Formale Semantik für asynchronen Methodenaufruf

U. Nestmann **M. Steffen** **T. Stroup**

Erlangen, May 1995 TR-I7-95-11

erscheint im Tagungsband zum 5. GI/ITG-Fachgespäch

Formale Beschreibungstechniken für verteilte Systeme

Formale Semantik für asynchronen Methodenaufruf*

Uwe Nestmann†, Martin Steffen, Terry Stroup
Informatik VII, Universität Erlangen
Martensstraße 3, 91058 Erlangen

Zusammenfassung

Das Methodenaufrufschema *wait-by-necessity* zum Verbergen expliziter Parallelität hinter asynchronen Wertzuweisungen wird exemplarisch durch eine *Strukturelle Operationelle Semantik* (SOS) formalisiert. Diese Spezifikation repräsentiert ein eindeutiges Referenzmodell und wurde darüberhinaus mittels einer semantisch korrekten Übersetzung in den π-Kalkül in eine prototypische Implementierung überführt. Sie dienen neben der Validierung der Intuition des Programmiersprach-Ingenieurs auch als Basis für den Entwurf korrekter Compiler sowie zum Testen von Beispielprogrammen. Das vorliegende Papier konzentriert sich auf die Darstellung der SOS-Variante.

Themen: Objektorientierung, theoretische Modelle, FDT-basierte Implementierung

1 Asynchroner Methodenaufruf

Ein wesentliches Merkmal objektorientierter Programmiermodelle ist die Kapselung von Zuständen und, damit eng verbunden, die eingeschränkte Manipulation von Zuständen durch dedizierte Operationen, Methoden genannt. Parallelität in solchen Modellen wird durch verschiedene Mechanismen realisiert: Objekte können aktiv sein und gleichzeitig zur Berechnung beitragen; in manchen Modellen können innerhalb von Objekten gleichzeitig verschiedene Aktivitäten ausgeführt werden. Problematisch ist die Kontrolle über die auftretenden Interferenzen zwischen den Aktivitäten [Jon93]. In den letzten Jahren wurden neue Konzepte der parallelen objektorientierten Programmierung entwickelt, bei denen auch sequentielle Methoden durch geringfügige Erweiterung ertragreich eingesetzt werden können.

Ein solches verträgliches Parallelitäts-Prinzip ist das des asynchronen Methodenaufrufs mittels *wait-by-necessity* [Car90]. Es besteht im syntaktischen Verbergen expliziter *fork-and-join*-Parallelität hinter nichtblockierenden, d.h. asynchronen, Wertzuweisungen. Nach einer solchen Zuweisung, deren Wert durch den Aufruf (*fork*) einer Methode erst berechnet werden muß, darf der aufrufende Prozeß weiterbearbeitet werden, bis er sich eventuell an einem Lesezugriff (*join*) blockiert, d.h. bis die vorher zugewiesene Variable referenzierend verwendet wird. Ist die asynchrone Berechnung zum Zeitpunkt des Lesezugriffs bereits abgeschlossen, so tritt keine Blockade des Prozesses ein (siehe Abbildung 1).

Dieses Konzept ist z.B. dann nützlich, wenn viele Ressourcen zur Verfügung stehen. Diese fordert man im Programm zunächst ohne weitere Beschränkung asynchron an, so daß Engpässe in der Bearbeitung erst dann entstehen, wenn auf die Ergebnisse der angestoßenen Teilberechnungen zugegriffen werden muß. Solche Abläufe kommen deshalb z.B. der Programmierung von Datenflußrechnern sehr nahe [RK94]. Die Semantik der asynchronen Abläufe sollte eng an der entsprechenden sequentiellen Interpretation angelehnt sein.

*gefördert durch die Deutsche Forschungsgemeinschaft im *SFB 182, Teilprojekt C2*
†nestmann@informatik.uni-erlangen.de

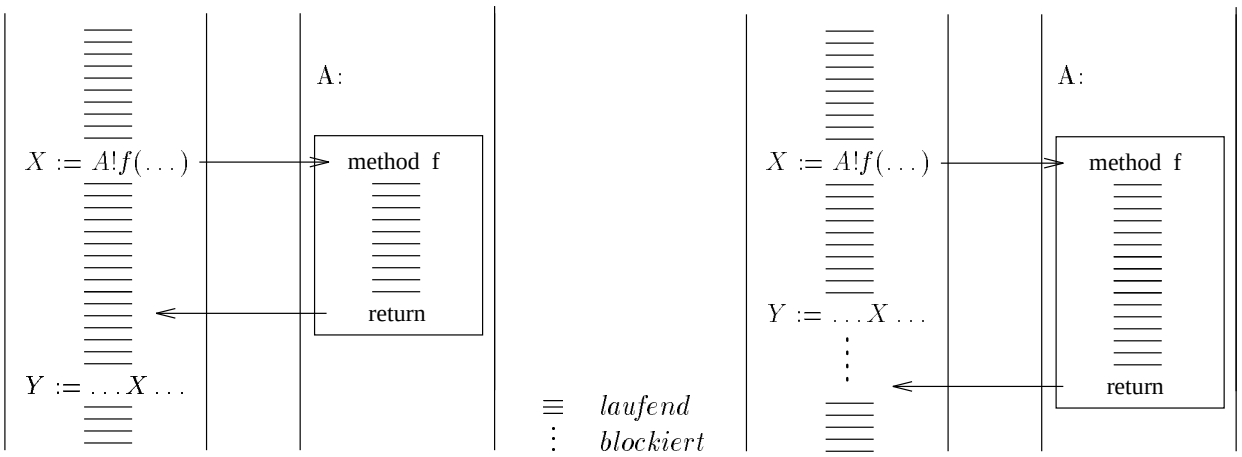

Abbildung 1: Aufrufschema

2 Die Beispielsprache CLOWN

Auf der Basis der theoretisch fundierten objektorientierten Sprache POOL [Ame92] entwickelten wir vermittels des Austauschs des dort verwendeten Methodenaufrufkonzeptes durch *wait-by-necessity* eine kompakte Beispielsprache, anhand derer die wesentlichen Aspekte des asynchronen Methodenaufrufs in Reinform untersucht werden können.

Struktur des Objektmodells Das unseren Untersuchungen zugrundeliegende Modell ist herkömmlich in dem Sinn, daß die die aus der objektorientierten Programmierung bekannten Begriffe wie *Klasse*, *Objekt* und *Methode* ihre herkömmliche Bedeutung haben.

Klasse Programmiersprachliche Einheit, die die gemeinsame Struktur gleichartiger Laufzeitobjekte festlegt: enthält eventuell mehrere Variablen- und Methodendeklarationen sowie eine Aktivitätsträgerbeschreibung.

Objekt Laufzeitgegenstand, der aus Klassenbeschreibungen dynamisch generiert wird. Objekte sind aktiv, besitzen Identität und einen internen gekapselten Zustand, auf den ausschließlich über zugeordenete *Methoden* zugegriffen werden kann. Zugriffe können sowohl lesend als auch schreibend sein.

Methode Programmiersprachliche Einheit, die die Struktur der Zugriffsmechanismen zu den internen Zuständen der Objekte einer Klasse festlegt: enthält eventuell mehrere Variablendeklarationen sowie eine Aktivitätsträgerbeschreibung.

Auftrag Laufzeitgegenstand, der durch Methodenaufruf aus den zugeordneten Methodenbeschreibungen eines Objekts dynamisch generiert wird. Aufrufe sind uniform asynchron — Aufrufer werden nicht automatisch blockiert — und, im Gegensatz zu bloßem Nachrichtenaustausch, strukturierte Interaktionen darstellen. Es können zu jedem Zeitpunkt mehrere Aufträge in einem Objekt aktiv sein.

Variable Platzhalter für Objekt-Referenzen.

Syntax und informelle Semantik Die syntaktischen Kategorien von `CLOWN` werden durch die nachfolgende Grammatik präsentiert. Gegeben seien Klassennamen $C \in \mathcal{C}$, (über Klassennamen) typisierte Methodennamen $M \in \mathcal{M}$ und ebenso typisierte Variablen $X \in \mathcal{X}$. Terminalsymbole werden mittels SMALL CAPS kenntlich gemacht.

$$\mathit{Prog} \quad ::= \quad \mathit{Cdec}_1, \ldots, \mathit{Cdec}_n \text{ ROOT } C$$

Ein `CLOWN`-Programm besteht aus einer Folge von Klassendefinitionen und einer ausgewählten Wurzelklasse C. Beim Programmstart wird ein Objekt der Wurzelklasse erzeugt. Vom Initialisierungscode dieses Objekts wird das eigentliche Programm angestoßen.

$$\begin{aligned}\mathit{Cdec} \quad ::= \quad & \text{CLASS } C \text{ ;} \\ & [\text{VAR } X_{1_1}, \ldots, X_{1_s} : C_1; \ \ldots \ ; \ X_{n_1}, \ldots, X_{n_t} : C_n;] \\ & \mathit{Mdec}_1 \ \ldots \ \mathit{Mdec}_m \\ & \text{BODY } S \text{ END;}\end{aligned}$$

Cdec ist die Definition einer Klasse mit Namen C. Die VAR-Deklaration von typisierten Objektvariablen kann entfallen. Es muß mindestens eine Methodendefinition (*Mdec*) vorhanden sein. S bezeichnet die Anweisungen zur Initialisierung eines frisch erzeugten Objektes.

$$\begin{aligned}\mathit{Mdec} \quad ::= \quad & \text{METHOD } M \ ([X_{1_1}, \ldots, X_{1_s} : C_1; \ \ldots \ ; \ X_{n_1}, \ldots, X_{n_t} : C_n;]) : C; \\ & [\text{VAR } X_{1_1}, \ldots, X_{1_s} : C_1; \ \ldots \ ; \ X_{m_1}, \ldots, X_{m_t} : C_m;] \\ & \text{BODY } S \text{ END}\end{aligned}$$

Mdec definiert eine Methode mit Namen M. Die Parameterliste kann leer sein. Auch auf Auftragsvariablen kann verzichtet werden. S sind die Anweisungen des Methodenrumpfes.

$$\begin{aligned}S \quad ::= \quad & X := E; \quad | \quad \text{RETURN } E; \quad | \quad S_1 \ S_2 \\ | \quad & \text{IF } E \text{ THEN } S_1 \ [\text{ELSE } S_2] \text{ END;} \quad | \quad \text{WHILE } E \text{ DO } S \text{ END;}\end{aligned}$$

Zuweisungen $x := E$ dürfen beliebige Werte-Ausdrücke enthalten. Andere Anweisungsfolgen S unterliegen leichten Einschränkungen bezüglich des Typs des vorkommenden Werteausdrucks E: Im RETURN dürfen alle Ausdrücke außer einem Methodenaufruf für E eingesetzt werden. Der Ausdruck in IF und WHILE muß sich zu TRUE oder FALSE auswerten lassen.

$$\begin{aligned}E \quad ::= \quad & X \quad | \quad \text{SELF} \quad | \quad \text{NEW}(C) \\ | \quad & \text{TRUE} \quad | \quad \text{FALSE} \quad | \quad \ldots, \tilde{}3, \tilde{}2, \tilde{}1{,}0{,}1{,}2{,}3, \ldots \\ | \quad & E \ ! \ M([X_1, \ldots, X_n])\end{aligned}$$

Werteausdrücke E in `CLOWN` haben gemeinsam, daß ihre Berechnung zu einem Wert, im Allgemeinen eine Objektreferenz, führt. Die Auswertung von Variablen liefert die Referenz, die sie enthalten. SELF liefert die Referenz des Objektes zurück, in dem der Ausdruck ausgewertet wird. NEW(C) erzeugt ein neues Objekt der Klasse C und liefert eine eindeutige

Referenz darauf zurück. TRUE und FALSE werten sich zu Objektreferenzen der vordefinierten Klasse BOOLEAN aus. BOOLEAN besitzt drei Methoden `b_not()`, `b_and()` und `b_or()`. Zahlen liefern Referenzen auf ihr Zahlobjekt der vordefinierten Klasse INTEGER zurück. INTEGER besitzt die Methoden `succ()`, `pred()` und `is_zero()`.

Ebenfalls zur Kategorie der Werteausdrücke gehören (asynchrone) Methodenaufrufe. Als (optionale) Parameter dürfen lediglich Variablennamen eingesetzt werden. Die Semantik von Methodenaufrufen folge dem Schema aus Abbildung 1. Informellen Beschreibungen ist stets inhärent, daß gelegentlich nicht vorhergesehene Fälle in der Abarbeitung von Programmen auftreten, bei denen mehrere sinnvolle Möglichkeiten der Semantikgebung existieren. Wir unterscheiden bezüglich der Seite des Aufrufers (*Client*) und des Aufgerufenen (*Server*):

Client Derselben Variable werden nacheinander verschiedene Werte mittels asynchroner Aufrufe zugewiesen. Die Reihenfolge ihrer Rückantworten muß aufgrund unterschiedlicher Rechen- und Verzögerungszeiten nicht notwendigerweise mit der Aufrufreihenfolge übereinstimmen. Welche Rückantwort soll erfolgreich ihren Wert in der Variablen hinterlegen dürfen und die Variable aus ihrem Warte-Zustand befreien? Was ist das intendierte Verhalten aus der Sicht der sequentiellen Semantik?

Server Soll es möglich sein, daß Methoden nach Ausführung einer RETURN-Anweisung noch weitere Anweisungen, sogenannte *post-actions*, bearbeiten können? Sollen solche Anweisungen auch dann ausgeführt werden (und evtl. weitere Seiteneffekte verursachen), wenn die Rückantwort, wie oben angedeutet, mittlerweile obsolet geworden ist?

Eine formale Semantik soll hier wie auch im allgemeinen Programmkontext zum einen helfen, die verschiedenen Möglichkeiten der Semantikgebung präzise gegeneinander abzuwägen, ist dagegen auch eine conditio sine qua non für die Verifikation von Programmen.

3 Formale Semantik

Zur Konzentration auf die Aspekte der Methodenaufrufsemantik basiert unsere Beispielsprache `CLOWN` auf der parallelen objektorientierten Sprache POOL. Die Semantik von POOL wurde mittels verschiedener semantischer Techniken formalisiert: durch Kodierung in Prozeßkalküle [Wal92] und direkt durch eine Strukturelle Operationelle Semantik (SOS) [Ame92].

Wir haben beide Techniken auch für `CLOWN` alternierend eingesetzt. Die Kodierung von `CLOWN` in den π-Kalkül [MPW89] bzw. dessen Implementierung PICT [PT94] gestattete uns, kleinere Programme schon frühzeitig zu testen. Details finden sich in [Boh94]. Mit der Formalisierung durch Abbildung in PICT erhält man ein abstraktes Ausführungsmodell, das allerdings nicht direkt über die objektorientierten Entitäten spricht, sondern lediglich indirekt, vermittels ihrer Kodierungen als Prozeßsysteme. Eine SOS dagegen spezifiziert die Semantik direkt. Sie legt Übergänge zwischen den Zuständen eines Transitionssystems fest. Letztere sind häufig die Programmterme selbst. *Strukturell* ist eine solche Semantik dann, wenn sich die Definition der Übergangsrelation streng am syntaktischen Aufbau der Programme orientiert. Die Übergangsrelation definiert man inferentiell als diejenige, die eine Menge von strukturellen Axiomen und Regeln erfüllt. Damit ist klar, daß nur solche Übergänge möglich sind, die mittels der Anwendung der Regeln hergeleitet werden können.

3.1 Zustände

Wir erweitern die Syntax von Anweisungen und Werteausdrücken, um in der Semantik auftretende Zwischenzustände uniform pseudosyntaktisch behandeln zu können:

$$\begin{array}{lcl} S & ::= & \ldots \mid (\alpha, E) \mid \epsilon; \\ E & ::= & \ldots \mid \omega \end{array}$$

Die Hilfsanweisung (α, E) deutet an, daß das Ergebnis des Ausdrucks E als Ergebnis des Auftrags α gedeutet werden soll. Es bezeichnet somit einen Zwischenzustand einer gerade in Bearbeitung befindlichen Wertzuweisung. Anweisungsfolgen können abgearbeitet werden und dann leer, bezeichnet mit ϵ, werden. Werteausdrücke für Programmzustände können während des Programmverlaufs zu Objektreferenzen ω werden.

Wir unterscheiden drei verschiedene Arten von Gegenständen, die Teil eines Zustands sind: syntaktische Deklarationen (Ausdrücke, Anweisungen, Methoden, Klassen), semantische Laufzeitgegenstände (Variablen, Aufträge, Objekte), und schließlich semantische Referenzen auf Objekte und Aufträge. Seien dazu neben $\mathcal{C}, \mathcal{M}, \mathcal{X}$ wie oben gegeben: $\mathcal{O} \cup \mathcal{A}$ mit $\mathcal{O} \cap \mathcal{A} = \{\uparrow\}$ die entsprechenden Mengen von Objekt- und Auftragsreferenzen, wobei $\uparrow$ die undefinierte Referenz bezeichnet; die Gesamtmenge der Referenzen enthalte dann $\mathcal{R} = \mathcal{N} \uplus \mathcal{B} \uplus (\mathcal{O} \cup \mathcal{A})$ auch Booleans und Zahlen.

Definition 3.1 (Ausdruck, Anweisung) *S und E seien die Mengen der Anweisungen S und Ausdrücke E aus der erweiterten Grammatik der Programmiersprachsyntax.* □

Definition 3.2 (Methode, Klasse) $\mathsf{M} \stackrel{def}{=} \mathcal{M} \times (2^{\mathcal{X}} \times \mathsf{S})$, $\mathsf{C} \stackrel{def}{=} \mathcal{C} \times (2^{\mathcal{X}} \times \mathsf{S}) \times 2^{\mathsf{M}}$.
Klassendeklarationen $\langle C, \langle \tilde{X}_C, S_C \rangle, \mathsf{M}_C \rangle \in \mathsf{C}$ und die dazugehörigen Methodendeklarationen $\langle M, \langle \tilde{X}_M, S_M \rangle \rangle \in \mathsf{M}$ sind Tupel, die direkt anhand der Syntax bestimmt werden. □

Definition 3.3 (Variable) $\mathsf{V} \stackrel{def}{=} \mathcal{X} \times \mathcal{O} \times \mathcal{A}$, $\mathsf{V}^+ \stackrel{def}{=} \mathcal{X} \times \mathcal{O} \times \mathcal{A} \times 2^{\mathcal{A}}$.
Im Zustand $\langle X, \gamma_X, \alpha_X \rangle \in \mathsf{V}$ einer Variablen *bezeichnet X den Namen, γ_X den gegenwärtig gültigen Inhalt und α_X die aktuell zugeordnete Auftragsreferenz. In $\langle X, \gamma_X, \alpha_X, \mathrm{ref}_X \rangle \in \mathsf{V}^+$ enthält ref_X die Referenzen der obsolet gewordenen Aufträge. Es gelte stets $\alpha_X \notin \mathrm{ref}_X$.* □

Die Komponente α_X entscheidet darüber, ob eine Variable im Zustand WAIT ($\alpha_X \neq \uparrow$) ist, oder im Zustand READY ($\alpha_X = \uparrow$). Der Initialzustand $\mathsf{V}_{\mathrm{init}}(X)$ ist gegeben durch $\langle X, \uparrow, \uparrow, \emptyset \rangle$.

Definition 3.4 (Auftrag, Objekt) $\mathsf{A} \stackrel{def}{=} \mathcal{A} \times (2^{\mathsf{V}} \times \mathsf{S})$, $\mathsf{O} \stackrel{def}{=} \mathcal{O} \times (2^{\mathsf{V}} \times \mathsf{S}) \times 2^{\mathcal{A}}$.
$\xi : (\mathcal{A} \cup \mathcal{O}) \to 2^{\mathsf{V}} \times \mathsf{S}$ *bildet Objekt- und Auftragsreferenzen auf die* Befehlszähler *ihrer zugeordneten Laufzeitgegenstände ab. Der Zustand $\langle \alpha, \xi(\alpha) \rangle \in \mathsf{A}$ eines* Auftrags *sowie der Zustand $\langle \omega, \xi(\omega), \mathcal{A}_\omega \rangle \in \mathsf{O}$ eines* Objekts *beruhen lediglich auf ihrer Referenz und dem Befehlszähler. Objekte beinhalten zusätzlich die zugehörigen momentan aktiven Aufträge.* □

Die S-Komponente des Befehlszählers gibt in syntaxgerichteter Weise an, welche Anweisungen jeweils noch zur Ausführung des Auftrags oder des Objekts anstehen.

Definition 3.5 (Konfiguration) *Eine* CLOWN-Konfiguration *zu einem Programm P ist ein Tripel $\langle \Delta_P, \mathcal{R}, \Gamma \rangle \in 2^{\mathsf{C}} \times \mathcal{R} \times 2^{\mathsf{O}}$, wobei $\Gamma \neq \emptyset$ und $\mathcal{R}(\Gamma) \in \mathcal{R}$.* □

Methoden- und Variablendeklarationen sind Teil der Klassendeklarationen, Aufträge und Variablen sind Teil der Objekte. Die Nebenbedingung für Referenzen $\mathcal{R}$ stellt sicher, daß alle in den Γ-Objekten vorkommenden Referenzen in der Menge $\mathcal{R}$ protokolliert wurden.

3.2 Übergänge

Mit der nachfolgenden mehrstufigen Semantik geben wir ein hierarchisches Definitionsgerüst an, in welchem Übergänge von globalen Konfigurationen aus den lokalen Übergängen von Objekten, deren Aufträgen, und schließlich deren einzelnen Abarbeitungsschritten und Variablen-Übergängen abgeleitet werden.

Notation $\omega, \gamma \in \mathcal{O}$ bezeichnen immer Objektvariablen. $\alpha, \beta \in \mathcal{A}$ stehen stets für Auftragsvariablen. $m + e$ für beliebige Mengen m und beliebige Elemente e sei gleichbedeutend mit der disjunkten Vereinigung $m \uplus \{e\}$. Wir verwenden die Notation $\tilde{X}$ polymorph für Tupel und Mengen. $\mathcal{X}, \mathsf{V}, \mathsf{S}$ werden gelegentlich als Projektionsfunktionen auf Tupeln verwendet. $\mathsf{V}(X)$ bezeichne einen beliebigen erlaubten Zustand $\langle\ X, \gamma_X, \alpha_X, \mathrm{ref}_X\ \rangle$ der Variablen X. $\sigma \subseteq \mathsf{V}$ sei eine Menge von Variablen. Wir führen weiter verschiedene über Variablennamen parametrisierte Labels ein: wait_X, inp_X, out_X, block_X, write_X, read_X.

3.2.1 Variablen

Die einfache Representation von Variablen ignoriert ungültig gewordene Auftragsreferenzen.

$$
\begin{array}{ll}
\mathit{VAR\text{-}W} & \langle\ X, \gamma, \alpha\ \rangle \xrightarrow{\mathrm{wait}_X(\beta)} \langle\ X, \gamma, \beta\ \rangle \\
\mathit{VAR\text{-}R} & \langle\ X, \gamma, \alpha\ \rangle \xrightarrow{\mathrm{inp}_X(\omega,\alpha)} \langle\ X, \omega, \uparrow\ \rangle \\
\mathit{VAR\text{-}C} & \langle\ X, \gamma, \uparrow\ \rangle \xrightarrow{\mathrm{out}_X(\gamma)} \langle\ X, \gamma, \uparrow\ \rangle
\end{array}
$$

Regel *VAR-W* besagt, daß die momentan gültige Auftragsreferenz jederzeit durch das Anstoßen eines aktuelleren Auftrags überschrieben werden kann. Dies gilt auch für den Fall $\alpha = \uparrow$, welcher das erste Blockieren der Variable repräsentiert. Regel *VAR-R* verdeutlicht, daß nur der Auftrag mit der passenden Referenz α mit der Variable interagieren kann. Regel *VAR-C* besagt, daß eine Variable nur dann ihren Wert preisgibt, wenn er gültig ist, d.h. wenn momentan kein Auftrag angestoßen wurde, dessen Wert die Variable annehmen soll.

Die erweiterte Modellierung von Variablen führt in der Komponente ref explizit Buch darüber, welche Aufträge zunächst angestoßen wurden, um ihr Ergebnis in der Variablen X abzulegen, deren Effekt aber zwischenzeitlich überschrieben wurde.

$$
\begin{array}{ll}
\mathit{VAR^+\text{-}W} & \langle\ X, \gamma, \alpha, \mathrm{ref}\ \rangle \xrightarrow{\mathrm{wait}_X(\beta)} \langle\ X, \gamma, \beta, \mathrm{ref} + \alpha\ \rangle \\
\mathit{VAR^+\text{-}R1} & \langle\ X, \gamma, \alpha, \mathrm{ref}\ \rangle \xrightarrow{\mathrm{inp}_X(\omega,\alpha)} \langle\ X, \omega, \uparrow, \mathrm{ref}\ \rangle \\
\mathit{VAR^+\text{-}R2} & \langle\ X, \gamma, \alpha, \mathrm{ref} + \beta\ \rangle \xrightarrow{\mathrm{inp}_X(\omega,\beta)} \langle\ X, \gamma, \alpha, \mathrm{ref}\ \rangle \\
\mathit{VAR^+\text{-}C} & \langle\ X, \gamma, \uparrow, \mathrm{ref}\ \rangle \xrightarrow{\mathrm{out}_X(\gamma)} \langle\ X, \gamma, \uparrow, \mathrm{ref}\ \rangle
\end{array}
$$

Regel *VAR$^+$-R2* spezifiziert das Verhalten, nach dem auch solche mittlerweile ungültigen Aufträge ihren RETURN-Wert abliefern können, um dann eventuelle *post-actions* auszuführen (siehe Abschnitt *Gebundene Anweisungen* in 3.2.2). Allerdings wird ihr RETURN-Wert nicht weiter verwendet. Lediglich die nunmehr terminierte Auftragsreferenz kann aus der ref-Komponente gelöscht werden.

3.2.2 Auswertung

Ausdrücke Die folgenden kontextabhängigen Regeln beschreiben, daß sich programmiersprachliche Konstanten zu den korrespondierenden Referenzen auswerten lassen. Der jeweilige Auswertungskontext $\langle \omega, \alpha, \mathcal{R} \rangle$ besteht aus der momentanen Objekt- und Auftragsreferenz, sowie der globalen Menge der bisher bereits vergebenen Referenzen. Falls $\alpha = \uparrow$, dann ist die Auswertung Teil des Initialisierungscodes von ω. Die Auswertung von einfachen Ausdrücken werden durch Axiome beschrieben:

$$
\begin{array}{lll}
\mathit{BOOL} & \langle\, \omega, \alpha, \mathcal{R} \,\rangle & \vdash \quad \texttt{bool} \longmapsto b \\
\mathit{NAT} & \langle\, \omega, \alpha, \mathcal{R} \,\rangle & \vdash \quad \texttt{i} \longmapsto i \\
\mathit{SELF} & \langle\, \omega, \alpha, \mathcal{R} \,\rangle & \vdash \quad \texttt{self} \longmapsto \omega \\
\mathit{VAR} & \langle\, \omega, \alpha, \mathcal{R} \,\rangle & \vdash \quad X \overset{\mathrm{read}_X(\gamma)}{\longmapsto} \gamma \\
\mathit{NEW} & \langle\, \omega, \alpha, \mathcal{R} \,\rangle & \vdash \quad \mathrm{NEW}(C) \overset{\mathrm{new}_C(\gamma)}{\longmapsto} \gamma
\end{array}
$$

Wir verzichten hier auf die Darstellung der weiteren Regeln für die Auswertung von Zahl- und bool'schen Operationen. Sie können in [Boh94] nachgelesen werden.

Zusammengesetzte Ausdrücke folgen dem strukturellen Prinzip der SOS-Technik. Die nachfolgenden Regeln bestimmen die Auswertereihenfolge *von-links-nach-rechts*, falls mehrere Komponenten gegenwärtig zur Auswertung anstehen.

$$
\mathit{METH1} \quad \frac{\langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; E \overset{l}{\longmapsto} E'}{\langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; E \,!\, M(X_1, \ldots, X_n) \overset{l}{\longmapsto} E' \,!\, M(X_1, \ldots, X_n)}
$$

$$
\mathit{METH2} \quad \frac{\langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; X_i \overset{l}{\longmapsto} \gamma_i}{\langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; \gamma \,!\, M(\ldots, \gamma_{i-1}, X_i, X_{i+1}, \ldots) \overset{l}{\longmapsto} \gamma \,!\, M(\ldots, \gamma_{i-1}, \gamma, X_{i+1}, \ldots)}
$$

Anweisungen Zuweisungen werden zunächst blockiert und erzeugen dabei einen Auftrag, in Regel *ZUW1* dargestellt durch das oben eingeführte syntaktische Hilfskonstrukt, für den eine Auftragsreferenz β generiert wird. Diese bildet das eindeutige Bindeglied zwischen asynchronem Auftrag und der Zielvariablen X. Die Regel *CALL* auf Seite 9 garantiert die Eindeutigkeit von Auftragsreferenzen in Programmabläufen.

$$
\begin{array}{lll}
\mathit{ZUW1} & \langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; X := E \xrightarrow{\mathrm{block}_X(\beta)} (\beta, E) & \textit{für} \quad \beta \notin \mathcal{R} \\
\mathit{ZUW2} & \langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; (\beta, \gamma \,!\, M(\gamma_1, \ldots, \gamma_n)) \xrightarrow{\mathrm{call}(\beta, \gamma, M, \tilde{\gamma})} \epsilon & \\
\mathit{ZUW3} & \langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; (\beta, \gamma) \xrightarrow{\mathrm{write}_X(\beta, \gamma)} \epsilon & \\
\mathit{ZUW4} & \langle\, \omega, \beta, \mathcal{R} \,\rangle \;\vdash\; (\mathrm{RETURN}\ \gamma) \xrightarrow{\mathrm{write}_X(\beta, \gamma)} \epsilon &
\end{array}
$$

Axiom *ZUW2* besagt, daß ein Methodenaufruf abgesetzt werden kann, sobald das Server-Objekt sowie die Parameter zu Referenzen ausgewertet worden sind. Man beachte den Unterschied im Kontext der Regeln *ZUW3* und *ZUW4*: Ist die rechte Seite der Zuweisung selbst eine Variable, wird der Wert, sofern möglich, sofort vom Aufrufer α bzw. ω selbst gelesen; ist die rechte Seite der Zuweisung dagegen ein Methodenaufruf, so wird die eigentliche Zuweisung vom aufgerufenen Auftrag β vollzogen.

Wir verzichten auf eine Darstellung der Regeln für IF und WHILE, da sie lediglich kompositionelles Weitergeben von Übergängen beinhalten. Sequenzen von Anweisungen folgen der intendierten sequentiellen Abarbeitungsreihenfolge *von-links-nach-rechts*.

$$\textit{SEQ1} \qquad \langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; \epsilon\,;S \xrightarrow{\;l\;} S$$

$$\textit{SEQ2} \qquad \frac{\langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; S_1 \xrightarrow{\;l\;} S_1'}{\langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; S_1\,;S_2 \xrightarrow{\;l\;} S_1'\,;S_2}$$

Gebundene Anweisungen Die freien Anweisungen werden nun in den Kontext transferiert, in dem die jeweils gültigen Variablen explizit vorhanden sind und bei Bedarf gelesen werden können. Die Regeln realisieren die verborgene Synchronisation des *fork-and-join*.

$$\textit{BLOCK} \qquad \frac{\langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; S \xrightarrow{\mathrm{block}_X(\beta)} S' \qquad \mathsf{V}(X) \xrightarrow{\mathrm{wait}_X(\beta)} \mathsf{V}'(X)}{\langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; \langle \sigma + \mathsf{V}(X)\,;S\rangle \longrightarrow \langle \sigma + \mathsf{V}'(X)\,;S'\rangle}$$

$$\textit{WRITE} \qquad \frac{\langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; S \xrightarrow{\mathrm{write}_X(\gamma,\beta)} S' \qquad \mathsf{V}(X) \xrightarrow{\mathrm{inp}_X(\gamma,\beta)} \mathsf{V}'(X)}{\langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; \langle \sigma + \mathsf{V}(X)\,;S\rangle \longrightarrow \langle \sigma + \mathsf{V}'(X)\,;S'\rangle}$$

$$\textit{READ} \qquad \frac{\langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; S \xrightarrow{\mathrm{read}_X(\gamma)} S' \qquad \mathsf{V}(X) \xrightarrow{\mathrm{out}_X(\gamma)} \mathsf{V}(X)}{\langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; \langle \sigma + \mathsf{V}(X)\,;S\rangle \longrightarrow \langle \sigma + \mathsf{V}(X)\,;S'\rangle}$$

Insbesondere sorgt die Regel *WRITE* bei der erweiterten Modellierung von Variablen dafür, daß bei Entgegennahme des RETURN-Wertes durch die Variable, der Server-Auftrag im Sinne von *post-actions* weiterarbeiten kann, unabhängig davon, ob er obsolet geworden war, oder nicht. In der einfachen Version würde die *WRITE*-Regel nicht zur Anwendung kommen können, weil die Variable keinen passenden inp_X-Übergang anbieten würde.

Da eine Variable aber nicht nur von Aufträgen innerhalb des eigenen Objekts beschrieben werden kann, muß man dafür sorgen, daß die Aufnahmebereitschaft dieser Variablen auch auf der Ebene der globalen Konfiguration bekannt wird. Als ersten Schritt hierzu liften wir diesen Übergang auf die Ebene der gebundenen Anweisungen.

$$\textit{REMOTE} \qquad \frac{\mathsf{V}(X) \xrightarrow{\mathrm{inp}_X(\gamma,\beta)} \mathsf{V}'(X)}{\langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; \langle \sigma + \mathsf{V}(X)\,;S\rangle \longrightarrow \langle \sigma + \mathsf{V}'(X)\,;S\rangle}$$

3.2.3 Aufträge und Objekte

Wenn man das lokale Verhalten individueller Aufträge oder die Abarbeitung des Initialisierungscodes eines Objektes nachvollziehen möchte, helfen die folgenden Regeln. Sie stellen sicher, daß der entsprechende Code über die Semantik von gebundenen Anweisungen abgearbeitet werden muß.

$$\textit{AUFTRAG} \qquad \frac{\langle\, \omega, \alpha, \mathcal{R} \,\rangle \;\vdash\; \langle \sigma\,;S\rangle \longrightarrow \langle \sigma'\,;S'\rangle}{\mathcal{R} \;\vdash\; \langle \alpha, \langle \sigma; S\rangle\rangle \longrightarrow \langle \alpha, \langle \sigma'; S'\rangle\rangle}$$

$$\textit{OBJEKT} \qquad \frac{\langle\, \omega, \uparrow, \mathcal{R} \,\rangle \;\vdash\; \langle \sigma; S\rangle \longrightarrow \langle \sigma'; S'\rangle}{\mathcal{R} \;\vdash\; \langle \omega, \langle \sigma; S\rangle, \mathcal{A}_\omega\rangle \longrightarrow \langle \omega, \langle \sigma'; S'\rangle, \mathcal{A}_\omega\rangle}$$

Eventuelle Überschneidungen bezüglich der verwendeten Variablennamen in Klassen und Methoden sollen sich nicht störend auswirken, sondern gemäß der Blockstruktur gehandhabt werden. In [Boh94] findet sich zu diesem Zweck noch die lediglich technische Regel *SCOPING*.

3.2.4 Programme

Die Interaktionen zwischen verschiedenen Objekten (und deren Aufträgen) bestimmen die Übergänge der globalen Programmkonfigurationen. Alle *lokalen* Übergänge individueller Objekte sind auch im Gesamtsystem beobachtbar. Sei für die folgenden Regeln $\mathsf{O}(\omega_i) = \langle \omega_i, \xi(\omega_i), \mathcal{A}(\omega_i) \rangle$ und $\mathsf{O}'(\omega_i) = \langle \omega_i, \xi'(\omega_i), \mathcal{A}'(\omega_i) \rangle$.

$$\textit{LOKAL} \quad \frac{\mathcal{R} \vdash \mathsf{O}(\omega) \longrightarrow \mathsf{O}'(\omega)}{\langle\, \Delta, \mathcal{R}, \Gamma + \mathsf{O}(\omega) \,\rangle \longrightarrow \langle\, \Delta, \mathcal{R}, \Gamma + \mathsf{O}'(\omega) \,\rangle}$$

$$\textit{WRITE-G} \quad \frac{\mathcal{R} \vdash \mathsf{O}(\omega_1) \xrightarrow{\mathrm{write}_X(\gamma,\beta)} \mathsf{O}(\omega_1') \qquad \mathcal{R} \vdash \mathsf{O}(\omega_2) \xrightarrow{\mathrm{inp}_X(\gamma,\beta)} \mathsf{O}(\omega_2')}{\langle\, \Delta, \mathcal{R}, \Gamma \uplus \{\mathsf{O}(\omega_1), \mathsf{O}(\omega_2)\} \,\rangle \longrightarrow \langle\, \Delta, \mathcal{R}, \Gamma \uplus \{\mathsf{O}(\omega_1'), \mathsf{O}(\omega_2')\} \,\rangle}$$

Die Regel *WRITE-G* schließt ab, was mit der Regel *REMOTE* vorbereitet wurde: Der Auftrag mit der Referenz β im Objekt an der Referenz ω_1 soll der Variablen X im Objekt an der Referenz ω_2 den Wert γ zuweisen. Hierzu mußte das inp_X-label nach außen propagiert werden.

Die beiden letzten Regeln formulieren die Bedingungen für das generieren von frischen Referenzen in Objekten (NEW) und Aufträgen (CALL). In NEW wird ein neues Objekt eingetragen. In CALL wird ein Auftrag angestoßen und in seine Referenz in das Auftragsbuch seines Serverobjekts eingetragen.

$$\textit{NEW} \quad \frac{\mathcal{R} \vdash \mathsf{O}(\omega) \xrightarrow{\mathrm{new}_C(\gamma)} \mathsf{O}'(\omega)}{\langle\, \Delta, \mathcal{R}, \Gamma + \mathsf{O}(\omega) \,\rangle \longrightarrow \langle\, \Delta, \mathcal{R} + \gamma, \Gamma \uplus \{\mathsf{O}'(\omega), \mathsf{O}(\gamma)\} \,\rangle}$$

wobei C *in* Δ*, so daß:* $\quad \mathsf{O}(\gamma) \stackrel{\mathrm{def}}{=} \langle\, \gamma, \langle \mathsf{V}_{\mathrm{init}}(\mathcal{X}(C)), \mathsf{S}_{\mathrm{init}}(C) \rangle, \emptyset \,\rangle$

$$\textit{CALL} \quad \frac{\mathcal{R} \vdash \mathsf{O}(\omega) \xrightarrow{\mathrm{call}(\beta,\gamma,M,\tilde{\gamma})} \mathsf{O}'(\omega)}{\langle\, \Delta, \mathcal{R}, \Gamma \uplus \{\mathsf{O}(\omega), \mathsf{O}(\gamma)\} \,\rangle \longrightarrow \langle\, \Delta, \mathcal{R} + \beta, \Gamma \uplus \{\mathsf{O}'(\omega), \langle \gamma, \xi(\gamma), \mathcal{A}_\gamma + \mathsf{A}'(\beta) \rangle\} \,\rangle}$$

falls $\mathsf{S}(\omega) = \epsilon$*; und* $\gamma : C$

und C, M *in* Δ*, so daß:* $\quad \mathsf{A}'(\beta) \stackrel{\mathrm{def}}{=} \langle\, \beta, \langle \mathsf{V}_{\mathrm{init}}(\mathcal{X}(M)), \mathsf{S}_{\mathrm{init}}(M) \rangle \,\rangle$

SOS Die Semantik eines Programmes besteht schließlich in der Abbildung auf eine ihm entsprechende initiale `CLOWN`-Konfiguration, d.h. einem Zustand im Transitionssystem, dessen Übergänge durch die oben angegebenen Regeln spezifiziert sind.

$$[\![\, P \,]\!] \stackrel{\mathrm{def}}{=} \langle\, \Delta_P, \mathcal{R}_{\mathrm{init}}, \{\mathsf{O}(\omega)\} \,\rangle$$

Der Startzustand eines Programms P besteht aus der Menge Δ seiner Klassendefinitionen, einschließlich der Variablen- und Methodendeklarationen, der initialen Referenzenmenge $\mathcal{R}_{\mathrm{init}} = (\mathcal{N} \uplus \mathcal{B}) + \omega$, und dem zu initialisierenden Wurzelobjekt $\mathsf{O}(\omega)$, welches analog zu $\mathsf{O}(\gamma)$ in Regel *NEW* definiert wird.

4 Schlußfolgerung

Die Formalisierung des Methodenaufrufprotokolls lieferte ein eindeutiges Referenzmodell, an dem verschiedene Unklarheiten des Sprachentwurfs bereinigt werden konnten. Insbesondere konnte formal modelliert werden, was mit obsoleten Aufträgen passieren soll, denen ein anderer Auftrag bei der Rückgabe eines Ergebnisses zuvor gekommen ist: Explizites Protokollieren der Auftragsreferenzen verhilft hier dem Compiler-Entwickler dazu, die zugehörige *Garbage-Collection* gezielt zu gestalten. Der Umgang mit verschiedenen semantischen Techniken, um die Eigenarten bestimmter Sprachkonzepte zu untersuchen, haben sich als sehr nützlich erwiesen, da sie jeweils unterschiedliche Aspekte betonen: die implizite Parallelität des Aufrufschemas erfuhr beispielsweise nur durch die Kodierung in einen unterliegenden Prozeßkalkül eine explizite Darstellung. Das Aufsammeln unberechtigter Auftragsreferenzen wiederum war nur in der SOS explizit. Beide Techniken wurden daher in unserem Fall abwechselnd und iteriert angewandt. Der Einsatz der Formalisierung des Methodenaufrufprotokolls zur Programmverifikation steht noch aus.

Literatur

[Ame92] Pierre America. Formal techniques for parallel object-oriented languages. In *Object-Based Concurrent Computing 1991, LNCS 612*, pages 119–140. 1992.

[Boh94] Henrik Bohnenkamp. *Concurrent Language with Objects and Wait-by-Necessity.* Studienarbeit, Universität Erlangen, 1994.

[Car90] Denis Caromel. Service, asynchrony and wait-by-necessity. *Journal of Object-Oriented Programming*, 2(4):12–22, November 1990.

[Jon93] Cliff Jones. Constraining Interference in an Object-Based Design Method. In *TAPSOFT '93, LNCS 668*, pages 136–150. 1993.

[MPW89] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I/II. Technical Report ECS-LFCS-89-85/86, University of Edinburgh, June 1989. Published in *Information and Computation* 100:1–77, 1992.

[PT94] Benjamin Pierce and David N. Turner. The PICT manual. 1994.

[RK94] Jan-Peter Richter and Jürgen Kleinöder. Fine-grained Parallel Computation in the PM/OM Object Model. Bericht TR-I4-04-94, Universität Erlangen, 1994.

[Wal92] David Walker. Objects in the π-calculus. Report RR 217, University of Warwick, April 1992. Published in *Information and Computation* 116(2):253–271, 1995.