Simulation in Produktion und Logistik
Entscheidungsunterstützung von der Planung bis zur Steuerung
Wilhelm Dangelmaier, Christoph Laroque & Alexander Klaas (Hrsg.)
Paderborn, HNI-Verlagsschriftenreihe 2013

SpeedDeBlock – ein allgemeines generisches Blockade-Management System

SpeedDeBlock - an universal generic blockation management system

Karsten Wendt, Reik Zettl, Heike Wilson, Dualis GmbH IT Solution, Dresden (Germany), kwendt@dualis-it.de, rzettl@dualis.it.de, hwilson@dualis-it.de

Abstract: Event-based simulation tools and projects are often faced with the problem to handle upcoming blockations properly. This means, a certain number of elements reclaim under certain conditions the allocation of a subset of resources (other elements), often challenging among themselves. Typically a lot of programming capacities are bound to solve tasks like to check the conditions to decide when and which elements are allowed to allocate, notify them, managing queues and so on. Hereby we demonstrate a novel blockation handling system, which on one hand covers the most common blocking situations (from 1-1 blockations to n-m with FIFO queue blockations) and on the other hand provides interfaces to implement new types of blockation handling, sharing them among the entire simulation framework. To clarify the concept, illustrating examples are attached.

1 Einleitung

Zunächst werden in Abschnitt 1.1 und 1.2 die Motivation zur und die Abgrenzung der vorgestellten Arbeit dargelegt. Danach beschreibt Abschnitt 2 die Analyse der Problematik und zeigt den allgemeinen Lösungsansatz und dessen informelle Verarbeitung. Die Anwendung in einem komplexen Beispiel wird in Abschnitt 3 beschrieben. Abschließend erfolgt die Zusammenfassung und die Diskussion in 4.

1.1 Motivation

Ereignis-diskrete, objekt-orientierte Simulationen gewinnen heutzutage in immer mehr Bereichen an Bedeutung; mit großen und sehr großen Modellen werden komplexe Prozesse von der industriellen Fertigung bis zum Workflow großer Unternehmen abgebildet, validiert, überprüft und optimiert. Entscheidend für erfolgreiche Simulationsprojekte sind dabei weniger die bereits ausgereiften Simulationstechniken, sondern die Beherrschbarkeit, Validierung und Wiederverwendbarkeit der Modelle (Robinson 2005).

Bei der Umsetzung komplexer Prozessketten und –netzen sind Modellierer häufig mit Blockaden konfrontiert: Simulationsobjekte (SOs) allokieren unter bestimmten Bedingungen eine Teilmenge anderer SOs (Ressourcen). Schlägt die Allokation fehl, d.h. die Ressourcen sind bereits belegt, wechseln die betreffenden SOs in den Blockiert-Zustand und warten auf die erforderlichen Freigaben. Das Management solcher Situationen, d.h. Prüfen der Freigabe-Bedingungen, Verarbeiten konkurrierender und obsoleter Blockaden, Warteschlangen und das Benachrichtigen der blockierten SOs mit entsprechendem Kontext binden viel Kapazitäten, obwohl die Probleme oft ähnlich und verallgemeinerbar sind.

Mit *SpeedDeBlock* präsentieren wir ein allgemeines und erweiterbares Blockade-Management-System, welches die Blockade-Situationen abstrahiert, auf allgemeine Formen zurückführt, automatisch verarbeitet und somit die Validität, Wiederverwendbarkeit und Effizienz der Modell-Erstellung verbessert.

1.2 Abgrenzung

SpeedDeBlock ist ein Hilfsmittel, um wiederkehrende Modellierungssituationen in großen Modellen, wie exemplarisch aufgezeigt in Kleijnen (2005), effizient zu behandeln. Der allgemeine Ansatz der „Coupled Multicomponent Systems“ (Zeigler 2000) wird erweitert und gekapselt, sodass das Prüfen und Einplanen Blockade-relevanter Ereignisse entkoppelt vom eigentlichen Modell im Simulations-Framework eingebettet liegt. Es wird keine Optimierung hinsichtlich der Auflösung oder Vermeidung der Blockaden vorgenommen. Ohne Einschränkung des Modellierungsanspruchs wird dennoch eine verbesserte Analysierbarkeit der Abhängigkeiten im Modell erzielt (vgl. Venkatesh 2010).

2 Methoden und Verfahrensweisen

Im folgenden Abschnitt werden zunächst die Analyse und Dekomposition von allgemeinen Blockade-Situationen erläutert (2.1). Abschnitte 2.2, 2.3 und 2.4 definieren die daraus abgeleiteten Modellierungselemente. Anhand eines Beispiels zeigt 2.5 die Verwendung der Elemente zum Anlegen und Auflösen von Blockaden, 2.6 beschreibt mögliche Element-Erweiterungen.

2.1 Blockade-Analyse

Ziel ist eine allgemeine Beschreibung von Blockade-Situationen, dazu werden unter 2.1.1 zunächst allgemeine Szenarios definiert. Abschnitt 2.1.2 zerlegt die Beschreibungen in ihre informellen Bestandteile.

2.1.1 Analyse allgemeiner Blockade-Szenarios

Zunächst wurden, ausgehend von einem blockierten Referenz-SO SO_r, folgende erschöpfenden Blockade-Situationen erstellt:

- *1:1* SO_r ist ohne Konkurrenz von einem einzelnen SO_b blockiert
- *1:M* SO_r ist ohne Konkurrenz von *M* SO_b blockiert
- *N:1* SO_r ist von einem einzelnen SO_b blockiert und konkurriert mit *(N-1)* SO_c um diese Ressource
- *N:M* SO_r ist von *M* SO_b blockiert und konkurriert mit *(N-1)* SO_c um (eine Untermenge) diese(r) Ressourcen

Abbildung 1 zeigt schematisch diese vier Situationen, welche die Grundlage für die weitere Analyse darstellen. Demnach kann eine Blockade durch ihre Auflöse-Anforderungsbeschreibung, die Maßnahmen nach ihrer Auflösung und die Behandlung von Konkurrenz definiert werden.

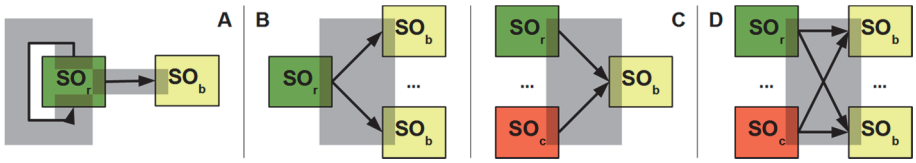

Abbildung 1: *Blockade-Szenarios; A: 1:1-Blockade (incl. Selbst-Blockade); B: 1:M-Blockade; C: N:1-Blockade; D: N:M-Blockade*

2.1.2 Blockade-Dekomposition

Basierend auf den Szenarios aus Abschnitt 2.1.1 wurde eine Blockade-Situation in ein *Requirement*, ein *Release-Handling* und die Zuordnung zu einer *Competition Group* zerlegt. Abbildung 2 zeigt die schematische Darstellung der Dekomposition. *Requirements* modellieren die Anforderungen zur Auslösung einer Blockade, *Release-Handlings* bilden den Vorgang beim Auslösen einer Blockade ab. Mehrere Blockaden integrierende *Competition Groups* lösen Konkurrenz unter wartenden SOs.

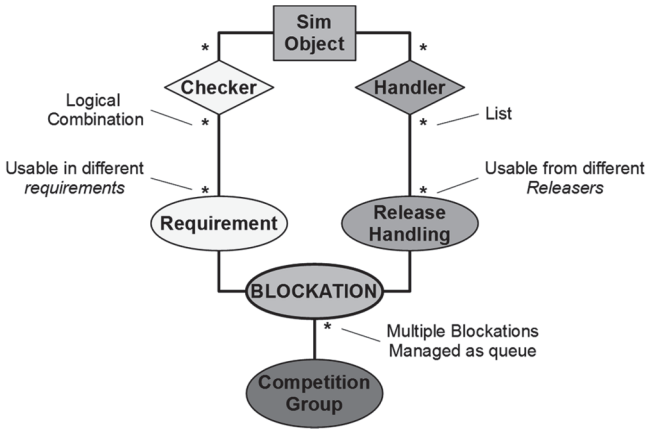

Abbildung 2: *Schematische Darstellung der Modellierung von Blockaden. Simulationsobjekte exportieren „Checker"- und „Handler"-Funktionen, die zum Überprüfen von Bedingungen zu Blockade-Auflösungen, bzw. zur Behandlungen von Blockade-Auflösungen genutzt werden. Die Checker werden in kombinatorischen Ausdrücken in Requirements gebündelt, die Handler als Liste in Release-Handlings. Requirement und Release-Handling beschreiben zusammen eine Blockade, die einer Competition Group zugewiesen wird. Eine Competition Group behandelt die Konkurrenz auslösbarer Blockaden.*

2.2 Requirements

Der folgende Abschnitt definiert des Modellierungselement *Requirement*, als Anforderungsbeschreibung zur Auflösung einer Blockade unter der Verwendung von *Checkern* (2.2.1 und 2.2.2). Abschnitt 2.2.3 beschreibt vordefinierte, häufig verwendete *Requirement*-Typen.

2.2.1 Checker-Delegaten

Die in Abschnitt 2.1.2 eingeführten *Checker*, oder *Checker*-Delegaten, sind Funktionen der Form

```
bool Check() (z.B. bool CanReceiveFlowObject())
```

und liefern eine Aussage darüber, ob eine singuläre Anforderung erfüllt ist.

2.2.2 Definition Requirement

Requirements beschreiben die Integration von mehreren *Checkern* C_i (siehe Abschnitt 2.2.1) als kombinatorischer Logik-Ausdruck der Form

$$R = A_0 \vee ... \vee A_n \;\; mit \;\; A = C_0 \wedge ... \wedge C_m \tag{1}$$

Ein *Requirement* gilt dann als erfüllt, wenn der logische Ausdruck wahr wird, z.B.:

$$R = Source.CanSend() \wedge Sink.Can\,\mathrm{Re}ceive() \tag{2}$$

2.2.3 Vordefinierte Requirement-Arten

SpeedDeBlock enthält unter Verwendung der Definition in Abschnitt 2.2.2 die in Abbildung 3 gezeigten vordefinierten *Requirement*-Typen zur Modellierung von Blockade-Anforderungen.

2.3 Release-Handling

Das Modellierungselement *Release-Handling* enthält die nötigen *Handler* (2.3.1) und Daten, welche im Falle der Blockade-Auflösung ausgelöst und verwendet werden (2.3.2).

2.3.1 Handler-Delegaten

Die in Abschnitt 2.1.2 eingeführten *Handler*, oder *Handler*-Delegaten, sind Funktionen der Form

```
void Handle(DataType ContextData) (z.B. void Send(Ptr Dest))
```

und bilden eine singuläre Aktion nach Auflösung einer Blockade ab.

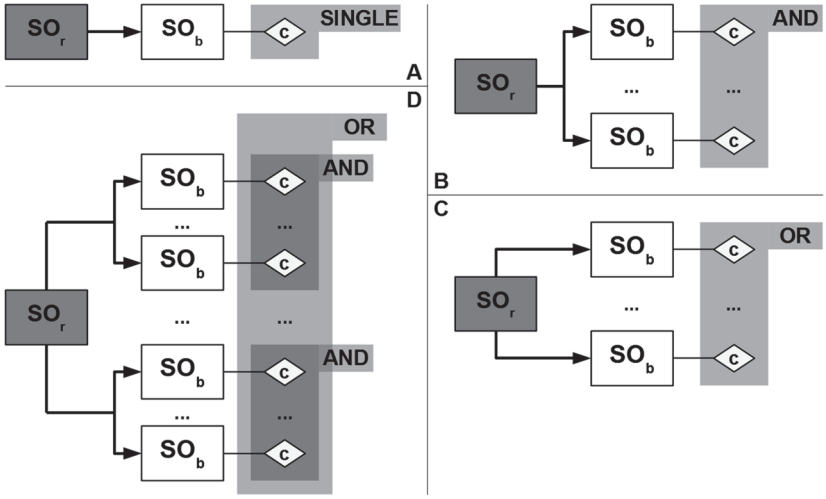

Abbildung 3: *Übersicht über vordefinierte Requirement Arten; A: Single-Req (d.h. Einzel-Checker muss wahr liefern); B: AND-Req. (d.h. alle Checker müssen wahr liefern); C: OR-Req. (d.h. mind. ein Checker muss wahr liefern); D: AND-OR-Req. (ODER-verknüpfte UND-Gruppen von Checkern)*

2.3.2 Definition Release-Handling

Das *Release-Handling* stellt eine Liste von *Handler*-Delegaten mit zugehörigen Kontext-Daten dar (s. Abs. 2.3.1), welche nach Blockade-Auflösung sequentiell aufgerufen werden.

2.4 Competition Groups

Zur Behandlung von Konkurrenz unter den wartenden SOs werden *Competition Groups* eingeführt (2.4.1) und häufig verwendete Typen erläutert (2.4.2).

2.4.1 Definition Competition Group

Während *Requirements* (s. 2.2) und *Release-Handlings* (siehe 2.3) nur im Kontext einer Blockade existieren, vereinen *Competition Groups* (siehe Einführung Abschnitt 2.1.2) mehrere konkurrierende Blockaden und lösen so das Reihenfolgeproblem bei mehrfach benötigten Ressourcen. *Competition Groups* bestehen aus einer Liste ihnen zugewiesener Blockaden (s. Abb. 2) und entscheiden gem. ihrer internen Logik, welche Blockade/n im Fall von mehreren auflösbaren Blockaden aufgelöst wird/werden.

2.4.2 Vordefinierte Competition Groups

SpeedDeBlock enthält unter Verwendung der Definition in Abschnitt 2.4.1 folgende vordefinierten *Competition Group*-Typen zur Modellierung von Blockade-Anforderungen:

- NONE: keine Konkurrenz-Behandlung; alle *Release-Handlings* werden ausgelöst
- FIFO: Warteschlange; das am längsten wartende *Release-Handling* wird ausgelöst

- LIFO: Stapel; das am kürzesten wartende *Release-Handling* wird ausgelöst
- PRIORITY: Warteschlange / Stapel mit Prioritäten; das am längsten / kürzesten wartende *Release-Handling* mit der höchsten Priorität wird ausgelöst

2.5 Modellierung

Im Folgenden werden das Anlegen, sowie das Auflösen einer Blockade in Konkurrenz mit Hilfe der definierten Elemente *Requirement*, *Release-Handling* und *Competition Group* (s. Abs. 2.2, 2.3 und 2.4) anhand eines vereinfachten Beispiels dargestellt (s. Abb. 4, schwarzer Rahmen). Ein Referenz-SO (SO_r) wartet auf benötigte Zustände zweier blockierender SOs (SO_b^0 und SO_b^1) und sich selbst. Zugleich konkurriert SO_r mit einem weiteres SO (SO_c) um eine der beiden Ressourcen SO_b^1. Abschnitt 2.5.1 beschreibt das Anlegen der Blockade, Abschnitt 0 das konkurrierende Freigeben.

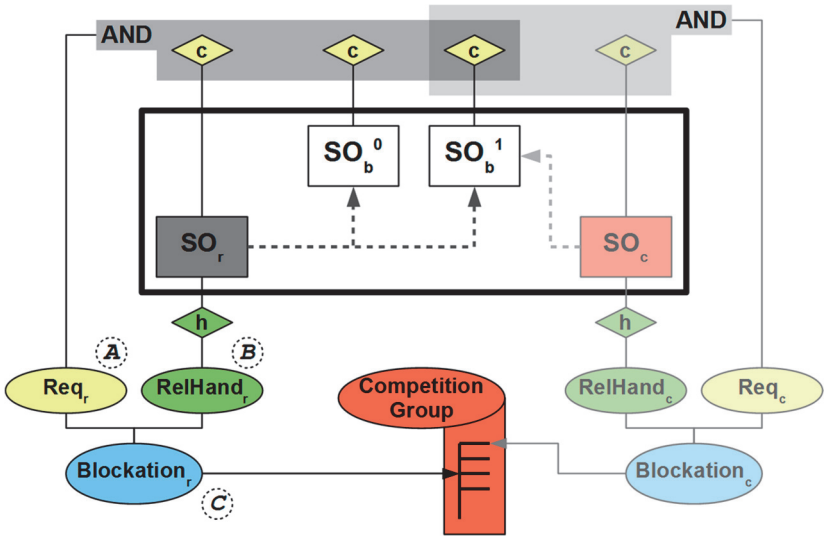

Abbildung 4: *Schematische Darstellung einer angelegten Blockade anhand eines vereinfachten Beispiels (blockiertes SO: SOr, blockierende SOs SOb0,1 und konkurrierendes SO SOc). A: Logische UND-Verknüpfung der Checker „c“. B: Allokation des Release-Handlers „h“.C: Anlegen der Blockade (blau) und Zuweisung zu einer Competition Group. Die Blockade des konkurrierenden Sim.-Obj. SOc ist bereits angelegt.*

2.5.1 Anlegen einer Blockade

Nach erfolgter Detektion einer Blockade werden die involvierten *Checker* „c“ allokiert und als kombinatorische AND-Funktion in einem vordefinierten *Requirement* (s. Abs. 2.2.3) zusammengefasst (Abb. 4, A). Der aufzurufende *Handler* „h“ von SO_r nach Auflösung der Blockade wird im *Release-Handling* ablegt (B). *Requirement* und *Release-Handling* bilden zusammen die Blockade,

welche abschließend in der *Competition Group* angemeldet wird (C). Algorithmus 1 zeigt eine mögliche Befehlsabfolge als Pseudo-Code.

Algorithmus 1:

```
Req_r         = new ANDReq(SO_r.CanSend, SO_b^0.CanReceive,
                  SO_b^1.CanReceive)
RelHand_r     = new RelDataCol(SO_r.Send)
CompGrp       = AllocateCC(„FIFOCC", Type=Fifo)
Blockation_r  = new Blockation(Req_r, RelHand_r, CompGrp)
```

2.5.2 Auflösen einer Blockade

Bei jeder Möglichkeit eine Blockade aufzulösen, rufen die SOs *Release()* auf. Das Framework übernimmt dann die Auswertung der betreffenden *Checker*, die Konkurrenzabwicklung in den *Competition Groups* und das Auslösen oder Einplanen der *Release-Handler*.

In Abbildung 5 löst SO_b^1 nach einer Zustandsänderung auf Verdacht *Release()* aus. *SpeedDeBlock* ermittelt alle involvierten Blockaden und prüft die *Checker* c der kombinatorischen *Requirements*. In diesem Beispiel liefern alle *Checker* c und somit alle AND-Gruppen wahr, sodass beide Blockaden $Blockation_r$ und $Blockation_c$ als auflösbar gelten. Die *Competition Group* (FIFO-Typ) verwertet diese Konkurrenzsituation und ermittelt $Blockation_c$ als ältere und somit aufzulösende Blockade. Mit dem Aufruf / Einplanen der *Release-Handler* h aus $RelHand_c$ und der Löschung der $Blockation_c$ ist der Auflösungsvorgang abgeschlossen.

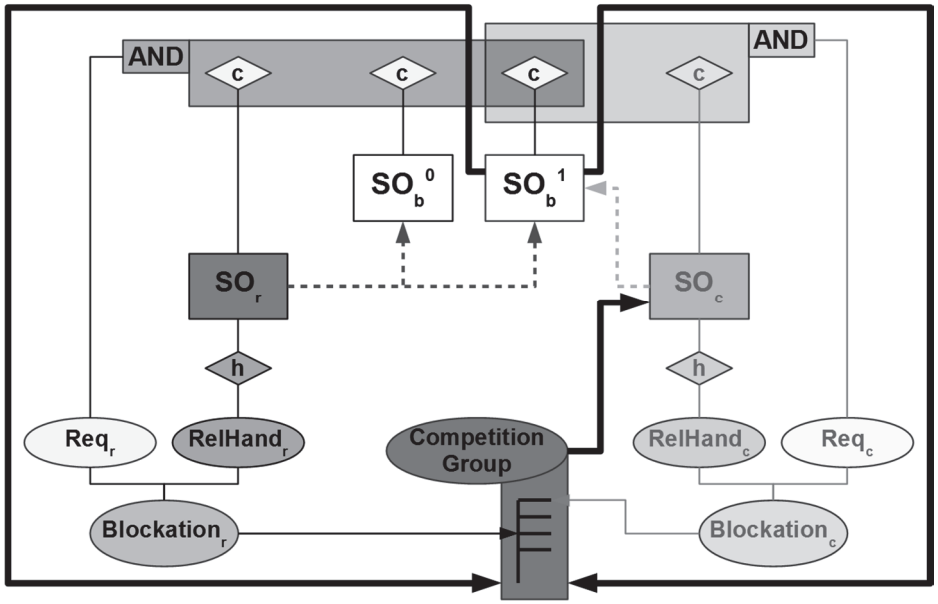

Abbildung 5: *Schematische Darstellung einer Blockade-Auflösung anhand eines vereinfachten Beispiels. Ausgehend von SOb1, verursacht Release() den Aufruf aller relvanten Checker c. In diesem Beispiel erweisen sich beide Blockaden als auflösbar. Die Competition Group der beiden auslösbaren Blockaden entscheidet über den Vorrang der beiden Blockaden. In diesem Fall ist es die ältere Blockationc, sodaß dessen Handler h aufgerufen wird.*

2.6 Erweiterbarkeit

Sowohl *Requirements* als auch *Competition Groups* sind als abstrakte Basis-Klassen definiert und lassen sich auf dieser Grundlage erweitern. Zu diesem Zweck muss jeweils die *Check()*-, bzw. die *PerformCompetition()*-Funktion überschrieben werden, um individuelle Verhaltensweisen anzulegen. So definierte *Requirement* und *Competition Groups* lassen sich wie aufgezeigt in der Modellierung verwenden (s. Abs. 2.5).

3 Ergebnisse

Dieser Abschnitt stellt zunächst in 3.1 die Anwendung von *SpeedDeBlock* in einem komplexen Beispiel dar. In 0 werden die Ergebnisse der vorgestellten Arbeit zusammengefasst.

3.1 Beispiel-Modellierung

In den folgenden beiden Abschnitten 3.1.1 und 3.1.2 werden die verwendeten SOs, sowie das genutzte Beispielmodell erläutert. 3.1.3 erläutert die Anwendung von *SpeedDeBlock* auf das Beispiel.

3.1.1 Simulationsobjekte

Die verwendeten SOs besitzen vier Zustände, welche sequentiell ineinander übergehen (s. Abb. 6). Im BUSY-Mode vergeht eine zufällig schwankende Zeitspanne bis zur Erstellung neuer Flussobjekte. Der READY-Mode beschreibt den Zustand der Sende-Bereitschaft der Flussobjekte. Nach dem Versenden wartet das SO im IDLE-Mode auf das Versenden eingehender Flussobjekte. Sind diese versandt, wird in den WAITING-Mode gewechselt. Weiterhin besitzen die SOs einen Input- und Output-Modus, welcher jeweils OR (von/zu irgendeinem Sender/Empfänger) oder AND (von/zu allen Sendern/Empfängern). Die SOs sind untereinander mit Förderstrecken verbunden, die Flussobjekte mit konstanter Geschwindigkeit transportieren.

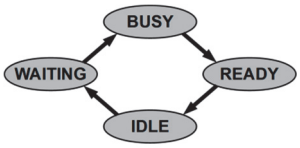

Abbildung 6: *Mögliche Zustände der SOs im Beispielmodell.*

3.1.2 Beispiel-Modell

Wie in Abbildung 7 gezeigt, besteht das Beispielmodell aus jeweils 3 Quellen, Prozessstationen und Senken, basierend auf den unter 3.1.1 beschriebenen SOs. Die mittlere Quelle arbeitet langsamer und beliefert alle Prozessstationen simultan (AND-Output-Modus) und muss daher priorisiert werden. Nach Verarbeitung in den Prozessstationen verschwinden die Flussobjekte in einer der drei Senken.

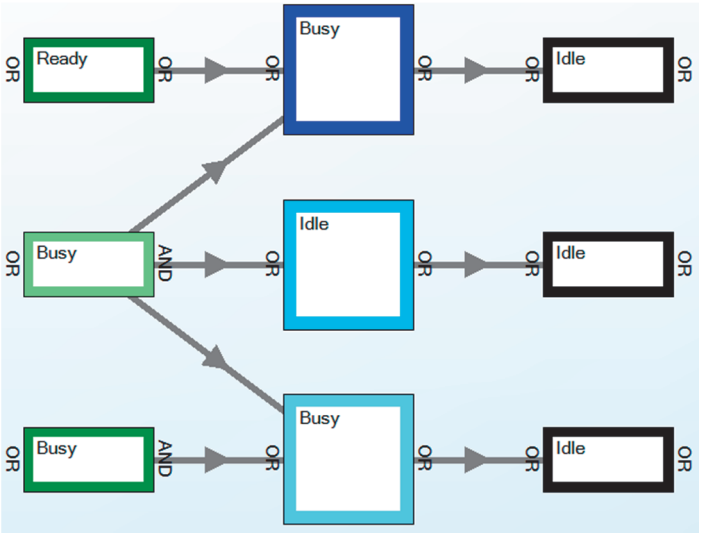

Abbildung 7: *Beispiel-Modell; Eine höher priorisierte Quelle (links, Mitte) prozessiert langsamer als die beiden anderen Quellen und muss an alle drei*

Prozessstationen (mittlere Spalte) gleichzeitig versenden. Zur Abbildung des Verhaltens werden Blockaden mit AND-Requirements (obere Quelle + obere Prozessstation, untere Quelle + untere Prozessstation, mittlere (priore) Quelle + alle Prozessstationen), siehe 2.2.3, verwendet und einer PRIORITY-Competition Group zugeordnet, siehe 2.4.2.

3.1.3 Verhaltensmodellierung mittels SpeedDeBlock

Algorithmus 2 (obere und untere Quelle) und 3 (mittlere Quelle) zeigen exemplarisch den Code-Aufwand zur Modellierung des unter 3.1.2 beschriebenen Verhaltens als Blockaden der Senken. Weiterhin wird bei jedem Zustandswechsels eines SOs in READY oder IDLE *Release()* von diesem aufgerufen, um verursachte Blockaden vom Framework auflösen zu lassen. Die Modellierung des restlichen (Blockade-unabhängigen) Verhaltens ist nicht aufgezeigt.

Algorithmus 2:

```
Req_ORSource =       new ANDReq(Source.CanSend, Target.CanReceive)
RelHand_ORSource =   new RelDataCol(Source.Send)
CompGrp =            Allocate(„PriorityGroup", Type=Prio)
Block_ORSource =     new Blockation(Req_ORSource, RelHand_ORSource,
                     CompGrp, Priority = 0)
```

Algorithmus 3:

```
Req_ANDSource =      new ANDReq(Source.CanSend,
                     Target_0.CanReceive .. Target_1.CanReceive)
RelHand_ANDSource =  new RelDataCol(Source.Send)
CompGrp =            Allocate(„PriorityGroup", Type=Prio)
Block_ANDSource =    new Blockation(Req_ANDSource, RelHand_ANDSource,
                     CompGrp, Priority = 1, PriorityCheck =
                     CanSend)
```

3.2 Zusammenfassung

Die Zerlegung in die allgemeine Modellierungselemente *Requirements*, *Release-Handlings* und *Competition Groups* (s. 2) eröffnet die Möglichkeit, Blockade-Situationen allgemein und kompakt zu modellieren. Die Auflösung geschieht automatisch und ohne weiteren Coding-Aufwand durch den Aufruf des Befehls *Release()* bei potentieller Auflösbarkeit.

In einem nicht-trivialen Beispiel (s. 3.1) konnte gezeigt werden, dass auch Blockierungen innerhalb komplexer Netze und Verhaltensweisen mittels *SpeedDeBlock* beherrschbar sind.

4 Diskussion

Im folgenden werden aufgetretene Probleme bei und mit der Arbeit des Blockade-Management-Systems (4.1), sowie Erweiterungsmöglichkeiten erläutert (4.2).

4.1 Probleme

Operative Schwierigkeiten ergeben sich bei der Umsetzung aufwendiger und großer Modelle aufgrund des hohen Abstraktionsgrades der Blockade-Modellierung. An dieser Stelle ist entweder längere Einarbeitung oder eine bessere Schnittstelle (API) notwendig. Weiterhin fehlt bisher der Nachweis der Allgemeingültigkeit des Ansatzes. Die Frage, ob sich jedes denkbare Szenario auf diese Weise abbilden lässt, ist aufgrund der hohen Komplexität schwierig.

4.2 Ausblick

Ausblickend kann zusammengefasst werden, dass das Framework zum einen zur statistischen Auswertung von Blockaden (Was von wem wie lange blockiert wurde) erweitert oder zum anderen zur Erstellung von generischem Verhalten (automatisiert erstellte Verhaltenslogik) oder zur vorrausschauenden Vermeidung von Deadlocks verwendet werden könnte.

Literatur

Kleijnen, J. P. C. et al.: Finding the Important Factors in Large-Scale Discrete-Event Simulations: Sequential Bifurcation and its Applications, CentER Discussion Paper (2003)

Robinson: Discrete-event simulation: from the pioneers to the present, what next. Journal of the Operational Research Society (2005)

Venkatesh, S. et al.: Deadlock Detection and Resolution for Discrete Event Simulation: Multiple-Unit Seizes, College Station (2010)

Zeigler, B. P. et al.: Theory of Modeling and Simulation, Second Edition, Elsevier Service Academic Press, 2000