignisse	**{ Kommt_an (i, andereEtage (i, 2)) @** **(t + Gleich_ZV (a, b)),** **Erhält_Erlaubnis (andererFahrstuhl (i) @ (t + Δ)}**

Regel 41: Fährt_an.1

Ansonsten wird nur in den Bewegungszustand *fahrend* gewechselt und als Folgeereignis die Ankunft des Fahrstuhls erzeugt. (Äquivalent: Regel 18)

Trigger	**Fährt_an (i, j) @ t**
Triggerbedingung	–
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	–
Zustandseffekt	**bewZust = fahrend**
Folgeereignisse	**{ Kommt_an (i, andereEtage (i, j)) @ (t + Gleich_ZV (a, b)) }**

Regel 42: Fährt_an.2

3.4.4.6 Ereignisausführung Erhält_Erlaubnisanfrage

Das Ereignis *Erhält_Erlaubnisanfrage* stellt dar, dass ein Fahrstuhl vom anderen Fahrstuhl eine Erlaubnisanfrage für das Fahren in Etage 2 erhält. Parameter ist die Nummer des Fahrstuhls, der die Anfrage erhält.

Als Zusicherung kann gegeben werden, dass der Fahrstuhl sich vorher noch nicht vorgemerkt hat, dass der andere Fahrstuhl auf eine Erlaubnis von ihm wartet.

Ereignis	**Erhält_Erlaubnisanfrage (i) @ t**
Zusicherung	**context Fahrstuhl [i]** **not vorgemerkt**

Zusicherung Erhält_Erlaubnisanfrage

Befindet sich der Fahrstuhl, der die Anfrage erhält, auf dem Weg von Etage 2 weg, so kann er bedenkenlos eine Erlaubnis erteilen. (Äquivalent: Regel 19)

Trigger	**Erhält_Erlaubnisanfrage (i) @ t**
Triggerbedingung	–
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	**etage = 2 and bewZust = fahrend**
Zustandseffekt	–
Folgeereignisse	**{ Erhält_Erlaubnis (andererFahrstuhl (i) @ (t + Δ) }**

Regel 43: Erhält_Erlaubnisanfrage.1

Befindet sich der Fahrstuhl, der die Anfrage erhält, in Etage 2, so kann er dem anderen Fahrstuhl im Moment keine Erlaubnis erteilen, merkt sich aber vor, dass er ihm eine Erlaubnis erteilt, sobald er Etage 2 verlässt. Außerdem wird er (spätestens) in c_2 Zeiteinheiten anfahren, um dem anderen Fahrstuhl dann eine Fahrt in Etage 2 zu ermöglichen. (Äquivalent: Regel 20)

Trigger	**Erhält_Erlaubnisanfrage (i) @ t**
Triggerbedingung	–
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	**etage = 2**
Zustandseffekt	**vorgemerkt**
Folgeereignisse	**{ Will_fahren (i, 2) @ (t +** c_2 **)}**

Regel 44: Erhält_Erlaubnisanfrage.2

Befindet sich der Fahrstuhl, der die Anfrage erhält, gerade auf dem Weg zu Etage 2 hin, so kann er dem anderen Fahrstuhl im Moment keine Erlaubnis erteilen, merkt sich aber vor, dass er ihm eine Erlaubnis erteilt, sobald er Etage 2 wieder verlässt. (Äquivalent: Regel 21)

Trigger	**Erhält_Erlaubnisanfrage (i) @ t**
Triggerbedingung	–
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	**fahrend**
Zustandseffekt	**vorgemerkt**
Folgeereignisse	**Ø**

Regel 45: Erhält_Erlaubnisanfrage.3

Befindet er sich in seiner Heimatetage und hat selbst noch keine Erlaubnisanfrage gestellt (wartet also nicht seinerseits auf eine Erlaubnis des anderen Fahrstuhls), so erteilt er dem anderen die Erlaubnis. (Äquivalent: Regel 22)

Trigger	**Erhält_Erlaubnisanfrage (i) @ t**
Triggerbedingung	–
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	**not wartend**
Zustandseffekt	–
Folgeereignisse	**{ Erhält_Erlaubnis (andererFahrstuhl (i) @ (t +** Δ **) }**

Regel 46: Erhält_Erlaubnisanfrage.4

Befindet er sich in seiner Heimatetage und hat ebenfalls schon eine Erlaubnisanfrage gestellt, so haben sich also die Erlaubnisanfragen der beiden Fahrstühle gekreuzt. Demnach ist es entscheidend, welcher Fahrstuhl der Bevorrechtigte ist. Ist der Empfänger der Nachricht der Bevorrechtigte, so erteilt er dem anderen Fahrstuhl keine Erlaubnis, merkt sich aber vor, dass er die Erlaubnis erteilt, sobald er dann später irgendwann Etage 2 erreicht und auch wieder verlassen hat. Dadurch wird der andere Fahrstuhl der Bevorrechtigte. (Äquivalent: Regel 23)

Trigger	**Erhält_Erlaubnisanfrage (i) @ t**
Triggerbedingung	–
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	**bevorrechtigt**
Zustandseffekt	**not bevorrechtigt and vorgemerkt**
Folgeereignisse	**ø**

Regel 47: Erhält_Erlaubnisanfrage.5

Ist hingegen der Empfänger der Nachricht nicht der Bevorrechtigte, so erteilt er dem anderen Fahrstuhl die Erlaubnis. Dadurch wird er selbst der Bevorrechtigte. (Äquivalent: Regel 24)

Trigger	**Erhält_Erlaubnisanfrage (i) @ t**
Triggerbedingung	–
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	–
Zustandseffekt	**bevorrechtigt**
Folgeereignisse	**{ Erhält_Erlaubnis (andererFahrstuhl (i) @ (t + Δ) }**

Regel 48: Erhält_Erlaubnisanfrage.6

3.4.4.7 Ereignisausführung Erhält_Erlaubnis

Das Ereignis *Erhält_Erlaubnis* stellt dar, dass ein Fahrstuhl vom anderen Fahrstuhl eine Erlaubnis für das Fahren in Etage 2 erhält. Parameter ist die Nummer des Fahrstuhls, der die Erlaubnis erhält.

Als Zusicherung kann gegeben werden, dass der Fahrstuhl in seiner Heimatetage steht und auf eine Erlaubnis wartet.

Ereignis	**Erhält_Erlaubnis (i) @ t**
Zusicherung	**context Fahrstuhl [i]** **etage = andereEtage (i , 2) and bewZust = stehend and wartend**

Zusicherung Erhält_Erlaubnis

Der Empfänger der Nachricht ändert seinen Zustand dahingehend, dass er jetzt nicht mehr auf die Erlaubnis wartet. Er fährt sofort an. (Äquivalent: Regel 25)

Trigger	**Erhält_Erlaubnis (i) @ t**
Triggerbedingung	–
Reaktor	**Fahrstuhl [i]**
Zustandsbedingung	–
Zustandseffekt	**not wartend**
Folgeereignisse	**{ Fährt_an (i, andereEtage (i , 2)) @ (t) }**

Regel 49: Erhält_Erlaubnis.1