Aktive und mobile Objekte als Modellierungskonzept für dezentrale Ingenieuranwendungen

L. Keller *C. Kilger* *D. Kottmann* *G. Moerkotte* *A. Schill*
H.-D. Walter *A. Zachmann*

Fakultät für Informatik
Universität Karlsruhe
Postfach 6980
D–76128 Karlsruhe
kilger|moer|walter|zachmann @ira.uka.de
lkeller|kottmann|schill @telematik.informatik.uni-karlsruhe.de

Zusammenfassung

Dieses Papier zeigt, wie schon durch sehr einfache Erweiterungen einer herkömmlichen objektorientierten Datenbankprogrammiersprache um Basiskonzepte zur Modellierung aktiver und mobiler Objekte technische Anwendungen, in denen dezentrale und (geographisch) verteilte Informationsverarbeitung eine bedeutende Rolle spielen, wirkungsvoll unterstützt werden können.

Die Ideen werden an einem konkreten Beispiel aus dem Fertigungsbereich demonstriert.

1 Einleitung

Die Integration rechnergestützter Werkzeuge, wie z.B. CAD- oder PPS–Systeme im ingenieurwissenschaftlichen Bereich, zu einem unternehmensweiten Verbund basiert auf der Möglichkeit des Datenaustauschs zwischen den beteiligten Komponenten. Dazu ist nach den heutigen Erfahrungen die Datenbanktechnologie das am besten geeignete Medium. Allerdings stellen moderne Anwendungsgebiete Anforderungen, die mit klassischer Datenbanktechnolgie — wie bspw. den relationalen Systemen — nicht erfüllt werden kann.

Seit den 80er Jahren wurden die Unzulänglichkeiten herkömmlicher, relationaler Datenbanksysteme erkannt (z.B. [29, 38]). Die relationale Modellierung von Anwendungsobjekten ist mit zahlreichen Nachteilen verbunden. So ist mit der Segmentierung der Objekte auf verschiedene Relationen und ihre Wiedergewinnung über künstliche Schlüsselattribute eine hohe Redundanz verbunden, die zu wesentlichen Verarbeitungsproblemen und signifikanten Effizienzverlusten führen. Objekte ingenieurwissenschaftlicher Anwendungen weisen oft ein anwendungsspezifisches Verhalten auf, das in einem relationalen Schema nicht erfaßt werden kann. Diese Probleme führten in den 80er Jahren zu einem

Paradigmenwechsel hin zu objektorientierten Datenbanken, deren Datenmodell — an objektorientierte Programmiersprachen angelehnt — die angesprochenen Probleme vermeidet. In ihnen wird ein Objekt als Aggregation von Struktur und Verhalten definiert. Über systemweit eindeutige Objektkennungen können beliebige Objektstrukturen konstruiert und referenziert werden.

Die über die Datenbank zu integrierende rechnergestützten Werkzeuge sind jedoch über Büros, Gebäude und oft sogar Standorte verteilt. Da zudem zwischen den Anwendern eine enge Kooperation besteht, die sich durch iterative, ineinander verzahnte und auch parallel ausführbare Tätigkeiten auszeichnet, sind eine geeignete Berücksichtigung der Kommunikation zwischen Prozessen der Anwendungswelt und ihre Abwicklung in einer verteilten Umgebung entscheidende Aspekte.

Das vorliegende Papier stellt ein integriertes Systemmodell vor, welches neben rein passiven Objekten auch aktive und in einer verteilten Umgebung mobile Objekte unterstützt. Dieses Modell wurde im Rahmen des Sonderforschungsbereichs 346 (Rechnerintegrierte Konstruktion und Fertigung von Bauteilen) an der Universität Karlsruhe entwickelt und wird in einer Modellfabrik (dem *Produktionstechnischen Labor* des Instituts für Werkzeugmaschinen und Betriebstechnik der Universität Karlsruhe) erprobt.

Im nächsten Abschnitt wird zunächst ein Anwendungsszenario aus der Fertigung beschrieben, welches die Anforderungen an ein integriertes Systemmodell konkretisiert. Anschließend wird in Abschnitt 3 das Szenario mittels der im SFB 346 entwickelten objektorientierten Datenbanksprache GOM stellvertretend für heute gängige Systeme modelliert. Diese Modellierung wird anschließend bewertet und daran Anforderungen für notwendige Erweiterungen abgeleitet. Eine mögliche Umsetzung dieser Anforderungen wird in Abschnitt 4 vorgestellt und die Vorteile bei der Anwendung auf unser Szenario gezeigt. Der Abschnitt wird dann mit einer Diskussion der Grenzen dieses Ansatzes abgeschlossen. In Abschnitt 5 erfolgen dann einige Anmerkungen zur Implementierung dieses Ansatzes. Abschnitt 6 gibt eine abschließende Zusammenfassung und erläutert die Stellung dieser Arbeiten im SFB 346.

2 Anwendungsszenario

Das Ziel des Szenarios ist es, die Schwächen derzeitiger objektorientierter Datenbanktechnologie beim Einsatz in verteilten Ingenieuranwendungen zu illustrieren. Als Anwendungsbereich wählen wir einen Ausschnitts aus der Produktionsplanung und -steuerung eines Unternehmens.

Kunden können Aufträge zur Fertigung einer bestimmten Menge eines bestimmten Teils erteilen. Diese Aufträge werden zu bestimmten Zeitpunkten in den laufenden Betrieb der Fertigung eingelastet. Hierzu wird ein zu dem Teil gehöriger Arbeitsplan ermittelt und abhängig von der aktuellen Auslastung der Betriebsmittel ein Durchlaufplan generiert.

Ein Arbeitsplan besteht aus einer Folge von Arbeitsgängen. Ein Arbeitsgang wiederum ist durch eine Menge von Kennzahlen (z.B. Übergangszeit, Maschinenzeit, Rüstzeit) beschrieben. Ferner sind jedem Arbeitsgang ein Betriebsmitteltyp, eine Menge von Werkzeugen, die zu seiner Durchführung benötig werden, und ein technisches Verfahren zugeordnet. Werkzeuge und technisches Verfahren werden im folgenden nicht mehr betrachtet.

Zur Einlastung eines Auftrags in die Fertigung wird ein Arbeitsplan für das zu fertigende Teil ausgewählt, und daraus ein Durchlaufplan erzeugt. Ein Durchlaufplan besteht

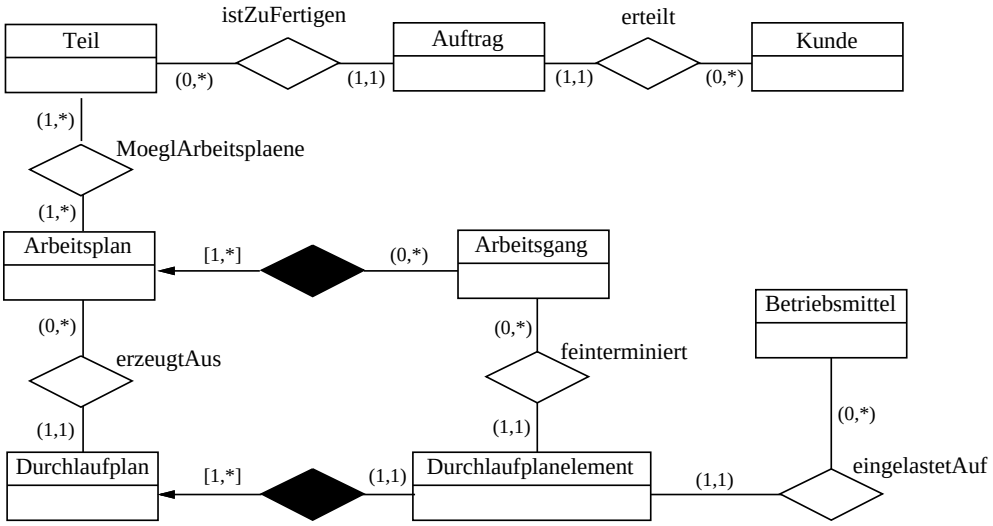

Abbildung 1: Das OMK-Schema des Anwendungsszenarios

aus einer Folge von Durchlaufplanelementen. Ein Durchlaufplanelement repräsentiert die Einlastung eines Arbeitsgangs des ausgewählten Arbeitsplans auf einem konkreten Betriebsmittel. Jedes Durchlaufplanelement enthält u.a. die genaue Belegungszeit des Betriebsmittels, die abhängig von der Auslastung des Betriebsmittels ermittelt wird. Ein Durchlaufplan legt somit fest, welche Arbeitsschritte auf welchem Betriebsmittel zu welchem Zeitpunkt durchgeführt werden. Hieraus ergibt sich auch der Fertigstellungszeitpunkt für den gesamten Auftrag.

Abb. 1 zeigt einen Ausschnitt des konzeptuellen Schemas für dieses Beispiel. Das konzeptuelle Schema ist in OMK[1] spezifiziert, einer auf dem ER-Modell [9] und der Modelliersungsmethode OMT [34] basierenden grafischen Modellierungssprache. Rechtecke stellen Objekttypen dar, Rauten repräsentieren Beziehungstypen zwischen Objekttypen. Die schwarz ausgefüllten Rauten bezeichnen die Aggregation von Unterobjekten zu Oberobjekten. Beispielsweise besteht ein Arbeitsplan aus einem oder mehreren Arbeitsgängen.

Das hier auf konzeptueller Ebene eingeführte Datenbankschema wird mit Hilfe eines objektorientierten Datenmodells auf ein logisches Datenbankschema abgebildet (siehe Abschnitt 3). Aufbauend auf diesem Schema werden dann die eigentlichen Anwendungen erstellt, die die Einplanung der Aufträge und die Überwachung und Steuerung der Produktion vornehmen. Wir werden diese Anwendungsprogramme im nächsten Abschnitt beschreiben und ihre Anforderungen an die Datenbank analysieren.

3 Modellierung mit der objektorientierten Datenbankprogrammiersprache GOM

In diesem Abschnitt werden die Basiskonzepte der objektorientierten Datenbankprogrammiersprache GOM auf informelle Weise anhand unseres Beispiels beschrieben. Im wesentlichen beinhaltet GOM alle für ein objektorientiertes Datenbankmodell als notwendig identifizierten Eigenschaften [2] in einem orthogonalen syntaktischen Rahmen. Wir be-

[1] **O**bjektorientierte **M**odellierungsmethode **K**arlsruhe

schränken uns an dieser Stelle jedoch auf die Eigenschaften von GOM, die für das weitere Verständnis unbedingt notwendig sind. Leser, die sich weitergehend für GOM interessieren, seien auf [25] und [23] verwiesen.

3.1 Kurze Einführung in GOM

3.1.1 Objekttypen

Objekte in GOM beinhalten eine *strukturelle* und eine *verhaltensmäßige* Beschreibung. Objekte mit ähnlichen Eigenschaften, d.h. ähnlicher Struktur und ähnlichem Verhalten, werden zu Objekttypen zusammengefaßt. Objekttypen werden über den *Typdefinitionsrahmen* eingeführt; die Struktur des Typdefinitionsrahmens zeigt **Abb. 2** am Beispiel des Objekttyps Betriebsmittel.

Jeder neu definierte Objekttyp bekommt einen eindeutigen Typnamen zugeordnet. Die **body**-Klausel enthält die Definition der strukturellen Repräsentation des Typs. Wir unterscheiden zwischen tupelstrukturierten Typen und Kollektionstypen. Ein tupelstrukturierter Typ wird durch die Angabe seiner Attribute $[A_1 : t_1, \ldots, A_n : t_n]$ definiert; die A_i sind die Namen der Attribute, die t_i die Namen der Attributtypen. Ein Beispiel für einen tupelstrukturierten Typ ist der Typ Betriebsmittel in Abb. 2. Kollektionstypen sind mengen- oder listenstrukturiert. Ein Mengentyp wird mit $\{\, t \,\}$ und ein Listentyp wird mit $\langle\, t \,\rangle$ bezeichnet; t ist hierbei der Name des Elementtyps.

Das Verhalten von Objekten (eines Typs) wird durch eine Menge typ-assoziierter Operationen spezifiziert. Die **operations**-Klausel enthält die abstrakten Signaturen dieser Operationen. Jede Operationssignatur besteht aus dem Operationsnamen, einer Liste von Eingabeparametern und einem Ergebnistyp. Die Implementierung dieser Operationen erfolgt in der **implementations**-Klausel in einer C-ähnlichen Syntax, auf die hier nicht näher eingegangen werden soll.

Zur Strukturierung der Menge der Objekttypen können diese in einer Typhierarchie angeordnet werden. Hierzu kann (optional) für jeden Objekttyp ein Obertyp in der **supertype**-Klausel angegeben werden. Die Struktur und die Operationen eines Obertyps werden an alle seine Untertypen vererbt.

GOM bietet für bestimmte Objekttypen vordefinierte Operationen an:

Tupeltypen Für tupelstrukturierte Typen gibt es vordefinierte Operationen, die den Zugriff auf die Attribute realisieren ($\langle$*Attributname*$\rangle\rightarrow$zum Lesen des bezeichneten Attributs, $\langle$*Attributname*$\rangle\leftarrow$ zum Modifizieren des bezeichneten Attributs).

Kollektionstypen Analog zu den Operationen auf tupelstrukturierten Objekten gibt es auch vordefinierte Operationen auf mengen- und listenstrukturierten Objekten. Für Mengen werden z.B. Operationen zum Ein- und Ausfügen von Elementen (*insert* und *remove*) angeboten, in Listen kann das n-te Element (*n_th*) ermittelt werden.

Eine aus Datenbanksicht essentielle Operation stellt *select* dar, mit der aus einer Menge die Objekte ermittelt werden, die ein bestimmtes Selektionsprädikat erfüllen.

Auf die Daten eines Objektes — bei Tupelobjekten die Attribute, bei Kollektionen die Elemente — kann nur mit Hilfe der vordefinierten bzw. vom Benutzer definierten Operationen zugegriffen werden; direkte Lese- bzw. Schreibzugriffe auf die Daten eines Objektes sind nicht möglich (*Objektkapselung*).

```
type Betriebsmittel is
   body [typ: string;
         belegungen: Durchlaufplanelementfolge;]
   operations
      declare Betriebsmittel: string →void;
      declare belegen: Durchlaufplanelement, int →void;
      declare freigeben: Durchlaufplanelement: →void;
      declare bearbeiten: Durchlaufplanelement: →void;
   implementation ...
end type Betriebsmittel;
```

Abbildung 2: Objekttypdefinition *Betriebsmittel*

3.1.2 Objekte

Für alle Objekttypen wird der Konstruktur *create* angeboten, mit dem eine Instanz des Typs (ein Objekt) erzeugt werden kann. Bei der Erzeugung eines Objekts wird automatisch — falls vom Benutzer definiert — ein Initialisator aufgerufen, der als typassoziierte Operation definiert sein muß. Initialisatoren besitzen immer den gleichen Namen wie der Typ, mit dem sie assoziiert sind.

Jedes Objekt erhält bei seiner Instantiierung eine *Objektidentität*, genannt OID, zugeordnet, die sich während seiner Lebenszeit nicht ändert. Die Objektidentität ist vom Speicherort des Objekts und vom internen Zustand des Objektes unabhängig. Die OID eines Objekts wird systemintern benutzt, um das Objekt eindeutig zu referenzieren.

Objekte kann man als Tripel (*OID*, *Typ*, *Zust*) auffassen. Dabei bezeichnet *OID* die Objektidentität des Objekts, *Typ* bezeichnet den Typ, von dem das Objekt instantiiert wurde, und *Zust* bezeichnet den internen Zustand (bei Tupeltypen sind dies die Attributwerte) des Objekts.

3.1.3 Persistenz

Unter Persistenz versteht man das Überleben von Programmkomponenten über die Ausführungszeit des Programms hinweg. Dazu müssen diese Komponenten natürlich dauerhaft auf dem Hintergrundspeicher abgespeichert werden. In GOM unterscheidet man zwei unterschiedliche Komponenten, die potentiell persistent sein können: Objekte und Variablen.

Jedes GOM-Objekt, das durch Instantiierung eines Typs erzeugt wurde, „versteht“ den Operationsaufruf *persistent*. Erst nach Aufruf der *persistent*-Operation wird das Objekt dauerhaft in der Objektbank gespeichert.

Man kann in einem GOM-Programm Variablen „persistent machen“, indem man bei der Deklaration das Schlüsselwort **persistent** voranstellt. Wird ein Programm, in dem persistente Variablen definiert sind, beendet und zu einem späteren Zeitpunkt neu gestartet, so besitzt die Variable den letzten, im vorherigen Programmlauf zugewiesenen Wert. Persistente Variablen stellen somit die „Einstiegspunkte“ in die Datenbasis dar.

3.1.4 Anwendungsprozesse und Transaktionen

Um Anwendungen auf einer GOM-Objektbank zu implementieren, wird die Möglichkeit angeboten, Prozesse zu definieren, z.B.

process Auftrag_einlasten **is** ...; !! Rumpf der Prozeßdefinition

Der Rumpf einer Prozeß-Definition wird wie eine typspezifische oder freie Operation implementiert. Die **process**-Klausel entspricht in etwa dem Schlüsselwort **main** aus C++.

Der Zugriff auf die Objektbank erfolgt immer im Rahmen einer Transaktion [3]. Mit dem Aufruf eines Prozesses wird daher auch automatisch eine Transaktion gestartet. Da solche Prozesse auch langlebig sein können (z.B. interaktive Anwendungsprogramme), wird es dem Anwendungsprogrammierer ermöglicht, durch Aufruf einer vordefinierten Operation *commit* die Transaktion innerhalb eines Prozesses zu beenden. Nach einem *commit* wird automatisch eine neue Transaktion gestartet. Ein Prozeß besteht somit aus einer Folge von Transaktionen.

3.2 Modellierung des Beispiels

In diesem Abschitt beschreiben wir, wie unser Anwendungsszenario mit Hilfe der „klassischen" Modellierungskonstrukte von GOM modelliert werden kann.

Abb. 2 zeigt die GOM-Definition des Objekttyps *Betriebsmittel*. Das Attribut *typ* enthält den Bezeichner für die Art des Betriebsmittels (z.B. „Fräsmaschine"), *belegungen* verweist auf ein Listenobjekt, das alle für diese Maschine disponierten Fertigungsschritte (in Form von Durchlaufplanelement-Objekten) enthält.

Bei der Instantiierung des Typs wird über den Initialisator (Operation *Betriebsmittel*) gleich der Typ des Betriebsmittels eingetragen. Mit der Operation *belegen* wird ein Betriebsmittel mit einem Durchlaufplanelement für eine bestimmte Dauer belegt. Durch Aufruf der Operation *freigeben* wird die Belegung des Betriebsmittels mit dem als Parameter übergebenen Durchlaufplanelement wieder aufgehoben; dies kann sich bspw. im Rahmen einer Umplanung ergeben.

Die Operation *bearbeiten* führt die eigentliche Steuerung des jeweiligen Fertigungsschrittes aus.[2] Einziger Eingabeparameter für *bearbeiten* ist das zu bearbeitende Durchlaufplanelement.

Abb. 3 zeigt die zur Beschreibung eines Durchlaufplans notwendigen Objekttypen. Der Typ *Durchlaufplanelement* enthält auftragsunabhängige Informationen (Attribut *vorgang* vom Typ *Arbeitsgang*), die disponierte Start- und Endzeit des Bearbeitungsvorgangs (*von, bis*), eine Referenz auf das Durchlaufplanobjekt, zu dem das Durchlaufplanelement gehört, sowie eine Referenz auf das Betriebsmittel, auf dem der Vorgang eingeplant ist.

Als einzige Operation neben den Attributzugriffsoperationen ist auf dem Typ Durchlaufplanelement ein Initialisator definiert. Bei der Initialisierung wird gleich das zugehörige Betriebsmittel durch Aufruf der Operation *belegen* belegt und die Attribute *von* und *bis* gesetzt.

Der Typ *Durchlaufplan* besteht aus einer Liste von Durchlaufplanelementen (Attribut *elemente*), sowie dem zugehörigen Arbeitsplan (*plan*). Bei der Instantiierung eines Durchlaufplans werden Auftrag und Arbeitsplan angegeben, und aus jedem in dem Arbeitsplan

[2] Man stelle sich etwa ein NC-Programm vor, das durch die Operation *bearbeiten* gestartet wird.

enthaltenen Arbeitsgang wird ein Durchlaufplanelement-Objekt erzeugt. Diese wiederum werden — wie wir oben gesehen haben — gleich den entsprechenden Betriebsmitteln zugeordnet und mit den Bearbeitungsterminen versehen.

Im Falle einer Verzögerung eines bestimmten Fertigungsschritts kann es notwendig sein, alle nachfolgenden Schritte vorübergehend zu suspendieren (Operation *suspendieren*). Der Index des ersten suspendierten Durchlaufplanelements wird im Attribut *abSuspendiert* gespeichert. Die Neu-Einlastung erfolgt dann mittels der Operation *korrigieren*, beginnend mit dem ersten suspendierten Durchlaufplanelement.

Wir gehen davon aus, daß die Einlastung eines Durchlaufplanelements im laufenden Betrieb an einem Betriebsmittel nicht atomar ablaufen kann. Daher wird eine Operation *ist_eingelastet* zur Verfügung gestellt, mittels der eine erfolgreiche Einlastung eines Durchlaufplanelements beim Durchlaufplan gemeldet werden kann. Die Ausführung des nächsten Bearbeitungsschritts des — in unserem Szenario sequentiellen — Durchlaufplans wird durch Aufruf der Operation *nächsten_Schritt_ausführen* angestoßen. Hierzu kann ein Durchlaufplan-Objekt auf den Index des jeweils aktuellen Bearbeitungsschritts der Elementfolge im Attribut *aktuellerSchritt* zugreifen. Auch bei der Ausführung eines Bearbeitungsschritts gehen wir davon aus, daß dieser nicht atomar erfolgen kann. Daher benötigen wir wiederum eine zweite Operation (*ist_ausgeführt*), mit der beim Durchlaufplan die Beendigung des aktuellen Schritts gemeldet werden kann. Der Eingabeparameter enthält den OID des abgearbeiteten Durchlaufplanelements.

Die Beziehung zwischen Durchlaufplänen und einzelnen Durchlaufplanelementen einerseits und Betriebsmitteln und einzelnen Durchlaufplanelementen andererseits wird über Objekte des Listentyps *Durchlaufplanelementfolge* realisiert.

Betriebsmittel und Durchlaufpläne teilen sich Durchlaufplanelement-Objekte über (indirekt via Durchlaufplanelementfolge-Objekte laufende) Referenzen. Diese Modellierung erlaubt somit sowohl eine auftragsbezogene als auch eine betriebsmittelbezogene Sicht auf die Fertigung.

3.3 Beispielanwendungen

Als Beispielanwendungen betrachten wir die Überwachung und die Steuerung der Bearbeitung von Durchlaufplänen.

Fertigung überwachen Wesentlich für eine genaue Feinplanung, wie sie im Rahmen der Einlastung erfolgt, ist, daß genaue Rückmeldungen aus der Fertigung vorliegen [39]. Aus Datenbanksicht bedeutet dies, daß die Objekte, die die Fertigung modellieren, die reale Situation in der Fertigung möglichst genau widerspiegeln müssen. In unserem Szenario müssen bspw. die Werte der *von*- und *bis*-Attribute der Durchlaufplanelemente, die initial ja lediglich die geplanten Start- und Endtermine enthalten, ständig aktualisiert werden, um ein genaues Abbild der Auslastung der Betriebsmittel widerzugeben. Wir benötigen somit eine Anwendung, die im Dialog mit der Realwelt (z.B. über interaktive Eingabegeräte, Sensoren u.ä.) den jeweils aktuellen Stand der Fertigung ermittelt und die Datenbasis entsprechend aktualisiert.

Fertigung steuern Darüberhinaus wird eine zweite Anwendung benötigt, die die einzelnen Fertigungsschritte anstößt. Beim Erkennen kritischer Situationen, z.B. wenn die aktuellen Fertigungsdaten darauf hinweisen, daß der ursprünglich geplante Endtermin,

```
type Durchlaufplanelement is
   body [vorgang: Arbeitsgang;
         von: int;
         bis: int;
         gehoertZu: Durchlaufplan;
         eingelastet: Betriebsmittel; !! Betriebsmittel, auf dem vorgang ausgeführt wird
         ...]
   operations
      declare Durchlaufplanelement: Arbeitsgang, Durchlaufplan →void;
   implementation ...
end type Durchlaufplanelement;

type Durchlaufplanelementfolge is <Durchlaufplanelement>;

type Durchlaufplan is
   body [elemente: Durchlaufplanelementfolge;
         plan: Arbeitsplan;
         aktuellerSchritt: int; !! Index des aktuellen Durchlaufplanelements
         abSuspendiert: int; !! Index des ersten suspendierten Durchlaufplanelements
         ...]
   operations
      declare Durchlaufplan: Arbeitsplan, Auftrag →void;
      declare suspendieren: Durchlaufplanelement →void;
      declare korrigieren: →void;
      declare ist_eingelastet: Durchlaufplanelement, int →void;
      declare nächsten_Schritt_ausführen: →void;
      declare ist_ausgeführt: Durchlaufplanelement →void;
   implementation ...
end type Durchlaufplan;
```

Abbildung 3: Objekttypdefinitionen *Durchlaufplanelement*, *Durchlaufplanelementfolge* und *Durchlaufplanelement*

der für einen Durchlaufplan errechnet wurde, nicht eingehalten werden kann, leitet die Steuerung entsprechende Reaktionen ein, bspw. eine Umplanung.

3.4 Bewertung und Anforderungen

Die oben beschriebene objektorientierte Modellierung bietet erhebliche Vorteile gegenüber einer Modellierung mit dem relationalen Modell. Durch sie werden die Nachteile Segmentierung von Objekten der realen Welt in Relationen, Einführung künstlicher Schlüsselattribute, fehlendes Verhalten und der „impedance mismatch“ der mengenorientierten relationalen Schnittstelle zu Programmiersprachen vermieden (siehe z.B. [23]). Dennoch weist sie noch Unzulänglichkeiten auf, die sich aus dem komplexen Zusammenwirken der Objekte der realen Welt (Durchlaufpläne, Betriebsmittel, usw.) ergeben, und die die Einsatzfähigkeit heutiger objektorientierter Datenbanksysteme im Produktionsbereich fragwürdig erscheinen lassen.

Die festgestellten Unzulänglichkeiten sind dabei unabhängig von dem Einsatz von GOM für die Datenmodellierung; die getroffenen Aussagen lassen sich auf die meisten der heute verfügbaren objektorientierten Datenbanksysteme übertragen (z.B. O_2 [11], GemStone [5], Objectstore [26]).

Beherrschung der Komplexität Viele Abläufe einer Produktion sind sehr komplex und werden daher kooperativ von mehreren autonomen, nur lose gekoppelten Verarbeitungseinheiten abgewickelt. Dies sollte sich auch in der Informationswelt widerspiegeln, was einer Modellierung durch einen zentralen Kontrollfluß — eine typspezifische Operation oder eine Aufrufsequenz von ihnen — entgegensteht. Stattdessen sollten mehrere Kontrollflüsse geschaffen werden, die jeweils das Verhalten einer aktiven Instanz der Produktionswelt — bspw. eines Betriebsmittels, eines Durchlaufplans usw. — modellieren. Um den Zusammenhang zwischen den autonomen Instanzen der realen Welt und den Kontrollflüssen im Informationssystem explizit zu machen, sollte es möglich sein, direkt den Objekten im Datenbanksystem einen eigenen Kontrollfluß zuzuweisen [22]. Objekte werden dadurch zu autonomen, nebenläufigen Verarbeitungseinheiten im Datenbanksystem. Bspw. könnte Objekten des Typs *Betriebsmittel* jeweils ein eigener Kontrollfluß zugeordnet werden, der die Abarbeitung der an dem Betriebsmittel wartenden Durchlaufplanelemente steuert; dieser Kontrollfluß korrespondiert dann mit der Steuerung des Betriebsmittels in der realen Fertigung.

Objekteigene Kontrollflüsse bezeichnen wir in der Folge als das *aktive Verhalten* eines Objektes; Objekte, die zu aktivem Verhalten fähig sind, nennen wir *aktive Objekte*.

Reaktion auf Ereignisse Abläufe in der Produktion werden häufig durch Ereignisse von außen beeinflußt, die bei der Planung nicht immer vollständig berücksichtigt werden können. Bspw. können bereits während der Einplanung eines Durchlaufplans Betriebsmittel ausfallen, so daß direkt eine Umplanung erforderlich wird.

Für das Informationssytem bedeutet dies, daß die Kontrollflüsse, die die Abläufe der Produktionswelt modellieren, auf Ereignisse von außen reagieren müssen. Auf der Basis herkömmlicher Objektmodelle können Abläufe nur mit Hilfe synchroner Operationsaufrufe beschrieben werden. Dies führt bei der Bearbeitung lang andauernder Operationen aber zu einer Wartesituation für die Anwendung, die die Operation aufgerufen hat. In dieser Zeit kann nicht auf Ereignisse von außen reagiert werden. Stattdessen sollte es möglich sein, Operationen *asynchron* aufzurufen; der Aufrufer kann dann nach Absetzen des Operationsaufrufs direkt weiterarbeiten. Bspw. könnte die Einplanung eines Durchlaufplans asynchron abgesetzt werden; wird danach der aufrufenden Anwendung der Ausfall eines Betriebsmittels gemeldet, so kann sie direkt darauf reagieren.

Repräsentation der Verteilungsstruktur Die Abläufe in der Produktion finden in einer verteilten Umgebung statt. So erfolgt die Steuerung eines Betriebsmittels üblicherweise auf einem anderen Rechner als die Einlastung von Aufträgen. Die Verteilungsstruktur des Informationssystems sollte in dem Datenbanksystem repräsentiert werden, um Daten (Objekte) zu jedem Zeitpunkt an den Ort bringen zu können, an dem das Zentrum ihrer Aktivitäten liegt [24]. Hierzu ist es notwendig, Objekten einen Aufenthaltsort zuordnen zu können, an dem sie sich für einen bestimmten Zeitraum aufhalten.

Die Verteilung und Migration von Objekten sollte anwendungsspezifisch erfolgen,

um z.B. beim Zugriff auf Objekte (Netz)-Kommunikationskosten einzusparen. Als Beispiel betrachten wir die sequentielle Bearbeitung eines Bauteils auf einer Reihe von Betriebsmitteln. In heutigen Fertigungskonzepten wird häufig auf Laufkarten, die das Bauteil begleiten, der Fortschritt bei der Bearbeitung vermerkt [39]. In unserem Beispiel-Datenbankschema könnte dies durch die Migration von Objekten des Typs Durchlaufplan zu den jeweiligen Prozeßrechnern modelliert werden, die den aktuellen Bearbeitungsschritt steuern.

Trotzdem sollten Anwendungen, die selten oder ohne hohe Effizienzanforderungen auf Durchlaufplanobjekte zugreifen wollen (z.B. Statistik-Routinen) von der Migration der Objekte zu den Prozeßrechnern nicht berührt werden. Das Datenbanksystem muß daher dafür sorgen, daß Objekte unabhängig von ihrem aktuellen Aufenthaltsort referenziert werden können.

Objekte, die durch Benutzereinfluß in einem Rechnernetz migrieren können, bezeichnen wir als *mobile Objekte*.

Zusammenfassend kann man sagen, daß objektorientierte Datenbanksysteme im Vergleich zu relationalen Systemen eine wesentliche höhere Funktionalität bei der Modellierung einer komplexen Miniwelt bieten. Ein großer Nachteil heutiger Objektmodelle liegt jedoch darin, daß die Kooperation autonomer Verarbeitungseinheiten sowie die Verteilungsstruktur der realen Welt nur unzureichend im Datenbanksystem erfaßt werden. Im nachfolgenden schlagen wir daher Erweiterungen des Objektmodells GOM um aktive und mobile Objekte vor, die einen ersten Schritt zur Lösung dieses Problems darstellen.

4 dGOM — Aktive und mobile Objekte in GOM

4.1 Aktive Objekte

Aktive Objekte sind Instanzen eines „aktiven Objekttyps“. Der Definitionsrahmen für aktive Objekttypen wird durch Voranstellen des Schlüsselwortes **active** gekennzeichnet. Aktive Objekte werden genauso wie passive Objekte durch einen systemweit eindeutigen OID identifiziert; dieser wird wie bei passiven Objekten zur Referenzierung des Objektes verwendet.

Aktive Objekte verfügen über einen eigenen Kontrollfluß. Nachrichten zwischen aktiven Objekten können — entsprechend der Forderung aus Abschnitt 3.4 — sowohl synchron als auch asynchron ausgetauscht werden. Syntaktisch entspricht die synchrone Kommunikation dem herkömmlichen Prozeduraufruf, der nun als *Remote Procedure Call* interpretiert werden muß. Ein asynchroner Operationsaufruf auf einem aktiven Objekt erfolgt durch Voranstellen des Schlüsselwortes **send**. Bspw. bewirkt folgende Anweisung, daß das Betriebsmittel *bm_4711* mit dem Durchlaufplanelement *dlp_elem_1* belegt wird; die Dauer der Belegung ist 30 Minuten:

```
send bm_4711.belegen(dlp_elem_1, 30);
```

Das Schlüsselwort **send** zeigt dabei an, daß die Operation *belegen* asynchron aufgerufen wird; der Aufrufer kann daher nach dem Operationsaufruf direkt weiterarbeiten, ohne auf das Ende des Belegungsvorgangs warten zu müssen.

Im Gegensatz zu anderen Ansätzen (z.B. [17]) kann in dGOM jede Operation, ob sie ein Ergebnis zurückliefert oder nicht, sowohl synchron als auch asynchron aufgerufen werden.

Dies ist sinnvoll, da es auch bei Operationen, die kein Ergebnis liefern, aus objektinternen Synchronisationsgründen notwendig sein kann, die Abarbeitung der Operation auf dem entfernten Objekt abzuwarten. Wird eine Operation, die ein Ergebnis liefert, asynchron aufgerufen, so wird das Ergebnis ignoriert (siehe hierzu auch Abschnitt 4.4.2).

Nachrichten, d.h. entfernte Operationsaufrufe, an ein aktives Objekt werden von diesem in einer fest vorgebenen (FIFO) Strategie abgearbeitet. Zur Pufferung von Nachrichten verfügt jedes aktive Objekt intern über eine *Mailbox*; diese ist dem Anwendungsprogrammierer nicht zugänglich. Die FIFO–Strategie wurde gewählt, um sicherzustellen, daß jeder Aufruf irgendwann abgearbeitet wird. Da der Nutzen der Verwendung alternativer Strategien noch unbekannt ist, wäre es unserer Ansicht nach unnötiger Aufwand, dem Benutzer die Auswahl zwischen mehreren Alternativen zu ermöglichen. Durch den praktischen Einsatz des jetzigen Systems soll evaluiert werden, ob eine derartige Modellerweiterung notwendig ist.

Aktive Objektypen können als Subtypen passiver Objekttypen definiert werden, aber nicht umgekehrt. Bspw. könnte man eine folgende Typ-Hierarchie aufstellen:

type Betriebsmittel **is** ...
    !! abstrakte Definition eines Betriebsmittels.

type Bohrer **supertype** Betriebsmittel **is** ...
    !! ein passives Betriebsmittel

active type Fräsmaschine **supertype** Betriebsmittel **is** ...
    !! ein aktives Betriebsmittel

Der Typ Betriebsmittel ist dabei wie in Abb. 2 definiert. Einen Bohrer würde man ebenfalls als passiven Typ vereinbaren, da er auch in der realen Welt kein selbständiges Verhalten zeigt, und physisch zu einem Zeitpunkt immer nur von einer Maschine verwendet wird. Auf der Objektebene schlägt sich dies darin nieder, daß auf ein Bohrer-Objekt nur jeweils über ein Betriebsmittel-Objekt zugegriffen wird.[3] Fräsmaschinen bieten sich dagegen als aktive Objekte an, da ihre Steuerung möglicherweise mehrere Motoren regeln (z.B. je Achse einen), die wiederum Rückmeldungen an die Steuerung abgeben und die trotzdem während ihrer Tätigkeit noch von außen beeinflußbar bleiben müssen (z.B. „Notaus“).

Die Eigenschaft, aktives Verhalten zu besitzen, wird automatisch an alle Untertypen vererbt, um die Substituierbarkeit von Instanzen des Subtyps zu ihren Obertyp — wie sie in [25] für GOM definiert wurde — zu gewährleisten. Damit ist es ausgeschlossen, daß passive Objekttypen Subtypen aktiver Objekttypen sind.

Das Empfänger-Objekt einer mit **send** invokierten Operation kann prinzipiell auch ein passives Objekt sein. In diesem Fall hat **send** jedoch keine Auswirkung — die Operation wird synchron abgearbeitet. Da jedoch aktive Typen Subtypen (passiver) Typen sein können, und aktive Objekte somit für passive Objekte substituiert werden können, wird hierdurch gewährleistet, daß die Asynchronität ausgenutzt wird, falls ein aktives Objekt adressiert wird. Die folgende Sequenz von Anweisungen zeigt ein Beispiel hierfür. Es bezieht sich auf obige Typdefinitionen von Betriebsmittel, Bohrer und Fräsmaschine.

persistent var bemi: Betriebsmittel;
persistent var bohrer: Bohrer;
persistent var fräser: Fräsmaschine;

[3] Dies wird häufig als ein Kriterium zur Entscheidung, ob Objekte aktiv oder passiv modelliert werden sollen, angeführt (z.B. [6]).

```
...
bemi:= bohrer;
send bemi.belegen(...);      !! Operation wird synchron abgearbeitet.
bemi:= fräser;
send bemi.belegen(...);      !! Operation wird asynchron abgearbeitet.
```

Über die Referenz *bemi* wird zunächst ein passives Objekt (*bohrer*) referenziert; die Operation *belegen* wird daher synchron ausgeführt. Bei dem zweiten Operationsaufruf verweist *bemi* dagegen auf eine Instanz des aktiven Typs *Fräsmaschine*; die Operation *belegen* wird in diesem Fall asynchron abgearbeitet.

Prozesse und Transaktionen in dGOM Es stellt sich nun die Frage, was aus den ursprünglich zur Beschreibung von Kontrollflüssen in GOM vorgesehenen Konzepten „Prozeß“ und „Transaktion“ (siehe Abschnitt 3.1.4) wird.

Prozesse bleiben dem Programmierer weiterhin erhalten. Sie lassen sich als eine spezielle Art von aktiven Objekten auffassen: ein Prozeß ist ein aktives Objekt, auf dem genau eine Operation — Konstruktor und Initialisator (siehe Abschnitt 3.1.2) — aufgerufen werden kann. Am Ende dieser Operation wird das „Prozeß-Objekt“ jedoch gleich wieder gelöscht. Der in Abschnitt 3.1.4 angeführte Prozeß *Auftrag_einlasten* läßt sich somit wie folgt interpretieren:

```
active type Auftrag_einlasten is
   operations
      declare Auftrag_einlasten: →void;
   implementation
      define Auftrag_einlasten is
         begin
            !! Prozeßrumpf-Definition
            self.delete;
         end define Auftrag_einlasten;
end active type Auftrag_einlasten;
```

Eine Invokation des Prozesses, die durch Eingabe des Prozeßnamens von einer Betriebssystem-Shell heraus erfolgt, entspricht dann folgendem Programmausschnitt:

```
var pid: Auftrag_einlasten;
...
pid.create;
```

Prozesse sollen einem Programmierer vor allem dazu dienen, eine initiale Population von aktiven Objekten zu erzeugen.

Die Unzulänglichkeiten von Transaktionen in technischen Anwendungen aufgrund ihrer häufig langen Dauer sowie ihrer ggf. vorhandenen Auswirkungen in der realen Welt sind schon seit längerem bekannt und Alternativen zu ihnen Gegenstand der Forschung (z.B. [21, 14, 33]). Die Beziehung zwischen den in der Literatur vorgeschlagenen Lösungen (eine Übersicht gibt z.B. [13]) und aktiven Objekten mit eigenem Kontrollfluß und ggf. asynchroner Kommunikation sind jedoch heute noch unklar.

In dGOM wurde daher auf die Transaktionssemantik der Prozesse verzichtet. Programmierern steht somit auch die vordefinierte Operation *commit* für Zwischensicherungspunkte nicht mehr zur Verfügung.

Als Folge davon ist daher nur noch eine vorwärts gerichtete Recovery [20] durch auf Objekttypen benutzerdefinierte Operationen, die der Kompensation von Fehlern dienen, möglich[4].

Eine sehr rudimentäre Form der Synchronisation zwischen aktiven Objekten wird durch die *RPC*-Semantik der Operationsaufrufe erzielt. Ferner wird bzgl. des Zugriffs auf die ggf. persistente Zustandsinformation von aktiven Objekten — die auch Referenzen auf passive persistente Objekte enthalten kann — der gegenseitige Ausschluß garantiert[5].

Auf das Problem der Konsistenzerhaltung, für das Transaktionen speziell in Gegenwart von Nebenläufigkeit und im Fehlerfall eine Lösung darstellen, die uns in dGOM nun nicht mehr zur Verfügung steht, wird im allgemeinen noch in Abschnitt 4.4 eingegangen.

4.2 Mobile Objekte

Die Grundidee ist, daß Objekte als Einheit von Daten und Operationen flexibel auf unterschiedliche Rechner verteilt werden und dabei gleichzeitig mittels Kommunikationsmechanismen im Rahmen einer gemeinsamen Anwendung kooperieren. Nachrichten können an Objekte unabhängig von ihrem Aufenthaltsort gesendet werden; Anwendungen können daher vollständig von der Verteilungsstruktur der Objekte abstrahieren (*lokationsunabhängige Operationsaufrufe*). Dennoch ist es möglich, bei Bedarf explizit auf die Objektverteilung Einfluß zu nehmen, um z.B. kommunizierende Objekte auf einem gemeinsamen Rechner zu plazieren oder parallele Verarbeitungsvorgänge auf verschiedene Rechner zu verteilen.

In dGOM sind alle Objekte (aktive wie passive) mobil; die Mobilität von Objekten muß daher nicht explizit auf Typebene spezifiziert werden. dGOM-Objekte können dynamisch ihren Aufenthaltsort durch Migration wechseln. In dGOM ist der „Aufenthaltsort" eines Objektes wiederum ein Objekt, d.h., Objekte können als „logische" Orte für andere Objekte dienen. Jedes Objekt kann höchstens einen Aufenthaltsort besitzen; es ist aber durchaus möglich — und, wie wir sehen werden, auch notwendig —, daß einem Objekt kein Aufenthaltsort zugewiesen ist. Der (logische) Ort eines Objektes wird durch die *migrate*-Operation festgelegt. Bspw. bewirkt folgende Anweisung, daß das Durchlaufplanelement-Objekt *dlp_elem_1* zu dem Betriebsmittel-Objekt *bemi_4711* migriert:

```
dlp_elem_1.migrate(bemi_4711);
```

Hierdurch wird dem Datenbanksystem bekannt gegeben, daß der (logische) Aufenthaltsort des Objektes *dlp_elem_1* das Objekt *bemi_4711* ist. Als Folge befindet sich das Objekt *dlp_elem_1* auch immer auf dem gleichen Rechnerknoten (diesen könnte man als physischen Aufenthaltsort bezeichnen), wie das Objekt *bemi_4711*. Migriert *bemi_4711* zu einem anderen Ort, dann migrieren automatisch alle Objekte, die *bemi_4711* als Aufenthaltsort besitzen, mit. Der Aufenthaltsort eines Objektes kann durch die Operation *locate* abgefragt werden. So liefert bspw. der Ausdruck *dlp_elem_1.locate* als Ergebnis den OID des Objektes, an dem sich *dlp_elem_1* befindet (in obigem Beispiel ist dies *bemi_4711*).

Durch die Aufrufe der Operation *migrate* wird eine Beziehung zwischen Objekten etabliert, die wir als *location-of*-Beziehung bezeichnen wollen [24]. Die location-of-Beziehung muß hierarchisch sein, d.h., ein Objekt darf höchstens einen Ort besitzen, und es dürfen keine Zyklen in der Beziehung bestehen. **Abb. 4** zeigt ein Beispiel für die location-of-Beziehung.

[4] Plattenfehler (Crash-Recovery) werden in den aktuellen Arbeiten nicht berücksichtigt.

[5] Dies wird bei der aktuellen Implementierung durch die interne Verwendung von Transaktionen an der Schnittstelle des Speicherverwaltungssystems (in diesem Fall der EXODUS Storage Manager) realisiert.

```
var bemi_4711, bemi_4712: Betriebsmittel;
    dlp_elem_1: Durchlaufplanelement;
    bohrer_8mm: Bohrer;
    Prozessrechner_XYZ: Location;

...;
dlp_elem_1.migrate(bemi_4711);
bohrer_8mm.migrate(bemi_4711);
bemi_4711.migrate(Prozessrechner_XYZ);
bemi_4712.migrate(Prozessrechner_XYZ);
```

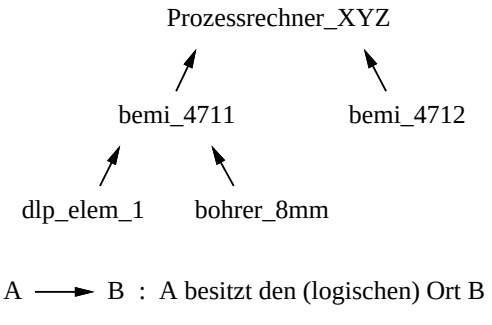

Abbildung 4: Die location-of-Beziehung zwischen Objekten

Um nicht nur den logischen, sondern auch den physischen Ort festlegen zu können, bietet dGOM den Objekttyp *Location* an; Instanzen dieses Typs repräsentieren die Knoten des Rechnernetzes, auf dem die Datenbasis arbeitet. In Abb. 4 haben wir bspw. die Variable *Prozessrechner_XYZ* vom Typ *Location* eingeführt, die den Prozeßrechner repräsentieren soll, der die beiden Betriebsmittel *bemi_4711* und *bemi_4712* steuert. Aus diesem Grund ist es sinnvoll, diese beiden Betriebsmittel-Objekte zu dem Objekt *Prozessrechner_XYZ* migrieren zu lassen. Objekte des Typs *Location* befinden sich immer (physisch) an dem Rechnerknoten, den sie repräsentieren; sie dürfen daher nicht zu anderen Objekten migrieren und bilden immer eine Wurzel in der location-of-Beziehung.[6]

Initial besitzen Objekte keinen logischen Aufenthaltsort. Dieser muß ihnen durch den Anwender explizit zugewiesen werden. Damit dies nicht „vergessen“ wird, bietet es sich an, den Initialisator eines benutzerdefinierten Objekttyps zu überschreiben. Nehmen wir an, wir wollten jedem Objekt eines Typs X als initialen Aufenthaltsort den Ort des erzeugenden Objekts[7] zuordnen. Der Initialisator des Typs X sähe dann wie folgt aus:

```
type X is ...
  operations
    declare X: ANY →void; ...
  implementation
    define X (loc) is self.migrate(loc); ...
end type X;
```

Die entsprechende Anweisung in der Operationsdefinition des erzeugenden Objekts könnte dann wie folgt aussehen (*einVariable* sei vom Typ X):

```
define erzeugendeOperation is
  ...
  eineVariable.create(self.locate);
```

4.3 Anwendung von dGOM

In diesem Abschnitt wollen wir zeigen, wie das Beispielszenario mit Hilfe von dGOM modelliert werden kann. In Abschnitt 4.4 vergleichen wir die Modellierung in dGOM mit

[6] Um dies durch das System zuzusichern, wurde die Operation *migrate* auf dem Typ *location* derart überschrieben (verfeinert), daß sie bei einem Aufruf einen Laufzeitfehler produziert.

[7] Das erzeugende Objekt ist das Objekt, auf dem eine Operation ausgeführt wird, in deren Implementierung die Erzeugung erfolgt.

der „klassischen“ Modellierung in GOM und diskutieren die Vorteile und Grenzen des Ansatzes von dGOM. Wir konzentrieren uns im folgenden auf die Steuerung und Überwachung der Fertigung. Konkret wollen wir die Situation behandeln, daß eine Umplanung der Betriebsmittel im laufenden Betrieb aufgrund einer Verzögerung während eines Bearbeitungsschrittes vorgenommen werden muß.

Bei der Umplanung sind vielfältige Konsistenzbedingungen zu berücksichtigen. Bspw. dürfen Betriebsmittel in einem Zeitraum nicht mehrfach belegt werden, und die einzelnen Schritte eines Durchlaufsplans dürfen sich nicht überlappen,[8] Weiterhin muß die Umplanung während des laufenden Betriebs erfolgen; während der Umplanung können somit Ereignisse eintreten, die das Planungsergebnis beeinflussen (Ausfall von Betriebsmitteln, Einlastung eines Auftrags mit höherer Priorität, usw.).

Wir wollen nun zeigen, wie die Mechanismen von dGOM dazu verwendet werden können, um diese Sachverhalte im Datenbanksystem zu erfassen. Die Objekttypen Betriebsmittel und Durchlaufplan werden als aktive Objekttypen definiert; hierzu müssen lediglich die in Abb. 2 und Abb. 3 vereinbarten Typdefinitionen um das Schlüsselwort *active* erweitert werden:

active type Betriebsmittel ... !! Definition wie in Abb. 2
active type Durchlaufplan ... !! Definition wie in Abb. 3

Um die Einhaltung der o.g. Konsistenzbedingungen zu gewährleisten, halten wir uns bei der Modellierung an folgendes „Protokoll“:

(1) Die Betriebsmittel sind dafür verantwortlich, daß die bei ihnen eingeplanten Durchlaufplanelemente konfliktfrei abgearbeitet werden können, ohne daß mehrere Durchlaufplanelemente zum gleichen Zeitpunkt eingeplant sein.

(2) Die Durchlaufpläne sind dafür verantwortlich, daß die einzelnen Teilelemente eines Durchlaufplans in der vorgesehenen Reihenfolge abgearbeitet werden: Jeder dieser Durchlaufplanelemente muß an genau einem Betriebsmittel eingeplant sein und der Anfangstermin eines Elements darf den Endetermin seiner Vorgänger nicht überschreiten.

Den von den Betriebsmitteln einzuhaltende Teil des „Vertrags“ (1) erfüllen wir dadurch, daß die *bis* Attribute der Durchlaufplanelemente nur von Betriebsmitteln verändert werden können, daß ein Durchlaufplanelement zu jedem Zeitpunkt nur von einem einzigen Betriebsmittel referenziert wird, und daß die Operation *belegen* so implementiert wird, daß das einzulastende Durchlaufplanelement sich zeitlich mit keinem anderen bereits für dieses Betriebsmittel eingelasteten Durchlaufplanelement überschneidet.

Der zweite Teil des Vertrags (2) wird auf ähnliche Weise erfüllt: Wir garantieren, daß ein Durchlaufplanelement höchstens in einem einzigen Durchlaufplan enthalten ist; die Reihenfolge der Durchlaufplanelemente in einem Durchlaufplan wird ausschließlich von *Durchlaufplan*-Objekten verändert; das Durchlaufplanobjekt sorgt dafür, daß bei Einlastung des Elements $I + 1$ eines Plans als frühester Starttermin der (voraussichtliche) Endzeitpunkt des Elements I genommen wird.

[8] Wir gehen hier von einer strikt sequentiellen Abarbeitung eines Durchlaufplans aus. Komplexere Bearbeitungsstrategien wie Lossplittung oder parallele Teilfertigung betrachten wir nicht.

Zunächst wollen wir die Abarbeitung eines Durchlaufplans darstellen. Diese wird durch den Aufruf der Operation *nächsten_Schritt_ausführen* von außen angestoßen.[9] Zur Vermeidung von Kommunikationskosten migriert das Durchlaufplan-Objekt samt seinen Elementen zu dem Betriebsmittel-Objekt, auf dem das aktuelle Element des Durchlaufplans abgearbeitet wird. Das Durchlaufplan-Objekt ruft asynchron die Operation *bearbeiten* auf dem Betriebsmittel-Objekt auf; asynchron deshalb, um ggf. auch während der Ausführung des Bearbeitungsschritts auf Korrektur-Nachrichten reagieren zu können. Ist die Fertigmeldung seitens des Betriebmittels erfolgt (Operation *ist_ausgeführt*) wird der nächste Schritt ausgeführt (Aufruf der Operation *nächsten_Schritt_ausführen*). Einmal angestoßen, wechseln sich Durchlaufplan-Objekte und Betriebsmittel-Objekte während der Abarbeitung eines Durchlaufplans in ihren Aktionen ab. Alle beteiligten Objekte sind jedoch während der meisten Zeit der Abarbeitung bereit, Nachrichten zu empfangen, um ggf. auf neue Situationen zu reagieren.

Nun wollen wir diese heile Welt etwas in Unordnung bringen und annehmen, das — bspw. aufgrund einer Verzögerung an einer Maschine — die *von*-Zeit des Durchlaufplanelements, das als nächstes an der Reihe ist, erreicht wird, ohne daß eine Fertigmeldung des aktuellen Schritts erfolgt. Da aktive Objekte in dGOM nur auf Nachrichten von außen reagieren, muß die Entdeckung einer solchen Situation ebenfalls von außen erfolgen. Hierzu wird die Operation *suspendieren* auf dem Durchlaufplan-Objekt aufgerufen; das erste nicht mehr fristgerecht zu startende Durchlaufplanelement-Objekt wird als Parameter übergeben.

Eine Korrektur eines solchen Fehlerfalls erfolgt dadurch, daß alle noch ausstehenden Durchlaufplanelemente auf den für sie vorgesehenen Betriebsmitteln vorübergehend freigegegen werden (*freigeben* auf Betriebsmittel). Nach der Beendigung des verspäteten Durchlaufplanelements werden sämtliche noch ausstehenden Durchlaufplanelemente neu eingelastet. Hierbei wird die Operation *einlasten* synchron aufgerufen, da das Ergebnis der Einlastung — der Endtermin des Schritts — als Starttermin des nächsten Schritts benötigt wird.

Ähnlich kann eine Kompensation von Fehlerfällen auch lokal bei Betriebsmitteln vorgenommen werden. Ein Betriebsmittel, das erkennt, daß es den vorgegebenen Endtermin nicht einhalten kann, kann z.B. versuchen, bestimmte eingelastete Elemente auf andere geeignete Betriebsmittel zu verteilen. So können unter Umständen Folgeeffekte einer Verzögerung verhindert werden.

4.4 Bewertung der Erweiterungen

4.4.1 Vorteile

Aktives Verhalten von Objekten Abläufe der realen Welt werden direkt über das aktive Verhalten von Objekten modelliert. Anwendungsspezifische Konsistenzbedingungen werden objektlokal überprüft — mit den in Abschnitt 4.4.2 erörterten Einschränkungen —; den Anstoß für die Überprüfung kann dabei das aktive Objekt selbst geben. Diese Konsistenzbedingungen können insbesondere auch Aspekte der nebenläufigen Benutzung der Objekte berücksichtigen [37, 10].

Es handelt sich bei dGOM um einen heterogenen Ansatz [31], d.h. es werden sowohl

[9] Die Operationen der Objekttypen Betriebsmittel, Durchlaufplan und Durchlaufplanelement sind in den Abbildungen 2 und 3 definiert.

aktive als auch passive Objekte unterstützt. Dieser Ansatz weist gegenüber dem orthogonalen und homogenen Ansatz folgende Vorteile auf:

- Im orthogonalen Ansatz erfolgt die nebenläufige Ausführung von Kontrollflüssen unabhängig von den Objekten. Das aktive Verhalten wird dabei losgelöst von den Objekten definiert; der Zusammenhang zwischen den Objekten und ihrem Verhalten wird dadurch nicht explizit im Modell repräsentiert. Beispiele für Sprachen, die den orthogonalen Ansatz verfolgen, sind Smalltalk 80 [19], Emerald [4] und Concurrent C++ [17]. In diese Gruppe lassen sich ferner auch alle Vorschläge für aktive objektorientierter Datenbanksysteme einordnen, z.B. ODE [18], Sentinel [7], ADAM [12] und SAMOS [16].

- In homogenen Ansätzen gibt es nur aktive Objekte. Ein Beispiel hierfür ist ABCL/1 [40]. Dies kann zu unnötigen zusätzlichen Kosten führen, da jedes Objekt Code für die Synchronisation der Nebenläufigkeit bereitstellen muß — unabhängig davon, ob dieser letztendlich benötigt wird oder nicht.

  Ebenfalls als homogen klassifizieren Papathomas et al. ihr Objektmodell Hybrid [32]. Zudem wird für Hybrid der Anspruch erhoben, daß die vorerwähnten zusätzlichen Kosten vermieden werden. Hierfür wird jedoch Heterogenität an anderer Stelle wieder eingeführt: bei Attributen, die auf andere Objekte verweisen, wird zwischen lokalen und geteilten Referenzen unterschieden. Je nachdem entlang welcher Art von Objektreferenz auf ein Objekt zugegriffen wird, wird dieser Zugriff synchronisiert (geteilt) oder nicht (lokal). Es ist nun leicht einzusehen, daß das Nebeneinander von lokalen und geteilten Referenzen auf möglicherweise ein- und dasselbe Objekt Konsistenzprobleme aufwirft. Um diese zu vermeiden, muß sichergestellt werden, daß zum Zeitpunkt lokaler Zugriffe auf ein Objekt keine weiteren Zugriffe von anderen Objekten zu gleicher Zeit erfolgen können. Dies heißt jedoch in Hybrid nichts anderes, als daß doch zwischen aktiven und passiven Objekten unterschieden wird, wobei diese Unterscheidung allerdings nicht auf Typebene, sondern für einzelne Instanzen — und zwar über die Art, wie sie referenziert werden — erfolgt. Es ist sicherlich bedenkenswert, aktives Verhalten nicht auf Typebene zu beschreiben, sondern dieses für einzelne Instanzen vorzusehen. Die in Hybrid vorgeschlagene Lösung widerspricht jedoch in manchen Teilen der Intention der Objektkapselung, da Objektimplementierungen abhängig werden von der Implementierung derjenigen Typen, die sie benutzen [32].

Synchrone und asynchrone entfernte Operationsaufrufe Durch Operationsaufrufe können aktive Objekte miteinander kommunizieren und so gegenseitig Einfluß aufeinander ausüben. Durch asynchrone Kommunikation bleibt diese Einflußmöglichkeit auch bei komplexen, längere Zeit in Anspruch nehmenden Operationen erhalten. Auf diese Art ist eine sehr flexible Reaktion auf Abweichungen vom geplanten Ablauf möglich.

Logische Verteilung von Objekten Durch die Mobilität von Objekten kann Einfluß auf ihre Verteilung in einem Rechnernetz genommen werden. Die Verteilung in dGOM ist dabei weitgehend unabhängig von konkreten Rechnerknoten; sie stützt sich dagegen auf logische Beziehungen zwischen Objekten ab. Hierbei kann die Semantik der Anwendungs-

welt für die Minimierung der Kommunikationskosten ausgenutzt werden. Dieser Vorteil wird ohne Aufgabe lokalisationsunabhängiger Operationsaufrufe auf Objekten erzielt.

Einfachheit der Erweiterungen Die Erweiterungen wurden sehr einfach gehalten, so daß ein Anwendungsprogrammierer bei der Erstellung von Modellen beinahe in gewohnter (sequentieller) objektorientierter Weise programmieren kann. Dies hat uns auch ermöglicht, aktive Objekttypen homogen in das Typsystem von GOM [25] einzufügen.

4.4.2 Grenzen des Ansatzes

Spezifikation aktiven Verhaltens Nachrichten werden — ähnlich der ADA-Eingänge [27] — unbedingt akzeptiert und nach einer fest vorgebenen Strategie von dem Objekt abgearbeitet. Dieser einfache Mechanismus ist aber nur dann zufriedenstellend anwendbar, wenn die Entscheidung, wie das Objekt auf eine Nachricht reagiert, weitgehend unabhängig von der Nachrichtenhistorie ist, da diese explizit in Form von Zustandsinformation von dem Objekt verwaltet werden muß. Diese Verwaltung wiederum muß Teil der Implementierung der jeweiligen Operationen sein sein, was diese unnötig aufbläht. Insgesamt führt eine solche Vorgehensweise zudem auch zu ineffizienten Lösungen.

Es bietet sich daher an, den Empfang von Nachrichten an Bedingungen zu knüpfen. Hierzu werden in der Literatur drei prinzipiell verschiedene, sich jedoch nicht gegenseitig ausschließende Möglichkeiten vorgeschlagen:

1. Explizites programmgesteuertes Empfangen von Nachrichten (z.B. über eine *accept*-Anweisung, wie sie z.B. in Concurrent C++ [17] angeboten wird).

2. Bedingungen über den Zustand des Objekts (z.B. ADA [27], SR [1], Autonome Objekte [22]) und die Nachrichtenhistorie [30].

3. Interpretation von Meta-Information zur Laufzeit, was neuerdings häufig mit *Reflexion* bezeichnet wird z.B. ABCL/R [40].

Ferner kann einem Programmierer expliziter Einfluß auf den Kontrollfluß des Objekts gegeben werden. Eine Möglichkeit hierfür besteht in der Definition eines Objekttyps *PROCESS* mit einer Operation *live*, in der das standardmäßige aktive Verhalten von Objekten spezifiziert ist, z.B. die FIFO-Strategie der Abarbeitung von Nachrichten. Von diesem Typ erben alle aktiven Objekttypen. Die *live*-Operation kann in Subtypen überschrieben werden [6]. Voraussetzung für eine solche Lösung ist die Möglichkeit der Mehrfach-Vererbung.

Ein weiteres Problem stellt der Umgang mit Situationen dar, in denen ein Objekt auf eine bestimmte Nachricht wartet, diese aber nicht eintrifft, wodurch die Konsistenz des Objekte gefährdet wird. In unserem Beispiel sind wir auf eine solche Situation bei der Entdeckung von Verzögerungen eines Betriebsmittels gestoßen. Durch die Möglichkeit, *Zeitereignisse* zu spezifzieren, wird dieses Problem gelöst. Vorschläge finden sich z.B. in [8, 15].

Soll in dGOM eine Operation, die ein Ergebnis liefert, asynchron aufgerufen werden, so muß dieses Ergebnis durch das Server-Objekt durch eine weitere Nachricht an den Klienten übermittelt werden, wobei die Kontextinformation (z.B. auf welche Nachricht sich die Antwort bezieht) explizit verwaltet werden muß (z.B. durch Parameter und/oder Zustandsinformation). Eine Möglichkeit, die zwischen den Extremlösungen des

voll-synchronen Operationsaufruf und der Kommunikation über zwei gleichwertige asynchrone Nachrichten liegt, besteht in der datengetriebene Synchronisation von Anfrage und Antwort (*wait-by-necessity*) [6], bei der Objekte Nachrichten versenden können und danach weiterarbeiten, ohne auf die Antwort zu warten. Erst wenn auf das Ergebnis zugegriffen wird (z.B. auf ein Attribut oder eine Operation), wird das Sender-Objekt blockiert (z.B. *futures* in ABCL/1 [40]).

Objektübergreifende Konsistenzbedingungen Die bisher vorgestellten Konzepte zur Beschreibung aktiven Verhaltens (sowohl die von dGOM als auch die im letzten Paragraphen angeführten weitergehenden Vorschläge aus der Literatur) unterstützen nur objektlokale Konsistenzbedingungen. Unter der Voraussetzung, daß wir Objekte als lose gekoppelte, autonome Einheiten begreifen, können objektübergreifende Konsistenzbedingungen nur in Bezug auf das Kommunikationsverhalten der Objekte formuliert werden, bspw. durch das Anhängen von Kontextinformation an Nachrichten (Sender der Nachricht, Kommunikationshistorie, usw.). Das Empfängerobjekt kann dann in Abhängigkeit des Kontextes einer Nachricht entscheiden, ob Konsistenzbedingungen verletzt sind. In unserem Beispiel stellt das zeitliche Überlappungsverbot von Durchlaufplanelementen eine objektübergreifende Konsistenzbedingung dar.

Zur Durchsetzung dieser Konsistenzbedingung haben wir ein Protokoll für die Kommunikation zwischen Durchlaufplan- und Betriebsmittel-Objekten aufgestellt (nur Betriebsmittel dürfen die Anfangs- und Endzeit eines Elements verändern; Durchlaufplanelemente geben den frühesten Anfangstermin entsprechend der Einlastung des Vorgängerelementes vor). Das Protokoll ist in den Operationen der beiden Typen kodiert.

Diese „Lösung“ zur Durchsetzung objektübergreifender Konsistenzbedingungen besitzt jedoch folgenden Nachteil: Aktive Objekte sollen als lose gekoppelte, autonome Einheiten kooperieren, ohne daß die Art und die Partner von Kommunikationsvorgängen a priori bekannt sein muß. Die Formulierung objektübergreifender Konsistenzbedingungen in der obigen Form erfordert es aber, daß sämtliche Kommunikationskontexte eines Objektes a priori bekannt sind, damit die dazugehörigen Konsistenzbedingungen in den typ-assoziierten Operationen implementiert werden können. Hierdurch kann sowohl die Implementierung als auch die Schnittstelle eines Typs von der Benutzung seiner Instanzen abhängig werden. Ändert sich die Benutzung der Objekte eines Typs, so erfordert dies die Erweiterung seiner Implementierung um zusätzliche Konsistenzbedingungen. In unserem Szenario würde sich bspw. die Benutzung von Betriebsmittel-Objekten durch die Einführung von Leitstand-Objekten ändern (ein Leitstand überwacht die Einlastung von Durchlaufplanelementen auf einer Gruppe von Betriebsmitteln). Um seine Kontrollfunktion auszuüben, muß der Leitstand die Kommunikation zwischen Betriebsmittel und Durchlaufplan überwachen; hierzu ist aber die Anpassung der Operationen des Typs Betriebsmittel erforderlich.

Wir benötigen also zusätzliche Mittel, um objektübergreifender Konsistenzbedingungen zu sichern. Diese sollen jedoch weder eine globale Sicht auf die Objektwelt voraussetzen, wie dies bei heutigen Datenbanksystemen der Fall ist, noch uns nötigen, die Implementierung der beteiligten Typen abändern zu müssen.

Erste Ansätze in diese Richtung stellen [28] und [30] dar. Beide bieten auf sehr unterschiedliche Art eine Möglichkeit an, das Kommunikationsverhalten von Objekten in bestimmten Kontexten separat zu spezifizieren. Während die Betonung von [28] auf der Spezifikation von Kommunikationsregeln auf konzeptueller Ebene liegt, bietet [30] eine

Möglichkeit, Kommunikationsprotokolle für Objekte — in Analogie zu Telekommunikationprotokollen in verteilten Systemen — in eine Objektbank mitaufzunehmen. Eine detailliertere Vorstellung dieser Ansätze würde über den Rahmen dieses Beitrags hinausgehen.

5 Implementierung

5.1 Aktive Objekte

Die vorgestellten Konzepte werden zur Zeit in laufenden Arbeiten validiert, indem der bereits vorhandene Prototyp von GOM um Verteilungsmechanismen erweitert wird.

Besondere Aufmerksamkeit erfordert die Integration der Persistenz und Aktivität. Passive Objekte existieren bis zum Zeitpunkt ihres Löschens als Strukturen auf dem Hintergrundspeicher. Während ihrer Benutzung werden im Kontext eines Prozesses Laufzeitabbilder erzeugt, auf die mittels Kontrollflüssen zugegriffen werden kann. Dies ist bei aktiven Objekten nicht mehr möglich, sollen sie doch unabhängig von zugreifenden Anwendungsprozessen und deren Lokation auf Ereignisse reagieren können. Folglich werden aktive Objekte durch eigenständige Laufzeitstrukturen mit eigenem Kontrollfluß repräsentiert. Damit ergibt sich nun die Frage, zu welchen Zeitpunkten diese aktiven Strukturen allokiert sein sollen. Extrempunkte hierfür sind:

1. Ein aktives Objekt besitzt für die gesamte Dauer seiner Existenz eine aktive Laufzeitstruktur.

2. Die Laufzeitstruktur ist genau für die Zeitspanne eines Aufrufs allokiert, wird bei dessen Terminierung automatisch abgebaut und bei jedem neuen Aufruf wieder eingerichtet.

Ist die Effizienz–Problematik der zweiten Alternative offensichtlich, so liegt die Problematik der ersten nicht so schnell auf der Hand. Eine Laufzeitrepräsentation mit derselben Lebensdauer wie die persistente Hintergrundspeicherstruktur ist unmöglich, sofern der Rechner, auf dem die Repräsentation allokiert ist, ausgeschaltet werden kann oder Fehlerfälle, z.B. Stromausfälle, mit berücksichtigt werden. Zudem würden alle aktiven Objekte ständig Ressourcen konsumieren, was deren mögliche Populationsgröße sehr stark beeinträchtigen würde[10].

Da andererseits die Aktivitätsperioden durch Wissen über die Anwendungssemantik sehr genau bestimmt werden könnte[11], ist eine Beeinflußbarkeit auf Modellebene wünschenswert. Dies geschieht in dGOM über die implizit auf aktiven Objekten definierten Operationen *sleep* und *nosleep*. Ein Objekt ist dabei solange aktiv, wie die Zahl der abgesetzten *nosleep*–Operationen die der *sleep*–Operatinen übersteigt. Anschließend fällt es in eine passive Phase zurück, in der es wieder auf den Hintergrundspeicher ausgelagert werden kann. Bspw. bewirkt die folgende Sequenz von Aufrufen

[10] Erste Erfahrungen mit einem unoptimierten Lisp-Prototyp haben gezeigt, daß etwa 50 aktive Objekte auf einer SUN 3/60 Workstation unter UNIX gleichzeitig operieren können, ohne daß das System überlastet ist.

[11] Man denke an ein aktives Objekt, das ein Betriebsmittel repräsentiert. Gleicht man dessen Aktivitätsperioden an den Benutzungszeitraum des Betriebsmittels an, so erhält man die Vorteile aktiver Objekte mit einem vertretbaren Ressourcenverbrauch und mit der nötigen Effizienz.

```
bemi_4711.nosleep;
...;                        !! Irgendwelche Operationen auf bemi_4711
bemi_4711.sleep;
```

daß das Objekt *bemi_4711* wieder passiv wird, sofern nicht noch andere *nosleep*-Aufrufe auf *bemi_4711* stattgefunden haben. Änderungen am Zustand werden in der aktiven Phase atomar — durch lokale Transaktionen [3] — auf einer passiven Kopie durchgeführt, um die Integrität der Zustandsinformation zu garantieren.

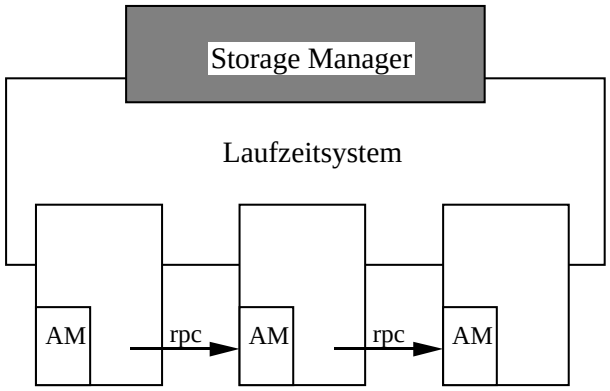

Abbildung 5: Architektur des verteilten GOM

In **Abb. 5** ist das erweiterte Systemmodell von GOM schematisch dargestellt. Die ursprüngliche Funktionalität von GOM wird durch das Laufzeitsystem erbracht, das auf die Dienste eines Storage Managers zur persistenten Sicherung von Objekten zurückgreift. Dieses Laufzeitsystem wird nun um die Funktionen zur Erzeugung und zum Aufruf aktiver Objekte erweitert. Wie die persistenten, passiven Objekte werden hierzu die aktiven Objekte in einer persistenten Objekttabelle verwaltet, mit der die Objekte aufgefunden werden können. Zur Verwaltung der aktiven Objekte wird zusätzlich eine weitere Tabelle benötigt, mit der festgestellt werden kann, inwieweit zu einem aktiven Objekt ein Prozeß verfügbar ist, und auf welchem Rechner dieser sich befindet. Mit der dort abgespeicherten Information können dann die Aufrufe adressiert werden. Dies entspricht im wesentlichen der Basisfunktionalität eines Namensverwaltungssystems im Rahmen der klassischen Architektur verteilter Systeme.

Mit dem Erzeugen eines aktiven Objekts wird lediglich das Objekt in den beiden Tabellen registriert. Durch synchronen oder asynchronen Aufruf einer Operation des aktiven Objekts wird in der zusätzlichen Tabelle die Existenz einer Laufzeitrepräsentation überprüft und falls nötig erzeugt. Diese Repräsentation verfügt über einen *Aktivitätsmanager* (AM), der die ankommenden entfernten Operationsaufrufe entgegennimmt, in einem Warteschlangenobjekt verwaltet und deren Bearbeitung gemäß einer speziellen FIFO–basierten Scheduling–Strategie anstößt. Letzteres ist nötig, da ansonsten die Beschränkung, daß zu einem Zeitpunkt nur ein Aufruf pro aktivem Objekt tätig sein kann, bei rekursiver Objektbenutzung zu einem Deadlock eines einzigen Kontrollflusses mit sich selbst führen würe. Auch werden vom Aktivitätsmanager die Behandlung der sleep/nosleep–Operationen übernommen.

Da die Verteilung transparent ist, erscheint das Laufzeitsystem monolithisch, obwohl es in Wirklichkeit auf eine Vielzahl Prozessen und Rechnern verteilt sein kann. Allerdings ist dabei nur die Erreichbarkeit entfernter Objekte allgegenwärtig, nicht das Wissen über

deren Existenz. Dieses Wissen wird über persistente Variablen als „Einstiegspunkte“ in die Datenbank erreicht.

5.2 Mobile Objekte

Die Integration von Mobilität in die oben geschilderte Implementierung ist sehr einfach möglich. Über eine Sperroperation auf der Tabelle der aktiven Objeke und eine Änderung der dort zu findenden Eintragungen analog zur initialen Erzeugung eines aktiven Objekts ließe sich die Migration gestalten. Durch dieses Konzept kann allerdings nur eine Dimension der Forderung nach Mobilität befriedigt werden: die Verlagerung von Objekten auf die mit Sensoren und Aktoren bestückten Rechner, um das Fehlerverhalten der Schnittstellen zur externen Welt mit den Kontrollobjekten zu koordinieren — d.h. den Einfluß weiterer Komponenten, wie Kommunikationsmedien, auf die Kontrollierbarkeit externer Einheiten auszuschalten. Nicht erfüllt wird jedoch die Erhöhung der Effizienz durch Zusammenführung kommunizierender Objekte, da durch die zentrale Objekttabelle Kommunikation erzwungen wird.

Alternative Möglichkeiten der Realisierung mobiler Objekten — unter Verzicht auf Persistenz — konnte durch eine prototypische Implementierung, dem System DC++ [35] [36], als C++-Klassenbibliothek auf der Basis von DCE [35] gewonnen werden.

Neben den Eigenschaften der Verteilung der Objekte, der systemweiten Objektidentifizierung und der lokationsunabhängigen Aufruftechnik weist das DC++–Objektmodell folgende Eigenschaften aus:

- **Verteilte Objekte** Ein C++–Objekt kann auf jedem Rechner lokal oder entfernt erzeugt werden. Die Verteilung der Objekte wird durch die Anwendung kontrolliert. Ein Objekt stellt in sich eine elementare Verteilungseinheit dar.

- **Dynamische Verlagerung durch Migration** Objekte können dynamisch von einem Rechner auf einen anderen verlagert werden (Objektmigration), z.B. um kommunizierende Objekte lokal zusammenzuführen. Nach der Migration ist ein Objekt unverändert weiterhin aufrufbar. Die Initiierung von Migrationsanforderungen obliegt der Anwendung und erfolgt durch Aufruf spezieller Objektoperationen.

- **Integration mit DCE** Neben der Nutzung des RPC und der UUIDs wird bei DC++ auch die DCE-eigene Sprache IDL verwendet, um zu Objektklassen konforme Schnittstellenbeschreibungen zu spezifizieren. DCE Threads werden der Anwendung über C++-Klassen direkt angeboten und intern auch für die nebenläufige Aufrufbearbeitung genutzt. CDS kann optional zur Identifikation von Objekten per Namen eingesetzt werden, wird aber aus Effizienzgründen nicht für den Lokalisierungsvorgang herangezogen.

Diese Eigenschaften werden durch eine Klassenbibliothek realisiert.

Zugriff auf entfernte Objekte Um die genannten Eigenschaften zu erzielen, wird zunächst eine Indirektion bei der Referenzierung von Objekten mittels einer Hilfsklasse eingeführt. Ein Anwendungsobjekt referenziert damit jeweils ein Hilfsobjekt, das schließlich entweder auf die Speicheradresse eines lokalen Objektes oder auf die Lokation eines entfernten Objektes verweist. Im letzteren Fall leitet das Hilfsobjekt Aufrufe transparent für die Anwendung mittels des DCE RPC weiter; es wird dann als *Stellvertreter* des entfernten Objektes bezeichnet.

Ein Stellvertreter für ein Objekt wird nun in folgenden Fällen auf einem Rechner installiert:

1. ein Objekt O migriert von R1 an einen anderen Rechner R2 und hinterläßt auf R1 einen Stellvertreter für O mit einem Verweis auf R2;

2. ein Rechner erhält Kenntnis von einem entfernten Objekt O (z.B. über einen Referenzparameter bei einem entfernten Aufruf) und installiert einen Stellvertreter für O mit einem Verweis auf die vermutete Lokation von O und

3. Fall 2 gilt auch für alle Objektreferenzen, die von einem nach Rechner R migrierten Objekt O ausgehen; referenziert O beispielsweise die Objekte P und Q entfernt, so werden nach der Migration für P und Q Stellvertreter auf R installiert.

Bei mehrmaliger Migration eines Objektes entsteht eine Verweiskette über mehrere Stellvertreter, die erst nach Absetzen eines Aufrufs an das Objekt mit der aktuellen Objektlokation aktualisiert wird. Zusätzlich hält sich jedoch der Rechner, der ein bestimmtes Objekt erzeugt hat, einen stets aktualisierten Verweis auf dessen momentane Lokation; dies bietet eine alternative Möglichkeit zur Lokalisierung von Objekten, z.B. bei Rechnerausfällen. Hierzu umfaßt jeder Stellvertreter die Angabe neben der vermuteten Lokation auch die Angabe des erzeugenden Rechners eines Objektes.

Untersuchungen am System zeigten, daß eine schnelle Verkürzung von Referenzketten migrierter Objekte den Regelfall darstellt. Die mittlere Länge einer solchen Kette liegt bei knapp über 1. Daher ist es im Mittel effizienter, Objekte über die Verweiskette mit meist einem einzigen RPC aufzurufen als etwa die Alternative über den erzeugenden Rechner (hier R1 für O1) mit zwei RPCs zu wählen. Diese Erfahrung ist prinzipiell auf die Tabelle der aktiven Objekte in dGOM übertragbar.

Eine tiefergehende Darstellung der Realisierung von DC++, speziell der entwickelten Klassenbibliotheken, oder sonstiger Erfahrungen, wie Laufzeitmessungen, würde den hier gegebenen Rahmen sprengen, ist jedoch in [35, 36] zu finden.

6 Zusammenfassung

Der Beitrag gab einen Überblick über ein neues Modellierungskonzept für dezentrale Ingenieuranwendungen basierend auf aktiven und mobilen Objekten. Es wurde gezeigt, daß objektorientierte Datenbanksysteme zwar ein geeignetes Medium zur Integration der inhärenten Objektverteilung ingenieurwissenschaftlicher Applikationen darstellt, jedoch hinsichtlich der ablauforientierten Beschreibung wesentliche Mängel aufweisen.

Diese Unzulänglichkeit kann z.T. durch die Verwendung aktiver Objekte behoben werden, die sich durch synchronen und asynchronen Nachrichtentransfer gegenseitig beeinflussen können. Erste Erfahrungen konnten mit der objektorientierten Datenbanksprache GOM und dem verteilten objektorientierten System DC++ gewonnen werden.

Zukünftig sollen die in Abschnitt 4.4.2 erörterten Defizite hinsichtlich der Spezifikation aktiven Verhaltens sowie der Wahrung objektübergreifender Konsistenzbedingungen unter den Voraussetzungen, daß Objekte autonom sind und möglicherweise in einer verteilten Umgebung über unzuverlässige Medien kommunizieren, behoben werden. Dabei soll auch die Effizienz des Systems nicht außer acht gelassen werden. Aus den bisher gewonnenen Erfahrungen stellt dabei die ubiquitäre Natur des Ansatzes, der sich in der globalen

Objekttabellen niederschlägt, ein vordringlich zu lösendes Problem dar. Ein solcher Ansatz stellt keine realistische Lösung für eine unternehmensweite Infrastrukturbasis dar. Ein globale Objekttabelle würde einen schwerwiegenden Engpaß bilden. Betrachtet man zudem die Infrastrukturanforderungen für eine standortübergreifende enge Kopplung, so ist eine Aufgliederung des Systems wünschenswert. Im Rahmen der dabei entstehenden *GOM–Cluster* bilden vor allem mobile Objekte eine wesentliche Modellierungstechnik. Kann dies basierend auf dem DC++–Konzept gelöst werden, so stellen vor allem Fragen des Transfers von Wissen über eine bestehende Objektpopulation zwischen Clustern neue Fragen.

Das beschriebene System wird den Maschinenbauinstituten des Sonderforschungsbereichs 346 zur Verfügung gestellt werden und dort das jetzt eingesetzte System GOM ersetzen. Es ist in diesem Kontext als informationstechnisches Rückgrat ingenieurwissenschaftlicher Anwendungen zu sehen. Da zum Informationaustausch die bloße Möglichkeit des Zugriffs auf eine gemeinsame Objektbank nicht ausreicht, sondern Wissen über die Semantik der abgelegten Daten notwendig ist, wird parallel ein konzeptionelles Schema für den Ingenieurbereich entwickelt. Die Integration der Anwendungen basiert dann auf der Umsetzung dieses Schemas — des Produkt–/Produktionsmodells — auf GOM bzw. dGOM.

Literatur

[1] G. R. Andrews. Synchronizing resources. In N. Gehani and A. D. McGettrick, editors, *Concurrent Programming*, pages 184–215. Addison Wesley, Reading, MA, 1988.

[2] M. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. Int. Conf on Deductive and Object-Oriented Databases*, pages 40–57, Kyoto, Japan, Dec 1989.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[4] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Trans. on Software Engineering*, 13(1), Jan 1987.

[5] P. Butterworth, A. Otis, and J. Stein. The gemstone object database management system. *Communications of the ACM*, 34(10):64–77, Oct 1991.

[6] D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, Sep. 1993.

[7] S. Chakravarthy, E. Anwar, and L. Maugis. Design and implementation of active capability for an object-oriented database. Technical Report UF-CIS-TR-93-001, University of Florida, 1993.

[8] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. Technical Report UF-CIS-TR-93-007, University of Florida, Gainesville, Florida 32611, Mar. 1993.

[9] P. P. S. Chen. The Entity Relationship model: Toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, Mar 1976.

[10] P.K. Chrysanthis, S. Raghuram, and K. Ramamritham. Extracting concurrency from objects: A methodology. In J. Clifford and R. King, editors, *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 108–117, Denver, Colorado, May 1991.

[11] O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10):34–48, Oct 1991.

[12] O. Diaz, N. Paton, and P. Gray. Rule management in object oriented databases: A uniform approach. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, Barcelona, Spain, 1991.

[13] A. K. Elmargarmid, editor. *Database Transaction Models For Advanced Applications.* Morgan Kaufmann Publishers, Inc., 1992.

[14] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, May 87.

[15] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Proc. of 1st Intl. Workshop on Rules in Database Systems*, Edinburgh, UK, 1993.

[16] S. Gatziu and K. R. Dittrich. Samos: An active object-oriented database system. *IEEE Quarterly Bulletin on Data Engineering*, Jan. 1993.

[17] N. Gehani and W. D. Roome. *The Concurrent C Programming Language.* Silicon Press, 1989.

[18] N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, 1991.

[19] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[20] J. Gray and A. Reuter. *Transaction Processing: Concept und Techniques.* Morgan Kaufmann, New York, 1992.

[21] J. N. Gray. The transaction concept: Virtues and limitations. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 144–154, 1981.

[22] A. Kemper, P. C. Lockemann, G. Moerkotte, and H. D. Walter. Autonomous objects: A natural model for complex applications. In *Proc. of 1st Intl. Workshop on Next Generation Information Technology and Systems*, Haifa, Israel, Jun 1993.

[23] A. Kemper and G. Moerkotte. Basiskonzepte objektorientierter Datenbanksysteme. *Informatik-Spektrum*, 16(2), Apr 1993.

[24] A. Kemper, G. Moerkotte, and H.-D. Walter. Structuring the distributed object-world of CIM. In *INCOM'92 – 7th IFAC/IFIP/IFORS/IMACS/ISPE Symposium on Information Control Problems in Manufacturing Technology*, Toronto, Canada, May 1992.

[25] A. Kemper, G. Moerkotte, H.-D. Walter, and A. Zachmann. GOM: a strongly typed, persistent object model with polymorphism. In *Proc. of the German Conf. on Databases in Office, Engineering and Science (BTW)*, pages 198–217, Kaiserslautern, Mar 1991. Springer-Verlag, Informatik Fachberichte Nr. 270.

[26] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The Objectstore database system. *Communications of the ACM*, 34(10):50–63, Oct 1991.

[27] H. Ledgard. *Ada – An Introduction.* Springer-Verlag, New York, Heidelberg, Berlin, 1981.

[28] L. Liu and R. Meersman. Activity model: Declarative approach for capturing communication behaviour in object-oriented databases. In *18th International Conference on Very Large Data Bases*, Vancouver, British Columbia, Canada, 1992.

[29] P. C. Lockemann et al. Anforderungen technischer Anwendungen an Datenbanksysteme. In *Informatik Fachberichte No. 94*, pages 1–26, Berlin, Heidelberg, 1985. Springer Verlag.

[30] P.C. Lockemann and H.-D. Walter. Activities in object bases. In *Proc. of 1st Int. Workshop on Rules in Database Systems*, Edinburgh, UK, 1993.

[31] M. Papathomas. *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming.* PhD thesis, Université de Genève, 1992.

[32] M. Papathomas and D. Konstantas. Integrating concurrency and object-oriented programming: An evaluation of hybrid. In D. Tsichritzis, editor, *Object Management*, pages 229–245. Centre Universitaire d'Informatique, Université de Genève, 1990.

[33] A. Reuter and H. Wächter. The contract model. *IEEE Data Engineering Bulletin*, 14(1), March 1991.

[34] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[35] A. Schill and M. Mock. Dce++: Distributed object-oriented system support on top of osf dce. *Distributed Systems Engineering Journal*, 1(2), 1993.

[36] A. Schill and M. Person. Verteilte objektorientierte Erweiterung des OSF Distributed Computing Environments. *Offene Systeme*, 3:94–100, 1993.

[37] P. M. Schwarz and A. Z. Spector. Synchronizing abstract types. *ACM Trans. Computer Systems*, 2(3):223–250, 1984.

[38] K. E. Smith and S. B. Zdonik. Intermedia: A case study of the differences between relational and object-oriented database systems. In *Proc. of the ACM Conf on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 452–465, Oct 1987.

[39] H.-P. Wiendahl. *Betriebsorganisation für Ingenieure.* Carl Hanser Verlag, 2nd edition, 1986. in german.

[40] A. Yonezawa, editor. *ABCL—An Object-Oriented Concurrent System.* The MIT Press, 1990.