Simulationstechniken

Edwin Jean Baptiste

17. November 1994

Seminar
der
Projektgruppe Systementwurf
WS94/95 und SS 95

Inhaltsverzeichnis

1 Simulation

Der Begriff Simulation stammt vom lateinischen simulare und meint somit ähnlich-machen. Es soll etwas geschehen, ablaufen, durchgeführt werden, das einem Urbild ähnlich ist.
Simulieren setzt meistens ein zu auszuführendes Modell des betrachteten Objektes voraus. Durch Simulation sorgt man dann dafür, daß sich ein Rechner wie das Objekt verhält. Die Simulation soll aber nur eine Schritt im Prozeß der Verifikation sein. Sie produziert nämlich Daten, die danach für die eigentliche Verifikation untersucht werden müssen.
Somit bedeutet Verifikation auf der Basis von Simulation letztendlich, zu analysieren, ob das Modell des Enwurfsobjektes und die Daten von der Umgebung die korrekten Ergebnisdaten produziert. Demnach besteht ein komplettes Verifikationssystem das auf Simulation basiert aus vier Hauptkomponenten:

- Ein Generator für ausfürbare Modelle von Objektumgebungen,
- ein Generator für ausfürbare Modelle der zu simulierenden Objekte,
- ein Laufzeitsystem für das Modell zusammen mit seiner Umgebung und
- ein Ergebnisanalysator

2 Simulationstechniken

Die Aufgabe eines Simulationsalgorithmus ist es, ein internes Modellierungskonzept des Modellierungssystems auf die Architektur des Gastrechners abzubilden. Falls das Original selber eine zeitliche Komponente hat, ist der Zeitbegriff von Anfang an vorhanden. Allerdings ist das zeitliche Element nicht in allen Systemen dominant. Das Hauptproblem der Abbildung ist der hohe Grad an parallelismus der üblicherweise in den Modellierungskonzepten zu finden ist, da in den meisten Fällen ein strikt sequentieller Rechner benutzt wird.

Es gibt drei Haupttechniken, um dieses Problem zu lösen:

- Streamline Code Simulation (SCS)
- Equitemporal Iteration (EI)
- Critical Event Scheduling (CES)

2.1 Streamline Code Simulation

Die Idee bei dieser Technik ist, unmittelbar ausfühbaren Code des Gastrechners aus der Schaltungsbeschreigung zu generieren. Die Beschränkungen dazu sind:

- Das Modellierungskonzept ist kontinuierliche Auswertung,
- das zu simulierende Objekt ist entweder kombinatorisch oder strikt synchron und
- es besteht kein Bedarf an Zeitinformation.

Die klassische Anwendung ist Simulation auf de Gatterebene. Das kombinatorische Schaltnetz kann als Directed Acyclic Graph (dag) dargestellt werden. Dabei werden die Gatter als Knoten dargestellt, und die Verbindungen als Kanten.
Beispiel Abb. 1 zeigt ein Halbaddierer und Abb. 2 seine Darstellung als dag.

Die Knoten des dag können bezüglich des längsten Pfades von Knoten zu Primär Eingängen halbgeordnet werden. Diese Technik wird Levelizing genannt. Dabei erhält ein Primäreingang in_i level(in_i) = 0 und jeder andere Knoten n_j erhält level(n_j) = (1+maxlevel(n_k) | Es gibt eine Kante von n_k zu n_j).
Abb.3 zeigt das Beispiel nach dem levelizing.

Die Ebenen können wie folgt interpretiert werden:

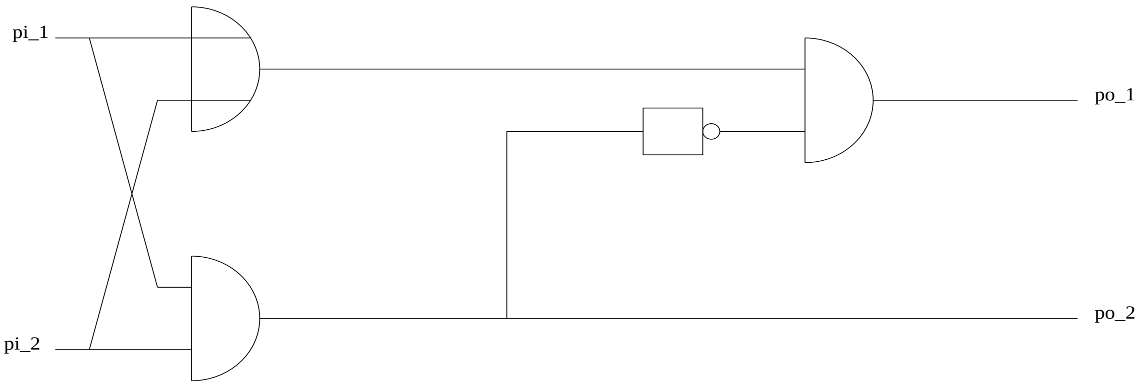

Abbildung 1: Kombinatorisches Schaltnetz

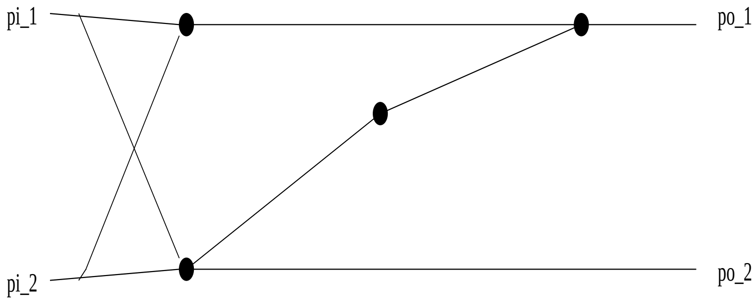

Abbildung 2: Darstellung als dag

- Kein Knoten auf einer höheren Ebene kann einen solchen auf einer niedrigeren beeinflussen,
- Knoten auf derselben Ebene beeinflussen sich gegenseitig nicht,
- Ein Knoten auf einer niedrigeren Ebene kann jeden Knoten auf einer höheren beeinflussen.

Die Abhängigkeitsrelation die durch das Levelizing auf der Schaltung eingeführt wird ist präzise genug, um eine Sequenz von Berechnungen der einzelnen Knoten aufzustellen. Im Fall der Simulation auf Gatterebene werden die Netze der Schaltung als Variable dargestellt. Der Code der für ein Gatter benötigt wird besteht aus sehr wenigen Instruktionen der Zielmaschine. Das obige Beispiel kann zu folgendem Code führen:

```
var    pi_1, pi_2 : word;
       in_1, in_2, in_3 : word;
       po_1, po_2 : word;
begin
       in_1 := pi_1 or pi_2;          {level 1 computation}
       in_2 := pi_1 and pi_2;         {level 1 computation}
       in_3 := not(in_2);             {level 2 computation}
       po_1 := in_1 and in_3;         {level 3 computation}
       po_2 := in_2;                  {level 3 computation}
end;
```

Der Code der hier benutzt wurde kann einfach durch den Maschinencode eines Rechners ersetzt werden. Das Programm beschreibt die Berechnung auf der Basis eines einzelnen Eingabemusters. Es kann jedoch so erweitert werden (z.B. durch das einfügen einer Schleife), daß es eine beliebige Sequenz von Mustern verarbeitert.

Das Zeitverhalten des simulierten Systems wird durch SCS nur sehr grob dargestellt. Es können zwar extrem schnelle Simulationszeiten erreicht werden, doch bleibt zu fragen, für welche Anwendungen eine derartige Simulation benötigt wird. Es sei denn zum Beispiel nicht bekannt welch eine Funktion zu

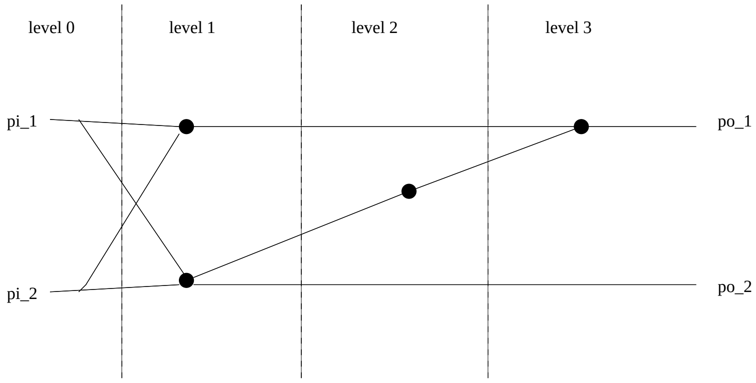

Abbildung 3: Levelizing angewandt auf ein dag

implementieren ist. Denn falls die Funktion bekannt ist, kann durch einfache algebraische Methoden überprüft werden, ob eine Implementierung korrekt ist.

2.2 Equitemporal Iteration (EI)

Wie bei SCS findet hier iterativ ein überstreichen des zu simulierenden Systems statt. Nach jedem Überstreichen wird die globale Zeit um eine Schrittweite erhöht. Diese Schrittweite kann von Iteration zu Iteration variieren kann, ist aber für alle Komponenten gleich (daher der Begriff Equitemporal). Für die Simulation wird das System durch Komponenten beschrieben. Jede Komponente i wird durch ein Tripel (c_i, a_i, d_i) modelliert, mit c_i als Ausführbarkeitsbedingung der Komponente, a_i als ihre Aktion und d_i als Zieldatenobjekt.
Es sei nun angenommen, daß die Bedingung c_i deiner bestimmten Komponente i während einer Iteration wahr ist. Als Konsequenz wird die Aktion a_i ausgeführt. Als Auswirkung dieser Ausführung erhält die Zielvariable d_i ein neuer Wert. Diese Wertzuweisung wird aber nicht direkt auf die Zielvariable ausgeführt, sondern auf dedizierte Puffer. Erst nach dem vollständigem Durchgang werden alle Puffer in die Zielvariablen kopiert. Dadurch hat die Sequenz in der Komponenten besucht werden keinen Einfluß auf das Verhalten des Simulators. Ein derartiger Algorithmus könnte wie folgt aussehen:

```
begin
      time := 0;
      final_time := stop_time;
      while time ¡= final_time do
           begin
             for i := 1 to component_number do
                    if c(i) then d_buffer[i] := a(i);
             time := time + increment;
             for i:= 1 to component_number do
                    d[i] := d_buffer[i]
           end;
end.
```

Beispiel:
Angenommen, der Halbaddierer (wie bei SCS) soll simuliert werden, aber mit einer individuellen Verzögerung von 3 Zeiteinheiten bei jedem AND-Gatter, von 4 Zeiteinheiten bei jedem OR-Gatter, von 2 Zeiteinheiten beim NOR-Gatter und 7 Zeiteinheiten zum Speichern des neuen Zustands (Abb. 4).
Die Datenstruktur für die zu simulierende Schaltung kann wie folgt dargestellt werden:

```
definition module circuit;
      const component_number = 8;
      type word = bit(32);
```

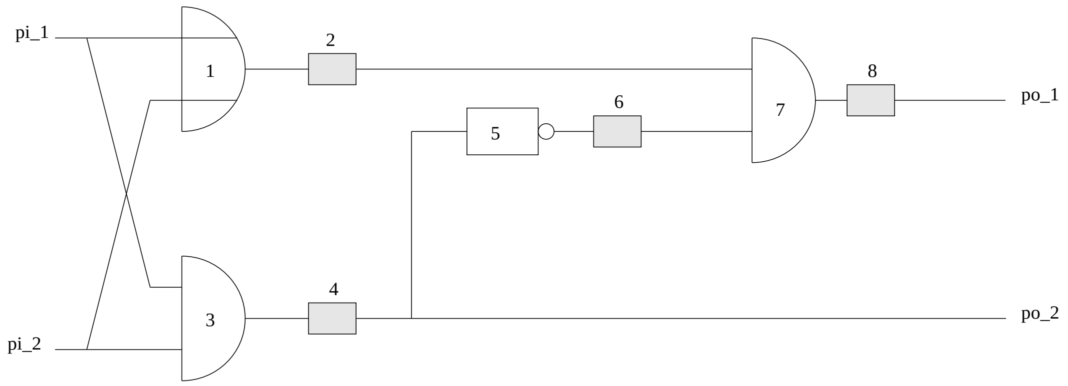

Abbildung 4: Schaltwerk mit indiduellen Zeitverzögerungswerten

```
            var d : array[1:8] of word;
                increment, stop_time    : integer;
                function c(in_i : integer) : bit;
                function a(in_i : integer) : word;
end circuit;

implementation module circuit;
        const component_number = 12;
        type word = bit(32);
        var d : array[1:8] of word;
                gate_time  : array[1:8] of integer;
                increment, stop_time : integer;
                clock : bit;                        {soll extern gesetzt werden}

function delayer (in index : integer; in value : word) : word;
var shifter : array[1:4] of array [0:4] of word;
begin       case index of        {Delay durch Pipeline: je länger die Röhre, desto länger die Verzögerung}
            1 : begin
                    delayer := shifter [1,1];
                    shifter [1,1] := shifter [1,0];
                    shifter [1,0] := value;
            end;
            3 : begin
                    delayer := shifter [2,0];
                    shifter [2,0] := value;
            end;
            5: begin
                    delayer := value;
            end;
            7 : begin
                    delayer := shifter [4,0];
                    shifter [4,0] := value;
            end;
end;

function c(in i : integer) : bit;          {Ausfürbarkeitsbedingungen festlegen, wegen
                                            kontinuierlicher Auswertung immer auf 1 gesetzt}
begin
        c := 1;
end;
```

```
function a(in i : integer) : word;                {Aktionen, logische Verknüpfungen oder Delays}
begin
        case i of
                1 : begin
                        a := pi_1 | pi_2;
                end;
                2 : begin
                        a := delayer(1,d[1])
                end;
                3 : begin
                        a := pi_1 & pi_2                    end;
                4 : begin
                        a := delayer(3,d[3])
                end;
                5 : begin
                        a := not(d[4]);
                end;
                6 : begin
                        a := delayer(5,d[5])
                end;
                7 : begin
                        a := d[2] & d[6]
                end;
                8 : begin
                        a := delayer(7,d[7])
                end;
        end;
end circuit;
```

Das Gerüst für den Simulationsalgorithmus sieht dann wie folgt aus:

```
module main;
from circuit import component_number, word, increment, stop_time, d, c, a;
var     time, final_time : integer;
        d_buffer : array[1:component_number] of word;
    begin
        time := 0;
        final_time := stop_time;              {stop_time soll von außen gesetzt werden}
        while time ¡= final_time do
            begin
                for i := 1 to component_number do
                    if c(i) then d_buffer[i] := a(i) else;
                time := time + increment;
                for i := 1 to component_number do
                    d[i] := d_buffer[i]
            end;
    end;
end main.
```

In diesem Beispiel ist ein Modul für jedes Gatter in der Datenstruktur vorgesehen. Zusätzlich wurde für jedes Gatter ein weiteres Modul eigeführt, um die dem Gatter zugewiesene Verzögerung zu beschreiben. Die Aktionen sind eifach die Gatterfunktionen. Für die Verzögerungen wird ein Shifter-Modell benutzt. Neue Werte müssen durch eine Röhre fließen, wobei die Länge dieser Röhre die Verzögerung beschreibt. Abbildung 5 skizziert diesen Algorithmus.

Der EI-Algorithmus ist sehr einfach und leicht zu implementieren. Leider ist er meistens sehr ineffizient, weil typischerweise zu einem bestimmten Zeitpunkt, mehr als 95 % der Komponenten stabil sind.

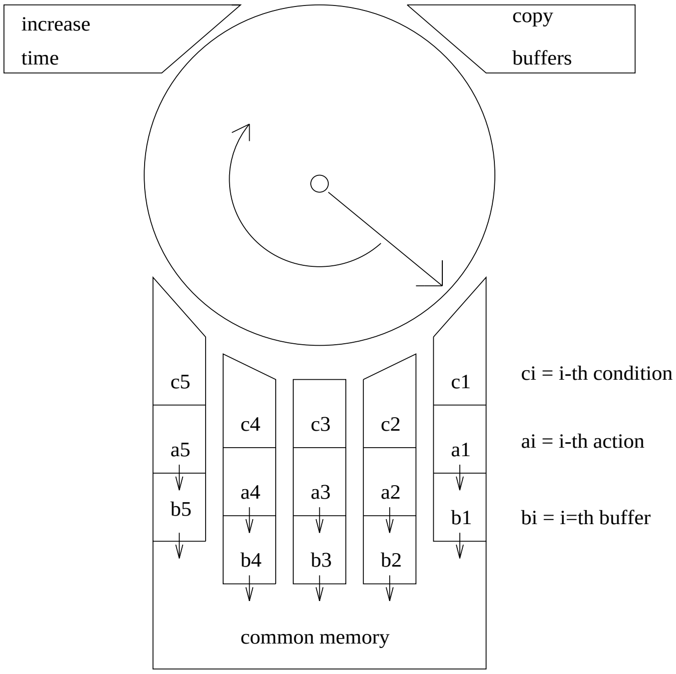

Abbildung 5: EI-Algorithmus

2.3 Critical Event Scheduling

Diese Simulationstechnik ist ein Versuch, die Effizienzprobleme von EI zu lösen. Man beschränkt sich hierbei auf nichttriviale Berechnungen. Critical Event Schedulibg (CES) ist auf Modellierungskonzepte anwendbar, die folgende einshcrängungen genügen:

- Der Zeitpunkt des nächsten Auftretens eines Ereignisses ist vorhersagbar.
- Falls der Zeitpunkt des nächsten Auftretens eines bestimmten Ereignis nicht vorhersagbar ist, findet dieses Ereignis nicht statt, bevor es nicht durch das Stattfinden anderer Ereignisse vorhersagbar geworden ist.

CES nimmt an, daß das System zur Compile-Zeit in Komponenten zerteilt wird und ein Abhängigkeitsbeziehung auf diesen bestimmt wird. Ist Komponente B direkt abhängig von Komponente A, so ist wird A Beeinflußer von B und B Beeinflußter von A genannt. Durch diese Beziehung kann CES bestimmen, welche Komponenten durch die Zuweisung eines neuen Wertes an ein Datenobjekt beeinflußt werden. Nur diese Teile der Schaltung werden weiter verfolgt. Somit ist CES ein lokaler Ansatz (im Gegensatz zu EI und SCS). Das folgende Gerüst zeigt den Algorithmus:

```
module main;
  from circuit import circuit_size, word. data, influencee_nr, influencee, executable, action, elapses;
    const empty = 0;
    type event = record
            begin
                component_id : integer;      {Ereignisnummer}
                event_time : integer;        {Ereigniszeit = Simulationszeit}
                new_value : word;             {Wert}
            end;
    var             cmcurrent_event, new_event : event;
        current_time, queue_fill : integer;
        changed             bit;
    export (insert, remove, test) procedure event_queue;
```

```
        const queue_length = ...;
        var queue : array[0 : queue_length] of event;
          top_of_queue : integer := 0:
        procedure insert (in item : event ):
          begin
            sort_in (item);
            {assume that sort_in (item) inserts item properly keeping queue sorted
             in ascending order with respect to event_time}
            top_of_queue = top_of_queu + 1;
          end;
        function remove : event;
          begin;
            remove := queue[top_of_queue];
            top_of_queue :=top_of_queue - 1;
          end;
        function test : integer;
          begin
            test := top_of_queue;
          end;
      end;        {event queue}
begin
      time := stop_time ; {stop_time to be supplied externally }
      while time ¡= final_time & queue_fill ¡¿ empty do
        begin
          current_event := event_queue.remove
          current_time := current_event.event_time;
          changed := data[current_event.component_id] ¡¿
                              [current_event.new_value;
          if changed & influencee_nr [current_event.component_id] ¿ 0 then
          begin
            for i := 1 to influencee_nr[current_event.component_id]
              do
                begin
                  component := influencee[current_event.component_id,i];
                  if executable(component) then
                  begin
                    new_event.component_id := component;
                    new_event.new_value := action(component);
                    new_event.event_time := current_time + elapses(component);
                    event_queue.insert(new_event);
                  end;
                end;

          else;
            queue_fill := event_queue.test
          end;
      end;
end main.
```

Dieser Algorithnus beinhaltet die Verwaltung einer Ereignisschlange (event_queue), die in aufsteigender Reihenfolge bezüglich der Ereigniszeitpunkte (event_time) sortiert ist. Diese Verwaltung wird durch die Exportprozedur event_queue durchgeführt. Man beachte, daß es kein festes Zeitinkrement gibt, sondern daß die Zeit zu dem Zeitpunkt, an dem das nächste Ereignis stattfindet, fortgeschaltet wird. Es wird nun übeprüft, ob es eine Wertänderung als Effekt des Ereignisses gibt.

3 Time Warp Mechanism

Die Simulation wird in interaktive Prozesse *Objekte* aufgeteilt. Der Austausch von Informationen geschieht durch Versenden von Ereignis-Nachrichten *eventmessages* untereinander. Jedes Objekt be-

sitzt eine Input-Queue für die Nachrichten.
Im folgenden wird der Begriff *virtuelle Zeit* gleichgestellt mit *Simulationszeit*. Jedes Objekt hat seine eigene LVT (Local Virtual Time; lokale virtuelle Zeit), welche als Simulationsuhr fungiert. Wenn LVT die Größe t hat, dann arbeitet das Objekt zur Simulationszeit t. Nachrichten mit kleinerem t, die and diesem Objekt ankommen, wurden schon bearbeitet. Nachrichten mit Zeitstempeln größer als t sind es aber noch nicht.
Ein Objekt arbeitet immer, solange seine Input-Queue noch nicht erschöpft ist. Die Bearbeitungsstrategie ist *optimistisch*, d.h. man geht das Risiko ein, daß ein Nachzügler *straggler* z.B. eine Nachricht mit einem Zeitstempel kleiner oder gleich dem LVT des Empfängers eintrifft. Falls ein Nachzügler erscheint, wird das Objekt soweit zurückgesetzt, bis dessen virtuelle Zeit kleiner als der Zeitstempel der verspäteten Nachricht ist *rollback*. Dann wird die Nachricht in die Queue einsortiert und das Objekt kann seine Arbeit fortführen.

Nachrichten, Anti-Nachrichten Queues und Objekte

Eine Event Message unter Time Warp besitzt 6 Komponenten (Abb. 6).

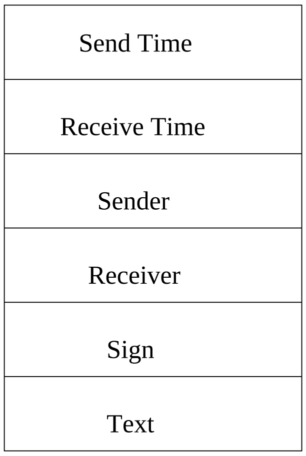

Abbildung 6: Struktur einer Nachricht

Sie enthält folgende Informationen:

1. **Send time:** die LVT (Local Virtual Time) des Senders im Moment des Sendens. Sie bestimmt die Position der Nachricht in der Out-Queue des Senders.
2. **Receive time,** auch *Zeitstempel* genannt: Sie gibt die Simulationszeit des Ereignisses an. Dieses Feld bestimmt die Position der Nachricht in der Input-Queue des Empfängers, wenn die Nachricht bearbeitet wird.
3. **Sender:** der Name des Objekts, welches die Nachricht geschickt hat.
4. **Receiver:** der Name des Objektes, das die Nachricht empfängt.
5. **Sign:** ein Bit, das bestimmt, ob die Nachricht positiv (+) oder negativ (-) ist. Dies wird für den rollback-Mechanismus benötigt.
6. **Text:** Zusäzliche Informationen, die für die Simulation des Ereignisses notwendig sind.

Nachrichten, die sich nur durch ihre Signs unterscheiden und sonst völlig identisch sind, werden als Anti-Nachrichten zueinander bezeichnet. Nachrichten, die durch Benutzerprogramme entstanden sind,

sind imme positiv; negative Nachrichten können nur durch ein Zurückrollen (rollback) erzeugt werden und sind für den Simulationsprogrammierer nicht sichtbar.
Eine Nachricht wird durch ihre virtuelle Sende-Zeit in die Output-Queue oder durch ihre virtuelle Empfangs-Zeit in die Input-Queue einsortiert. Falls diese Queue eine Anti-Nachricht enthält, so werden beide Nachrichten sofort vernichtet.
Ein Objekt besitzt 5 Komponenten, wie in Abb. 7 gezeigt ist:

Current State

4	12

LVT

162

Input Message Queue

110	105	120	125	130	146	175	176	Sending Tme
112	119	121	141	156	162	181	182	Receiving Time
E	C	B	E	B	D	B	B	Sender
A	A	A	A	A	A	A	A	Receiver
+	+	+	+	+	+	+	+	Sign
								Text

Output Message Queue

119	121	141	141	156	156	162	Sending Tme
141	122	142	196	180	157	163	Receiving Time
A	A	A	A	A	A	A	Sender
A	B	B	C	C	D	C	Receiver
+	+	+	+	+	+	+	Sign
							Text

State Queue

82		90		106		119		141		LVT of State
3	12	7	12	-2	12	8	12	0	12	Saved State

Abbildung 7: Struktur eines Objektes

1. Das LVT-Register zeigt die LVT für dieses Objekt also die virtuelle Zeit für das aktuelle Ereignis. Es beschreibt also die lokale Simulationszeit.
2. Current State enthält den aktuellen Status des Objekts zur virtuellen Zeit LVT.
3. in der Input Message Queue stehen alle eingegangen Nachrichten nach ihrer virtuellen Empfangszeit sortiert.
4. Die Output Message Queue enthält Kopien der verschikten Nachrichten, geordnet nach ihrer virtuellen Sendezeit.
5. In der State Queue stehen einige Momentaufnahmen von Zuständen der Vergangenheit des Objektes, geordnet durch di LVT. Meistens werden nicht alle Zustände gespeichert.

Der Rollback Mechanisus:
Das Objekt in Abb. 7 hat eine LVT von 162 und bearbeitet die Nachricht mit dem Zeitstempel

162. Falls eine Nachricht mit Zeitstempel größer als 162 eintrifft, si wird sie ohne Probleme in die Input-Queue eingefügt. Falls ein Nachzügler (Zeitstempel kleiner als 162) ankommt, muß ein rollback durchgefüht werden. Dazu müßen alle Nebeneffekte behoben werden die durch Nachirichten mit Zeitstempeln grösser als t hervorgerufen worden sind. DAs Objekt wird zu einem Zeitpunkt zurückgesetzt, welcher noch vor t liegt, und die Simulation ab diesen Zeitpunkt weitergefüht.
Z. B. angenommen Objekt E sendet eine Nachricht Mit Zeitstempel 135 an Objekt A. Da die LVT von a 162 ist, sollte A diese Nachricht schon bearbeitet haben. A muß dann zur Zeit 135 zurückrollen. Der rollback Mechanismus ist in 3 Phasen unterteilt: Rückroll-Phase, Rücknahme-Phase und Vorrück-Phase.

Die Rückroll-Phase

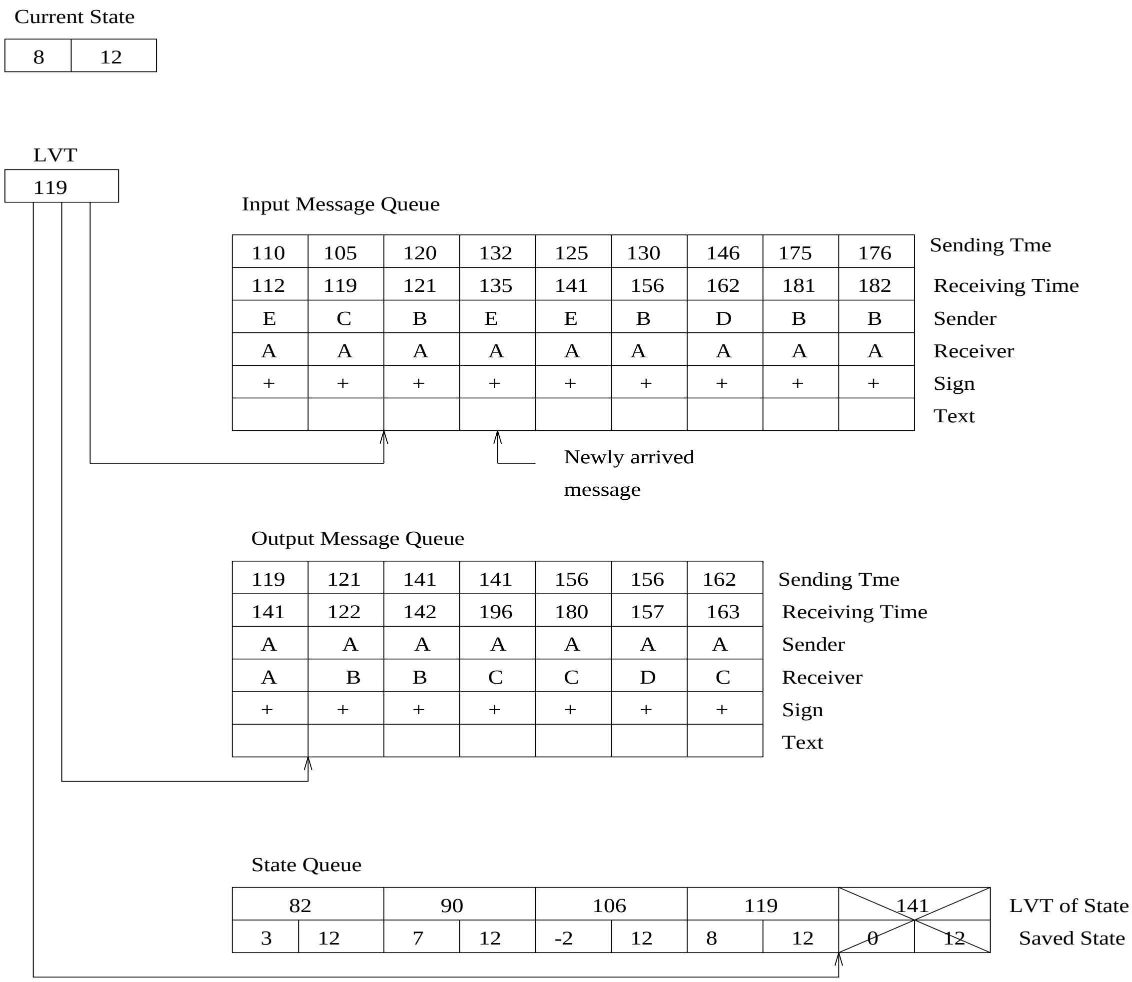

Abbildung 8: Rückroll-Phase

Zuerst wird der Nachzügler in seine Position (135) in die Input-Queue eingefügt. In der Rückroll-Phase wird in der State-Queue von A nach einem Zustand gesucht, dessen LVT kleiner als 135 ist: in diesem Fall die Zeit 119. Dieser Zustand wird wiederhergestellt. Zustände mit LVT größer als 135 werden aus der State-Queue gelöscht, da sie sich nun zu früh bearbeitet wurden.

Die Rücknahme-Phase

Hier müßen alle Seiteneffekt rückgängig gemacht werden, die durch Nachrichten enstanden sind die A nach t=135 geschickt hat. Dies geschiet ganz einfach: *A sendet die Anti-Nachrichten dieser Nachrichten.*
In Bild 9 werden 5 Anti-Nachrichten verschikt.
Beispiel: Zur Simulationszeit 141 ist eine Nachricht nach B gesendet worden. Wir konstruieren die Anti-Nachricht, fügen eine Kopie in die Output-Queue von A und senden eine nach B. In der Output-Queue von A trifft sich die Anti-Nachricht mit der normalen Nachricht und werden beide gelöscht.

Die andere Anti-Nachricht wird in die Input-Queue von B geschrieben. Hier gibt es 3 mögliche Fälle:

1. Sie erreicht B in dessen Zukunft. In diesem Fall vernichtet die Anti-Nachricht die Originalnachricht sofort. Nach außen sieht es so aus, als wäre die Nachricht nie gesendet woeden.
2. Sie erreicht B in dessen Vergangenheit. Auch hier vernichten sich beide Nachrichten. Da die Anti-Nachricht in der Vergangenheit angekommen war, muß B nun einen Rollback durchfihren bis zur Zeit 142. Neue Anti-Nachrichten können dabei enstehen.
3. Die Anti-Nachricht erreicht B vor der normalen Nachricht. Je nach Zeit kann auch hiet ein Rollback geschehen. Wenn die positive Nachricht eintrifft, werden sie beide gelöscht.

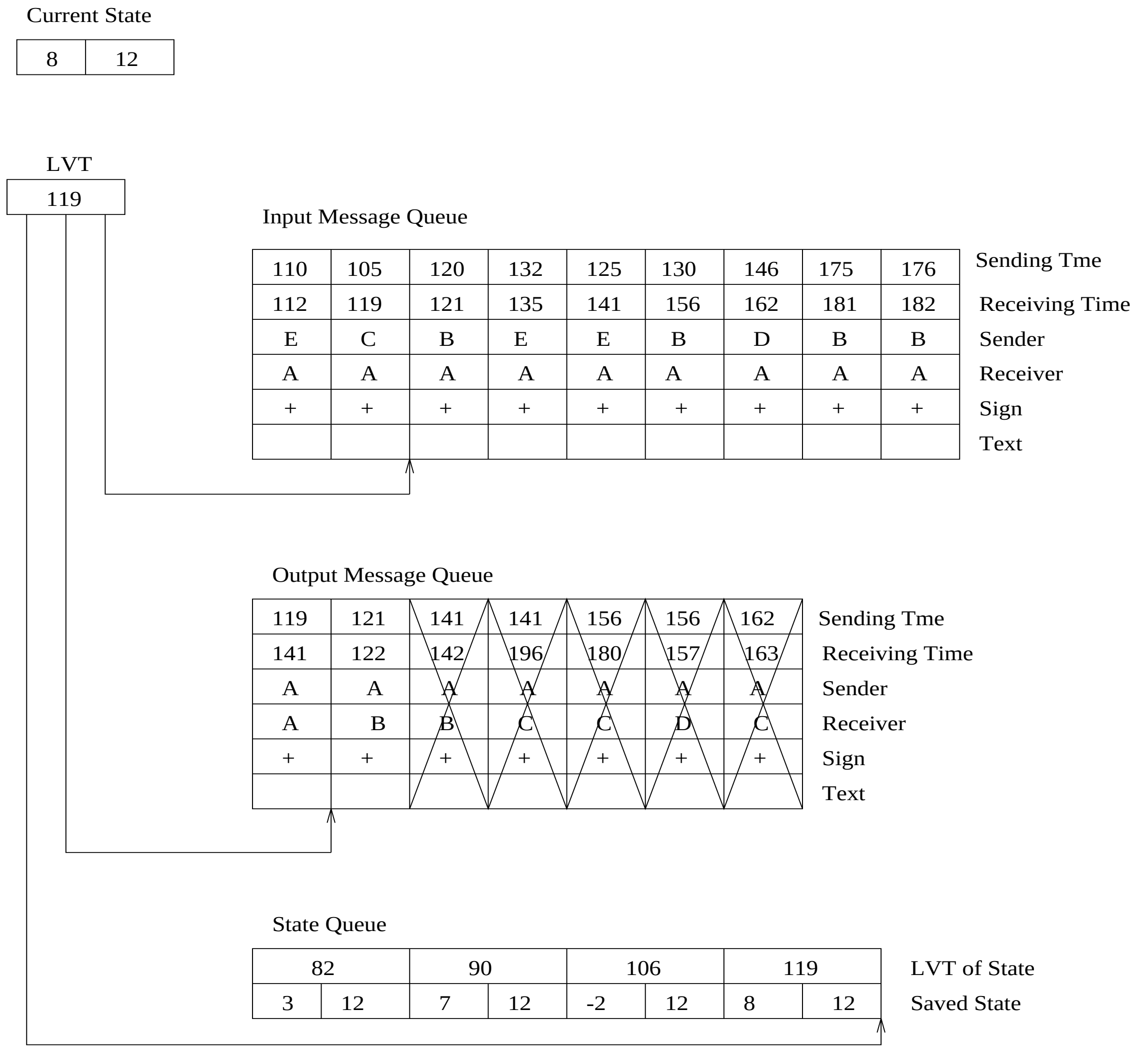

Abbildung 9: Vor der Rücknahme-Phase

Die Vorrück-Phase

In der Phase führt das Objekt ganz normal seine Simulation aus, außer wenn es eine Nachricht zu senden hat. Diese Nachricht wird nicht gesendet, da die Output-Queue des Senders und die Input-Queue des Empfängers bereits Kopien besitzen. E s werden nur lokale Aktionen durchgeführt. Bild 7 stellt A dar, nachdem 5 Nachrichten gelöscht worden sind. Die Nachricht zur Zeit 121 wurde nicht gestrichen, da A sonst die gleichen Berechnungen von 119 bis 135 durchführen würde. Dies dient der Optimierung, da dadurch einige rollbacks manchmal vermieden werden. Im Bild 8 hat A den korrekten Zustand erreicht, um den Nachzügler zu verarbeiten und mit der Simulation fortzufahren.

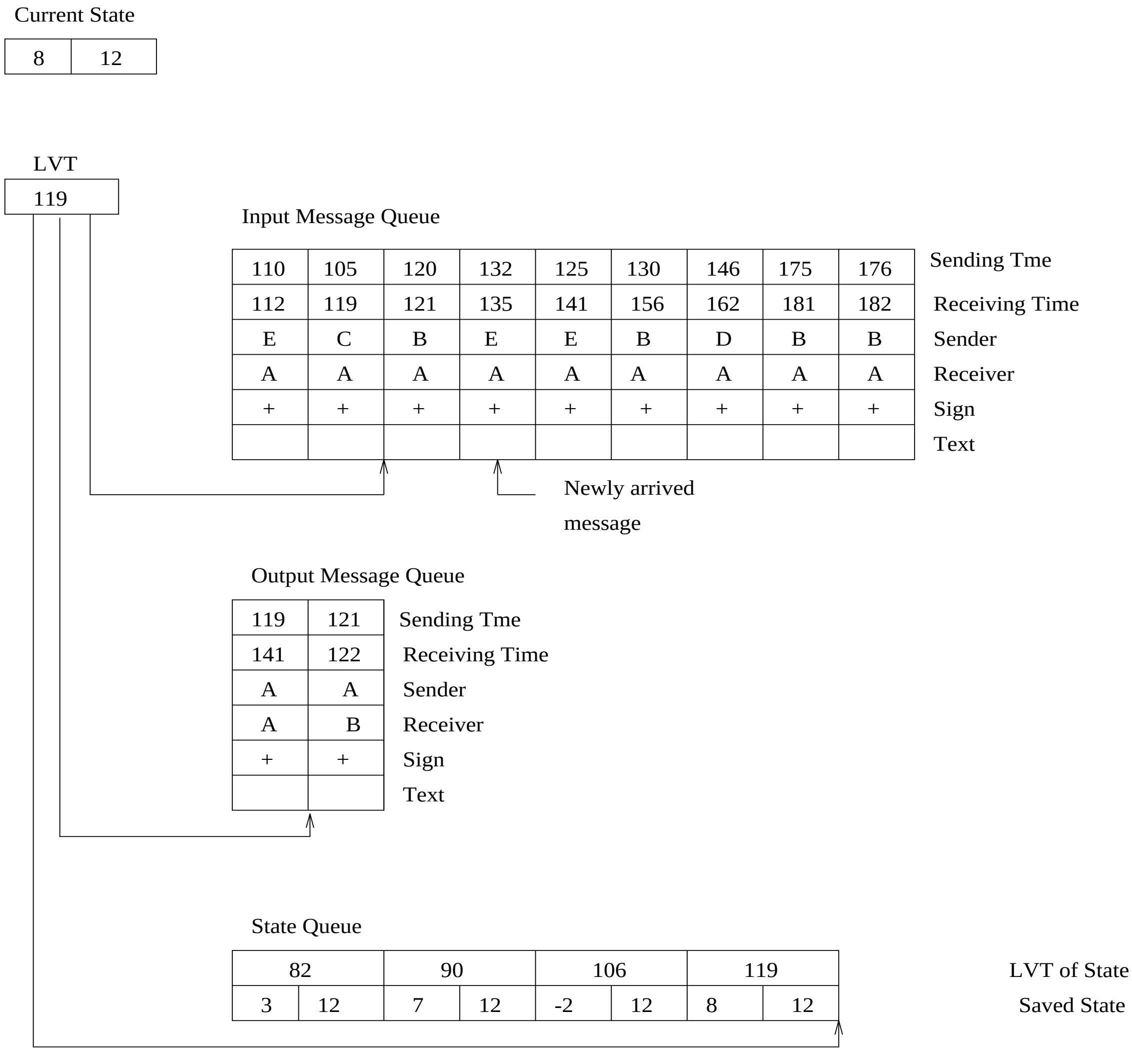

Abbildung 10: Nach der Rücknahme-Phase

Leistungen:
Obwohl das Time Warp Mechanismus keine Deadlocks erzeugt, könnte man meinen, daß es zu langsam ist. Man könnte sich ein Objekt voirstellen, daß durch seinen Rollback eine Kettenreaktion von Rollbacks von anderen Objekten auslöst. Im prinzip ist dies möglich. Es gibt aber Argumente dafür, daß trotz den Rollbacks relativ schnell bleibt.

- Wenn ein Objekt einen Rollback durführt und dabei Anti-Nachrichten verschickt, ist die Anzahl der Anti-Nachrichten proportional dazu, wie weit andere Objekte zurückrollen müßen. In Simulationen die fast synchron laufen, ist aber die Anzahl von Anti-Nachrichten klein. Weiterhin führt ein rollback dazu, daß die Simulation synchronisiert wird. Dadurch wird in naher Zukunft kein rollback mehr stattfinden.

- Wenn Anti-Nachrichten verschickt werden, wird nur ein Teil von ihnen in der Vergangenheit des empfängers ankommen und dort weitere rollbacks auslösen. Die Anzahl der Rollbacks ist also kleiner als die Anzahl weiter verschickter Anti-Nachrichten.

- Falls ein Objekt mehrere Anti-Nachrichten bekommt, muß es *nur ein rollback* durchführen.