Mitteilung Nr. 219

Eine Estelle-Erweiterung zur Steigerung der Nebenläufigkeit

Jan Bredereke Reinhard Gotzhein

FBI-HH-M-219/93

Februar 1993

Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Straße 30

W-2000 Hamburg 54

Eine Estelle-Erweiterung zur Steigerung der Nebenläufigkeit

Jan Bredereke Reinhard Gotzhein
Universität Hamburg, Vogt-Kölln-Str. 30, W-2000 Hamburg 54

E-Mail: `bredereke@informatik.uni-hamburg.de`

Zusammenfassung

Überlegungen zur parallelen Implementierung des ISO-Protokolls für verteilte Transaktionsverarbeitung haben ergeben, daß der in Estelle spezifizierbare Parallelitätsgrad für eine effiziente Implementierung auf einer Mehrprozessoranlage nicht ausreicht. Daher wird eine geringfügige, kompatible Spracherweiterung von Estelle vorgeschlagen, die eine deutliche Steigerung der Nebenläufigkeit gestattet. Die dazu erforderlichen Änderungen der formalen Estelle-Semantik werden beschrieben. Ferner wird die Implementierbarkeit der Erweiterung untersucht.

Abstract

Considerations concerning the parallel implementation of the ISO protocol for distributed transaction processing led to the conclusion that the degree of parallelism which can be specified in Estelle is not sufficient for an efficient implementation on a multiprocessor system. Therefore, we propose a minor, compatible language extension for Estelle which allows a significant increase in concurrency. The changes necessary in the formal Estelle semantics are described. Also, we examine how the extension can be implemented.

1 Einleitung

Die Nutzung der Übertragungskapazität von Hochgeschwindigkeitsnetzen für Endanwendungen erfordert eine deutlich höhere Geschwindigkeit bei der Bereitstellung von Daten in den Kommunikationssubsystemen. Eine Verbesserung läßt sich hier durch die Entwicklung neuer Kommunikationsarchitekturen und -protokolle erzielen. Zusätzliches Potential zur Steigerung der Verarbeitungsgeschwindigkeit liegt in der parallelen Implementierung der Protokolle.

Im Rahmen eines DFG-Projektes (Vo 543/1-1) untersuchen wir derzeit Parallelisierungsmöglichkeiten für das ISO-Protokoll zur verteilten Transaktionsverarbeitung (ISO-TP, [ISO92]). Innerhalb des OSI-Basisreferenzmodells ([ISO81]) wird die verteilte Transaktionsverarbeitung der Anwendungsschicht (Schicht 7) zugeordnet. Als Nutzer des von ISO-TP erbrachten Dienstes kommen die Anwendungsprozesse der Schicht 7 in Betracht, gegebenenfalls auch unter Zusammenarbeit von ISO-TP mit anderen Diensterbringern der Anwendungsschicht wie z.B. Remote Database Access (RDA).

Eine (verteilte) Transaktion läßt sich als partielle Ordnung von Operationen, welche eine Datenbasis von einem konsistenten Zustand in einen konsistenten Folgezustand überführt, verstehen. Dabei gilt, daß eine Transaktion entweder vollständig oder gar nicht ausgeführt wird. Die Wirkung tritt vom konzeptuellen Standpunkt aus also atomar ein. Verteilte Transaktionen findet man beispielsweise in verteilten Datenbanken vor. Hier ist es allerdings üblich, die Verteilung des Datenbestandes und damit der Transaktionen vor dem Anwender zu verbergen, was dadurch, daß der gesamte Datenbestand unter der Kontrolle eines einzigen (verteilten) Datenverwaltungssystems steht, recht gut realisierbar ist. Eine grundsätzlich andere Situation liegt dann vor, wenn mehrere unabhängige Datenverwaltungssysteme an der Ausführung einer verteilten Transaktion beteiligt sind. Das kann sogar so weit gehen, daß die beteiligten Systeme dynamisch erst während des Ablaufs der Transaktion bestimmt werden können. Um Transaktionen dieses Typs auszuführen, muß es möglich sein, die jeweils betroffenen Datenbestände (mindestens) für die Dauer einer Transaktion als logische Einheit aufzufassen. Das bedeutet insbesondere, daß die Transaktion neben der Konsistenz der einzelnen Datenbestände auch die Konsistenz der daraus (möglicherweise nur für die Dauer der Transaktion) gebildeten logischen Einheit erhalten muß. Der von ISO-TP erbrachte Dienst unterstützt den Anwendungsprozeß bei der Ausführung verteilter Transaktionen.

Aus der Norm über die Architektur der Anwendungsschicht ([ISO89a]) und dem ISO-TP-Modell ([ISO92]) ergibt sich die in Abbildung 1 gezeigte konzeptuelle Architektur von ISO-TP. Die als TPSUI (**TP S**ervice **U**ser **I**nstance) bezeichneten Komponenten sind die Nutzer des von den **TP-P**rotokoll**m**aschinen (TPPM) erbrachten TP-Dienstes, aktuelle Exemplare eines solchen Paares werden an einem physikalischen Rechner-Aufstellungsort (Site) dynamisch erzeugt und freigegeben. Jede TPPM enthält eine MACF-Komponente (**M**ultiple **A**ssociation **C**ontrol **F**unction) sowie eine Anzahl von SAO-Komponenten (**S**ingle **A**ssociation **O**bject). Das SAO beinhaltet eine Anzahl von Anwendungsbausteinen sowie eine Koordinierungskomponente SACF (**S**ingle **A**ssociation **C**ontrol **F**unction). Als mögliche Anwendungsbausteine sind in der Abbildung aufgeführt:

- Der in Abhängigkeit von der Anwendung des TP-Dienstes zu definierende Baustein UASE (**U**ser **A**pplication **S**ervice **E**lement)
- TPASE (**TP A**pplication **S**ervice **E**lement) zur Abwicklung des TP-Proto-

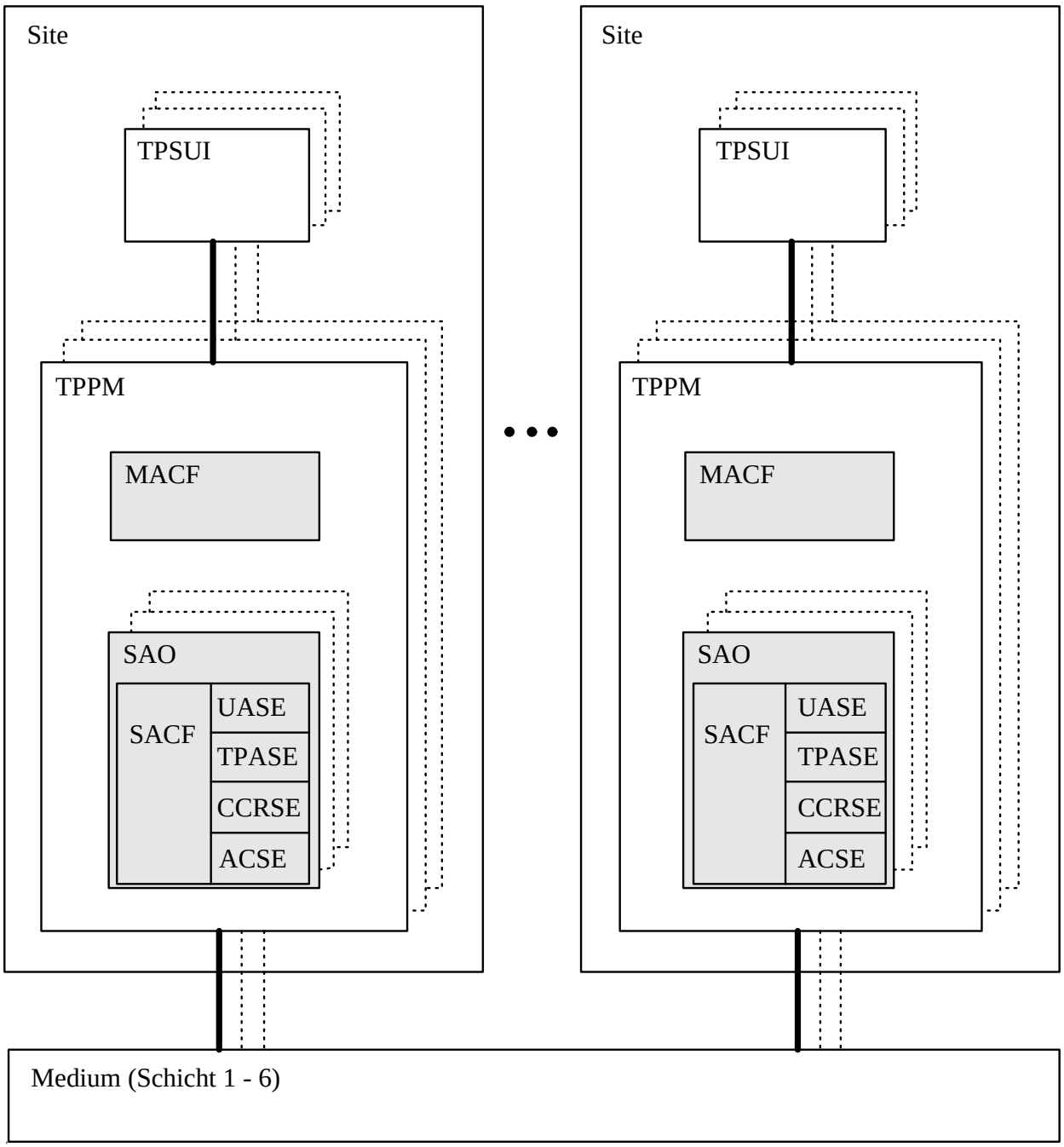

Abbildung 1: Konzeptuelle Architektur von ISO-TP

kolls auf einer Assoziation

- CCRSE (**C**ommitment, **C**oncurrency and **R**ecovery **S**ervice **E**lement) zur Abwicklung des 2-Phasen-Commit-Protokolls
- ACSE (**A**ssociation **C**ontrol **S**ervice **E**lement) zum Auf- und Abbau von Assoziationen

Ein SAO ist genau einer Assoziation zugeordnet. Die Koordinierung mehrerer Assoziationen obliegt der MACF-Komponente. Genauere Angaben über die Aufgaben der genannten Anwendungsbausteine und der weiteren Komponenten sind [Ker92] und der Normungsliteratur zu entnehmen.

Die in Abbildung 1 gezeigte konzeptuelle Architektur gestattet zunächst prinzipiell das parallele Arbeiten der TPSUI- und TPPM-Komponenten (sowohl an verschiedenen Orten (Sites) wie auch insbesondere am selben Ort), sowie des Mediums. Weitere Parallelität ist innerhalb einzelner Komponenten erzielbar, etwa bei den MACF- und SAO-Komponenten einer TPPM, aber auch innerhalb des (hier nicht strukturierten) Mediums. Dabei ist natürlich zu berücksichtigen, daß zwischen den Aktivitäten der verschiedenen Komponenten Abhängigkeiten bestehen, die zu einer Verringerung des Parallelitätsgrades führen können. Trotzdem ergibt sich aus der gezeigten Architektur ein hohes Parallelisierungspotential, welches den Ausgangspunkt für weitere Betrachtungen bildet.

Im Rahmen des DFG-Projektes ist geplant, ISO-TP zunächst in der formalen Beschreibungstechnik Estelle ([ISO89]) zu spezifizieren. Ein Estelle-Compiler soll anschließend aus dieser Spezifikation eine nebenläufig auf einem Mehrprozessorsystem ausführbare Implementierung generieren. Bei der Spezifikation von ISO-TP ist daher darauf zu achten, daß ein möglichst hoher Grad an potentieller Parallelität beschrieben wird. Das erfordert zum einen eine geeignete konzeptuelle Architektur von ISO-TP (siehe Abbildung 1) und zum anderen die Repräsentation dieser Architektur in der Spezifikation. Dabei soll das konzeptuell erzielte Parallelisierungspotential bei der Umsetzung in die Spezifikation nach Möglichkeit vollständig genutzt werden. Genau dies bereitet jedoch bei der Spezifikation von ISO-TP in Estelle Probleme. Im folgenden werden wir die grundsätzlichen Probleme am Beispiel von ISO-TP herausarbeiten. Anschließend stellen wir zwei Lösungsansätze vor und entwickeln eine Lösung vollständig. Die gewählte Lösung besteht in einer geringfügigen, kompatiblen Erweiterung des Estelle-Sprachumfangs. Die Auswirkungen auf die mit Estelle spezifizierbare Nebenläufigkeit sind indes nicht geringfügig, sondern erlauben wesentlich effizientere parallele Implementierungen.

2 Überblick über Estelle

Der folgende Überblick über Estelle soll einige der wichtigsten Aspekte der Sprache kurz und knapp vorstellen. Ausführliche Einführungen in Estelle geben

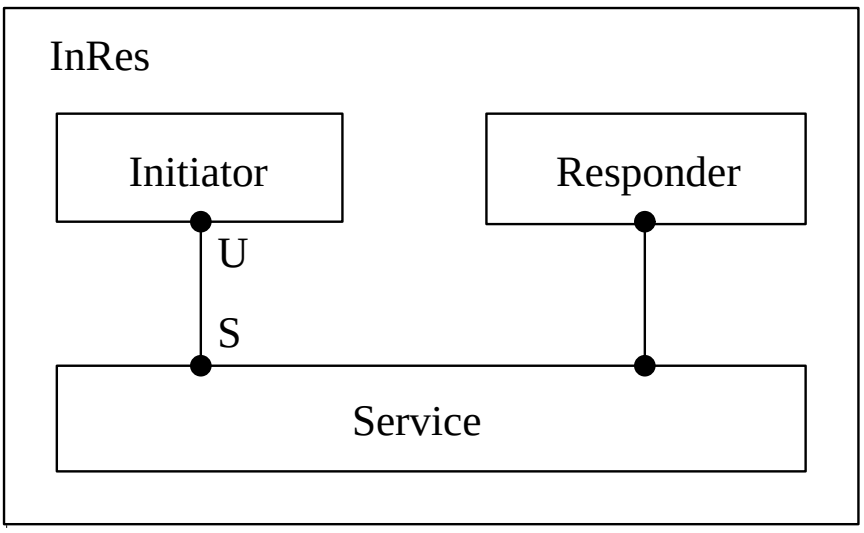

Abbildung 2: Eine Estelle-Systemarchitektur

[Hog89] sowie [DeBu89]. Für ein vertiefendes Studium eignet sich der Estelle-Standard ([ISO89]).

Eine Estelle-Spezifikation beschreibt ein hierarchisch aufgebautes System von nicht-deterministischen, sequentiellen Komponenten, den *Modulinstanzen*. Ein Beispiel für eine Systemarchitektur ist in Abbildung 2 dargestellt. Sie ergibt sich aus der Estelle-Spezifikation in Abbildung 3 durch Kreierung der Modulinstanzen *Initiator* und *Service* im Initialisierungsteil der Spezifikation *InRes* (Zeile 38–39)[1]. Modulinstanzen können über bidirektionale *Kanäle*, die jeweils zwei *Interaktionspunkte* miteinander verbinden, kommunizieren. Dies ist für die Interaktionspunkte `U` und `S` ebenfalls in Abbildung 2 dargestellt und in Zeile 40 (Abbildung 3) spezifiziert. Sowohl die Systemstruktur als auch die Verbindungsstruktur werden also in der Spezifikation festgelegt und können sich dynamisch verändern.

Daß es sich um *bi*direktionale Kanäle handelt, kommt in der Kanaldefinition (Zeile 3–5) zum Ausdruck: Für jeden Kanal sind zwei *Rollen* (z. B. User — Provider) zu definieren. Für jede dieser Rollen ist dann eine Menge von Nachrichtentypen angegeben, die über diesen Kanal gesendet werden dürfen.

Eine Modulinstanz ist gegeben durch einen Modulkopf und einen Modulrumpf. Der Modulkopf (Zeile 7–12) beschreibt das Format der externen Schnittstelle, die durch eine endliche Menge von Interaktionspunkten (Zeile 8–11) sowie eine (im Beispiel leere) Menge exportierter Variablen gegeben ist. Durch Zuordnung eines Kanals und einer Rolle zu jedem Interaktionspunkt wird festgelegt, mit welchen anderen Interaktionspunkten eine Verbindung (Zeile 40) erlaubt ist, und welche Nachrichten gesendet werden dürfen. Jedem Interaktionspunkt ist

[1] Die Estelle-Spezifikation in Abbildung 3 ist unvollständig, beispielsweise ist dort der Responder nicht berücksichtigt.

```
 1  SPECIFICATION InRes;
 2    ...
 3    CHANNEL User_access_point (User,Provider);
 4      BY User:     Connect_request;
 5      BY Provider: Connect_confirm;
 6    ...
 7    MODULE Initiator_type SYSTEMPROCESS;
 8      IP U: User_access_point (User) INDIVIDUAL QUEUE;
 9    END;
10    MODULE Service_type SYSTEMPROCESS;
11      IP  S: User_access_point (Provider) INDIVIDUAL QUEUE;
12    END;
13    ...
14    BODY Initiator_body FOR Initiator_type;
15      ...
16      VAR sequence_number: INTEGER;
17      ...
18      STATE disconnected, connection_requested, connected;
19      INITIALIZE TO disconnected
20        BEGIN sequece_number := 0 END;
21      TRANS
22        FROM disconnected TO connection_requested
23        BEGIN OUTPUT U.Connect_request END;
24      TRANS
25        FROM connection_requested TO connected
26        WHEN U.Connect_confirm
27        BEGIN END;
28      ...
29    END {Initiator-body};
30    BODY Service_body FOR Service_type; EXTERNAL;
31    ...
32    MODVAR
33      Initiator: Initiator_type;
34      Service:   Service_type;
35      ...
36    INITIALIZE
37      BEGIN
38        INIT Initiator WITH Initiator_body;
39        INIT Service   WITH Service_body;
40        CONNECT Initiator.U TO Service.S;
41        ...
42      END;
43  END {InRes}.
```

Abbildung 3: Auszug aus einer Estelle-Spezifikation

eine FIFO-Warteschlange unbegrenzter Kapazität zugeordnet[2]. Nachrichten, die über einen Kanal an eine Modulinstanz gerichtet sind, werden zunächst in die Warteschlange des betreffenden Interaktionspunktes eingereiht. Außerdem kann eine Modulinstanz über ihre Interaktionspunkte Nachrichten an andere Modulinstanzen senden. Der jeweilige Adressat ist durch die aktuelle Verbindungsstruktur eindeutig definiert.

Der Modulrumpf (Zeile 14–29, 30) beschreibt das mögliche Verhalten einer Modulinstanz. Es wird charakterisiert durch die Menge der (internen) Zustände, einen Anfangszustand sowie die Menge möglicher Zustandsübergänge. Der interne Zustand einer Modulinstanz ergibt sich aus dem Hauptzustand (Zeile 18), dem Zustand lokaler Variablen (Zeile 16) sowie dem Inhalt der Warteschlangen. Der Anfangszustand wird im Initialisierungsteil (Zeile 19–20) festgelegt. Erwähnt sei hier noch, daß der *Hauptzustand* eine endliche Wertemenge besitzt (Zeile 18). Damit eignet sich das Modell des erweiterten endlichen Automaten als Basis für die Festlegung der Estelle-Semantik.

Die Menge möglicher Zustandsübergänge wird durch eine Anzahl von Transitionsdeklarationen (Zeile 21–27) definiert. Eine *Transition* kann schalten, wenn die im Bedingungsteil (Zeile 22, 25–26) spezifizierten Restriktionen erfüllt sind. In Zeile 22 wird beispielsweise verlangt, daß die Modulinstanz sich im Hauptzustand `disconnected` befindet. Die `WHEN`-Klausel in Zeile 26 legt fest, daß sich eine Nachricht vom Typ `connect_confirm` an der Spitze der Warteschlange des Interaktionspunktes `U` befinden muß. Weitere Bedingungen können sich auf den sonstigen Zustand sowie die Parameterwerte in den Nachrichten beziehen. Das Schalten einer Transition bewirkt einen atomaren Zustandsübergang, welcher als Folge PASCAL-ähnlicher Anweisungen im Transitionsblock (Zeile 23, 27) sowie in der `TO`-Klausel (Zeile 22, 25) beschrieben ist. Das Schalten der ersten Transition (Zeile 21–23) hat beispielsweise zur Folge, daß die Modulinstanz in den Hauptzustand `connection_requested` wechselt und eine Nachricht vom Typ `connect_request` über den Interaktionspunkt `U` sendet. Der Adressat dieser Nachricht ist die durch die aktuelle Verbindungsstruktur festgelegte Modulinstanz `Service`, die Nachricht wird an die Warteschlange des Interaktionspunktes `S` angehängt. Das Schalten der zweiten Transition (Zeile 24–27) bewirkt u.a. das Entfernen der Nachricht `connect_confirm` aus der Warteschlange des Interaktionspunktes `U`. Dies setzt natürlich voraus, daß die Modulinstanz `Service` diese Nachricht zu einem früheren Zeitpunkt gesendet hat.

Die Struktur der Estelle-Spezifikation in Abbildung 3 kann in einem nachfolgenden Entwurfsschritt verfeinert werden. Beispielsweise ist es möglich, der Modulinstanz `Service` durch Angabe mehrerer Sohnmodulinstanzen sowie ihrer Verbindungsstruktur eine verfeinerte Struktur zu geben. Damit entsteht ein hierarchisch aufgebautes System von Modulinstanzen.

[2]Daneben gibt es auch die Möglichkeit, mehreren Interaktionspunkten dieselbe Warteschlange zuzuordnen.

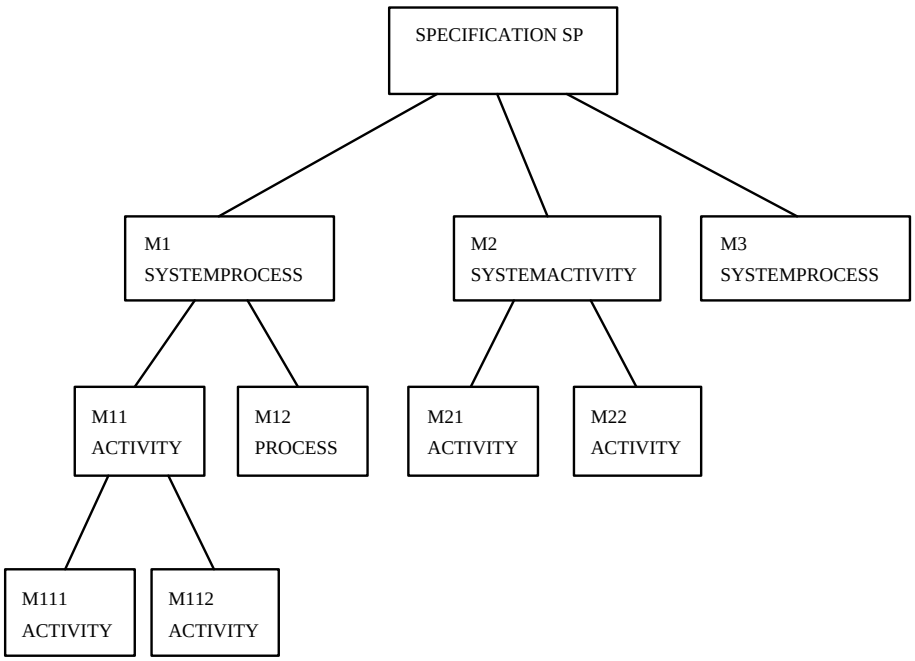

Abbildung 4: Ein Estelle-System

Das Gesamtverhalten eines in Estelle spezifizierten Systems ist durch ein im Estelle-Ausführungsmodell festgelegtes, teilweise synchrones Voranschreiten der Modulinstanzen unter Beachtung von Vorrangregelungen gegeben. Diese Synchronisation kann zum Teil durch die Attributierung von Modulen (z. B. `SYSTEMPROCESS`, Zeile 7, 10) spezifiziert werden. Da sich aus diesen Vorschriften das in der Einleitung genannte Problem ergibt, werden sie nachfolgend im Detail vorgestellt.

3 Nebenläufigkeit in Estelle

Ein in Estelle spezifiziertes System besteht aus einer Anzahl von Subsystemen. Ein Subsystem ist wie das Gesamtsystem hierarchisch aufgebaut und besteht aus einer als `SYSTEMPROCESS` bzw. `SYSTEMACTIVITY` attributierten Modulinstanz (so attributierte Module werden im folgenden auch „Systemmodule“ genannt) sowie allen ihren Nachkommen. Ein Beispiel für ein Estelle-System ist in Abbildung 4 dargestellt. Es enthält drei Systemmodulinstanzen und besteht somit aus drei Subsystemen. Die Attributierungsregeln legen fest, daß die Nachkommen einer Systemmodulinstanz selbst keine Systemmodulinstanzen sein können. Gemäß unserer Definition von „Subsystem“ kann also ein Subsystem keine weiteren Subsysteme enthalten.

Für die nachfolgenden Überlegungen sind zwei Eigenschaften von Subsystemen von Bedeutung. Erstens gibt es zwischen den Modulinstanzen verschiede-

ner Subsysteme keinerlei vom Ausführungsmodell vorgegebene Synchronisation oder Vorrangregelung. Das bedeutet, daß die Subsysteme unterschiedlich schnell arbeiten können, es sei denn, daß die Spezifikation selbst eine Synchronisation per Nachrichtenaustausch zwischen verschiedenen Subsystemen vorsieht. Generell arbeiten Subsysteme jedoch völlig asynchron. Die zweite Eigenschaft ergibt sich indirekt aus den Attributierungsregeln. Da die Vorfahren einer Systemmodulinstanz zwar eine Initialisierungstransition, jedoch keine sonstigen Transitionen besitzen (siehe [ISO89, Kapitel 5.2.1, Punkt (b)]), sind die Menge der Subsysteme und die Verbindungen zwischen Subsystemen nach Abschluß der Initialisierungsphase nicht mehr veränderbar. Damit ist die Grobstruktur eines Estelle-Systems statisch.

Innerhalb eines Subsystems gelten sowohl feste Vorrangbeziehungen als auch weitere, über die Modulattributierung spezifizierbare Synchronisationsvorschriften. Diese Festlegungen bewirken, daß das Schalten von Transitionen im Subsystem in „Runden“ (Schaltzyklen) geschieht. In jeder Runde werden gemäß bestimmter Vorschriften ausführbare Transitionen ausgewählt und geschaltet. Dabei gilt grundsätzlich, daß jede Modulinstanz — unabhängig von ihrer Attributierung — Vorrang vor allen ihren Nachkommen hat. Wenn also eine Modulinstanz eine ausführbare Transition besitzt, bedeutet dies in jedem Fall, daß alle Nachkommen in der aktuellen Runde keine Transition ausführen dürfen. Ob diese Modulinstanz selbst schalten darf, hängt von weiteren Bedingungen ab.

Zum Beginn und Ende jeder Runde besitzt die Systemmodulinstanz das Ausführungsrecht. Das bedeutet, daß sie selbst eine ausführbare Transition schalten darf. Falls sie eine Transition schalten konnte, ist die Runde beendet, und die nächste Runde kann beginnen. Falls keine eigene Transition schaltbar war, gibt die Systemmodulinstanz das Ausführungsrecht — abhängig von ihrer Attributierung — an Sohnmodulinstanzen weiter. Von dort erhält sie es später zurück, womit die Runde beendet ist. In analoger Weise verfahren die Nachkommen der Systemmodulinstanz. Sobald ein Nachkomme das Ausführungsrecht erhält, darf er eine eigene Transition schalten. Konnte eine Transition geschaltet werden, gibt er das Ausführungsrecht an die Vatermodulinstanz zurück. War keine eigene Transition schaltbar, wird das Ausführungsrecht — abhängig von seiner Attributierung — an Sohnmodulinstanzen weitergegeben. Sobald er es von dort wieder erhalten hat, gibt er es an die Vatermodulinstanz zurück.

Die Attributierung einer Modulinstanz legt fest, wie das Ausführungsrecht dann, wenn keine eigene Transition ausführbar ist, an Sohnmodulinstanzen weitergegeben wird. Ist die Modulinstanz mit `PROCESS` oder `SYSTEMPROCESS` attributiert, so gibt sie das Ausführungsrecht unter den bereits genannten Voraussetzungen an *alle* Sohnmodulinstanzen weiter. Die Sohnmodulinstanzen können dann unabhängig voneinander je eine eigene Transition schalten bzw. das Ausführungsrecht wie oben beschrieben weiterreichen. Damit läßt diese Form der Attributierung echte Nebenläufigkeit zu.

Ist die Modulinstanz mit `ACTIVITY` oder `SYSTEMACTIVITY` attributiert, so gibt

sie das Ausführungsrecht unter den bereits genannten Voraussetzungen an *eine* einzelne Sohnmodulinstanz weiter, die sie nichtdeterministisch auswählt. Falls die Sohnmodulinstanz oder einer ihrer Nachkommen eine Transition schalten konnten, kehrt das Ausführungsrecht an die Vatermodulinstanz zurück. Andernfalls wählt die Modulinstanz eine andere Sohnmodulinstanz aus, solange bis eine Transition geschaltet hat oder alle Sohnmodulinstanzen das Ausführungsrecht einmal erhalten haben. Anschließend gibt sie das Ausführungsrecht an ihre Vatermodulinstanz zurück. Damit erzwingt diese Form der Attributierung die Sequentialisierung von Transitionen.

Aus den Vorrangbeziehungen und Synchronisationsvorschriften folgt, daß die Modulinstanzen eines Subsystems in jeder Runde höchstens *eine* eigene Transition schalten können und mit dem Schalten weiterer Transitionen mindestens bis zur nächsten Runde warten müssen. Auch wenn also die Sohnmodulinstanzen einer mit `PROCESS` oder `SYSTEMPROCESS` attributierten Modulinstanz nebenläufig schalten dürfen, so können sie doch keine unterschiedlichen Geschwindigkeiten entwickeln, wie es zwischen verschiedenen Subsystemen der Fall sein kann. Dies schränkt die Brauchbarkeit eines aus mehreren Modulinstanzen bestehenden Subsystems für die parallele Implementierung sehr stark ein.

Sucht man nach den Gründen für diese ausgesprochen restriktiven Regelungen innerhalb eines Subsystems, so stößt man auf zwei Punkte. Eine Modulinstanz kann exportierte Variablen besitzen, auf die außer ihr selbst die Vatermodulinstanz zugreifen darf. Durch die getroffenen Regelungen ist sichergestellt, daß die Transitionen dieser Modulinstanzen niemals gleichzeitig schalten können, wodurch ein simultaner Zugriff auf exportierte Variablen ausgeschlossen ist. Weiterhin kann die Vatermodulinstanz bei der Bestimmung ihrer eigenen Schaltbereitschaft den Wert einer exportierten Variablen einer Sohnmodulinstanz (mit einer `PROVIDED`-Klausel) prüfen und danach beim Schalten der Transition davon ausgehen, daß sich der Wert bis dahin nicht verändert hat. Der zweite Punkt betrifft die dynamische Modifikation von Verbindungen sowie die Freigabe von Modulinstanzen, was innerhalb eines Subsystems zulässig ist. Aus denselben Gründen wie zuvor ist sichergestellt, daß keinerlei Konflikte auftreten können.

Generell wird so das Prinzip des atomaren Schaltens von Transitionen gewahrt. Jede Implementation braucht zur Berechnung des Folgezustandes eine endliche Zeit, in welcher Zwischenzustände auftreten. Durch die Einschränkungen der Parallelität wird ohne weiteren Aufwand von *jeder* Implementation gewährleistet, daß diese Zwischenzustände niemals von anderen Modulinstanzen beobachtet werden können, weshalb man immer konzeptuell annehmen darf, daß die Transition atomar schaltet und daß der Schaltzeitpunkt das Ende der Berechnung des Folgezustandes ist.

Zum Schluß dieses Kapitels sei angemerkt, daß die Estelle-Semantik in [ISO89] die Abläufe in den zuvor beschriebenen Runden in anderer Weise festlegt. Jede Runde besteht dort aus zwei Phasen. In der ersten Phase werden die zu schaltenden Transitionen eines Subsystems ausgewählt, in der zweiten Phase schal-

ten sie in beliebiger Reihenfolge. Dabei geschieht das Schalten einer Transition immer atomar. Auf diese Weise wird eine Interleaving-Semantik ohne explizite Nebenläufigkeit für Estelle angegeben. Die weiter oben beschriebene Ablaufregelung steht jedoch wegen der zuletzt genannten Punkte im Einklang mit der Semantik aus [ISO89], weshalb sie für eine nebenläufige Implementierung von Estelle-Spezifikationen verwendet werden darf.

4 Probleme der nebenläufigen Realisierung von ISO-TP mit Estelle

Ausgangspunkt für die Spezifikation von ISO-TP in Estelle ist die Architektur aus Abbildung 1. Dabei geht es uns im ersten Entwurfsschritt vor allem darum, das konzeptuell erzielte Parallelisierungspotential im Entwurf möglichst vollständig zu erhalten. Die Spezifikation des Komponentenverhaltens erfolgt in späteren Entwurfsschritten und ist nicht mehr Gegenstand dieser Arbeit.

Eine naheliegende Vorgehensweise wäre es zu versuchen, die Architektur aus Abbildung 1 direkt in Estelle wiederzugeben. Wir könnten etwa Module `TPSUI`, `TPPM` und `Medium` in Estelle vereinbaren und anschließend die erforderliche Anzahl von Modulinstanzen generieren. Zur Erzielung des maximalen Parallelitätsgrades könnten wir diese Module mit `SYSTEMPROCESS` (vgl. Kapitel 3) attributieren. Dies hätte jedoch die nachteilige Konsequenz, daß die so erzielte Estelle-Architektur statisch wäre, während die konzeptuelle Architektur die dynamische Erzeugung und Freigabe von TPSUI- und TPPM-Komponenten vorsieht. Letzteres könnten wir in Estelle dadurch erreichen, daß wir ein übergeordnetes Modul `Site` mit `SYSTEMPROCESS` attributieren, das diese dynamischen Änderungen der Estelle-Architektur vornimmt. Die Folge wäre allerdings eine drastische Reduzierung des Parallelitätsgrades, da alle TPSUI- und TPPM-Komponenten Bestandteile desselben Subsystems wären und damit keine unterschiedlichen Geschwindigkeiten beim Schalten von Transitionen mehr entwickeln könnten. Sie müßten sich nach dem Schalten *jeder* Transition synchronisieren, und dies nur aufgrund der Semantik von Estelle (vgl. Kapitel 3), nicht aufgrund der Anwendung.

Ähnliche Betrachtungen können wir für die Wiedergabe der TPPM-Architektur in Estelle anstellen. Jede der beiden zuvor diskutierten Varianten hat zur Folge, daß die Module MACF und SAO nicht mehr als Systemmodule definiert werden könnten, daß also eine asynchrone Ausführung ausscheiden müßte. Damit ergibt sich spätestens an dieser Stelle eine deutliche Einschränkung des konzeptuellen Parallelisierungspotentials. Wenn wir aber dieses konzeptuelle Parallelisierungspotential in unserer Estelle-Spezifikation nicht umsetzen können, verfehlen wir unser Gesamtziel einer Geschwindigkeitssteigerung durch parallele Implementierung (siehe den Beginn von Kapitel 1).

5 Lösungsansätze

Es sind zwei Ansätze zur Lösung dieses Problems denkbar. Entweder wird die notwendige Parallelität außerhalb von Estelle ausgedrückt, indem nur die einzelnen Komponenten von ISO-TP in getrennten Estelle-Spezifikationen beschrieben werden. Dies würde die Beschreibung allerdings inhomogen machen und dazu einen Verzicht auf die Vorteile einer *formalen* Spezifikation bedeuten, da gerade die kritischen Aspekte des Zusammenspiels der Komponenten nur noch informell ausgedrückt wären.

Oder wir erweitern Estelle dahingehend, daß es den beschriebenen Ansprüchen genügt. Dabei bieten sich die folgenden Alternativen für eine Erweiterung von Estelle an:

1. Die Kommunikations- und die Modulinstanzstruktur der asynchronen Systemmodulinstanzen[3] kann auch dynamisch sein.

   Dazu muß dann eine Systemmodulinstanz selbst Brudermodulinstanzen erzeugen können. Bisher kann sie nur Sohnmodulinstanzen erzeugen.

2. Die Modulattributierung wird so erweitert, daß auch asynchrone Sohnmodule von aktiven Modulen möglich werden.

Die erste Alternative bedeutet, daß das Prinzip des hierarchischen Aufbaus der Modulinstanzstruktur aufgeweicht wird. Es wird zum Beispiel eine ganz neue Art der Modulinstanzerzeugung notwendig. Entweder ist die zur neuen Instanz gehörige Moduldefinition syntaktisch in die Definition des Erzeugers eingeschachtelt wie ein Sohnmodul. Dann braucht man eine neue Operation, die dazu ein Brudermodul erzeugt. Oder beide Moduldefinitionen sind gleichrangig. Dann braucht man Ausdrucksmittel, um auf eine gleichrangige Moduldefinition zugreifen zu können, da dies bisher durch die Definition der Gültigkeitsbereiche von Bezeichnern ausgeschlossen ist. In beiden Fällen müßte der bisherige hierarchische Aufbau der Gültigkeitsbereiche (der auf dem von Pascal basiert) erheblich verändert werden.

Darüberhinaus sind auch semantische Fragen zu klären, etwa welche Systemmodulinstanzen Referenzen auf welche ihrer Brüder bekommen. Dies ist zum Beispiel dann wichtig, wenn eine Systemmodulinstanz freigegeben werden soll, oder wenn die Kommunikationsstruktur über den Systemmodulinstanzen verändert werden soll. Letzteres war bisher nicht möglich, da dies immer nur die Vatermodulinstanz leisten konnte, und der Vater eine Systemmodulinstanz immer inaktiv ist. Nun müssen also auch die Systemmodulinstanzen selbst das Recht bekommen, ihre eigene Kommunikationsstruktur zu verändern. Auch hier sind also ganz neue Zugriffsarten notwendig.

[3] Eine Erläuterung des Begriffs der Systemmodulinstanzen findet sich am Anfang von Kapitel 3.

Außerdem verfehlt diese Alternative das in Kapitel 4 genannte Ziel, auch innerhalb der asynchronen TPPM-Module noch weitere asynchron parallele Unterstrukturen spezifizieren zu können. Asynchronität ist auf diese Weise weiterhin nur auf einer einzigen Hierarchieebene möglich.

Und schließlich kann es nun zu den in Kapitel 3, Seite 10 genannten Zugriffskonflikten kommen, die bisher dadurch vermieden wurden, daß einerseits das Vater-Sohn-Vorrangprinzip für Transitionen galt und andererseits die Struktur, auch die Kommunikationsstruktur, der nicht synchronisierten Systemmodulinstanzen statisch war.

Diese Zugriffskonflikte können ebenso auch bei unserer zweiten Alternative zur Erweiterung von Estelle auftreten, da dort zwar die Struktur der Systemmodulinstanzen statisch bleibt, wir aber das Vater-Sohn-Vorrangprinzip aufheben müssen, damit die neuen Sohnmodulinstanzen wirklich asynchron arbeiten können und sich nicht nach jeder Transitionsrunde mit ihrem Vater synchronisieren müssen.

Alle anderen genannten Schwierigkeiten treten allerdings bei der zweiten Möglichkeit nicht auf, da dort die bisherige hierarchische Struktur erhalten bleibt. Es wird lediglich notwendig, die Regeln für den Ablauf einer Transitionsrunde geringfügig zu erweitern, aber diese Erweiterung bestünde im wesentlichen darin, daß die neuen asynchronen Sohnmodulinstanzen von der Transitionsrunde ihres Vaters ausgenommen werden und sich ähnlich wie Systemmodulinstanzen verhalten werden, wie wir weiter unten noch sehen werden.

In Abbildung 5 findet sich der Entwurf einer auf der zweiten Alternative basierenden Struktur von Estelle-Moduldefinitionen für die konzeptuelle Architektur des nebenläufigen ISO-TP aus Abbildung 1. Die obersten beiden Hierarchieebenen beschreiben die statische Struktur der `Site`-Modulinstanzen. Da die nicht attributierte Wurzelmodulinstanz inaktiv sein muß, kann sie nur zum Systemstart einmalig eine Anzahl von gleichartigen `Site`s und das `Medium` generieren, deren Anordnung dann im weiteren statisch bleibt. Diese Modulinstanzen sind dann aktiv, sie können ihre eigenen Sohnmodulinstanzen auch dynamisch erzeugen und freigeben.[4] So soll zum Beispiel für jeden Transaktionszweig, der an einem Aufstellungsort ins Leben gerufen wird, ein Paar aus TP-Nutzerinstanz `TPSUI` und TP-Protokollmaschine `TPPM` erzeugt werden. Durch die Attributierung der `Site`-Modulinstanzen mit einem zusätzlichen Attribut spezifizieren wir nun, daß alle dort dynamisch erzeugten `TPSUI` und `TPPM` vollkommen asynchron zueinander laufen sollen und nur noch dann wegen einer Synchronisierung aufeinander warten müssen, wenn die Semantik von ISO-TP dies über expliziten Nachrichtenaustausch fordert. Ganz analog sieht es eine Hierarchieebene tiefer aus, da auch die `TPPM` in dynamisch erzeugte, asynchrone Sohnmodulinstanzen

[4] Sollte für eine andere Anwendung keine statische Grobstruktur benötigt werden, sondern die gesamte Systemstruktur dynamisch sein, so kann man die oberste Hierarchieebene mit der inaktiven Modulinstanz einfach fortlassen.

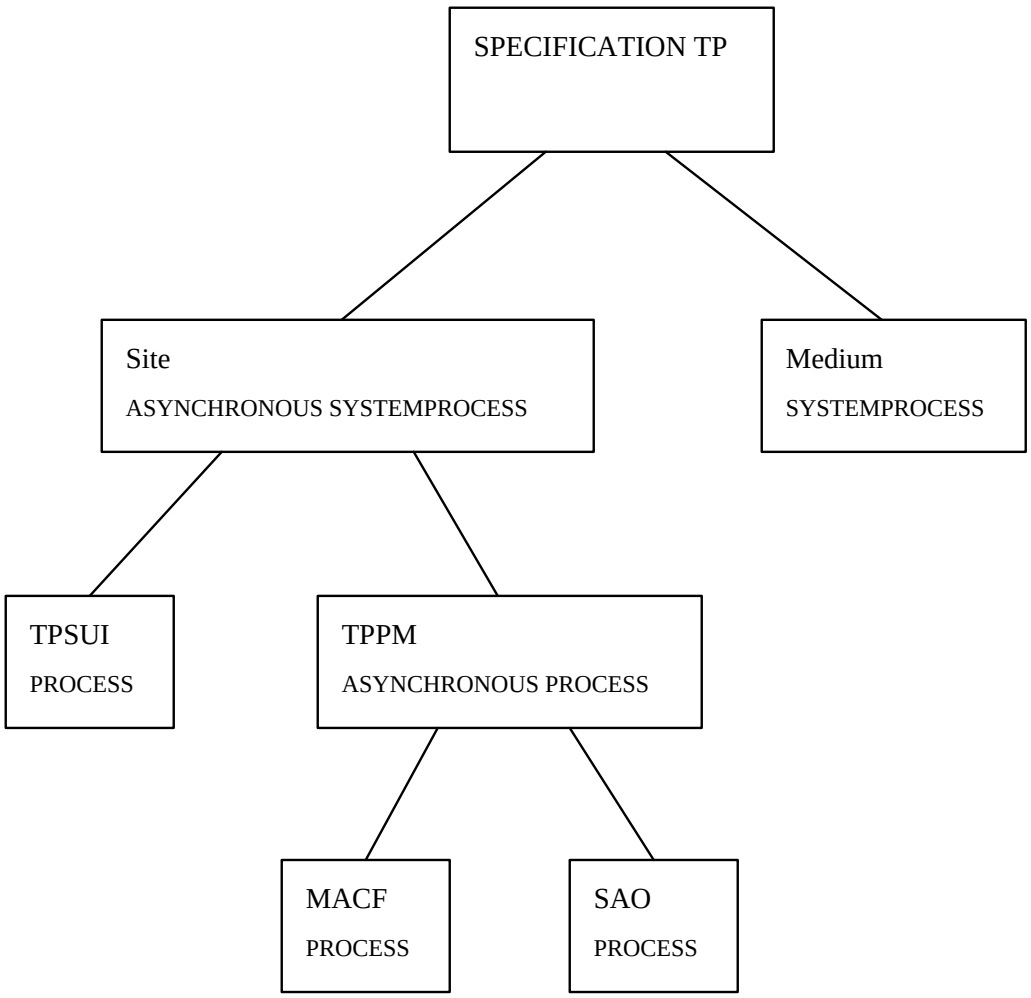

Abbildung 5: Umsetzung der konzeptuellen Architektur von ISO-TP in (erweitertem) Estelle

unterstrukturiert ist. Eine noch weitergehende Unterstrukturierung in asynchrone Sohnmodulinstanzen, zum Beispiel der `SAO`s oder des `Medium`s, ist möglich, wird hier aber nicht weiter betrachtet.

Was die oben erwähnten Zugriffskonflikte betrifft, so werden wir in Kapitel 6 zeigen, daß sie für die Definition der Semantik keine Schwierigkeit darstellen und daß sie sich in einer Implementation effizient auflösen lassen.

Damit fällen wir die Entscheidung, die zweite Alternative weiter zu verfolgen.

6 Erweiterung

Wir erweitern also die Modulattributierung im folgenden so, daß auch asynchrone Sohnmodule von aktiven Modulen möglich werden.

Zuerst wollen wir beleuchten, welche Punkte dabei grundsätzlich beachtet werden müssen. Dann werden wir beschreiben, wie die Syntax und die Semantik im Estelle-Standard ([ISO89]) erweitert werden müssen, wobei die detaillierten Textänderungen im Anhang zu finden sind. Außerdem werden wir zeigen, wie die Erweiterung der Semantik (einfach) in der von uns redefinierten, für Zwecke der Verifikation besonders geeigneten Semantik ([Bre92], [BrGoVo92]) durchgeführt werden kann. Schließlich werden wir im einzelnen untersuchen, welche Probleme bei einer Implementation der erweiterten Semantik auftreten können, und wie diese gelöst werden können.

6.1 Vater-Sohn-Vorrangprinzip

Indem wir zulassen, daß Sohnmodulinstanzen asynchron arbeiten können, müssen wir für diese Modulinstanzen ein wichtiges Prinzip aufgeben, das Prinzip des Vorrangs einer Vatermodulinstanz vor ihren Sohnmodulinstanzen. Denn bereits bisher arbeiteten die Sohnmodulinstanzen eines (als `PROCESS` attributierten) Vaters innerhalb einer einzelnen Transitionsrunde asynchron. Erst der Wechsel zur nächsten Runde, bei dem die Schaltbereitschaft des Vaters überprüft wurde, erzwang auch gleichzeitig eine Synchronisation der Söhne. Daher müssen wir nun untersuchen, was es bedeutet, das Vater-Sohn-Vorrangprinzip fallenzulassen.

Mit dem Vater-Sohn-Vorrangprinzip wurden verschiedene Arten von Zugriffskonflikten zwischen Vater und Sohn aufgelöst. Allerdings traten auch weiterhin bestimmte Arten von Zugriffskonflikten auf, etwa das gleichzeitige Schreiben und Lesen zweier Systemmodulinstanzen in einer Nachrichtenwarteschlange. Bei diesen Konflikten müssen wir zwei Aspekte trennen: Einerseits, wie diese Konflikte in der Definition der Semantik aufgelöst werden, und andererseits, wie eine Implementation gefunden werden kann, die eine derartige Semantik möglichst effizient implementiert.

Bereits bisher werden alle Arten von Zugriffskonflikten in der Definition der Semantik durch zwei Grundkonzepte aufgelöst: Durch die Atomizität des Schal-

tens von Transitionen und durch Interleaving. Das Prinzip der Atomizität besagt, daß die (sichtbare) Wirkung einer Transition von anderen Teilen des Systems entweder noch gar nicht oder bereits vollständig beobachtet werden kann. Transitionen schalten also konzeptuell „blitzartig“ von einem Moment zum anderen. Dieses Konzept allein reicht aber noch nicht aus, um die Zugriffskonflikte aufzulösen. Denn es bleibt offen, was geschieht, wenn zwei Transitionen „gleichzeitig“ schalten. Die Wirkung von „gleichzeitig“ schaltenden Transitionen wird daher in der Definition der Semantik durch das sogenannte Interleaving beschrieben. Die kombinierte Wirkung dieser Transitionen kann jede beliebige (d.h. nichtdeterministische) Auswahl einer sequentiellen Anordnung der Wirkungen sein.

Daher können wir diese Auflösung von Zugriffskonflikten nicht nur auf die bisher möglichen Konflikte anwenden, sondern auf ganz beliebige Wirkungen von Transitionen. Somit können wir das Vater-Sohn-Vorrangprinzip in der Definition der Semantik ohne jede Schwierigkeit fallen lassen.

Anders sieht es dagegen bei der Aufgabe aus, eine einigermaßen effiziente Implementation für diese Semantik zu finden. In Kapitel 6.4 werden wir die einzelnen Arten von Zugriffskonflikten im Detail untersuchen. Dort werden wir sehen, daß bestimmte, zusätzliche Maßnahmen notwendig werden. Insgesamt wird sich eine effiziente Implementation trotzdem als möglich erweisen, mit Ausnahme des Falles der exportierten Variablen. Sohnmodulinstanzen können bestimmte ihrer Variablen als exportiert deklarieren, wodurch ihre Vatermodulinstanz auf diese Variablen ebenfalls lesend und schreibend zugreifen darf. Um hierbei Kollisionen auszuschließen, würde ein unangemessen aufwendiges Protokoll notwendig werden. Wie sich auf Seite 24 allerdings zeigen wird, ist es nicht sinnvoll, asynchrone Sohnmodulinstanzen mit exportierten Variablen zu spezifizieren. Daher nehmen wir diese Anforderung von der Implementationsseite in die Definition der Semantik auf und verbieten die Verwendung von exportierten Variablen in unserer Erweiterung, also für asynchrone Sohnmodulinstanzen. Die bisherigen synchron parallelen oder sequentiellen Sohnmodulinstanzen bleiben davon unberührt.

Wir wollen nun noch anhand eines Beispiel zeigen, was die Aufgabe des Vater-Sohn-Vorrangprinzips konkret bedeutet, einerseits für die Definition der Semantik und andererseits für eine Implementation.

In den Kapiteln 7.6.6 und 9.6.6.2.2 bis 9.6.6.2.4 des Estelle-Standards [ISO89] wird die `DETACH`-Operation definiert, mit der eine Kommunikationsverbindung aufgelöst werden kann. An dieser Definition muß *nichts* geändert werden, auch bei Aufgabe des Vater-Sohn-Vorrangprinzips. Zwei konkurrierende `DETACH`- und `OUTPUT`-Operationen müssen nur gemäß dem Atomizitäts- und Interleaving-Prinzip sequentialisiert werden. Sollte eine Ausgabe einer Nachricht durch die `OUTPUT`-Operation erfolgen, nachdem die Kommunikationsverbindung durch die `DETACH`-Operation aufgelöst worden ist, so schadet dies von der Semantik her nichts. Auch bisher schon war für einen solchen Fall definiert, daß eine derartige Ausgabeoperation ohne Wirkung bleibt, es soll sich ausdrücklich um keinen Fehler handeln.

Allerdings wird ein Spezifierer sicherlich oft durch geeignete Vorkehrungen sicherstellen, daß ein solcher Fall nicht eintreten kann, da er für die Semantik seiner Anwendung nicht sinnvoll ist. Es bleibt festzuhalten, daß die Semantik unserer Erweiterung auch im allgemeinen Fall wohldefiniert ist, daß in der Praxis aber wohl zumeist nur einige einfachere Spezialfälle auftreten werden.

Eine Implementation ist in unserem Beispiel ebenfalls nicht schwer zu finden, da laut Estelle-Semantik alle Ausgaben des Sohnes über den Vater weiter in Richtung Ziel geleitet werden. Damit hat der Vater ohnehin die Kontrolle über alle Nachrichten auf den Nachrichtenverbindungen, die er kontrolliert, und es ist lediglich noch lokal bei dem Vater eine Sequentialisierung des Zugriffs notwendig.

Noch eine weitere Konsequenz aus der Aufgabe des Vater-Sohn-Vorrangprinzips bleibt zu untersuchen. Bisher stellte dieses Prinzip zwei Eigenschaften sicher:

(a) Sobald eine Transition zum Schalten ausgewählt war,[5] konnte sie dies spontan tun, ohne irgendwelche weiteren Bedingungen zu erfüllen.

(b) Die Modulinstanz der Transition konnte nicht vom Vater freigegeben (oder anderweitig modifiziert) werden, bis die Transition geschaltet hatte. (Damit wurde sichergestellt, daß eine Implementation nicht etwa intern bereits mit der Berechnung des Nachfolgezustandes des Sohnes begonnen hatte, wenn der Vater den Sohn freigab. Der Sohn war bei seiner Freigabe also immer in einem „sauberen Zustand“.)

Indem wir das Vater-Sohn-Vorrangprinzip für asynchrone Prozesse aufheben, wird die zweite Eigenschaft nicht mehr erfüllt. Sie ließe sich durch die Forderung an die Semantik wieder erreichen, daß eine Sohnmodulinstanz nur in einem Zustand freigegeben werden darf, in dem sie keine Transition ausgewählt hat. Aber dies würde zwei neue Probleme schaffen.

Zum einen könnte die asynchrone Sohnmodulinstanz einen weiteren asynchronen Sohn (d.h. Enkel ihres Vaters) haben. Sohn und Enkel müßten aufgrund der Estelle-Semantik gleichzeitig freigegeben werden, aber es wäre nicht sichergestellt, daß sie jemals gleichzeitig in einen Zustand kämen, in dem beide keine Transition mehr ausgewählt haben. Damit wäre die Termination der Berechnung der Wirkung der Transition des Vaters nicht mehr garantiert. Im allgemeinen Fall eines ganzen Baumes asynchroner Nachfahren wird dieses Problem noch verschärft.

Zum anderen würde der bereits zum Schalten ausgewählten Transition des Vaters eine nachträgliche Schaltbedingung auferlegt. Damit wäre die oben angeführte Eigenschaft (a) nicht mehr erfüllt. Und diese Eigenschaft ist eine sehr grundlegende in der Semantik, nach unserer Ansicht ist sie wichtiger als die zweite. Die gesamte Aufteilung der Transitionsbearbeitung in zwei Phasen, wobei in

[5] Wir beziehen uns nun auf die formale Semantik in [ISO89] mit dem Zwei-Phasen-Modell von abwechselndem Auswählen und Schalten, nicht mehr auf das (anschaulichere) Modell der Weitergabe von Ausführungsrechten aus Kapitel 3.

der ersten Phase *alle* Schaltbedingungen der Transitionen geprüft und in der zweiten Phase die Schaltwirkungen bestimmt werden, würde anderenfalls als Konzept aufgehoben. Die zweite Eigenschaft dagegen ist mehr für Implementationen interessant. Aber hier gibt es keine grundsätzlichen Probleme bei der Realisierung einer Modifikation, da die Transitionen ohnehin ihre Atomizität wahren müssen. Eventuell müßten nur die lokal berechneten Zwischenergebnisse einer „teilweise ausgeführten“ Transition verworfen werden. (Und selbst dies ist in einer Implementation nicht unbedingt notwendig, wie wir in Kapitel 6.3 sehen werden.)

Daher entscheiden wir uns dafür, die erste Eigenschaft zu erhalten und die zweite fallenzulassen.

Anmerkung: In Kapitel 5.3.4 des Estelle-Standards [ISO89] wird nach Erwähnung der zweiten Eigenschaft argumentiert, daß aufgrund dieser Eigenschaft die mögliche nebenläufige Ausführung von Transitionen durch Interleaving beschrieben werden kann. Dies gilt aber auch dann noch genauso, wenn eine Transition nach ihrer Auswahl zum Schalten nicht mehr schaltet, weil ihre Modulinstanz freigegeben worden ist.

6.2 Syntax

Asynchrone Sohnmodule lassen sich leicht in die Syntax integrieren. Bisher konnten die Definitionen von Vatermodulen auf fünf verschiedene Weisen attributiert sein:

- `PROCESS`
- `SYSTEMPROCESS`
- `ACTIVITY`
- `SYSTEMACTIVITY`
- — ohne Attribut —

Hiermit wurde der Grad der Parallelität der Sohnmodule spezifiziert. Wir fügen einfach ein einziges Schlüsselwort und damit zwei weitere Möglichkeiten hinzu:

- `ASYNCHRONOUS PROCESS`
- `ASYNCHRONOUS SYSTEMPROCESS`

Die *Sohn*module der so attributierten Module sollen, wie der Leser wohl bereits vermutet hat, asynchron parallel arbeiten.[6] So fügt sich die Erweiterung sehr

[6]Anmerkung: Man könnte sich die Frage stellen, ob es sinnvoll wäre, zum Beispiel aus Symmetriegründen auch „asynchrone Aktivitäten“ zuzulassen. Untersucht man aber die Semantik,

gut in die bisherige syntaktische Struktur ein, und die Bedeutung wird intuitiv sofort klar.

Es ist zwar nicht zwingend erforderlich, aber aus Gründen der Konsistenz sollte das Wort `ASYNCHRONOUS` als weiteres Schlüsselwort definiert werden. Dann ist aufgrund einer in Kapitel 7.7.2 des Estelle-Standards [ISO89] formulierten Bedingung ausgeschlossen, daß es als Bezeichner verwendet werden kann, zum Beispiel für Variablen.

Daraus ergibt sich als Nachteil, daß in bereits existierende Spezifikationen, die dieses Wort bereits anderweitig, zum Beispiel als Variablenbezeichner, verwenden, eine Umbenennung erfolgen müßte. (Obwohl dies durch die syntaktische Position eindeutig unterscheidbar wäre.) Dies wird aber dadurch aufgewogen, daß wir mit dem bisherigen Vorgehen in [ISO89] konsistent bleiben. Denn wie in Sprachdefinitionen meist üblich, wird auch sonst die Verwendung derartiger Worte eingeschränkt. Auf diese Weise wird die Spezifikation übersichtlicher gehalten und einem Parser die Arbeit erleichtert. (Er kann nach einem Syntaxfehler leichter wieder aufsetzen und den restlichen Spezifikationstext auf weitere Fehler untersuchen.)

Wie die Definition der Syntax in [ISO89] im Detail angepaßt werden muß, findet sich im Anhang.

6.3 Semantik

Bei der Erweiterung der Semantik haben wir großen Wert auf Kompatibilität zur bisherigen Semantik gelegt. Die Semantik von bereits verfaßten Spezifikationen wird in keinerlei Weise verändert,[7] [8] es gibt lediglich zusätzliche Aus-

die solche Modulinstanzen aufgrund unserer Intuition haben sollten, so stellt man fest, daß diese genau der Semantik von asynchronen Prozessen entspricht. Denn die Vatermodulinstanz von Aktivitäten wählt genau einen ihrer Söhne nichtdeterministisch zum Schalten aus, woraufhin dieser schaltet und dann eine neue Auswahlrunde beginnt. Läßt man hierbei das Vater-Sohn-Vorrangprinzip fort, so müßte jeder Sohn zu jedem beliebigen Zeitpunkt sich selbst auswählen und dann schalten können. Für das intuitive Verständnis einer gegebenen Modulattributierung ist aber sicherlich der Begriff „asynchroner Prozeß“ viel klarer, so daß wir auf die zweite syntaktische Möglichkeit verzichten.

(Ihre Nutzung könnte interessant werden, wenn man auch zulassen möchte, daß innerhalb eines sequentiell arbeitenden, mit `ACTIVITY` attributierten Teilsystems erneut (asynchron) parallele Komponenten vorkommen dürfen. Denn die Attributierungsregeln verbieten bisher, daß zu Modulen, die mit `ACTIVITY` attributiert sind, mit `PROCESS` attributierten Untermodule spezifiziert werden. Allerdings sehen wir keine besonderen Nutzen darin, dies zuzulassen, da ein sequentielles Teilsystem in unserem Verständnis — und in dem der ursprünglichen Entwerfer von Estelle — keine parallelen Komponenten mehr enthält.)

[7] Es wäre auch möglich gewesen, die Semantik der synchronen Parallelität inkompatibel in eine der asynchronen Parallelität zu verändern und so keine neue Parallelitätsklasse einzuführen.

[8] Der einzig denkbare Kollisionspunkt mit vorhandenen Spezifikationen ist die Einführung des neuen Schlüsselwortes `ASYNCHRONOUS`. Da wir wie in Kapitel 6.2 beschrieben den üblichen Ansatz verfolgen, daß Schlüsselworte nicht auch auf andere Weise verwendet werden dürfen, könnten einige einfache Umbenennungen notwendig werden.

drucksmöglichkeiten.

6.3.1 Erweiterung der Semantik im ISO-Standard

In Kapitel 3 haben wir bereits informell das Ausführungsmodell von Estelle beschrieben, also die Regeln, nach denen Transitionen schalten können. Dabei haben wir bereits die Unterscheidung zwischen Systemmodulinstanzen einerseits und ihren weiteren Nachkommen andererseits gemacht. Erstere arbeiten vollständig asynchron, letztere unterliegen dem Prinzip der Transitionsrunden. Auch in der formalen Definition der Semantik von Estelle in [ISO89] wird diese Unterscheidung genauso gemacht.

Es wird dort eine Menge von asynchronen Systemmodulinstanzen $S_1, \ldots, S_n$ definiert. Diese Menge ist statisch. Und es wird dort zu jeder (System-)Modulinstanz P eine Beschreibung des Gesamtzustandes gid_P definiert, diese enthält unter anderem auch die Menge der Beschreibungen aller Sohnmodulinstanzen. Da es in Estelle möglich ist, weitere Sohnmodulinstanzen dynamisch zu erzeugen oder existierende freizugeben, ist diese Menge der Sohnmodulinstanzen dynamisch.

Die Erweiterung der Semantik wird nun folgendermaßen durchgeführt. Die Menge der Systemmodulinstanzen $S_1, \ldots, S_n$ wird erweitert um die (dynamisch erzeugten) asynchronen Modulinstanzen. Auf diese Weise können alle Regeln, die das Schalten von Transitionen beschreiben, unverändert bleiben. Der Ablauf einer Transitionsrunde zum Beispiel verläuft genau wie bisher, da asynchrone Sohnmodulinstanzen dabei nicht als Nachfahren ihrer Väter zählen. Stattdessen wird die Asynchronität dieser Modulinstanzen durch dieselben Mittel beschrieben, mit denen wir auch schon die Asynchronität der Systemmodulinstanzen beschrieben haben.

Die wenigen Textänderungen, die in [ISO89] notwendig werden, sind im Anhang im Detail aufgeführt.

6.3.2 Die Erweiterung in einer alternativen Semantik

In [Bre92] und [BrGoVo92] haben wir den Entwurf einer Estelle-Semantik beschrieben, die für die formale Verifikation besonders geeignet ist. Diese Semantik wird in einem formalen Kalkül definiert, das auf Lamports Temporaler Logik der Aktionen und auf Dijkstras Prädikatentransformatoren basiert. Wir haben eine Bedeutungsfunktion entworfen, die die Bedeutung einer Estelle-Spezifikation als logische Formel angibt. Ein Teil dieser Formel ist dabei immer das Ausführungsmodell von Estelle, das wir als eine einzige, kompakte Teilformel beschrieben haben.

In dieser Semantik wird unsere Erweiterung nun noch viel einfacher. Die Bedeutungsfunktion muß lediglich auch das neue Modulattribut erkennen können, die Teilformel für das Ausführungsmodell bleibt *völlig unverändert*. Denn dort

werden nur Einschränkungen der Nebenläufigkeit beschrieben, und diese gibt es für asynchrone Sohnmodulinstanzen, genau wie für Systemmodulinstanzen, nicht.

6.4 Parallele Implementierung

Die bisherige synchrone Parallelität ist, semantisch gesehen, ein Spezialfall der neuen asynchronen Parallelität. Dies wird dann klar, wenn man die Menge der möglichen Folgen von Transitions-Schaltereignissen (Menge der Traces) betrachtet. Die Transitionen von asynchronen Sohnmodulinstanzen können in beliebiger Reihenfolge in einem Trace vorkommen, sofern es keine Einschränkungen durch andere Bedingungen gibt. Für synchron parallele Sohnmodulinstanzen gibt es dagegen weitere Beschränkungen in der Anordnung, so daß die Menge der Traces im letzteren Fall eine Teilmenge der des ersteren Falls ist.

Wenn wir nun eine Implementation zu einer Spezifikation erzeugen, so ist diese korrekt (bezüglich der „Safety“), wenn sie bei jedem Ausführungslauf immer einen jener Traces auswählt, den die Spezifikation beschreibt. (D.h. „die Implementation tut nur das Erlaubte, nichts Falsches“.)

Damit ist eine Implementation der bisherigen synchronen Parallelität eine korrekte (bezüglich der „Safety“) Implementation auch der asynchronen Parallelität! Da die Auswahl unter mehreren schaltbereiten Transitionen in Estelle nichtdeterministisch erfolgt, steht es einer Implementation frei, auch immer nur bestimmte Transitionen zum Schalten auszuwählen.

Oder mit anderen Worten: Wenn wir das neue Schlüsselwort `ASYNCHRONOUS` aus dem Spezifikationstext einfach überall auskommentieren und das Ergebnis in einen der bisherigen Estelle-Compiler geben, so erhalten wir eine korrekte (bezüglich der „Safety“) Implementation. Diese Implementation ist natürlich recht ineffizient, weil sie sich unnötigerweise nach jeder Transition synchronisiert, aber sie erfüllt weiterhin alle Safety-Eigenschaften.

Allerdings haben wir bisher nur einen recht eingeschränkten Begriff von Korrektheit verwendet. Denn man kann auch sogenannte Liveness-Eigenschaften fordern, d.h., daß in gewissen Situationen etwas Bestimmtes passieren *muß*. Solche Liveness-Eigenschaften werden von einer derart reduzierten Implementation im allgemeinen nicht mehr erfüllt.

Aber hier kommt uns zu Hilfe, daß in der formalen Definition von Estelle Liveness-Zusicherungen für Estelle-Implementationen nicht[9] gefordert werden.

Somit haben wir also bereits viele „Prototyp-Werkzeuge“ für unsere Erweiterung, nämlich alle existierenden Werkzeuge für die bisherige Semantik. Trotzdem macht natürlich die ganze Erweiterung letztlich nur Sinn, wenn insbesondere in den vorhandenen Compilern die echte asynchrone Parallelität ergänzt wird.

Daher werden wir im folgenden im einzelnen untersuchen, was dafür zu tun ist und welchen Aufwand dies bedeutet. Wir werden dabei von einem

[9] Beziehungsweise fast nicht in [Bre92] und [BrGoVo92].

Compiler ausgehen, der ohnehin schon die verteilte Ausführung von Estelle-Spezifikationen unterstützt (etwa Pet/Dingo, [SiSt91b], oder der UHH-Compiler, [Pet91], [KrGo93]). Abgesehen davon, daß die neuen Sohnmodulinstanzen nun auch asynchron laufen können sollen, muß diese neue Version einer Implementation zusätzlich nur die Zugriffskonflikte auflösen, die durch die Aufgabe des Vater-Sohn-Vorrangprinzips entstehen. Kommunikation zwischen Vater und Sohn kann entstehen bei der Übertragung von Nachrichten, einer Änderung der Estelle-Kommunikationsstruktur, der Erzeugung oder Freigabe einer Sohnmodulinstanz und, sofern wir ihn zulassen würden, bei dem Zugriff auf eine exportierte Variable.

Übertragung von Nachrichten

Hier handelt es sich um das Problem der Übertragung von Nachrichten zwischen zwei asynchron arbeitenden Modulinstanzen, unabhängig davon, ob die Nachrichten vom Vater zum Sohn oder umgekehrt übertragen werden. Der Sender schreibt in eine oder mehrere Warteschlangen des Empfängers, wobei dieser bis zum Abschluß der Aktion nicht lesend zugreifen darf, um inkonsistente Ergebnisse zu vermeiden und um die Atomizität der sendenden Transition zu erhalten. Weiterhin kann es vorkommen, daß ein Sender in derselben Transition Nachrichten an mehrere, asynchrone Empfänger ausgibt, und auch diese Aktion muß atomar geschehen. Schließlich ist es möglich, daß mehrere asynchrone Sender in verschiedene Kanäle, aber eine gemeinsame Warteschlange („`COMMON QUEUE`“) schreiben, und diese Schreiboperationen müssen sequentialisiert werden.

Dieses Problem ist nicht neu. Es kommt genauso bereits in der bisherigen Semantik von Estelle vor, und zwar bei der Kommunikation zwischen Systemmodulinstanzen. Wir können also die dort vorhandenen Lösungen direkt verwenden, sie müssen in einem Compiler, der die verteilte Ausführung unterstützt, ohnehin implementiert sein.[10]

Änderung der Kommunikationsstruktur

Eine Vatermodulinstanz darf die Kommunikationsstruktur ihrer Söhne verändern. Dabei ist nun zu beachten, daß sie dies nicht im selben Moment tut, in dem ein asynchroner Sohn versucht, die Kommunikationsverbindung zu nutzen.

Es handelt sich also um das Problem der Serialisierung des Zugriffs auf die Kommunikationsstruktur. Es ist das gleiche Problem, das auch bei dem konkurrierenden Zugriff auf die Warteschlange des Empfängers einer Nachricht auftrat, und folglich sind auch die dortigen Lösungsmöglichkeiten anwendbar.

[10] Nur die Realisierung atomarer, nichtlokaler (Sende-)Aktionen, an der mehr als zwei Modulinstanzen beteiligt sind, ist unter gewissen weiteren Randbedingungen schwierig, existierende Compiler realisieren hier bisher nicht die Standard-Semantik. Eine Diskussion dieses Problems einschließlich der Möglichkeiten zur Abhilfe werden wir an anderer Stelle veröffentlichen.

(Anmerkung: Es ist nicht problematisch, wenn ein Sohn versucht, eine Nachricht über einen Kanal zu senden, den der Vater gerade im Begriff ist aufzulösen. Entweder sendet der Sohn gerade noch vorher und ist erfolgreich, oder er sendet auf einen unverbundenen Interaktionspunkt, was nach der (auch bisherigen) Semantik von Estelle keinerlei Wirkung hat, aber auch kein Fehler ist. Siehe auch Kapitel 6.1. Falls solche Fälle jedoch vermieden werden sollen, ist es die Aufgabe des Spezifizierers, hierfür geeignete Vorkehrungen zu treffen.)

Erzeugung einer Sohnmodulinstanz

Bei der Erzeugung einer neuen Sohnmodulinstanz durch eine Vater-Modulinstanz bestehen überhaupt keinerlei Synchronisationsprobleme. Denn die Aktion der Erzeugung ist ohnehin gleichzeitig ein Akt der Synchronisierung. Erst danach beginnt der Sohn mit eigenständigen, asynchronen Tätigkeiten.

Freigabe einer Sohnmodulinstanz

Hier sind Vater und Sohn zunächst nicht synchronisiert. Aber eine Synchronisation läßt sich in einer Implementation leicht durch Nachrichtenübertragung erreichen, selbst wenn die Nachrichten eine gewisse Laufzeit haben: Zum Beispiel sendet der Vater dem Sohn eine spezielle Nachricht, daß er terminieren soll. Falls in einer Implementation alle Kommunikation des Sohnes über den Vater geleitet wird (s.u.), so muß der Vater ab nun alle Ausgaben des Sohnes vernichten, bis der Sohn seine Arbeit einstellt. Falls aber die Kommunikation des Sohnes aus Gründen der Effizienz am Vater vorbei direkt zum Empfänger geleitet wird, ist noch eine Quittung des Sohnes notwendig, mit der er zusichert, keine weiteren Ausgaben mehr durchzuführen. Erst bei Empfang dieser Quittung kann dann der Vater seine atomare Transition beenden und mit seiner Arbeit fortfahren.

Auf diese Weise kann in einer Implementation auch ein Problem gelöst werden, das durch die auf Seite 17 diskutierte Semantik entstehen kann, wenn die Implementation eines Sohnes intern berechnete Zwischenergebnisse verwerfen soll, weil der Vater den Sohn freigeben will. Aus konzeptuellen Gründen wollten wir dort die nichtdeterministische Entscheidung in der Semantik erhalten, ob die bereits ausgewählte Transition des Sohnes noch schaltet, bevor der Vater den Sohn freigibt, oder nicht. Eine Implementation könnte diese Entscheidung zum Beispiel aufgrund dessen treffen, ob der Sohn bereits irreversible interne Zustandsänderungen durchgeführt hat oder nicht. In dem obigen Beispielprotokoll würde der Sohn mit seiner Quittung dann genau solange warten, bis er diese eine Transition beendet hat.

Also läßt sich auch das Problem der Freigabe einer Sohnmodulinstanz auf das Problem der Übertragung von Nachrichten abbilden.

Zugriff auf eine exportierte Variable

Exportierte Variablen stellen das schwierigste Problem dar. Hier werden entweder Semaphore o.ä. notwendig, falls die Kommunikation über gemeinsamen Speicher erfolgen kann, sonst müssen die exportierten Variablen ebenfalls auf Nachrichtenaustausch abgebildet werden. (Dies ist ohnehin auch bei der bisherigen Art von Parallelität der Fall, wenn die Implementation verteilt realisiert wird. Siehe das Pet/Dingo-Werkzeug, [SiSt91b], und den UHH-Compiler, [Pet91], [KrGo93].) Damit hätten wir zwar in jedem Fall eine Lösung, aber beide wären wesentlich ineffizienter als ein einfacher Speicherzugriff auf eine Variable.

Wir stellen stattdessen die Frage, ob exportierte Variablen ein sinnvolles Konzept für asynchrone Modulinstanzen sind. Denn mit ihnen wollen wir üblicherweise unabhängig voneinander arbeitende Prozessoren (CPUs) modellieren, die keinen gemeinsamen Speicher besitzen.[11] Und ohne gemeinsamen Speicher macht die Spezifikation von exportierten Variablen keinen Sinn.

Daher schlagen wir vor, exportierte Variablen für asynchrone Sohn-Modulinstanzen zu verbieten. (Für synchron parallel arbeitende Sohnmodulinstanzen bleiben sie aber weiterhin erhalten.)

Optimierung der Nachrichtenübermittlung

In Abbildung 6 haben wir das Estelle-System aus Abbildung 4 noch einmal dargestellt, die hierarchische Modulstruktur wird nun allerdings durch Schachtelung ausgedrückt. Zusätzlich haben wir zwei Kommunikationsverbindungen eingezeichnet, eine zwischen den Modulinstanzen M11 und M12 und eine zwischen M111 und M22.

Zwei Interaktionspunkte von Sohnmodulinstanzen können von ihrem Vater durch eine `CONNECT`-Operation verbunden werden, wie dies hier zwischen M11 und M12 beziehungsweise zwischen M1 und M2 geschehen ist. Außerdem kann der Vater einen seiner äußeren Interaktionspunkte mit dem eines Sohnes durch eine `ATTACH`-Operation verbinden, woraufhin der Sohn anstelle des Vaters der Empfänger von Nachrichten wird (M1–M11, M11–M111, M2–M22). Der Vater reicht dann die Nachrichten in beiden Richtungen nur noch durch.

Man sieht, daß in der Definition der Semantik die Kommunikation zwischen zwei Modulinstanzen immer auch einen oder mehrere ihrer Vorfahren einbezieht, die die Nachrichten weiterreichen. Für eine Implementierung können diese Vorfahren leicht zu einem Flaschenhals werden, besonders wenn die Kommunikationsverbindung in der Modulhierarchie weit hinauf reicht. Dies gilt umso mehr, wenn die einzelnen Modulinstanzen auf verschiedenen Prozessoren ausgeführt werden, so daß die Kommunikation aufwendiger wird.

[11] Mehrere Prozesse auf einem Prozessor im Zeitscheibenbetrieb dagegen kann man gut als „`ACTIVITY`“, also als nichtdeterministisch sequentielle Bearbeitung, modellieren, sofern dabei die atomare Ausführung der einzelnen Transitionen gesichert ist.

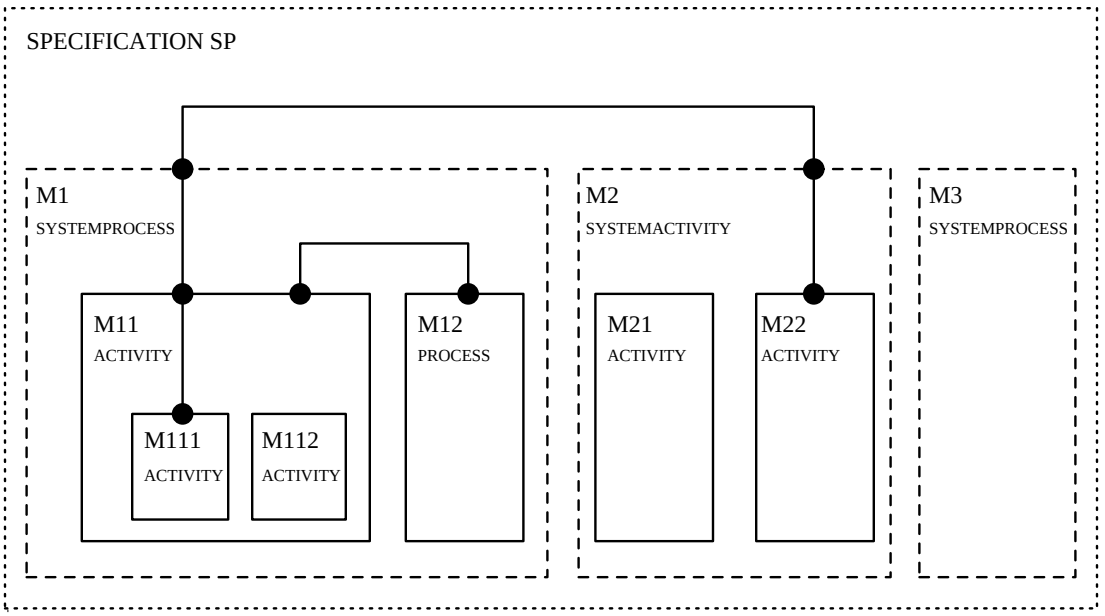

Abbildung 6: Kommunikationsverbindungen in einem Estelle-System

Aus diesem Grunde sollte man über eine Optimierung der Nachrichtenübermittlung in einer Implementation nachdenken. Ein Fall einer derartigen Optimierung existiert bereits. Das Pet/Dingo-Werkzeug ([SiSt91b]) erlaubt, Modulinstanzen auf beliebigen Prozessoren zum Ablauf zu bringen. Und da die Kommunikationsstruktur zwischen Systemmodulinstanzen völlig statisch ist, werden für diesen Fall den jeweiligen Partnern zu Anfang die Netzwerkadressen ihrer Gegenüber mitgeteilt, die sie dann anschließend direkt adressieren.

Eine solche Optimierung ist auch für die neuen asynchronen Sohnmodulinstanzen sehr wünschenswert, damit der Effizienzgewinn durch die Asynchronität nicht wieder aufgehoben wird durch den Aufwand zur Synchronisation mit dem Vater, nur damit dieser die Nachrichten weiterreichen kann. Allerdings wird die Aufgabe etwas schwieriger, da die Kommunikationsstruktur dieser Modulinstanzen auch dynamisch sein kann.

Daher nutzen wir die Beobachtung, daß Änderungen der Kommunikationsstruktur in den meisten Anwendungen seltener vorkommen als die Übermittlung von Nachrichten über sie. In solchen Fällen schadet es wenig, wenn wir zur Veränderung der Kommunikationsstruktur eine etwas aufwendigere Synchronisation zwischen Vater und Sohn durchführen müssen, die die Atomizität der Transitionen sowohl des Vaters wie auch des Sohnes sichert. Dafür können die Söhne ansonsten Nachrichten — völlig ungehindert durch den Vater — direkt übermitteln (sofern die Topologie der physikalischen Nachrichtenverbindungen in der Implementation dies zuläßt).

Es gibt allerdings eine Ausnahme, bei der alle Nachrichten auch weiterhin über

den Vater geleitet werden müssen. Wurde zum Beispiel für die beiden Interaktionspunkte der Modulinstanz M11 in Abbildung 6 eine gemeinsame Warteschlange (`COMMON QUEUE`) spezifiziert, so sind sie nicht mehr unabhängig voneinander. (Die Semantik schreibt dann vor, daß eine bestimmte Reihenfolge der Nachrichten eingehalten wird.) Diesen Zwang zur Ineffizienz kann ein Spezifizierer allerdings sehr leicht vermeiden, indem er jedem Interaktionspunkt asynchroner Modulinstanzen eine individuelle Nachrichtenwarteschlange zuteilt.

Die Möglichkeit zu der beschriebenen Optimierung ist im übrigen ein weiterer Vorteil unserer Erweiterung von Estelle. Für den Fall der synchronen Parallelität wäre eine vergleichbare Optimierung nutzlos, da dort ohnehin in jeder Transitionsrunde eine Synchronisation stattfinden muß.

7 Zusammenfassung

ISO-TP ist ein Beispiel für eine komplexe Anwendung mit viel potentieller innerer Nebenläufigkeit. Diese kann dazu genutzt werden, eine Steigerung der Verarbeitungsgeschwindigkeit durch Parallelisierung zu erreichen. Voraussetzung dazu sind geeignete Ausdrucksmittel für Nebenläufigkeit in der verwendeten Spezifikationstechnik, etwa Estelle. Es zeigte sich, daß Estelle diese Mittel bisher noch nicht ausreichend bietet.

Um zu erreichen, daß in Estelle ein deutlich höherer Grad an Nebenläufigkeit beschreibbar wird, schlagen wir eine Erweiterung von Estelle vor. Sie besteht in einer Erweiterung der Modulattributierung, die auch asynchrone Sohnmodule von aktiven Modulen erlaubt (bei Verzicht auf exportierte Variablen für diese Sohnmodule). Sie bedarf nur sehr weniger Änderungen in der Definition der Syntax und Semantik.

Die Erweiterung ist vollständig kompatibel zu bereits vorhandenen Estelle-Spezifikationen. Es existiert darüber hinaus sogar bereits eine große Zahl an Werkzeugen, die die Erweiterung unterstützt, da jede bisherige Implementation eine zulässige, wenn auch ineffiziente Implementation der erweiterten Semantik ist (nach einer trivialen syntaktischen Ersetzung im Spezifikationstext).

Andererseits wird es durch die neuen Ausdrucksmittel für Nebenläufigkeit jetzt möglich, verteilte Implementationen zu erzeugen, die bei einer stark nebenläufigen Semantik der Anwendung wesentlich effizienter arbeiten. Dies wird einerseits dadurch erreicht, daß sich die Transitionen von asynchronen Sohnmodulinstanzen nicht mehr nach jedem Transitionsschritt synchronisieren müssen. In einer verteilten Implementation führte dies bisher zu sehr vielen Synchronisationsnachrichten, die nur durch das Ausführungsmodell von Estelle erforderlich wurden, oft aber nicht durch die Semantik der jeweiligen Anwendung bedingt waren.

Darüberhinaus wird es sogar möglich, in einer Implementation die Nachrichtenübermittlung zwischen den asynchronen (verteilten) Sohnmodulinstanzen zu

optimieren, da es die Semantik der Erweiterung nicht mehr unbedingt erforderlich macht, daß diese Nachrichtenübermittlung stets über den Flaschenhals der Vatermodulinstanz erfolgt. Bisher war eine solche Optimierung nur für die in ihrer Struktur statischen asynchronen Systemmodulinstanzen möglich.

Literatur

[Bre92] Bredereke, J.: *Entwurf einer formalen Semantik für Estelle unter Verwendung von TLA mit Prädikatentransformatoren.* Diplomarbeit Nr. 858, Universität Hamburg, Fachbereich Informatik, Juni 1992

[BrGoVo92] Bredereke, J., Gotzhein, R., Vogt, F. H.: *Design of a Formal Estelle Semantics for Verification.* In: Diaz M., Groz, R. (Hrsg.): Proceedings of the IFIP TC6/WG6.1 Fifth International Conference on Formal Description Techniques — FORTE '92, Perros-Guirec, Frankreich, 13.–16. Oktober 1992

[DeBu89] Dembinski, P., Budkowski, S.: *Specification Language Estelle.* In: Diaz, M. et al. (Hrsg.): The Formal Description Technique Estelle, North-Holland, 1989, S. 35–75

[Hog89] Hogrefe, D.: *Estelle, LOTOS und SDL. Standard-Spezifikationssprachen für verteilte Systeme.* Springer-Verlag, Berlin, 1989, 188 S.

[ISO81] ISO/TC 97/SC 16: *Data Processing — Open Systems Interconnection — Basic Reference Model.* Computer Networks 5, 1981, S. 81–118

[ISO89] *Estelle — A Formal Description Technique Based on an Extended State Transition Model.* ISO/TC97/SC21, IS 9074, 1989, 179 S.

[ISO89a] *Application Layer Structure.* ISO, IS 9545, 1989

[ISO92] *Information technology — Open Systems Interconnection — Distributed Transaction Processing.* ISO/IEC JTC 1/SC 21, IS 10026, 1992

[Ker92] Kerner, H.: *Rechnernetze nach OSI.* Addison-Wesley, Bonn, 1992, 471 S.

[KrGo93] Kreuz, D., Gotzhein, R.: *A Compiler for the Parallel Execution of Estelle Specifications.* Erscheint im FOKUS-Band des Saur-Verlags, 1993

[Pet91] Peter, D.: *Entwurf, Realisierung und Integration eines Protokolls zur verteilten Ausführung von Estelle-Spezifikationen.* Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Februar 1991

[SiSt91b] Sijelmassi, R., Strausser, B.: *NIST Integrated Tool Set for Estelle.* Proceedings of the IFIP Conference on Formal Description Techniques, FORTE'90, Madrid, Spanien, 1991, 5 S.

Anhang

A Die Änderungen im Text des Estelle-Standards

In diesem Anhang sind die Änderungen im Detail zusammengefaßt, die im Estelle-Standard [ISO89] notwendig werden.

A.1 Syntax

In Kapitel 7.3.6.1 von [ISO89] erweitern wir die BNF-Definition der Klasse einer Modulkopfdefinition um zwei Möglichkeiten (alle Änderungen sind in *kursiv* gesetzt):

class = [*"asynchronous"*] "systemprocess" | "systemactivity"
| [*"asynchronous"*] "process" | "activity"

In Kapitel 7.3.6.2 werden die kontextsensitiven Bedingungen für die Verwendung der Modulattribute spezifiziert. Diese passen wir geeignet an und fügen das Verbot von exportierten Variablen für Sohnmodule von asynchronen Modulen hinzu. Damit lautet die zweite Hälfte dieses Kapitels nunmehr:

> A **VALUE-PARAMETER-SPECIFICATION** of a **parameter-list** of a **module-header-definition** shall not contain a **TYPE-IDENTIFIER** denoting a pointer-containing type.
>
> *The optional **exported-variable-declaration** shall not be used in a **module-header-definition** closest-contained in a **module-body-definition** of which the associated **module-header-definition** used the keyword **asynchronous**.*
>
> The optional keyword **systemactivity** or **systemprocess** shall be used in a **module-header-definition** if the **transition-declaration-part** of at least one associated **body-definition** is non-empty and no enclosing module is attributed with the keyword **systemactivity** or **systemprocess**.
>
> The optional keyword **systemactivity** or **systemprocess** shall not be used if an enclosing (i.e., ancestor) module is attributed with the keyword **systemactivity** or **sytemprocess**.

The optional keyword **activity** of **process** shall be used in each enclosed (i.e., descendent) **module-header-definition** of a module that used a keyword **systemprocess** or **systemactivity**.

The optional keyword **activity** or **process** shall not be used in modules enclosing a module attributed with the keyword **systemprocess** or **systemactivity**.

A **module-body-definition** associated with a **module-header-definition** with a keyword **sytemactivity** or **activity** shall not enclose a **module-header-definition** whith a keyword **process**.

The optional keywords ***asynchronous systemprocess*** *and* ***asynchronous process*** *may be used exactly where otherwise the keywords* ***systemprocess*** *and* ***process*** *would be admissible, respectively. Likewise, all restrictions implied by the use of the keywords* ***systemprocess*** *and* ***process*** *are also implied by the use of the keywords* ***asynchronous systemprocess*** *and* ***asynchronous process****, respectively.*

Remark: From the above constraints one may derive that:

— each active module shall be attributed,

— each ancestor of a module attributed as **systemprocess**, ***asynchronous systemprocess*** or **systemactivity** shall be inactive and unattributed,

— each descendant of a module attributed as **systemprocess**, ***asynchronous systemprocess***, **process** *or* ***asynchronous process*** shall be attributed as **process**, ***asynchronous process*** or **activity**, and

— each descendant of a module attributed as **systemactivity** or **activity** shall be attributed **activity**.

Schließlich muß in Kapitel 7.7.1 die BNF-Definition von „key-words“ um den Text „| *asynchronous*“ erweitert werden.

A.2 Semantik

In Kapitel 5.3.3 von [ISO89] wird beschrieben, wie Transitionen zum Schalten ausgewählt werden. Ab dem ersten mit „**Remark**“ bezeichneten Abschnitt soll es dort nun heißen:

Remark: If P is an instance of an attributed module, then by the nesting principles of 5.2, P is a descendant of a system (or it is a

system itself, *or an asynchronous child process*). Therefore the definitions below serve to characterize the synchronized parallelism within (i.e., among the descendant instances of) a systemprocess, *the asynchronous parallelism within an asynchonous (system)process* and nondeterminism within a systemactivity.

Definition of $AS(gid_P)$:

(a) If the tree of gid_P is simply the single instance P, i.e., $gid_P = (P; s_P)$ and (s_P, t_P) is the local situation of P in gid_P, then $AS(gid_P) = \{t_P\}$ (the set {null} is identified with the empty set).

(b) Let $P_1, P_2, \ldots, P_k$, $k \geq 1$, be children instances of P in the tree of gid_P, and let (s_P, t_P) be the local situation of P in gid_P.

(1) If $t_P \neq$ null, then $AS(gid_P) = \{t_P\}$ (parent priority).

(2) If $t_P =$ null and P is an activity or systemactivity (this implies that each P_i is an activity), then $AS(gid_P)$ equals one of the nonempty sets $AS(gid_P)$, $i = 1, \ldots, k$, if such exists, and is empty otherwise. The choice is nondeterministic. (Notice that each $AS(gid_{P_i})$ is either empty or consists of exactly one transition since activities are substructured only into activities.)

(3) If $t_P =$ null and P is a process (or systemprocess), then $AS(gid_P) = \bigcup_{i=1}^{k} AS(gid_{P_i})$.

(4) If t_P = null and P is an asynchronous process (or asynchronous systemprocess), then $AS(gid_P)$ is empty.

NOTE — The above definition has the simple sense discussed in 5.2. If P is a system module instance, then $AS(gid_P)$ denotes the set of transitions that this system may currently (i.e., in gid_P) perform and will actually execute, in a synchronized and parallel way, from among those transitions which are chosen autonomously by every component instance of the system. Point (b)(1) expresses the parent/children priority principle; point (b)(2) shows nondeterministic execution among a hierarchy of activities, and point (b)(3) indicates that every transition offered by a process within a system, if not constrained by an ancestor/descendant priority conflict, will be selected and executed. *Point (b)(4) reflects that the children of asynchronous (system)processes proceed totally independent of their parent; case (b)(4), together with case (b)(1), is equivalent to case (a), i.e., the parent behaves like a leaf instance. See chapter 5.3.4 for how the transition selection of systemprocesses and children of asynchronous processes is done.*

In Kapitel 5.3.4 wird die Entwicklung eines Gesamtsystems beschrieben, also einer Menge von Systemmodulinstanzen. Zu einer Spezifikationsmodulinstanz SP

wird dort die Menge der Systemmodulinstanzen $S_1, \ldots, S_n$ definiert, wobei n aufgrund der statischen Struktur der Systemmodulinstanzen während der Lebenszeit des Systems ebenfalls fest ist. Die Systemmodulinstanzen laufen vollständig unabhängig voneinander und damit vollständig asynchron. Genau dies sollen auch die asynchronen Sohnmodulinstanzen tun, die wir neu einführen wollen. Daher erweitern wir diese Liste um die dynamisch erzeugten, asynchronen Sohnmodulinstanzen. Damit wird diese Liste in ihrer Länge dynamisch, ansonsten aber kann der Text des Standards fast unverändert bleiben. (Dies erreichen wir dadurch, daß wir ab jetzt die *Gesamt*-Liste mit $S_1, \ldots, S_n$ bezeichnen und den statischen ersten Teil dieser Liste mit $S_1, \ldots, S_m$.) Insbesondere bleibt die Regel zur Bestimmung der nächsten globalen Situation unverändert:

> For every $i = 1, \ldots, n$,
>
> (a) if $A_i = \emptyset$, then for every $AS(gid/S_i) \in AS^*(gid/S(_i)$, $(gid_{SP}; A_1, \ldots, AS(gid_{SP}/S_i), \ldots, A_n)$ is a next global situation of sit.
>
> (b) if $A_i \neq \emptyset$, then for every $t \in A_i$, $(t(gid_{SP}); A_1, \ldots, A_i \backslash \{t\}, \ldots, A_n)$ is a next global situation of sit.

Im Detail soll es ab dem zweiten Absatz von Kapitel 5.3.4 nun heißen:

> Denote these system modules $S_1, \ldots, S_m$, where $m \geq 1$, and m may vary depending on the initial gid of SP.
>
> *A module may be specified to run in an asynchronous parallel fashion by attributing its parent* ***asynchronous process*** *or* ***asynchronous systemprocess****. Denote the module instances created according to these module definitions by $S_{m+1}, \ldots, S_n$, where $n \geq m$, and n may vary depending on the dynamic structure of module instances.*
>
> Notice that, for every gid_{SP}, its part rooted at S_i ($i = 1, \ldots, m, \ldots, n$) is a [...]
>
> By a global situation of SP, we mean a tuple: $(gid_{SP}; A_1, \ldots, A_m, \ldots, A_n)$, where each A_i is a set of transitions of the component instances of the system rooted at S_i.
>
> A global situation of SP is said to be initial if and only if
>
> gid_{SP} is initial, and
> $A_i = \emptyset$, for $i = 1, \ldots, m$.

In Kapitel 6.1 hatten wir zwei Eigenschaften diskutiert, die unter unserer Erweiterung nicht beide erhalten werden konnten. Erstens kann bisher eine Transition (z.B. einer Vatermodulinstanz) spontan und ohne irgendwelche Vorbedingungen schalten, nachdem sie dazu ausgewählt worden ist. Und zweitens kann

bisher eine (z.B. Sohn-)Modulinstanz nicht freigegeben werden, solange sie eine Transition besitzt, die bereits zum Schalten ausgewählt worden ist.

Wir hatten uns entschieden, die erste Eigenschaft zu erhalten und die zweite fallenzulassen. Damit führen wir einen Absatz nach dem letzten Zitat folgende Änderung durch:

> The following property is assumed below:
>
> > if two transitions t_i and t_j are selected by S_i and S_j, respectively, and both are defined as partial functions for gid_{SP}, then t_i is also defined for any $t_j(gid_{SP})$ and similarly, t_j is defined for any $t_i(gid_{SP})$, *except if t_j destroys S_i or t_i destroys S_j*, respectively.
>
> NOTE — For a given gid_P, there are only three ways the execution of a transition t_i may make another transition t_j undefined: ... In each of these cases, t_i must be a transition of the parent instance of the module instance P_j in gid_{SP}. This is not possible by the parent/children priority principle of the definition of $AS(gid_{SP})$ from 5.3.3*, except for asynchronous children.*

A.3 Modell

In Kapitel 5 von [ISO89] wird das Estelle zugrunde liegende Modell beschrieben. Bei der Auflistung der möglichen Modulattribute müssen entsprechend die beiden neuen Attribute „***asynchronous process***“ und „***asynchronous systemprocess***“ ergänzt werden (Kapitel 5, fünfter Absatz, sowie Kapitel 5.2, erster und dritter Absatz und Kapitel 5.2.1, zweiter Absatz).

Außerdem muß in Kapitel 5.2.1 bei der Beschreibung der Regeln für die Modulverschachtelung ein weiter Punkt ergänzt werden:

> *(f) Asynchronous systemprocess modules or asynchronous process modules may be used exactly where otherwise systemprocess modules or process modules would be admissible, respectively. Likewise, all restrictions implied by systemprocess modules and process modules are also implied by asynchronous systemprocess modules and asynchronous process modules, respectively.*

In Kapitel 5.2.2 wird Parallelität behandelt. Im ersten Absatz muß dort eingefügt werden:

> As noted above, only active module instances may dynamically create, release, and change the connection configuration of its children instances; in doing this, an active instance acts as a supervising-like manager over its *non-asynchronous* children instances. [...]

Nach dem zweiten Absatz muß der Absatz eingefügt werden:

> *The children of asynchronous (system)processes behave as if they were systemprocesses themselves, but can be dynamically created and released.*

In der Aufzählung am Ende von Kapitel 5.2.2, in der die Ausdrucksmöglichkeiten beschrieben werden, die einem Spezifizierer zur Verfügung stehen, muß nach dem letzten Punkt ein weiterer eingefügt werden:

> In summary, a specification designer combining the general feature of parent/children priority with the nesting principles charecterized above may:
>
> > [...]
> >
> > — *Decide, at each level of substructuring of processes, that new asynchronous systems begin, running independently of their ancestors. These systems resemble the top-level systemprocesses but may be created and released dynamically.*