



Hauptseminar
Semantik der UML v.2.0

Semantik kommunizierender reaktiver Objekte
in eingebetteten Realzeitsystemen.

Bearbeiter: Sergius Schneider
Betreuer: Stefan Wagner

INHALTSVERZEICHNIS

1 EINLEITUNG	- 4 -
2 KONZEPTE UND ENTSCHEIDUNGEN	- 4 -
2.1 Kernsprache.....	- 5 -
2.2 PVS.....	- 5 -
2.3 Zeit.....	- 5 -
2.4 Activity Groups.....	- 6 -
2.5 Run To Completion	- 6 -
2.6 Operationsaufrufe	- 6 -
2.6.1 Ausgelöste Operationen.....	- 7 -
2.6.2 Primitive Operationen	- 8 -
2.7 Objekterzeugung	- 8 -
2.8 Generalisierung.....	- 8 -
3 ZUSTANDMASCHINENÜBERGÄNGE	- 9 -
4 STATUS EINES OBJEKTES	- 10 -
5 BESCHRIFTETE ÜBERGANGSSYSTEME	- 10 -
5.2 Semantik der Aktionsausführung	- 11 -
5.3 Auslösen eines Überganges in der Zustandmaschine.....	- 11 -
Untriggered Übergang.....	- 11 -
Call Event	- 12 -
Signal-Ereignis.....	- 13 -
5.4 Passing of Time	- 14 -
6 AUSFÜHRUNGSLÄUFE	- 14 -
7 ZUSAMMENFASSUNG	- 15 -
8 LITERATURVERZEICHNIS	- 16 -

1 Einleitung.

Diese Seminararbeit basiert auf [HvdZ03]: “*A Semantics of Communicating Reactive Objects with Timing. Specification and Validation of UML models for Real Time and Embedded Systems*” (SVERTS'03, Proceedings) 2003 von Jozef Hooman and Mark van der Zwaag (Department of Computing Science, University of Nijmegen, Netherlands).

Die Arbeit von Jozef Hooman and Mark van der Zwaag wurde im Kontext des EU-Projektes Omega (korrekte Entwicklung der eingebetteten Realzeitsysteme) durchgeführt. Dieses Projekt zielte darauf ab, die Qualität von Software für eingebettete Systeme durch den Gebrauch von formalen Techniken zu verbessern. Dieses Dokument ist besonders auf das Modellieren von *reaktiven eingebetteten Echtzeitsystemen* unter Verwendung von Klassendiagrammen und Zustandmaschinendiagrammen fokussiert. Das Ziel der Autoren war, eine formale Grundlage für die eindeutige Beschreibung dieser Systeme in UML zur Verfügung zu stellen.

Die Kernsprache, die in dieser Arbeit benutzt wurde ist nah an der Sprache des UML-Kernes, der in [HG97] beschrieben wird. Es hat sich herausgestellt, dass bei der Definition der formalen Semantik für diese Sprache viele Fragen entstanden. Diese betrafen zum Beispiel das Verschicken von Signalen, Synchronisierung der *Operation Calls*, usw. In diesem Dokument werden diese Probleme identifiziert und die Entscheidungen dargestellt, die getroffen werden müssen, um diese Probleme zu beheben. Die resultierende Semantik ist in der typisierten Logik des interaktiven Theorembeweislers *PVS* definiert (s. [ORS92, PVS]).

Es wird angenommen, dass der Leser bereits über die Kenntnisse der UML Notation verfügt, deswegen werden hier die grundlegenden Begriffe wie z.B. Zustandsmaschine, Zustand, Wächter usw. als bekannt vorausgesetzt.

Für weitere Informationen zu Grundlagen der UML kann [OMG03]: *Object Management Group. Unified Modeling Language Specification, Version 2.0 (Superstructure). Adopted draft, OMG 2003*, und Literatur aus dem Literaturverzeichnis benutzt werden.

2 Konzepte und Entscheidungen

Die *Reactive Objects* können mittels asynchroner Signale und synchronisierten *Operation Calls* kommunizieren. Der Begriff eines *Threads of Control* wird durch eine so genannte *Activity Group* beschrieben, die eine Menge der Objekte (Klasseninstanzen) ist (mit genau einem aktiven Objekt, das zu einem Zeitpunkt ausgeführt werden kann).

Die Formalisierung der aktiven Klassen und der zugehörigen Zustandmaschinen [RACH00] hängt stark mit dieser Arbeit zusammen. Dort wird ein beschriftetes Übergangssystem mit der algebraischen Spezifikationsprache CASL definiert, was

auch zu einer Anzahl von Fragen über die Deutung von UML Modellen führt. In [RACH00] wird der allgemeine Fall betrachtet. Zum Beispiel kann ein aktives Objekt einer willkürlichen Anzahl von *Threads* entsprechen und die Ereigniswarteschlange ist eine Multimenge von Ereignissen.

Die in diesem Dokument vorgestellten Entscheidungen basieren hauptsächlich auf der Rücksprache mit den industriellen Benutzern, auf den aktuellen UML - basierenden CASE - Werkzeugen für Echtzeitsysteme, mit dem Ziel, formale Verifikation der eingebetteten Systeme zu ermöglichen. Deswegen wurden mehr spezifischen Entscheidungen, wie ein einzelnes *Thread of Control* pro Objekt und eine einfache FIFO Ereigniswarteschlange getroffen.

2.1 Kernsprache

Es werden herkömmliche Klassendiagramme benutzt, mit Wertattributen und Bezugsattributen (Attribute, die sich auf einen Objekt beziehen) und Operationen. Verbindungen zwischen Klassen werden durch Bezugsattribute dargestellt. Andere Relationen beinhalten Erbschaft und Komposition. Es wird angenommen, dass alle Klassen eine Zustandmaschine haben (diese kann auch leer sein). Eine Zustandmaschine besteht aus einer Anzahl von Übergängen zwischen den Zuständen. Übergänge können einen Booleschen Wächter haben und können durch ein Signal oder einen Operationsaufruf ausgelöst werden. Es gibt einen *Aktionsteil* (eine nicht leere Liste der Aktionen) der ausgeführt wird, wenn der Übergang genommen wird.

2.2 PVS

PVS (Prototype verification system) ist ein Werkzeug, das für maschinelles Beweisen benutzt wird. Es existiert dazu auch die PVS-Spezifikationssprache, die benutzt wird, um die informell definierten Modelle formal darzustellen und mithilfe der PVS-Software maschinell zu beweisen, dass das Modell korrekt ist. [ORS92, PVS]

2.3 Zeit

Zeit wird durch nichtnegative reelle Zahlen repräsentiert. Zeitbegrenzungen in den Zustandmaschinen werden mithilfe der *lokalen Uhren* ausgedrückt, ein Objekt kann seine *Uhren* auf null setzen. (Ein Objekt kann, also, mehrere *Uhren* haben).

Das Verstreichen von Zeit wird durch eine globale Verzögerungsaktion modelliert, die die globale Zeit erhöht und alle Systemtakte dementsprechend justiert.

Wenn ein Zustandsmaschinenübergang in irgendeinem Objekt durchgeführt wird, dann soll keine Zeit zwischen dem Auslösen des Überganges (eine Synchronisierung mit dem Aufruferobjekt in Falle eines Aufrufereignisses oder der Annahme eines Signals im Falle eines Signalereignisses) und der Auswertung der Wächter vergehen.

Mit dem Auslösen ändert sich der Zustandmaschinenzustand des Ausführungsobjektes von dem aktuellen Zustand zum Zielzustand. Die Ausführung einer Aktion ist ein atomarer Schritt des Systems, der keine Zeit dauert. *Passing of time* wird vor und zwischen der Ausführung der Übergangsaaktionen erlaubt. Während der Ausführung der Aktionen bleibt das Objekt in der Zielposition.

2.4 Activity Groups

Die Menge der Objekte, die zurzeit *alive* (lebendig) sind, heißt *Activity Group*. Zu jedem beliebigen Zeitpunkt hat in jeder *Activity Group* immer genau ein Objekt die Programmkontrolle. Dieses Objekt wird *Control Object* der Aktivitätsgruppe genannt. Während der Ausführung kann sich die Programmkontrolle innerhalb einer Aktivitätsgruppe von einem Objekt auf ein anderes verschieben. Programmkontrolle kann dem anderen Objekt übergeben werden, falls ein Aufruf innerhalb derselben Aktivitätsgruppe passiert, andernfalls kann ein Objekt die Programmkontrolle nur verlieren, wenn er stabil ist.

Der Begriff der *Activity Groups* ist mit dem des *Threads of Control* vergleichbar: ein aktives Objekt entspricht einem *Thread of Control*, und höchstens ein *Thread* ist in jedem Objekt aktiv. Um ein Durcheinander mit z.B. Java *Threads* zu vermeiden, wird anstatt der Bezeichnung *Thread of Control* die Bezeichnung *Activity Group* verwendet.

2.5 Run To Completion

Run-to-completion Semantik, wird wie im Vorgehensmodell ROOM [SGW94] definiert. Das heißt, die Ausführung wird nach einem Signal- oder Operationsaufruf fortgesetzt, bis ein stabiler Zustand erreicht ist.

Ein Objekt ist im stabilen Zustand, wenn er aktiv ist und kein *ungetriggerteter* Übergang möglich ist, d.h. weitere Ausführung ist nur dann möglich, nachdem man einen Signal- oder Operationsaufruf empfangen hat.

2.6 Operationsaufrufe

Operationsaufrufe werden *synchron* durchgeführt (d.h. ein Operationsaufruf kann nur dann durchgeführt wenn das aufgerufene Objekt bereit ist, ihn anzunehmen, und das Objekt das den Aufruf macht wird bis Aufrufrückkehr blockiert), während Signal-basierende Kommunikation *asynchron* ist (das Signal wird in eine Warteschlange am Empfänger platziert, der Absender kann fortfahren).

Es werden zwei Arten von Operationen definiert:

- ausgelöste (*triggered*) Operationen, die als Auslöser für einen Übergang auftreten können und als Teil der Zustandmaschine des aufgerufenen Objekts implementiert werden

- *primitive* Operationen oder Methoden, die durch ein Stück Code implementiert werden und nicht als Auslöser für Zustandsübergänge verwendet werden können. Der Einfachheit halber wird angenommen, dass der Code keine Operationsaufrufe enthält.

Bemerkung. In der UML Notation wird kein Unterschied zwischen den beiden Operationstypen gemacht. Dort werden die beiden einfach Aktivitäten genannt.

2.6.1 Ausgelöste Operationen

Hinsichtlich der ausgelösten Operationen stellen sich Fragen über den Programmkontrollfluss und darüber, wann man das Resultat zurückführt und wann Kontrollübergabe passiert; wenn das aufgerufene Objekt stabil wird, oder sofort, wenn das Resultat vorhanden ist?

Die hier vorgestellte Lösung lautet:

Eine erfolgreiche Operationssynchronisierung erfordert, dass das Aufruferobjekt die Programmkontrolle hat und

- wenn das aufgerufene Objekt der gleichen *Activity Group* angehört, dann muss es bereit sein, den Aufruf anzunehmen und die Programmkontrolle wird vom Aufruferobjekt zum aufgerufenen Objekt übergeben
- wenn das aufgerufene Objekt in einer anderen Aktivitätsgruppe ist, dann muss es dort die Programmkontrolle haben und es muss bereit sein, den Aufruf anzunehmen. Das Aufruferobjekt behält die Programmkontrolle in seiner Gruppe bei. Das aufgerufene Objekt muss entweder die Programmkontrolle in seiner Aktivitätsgruppe bereits haben oder übernehmen.

Ausführung einer Rückgabeaktion bringt einen Wert (Änderung eines Attributes vom Aufruferobjekt) zurück und hat den Effekt, dass das Aufruferobjekt nicht mehr angehalten wird, aber es braucht, nicht zu einer Programmkontrollübergabe zu führen, wenn das aufgerufene Objekt und Aufruferobjekt in der gleichen *Activity Group* sind, dann bekommt das Aufruferobjekt die Programmkontrolle, sobald das aufgerufene Objekt stabil wird.

Hier wird nicht die erneute Ausführung der ausgelösten Operationen erlaubt, d.h. während der Ausführung der ausgelösten Operationen ist die Annahme neuer ausgelöster Operationen nicht möglich.

2.6.2 Primitive Operationen

Eine primitive Operation ist eine Methode, die in einem lokalen Zustand des aufgerufenen Objekts ausgeführt werden kann. Aus Gründen der Einfachheit wird der resultierende Wert sofort im Moment der Anforderung zurückgeliefert.

Die Frage ist, wann primitive Operationen durchgeführt werden können, z.B. wie ausgelöste Operationen nur in einem stabilen Zustand der nicht angehaltenen Objekte?

Antwort: Aufruf der primitiven Operationen ist in jedem möglichem Zustand erlaubt, d.h. das aufgerufene Objekt kann instabil oder angehalten sein.

Ein ähnliches Problem wird in [TS03] für rekursive Aufrufe und *Callbacks* besprochen. Dort wird nicht zwischen ausgelösten und primitiven Operationen unterschieden, es werden aber zwei Arten von Zustandsdiagrammen vorgestellt.

Der Grundgedanke hinter der Lösung in diesem Dokument ist, dass es nicht wünschenswert ist, jedes Verhalten durch Zustandmaschinen zu modellieren. Die Semantik die hier vorgestellt wird, wird nur dazu benutzt, um hauptsächlich reaktives Verhalten zu modellieren, d.h. die Interaktion zwischen Objekten und anderen Operationen können bequemer in irgendeiner Sprache ausgedrückt werden (oder abstrakt mithilfe OCL beschrieben werden).

2.7 Objekterzeugung

Ein Objekt kann ein neues Objekt erstellen, zu dem es eine Referenz bekommt. Die Ausgangsattributwerte des neuen Objektes werden vollständig durch seine Klasse definiert und nicht durch das Objekt das es erstellt hat.

2.8 Generalisierung

Hier werden die üblichen Definitionen der Vererbung benutzt. (s. [HK00]). Die Hauptfrage ist, in welchem Umfang das Verhalten einer Superklasse durch eine Unterklasse übernommen wird. In Übereinstimmung mit dem gegenwärtigen Gebrauch von Vererbung in den industriellen Anwendungen, wird zugelassen, dass eine Unterklasse das Verhalten einer übernommenen Operation überschreiben kann. Dann bleibt die Frage, ob die Definition einer ausgelösten Operation durch eine Unterklasse übernommen wird, wenn die Unterklasse sie nicht überschreibt. Es wird folgendes festgelegt:

- Wenn eine Unterklasse eine Zustandmaschine hat, dann überschreibt sie die Zustandmaschine des Elternteils vollständig
- andernfalls übernimmt sie die Zustandmaschine der Superklasse.

3 Zustandmaschinenübergänge

Um die Semantik der Zustandmaschinenübergänge besser beschreiben zu können, sollen die Zustandmaschinenaktionen detailliert beschrieben werden.

Zustandmaschinenaktionen können folgende Formen haben:

- *call(a, ref, op, exp)*: Ein Aufruf der Operation *op* vom Objekt *ref*, mit dem Wert des Datenausdrucks *exp* als Parameter. Falls ein Rückgabewert benötigt wird, wird er dem Attribut *a* zugewiesen.
- *return(result)*: Gibt den Wert des Ausdrucks zurück.
- *emitSignal(ref, signame, exp)*: Senden eines Signals *signame*, mit dem Wert des Datenausdruck *exp* als Parameter, zum Objekt, der durch Verweis *ref* definiert ist
- *assign(a, exp)*: Eine lokale Anweisung, weist den Wert des Datenausdrucks *exp* dem Attribut *a* zu.
- *methodCall (a, ref, meth)*: Ein Aufruf der Methode *meth* vom Objekt *ref*. Das Resultat wird dem Attribut *a* zugewiesen.
- *create(ref, c)*: Erzeugung eines Objektes vom Klass *c*, *ref* zeigt auf dieses neue Objekt
- *skip*: Tu nichts
- *reset(x)*: Clock *x* wird auf 0 gestellt.

Ein Triggerevent ist von einer der folgenden Formen:

- *callEvent(op, a)*: Für das Auslösen durch einen Aufruf der Operationen *op*. Der Parameter des Aufrufs wird dem Attribut *a* zugewiesen
- *signalEvent(signame, a)*: Für das Auslösen durch ein Signal *signame*. Der Parameter des Signals wird dem Attribut *a* zugewiesen.

Bemerkung. In der UML Spezifikation wird nicht wie hier zwischen zwei Triggereventtypen unterschieden.

Ein Zustandmaschinenübergang ist definiert als *record* mit Attributen für seine Quell- und Zielposition, *guard* (Wächter), *trigger* und *actions*. Ein Übergang hat auch ein Attribut für die Klasse, zu der er gehört. Das *actions field* enthält eine nicht leere Liste der Zustandmaschinenaktionen.

4 Status eines Objektes

Der Status eines Objektes kann einen von vier Werten annehmen:

- ein Objekt ist *dormant* (schlafend), bis es erzeugt wird
- ein Objekt ist *free* (frei), wenn es nicht gerade einen ausgelösten Operationsaufruf verarbeitet
- der Objektstatus ist *processingCall (caller, a)*, wenn das Objekt einen Aufruf vom Aufruferobjekt angenommen hat, für den es noch nicht das Resultat zurückgebracht hat. Der Rückgabewert muss dem Attribut *a* des Aufruferobjekts zugewiesen werden
- der Status ist *completingCall (caller)* wenn das Aufruferobjekt von der gleichen *activity group* ist und das Objekt hat, nachdem es den Rückgabewert ausgegeben hat, den Aufruf beendet, d.h. Durchlauf zum *completion/become stable* durchgeführt, bevor die Programmkontrolle zum Aufruferobjekt zurückgebracht wurde.

Gesamte Systemverhalten beginnt im Ausgangszustand, in dem es ein einzelnes *nondormant* (nicht schlafendes) *Root-Objekt* gibt, s. Abschnitt 6.

5 Beschriftete Übergangssysteme

5.1 Allgemeines

Einige zusätzliche Definitionen hinsichtlich der Stabilität und dem Programmkontrollfluß:

- ein Objekt ist *alive* (lebendig), wenn er nicht schlafend ist
- ein Objekt ist *ready* (bereit), wenn er lebendig ist, eine leere Aktionsliste hat und nicht angehalten ist
- ein Objekt ist *executing*, wenn er lebendig ist, eine nicht leere Aktionsliste hat und nicht angehalten ist
- Übergang ist *locally enabled* für ein Objekt in irgendeinem Zustand, wenn er zu der Objektzustandmaschine gehört, seine Quelle ist die gegenwärtige Position des Objektes, und der boolesche Wächter *true* ist
- Ein Objekt ist *stable* (stabil) in irgendeinem Zustand, wenn es nicht z.Z. einen Aufruf verarbeitet, und alle lokal möglichen Übergänge haben einen Auslöser.

5.2 Semantik der Aktionsausführung

Hier wird beschrieben, wie die Ausführung einer Zustandmaschinenaktion einen Zustand ändert. Diese Aktion soll das erste Element der Aktionsliste des ausführenden Objektes sein, nach der Ausführung wird sie von dieser Liste entfernt.

return(exp) ein Ausführungsobjekt, der einen Aufruf für irgendeinen Aufruferobjekt verarbeitet, kann eine Rückgabeaktion mit Resultat *exp* ausführen. Es wird gefordert, dass das Aufruferobjekt *alive* (lebendig) ist. Das Resultat ist, dass das Aufruferobjekt nicht mehr angehalten wird, der Wert des Ausdruckes wird dem gekennzeichneten Attribut des Aufruferobjekts zugewiesen. Wenn das Aufruferobjekt der gleichen Aktivitätsgruppe gehört, dann wird das aufgerufene Objekt frei, andernfalls wird sein Status *completingCall* und es muß bis zur Beendigung des Aufrufs (bis *stable*) laufen, erst danach wird die Programmkontrolle an Aufruferobjekt zurückgegeben.

assign(a, exp) Das Ausführungsobjekt weist den Wert des Ausdrucks *exp* seinem Attribut *a* zu.

emitSignal(ref, sn, exp) Ein neues Signal mit Signalname *sn* und dem Wert des Ausdruck *exp* wird in der Signalwarteschlange des Empfängers eingefügt

methodCall(a, ref, meth), das Resultat des Aufrufs einer Methode (*primitive operation*) wird als Wert im lokalen Zustand des aufgerufenen Objektes (der Objekt, auf den sich *ref* bezieht) zurückgegeben. Das Resultat wird dem Attribut *a* vom Aufruferobjekt zugewiesen. Das aufgerufene Objekt muss *alive* sein (*nondormant*).

skip, Keine Änderungen.

create(ref, c), Das Ausführungsobjekt kann ein neues Objekt vom Klasse *c* erstellen (mit Referenz *ref*). Das neue Objekt wird wie folgt initialisiert: seine Klasse ist *c*, es ist *free* und nicht angehalten, seine Signalwarteschlange und *action list* sind leer.

reset(x), *Clock x* ändert sein Wert auf null für das Objekt das Operation ausführt.

5.3 Auslösen eines Überganges in der Zustandmaschine

Die Semantik des Auslösens eines Zustandmaschinenüberganges für ein Objekt hängt von der Art des Auslösers ab.

Untriggered Übergang

Übergang kann für einen Objekt lokal ausgelöst werden, wenn er *untriggered* ist. Es wird gefordert, dass der Objekt *ready* ist (d.h., ist *alive* (lebendig), hat eine leere

action list und wird nicht angehalten), bereit ist Programmontrolle zu übernehmen und dass *untriggered* Übergang *local enabled* ist. Das Resultat ist, dass der Übergang genommen wird (die Zielposition wird die gegenwärtige Position des Objektes, die Tätigkeitsliste des Überganges wird in die *action list* des Objektes kopiert) und dass das Objekt die Programmkontrolle bekommt.

Call Event

Das Auslösen eines Überganges mit einem Aufrufereignis erfordert eine Synchronisierung mit dem Aufruferobjekt der Operation: es wird angefordert, dass es einen Aufruferobjekt gibt, der gerade ausgeführt wird und dessen erste *action* ein Aufruf zum Objekt *p* ist, der gleich dem Auslöserereignis des Überganges ist.

Außerdem ist es erforderlich, dass das aufgerufene Objekt stabil ist, und wenn das Aufruferobjekt einer anderen Aktivitätsgruppe gehört, dann muss das aufgerufene Objekt in der Lage sein, Programmkontrolle zu übernehmen. Schließlich muss der Übergang für das aufgerufene Objekt lokal möglich sein. (s. Definition in Bild 4).

```
acceptCallCondition(t, caller, s, p): bool =
  stable?(s`F(p)) AND executing?(s`F(caller))
  AND
  ((NOT samegroup?(s`F(p), s`F(caller)))
   IMPLIES takeControlCondition(s, p))
  AND
  LET action = car(s`F(caller)`alist) IN
  call?(action)
  AND
  callTriggersTransition?(action, t, s, caller, p)

acceptCallEffect(t, caller, s, p): State =
  IF executing?(s`F(caller)) AND
  call?(car(s`F(caller)`alist)) AND
  callEvent?(t`trigger)
  THEN
  LET action = car(s`F(caller)`alist) IN
  takeControlEffect(basicTriggeringEffect(t, s, p), p)
  WITH
  [(F)(p)(status):= processingCall(caller, a(action)),
   (F)(p)(val)(aval)(a(t`trigger)):= exp(action)(s`F(caller)`val),
   (F)(caller)(alist):= cdr(s`F(caller)`alist),
   (F)(caller)(suspended):= TRUE ]
  ELSE s ENDIF
```

Bild 4. Annahme eines Aufrufs.

Resultat der Synchronisierung ist, dass das der Übergang im aufgerufenen Objekt ausgelöst wird, (Objekt erhält die Programmkontrolle, ändert seine Position zum Ziel des Überganges und kopiert die *action list* des Überganges). Das Aufruferobjekt wird angehalten (es verliert die Programmkontrolle, wenn das aufgerufene Objekt der gleichen *activity group* gehört). Der Status des aufgerufenen Objekts wird *processingCall(caller, a)*, dem Attribut *a* wird das Resultat zugewiesen und der Wert des Aufrufparameters wird dem gekennzeichneten Attribut zugewiesen.

Signal-Ereignis

Im Bild 5 wird definiert, wie ein Übergang *t* für Objekt *p* durch ein Signal ausgelöst wird. Dabei ist *n* die Position des auslösenden Signals in der Signalwarteschlange *p*. Es wird gefordert, dass dieses Signal das erste Element der Warteschlange ist, die einen Übergang auslöst. Wir benötigen weiter, dass der Objekt *stable* ist und dass es die Programmkontrolle übernehmen kann.

Das Resultat ist, dass der Übergang ausgelöst wird und das auslösende Signal wird aus der Signalwarteschlange entfernt.

t: VAR (transitions); *n*: VAR nat; *s*: VAR State; *p*: VAR Object

```
acceptSignalCondition(t, n, s, p): bool = LET os = s.F(p) IN
  stable?(os) AND takeControlCondition(s, p)
  AND
  nonempty?(os.sq)
  AND n < length(os.sq)
  AND signalTriggersTransition?(nth(os.sq, n), t, os)
  AND
  FORALL (m: below[n]): NOT EXISTS (t1: (transitions)):
    signalTriggersTransition?(nth(os.sq, m), t1, os)
```

```
acceptSignalEffect(t, n, s, p): State =
  IF nonempty?(s.F(p).sq) AND signalEvent?(t.trigger)
  THEN takeControlEffect(basicTriggeringEffect(t, s, p), p)
  WITH [(F)(p)(sq):=
    cleanUp(s.F(p).sq, n, s.F(p).class, s.F(p).loc)]
  ELSE s ENDIF
```

Bild 5. Annahme eines Signals.

5.4 Passing of Time

Verzögerung mit Parameter dt (*delay time*) ($dt > 0$), bedeutet, dass Verzögerungszeit dt zu globaler Zeit addiert wird und dass alle lokale *clocks* dementsprechend justiert werden. (PVS-Definition s. Bild 6).

u : VAR Time; dt : VAR DelayTime

$\text{delayClocks}(\text{val}: \text{Valuation}, u): \text{Valuation} =$
 $\text{val WITH } [(cval):= \text{LAMBDA } (x: \text{Clock}): \text{val } cval(x) + u]$

$\text{delayEffect}(dt, s): \text{State} =$
 $s \text{ WITH } [(time):= s \text{ time} + dt,$
 $(F):= \text{LAMBDA } p:$
 $s \text{ F}(p) \text{ WITH } [(val):= \text{delayClocks}(s \text{ F}(p) \text{ val}, dt)]]$

Bild 6. *Passing of time.*

6 Ausführungsläufe

Zu den Verifikationszwecken wird das Verhalten eines Systems als Satz unendlichen Sequenzen von den Schritten definiert.

Solche Schrittsequenzen:

$$s_0 \xrightarrow{L^0} s_1 \xrightarrow{L^1} \dots$$

heißen Ausführungsläufe, oder *runs* des Systems. Dadurch können die Korrektheitseigenschaften der Systeme ausgedrückt werden.

Run werden als eine Sequenz von Schritten definiert (s. Bild 7). *Runs* des Systems werden als Menge von *runs* definiert, mit folgenden Beschränkungen

- Die Invariante *inv* ist erfüllt für jeden *run*, wenn sie für alle seine Zustände gilt, aber auch für alle Zwischenzustände, die durch das *passing of time* erreicht werden.
 - Der erste Zustand eines *runs* muss der *initial state* (Initialzustand) sein:
in diesem Zustand ist die globale Zeit gleich 0, und das Objekt *root* ist das einzige *alive* Objekt. Dieses *root* Objekt gehört der *root* Klasse, ist frei, nicht angehalten, hat Programmkontrolle und ist in der Ausgangsposition seiner Zustandmaschine. Seine Signalwarteschlange und *actions list* sind leer.
-

```
Run: TYPE = sequence[(steps)]

i, j: VAR nat; u: VAR Time; s: VAR State; r: VAR Run; l: VAR Label

nonZeno(r): bool = FORALL i, u: EXISTS j:
  r(i+j) `current` time > r(i) `current` time + u

delayTime(l): Time =
  CASES label OF delay(u): u ELSE 0 ENDCASES

invOK(s, u): bool =
  FORALL p: ready?(s `F(p)) IMPLIES
    inv(s `F(p) `class, s `F(p) `loc, delayClocks(s `F(p) `val, u))

invOK(r): bool =
  FORALL i, u: u <= delayTime(r(i) `label)
    IMPLIES
      invOK(r(i) `current, u)

runs: setof[Run] = LAMBDA r: (FORALL i: r(i) `next = r(i+1) `current)
  AND
  r(0) `current = initialState AND nonZeno(r) AND invOK(r)
```

Bild 7. Definition der Ausführungsläufe.

Anmerkung. Ausführungsläufe die zu *time deadlock* führen (einem Zustand, in dem keine weiteren Aktionen, auch Verzögerungen, nicht ohne Verletzung der Invarianten durchgeführt werden können), werden von der Menge der *runs* ausgeschlossen. Jedoch könnte man an solchem deterministischen Verhalten interessiert sein. Ein Trick um nichtdeterministische *runs* für deterministische Ausführung zu definieren, ist in der folgenden Erweiterung:

im Fall von *time deadlocks* kann eine *escape action* durchgeführt werden zu einem leeren Zustand d.h. einem Zustand, in dem alle Gegenstände schlafend sind. Von solch einem Zustand ist die einzige weitere mögliche Aktion das *passing of time*, während die Invariante immer (trivial) gültig ist.

7 Zusammenfassung

In diesem Dokument ([HvdZ03]) wurde eine formale operationale Semantik für den Kern der UML Sprache zum Modellieren von reaktiven Realzeitsystemen, mit einem Fokus auf die Kommunikation zwischen den reaktiven Objekten dargestellt.

Objekte gehören zu einer *activity group* und können mittels der asynchronen Signale oder der synchronen Operationen kommunizieren. Obwohl die Grundgedanken über

die beabsichtigte Semantik ziemlich klar waren, war es bei weitem nicht trivial, diese exakt zu bilden und es musste eine große Zahl von Fragen über Vererbung, Programmsteuerung, primitive und ausgelöste Operationen und Signale gelöst werden.

In diesem Dokument wurde die UML Notation zwar sehr stark verfeinert, (z.B. Unterscheidung zwischen ausgelösten und primitiven Operationen (Aktivitäten in UML), oder Einführung der Begriffe, wie Aktivitätsgruppe, Control Object, Status eines Objekts, die in UML Notation nicht definiert sind), es gibt aber keinen Widerspruch zu UML Spezifikation.

Eine sehr wichtige Eigenschaft dieser Arbeit liegt daran, dass die Semantik in der Spezifikationssprache des Werkzeugs PVS dargestellt wurde, was die Verifizierung des Modells (Korrektheitsbeweis) ermöglicht hat.

8 Literaturverzeichnis

[GO03] S. Graf and I. Ober. A real-time profile for UML and how to adapt it to SDL. In Proceedings of SDL 2003 Forum, LNCS, 2003.

[HG97] D. Harel and E. Gery. Executable object modelling with state charts. IEEE Computer, pages 31-42, 1997.

[HK00] D. Harel and O. Kupfermann. On the behavioural inheritance of state-based objects. In Proceedings, 34th Int. Conf. on Component and Object Technology. IEEE Computer Society, 2000.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In 11th Conference on Automated Deduction, volume 607 of Lecture Notes in Artificial Intelligence, pages 748-752. Springer-Verlag, 1992.

[PVS] PVS. Information, documentation, download. Available from SRI Computer Science Laboratory, <http://pvs.csl.sri.com/>.

[RACH00] G. Reggio, E. Astesiano, C. Choppy, and H. Husmann. Analysing UML Active classes and associated state charts - a lightweight formal approach. In Proceedings FASE 2000 - Fundamental Approaches to Software Engineering, LNCS 1783, pages 127-146, 2000.

[SGW94] B. Selic, G. Gullekson, and P.T.Ward. Real-Time Object-Oriented Modelling. John Wiley & Sons, 1994.

[TS03] J. Tenzer and P. Stevens. Modelling recursive calls with UML state diagrams. In Proc. FASE 2003 - Fundamental Approaches to Software Engineering, pages 135-149. LNCS 2621, Springer-Verlag, 2003.