Integration von Werkzeugen in heterogene, prozeßgesteuerte Software-Entwicklungsumgebungen

-

TOOL INTEGRATION CONCEPT

Olaf Neumann

Dissertation

zur Erlangung des Grades

Doktor der Naturwissenschaften

der Universität-Gesamthochschule Paderborn

im Fachbereich Mathematik - Informatik

Bad Soden am Taunus

März 2002

Ich bedanke mich bei Herrn Prof. Dr. W. Schäfer für die Betreuung der Arbeit und Herrn Prof. Dr. G. Engels für seine Bereitschaft, das Korreferat zu übernehmen.

Mein herzlicher Dank gilt insbesondere auch den vielen Kollegen und Freunden, die mich bei dieser Arbeit unterstützt haben:

Dr. S. Sachweh, Dr. G. Junkermann, Dr. S. Wolf, B. Peuschel, Dr. J. Jahnke, Dr. W. Emmerich und W. Reimer, die in vielen fruchtbaren Disskussionen durch wertvolle Anregungen wichtige Impulse gegeben haben,

W. Reimer, Dr. G. Junkermann und ganz besonders meiner Frau Inkemai für das Korrekturlesen der Arbeit.

Zum Schluß aber möchte ich vor allem meiner Familie danken, ohne deren Rückhalt diese Arbeit heute nicht so vorliegen würde. Meinen Eltern dafür, daß sie mir die Ausbildung ermöglicht und meine Interessen immer vorbehaltlos gefördert haben, und vor allem meinen Kindern, Deike, Timm, Imme und Tüge, die wegen dieser Arbeit viel auf ihren Vater verzichten mußten.

Bad Soden am Taunus, im Mai 2002

INHALTSVERZEICHNIS

1

EINLEITUNG

Weltweit gibt es einen großen Bedarf an Computerspezialisten, da Computer in immer mehr Bereichen Einzug finden. Entsprechend gibt es immer mehr und auch immer größere und komplexere Projekte im Bereich der Softwareentwicklung. Teilweise arbeiten mehr als hundert Entwickler auf mehrere Standorte verteilt gemeinsam an einem Software-Entwicklungsprojekt. Ein bekanntes Beispiel ist die Entwicklung der Software für die Airbus-Reihe. Solche Softwaresysteme sind sensible Gebilde. Je nachdem wie gut die Projekte zu ihrer Erstellung koordiniert werden können, fällt auch ihr Ergebnis im Sinne der Qualität der entwickelten Software und der Plangenauigkeit aus. Die Koordination umfaßt damit insbesondere qualitätssichernde Maßnahmen, die gerade in großen Projekten dringend benötigt werden.

Als Ausgangspunkt für eine erfolgreiche Entwicklungsplanung bietet sich die Benutzung von modernen Prozeßmodellierungssprachen an, im folgenden kurz PML (Process Modelling Language) genannt. Diese erlauben es, die einem Projekt zugrundeliegenden Abläufe und Ressourcen in einem generischen Modell projektunabhängig und damit wiederverwendbar zu beschreiben. Zu einer solchen Beschreibung gehört z.B. die Angabe der zu erstellenden Dokumenttypen und deren Abhängigkeiten untereinander, der Rollen, die Aufgaben im Projekt wahrnehmen und die Definition der Aufgaben, die zu verschiedenen Zeitpunkten im Projekt zu erledigen sind. Weiterhin wird die Dynamik des Prozesses beschrieben, so daß die Koordination der Aufgaben und ihrer Bearbeiter festgelegt ist (z.B. wer zu welchem Zeitpunkt auf einen Dokumenttyp X zugreifen darf).

Prozeßbeschreibungen, die mit PML arbeiten, unterscheiden sich von herkömmlichen, in Projekten eingesetzten Regelwerken im allgemeinen dadurch, daß sie ähnlich wie ein Programm ausgeführt werden können. PML, die eine Ausführung zulassen, sind daher eine Art applikationsspezifische Programmiersprache.

Werden solche Prozeßbeschreibungen zur Parametrisierung von modernen Workflow-Umgebungen eingesetzt, so erlauben sie eine Überwachung des Prozeßablaufs. Abweichungen vom vorgesehenen Ablauf können erkannt und gemeldet oder auch automatisch unterbunden werden. Solche Kontrollen tragen zur Qualitätssicherung bei und können im Rahmen von Zertifizierungsmaßnahmen wie ISO 9000ff interessant sein.

Wie gut eine Prozeßmodellierung außerdem zur Steuerung eines Prozesses verwendet werden kann, hängt im wesentlichen von zwei Faktoren ab. Zum einen sollte die benutzte PML möglichst problemadäquate Beschreibungsmöglichkeiten anbieten und wenig Overhead erzwingen. Zum anderen sollte die Prozeßbeschreibung in eine automatische Unterstützung bzw. Kontrolle der vorgesehenen Abläufe umgesetzt werden können. Die Umsetzung wird in einer Workflow-Umgebung realisiert, die im Umfeld der Softwareentwicklung prozeßgesteuerte Software-Entwicklungsumgebung (PSEU) genannt wird. Wie groß der Vorteil einer automatisierten Projektsteuerung durch eine PSEU im Vergleich zu einer herkömmlichen, manuellen Projektkoordination ist, hängt davon ab, wie die folgenden Punkte realisiert werden können:

- Überwachung der definierten Prozeßabläufe
  - auf die Einhaltung der Reihenfolge der einzelnen Prozeßschritte,
  - in der Zuordnung zu Rollen bzw. Ressourcen und
  - in den Prozeßaktivitäten
- Kontrolle der Projektkonsistenz
- Umsetzung von Versionierungskonzepten
- Umsetzung geeigneter Konzepte zur parallelen Arbeit
- Bereitstellung einer komfortablen Umgebung, um die Akzeptanz bei den Entwicklern zu gewährleisten

Diese Punkte können letztlich nur dann wirkungsvoll umgesetzt werden, wenn sich Informationen über den laufenden Entwicklungsprozeß automatisch gewinnen lassen. Solche Informationen können nur von den Werkzeugen automatisch gegeben werden, mit deren Hilfe die Prozeßaktivitäten durchgeführt werden.

Es gibt zwei Wege, um Werkzeuge so in eine PSEU zu integrieren, daß Prozesse kontrollierbar sind. Zum einen kann eine spezialisierte Umgebung erstellt werden, die alle notwendigen Werkzeuge als Bestandteil der Umgebung selbst implementiert. Diese Lösung erlaubt eine optimale Abstimmung von Umgebung und Werkzeugen, da die Werkzeuge als solche nicht mehr einzeln existieren. Insbesondere kann die Steuerung der PSEU auf Interna der Werkzeuge zugreifen. Dieser Weg erfordert einen hohen Arbeitsaufwand zur Erstellung der PSEU, den die Nutzer von PSEU in der Regel nicht selbst erbringen wollen/können und deshalb eine PSEU kaufen. Eine PSEU, deren Werkzeuge fest eingebaut sind, führt insofern zu mangelnder Flexibilität, als ein Anwender/Entwickler seine Projekte nur mit den vorgegebenen Werkzeugen durchführen kann. Er hat keine Möglichkeit, andere Anwendungen zu benutzen, die ihm unter Umständen passender für sein Problem erscheinen.

Bei der anderen Möglichkeit werden unabhängige Werkzeuge zur Bearbeitung der im Prozeß definierten Aufgaben verwendet. Diese Lösung läßt ein Höchstmaß an Flexibilität im Bereich der Bearbeitungswerkzeuge zu. Die Nachteile solcher Konzepte liegen bei bisher realisierten Ansätzen in einer fehlenden Abstimmung mit der Steuerungskomponente der Umgebung. Außerdem ist für die Anbindung der Werkzeuge wenig Unterstützung vorgesehen. Gerade in Bereichen mit

- häufig wechselnden Anforderungen an die benötigten Werkzeuge,
- einer großen Anzahl verschiedenartiger Werkzeugtypen (die i.d.R. nicht alle vom gleichen Hersteller angeboten werden können)
- oder dem Wunsch, Prozeßsteuerung in bereits laufende Projekte einzuarbeiten

ist es jedoch unbedingt nötig, flexibel zu sein und verschiedenartige Werkzeuge integrieren und nutzen zu können. Die vorliegende Arbeit entwickelt Konzepte, mit denen eine Basis geschaffen wird, um eine derartige Integration von Werkzeugen effektiv und effizient vorzunehmen.

2

PROZEßGESTEUERTE SOFTWARE-ENTWICKLUNGSUMGEBUNGEN

Dieses Kapitel zeigt, wie sich das Thema der Arbeit im Gesamtzusammenhang von prozeßgesteuerten Software-Entwicklungsumgebungen einordnet. In Kapitel 2.1 werden dazu einige grundlegende Begriffe definiert. In Kapitel 2.2 werden offene Fragen identifiziert, das Thema der Arbeit abgegrenzt und die Zielsetzung der Arbeit detailliert erläutert. Weiterhin werden in diesem Kapitel Bezüge zu anderen Arbeiten hergestellt. Kapitel 2.3 stellt dann den Aufbau der Arbeit dar.

2.1 Einführung grundlegender Begriffe

In diesem Kapitel geht es zunächst um Begriffe, die im Zusammenhang mit prozeßgesteuerten Software-Entwicklungsumgebungen benötigt werden. Es schließen sich Begriffe an, die bei der Integration von Werkzeugen benutzt werden. Diesen beiden Teilen werden die Definitionen für Werkzeuge und Software-Entwicklungsumgebungen vorangestellt, da sie als Grundlage für die beiden anderen Kapitel dienen.

Für die Definition des Werkzeugbegriffs finden sich in der Literatur verschiedene Ansätze, die teilweise sogar widersprüchlich sind (vgl. z.B. [Fug93], [Was89], [TN92], [Som92a], [SB93], [Lef95], [BM92], [Coa96], [Yan92], [Poh96, Teil 3] und auch Anhang D). In dieser Arbeit werden Werkzeuge wie folgt definiert:

Werkzeug: Jede in sich geschlossene, ausführbare Programmeinheit ist ein *Werkzeug*.

Werkzeuge sind beispielsweise Editoren oder Interpreter, die eigenständig aufgerufen werden. Nach dieser Definition sind beispielsweise aber auch Kombinationen aus Editoren und Interpretern, die nur gemeinsam benutzt werden können, als ein einziges Werkzeug zu verstehen. Ein Beispiel für eine solche Kombination ist eine PROLOG- oder Smalltalk-Umgebung, die zum einen die Eingabe von Programmen erlaubt, und auf der anderen Seite auch deren Ausführung als Teil der Funktionalität beinhaltet. Obwohl die Definition unscharf ist, läßt sich mit ihr in der Praxis gut arbeiten, da sie

der großen Bandbreite einbindbarer Werkzeuge gerecht wird. Programme sind per se in sich geschlossen und ausführbar. Auch die Module einer Bibliothek, deren Teile wieder zu größeren Einheiten zusammengesetzt werden können, sind in diesem Sinne abgeschlossen.

Software-Entwicklungsumgebung: Eine *Software-Entwicklungsumgebung* (kurz *SEU*) ist die Zusammenfassung einer Menge von Werkzeugen, die in ihrer Funktionalität den gesamten Software-Lebenszyklus abdeckt.

Setzt sich eine Software-Entwicklungsumgebung aus Teilen zusammen, die nicht in sich abgeschlossen sind, die also nicht aus der Umgebung herausgenommen werden können, so muß die Umgebung als ein einziges Werkzeug aufgefaßt werden. Die Menge von Werkzeugen enthält dann genau ein Element.

2.1.1 Prozeßgesteuerte Software-Entwicklungsumgebungen

Software-Entwicklungsumgebungen dienen dazu, die Entwicklung von Software durch eine zusammenhängende, den Software-Lebenszyklus abdeckende Menge von Werkzeugen zu unterstützen. Das Ergebnis der Entwicklung ist ein Softwareprodukt.

Softwareprodukt: Ein *Softwareprodukt* ist die Menge aller Objekte (auch: Dokumente), die während der Entwicklung von Software erstellt werden.

Bei der Entwicklung von Software entstehen verschiedene Objekte. Diese reichen von ersten Anforderungsbeschreibungen des Kunden über interne Richtlinien bis hin zum fertigen Programm und dem dazu erstellten Benutzungshandbuch. Auch Dokumente, die nur mittelbar mit der Entwicklung zusammenhängen und nicht an den Kunden ausgeliefert werden, zählen dazu, z.B. Projektpläne oder Aufgabenbeschreibungen.

Um ein Softwareprodukt zu entwickeln, arbeiten viele Entwickler zusammen. Sie führen eine Reihe von *Aktivitäten* aus, um Teile des Softwareproduktes zu erstellen oder zu verändern. Die Entwicklung eines Softwareproduktes findet im Rahmen eines Softwareprojektes statt.

Softwareprojekt: Ein *Softwareprojekt* (kurz: *Projekt*) beschreibt die Ablaufreihenfolge der Entwicklungsaktivitäten, die konkret zu erstellenden Dokumente, die einzusetzenden Entwickler und die sonstigen benutzten Ressourcen.

Die Definition des Softwareprojektes entspricht also der intuitiven Vorstellung, in der ein Softwareprodukt das Ergebnis einer Softwareentwicklung ist und das Projekt den Rahmen definiert, in dem das Produkt entwickelt wird. Die Abläufe während der Entstehung des Softwareprodukts werden als Softwareprozeß bezeichnet.

Softwareprozeß: Ein *Softwareprozeß* (kurz: *Prozeß*) umfaßt die im Rahmen einer Entwicklung von Software entstehenden/benutzten Objekte, Ressourcen und Aktivitäten.

Die Formulierung eines Prozesses wird heutzutage in der Praxis meist noch informal vorgenommen. Damit sind aber natürlich die bekannten Probleme wie z.B. Mehrdeutigkeiten oder auch Inkonsistenzen innerhalb der Beschreibung verbunden. Besser ist es, stattdessen eine formale Notation zu wählen. Dazu wird ein Modell des Software-

prozesses erstellt, das dazu dient, verschiedene Projekte gleichartig durchführen zu können und außerdem nicht jedes Projekt vollständig neu definieren zu müssen. Für solche generischen Beschreibungen wird der Begriff des Softwareprozeßmodells eingeführt.

Softwareprozeßmodell: Ein *Softwareprozeßmodell* (kurz: *Prozeßmodell*) ist die abstrakte Beschreibung der Ressourcen und Abläufe, die zur Entwicklung von Software nötig sind. Ein Softwareprozeßmodell ist demnach die abstrakte Beschreibung eines Softwareprozesses.

Ein Prozeßmodell beschreibt somit gewissermaßen eine Klasse von Projekten, die alle gleichartig durchgeführt werden sollen. Die in einem Prozeßmodell spezifizierten Klassen von Projekten können verschiedenster Natur sein. Es können z.B. sehr restriktive Entwicklungsabläufe für die Implementierung sicherheitsrelevanter Software beschrieben werden, die sich durch ausgeprägte Testzyklen ohne erlaubte Abweichungen vom Prozeß charakterisieren. Außerdem lassen sich auch Abläufe bei der Entwicklung termingebundener Standardsoftware realisieren, bei der die Einhaltung des Auslieferungstermins unter Umständen über die Ausführung aller Tests vor der ersten Auslieferung gestellt wird. Es wird nun für jedes Projekt einzeln entschieden, welcher Prozeß angewendet werden soll, so daß z.B. die Software für ein Flugzeug oder ein Kraftwerk auf der Basis des sicherheitsbetonten Prozesses entwickelt wird, während die Entwicklung eines Textverarbeitungsprogramms dem weniger restriktiven Prozeß folgt.

In einem Prozeßmodell werden die einen Prozeß beschreibenden Entitäten und ihre Zusammenhänge mit dem Ziel formalisiert, die oben erwähnten Mehrdeutigkeiten und Inkonsistenzen zu beseitigen. Die darin beschriebenen Bestandteile umfassen die Dokumenttypen, die auf den Dokumenten auszuführenden Aktivitäten und die Rollen der Entwickler. Beispielsweise könnte damit beschrieben werden, daß eine Architekturdefinition (Dokumenttyp) von einem Entwickler in seiner Eigenschaft als Softwaredesigner (Rolle) spezifiziert wird (Aktivität).

In einem Prozeßmodell werden typischerweise keine Bezüge zu einzelnen zu benutzenden Werkzeugen gemacht. Der Grund hierfür liegt darin, daß ein und dieselbe Aktivität eines Softwareprozesses in verschiedenen Projekten durch unterschiedliche Werkzeuge realisiert werden kann. Die Art des Werkzeugs (Architektureditor, Übersetzer, …) ist durch die Benennung der Aktivität (und möglicherweise durch ergänzende Kommentare) gegeben. Die Semantik der Aktivität und damit ihre konkrete Zuordnung zu einem Werkzeug wird in der Regel nur durch die Interpretation des Namens und durch die Zuordnung durch einen Projektadministrator vorgenommen.

Das formale Softwareprozeßmodell muß, um aufgeschrieben werden zu können, in einer geeigneten Sprache formuliert sein. Daß sich die natürliche Sprache nicht eignet, ist bereits klar geworden. Gängige Programmiersprachen eignen sich auch nur bedingt, da sie die oben angesprochenen Entitäten eines Prozeßmodells nicht von sich aus kennen und diese jedesmal nachbilden müßten. Es muß daher eine spezielle, dedizierte Sprache eingeführt werden.

Softwareprozeßmodellierungssprache: Eine *Softwareprozeßmodellierungssprache* (kurz: *Prozeßmodellierungssprache, PML* aus dem englischen *Process Modeling Language*) ist eine Sprache zur formalen Beschreibung eines Softwareprozesses.

In der Vergangenheit wurden schon viele unterschiedliche Sprachen definiert, die sich unter anderem durch verschiedene zugrundeliegende Paradigmen (imperativ, deklarativ, objektorientiert, petrinetzbasiert) unterscheiden. Eine Diskussion dieser verschiedenen Ansätze ist in der Literatur zu finden (vgl. z.B. [CLJ91], [ABGM92], [Lon94]) und wird hier nicht vorgenommen.

Der zusätzliche Aufwand, der bei der Verwendung einer Prozeßmodellierungssprache durch die Formalisierung des Prozesses entsteht, ist durch das Eliminieren von Mehrdeutigkeiten und Inkonsistenzen alleine jedoch noch nicht zu rechtfertigen. Eine Prozeßmodellierungssprache sollte zusätzlich ausführbar sein, um auch die korrekte Durchführung des spezifizierten Prozeßmodells gewährleisten zu können. Um ein Prozeßmodell ausführen zu können, ist die Verfeinerung der Prozeßbeschreibung mit zusätzlichen Informationen notwendig, die einer Instanziierung des Modells entspricht. Die Ausführung des Prozeßmodells erfolgt durch ein dediziertes Laufzeitsystem, die Prozeßsteuerungskomponente.

Prozeßsteuerungskomponente: Eine *Prozeßsteuerungskomponente* (auch *Prozeßmaschine*, *process engine*, *kurz PE*) führt ein instanziiertes Prozeßmodell aus und steuert damit ein Softwareprojekt.

In Analogie zu Programmiersprachen wird die Erstellung eines Prozeßmodells zur Ausführung mit einer Prozeßsteuerungskomponente auch *Prozeßprogrammierung* genannt ([Ost87]).

Die Prozeßsteuerungskomponente ist die Kernkomponente der Ausführung von Prozeßprogrammen. Sie steuert ein Projekt, indem sie deren Ausführung unterstützt und dabei erhaltenes Feedback mit berücksichtigt.

Prozeßgesteuerte Software-Entwicklungsumgebung: Eine *prozeßgesteuerte Software-Entwicklungsumgebung* (kurz *PSEU*) ist eine Software-Entwicklungsumgebung, in der die Ausführung von Aktivitäten und der Zugriff auf die Informationen eines Softwareprojekts durch eine Prozeßmaschine überwacht oder gesteuert werden.

Das Umfeld einer PSEU läßt sich in einzelne Teile untergliedern. Aus den verschiedenen Möglichkeiten (vgl. dazu z.B. [Lon93], [FH93], [Dow93], [Coa96]) wird in dieser Arbeit der Ansatz von Dowson und Fernström verwendet (siehe [DF94]). Die darin benutzten drei konzeptionell voneinander getrennten Teilbereiche sind in Abbildung 2.1 dargestellt. Es handelt sich dabei um:

- *Modellierung* (Model Domain): Der Bereich der Modellierung umfaßt alle Tätigkeiten zum Erstellen und Warten von Prozeßmodellen auf der Basis geeigneter Prozeßmodellierungssprachen. Die Prozeßmodelle sind dann automatisch ausführbar.
- *Modellausführung* (Enactment Domain): In diesem Bereich wird alles zusammengefaßt, was nötig ist, um einen Entwickler mit Hilfe einer PSEU bei seiner Tätigkeit passiv zu unterstützen (oder auch aktiv, einzuschränken beziehungsweise zu kontrollieren).
- *Prozeßausführung* (Performance Domain): Dieser Bereich umfaßt die tatsächlichen Projekttätigkeiten, die im Rahmen des Prozesses von den Entwicklern manuell (sog. human agents) oder vom Computer (sog. non-human agents) ausgeführt werden. Mittel zum Zweck sind dabei die durch die PSEU zur Verfügung gestellten Werkzeuge.

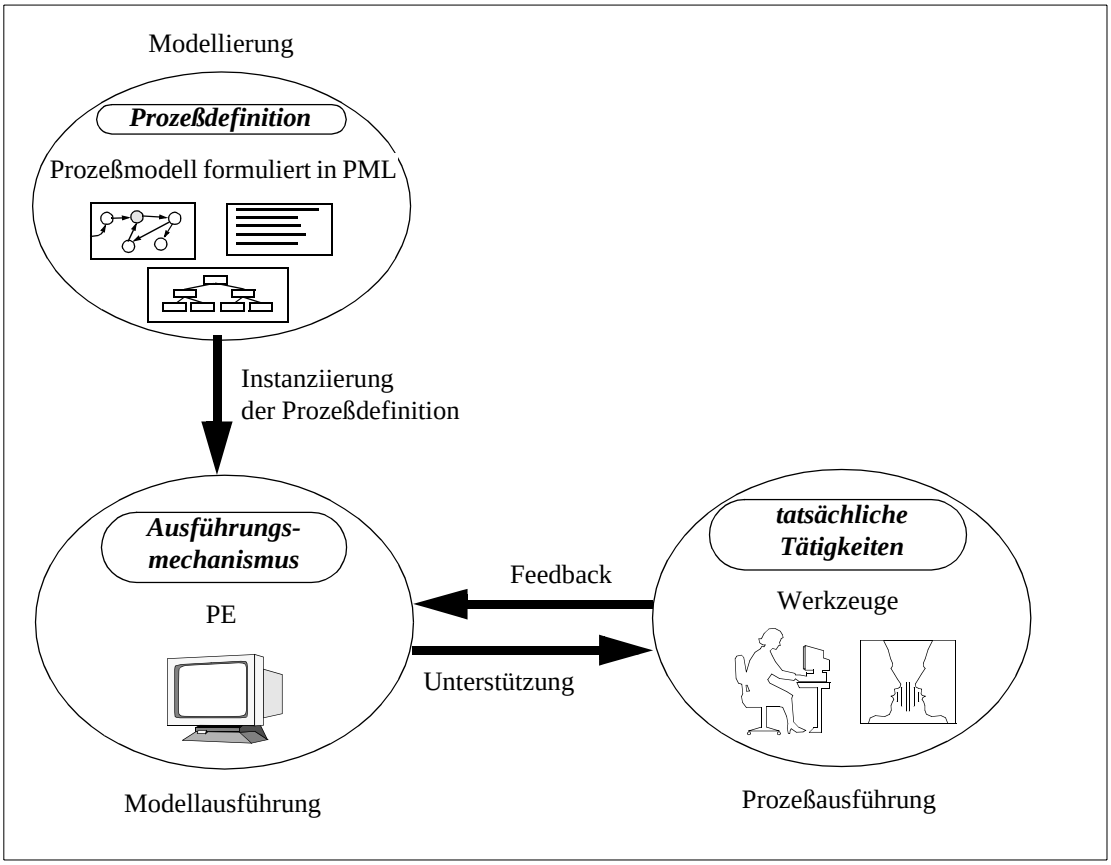


Abbildung 2.1 Teilbereiche prozeßgesteuerter Software-Entwicklungsumgebungen

Die Modellausführung wird durch die Instanziierung des in der Modellierung allgemein spezifizierten Prozesses parametrisiert. Die Modellausführung unterstützt und kontrolliert dann die Tätigkeiten der Prozeßausführung, die durch die Werkzeuge implementiert werden. Diese geben wiederum Feedback an die Modellausführung zurück, um deren Unterstützung und Kontrolle zu ermöglichen.

2.1.2 Werkzeugintegration

Die Durchführung von Aktivitäten in der Prozeßausführung wird durch die Ausführung von Werkzeugen in der Modellausführung realisiert. Diese Werkzeuge müssen in die Umgebung integriert werden, was im obigen Sinn bedeutet, daß Unterstützungs- und Feedbackmöglichkeiten geschaffen werden.

Werkzeugintegration: Unter *Werkzeugintegration* (kurz Integration) wird die Implementierung oder Anpassung von Werkzeugen verstanden, so daß die zu integrierenden Werkzeuge mit einem oder mehreren weiteren zusammenarbeiten können.

Es wird davon gesprochen, daß z.B. zwei Werkzeuge gut miteinander integriert sind oder auch ein Werkzeug gut in eine Umgebung integriert ist. Ein Werkzeug kann in Bezug auf ein anderes gute Integration aufweisen und in Bezug auf ein zweites eine schlechte oder sogar gar keine. Für Kriterien, was unter „gut integriert“ oder „schlecht integriert“ zu verstehen ist, sei an dieser Stelle auf die Literatur verwiesen, z.B. die Betrachtungen in [TN92]. Die Schwierigkeiten, die sich mit solchen Einstufungen ergeben, werden in Anhang C dargestellt.

In der Definition wird keine Aussage getroffen, wie eine Integration verschiedener Werkzeuge technisch realisiert werden kann/muß. Es wird allerdings implizit auf zwei unterschiedliche Ansätze zur Integration von Werkzeugen Bezug genommen. In der Definition werden eine Implementierung und eine Anpassung als mögliche Vorgehensweisen angesprochen. Dahinter verbirgt sich, daß zwei Werkzeuge schon während ihrer Implementierung miteinander integriert werden können, was durch eine Abstimmung der beiden Implementierungen erreicht wird. Dieser Fall wird auch als *a priori Integration* („Integration im Vorwege“) bezeichnet.

A priori Integration: Unter *a priori Integration* versteht man die Integration von Werkzeugen schon während ihrer Implementierung, z.B. durch Anwendung von Standards oder durch Verwendung gemeinsamer Schnittstellendefinitionen.

Die Benennung leitet sich davon ab, daß die einzusetzenden Techniken und Strategien schon definiert werden, bevor der eigentliche Bedarf dafür auftritt. Diese Form der Integration hat offensichtlich die besten Chancen auf eine „optimale“ Anpassung der betroffenen Werkzeuge unabhängig von der eingesetzten Integrationstechnik. Andererseits ist ebenso klar, daß auch eine a priori Integration nur so gut ist wie die Planung, die hinter den verwendeten Techniken und Strategien steht. Somit kann selbstverständlich auch die „optimale“ Anpassung zweier Werkzeuge eine sehr begrenzte Qualität der Integration bedeuten.

Die zweite Möglichkeit, Werkzeuge zu integrieren, ist die nachträgliche Anpassung der Werkzeuge mit der Absicht, ihre Zusammenarbeit zu verbessern. Diese Form der Integration wird entsprechend *a posteriori Integration* („Integration im nachhinein“) genannt.

A posteriori Integration: Unter *a posteriori Integration* versteht man die Anpassung von Werkzeugen, die erst dann durchgeführt wird, wenn der Bedarf zur Integration mit einem oder mehreren anderen Werkzeugen entsteht.

Es gibt im wesentlichen drei Möglichkeiten, ein Werkzeug mit einem zweiten a posteriori zu integrieren. Die erste Möglichkeit benötigt den Zugriff auf die Quelltexte des anzupassenden Werkzeugs. Sind die Quelltexte vorhanden, wird eine integrierte Version des Werkzeugs durch eine Veränderung des Quelltextes erstellt. Diese Möglichkeit ist vom Effekt her fast identisch mit der a priori Integration. Sie ermöglicht daher eine sehr gute Integration in der Regel mit einem hohen Integrationsaufwand, da eine vollkommen eigenständige Version des angepaßten Werkzeugs erstellt wird. Für reine Anwender eines Werkzeugs kommt diese Form der Anpassung meist schon deswegen nicht in Frage, weil die Quellen eines nicht selbst entwickelten Werkzeugs dem Anwender selten vorliegen (Ausnahme ist z.B. Open Source Software). Eine ähnliche Anpassungsmöglichkeit ergibt sich, wenn zumindest Objektdateien des anzupassenden Werkzeugs vorliegen, da dann einzelne Teile des Programms reimplementiert werden können. Die Bemerkungen sind die gleichen wie zur ersten Möglichkeit. Der für „normale“ Anwender häufigste Fall wird die Verwendung von Zusatzsoftware sein, die für die Integration verantwortlich ist. Die Grundidee dabei ist, daß ein Werkzeug durch die neue Software so gekapselt wird, daß sie gewissermaßen a priori mit den betroffenen Werkzeugen integriert wird und dann als eigentliche Funktionalität eine Schnittstellenumsetzung der einzelnen Schnittstellen aufeinander vornimmt. Solche Kapseln werden üblicherweise als Envelopes oder Wrapper bezeichnet (vgl. z.B. [KFP88], für Beispiele für Anwendungen siehe u.a. [VK95], [SPA95b], [Ger94]). Die dabei zu errei-

chende Qualität der Integration hängt stark von den zu integrierenden Werkzeugen ab, da die einzelnen Schnittstellen nicht erweitert oder verändert werden. Besitzt eines der beteiligten Werkzeuge nur geringe Möglichkeiten zur Integration mit anderen Werkzeugen, so werden diese Möglichkeiten durch eine solche Umsetzung der Schnittstelle nicht erweitert und die Integration bleibt auf dem angebotenen Level. Zusätzlich ergeben sich möglicherweise Einschränkungen dadurch, daß die Schnittstelle eines Werkzeugs nicht ohne weiteres auf die Schnittstelle eines anderen Werkzeugs abgebildet werden kann.

Für die Umgebungen, in die Werkzeuge integriert werden sollen, gilt, daß sie verschieden gut geeignet sein können, die Integration von Werkzeugen zu erlauben. Es gibt Umgebungen, die ausschließlich aus einer festen Menge von Umgebungskomponenten bestehen, die nicht veränderlich ist, andere Umgebungen können aber durch Hinzufügen von Komponenten verändert werden. Im ersten Fall spricht man von einer *geschlossenen Umgebung.*

Geschlossene Umgebung: Unter einer *geschlossenen Umgebung* ist eine Software-Entwicklungsumgebung zu verstehen, deren Komponenten fest vorgegeben und unveränderlich sind.

Das Gegenstück zu einer geschlossenen Umgebung ist eine *offene Umgebung*. Sie kann zumindest in Teilen verändert werden.

Offene Umgebung: Unter einer *offenen Umgebung* ist eine Software-Entwicklungsumgebung zu verstehen, deren Zusammensetzung verändert werden kann. Diese Veränderung kann durch Austausch bestehender Komponenten oder durch Hinzufügen neuer Komponenten erfolgen.

Eine offene Umgebung kann je nach Art der verwendeten Komponenten zu sehr verschiedenen Lösungen führen. Diese Tatsache wird durch die folgenden zwei Begriffe beschrieben.

Homogene Umgebung: Eine *homogene Umgebung* ist eine Umgebung, deren Werkzeuge unter möglichst vielen Aspekten gut miteinander integriert sind.

In einer homogenen Umgebung sind z.B. die Benutzungsschnittstellen der integrierten Werkzeuge ähnlich aufgebaut. Homogen bedeutet aber auch, daß die Schnittstellen der Werkzeuge auf gleichen Konzepten basieren und damit die Integration entsprechend gut ermöglichen. Eine homogene Umgebung wird in der Regel durch a priori integrierte Werkzeuge erreicht.

Heterogene Umgebung: Eine *heterogene Umgebung* ist eine Umgebung, deren Werkzeuge zwar miteinander integriert werden können, wobei diese aber nicht notwendigerweise für diese Integration vorbereitet sind.

Eine heterogene Umgebung entsteht häufig bei einer a posteriori Integration von Werkzeugen und bringt in der Regel mehr Einschränkungen hervor als eine Umgebung, deren Werkzeuge a priori integriert wurden.

Beziehen sich die gerade eingeführten Begriffe im wesentlichen darauf, zu welchem Zeitpunkt der Entwicklung die Integrationsmaßnahmen geplant und durchgeführt wer-

den, so beziehen sich die folgenden Unterscheidungen auf die Art der Werkzeuge, die integriert werden sollen. Der erste Begriff benennt die Integration von Werkzeugen innerhalb der Prozeßausführung. Diese Integration wird in Anlehnung an [FNO92] als *Interprodukt-Integration* bezeichnet.

Interprodukt-Integration: *Interprodukt-Integration* ist die Integration von Werkzeugen in der Prozeßausführung ohne direkten Einfluß auf die Integration mit der Prozeßsteuerungskomponente, d.h. ohne direkten Einfluß auf die Synchronisation von Prozeßausführung und Modellausführung.

Interprodukt-Integration liegt nicht im Fokus dieser Arbeit. Sie wird hier zur Abgrenzung gegen den folgenden Integrationsbegriff eingeführt. Dieser wird, ebenfalls in Anlehnung an [FNO92], als *Prozeßintegration* bezeichnet.

Prozeßintegration: *Prozeßintegration* bezeichnet Maßnahmen zur Integration von Werkzeugen mit der Prozeßsteuerungskomponente. Ziel der Prozeßintegration ist das Ermöglichen einer möglichst weitreichenden Synchronisation von Modellausführung und Prozeßausführung.

Der Begriff Prozeßintegration, wie er hier eingeführt ist, darf nicht mit dem häufig verwendeten Begriff der Prozeßintegration, wie er z.B. in [Was89] und [TN92] definiert und verwendet wird, verwechselt werden. Im Anhang C bzw. im Anhang D wird auf die dortige Sichtweise eingegangen.

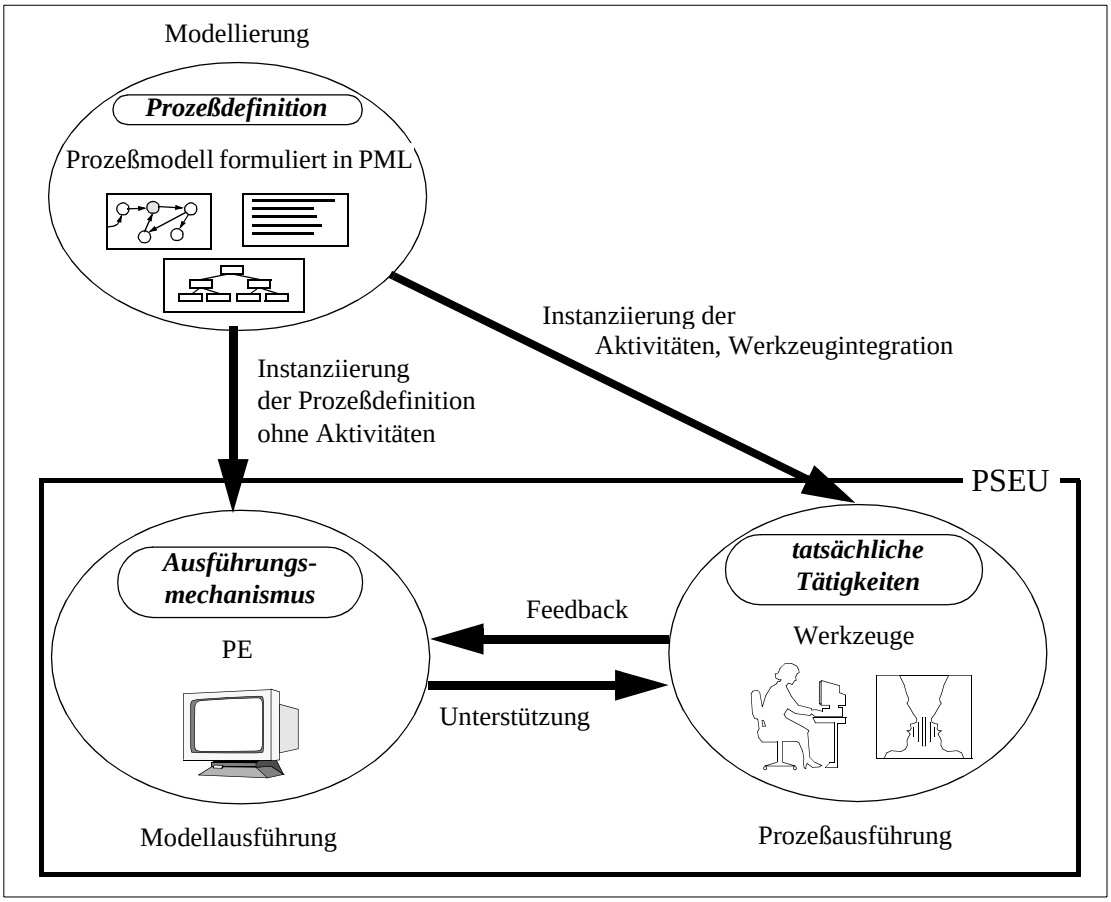


Abbildung 2.2 Begriffsdefinitionen im Kontext des Domänenmodells aus [DF94]

Im Rahmen der Prozeßintegration soll noch ein weiterer Begriff eingeführt werden. Dieser bezeichnet die Abhängigkeiten, die sich zwischen einem mit der Prozeßsteuerungskomponente zu integrierenden Werkzeug und der Prozeßmaschine ergeben. So kann die Ausführung eines Werkzeugs in der Prozeßausführung physikalische Daten verändern, die im Prozeßmodell referenziert werden. Die Veränderung solcher Daten kann eine Veränderung der Modellausführung zur Folge haben. Solche Daten werden als *prozeßrelevante Daten* bezeichnet.

Prozeßrelevante Daten: Daten der Prozeßausführung, die im Prozeßmodell referenziert werden und deren Veränderung möglicherweise eine Veränderung der gewünschten Modellausführung nach sich zieht, werden als *prozeßrelevante Daten* bezeichnet.

Damit sind die im Vorwege zu treffenden Definitionen vollständig. In Abbildung 2.2 werden einige der definierten Begriffe in den Kontext des Modells aus [DF94] eingeordnet.

2.2 Flexible Anbindung von Werkzeugen an offene, heterogene PSEU

Ist ein Prozeßmodell erstellt, so ist die Planung des Entwicklungsprozesses abgeschlossen. Zur Durchführung muß die Prozeßausführung die Funktionalität, die zur Entwicklung der zu erstellenden Software nötig ist, aber auch implementieren. Dabei gibt es zum einen die Möglichkeit, durch Kombinieren von Modellausführung und Prozeßausführung sehr gut integrierte (auch *hochintegrierte*), aber schwer zu erweiternde geschlossene Umgebungen einzusetzen. Zum anderen können auch offene, vollständig aus eigenständigen Programmen zusammengesetzte Umgebungen benutzt werden. Zwischen diesen beiden Extremen gibt es eine große Bandbreite von denkbaren Lösungen. Geschlossene Umgebungen haben dabei den Vorteil, daß durch die enge Kopplung alle Teile der Umgebung auf gemeinsamen Daten arbeiten können. Für PSEU ergibt sich insbesondere die Möglichkeit, die Daten der Modellausführung nicht von den Daten der Prozeßausführung zu trennen. Dadurch reduzieren sich die Möglichkeiten für Inkonsistenzen zwischen den beiden Prozeßsichten. Ist eine Umgebung hingegen aus unabhängigen Komponenten aufgebaut, so daß die Daten der Modellausführung getrennt von denen der Prozeßausführung gehalten werden, so ist potentiell die Gefahr von Inkonsistenzen gegeben. In der Regel sind daher insbesondere offene PSEU sehr viel schwieriger zu spezifizieren und realisieren, wenn die verwendeten Komponenten möglicherweise nicht auf den Einsatz in einer solchen Umgebung vorbereitet sind.

Ein wichtiger Grund, trotz der zu erwartenden Schwierigkeiten eine heterogene PSEU zu benutzen und dabei insbesondere auch nicht speziell vorbereitete Werkzeuge zu verwenden, ist z.B. die Notwendigkeit oder der Wunsch, bestehende Systeme bei der Einführung einer PSEU weiter- oder wiederzuverwenden. Entstehen kann eine solche Notwendigkeit z.B. aus Gründen des Investitionsschutzes oder weil keine geeigneten neuen Programme existieren, die in der Lage wären, bestehende Daten direkt zu verwenden oder zumindest vollständig und konsistent in neue zu konvertieren. In diesen Fällen wäre der Einsatz einer PSEU also erst in neuen Projekten möglich, und die bestehenden Altlasten müßten ohne die erweiterten Möglichkeiten weitergepflegt werden. Ein weiteres starkes Argument für den Einsatz von offenen Umgebungen ist aber auch die ansonsten drohende Gefahr der Bindung an einen Softwarehersteller. Mit

einer entsprechend flexiblen offenen Umgebung bleibt die Möglichkeit bestehen, Software bei einem Hersteller, dessen Software am besten zu den eigenen Anforderungen paßt, zu kaufen, Software, die am Markt nicht geeignet zu bekommen ist, selbst zu entwickeln oder sogar in verschiedenen Projekten in der gleichen (P)SEU Software verschiedener Hersteller einzusetzen.

Die Aufgaben, die sich bei der Erstellung einer offenen, heterogenen PSEU ergeben, sind mehrschichtig. Sie können in die Bereiche

- *Instanziierung der Aktivitäten* (vgl. Abbildung 2.2) und
- *Synchronisation* von Prozeß- und Modellausführung (Unterstützung und Feedback in Abbildung 2.2)

unterteilt werden. Beim Aufbau einer offenen, heterogenen PSEU stellt sich also zunächst die Aufgabe, die Aktivitäten zu instanziieren, d.h. geeignete Werkzeuge für die Umsetzung der Aktivitäten zu finden. Dieses ist zunächst noch kein Problem. Alle bereits existierenden PSEU, die offen aufgebaut sind, bieten auch Möglichkeiten, externe Werkzeuge aufzurufen. Das eigentliche Problem hinter diesem Instanziierungsschritt besteht darin, eine PSEU so zusammenzusetzen, daß die Steuerung des Prozesses als zentrale Aufgabe der PSEU durch den heterogenen Aufbau der Umgebung nicht verhindert wird. Hier enden die Fähigkeiten, die von existierenden PSEU angeboten werden, da sie, über den Aufruf von Werkzeugen hinaus, kaum Möglichkeiten anbieten, z.B. Feedback von den Werkzeugen aufzunehmen bzw. anzufordern.

Diese Arbeit beschäftigt sich daher damit,

- den Aufbau einer PSEU zu unterstützen und dabei sicherzustellen, daß die Instanziierung der Aktivitäten passend zum Prozeß vorgenommen wird und
- einen Aufbau für PSEU vorzuschlagen, der es erlaubt, die Steuerung der integrierten Werkzeuge durch die PE vorzunehmen.

Im Prozeß werden Aktivitäten spezifiziert, die auf einer hohen Abstraktionsebene beschreiben, was zu bestimmten Zeitpunkten im Prozeßablauf zu tun ist. Die Modellausführung enthält wiederum die Werkzeuge, mit denen die Implementierung der Aktivitäten erfolgen soll. Die Schwierigkeit im ersten oben genannten Punkt liegt darin, daß eine Abbildung von den Aktivitäten auf die Werkzeuge gefunden werden muß, die die Umsetzung der Aktivitäten sicherstellt. Gleichzeitig muß die Umgebung so aufgebaut werden, daß das Werkzeug in die Prozeßsteuerung eingebunden ist, d.h. daß die PE das Werkzeug unterstützt und das Werkzeug selbst Feedback an die PE liefert. Beispielsweise müßte also die Aktivität `bearbeiten` auf einen editor wie `vi` oder `notepad` abgebildet und dabei dafür gesorgt werden, daß der Prozeßablauf in der PSEU wie definiert abläuft, selbst wenn `vi` und `notepad` kein Feedback liefern. Um dies leisten zu können, muß zunächst erst einmal geklärt werden, welche Anforderungen überhaupt erfüllt werden müssen, um den definierten Prozeßablauf sicherstellen zu können. Existierende Ansätze helfen an dieser Stelle kaum. Es werden üblicherweise nur Möglichkeiten geboten, um einen Aufruf eines Werkzeugs durchzuführen. Was dann nicht mehr angeboten wird, sind Hilfen bei der Koordination der aufgerufenen Werkzeuge mit der Prozeßsteuerungskomponente. Durch das Fehlen von Koordination entstehen Umgebungen, die nur noch teilweise ihre Prozeßsteuerungsaufgabe wahrnehmen können.

Ist die Umgebungsinstanziierung vollständig, so kann die Ausführung des Prozesses beginnen. Hier kommt es zu Problemen der Synchronisation der Modellausführung mit der Prozeßausführung. Die Synchronisation ist so essentiell, weil die Steuerungskomponente einer PSEU nur dann korrekte Entscheidungen zur Steuerung des Prozesses treffen kann, wenn die von ihr benutzten Daten auch aktuell, d.h. konsistent mit dem tatsächlichen Prozeßzustand sind. Die Unabhängigkeit der Bereiche der Modellausführung und der Prozeßausführung hat Fernström in [Fer93b] durch ein Modell aus zwei unabhängigen Zustandsautomaten, die es zu koordinieren gilt, abstrahiert.

Mangelndes Feedback macht sich vornehmlich in Inkonsistenzen zwischen der Prozeßausführung und der Modellausführung bemerkbar. Probleme können aus der Inkonsistenz der Daten entstehen, wenn von der Modellausführung Annahmen gemacht werden, die durch einen laufenden Bearbeitungsschritt in der Prozeßausführung ungültig oder unvollständig gemacht werden. Die PSEU könnte beispielsweise die Information haben, daß zwischen zwei Dokumenten `A` und `B` keine Abhängigkeit besteht. Durch einen Bearbeitungsschritt könnte nun eine solche Abhängigkeit eingeführt werden, ohne daß die Daten der PSEU aktualisiert werden. Gibt es weiterhin eine Bedingung des Prozesses, die besagt, daß `A` nur in einen Zustand `S` kommen darf, wenn alle Dokumente vom Typ des Dokuments `B`, zu denen eine Abhängigkeit besteht, auch einen bestimmten Zustand erreicht haben, so wird aufgrund der fehlerhaften Daten nicht ausgewertet, in welchem Zustand sich `B` befindet, bevor `A` in den Zustand `S` gebracht wird.

Eine unzureichende Unterstützung macht sich dadurch bemerkbar, daß Veränderungen, die sich während der Modellausführung ergeben, nicht an die Prozeßausführung weitergegeben werden. So ist es z.B. denkbar, daß in der Prozeßausführung ein Werkzeug eingesetzt wird, das zur Bearbeitung einer Komponente eines zusammengesetzten Dokumentes eine Bearbeitung des gesamten Dokumentes startet. In dieser Situation muß das Werkzeug in der Lage sein, die durch das Prozeßmodell bestimmten Zugriffsrechte umzusetzen, da neben der eigentlich zu bearbeitenden Komponente des Dokuments auch noch alle anderen Komponenten implizit mitbearbeitet werden. Ist das Werkzeug dazu in der Lage, muß umgekehrt aber auch die Modellausführung die aktuellen Zugriffsrechte an das Werkzeug weitergeben oder zumindest deutlich machen, auf welcher Komponente das Werkzeug eigentlich aufgerufen worden ist. Im letzteren Fall kann das Werkzeug vereinfachend annehmen, daß ein verändernder Zugriff nur auf das eigentliche Ziel der Bearbeitung beschränkt werden kann.

Auf der technischen Ebene stellt sich bei der Integration von Werkzeugen in eine PSEU die Frage, welche Werkzeuge überhaupt zusammen mit der Umgebung eingesetzt werden können. Welche Basismechanismen werden unterstützt, z.B. nur Kommandozeilen-Werkzeuge, Werkzeuge auf der Basis von ToolTalk, SoftBench etc., CORBA-basierte Werkzeuge, ... Falls der notwendige Basismechanismus unterstützt wird, ist noch zusätzlich die Einhaltung eines bestimmten Protokolls notwendig? Dem Angebot sind keine Grenzen gesetzt, da es keine allgemeingültigen Standards gibt, nur kann die PSEU natürlich nicht alles anbieten. Es wäre also sinnvoll, wenn die PSEU in jedem Umfeld an die benötigten Basismechanismen und Protokolle adaptiert werden könnte. Auch hier muß beim Aufbau der PSEU eine Unterstützung in der Art gegeben werden, daß klar ist, wie Werkzeuge so in die PSEU eingebunden werden können, daß, wie oben gefordert, die Steuerung des Prozesses möglich ist.

Die weiter oben erwähnten Defizite bereits existierender Ansätze lassen sich darauf zurückführen, daß das Hauptinteresse in der Forschung bisher in den Bereichen der Modellierung und der Modellausführung angesiedelt war (siehe dazu z.B. [CLJ91], [Lon94], [GJ96]). Dabei sind verschiedene Prozeßmodellierungssprachen (z.B. *ESCAPE* [Jun95], *PML* [BGR+94], *SLANG* [SPA95a], *SOCCA* [EG94], weitere siehe z.B. in [ABGM92] und [FKN94]) und auch prozeßgesteuerte Software-Entwicklungsumgebungen (z.B. *Arcadia* [TSY+88], *Dynamite* [HJKW96], *Marvel* [KPBS93], *Melmac* [Gru91b], *Merlin* [JPSW94], *ProcessWeaver* [Fer93a], *Spade* [BBFL94], weitere siehe z.B. in [FKN94], [GJ96]) entstanden. Daß die Prozeßausführung dabei ebenfalls instanziiert werden muß, wurde bisher kaum betrachtet. Auch im Modell von Dowson und Fernström fehlt eine solche Interaktion.

In einigen Veröffentlichungen wurde darauf hingewiesen, daß die Tätigkeiten, die in der Modellausführung referenziert werden, in der Prozeßausführung auch an manuelle bzw. rechnerunterstützte „Implementierungen" gebunden werden müssen (z.B. [Lon94], [Was89], [Mon94], [Emm95]). Es wurde auch darauf hingewiesen, daß die Anbindung der Prozeßausführung an die anderen Teilbereiche komplexe Fragen aufwirft (vgl. z.B. [SW94], [ACF95], [EF96]). Arbeiten, die untersuchen, welche Abhängigkeiten sich zwischen Modellausführung und Prozeßausführung ergeben, wie die Unterstützung der im Bereich der Prozeßausführung liegenden Tätigkeiten und deren Feedback in einer PSEU in ein allgemeines Konzept gebracht werden kann und wie die Prozeßbeschreibung als Vorgabe der Instanziierung der Prozeßausführung berücksichtigt werden kann, hat es aber in der Vergangenheit bis auf wenige Ansätze (vgl. z.B. [Dei93], [Poh96], [VK95]) kaum gegeben. Diese Ansätze werden in Kapitel 5 näher besprochen.

2.3 Aufbau der Arbeit

In dieser Arbeit werden durchgängige Konzepte entwickelt, mit denen es möglich ist, eine flexible, prozeßgesteuerte Software-Entwicklungsumgebung zusammenzustellen. Die Anwendung dieser Konzepte wird am Beispiel einer konkreten PSEU gezeigt.

Nach der Einleitung (Kapitel 1) wurden in diesem Kapitel (Kapitel 2) einige Grundbegriffe definiert, die im Laufe der Arbeit benötigt werden. Darauf aufbauend wurde das Thema der Arbeit dargestellt und der Zusammenhang zu anderen Arbeiten aufgezeigt.

Das dritte Kapitel beschreibt ein Szenario, in dem gängige Probleme bei der Werkzeugintegration in eine flexible, heterogene PSEU auftreten. Aus diesem Szenario werden die Beispiele der Arbeit gewählt. An das Szenario schließen sich die Kapitel 4 bis 7, die den Kern dieser Arbeit bilden, mit den folgenden Inhalten an:

- Anforderungen an die Integration von Werkzeugen in eine PSEU (Kapitel 4)

  Es wird eine detaillierte Untersuchung durchgeführt, an welchen Stellen es bei der Anbindung von Werkzeugen an eine PSEU zu Problemen kommen kann. Diese Betrachtung ist allgemein und unabhängig von einer konkreten PSEU gehalten. Die Auswertung liefert eine Liste von Anforderungen, die in einer heterogenen PSEU erfüllt sein müssen, um die identifizierten Probleme vermeiden zu können.

- Beurteilung existierender Lösungsansätze (Kapitel 5)

  In diesem Kapitel werden allgemeine Integrationstechniken auf ihre Verwendbarkeit zur Lösung der in Kapitel 4 identifizierten Probleme untersucht. Weiterhin wird für einige Vertreter existierender PSEU betrachtet, inwieweit sie, gemessen an den Anforderungen aus Kapitel 4, bereits geeignete Lösungen anbieten.

- Vorstellung von Konzepten zur Erstellung von flexiblen, heterogenen PSEU unter Berücksichtigung der Anforderungen aus Kapitel 4 (Kapitel 6)

  Es werden Konzepte zur Lösung der Probleme entworfen, die in Kapitel 4 identifiziert wurden. Diese werden am konkreten Beispiel einer existierenden PSEU auch evaluiert. Die Konzepte bestehen aus verschiedenen Kernelementen:

  - Ergänzungen der Prozeßmodellierung

  Hier wird dargestellt, welche Konzepte in einer PML notwendig sind, so daß die Aktivitäten eines Prozesses so spezifiziert werden können (entsprechend der Anforderungen aus Kapitel 4), daß sie auf die Werkzeuge abgebildet werden können. Dadurch wird gewährleistet, daß sich die Werkzeuge der Prozeßausführung tatsächlich am definierten Prozeß orientieren.

  - Konzept zur Spezifikation der Abbildung von Aktivitäten auf Werkzeuge

  Es wird eine Spezifikation als Bindeglied zwischen Werkzeugen und Aktivitäten eingeführt, die dazu benutzt wird, die Abbildung der Aktivitäten auf die Werkzeuge zu beschreiben.

- Konzepte zur Integration von Werkzeugen in eine PSEU auf technischer Ebene, wobei der Integrationsaufwand durch Generierung von Komponenten minimiert wird (Kapitel 7)

  - Umsetzung der Modellierungskonzepte in der Implementierung

  Es werden Konzepte entwickelt, die es erlauben, Werkzeuge um eine Prozeßsteuerungskomponente herum zu einer Entwicklungsumgebung zu gruppieren. Auf diese Weise werden Prozeßmodelle an konkrete Werkzeuge angebunden, so daß die Kernkomponenten (z.B. Prozeßmaschine) beim Austausch von Werkzeugen unverändert bleiben können, so daß Werkzeuge auch für verschiedene Projekte/Prozesse nur genau einmal eingebunden werden müssen und so daß ein Prozeß mit einer Spezifkation unverändert in verschiedenen PSEU ausgeführt werden kann.

  - Minimieren des Aufwands zur Erstellung einer PSEU

  Die die Kernkomponenten und die Werkzeuge verbindenden Elemente werden in einem Generierungsschritt aus der erweiterten Spezifikation gewonnen. Dadurch wird der Aufwand bei der Einbindung von Werkzeugen bei einem entsprechend einmal vorbereiteten Werkzeug auf ein Minimum reduziert.

  - Architektur

  Es wird eine allgemeine Architektur definiert, die die Modellierungs- und Implementierungskonzepte einbezieht.

Das letzte Kapitel faßt die Ergebnisse der Arbeit zusammen und diskutiert mögliche Erweiterungen.

Im Anhang werden dann u.a. noch Überblicke über generelle Modelle zur Beschreibung von Integration (Anhang C) und über einige Beispiele für Anwendungen der Integrationstechniken aus dem fünften Kapitel gegeben (Anhang D).

3

SZENARIO

Eine Softwarefirma hat im Rahmen der Implementierung eines Krankenhausinformationssystems den Zuschlag zur Realisierung eines Teilprojekts bekommen. Der Auftraggeber ist Träger mehrerer Krankenhäuser. Die Betreuung des Projekts seitens des Auftraggebers erfolgt durch das von ihm betriebene Rechenzentrum. Die neu zu implementierende Software ist in eine bestehende Software- und Hardwarelandschaft einzupassen. Einige Anforderungen ergeben sich dabei insbesondere dadurch, daß die fertiggestellte Software im Rechenzentrum des Auftraggebers in eigener Zuständigkeit gewartet werden soll.

Teil des Auswahlverfahrens ist eine Qualitätsvorgabe zum Entwicklungsprozeß durch den Auftraggeber. Um diesen zu überzeugen, wurde eine Prozeßmodellspezifikation eingereicht, nach der das Projekt durchgeführt werden soll. Nach kleinen Änderungswünschen, die den geforderten Qualitätsstandard sichern sollen, hat der Auftraggeber dem Prozeßmodell zugestimmt.

Durch das Prozeßmodell sind unter anderem die Dokumenttypen der vertraglich festgelegten Zwischen- und Endergebnisse der Entwicklung vorgegeben. Zusätzlich hat der Auftraggeber Formate für die abzugebenden Dokumente definiert, um sicherzustellen, daß die Dokumente im eigenen Haus auch einbindbar sind.

Der Prozeß sieht vor, daß in einem ersten Schritt eine Anforderungsanalyse vorgenommen wird. Diese soll als erstes Teilergebnis dokumentiert und dann mit dem Auftraggeber abgestimmt werden. Im nächsten Schritt sollen die in der Anforderungsanalyse festgehaltenen Rahmenbedingungen in ein Pflichtenheft umgesetzt werden. Da das Pflichtenheft Bestandteil des Vertrags ist, soll es zunächst im Team mindestens einen Reviewzyklus durchlaufen, bevor es dem Auftraggeber vorgelegt wird. Dieser gibt es mit Änderungswünschen zurück oder gibt es für den weiteren Prozeß frei. Es wird weiterhin festgelegt, daß sich das Pflichtenheft in die Teile Systembeschreibung (Hardware, Betriebssystem, Methodenwahl, Programmiersprache, ...), Funktionsumfang (SA-Diagramme, Text, ...) und Front-End Beschreibung (Art der Oberfläche, Menüanordnung, ...) untergliedert. Nachdem auf dieser Basis der endgültige Auftrag festgelegt wird, wird eine Architektur entworfen. Die darin identifizierten Module werden nun in der festgelegten Programmiersprache implementiert. Jedes der Module wird einem

Modultest unterzogen, bevor in einem nächsten Schritt die Module integriert werden. Diese Integration kann mehrere Stufen durchlaufen.

Aus der umgangssprachlichen Beschreibung werden mehrere Dokumenttypen abgeleitet, die im Rahmen der Entwicklung zu erstellen sein werden: `Anforderungsanalyse`, `Pflichtenheft` (unterteilt in `Systembeschreibung`, `Funktionsbeschreibung` und `Benutzungsschnittstelle`), `Architektur`, `Quelltextdokument` (unterteilt in `Interface` und `Implementierung`), `Objektdatei`, `Testdokument` (unterteilt in `Testbeschreibung` und `Ergebnisdokumentation`) und `Programm`.

Zur Beschreibung der Abhängigkeiten zwischen den Dokumenten werden Beziehungstypen (kurz Beziehungen) zwischen den Dokumenttypen eingeführt. Zwischen den Dokumenttypen `Architektur` und `Quelltextdokument` wird z.B. die Beziehung `generiert_aus` eingeführt. Diese Beziehung soll es zur Laufzeit des Projekts ermöglichen, im Falle einer Änderung der Architektur die betroffenen Quelltextdokumente zu bestimmen und sicherzustellen, daß diese überarbeitet werden. Neben weiteren solcher „einfachen“ Beziehungen werden auch Aggregationsbeziehungen eingeführt. So wird z.B. direkt modelliert, daß ein `Quelltextdokument` aus einem `Interface` und einer `Implementierung` zusammengesetzt ist. Zwischen diesen beiden Dokumenttypen wird zusätzlich die Beziehung `benutzt` eingeführt, wobei die Leserichtung ist, daß eine Implementierung Interfaces benutzen kann.

Auf den Dokumenttypen basierend wird der geplante Ablauf der Entwicklung in zu erledigende Aktivitäten umgesetzt. So werden z.B. allen manuell zu erstellenden Dokumenttypen die Aktivitäten `bearbeiten`, `lesen` und `drucken` zugeordnet. Die Aktivitäten werden zusätzlich in eine Ordnung gebracht, um z.B. sicherzustellen, daß nicht mit der Formulierung einer Architektur begonnen wird, bevor das Pflichtenheft genehmigt ist und daß Module, die bereits getestet sind, nicht mehr ohne weiteres bearbeitet werden können. Die automatisch durchführbaren Aktivitäten (z.B. das Übersetzen eines Implementierungsmoduls) werden so in diese Ordnung eingebettet, daß sie geeignet aufgerufen werden. Es werden weiterhin Rollen festgelegt, in denen die Aktivitäten auf den Dokumenten von den einzelnen Entwicklern durchgeführt werden können.

Das Team, das nun zusammengestellt wird, um die Durchführung des Projekts zu übernehmen, ist auf mehrere Standorte der Firma verteilt. Weiterhin pflegt die Firma eine Kultur, die es den Mitarbeitern ohne direkten Kundenkontakt erlaubt, bis auf einen Tag in der Woche und präsenzpflichtige Termine, wie z.B. Gruppenbesprechungen, zuhause zu arbeiten. Die Infrastruktur dazu wird allerdings nur auf der Seite der Firma geschaffen. Die an diesem Angebot interessierten Mitarbeiter müssen sich um ihre heimische Infrastruktur selbst kümmern und sie auch selbst bezahlen.

Um die Einhaltung des Prozeßmodells sicherzustellen und diese bei den kommenden Projektreviews durch den Auftraggeber auch belegen zu können sowie um den Fortgang des Projekts zu dokumentieren, soll eine PSEU eingesetzt werden. Das Projekt mit einer geschlossenen Umgebung zu realisieren, ist aus mehreren Gründen unmöglich. Erstens ist es mit den verfügbaren Umgebungen nicht möglich, die Vorgaben bezüglich der abzugebenden Dokumentformate einzuhalten. Zweitens müßte die Umgebung an allen Standorten, an denen Entwickler arbeiten, auch verfügbar sein. Dies ist insbesondere wegen der großen Bandbreite der eingesetzten Betriebssysteme

nicht zu erreichen (Standort A: Sun Solaris 2.5, Standort B: Windows 2000, Entwickler zuhause: Linux und Windows XP). Ein weiteres Problem ergibt sich bei der Evaluation der in Frage kommenden Umgebungen beim Funktionsumfang. Ein kurzer Vergleich der Anforderungen der vor der Evaluation in der Firma gelaufenen Projekte zeigt einen immer wiederkehrenden Kern an benötigten Funktionen auf, doch ebenso regelmäßig gibt es einen wechselnden Anteil von Funktionen, z.B. durch wechselnde Testverfahren und verschiedene eingesetzte Programmiersprachen.

Die Firma wählt daher einen anderen Weg. Sie definiert für das Projekt eine speziell zugeschnittene PSEU, die später auch in anderen Projekten benutzt werden soll. Alle Komponenten außer der Prozeßsteuerung (das sind z.B. Editoren, Übersetzer, Testwerkzeuge etc.) werden entsprechend den konkreten Projektbegebenheiten ausgewählt und angepaßt. Insbesondere wird versucht, möglichst viele der bereits vorhandenen Entwicklungswerkzeuge einzubinden.

Der Aktivität `bearbeiten` des Dokumenttyps `Pflichtenheft` wird ein Textverarbeitungssystem zugeordnet, das in der Lage ist, ein Dokument aus einzelnen Teilen zu aggregieren. Ein solches aggregiertes Dokument besteht aus Dokumenten, die die eigentlichen Inhalte enthalten, und einem übergeordneten Dokument, das ausschließlich den Zusammenhang der anderen Dokumente beschreibt. Alle Dokumente werden als einzelne Dateien abgelegt. Durch dieses Vorgehen bleiben die verschiedenen Teile des Pflichtenhefts erkennbar und können einzeln bearbeitet werden. Als physikalische Struktur wird dem Pflichtenheft der beschreibende Dokumenttyp des Textverarbeitungssystems zugeordnet. Die einzelnen Teile des Pflichtenhefts werden durch tatsächlich Text enthaltende Bestandteile realisiert. Für die Aktivität `bearbeiten` der Architektur wird ein spezielles Werkzeug benutzt, das seine Daten in einer Datenbank ablegt. In diesem Dokument werden die späteren Implementierungsmodule auf abstrakter Ebene referenziert. Die späteren Programmdokumente werden durch Ausführung einer Aktivität `generieren` auf dem Architekturdokument erzeugt. Die gewählte Programmiersprache ist C. Entsprechend generiert die Funktion des Werkzeugs aus der Architekturbeschreibung je Modul eine Headerdatei und eine C-Datei. Die Aktivität `generieren` muß in dem Architektur-Werkzeug manuell durch einen Menüpunkt gestartet werden. Dabei werden nur die Bestandteile der Programmdokumente erzeugt. Es existiert damit keine reale Entsprechung zu den modellierten Programmdokumenten, d.h. keine Aggregation aus `Interface` und `Implementierung`. Die Aktivität `bearbeiten` auf den C-Dateien wird jeweils durch einen einfachen Texteditor realisiert. Auch die Aktivität `bearbeiten` des Dokumenttyps `Testdokument` wird durch einen einfachen Texteditor realisiert. Die entsprechenden Aktivitäten der Bestandteile des Testdokuments (Testbeschreibung und Ergebnisdokumentation) werden nicht getrennt realisiert. Die Bearbeitung erfolgt durch die Bearbeitung des Testdokuments. Auch die anderen Aktivitäten werden geeignet mit Werkzeugen belegt.

Bevor das Projekt wirklich starten kann, wird im letzten Schritt das Modell instanziiert. Dazu werden konkrete Entwickler benannt, und ihnen werden Rollen im Projekt zugeordnet. Über die Rollen und Entwicklernamen werden zur Laufzeit des Projekts auch die Zugriffsrechte überwacht und die Aufträge, welche Aktivitäten als nächste durchzuführen sind, vorgenommen. Dann wird als Startbelegung der Dokumentmenge ein noch leeres Dokument vom Typ `Anforderungsanalyse` erzeugt. Nun kann die Arbeit mit der Umgebung aufgenommen werden.

Bei der Definition von solchen spezialisierten PSEU entstehen im Laufe der Zeit z.T. wiederverwendbare Blöcke, die dann in späteren Projekten weiterbenutzt werden sollen. Diese sind aber nicht statisch und können damit auch sehr einfach angepaßt werden, z.B. an neue Produktreleases der eingesetzten Komponenten. Die Firma setzt in dieser Umgebung Komponenten ein, die möglichst optimal zur benötigten Funktionalität passen, die aber nicht unbedingt speziell für den Einsatz in einer solchen Umgebung vorgesehen sind.

Es entstehen mit dieser Lösung aber auch Probleme. Zunächst einmal ist die Einbindung eines Werkzeugs jeweils Einzelarbeit, obwohl sich die grundsätzlichen Techniken bei der Einbindung ähnlicher Werkzeuge durchaus wiederholen. Diese Arbeit muß außerdem für jedes Betriebssystem einzeln wiederholt werden. Eine projektweite (also für alle Produktionsstätten gültige) Definition von Schnittstellen wird nicht vorgenommen. Noch größere Probleme als bei der Vorbereitung der konkreten Umgebung ergeben sich im laufenden Betrieb. Dadurch, daß keine Konzepte und Mechanismen vorliegen, die es den Entwicklern erlauben, die Fortschritte ihrer Arbeit leicht mit dem Modell abzugleichen, kommt es oft zu Situationen, in denen die Prozeßsteuerungskomponente falsche Vorgaben macht.

In dieser Arbeit wird gezeigt, daß die oben beschriebenen Probleme entstehen, weil die aktuell verfügbaren Prozeßmodellierungssprachen für die Definition flexibel aufgebauter PSEU nicht ausreichend ausgestattet sind. Außerdem fehlen so aufgebauten PSEU weitgehend Abstimmungsmöglichkeiten zwischen den Werkzeugen und der Prozeßsteuerung. In den Kapiteln 6 und 7 werden Konzepte entwickelt, mit denen sich Modelle und Umgebungen spezifizieren lassen, in denen diese Problempunkte beseitigt sind.

4

PROBLEME FLEXIBLER, HETEROGENER PSEU

Bei der Erstellung einer heterogenen PSEU ergeben sich drei Problembereiche, die im wesentlichen voneinander unabhängig sind: Zum einen können Probleme bei der Instanziierung der PSEU auftreten. Dabei ist es wichtig, daß die Semantik der Werkzeuge der Semantik der Aktivitäten entspricht (Kapitel 4.1). Zum zweiten gibt es bei der Anbindung der Werkzeuge technische Probleme. Hier geht es darum, daß die Prozeßsteuerungskomponente und die Werkzeuge eine gemeinsame Kommunikationsbasis finden müssen (Kapitel 4.3). Zum dritten geht es um Probleme innerhalb der PSEU, die zur Laufzeit zu erwarten sind. Diese entstehen dadurch, daß Projektinformationen in der Modell- und der Prozeßausführung redundant gehalten werden, wodurch bei Änderungen Inkonsistenzen entstehen können (Kapitel 4.2). In der Abbildung 4.1 wird in einer Übersicht dargestellt, zu welchen Zeitpunkten bzw. an welchen Stellen die Probleme aus den obigen Problembereichen auftreten.

In den folgenden Kapiteln werden die Probleme identifiziert, die sich ergeben, und es werden Anforderungen zur Vermeidung oder Behandlung der identifizierten Probleme gestellt. Die technische Anbindung wird dabei in der Darstellung ans Ende gestellt, da sie durch die Anforderungen aus Kapitel 4.2 beeinflußt wird.

4.1 Probleme bei der Instanziierung der PSEU

Probleme bei der Instanziierung der PSEU treten auf, wenn die Werkzeuge, die als Implementierung von Aktivitäten in die Umgebung integriert werden sollen, die Spezifikation der jeweiligen Aktivität nicht erfüllen. Sie entstehen in der Hauptsache dadurch, daß ein Modell unabhängig von einer konkret vorhandenen Umgebung erstellt wird und in einer solchen Umgebung dann keine Instanziierungsmöglichkeiten bestehen, die die Spezifikation aus dem Prozeßmodell erfüllen.

Der Zeitpunkt, zu dem diese Probleme erkannt werden müssen, ist der Zeitpunkt, zu dem die Umgebung zusammengestellt wird. Sie müssen vor Projektbeginn beseitigt werden. Geht dies nicht, so müssen sie über die gesamte Laufzeit des Projekts ausgeglichen werden. Eine Möglichkeit, diese Probleme zu beheben, bestünde darin, die

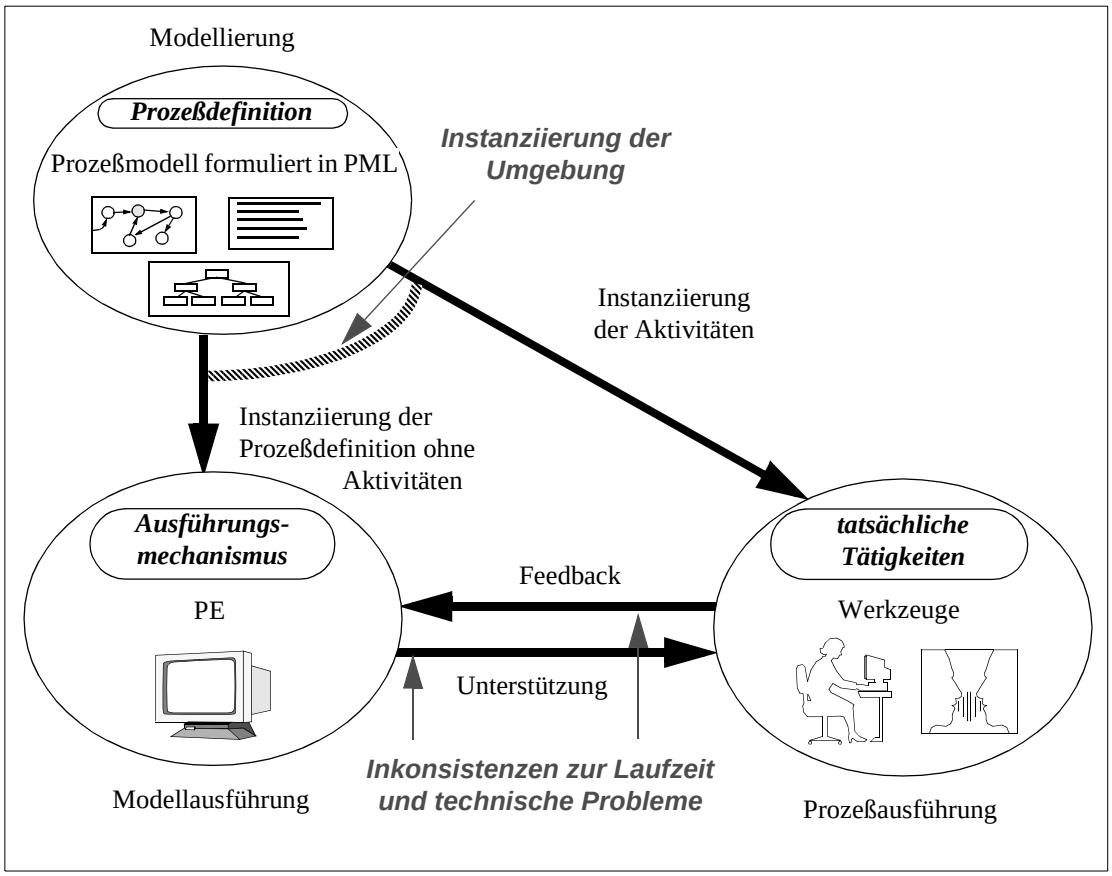


Abbildung 4.1 Übersicht über Problembereiche in einer heterogenen PSEU

Modellierung in Abhängigkeit von einer konkret vorhandenen Umgebung aufzubauen. Dies kommt jedoch aus zwei wichtigen Gründen nicht in Frage: Zunächst einmal ist es das Ziel einer Modellierung, Abläufe mit optimalen Entwicklungsergebnissen zu formulieren (z.B. Fehlerzahl gegen 0, kurze Entwicklungszeit, keine Nacharbeiten wegen mangelnder Koordination von Entwicklern, ...). Erst wenn diese Abläufe einmal formuliert sind, kann man abwägen, ob man vom Optimum abweichen will, indem man die vorhandenen Werkzeuge unverändert einsetzt und notfalls das Modell anpaßt, oder ob man andere Werkzeuge einsetzen kann. Will man zweitens eine umfangreiche Modellierung rechtfertigen, so muß diese Modellierung wiederverwendbar sein, d.h. unabhängig von einer konkret vorhandenen Umgebung formuliert werden.

Probleme bei der Instanziierung der PSEU betreffen die Semantik der Aktivitäten und der Werkzeuge. Um deren Semantik formal in der Spezifikation zu notieren, wird sie hier wie folgt unterteilt:

- Signatur
- Leistungsumfang
- Zustandsmodell

Die Signatur legt fest, welche Dokument- und Beziehungstypen als Eingabe benötigt bzw. als Ausgabe erzeugt werden. Das Zustandsmodell beschreibt, von welchem Zustand in welche Folgezustände das Dokument, auf dem die Aktivität aufgerufen wurde, überführt wird. Der Leistungsumfang der Aktivität wird dadurch beschrieben, daß spezifiziert wird, welche Dokument- und Beziehungstypen gelesen, geschrieben,

erzeugt oder gelöscht werden. Bezüglich der hier in dieser Arbeit vorgenommenen Betrachtungen werden keine über diese Punkte hinausgehenden Aspekte der Semantik von Aktivitäten und Werkzeugen vorgenommen. Die Darstellung der Semantik ist damit mit den drei genannten Punkten abgeschlossen. Eine vollständige Semantikdefinition von Aktivitäten wird nicht vorgenommen, da eine solche nicht notwendig ist, um eine Prozeßsteuerung zu gewährleisten.

In den folgenden Kapiteln werden die oben eingeführten Teile der Spezifikation auf die Probleme, die im Zuge der Instanziierung der Umgebung auftreten können, untersucht.

4.1.1 Signatur

Differenzen zwischen der Signatur der Aktivitäten und der Signatur der Werkzeuge entstehen dann, wenn Werkzeuge eingesetzt werden sollen, die nicht die bei den Aktivitäten spezifizierten Dokumenttypen als Eingabe benutzen oder nicht die dort vorgegebenen Dokumenttypen erzeugen. Dabei kann es zum einen vorkommen, daß mehr bzw. weniger Dokumenttypen benötigt bzw. erzeugt werden, zum anderen aber auch, daß die spezifizierten Dokumenttypen verschieden sind. Gibt es transiente Ausgaben, so sind sie zu den Ausgabedokumenten zu rechnen, sobald sie Angaben beeinhalten, die zur weiteren Prozeßausführung benötigt werden. Dabei ist es egal, ob sie persistent gemacht und damit als Dokument abgelegt werden oder nicht.

Beispiel: Im Szenario aus Kapitel 3 wird die Aktivität `generieren` auf dem Dokumenttyp `Architektur` durch ein Werkzeug implementiert, das nur die Komponente `Interface` des Dokumenttyps `Quelltextdokument` erzeugt. Dies weicht von der Spezifikation der Aktivität ab, da diese vorsieht, auch einen Rumpf der zweiten Komponente `Implementierung` zu erzeugen. Auch die Prozeßabläufe werden davon beeinflußt, da in der Modellausführung fehlerhaft angenommen wird, daß das Implementierungsdokument existiert, es aber tatsächlich noch erzeugt werden muß.

Um solche Situationen beheben zu können, ist es zunächst einmal nötig, daß zum einen die Signatur einer Aktivität im Prozeßmodell spezifiziert wird und zum anderen auch für die Werkzeuge eine solche Spezifikation existiert. Die Spezifikation der Signatur von Aktivitäten ist insbesondere dann wichtig, wenn Prozeßmodelle wiederverwendbar sein und damit auch von verschiedenen Personen benutzt, d.h. insbesondere geeignet instanziiert, werden sollen. Die Spezifikation der Signatur der Werkzeuge ist notwendig, um sicher sagen zu können, welche Dokumenttypen das Werkzeug benötigt/erzeugt. Eine solche Spezifikation ist für die Werkzeuge üblicherweise nicht vorhanden.

Fallen bei der Instanziierung Differenzen auf, so gibt es zwei Optionen zur Anpassung der Ein-/Ausgabedokumente. Die eine besteht darin, das Prozeßmodell zu ändern oder ein anderes Werkzeug zu benutzen. Die andere Option ist, das Werkzeug so zu kapseln, daß es doch zur Aktivität paßt. Die Wahl der richtigen Option auf dieser Ebene liegt nicht im Fokus der Betrachtung.

Die Signaturen von Aktivitäten und Werkzeugen müssen also bei der Instanziierung aufeinander abgestimmt werden, woraus sich die folgenden Anforderungen ableiten:

Spezifikation der Signatur Um feststellen zu können, ob ein Werkzeug als Implementierung einer Aktivität eingesetzt werden kann, muß es zwei Elemente geben: a) Die Signatur einer Aktivität muß im Prozeßmodell spezifiziert sein. b) Die Spezifikation der Signatur eines Werkzeugs, das eine Aktivität implementieren soll, muß vorliegen.

Relation der Signaturen Es muß möglich sein, eine Relation zwischen den Elementen der Signatur eines Werkzeugs und den Elementen der Signatur der implementierten Aktivität anzugeben. Dabei muß jedes Element der Signatur der Aktivität mit mindestens einem Element der Signatur des Werkzeugs in Relation stehen.

Wichtig ist, daß die Anforderung nicht besagt, daß die geforderten Spezifikationen alleine die Entscheidung auf die Eignung eines Werkzeugs erlauben. In den nächsten Kapiteln werden weitere Anforderungen formuliert, die dann gemeinsam mit dieser Anforderung betrachtet werden müssen.

4.1.2 Leistungsumfang

Bei der Instanziierung eines Prozeßmodells kann sich zeigen, daß ein zur Implementierung einer Aktivität vorgesehenes Werkzeug die Spezifikation des Leistungsumfangs der Aktivität nicht erfüllt. Der Leistungsumfang eines Werkzeugs ist mit dem Leistungsumfang einer Aktivität identisch, wenn er die gleichen lesenden Zugriffe und Veränderungen in der Prozeßausführung erlaubt, wie sie durch die Aktivität in der Modellausführung möglich sind. Der Leistungsumfang des Werkzeugs ist kleiner als der der Aktivität, falls im Werkzeug Zugriffe oder Veränderungsmöglichkeiten im Vergleich zu den Zugriffen oder Veränderungsmöglichkeiten der Aktivität fehlen. Ist der Leistungsumfang eines Werkzeugs kleiner als der der zu implementierenden Aktivität, so führt dies dazu, daß der vorgesehene Umfang und der Ablauf des Prozeßmodells nicht mehr sichergestellt ist. Es wäre z.B. möglich, daß einzelne Teile des Prozesses nie ausgeführt werden können. Ist der Leistungsumfang eines Werkzeugs größer als für die entsprechende Aktivität im Modell vorgesehen, so überlappt er möglicherweise mit den Inhalten anderer Aktivitäten. In diesem Fall kann es zu Verletzungen der vorgegebenen Prozeßabläufe kommen, falls auf diese Weise Tätigkeiten ausgeführt werden, die nach dem Prozeßmodell in derzeit nicht erlaubten Aktivitäten liegen sollten.

Beispiel: Im Szenario aus Kapitel 3 ist allen manuell zu verändernden Dokumenttypen die Aktivität `bearbeiten` zugeordnet worden. Diese Aktivität ist dann im Falle des Dokumenttyps `Implementierung` auf einen Texteditor abgebildet worden. Handelt es sich dabei um einen Editor mit heutzutage üblichen Fähigkeiten, so besitzt er mehr Möglichkeiten als die Bearbeitung eines einzelnen Dokuments. Beispiele für weitergehende Fähigkeiten sind die Erzeugung und das Öffnen weiterer Dokumente. Dabei ergibt sich auf der Ebene der Signatur noch kein Unterschied, so daß die Abweichung auf der Ebene der Signatur noch nicht festgestellt werden kann. Betrachtet man jedoch den Leistungsumfang, so ist sie erkennbar.

In diesem Beispiel soll eine Aktivität durch ein Werkzeug implementiert werden, das einen größeren Leistungsumfang besitzt als die zugehörige Aktivität. Diese Situation entsteht im Gegensatz zu den bisher dargestellten nur teilweise aufgrund der Trennung von Modellierung und Instanziierung. Es kommt vielmehr daher, daß die Aktivitäten in

der Modellierung typischerweise nur kleine Aufgabenbereiche abdecken. Auf der Werkzeugebene hingegen ist es immer attraktiv, solche Werkzeuge zu benutzen, die mehrere Tätigkeiten miteinander verbinden. Ein typisches Beispiel dafür sind Texteditoren, die mehr und mehr kleine Bearbeitungsumgebungen für sich selbst sind und deren Leistungsfähigkeit bei weitem nicht bei der reinen Bearbeitung eines einzelnen Textes endet. Eine Aktivität `bearbeiten`, wie sie im Beispiel eingeführt worden ist, deckt damit fast sicher einen kleineren Bereich als der zur Implementierung verwendete Editor ab. Wenn man eine solche Situation bei der Instanziierung bewußt zuläßt, so muß zur Laufzeit sichergestellt werden, daß die Änderungen, die das Werkzeug vornimmt, auch prozeßkonform sind. Dieser Aspekt wird in Kapitel 4.2.1.3 genauer betrachtet.

Wenn der Leistungsumfang eines Werkzeugs kleiner ist, als der Leistungsumfang der entsprechenden Aktivität, so kann dies nicht durch Maßnahmen zur Laufzeit ausgeglichen werden. Als Konsequenz bleibt also nur die Korrektur des Modells und/oder eine Veränderung der Wahl des Werkzeugs. Daher wird dieser Punkt in der vorliegenden Arbeit nicht mehr weiter betrachtet.

Für den Leistungsumfang von Aktivitäten und Werkzeugen muß also zum Zeitpunkt der Instanziierung folgendes sichergestellt sein:

Spezifikation des Leistungsumfangs Um erkennen zu können, ob ein Werkzeug als Implementierung einer Aktivität eingesetzt werden kann, muß es zwei Elemente geben: a) Der Leistungsumfang einer Aktivität muß im Prozeßmodell spezifiziert sein. b) Die Spezifikation des Leistungsumfangs eines Werkzeugs, das eine Aktivität implementieren soll, muß vorliegen.

Abbildung des Leistungsumfangs Bei der Instanziierung muß es möglich sein, den Leistungsumfang der Aktivitäten so auf den Leistungsumfang der Werkzeuge abzubilden, daß die durch die Relation der Signaturen von der Signatur der Aktivitäten auf die Signatur der Werkzeuge abgebildeten Dokumenttypen in der gleichen Weise gelesen bzw. verändert werden.

Ein gesondert zu betrachtender Aspekt ist der Aggregationen betreffende Teil des Leistungsumfangs von Aktivitäten und Werkzeugen. Dieser hebt sich von den bisherigen Betrachtungen des Leistungsumfangs dadurch ab, daß durch die spezielle Semantik der Aggregationsbeziehungen weitere Möglichkeiten zu Unstimmigkeiten entstehen, die in nicht aggregierten Dokumenttypen nicht vorkommen können.

4.1.2.1 Aggregationen

Bei der Instanziierung der Umgebung kann sich herausstellen, daß das Prozeßmodell zwar einen strukturierten Dokumenttyp spezifiziert, aber keine geeigneten Werkzeuge vorhanden sind, um die einzelnen Dokumenttypen zu bearbeiten, aus denen sich dieser zusammensetzt (sowohl Komponenten als auch Aggregat).

Diese Situation kann zu verschiedenen Problemen führen. Das erste tritt auf, falls ein Werkzeug gewählt wird, das ein durch eine Aggregation modelliertes Dokument monolithisch behandelt. Diese Situation ist in Abbildung 4.2 in der Alternative D dargestellt. In diesem Fall kann nach einer Bearbeitung eines Aggregats jede seiner Komponenten geändert worden sein. In Abbildung 4.3 kann man sehen, daß deswegen alle

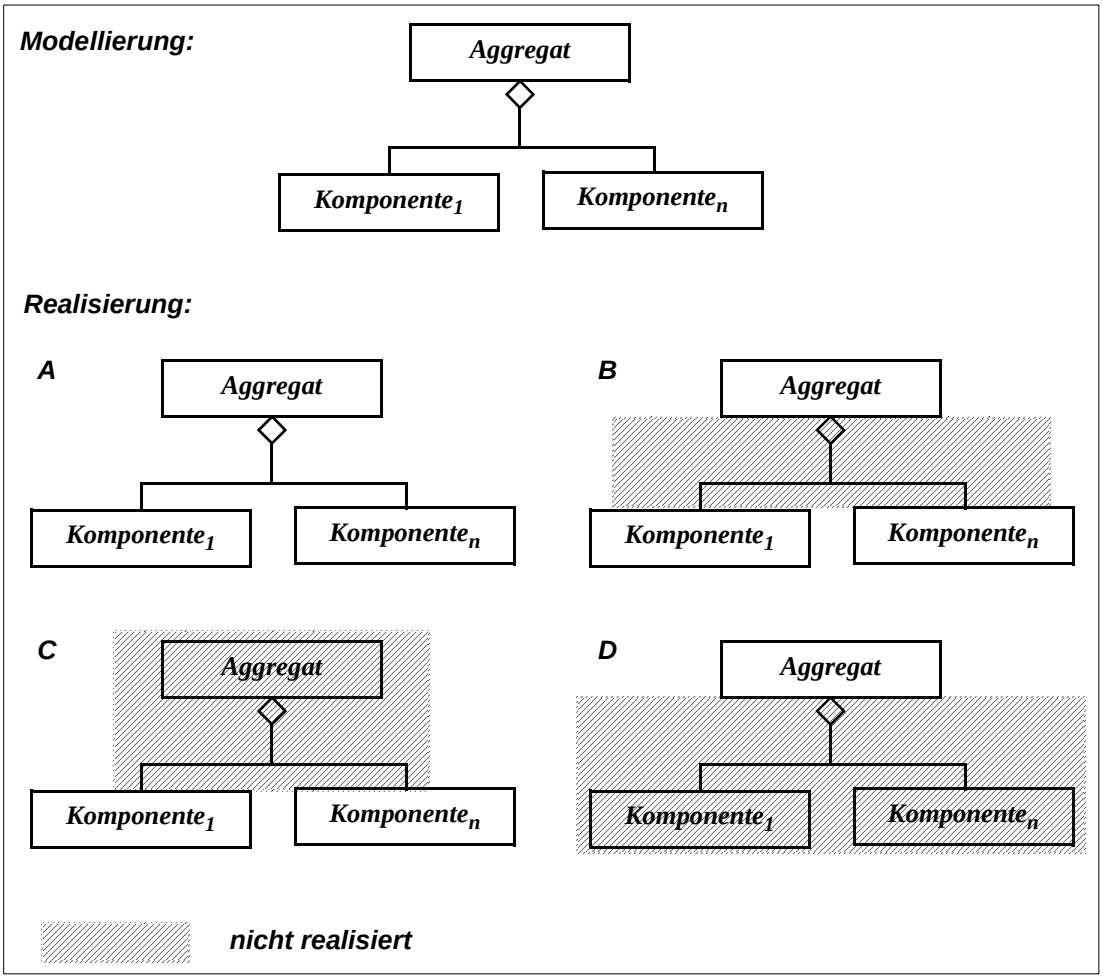


Abbildung 4.2 Alternativen zur Realisierung einer Aggregation

Dokumente, die vom Aggregat abhängen, auf Auswirkungen kontrolliert werden müssen. Darin sind auch die Dokumente enthalten, die von anderen als der bearbeiteten Komponente abhängen. Es müssen also i.d.R. unnötig viele Überprüfungen vorgenommen werden. Ein weiteres Problem der monolithischen Behandlung ergibt sich daraus, daß Aufrufe der Aktivitäten, die von der PE auf den nicht implementierten Komponenten (***Komponente*$_1$**, ..., ***Komponente*$_j$**) durchgeführt werden, auf Werkzeuge abgebildet werden müssen. Dieses Problem tritt in der Realisierungsalternative C (vgl. Abbildung 4.2) in umgekehrter Form auf. Dort ist unklar, wie Aufrufe von Aktivitäten auf dem Aggregat durchgeführt werden können. Alternative B (vgl. Abbildung 4.2) wirft das Problem auf, wie Aktivitäten auf einem Aggregat, die dessen Komponenten mit einbeziehen, realisiert werden können. Da die Realisierungsvariante auf Werkzeugen aufbaut, die keine Aggregationsinformation verwalten, können diese offensichtlich nicht ohne Unterstützung der Modellausführung ablaufen.

Beispiel: Im Szenario in Kapitel 3 muß die Aktivität `bearbeiten` auf dem `Testdokument` durch ein Werkzeug implementiert werden. Das eigentliche Aggregat `Testdokument` wird nicht realisiert, stattdessen werden die beiden Komponenten `Testbeschreibung` und `Ergebnisdokumentation` jeweils mit einem normalen Texteditor bearbeitet. Diese Konstellation entspricht der Realisierungsalternative C aus Abbildung 4.2. Wird nun beispielsweise die Aktivität `drucken` auf dem Aggregat aufgerufen, so ist unklar, wie diese Aktivität umzusetzen ist. Eine mögliche Umsetzung wäre es, wenn spezifiziert werden könnte, daß die Aktivität `drucken` durch den

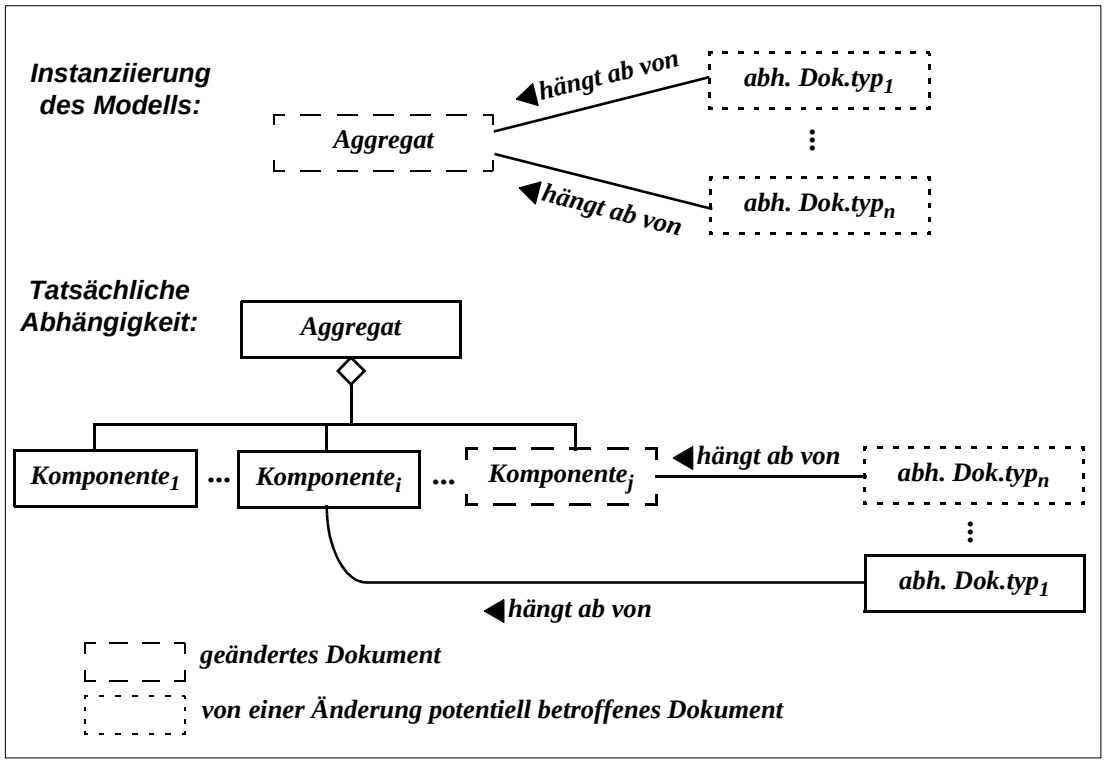


Abbildung 4.3 Mangelnde Modellierung von Abhängigkeiten bei Aggregaten

Aufruf der Aktivität `drucken` aller Komponenten simuliert werden kann (unter Verlust von Aggregationsinformationen wie z.B. der Reihenfolge der Komponentenausdrucke).

An das entwickelte Lösungskonzept ergibt sich die folgende Anforderung:

Implementierung modellierter Aggregationen Sind keine Werkzeuge vorhanden, um eine im Modell spezifizierte Aggregation auch in der Prozeßausführung als Aggregation zu behandeln, so sollen alternative Implementierungen, die die Aggregationsinformation aufgeben, erlaubt sein (vgl. Alternativen B-D aus Abbildung 4.2). Dazu muß spezifizierbar sein, wie mit nicht direkt abbildbaren Aktivitäten zu verfahren ist.

4.1.3 Zustandsmodell

Ein weiterer Punkt betrifft die Zustandsmodelle der Aktivitäten. Es kann vorkommen, daß deren Spezifikation im Prozeßmodell von den die Aktivitäten implementierenden Werkzeugen nicht erfüllt wird. Das Zustandsmodell einer Aktivität beschreibt dabei, daß ein Dokument, auf dem eine Aktivität aufgerufen wird, durch die Aktivität von seinem Ausgangszustand in genau einen von möglicherweise mehreren Folgezuständen überführt wird. Bei der Bearbeitung ebenfalls gelesene oder geschriebene Dokumente werden in ihrem Zustand dabei nicht verändert.

Beispiel: Im Szenario aus Kapitel 3 ist auf dem Dokumenttyp `Implementierung` eine Aktivität `übersetzen` definiert, die durch einen Compiler-Aufruf implementiert wird. Die Aktivität hat zwei verschiedene Ausgänge, nämlich eine fehlerfreie Übersetzung und eine, bei der Fehler entdeckt worden sind. Der Compiler kennt nun noch weitere mögliche Fälle. So kann es eine fehlerfreie Übersetzung geben, bei der

aber Warnungen erzeugt worden sind. Auch kann es zu Meldungen gekommen sein, bei denen nicht mehr festgestellt werden kann, ob der zu übersetzende Quelltext Fehler hat. Ein nicht ausreichender Hauptspeicher ist ein Beispiel für den zuletzt genannten Fall.

Um dieses Problem zu lösen, ist es notwendig, daß sich die zusätzlichen Zustände auf die im Prozeßmodell spezifizierten abbilden lassen. Dieses ist am einfachsten möglich, wenn die Zustandsmodelle gleich viele Zustände enthalten und jedem Zustand des Zustandsmodells der Aktivität in einer bijektiven Abbildung ein Zustand des Zustandsmodells des Werkzeugs zugeordnet wird. Enthält das Zustandsmodell eines Werkzeugs dagegen weniger Zustände als das der zugehörigen Aktivität, so können nicht alle Zustände der Aktivität im Prozeß erreicht werden. Ein Prozeßablauf, der der Modellierung entspricht, ist dann nicht mehr sichergestellt. Damit kann die Aktivität nicht durch dieses Werkzeug implementiert werden. Enthält das Zustandsmodell eines Werkzeugs umgekehrt mehr Zustände als das der zugehörigen Aktivität, so ist dies kein Problem, solange jedem der Zustände des Zustandsmodells der Aktivität mindestens ein Zustand aus dem Zustandsmodell des Werkzeugs zugeordnet werden kann.

Analog zu den vorangegangenen Anforderungen ergibt sich:

Spezifikation des Zustandsmodells Um zu erkennen, ob ein Werkzeug als Implementierung einer Aktivität eingesetzt werden kann, muß es zwei Elemente geben: a) Das Zustandsmodell einer Aktivität muß im Prozeßmodell spezifiziert sein. b) Die Spezifikation des Zustandsmodells eines Werkzeugs, das eine Aktivität implementieren soll, muß vorliegen.

Abbildung des Zustandsmodells Es muß bei der Instanziierung möglich sein zu spezifizieren, wie die Zustände des Zustandsmodells eines Werkzeugs auf die der Aktivitäten abgebildet werden. Dabei muß die Abbildung sicherstellen, daß jeder Zustand des Zustandsmodells einer Aktivität das Bild mindestens eines der Zustände aus dem Zustandsmodell der Werkzeuge ist. Darüber hinaus werden nur Ausgangszustände auf Ausgangszustände und Folgezustände auf Folgezustände abgebildet.

4.1.4 Prozeßunabhängige Ausführungsinformationen für Werkzeuge

Werden Werkzeuge als Implementierung der Aktivitäten festgelegt, so kann es durchaus vorkommen, daß die Informationen, die direkt aus der Instanz des Prozeßmodells gewonnen werden können, nicht ausreichen, um das Werkzeug aufzurufen. Unter Umständen sind weitere Informationen notwendig, die dann typischerweise die Aufgabe des Werkzeugs näher beschreiben und werkzeugspezifisch sind.

Beispiel: Im Szenario aus Kapitel 3 wird ein Compiler eingesetzt, der den Dokumenttyp `Implementierung` übersetzt. Der Compiler benötigt zum einen die Information, welches Dokument er übersetzen soll und welche Include-Dateien dazu benutzt werden sollen. Darüber hinaus benötigt er aber auch Informationen, ob er eine Objektdatei erzeugen, einen Linklauf automatisch anstoßen oder optimierten Code erzeugen soll. Dies sind Festlegungen, die die Aufgabe des Werkzeugs näher beschreiben und ohne die der vollständige Werkzeugaufruf nicht formuliert werden kann.

Um die vollständigen Werkzeugaufrufe beschreiben zu können, wird die folgende Anforderung formuliert:

Spezifikation der prozeßunabhängigen Eingaben Um Werkzeuge als Implementierung von Aktivitäten aufrufen zu können, muß zur Instanziierungszeit die Möglichkeit gegeben sein, die prozeßunabhängigen Eingaben zu spezifizieren.

Mit dieser Anforderung ist die Betrachtung der Aspekte, die bereits zur Instanziierungszeit behandelt werden können, abgeschlossen.

4.2 Inkonsistenzen zur Laufzeit

Inkonsistenzen zur Laufzeit treten dann auf, wenn in der Prozeßausführung prozeßrelevante Änderungen von Dokumenten oder Beziehungen vorgenommen werden, die der Prozeßsteuerung nicht bekannt sind. Sie treten zu Zeitpunkten auf, die sich vorher nicht benennen lassen und können in heterogenen PSEU temporär nicht vermieden werden. Um diese Inkonsistenzen zu beheben, ist es nötig, Feedback von den ändernden Werkzeugen oder vom Entwickler an die Prozeßsteuerung zu geben. Auf dieser Basis kann die Prozeßsteuerung nun korrekte Entscheidungen im Hinblick auf den Fortgang des Prozesses fällen. Entdeckt sie Fehler, so kann sie Schritte zu deren Korrektur einleiten.

Diese Inkonsistenzen lassen sich vom Typ her in zwei Gruppen unterteilen. Zunächst einmal gibt es Inkonsistenzen, die bei der regulären Prozeßausführung entstehen (Kapitel 4.2.1). Mit dem Begriff reguläre Prozeßausführung ist gemeint, daß es bei der Prozeßausführung nicht zu Werkzeug- oder PE-Abbrüchen kommt. Die Inkonsistenzen, die durch solche Abbrüche entstehen können, fallen in die zweite Gruppe (Kapitel 4.2.2).

4.2.1 Reguläre Prozeßausführung

Das Feedback, das bei der regulären Prozeßausführung gegeben wird, bezieht sich einerseits auf Änderungen von Dokumenten und Beziehungen (Kapitel 4.2.1.1), andererseits müssen aber auch Zustandsänderungen gemeldet werden (Kapitel 4.2.1.2).

Nach solchen Rückmeldungen ist es nötig, das erhaltene Feedback auf Prozeßkonformität zu überprüfen, um gegebenenfalls Fehler zu erkennen und beheben zu können (Kapitel 4.2.1.3, *Auswertung des Feedbacks*). Es gibt aber auch noch Situationen, in denen Fehler auftreten können, die nicht unbedingt im Feedback erkannt werden können. Dabei handelt es sich um den Fall, daß ein Werkzeug als Implementierung einer Aktivität zugelassen wurde, bei dem der Leistungsumfang der Aktivität nicht mit dem des Werkzeugs übereinstimmt. Auch für diese Fehler muß es Mechanismen geben, um sie beheben oder vermeiden zu können (Kapitel 4.2.1.3, *Kontrollen außerhalb der Feedbackmechanismen*).

4.2.1.1 Änderungen der Daten

Ist die initiale Instanziierung des Prozeßmodells sowohl in der Modellausführung als auch in der Prozeßausführung abgeschlossen, so werden die Instanziierungen während der Entwicklung ständig verändert. Die zur Prozeßsteuerung benötigten Informationen werden teilweise redundant in Modell- und Prozeßausführung gehalten. Damit ergibt sich durch die Veränderungen der Instanziierung das Problem, die Veränderungen redundant gehaltener Daten konsistenzerhaltend durchzuführen. Dieses Problem betrifft

das Feedback der Werkzeuge und dessen Auswertung in der Modellausführung. Der Kern des Problems liegt nicht darin, daß temporär Inkonsistenzen entstehen. Es geht vielmehr darum, daß solche temporären Inkonsistenzen vom Verursacher an die Prozeßsteuerung gemeldet werden müssen, damit diese korrekt arbeiten kann.

Beispiel: Ein Entwickler fügt in einem von ihm bearbeiteten C-Modul eine neue Include-Abhängigkeit ein. Ist diese Abhängigkeit zwischen den Dokumenttypen `Interface` und `Implementierung` aus dem Szenario aus Kapitel 3 auch im Modell als Beziehung modelliert, so muß sie auch in der Modellausführung nachgezogen werden. Dazu ist es zunächst notwendig, daß die Modellausführung überhaupt Kenntnis von der Veränderung erhält. Dies ist nicht unbedingt sichergestellt, da ein normaler Texteditor, wie er zur Implementierung der Aktivität `bearbeiten` eingesetzt wird, solche Informationen nicht an die Prozeßsteuerung weitermeldet.

Für ein Lösungskonzept ergibt sich damit folgende Anforderung:

Übermittlung von Feedback In einer heterogenen PSEU muß Feedback von den Werkzeugen an die Prozeßsteuerungskomponente gegeben werden. Es informiert über prozeßrelevante Änderungen der bearbeiteten Dokumente und Beziehungen.

Über den Zeitpunkt des Feedbacks wird zunächst keine Aussage gemacht. So sind Situationen denkbar, in denen es für den Prozeßablauf ausreichend ist, wenn nicht nach jedem Werkzeugaufruf Feedback gegeben wird, sondern z.B. nur vor einer Veränderung eines bearbeiteten Dokuments im Prozeß. Wird nach jeder Änderung oder nach jedem Werkzeugaufruf Feedback gegeben, so wird der Kommunikations- und Abstimmungsaufwand durch Feedback unnötig vergrößert. Umgekehrt kann es auch Fälle geben, in denen dieser Aufwand in Kauf genommen wird, um zu jedem Zeitpunkt eine konsistente Sicht des Prozeßmodells zu erhalten. Welcher der Fälle vorliegt, sollte spezifizierbar oder während der Einbindung eines Werkzeugs festlegbar sein.

Die Anforderung zur *Übermittlung von Feedback* beschreibt zunächst nur den Inhalt eines Feedbacks. Auf die Technik, die gewählt wird, wird im Kapitel 4.3 eingegangen.

4.2.1.2 Zustandsänderungen der Dokumente

Wird ein Dokument von einem Werkzeug bearbeitet, so wird es aus seinem aktuellen Zustand in einen Folgezustand überführt. Dieser Folgezustand muß von der Prozeßausführung an die Prozeßsteuerungskomponente rückgemeldet werden, um den (korrekten) Fortgang des Prozesses zu gewährleisten.

Beispiel: Wird das Werkzeug `compile` auf einem Implementierungsdokument ausgeführt, so gibt es nach Beendigung der Aktivität zwei mögliche Folgezustände, nämlich eine fehlerfreie Übersetzung (`compiled_ok`) und eine, bei der Fehler entdeckt worden sind (`compiled_with_errors`). Der jeweilige Zustand muß an die Prozeßsteuerungskomponente weitergegeben werden, da der weitere Prozeßablauf von diesem Zustand abhängt. Wird beispielsweise ein Fehler gemeldet (`compiled_with_errors`), so muß die Prozeßsteuerungskomponente veranlassen, daß das übersetzte Modul dem Entwickler zur erneuten Bearbeitung zur Verfügung gestellt wird.

Damit die Prozeßsteuerungskomponente ihre Steuerungsaufgabe wahrnehmen kann, muß die Information, in welchen Zustand das Werkzeug das Dokument, auf dem es

aufgerufen worden ist, überführt hat, von den Werkzeugen an die Prozeßsteuerungskomponente gemeldet werden. Bei der Instanziierung kann es dabei auch vorkommen, daß die Spezifikation einer Aktivität im Prozeßmodell andere Übergänge vorsieht, als sie durch das implementierende Werkzeug gemeldet werden. Nach der Anforderung zur *Abbildung des Zustandsmodells* gibt es dann jedoch eine Abbildung, die diesen Zustand auf einen von der Aktivität modellierten abbildet (vgl. Kapitel 4.1.3).

Es ergibt sich die folgende Anforderung, die sicherstellt, daß die bei der Prozeßmodellierung spezifizierte Abbildung des Zustandsmodells zur Laufzeit auch in passende Informationen umgesetzt wird:

Übermittlung von Zustandsinformation In einer PSEU muß als Abschluß jeder Werkzeugausführung eine Zustandsinformation an die Prozeßsteuerungskomponente übermittelt werden. Diese Rückmeldung muß ein Folgezustand sein, wie er in der Abbildung der Zustandsmodelle definiert wurde.

4.2.1.3 Sicherung der Prozeßkonformität

Auswertung des Feedbacks

Es kann vorkommen, daß Werkzeuge implementiert werden, die Änderungen an Dokumenten oder Beziehungen vornehmen, ohne vorher mit der PE abzuklären, ob diese Änderungen durch den Prozeß erlaubt sind. Die von solchen Änderungen betroffenen Dokumente oder Beziehungen liegen u.U. danach so vor, daß sie nicht mehr prozeßkonform sind, d.h. daß der Zustand nicht durch einen vorgesehenen Ablauf des Prozesses erreicht werden kann. Es ist wichtig, daß solche Fehler entdeckt werden. Falls sie nicht beseitigt werden können, so muß das Dokument entsprechend gekennzeichnet und der Prozeßablauf bezüglich dieses Dokuments eingefroren werden.

Beispiel: Im Szenario in Kapitel 3 ist vorgesehen, daß aus Dokumenten des Typs `Architektur` sowohl Include-Dateien wie auch Rümpfe der C-Module generiert werden. Bearbeitet nun ein Entwickler z.B. ein C-Modul, so darf er die generierten Teile, d.h. insbesondere die in der Architektur definierten Abhängigkeiten, nicht auf seiner Ebene verändern. Dies wird aber durch den eingesetzten Texteditor nicht sichergestellt. Ein weiteres Beispiel ist die Vergabe von Bezeichnern z.B. für ein neues Dokument. Wird hierbei ein doppelter Name gewählt, so muß dies erst korrigiert werden, bevor der Prozeß auf diesem Dokument basierend fortgesetzt werden kann.

Feedback-Mechanismen werden erst dann sinnvoll, wenn in der Modellausführung auch eine Kontrolle und Auswertung der Informationen erfolgt. Daraus ergibt sich die folgende Anforderung:

Auswertung von Feedback Um sicherzustellen, daß durchgeführte Änderungen prozeßkonform sind, müssen Rückmeldungen ausgewertet und auf ihre Gültigkeit überprüft werden. Werden unerlaubte Änderungen erkannt, muß die Prozeßsteuerungskomponente die betroffenen Dokumente solange als nicht prozeßkonform kennzeichnen, bis die Änderungen rückgängig gemacht worden sind. Diese Dokumente müssen bis zur Aufhebung dieser Kennzeichnung in dem Zustand verbleiben, in dem sie sich vor der Kennzeichnung befunden haben.

Kontrollen außerhalb der Feedbackmechanismen

Es kommt in der Praxis auch vor, daß eine Aktivität durch ein Werkzeug implementiert werden soll, dessen Leistungsumfang größer ist als der der zugehörigen Aktivität (vgl. Kapitel 4.1.2). Durch die Benutzung eines solchen Werkzeugs kann es zu unerwünschten Effekten kommen. Es könnte z.B. möglich sein, daß schreibend auf ein Dokument zugegriffen werden kann, obwohl das Prozeßmodell für dieses Dokument nur einen lesenden Zugriff vorsieht. In diesem Fall ist in der Prozeßausführung darauf zu achten, daß die Benutzung des Werkzeugs nicht dazu führt, daß die Prozeßsteuerung unterlaufen wird.

Beispiel: Auf dem Dokumenttyp `Anforderungsanalyse` aus dem Szenario aus Kapitel 3 wird eine Aktivität `review` definiert. Diese soll auf die Anforderungsanalyse nur lesend zugreifen. Die Kommentare des Reviews werden in einer weiteren Aktivität in einem Reviewdokument abgelegt. Wird diese Aktivität durch einen Texteditor realisiert, da kein echter Datei-Viewer ohne Schreibmöglichkeit zur Verfügung steht, besteht die Gefahr, daß in der Anforderungsanalyse Änderungen vorgenommen werden. Dies kann in diesem Fall z.B. dadurch unterbunden werden, daß nur eine Kopie des Dokuments an das Werkzeug übergeben wird, so daß das Originaldokument nach Beendigung der Aktivität sicher noch den gleichen Zustand hat.

Um Probleme aus den oben beschriebenen Situationen zu vermeiden, muß folgende Anforderung beachtet werden:

Kontrolle des Leistungsumfangs Wird eine Aktivität durch ein Werkzeug implementiert, das die Spezifikation des Leistungsumfangs der Aktivität nicht erfüllt, so muß es Mechanismen der Prozeßsteuerung geben, die die Konsistenz der Daten zur Laufzeit sichern.

4.2.2 Recovery nach Abbrüchen

4.2.2.1 Dokumentzustand nach Werkzeugabbrüchen

In jeder Umgebung kann es durch unvorhergesehene Ereignisse (z.B. Stromausfall) Werkzeugabbrüche geben. Im Regelfall ist nach einem solchen Abbruch unklar, in welchem Zustand sich das beim Abbruch bearbeitete Dokument befindet. Dies ist insbesondere deswegen wichtig, weil ein solcher Abbruch häufig auch eine Abweichung vom modellierten Prozeß bedeutet, die dann korrigiert werden muß.

Beispiel: Ein Werkzeug bricht mit der Feststellung ab, daß kein Hauptspeicher mehr verfügbar ist. Handelt es sich um ein Werkzeug, das das bearbeitete Dokument modifiziert, so ist nicht ohne Zusatzinformationen klar, ob die Modifikationen trotzdem schon vollständig sind, mitten in der Bearbeitung unterbrochen wurden oder noch nicht einmal begonnen haben.

Die Prozeßsteuerung muß also in einer Art Ausnahmebehandlung auf einen Werkzeugabbruch reagieren können, um wieder einen Zustand herstellen zu können, von dem aus der Prozeß normal weiterlaufen kann. Um auf dieses Problem reagieren zu können, wird daher die folgende Anforderung gestellt:

Zustandsbestimmung nach Werkzeugabbruch Für ein Werkzeug muß spezifiziert sein, in welchem Zustand es ein Dokument nach einem Abbruch hinterläßt, d.h. welchen Zustand die PE in diesem Fall für das Dokument setzen soll. Es soll dabei erlaubt sein zu spezifizieren, daß nach dem Abbruch eine manuelle Bestimmung erfolgen soll.

4.2.2.2 Zustandsbestimmung nach PE-Abbruch

Ebenso wie es zu Werkzeugabbrüchen kommen kann, kann es auch vorkommen, daß die Prozeßsteuerungskomponente ungeplant unterbrochen wird. Das ist kein Problem, wenn zum Zeitpunkt des Abbruchs keine Aktivitäten laufen, d.h. also keine Werkzeuge aktiv sind. Wird die Prozeßsteuerung jedoch unterbrochen, während Aktivitäten ausgeführt werden, ergeben sich verschiedene Teilprobleme. Zunächst muß die Prozeßsteuerungskomponente in der Lage sein, beim erneuten Start zu rekonstruieren, welche Aktivitäten zum Zeitpunkt des Abbruchs aktiv waren, um die Fälle zu identifizieren, in denen weitere Aktionen notwendig sind. Dann muß für jede Aktivität herausgefunden werden, welchen Zustand die jeweils bearbeiteten Dokumente nach Abschluß der Aktivität hatten, um die Daten der Prozeßausführung wieder aktualisieren zu können. Dies ist schwierig, da für die Aktivitäten drei Möglichkeiten denkbar sind: a) eine Aktivität läuft noch immer, b) eine Aktivität ist korrekt (nur eben ohne Feedback an die PE) beendet worden und c) eine Aktivität ist mit der Prozeßsteuerungskomponente abgebrochen worden (z.B. bei einem Rechnerausfall, wobei das Werkzeug und die PE auf dem gleichen Rechner ausgeführt wurden).

Beispiel: Im Szenario aus Kapitel 3 könnte ein solcher Fall darin bestehen, daß ein Entwickler die Architektur bearbeitet, während durch einen Fehler die Prozeßsteuerung ausfällt. In diesem Fall kann es sein, daß der Entwickler nichts vom Ausfall merkt, da er den Architektureditor im Vordergrund hat. Er arbeitet daher zunächst weiter. Irgendwann beendet er seine Tätigkeit und will nun festlegen, daß die Architektur im Prozeß als fertig gekennzeichnet werden soll. Letzteres kann er aber nicht tun, da die Prozeßsteuerungskomponente nicht zur Verfügung steht.

Daher ergibt sich als Anforderung:

Zustandsbestimmung nach PE-Abbruch Die Prozeßsteuerungskomponente muß in der Lage sein, nach einem Abbruch „Recovery“-Maßnahmen anzustoßen bzw. durchzuführen. Dabei müssen Änderungen, die nach dem Abbruch und vor einem erneuten Start der PE an den Instanzen der Prozeßausführung vollzogen wurden, in der Modellausführung nachgezogen werden.

4.3 Technische Probleme bei der Anbindung der Werkzeuge

Um die Anforderungen aus Kapitel 4.1 und insbesondere Kapitel 4.2 umsetzen zu können, muß ein geeigneter Integrationsmechanismus die Kommunikation zwischen der Prozeßsteuerungskomponente und den Werkzeugen implementieren. Was „geeignet“ bedeutet, wird in den folgenden Anforderungen festgelegt. Werkzeuge und Prozeßsteuerungskomponente müssen also im Sinne der Definition aus Kapitel 2.1.2 auf der technischen Ebene die Voraussetzung für eine Integration bieten. Dabei ist einer der wichtigsten Faktoren für eine heterogene PSEU, daß die Anbindung extrem flexibel sein muß.

4.3.1 Integrationstechnik

Bei der Integration von Werkzeugen und Prozeßsteuerungskomponente muß zunächst einmal eine beiden gemeinsame Technik gefunden werden, die einen Austausch von Informationen ermöglicht. Dies ist in einer heterogenen PSEU schwierig, da dort keine Annahmen über die bei den einzelnen Werkzeugen verfügbaren Techniken gemacht werden können. Es muß mit einer Vielfalt von Techniken umgegangen werden, die durch neue Entwicklungen jederzeit größer werden kann.

Beispiel: Im Kapitel 4.2.1.1 wurde gefordert, daß Feedback über Änderungen prozeßrelevanter Daten erfolgen soll. Handelt es sich dabei um mehrere eingefügte Beziehungen, so kann diese Information auf verschiedenste Weise weitergegeben werden: Nachrichtenmechanismen, gemeinsame Datenablagen, manuelle Abfragen etc. Ein einheitliches Kommunikationsprotokoll existiert nicht. Es kann aber aufgrund der notwendigen Flexibilität bei der Suche nach einem geeigneten Kommunikationsprotokoll keine Lösung sein, z.B. auf gemeinsamen Datenspeichern, wie einem Repository aufsetzen zu wollen. Welche Techniken geeignet sind, wird im Detail in Kapitel 5.1 behandelt.

Da es offensichtlich keine Lösung sein kann, für jede von den Werkzeugen benutzte Anbindungstechnik eine eigene Anbindungsmöglichkeit in der Prozeßsteuerungskomponente zu implementieren, werden die folgenden Anforderungen formuliert:

Flexibilität der Werkzeugwahl Um die Zahl der einbindbaren Werkzeuge möglichst groß zu halten, muß die Architektur einer PSEU konzeptionell eine Schicht zwischen Werkzeug und Prozeßsteuerungskomponente einziehen. Innerhalb dieser Schicht muß eine Schnittstellenumsetzung auf ein einheitliches Kommunikationsprotokoll vorgenommen werden. Dies soll nicht ausschließen, daß es speziell implementierte Werkzeuge geben darf, die direkt mit der Prozeßsteuerungskomponente kommunizieren.

Wiederverwendbarkeit der Anbindung Werden Werkzeuge angebunden, die die Anforderung zur *Flexibilität der Werkzeugwahl* erfüllen, so sollen sie auch in einer anderen, neu zusammengestellten PSEU benutzt werden können, ohne weiteren Anbindungsaufwand zu erzeugen.

4.3.2 Erkennung von Werkzeugabbrüchen

Ein Phänomen, das nicht nur in heterogenen PSEU ein Problem aufwirft, ist die Erkennung von Werkzeugabbrüchen (vgl. Kapitel 4.2.2.1). Dabei gibt es zwei „Qualitäten“ von Abbrüchen, nämlich kontrollierte und unkontrollierte. Bei kontrollierten Abbrüchen kommt es während der Programmausführung zu Fehlern, die zwar erkannt werden, aber die normale Beendigung des Werkzeugs nicht mehr zulassen. In diesem Fall wird nach der Anforderung zur *Zustandsbestimmung nach Werkzeugabbruch* (Kapitel 4.2.2.1) Feedback an die Prozeßsteuerung gegeben. Unkontrollierte Abbrüche hingegen entstehen durch Umstände, die durch das Werkzeug nicht erkannt werden oder nicht erkannt werden können. In diesem Fall gibt es keine Feedback mehr und es entsteht für die Prozeßsteuerung das Problem festzustellen, daß ein Abbruch eines Werkzeugs stattgefunden hat.

Beispiel: Ein Werkzeug stellt zur Laufzeit fest, daß kein Hauptspeicher mehr verfügbar ist, und betrachtet dieses als fatalen Fehler. In diesem Fall ist die günstigere Möglich-

keit, daß noch eine Information über den fehlgeschlagenen Verlauf an die Prozeßsteuerung übermittelt wird (vgl. Kapitel 4.2.1.2), wodurch es zu einem kontrollierten Abbruch kommt. Die ungünstigere Reaktion ist eine Beendigung ohne weitere Informationen. Dazu käme es auch im Falle eines unkontrollierten Abbruchs, wie z.B. bei einem Stromausfall.

Als Anforderung ergibt sich damit:

Erkennung von Werkzeugabbrüchen Die Anbindung der Werkzeuge muß so implementiert sein, daß ein Ausfall einzelner Werkzeuge zu ihrer Laufzeit oder spätestens beim nächsten Start der Prozeßsteuerungskomponente erkannt werden kann.

4.4 Zusammenfassung

In den vorangegangenen Kapiteln sind Probleme identifiziert worden, die bei der

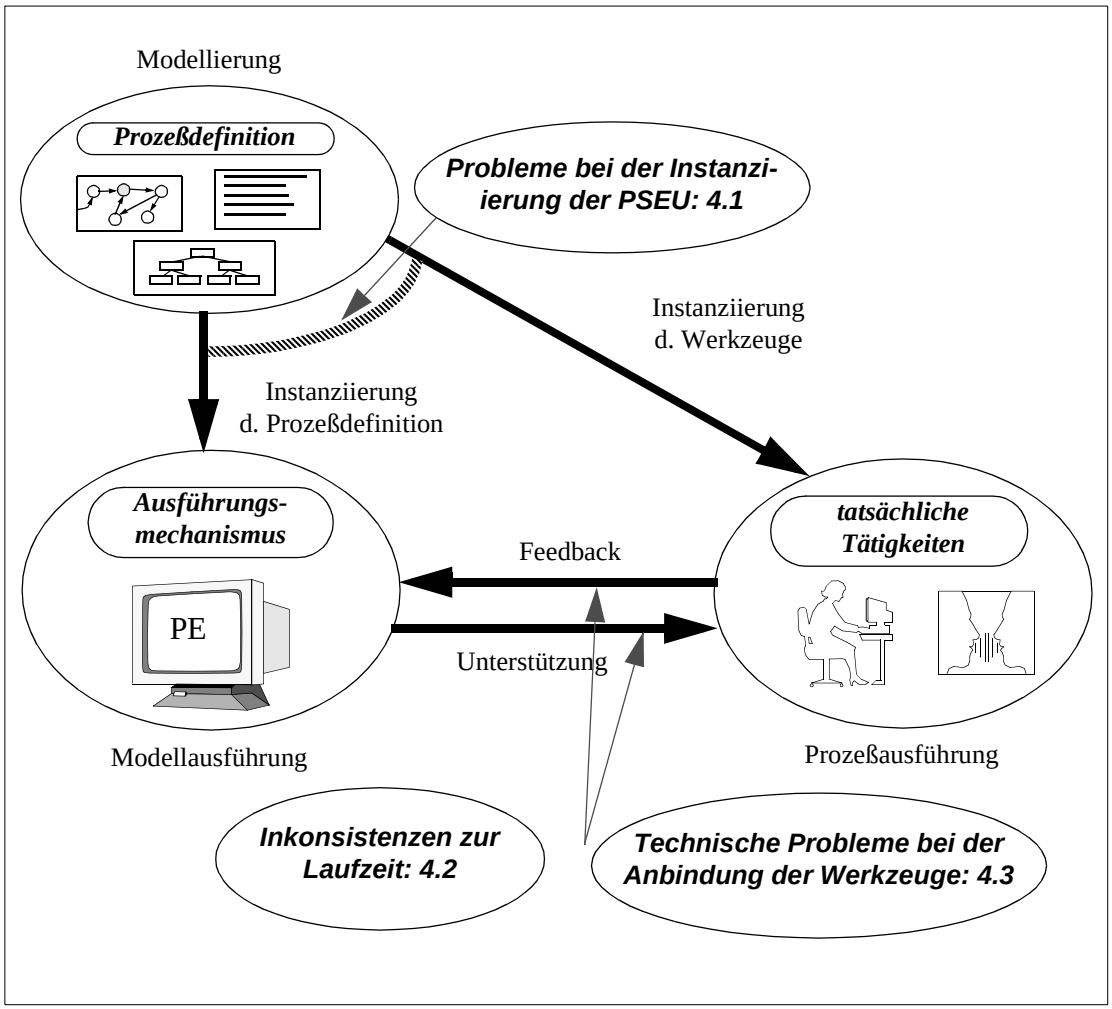


Abbildung 4.4 Zuordnung der identifizierten Probleme in heterogenen PSEU

Instanziierung des Prozeßmodells bzw. der Instanziierung der konkreten Werkzeuge der Umgebung auftreten, zur Laufzeit entstehen oder die primär etwas mit der technischen Seite der Integration zu tun haben. Daraus wurden Anforderungen abgeleitet,

deren Einhaltung die Vermeidung der identifizierten Probleme erlaubt. An diesen werden die in dieser Arbeit entwickelten Lösungen gemessen.

In Abbildung 4.4 werden die in den vorangegangenen Kapiteln im Detail untersuchten Probleme in die in Kapitel 2.1 eingeführten Bereiche eingeordnet. Es handelt sich dabei um eine Einordnung, an welchen Stellen die Probleme sichtbar werden und nicht, in welchen Bereichen nach einer Lösung gesucht wird. Wird also ein Problem z.B. in der Instanziierung der Umgebung sichtbar, so kann es unter Umständen sein, daß das Problem durch Veränderung eines vorangehenden Schrittes (hier z.B. der Modellierung) gelöst werden kann.

Im folgenden Kapitel werden Techniken und prozeßgesteuerte Umgebungen, die bereits existieren, daraufhin untersucht, welche der Anforderungen sie erfüllen. Die in diesem Kapitel definierten Anforderungen sind die Basis für diese Überprüfung.

5

EXISTIERENDE INTEGRATIONS-TECHNIKEN UND PSEU

In diesem Kapitel wird untersucht, inwieweit existierende Ansätze die Anforderungen aus Kapitel 4 bereits erfüllen. Dazu werden zwei Bereiche unterschieden. Der erste Bereich beschäftigt sich mit existierenden Integrationstechniken. Hier geht es darum festzustellen, inwieweit diese Integrationstechniken speziell auch in heterogenen PSEU eingesetzt werden können, d.h. ob es eine Integrationstechnik gibt, die eine flexible Integration von Werkzeugen in eine PSEU erlaubt. In Kapitel 5.2 werden dann exemplarisch einige existierende PSEU mit den von ihnen angebotenen Konzepten auf die Einhaltung der Anforderungen aus Kapitel 4 untersucht.

5.1 Integrationstechniken

Es gibt verschiedene Ansätze zur Integration von Werkzeugen. Ansätze zum Aufbau einfacher (d.h. nicht prozeßgesteuerter) SEU basieren auf den in diesen Ansätzen definierten Integrationstechniken. Die einzelnen Techniken können in zwei Gruppen aufgeteilt werden. Die einen betrachten die Integration unter dem Fokus der gemeinsamen Nutzung von Daten, die anderen legen den Schwerpunkt auf den gegenseitigen Einsatz von Funktionen (die dann ihrerseits Daten benutzen).

Datenorientierte Ansätze zur Integration von Werkzeugen basieren darauf, daß der wesentliche Problempunkt einer Integration die von den zu integrierenden Werkzeugen bearbeiteten Daten sind. Diese Herangehensweise führt zu Ansätzen, die sich mit der Frage beschäftigen, wie Daten, die in mehreren Werkzeugen benutzt werden, sinnvoll gemeinsam definiert und/oder ausgetauscht werden können.

In funktionsorientierten Ansätzen rückt die Funktionalität der Werkzeuge in den Vordergrund. Diese Ansätze können sich mit datenorientierten Ansätzen ergänzen, wenn als Effekt einer Funktion Daten zwischen Werkzeugen abgeglichen werden. Die Kategorien sind in diesem Sinne orthogonal zueinander.

5.1.1 Datenorientierte Integrationstechniken

5.1.1.1 Repositories

Die naheliegendste Lösung eines datenorientierten Integrationsansatzes ist die Verwendung eines Dateisystems zur Ablage gemeinsamer Daten. Auf dieser Ebene arbeiten Datenaustauschmechanismen wie UNIX-Pipes. Tatsächlich ist schon vor über fünfzehn Jahren direkt aufbauend auf dem Unix-Dateisystem eine Art Umgebung propagiert worden (*Unix PWB* [KR84], vgl. auch [Nag93]), die als offene Umgebung angelegt ist und die Unix-Basisdienste wie Dateiablage und Pipes als Integrationsmechanismen benutzt.

Die Verwendung von Dateien zur Integration auf der Datenebene ist nur dann ein brauchbares Mittel, wenn die auf die jeweiligen Dateien zugreifenden Werkzeuge Annahmen über die Struktur, d.h. über die Syntax, oder noch weitergehend sogar über die Semantik der darin gespeicherten Daten machen können. Es wurden daher Verfahren entwickelt, die eine strukturierte Definition und Ablage von Daten ermöglichen sollen, um damit solche Annahmen zu erlauben. Ein bekanntes Beispiel ist die *Portable Common Tool Environment*, kurz *PCTE* ([ECMA92]). Ziel von PCTE ist es, integrierte Umgebungen zu unterstützen (vgl. dazu und auch zu weiteren Zielen Kapitel 1.2 in [ECMA92]). Eine integrierte Umgebung wird im PCTE-Umfeld als Umgebung definiert, deren einzelne Werkzeuge eng zusammenarbeiten und keine heterogene Sammlung von Werkzeugen bilden (vgl. a priori Integration und homogene Umgebungen in Kapitel 2.1.2). Offenheit einer Umgebung bedeutet dann, daß es möglich ist, Werkzeuge zur Umgebung hinzuzufügen, falls diese mit den anderen Werkzeugen der Umgebung a priori integriert sind (vgl. offene Umgebungen in Kapitel 2.1.2). In PCTE wird sowohl die Spezifikation der Daten wie auch der Zugriff auf sie durch Konzepte des Entity-Relationship-Modells (ER-Modell, [Che76]) realisiert, wodurch die Struktur der Daten betont wird. Zur Unterstützung dieses Vorgehens ist ein wesentlicher Bestandteil von PCTE eine Datenbank-Implementierung. Integration basiert auf einer Definition der gemeinsamen Daten der zu integrierenden Werkzeuge im Rahmen eines Datenbankschemas. Der Zugriff der Werkzeuge auf die gemeinsam genutzten Daten in der Datenbank erfolgt auf der Basis dieser Schemadefinition. Die zur Integration notwendige Information ist also im wesentlichen das zuvor definierte Schema der Datenbank, auf die zugegriffen werden soll. Es handelt sich um ein *Repository* (zum Einsatz von PCTE als Basis von Werkzeugen einer SEU vgl. auch Kapitel 4.3.2 in [Emm95]).

Die Anforderung zur *Flexibilität der Werkzeugwahl* aus Kapitel 4.3.1, die auch a posteriori Integration möglich machen soll, steht in einem offensichtlichen Widerspruch zu den dargestellten, im Umfeld von PCTE formulierten Anforderungen. Dies gilt auch für andere Repositories, die auf den Ideen einer gemeinsamen Datendefinition beruhen. Interessant ist in diesem Zusammenhang auch eine Untersuchung in [BESS96], die allgemein den Aspekt von Datenbanken in PSEU betrachtet.

5.1.1.2 Datenmodellierung

Lacroix und Vanhoedenaghe ([LV89]) konzentrieren sich in ihrem ebenfalls datenorientierten Ansatz *COMS* auf die Ebene der Datenmodellierung, legen aber im Gegensatz zu PCTE keinen Wert auf eine zentrale Speicherung in einem gemeinsamen Repository. Auf der abstrakten Modellierungsebene ist die Intention wie bei PCTE, a

priori ein von mehreren Werkzeugen gemeinsam genutztes Datenbankschema zu definieren. Auf diese Weise soll die Integration von Werkzeugen vereinfacht werden, die nachträglich unter der Kenntnis des Schemas implementiert werden. Um dieses Ziel zu erreichen, werden eine einfache Datendefinitionssprache (DDL) zur Beschreibung und eine einfache Datenmanipulationssprache (DML) zur Bearbeitung der Daten definiert. Die DDL stützt sich dabei auf ER-Modell-Konzepten ab. Werkzeuge, die auf den Daten arbeiten, sind entweder von einem zugehörigen Kommandointerpreter ausgeführte Zusammenfassungen von DML-Aufrufen oder in einem Sprachzusatz zur DML definierte externe Werkzeuge. Bei diesen Werkzeugen handelt es sich um UNIX-Kommandozeilen-Werkzeuge, deren parametrisierter Aufruf als Werkzeugspezifikation definiert wird. Um die Zugriffe auf die Daten durch Werkzeuge und externe Werkzeuge kontrollieren zu können, wird ein zusätzliches Konzept eingeführt. Dieses erlaubt es, Regeln zu spezifizieren, die z.B. vor der Ausführung eines Werkzeugs Attributwerte der Daten testen können, um sicherzustellen, daß das Werkzeug nur in bestimmten, durch den Attributwert gekennzeichneten Situationen eingesetzt werden kann.

Die generelle Problematik eines a priori definierten Datenschemas, insbesondere im Bereich der Integration von Werkzeugen in eine prozeßgesteuerte Umgebung, ist im Kapitel 5.1.1.1 bereits angesprochen worden. Im Gegensatz zu den Repository-Ansätzen ist die Einbindung externer Werkzeuge als Versuch zu werten, a posteriori Integration zu vereinfachen. Interessant an diesem Ansatz ist auch, daß eine Definition von Regeln vorgesehen ist, die die Zugriffe kontrollieren und koordinieren. Dies erlaubt eine Kontrolle der auf den Daten arbeitenden Werkzeuge und bezieht somit auch Prozeßüberlegungen mit ein. Das Konzept bietet durch die notwendige Definition eines gemeinsamen Datenmodells a priori trotzdem noch zuwenig Flexibilität, um der Anforderung zur *Flexibilität der Werkzeugwahl* gerecht zu werden.

5.1.1.3 Datenmapping

In diesem Kapitel sollen datenorientierte Ansätze zur Integration von Werkzeugen beschrieben werden, die hier unter der Überschrift *Datenmapping* zusammengefaßt sind. Sie bilden getrennt voneinander modellierte und gespeicherte Daten aufeinander ab und erreichen dadurch die gewünschte Integration. Im Unterschied zu den Ansätzen aus den beiden vorangegangenen Kapiteln wird bei diesen Ansätzen weder eine zentrale Speicherung noch ein allen Werkzeugen gemeinsames Datenmodell vorausgesetzt.

Das *CASE Data Interchange Format* (*CDIF*, [Imb91], [CDI92]) ist eine Initiative, die einen Standard zum Austausch von Daten zwischen verschiedenen Software-Entwicklungswerkzeugen bzw. -umgebungen definiert hat. Das Hauptaugenmerk liegt dabei auf der Formulierbarkeit des gemeinsamen Datenschemas zum Austausch der entsprechenden Daten und nicht in der Betrachtung eines konkreten Speichermediums, also z.B. einer Datenbank. CDIF ist in mehreren Schichten aufgebaut. Das *CDIF Meta-Meta-Model* beschreibt, welche Elemente in einem *CDIF Meta-Model* vorkommen können, das CDIF Meta-Model beschreibt dann wiederum, welche Elemente in einem konkreten *CDIF Model* auftreten können. Über dessen Instanziierung können dann die eigentlichen Daten ausgetauscht werden. In CDIF werden Meta-Modelle für mehrere Bereiche vordefiniert, u.a. für Entity-Relationship Modelle und Datenflußdiagramme. CDIF kann nun z.B. dazu benutzt werden, ein unabhängiges Modell zum Austausch

von Daten zwischen zwei Werkzeugen zu definieren, die unabhängig voneinander PCTE benutzen. Ebenso können auch andere Repositories mit eingebunden werden. Dazu ist es notwendig, die konkret (z.B. in PCTE) spezifizierten Modelle in ein geeignetes CDIF Modell überführen zu können. Ein Vergleich der Möglichkeiten, die sich dazu in der Ausdrucksmächtigkeit des CDIF Meta Meta Modells und der sogenannten PCTE Foundation, dem Basismodell von PCTE, ergeben, ist in [PB92] nachzulesen. Die eigentliche Überführung der Daten erfolgt durch Exportieren der Daten des Quellsystems in einem speziellen Transferformat, das neben den eigentlichen Daten Zusatzinformationen beinhaltet, die zum Importieren auf der Zielsystemseite genutzt werden können.

CDIF und ähnliche Standards konzentrieren sich primär darauf, eine Art Import-/ Export-Mechanismus zum Datenaustausch zu ermöglichen. Dies deckt nur einen kleinen Teil der Anforderungen zur Vermeidung von Inkonsistenzen zur Laufzeit aus Kapitel 4.2 ab. Zudem muß für die eingesetzten Werkzeuge bei einem solchen Ansatz auch wieder eine Schnittstelle erstellt werden, die auf dem gemeinsamen Austauschformat basiert. Damit ist eine Verwendung im Umfeld heterogener PSEU ebenfalls nicht sinnvoll.

Aktueller als CDIF ist *XML* (*Extensible Markup Language*, vgl. [XML98]). XML ist eine Metasprache, die aus dem Web-Umfeld stammt und die ähnlich wie CDIF zur Abbildung zweier Formate aufeinander eingesetzt werden kann. Im Kontext heterogener PSEU weist sie allerdings die gleichen Probleme auf, wie sie für einen CDIF-Einsatz beschrieben worden sind.

Lefering ([Lef95]) setzt im Kontext des IPSEN-Projekts ein Mapping von Daten auf feingranularer Ebene ein. Mapping auf feingranularer Ebene bedeutet, daß die syntaktischen Strukturen der beteiligten Dokumente bekannt sind und miteinander in Beziehung gesetzt werden. Solche Abbildungen werden jeweils paarweise definiert. So ist es z.B. möglich, eine Abbildung von Datenflußdiagrammen auf die sie begleitenden Datenbeschreibungsdokumente zu definieren, um zu erreichen, daß Bezeichner, die in beiden Dokumenten vorkommen, konsistent gehalten werden können. Den Ausgangspunkt bildet also die Definition einer Abbildungsvorschrift zweier Sprachen aufeinander. Aus dieser Abbildungsvorschrift gewinnt Lefering dann Werkzeuge, die in der Lage sind, Dokumente in den jeweiligen Sprachen miteinander abzugleichen oder möglicherweise auch eines aus dem anderen zu erzeugen. Diese Werkzeuge bauen als Besonderheit des Ansatzes sogenannte *Transformationsdokumente* auf, die eine Beschreibung der erfolgten Abbildung enthalten. Dadurch kann die Abbildung nachvollzogen werden und ist in Teilen sogar umkehrbar. Verkamo und Lindén skizzieren in [VL95] ebenfalls einen Ansatz, der eine feingranulare Umsetzung von Dokumenten vornimmt. Sie gehen im Vergleich zu Lefering eher konventionell vor und generieren aus den Syntaxbeschreibungen der Sprachen, die aufeinander abgebildet werden sollen, einen Transformator, der aus einem Parser, einem Mapper und einem Unparser besteht. Eine Überprüfung der Abbildung wie durch das Transformationsdokument ist mit diesem Ansatz nicht möglich.

Die zuletzt beschriebenen Mapping-Ansätze basieren darauf, daß die syntaktische Struktur der zu integrierenden Daten bekannt ist. Zusätzlich dazu müssen diese Daten auch in einem Format abgelegt sein, das direkt lesbar ist. Nur dann können Mapping-Ansätze die erforderlichen Kontrollen bzw. Transformationen vornehmen. Für die Integration von Werkzeugen in eine prozeßgesteuerte Umgebung auf der Basis beliebi-

ger, nachträglich zu integrierender Werkzeuge, wie in Anforderung zur *Flexibilität der Werkzeugwahl* gefordert, kann ein solcher Zugriff nicht in jedem Fall sichergestellt werden. Einfachstes Beispiel ist ein Werkzeug, das seine Daten in einem speziellen Format ablegt, das Formatierungsinformation beinhaltet. Es reicht dann nicht aus, die syntaktische Struktur des Dokuments zu kennen. Außerdem müßte die syntaktische Struktur des Dateiformats bekannt sein.

5.1.2 Funktionsorientierte Integrationstechniken

5.1.2.1 Nachrichten

Die in diesem Kapitel vorgestellten Ansätze basieren auf der Definition einer Schnittstelle für Werkzeuge, die von außen (ohne Zugriff auf die Quellen des Werkzeugs) benutzt werden kann. Diese Schnittstelle ist vergleichbar mit einer Modulschnittstelle in einer Architektur aus mehreren Modulen. Da die Zusammenstellung von Werkzeugen zu einer gemeinsamen Umgebung aber nicht so statisch ist wie eine Architektur mit mehreren Modulen, ist das Schnittstellenkonzept mit dem Ziel gesteigerter Unabhängigkeit der Komponenten weiterentwickelt worden.

Reiss hat mit *Field* ([Rei90a], [Rei88], [Rei90b]) den Prototypen einer Reihe von darauf aufbauenden Implementierungen definiert. Field erlaubt dabei den Werkzeugen, die integriert werden sollen, *Nachrichten* miteinander auszutauschen. Wird von einem Werkzeug Funktionalität benötigt und bietet ein weiteres diese Funktionalität in seiner Schnittstelle an, so sendet das anfordernde Werkzeug eine Nachricht an das Werkzeug, das die Funktionalität bereitstellt. Dieses Konzept wäre ohne Erweiterungen recht unflexibel, da jedes Werkzeug genau wissen müßte, an welches andere es Nachrichten schicken kann (*Point-to-Point* Verbindung). Das hätte u.a. zur Folge, daß in den beteiligten Werkzeugen hart verdrahtet werden müßte, welche anderen Werkzeuge in dieser Form Kommunikationspartner sein könnten. Um zu vermeiden, daß solche Abhängigkeiten entstehen, ist in Field das Konzept des *Message Servers* eingeführt worden.

Der Message Server der Field Umgebung ist eine Vermittlungskomponente für Nachrichten. Jedes einzelne Werkzeug sendet Nachrichten nicht mehr direkt an ein anderes Werkzeug, sondern an den Message Server. Dadurch wird erreicht, daß jedes Werkzeug nur noch eine ausgezeichnete Komponente, den Message Server, kennen muß. Die Werkzeuge, mit denen kommuniziert werden soll, sind nicht mehr explizit bekannt. Zum Message Server wird eine Point-to-Point Verbindung aufgebaut, so daß der Message Server alle an der Umgebung beteiligten Komponenten kennt.

Für die Weitergabe einer Nachricht vom Message Server an einen Adressaten gibt es verschiedene Möglichkeiten. Zunächst müssen die Adressaten einer Nachricht bestimmt werden, da diese nicht mehr vom Absender festgelegt werden. Field nutzt dazu die Tatsache aus, daß die Werkzeuge ihr eigenes Interface genau kennen. In Field meldet daher jedes Werkzeug beim Message Server an, welche Nachrichten verarbeitet werden können, d.h. es macht sein Interface beim Message Server bekannt. Der Message Server verteilt eine eingehende Nachricht an die Werkzeuge, die durch eine passende Schnittstelle ein Interesse an dieser Nachricht angemeldet haben. Dieses Vorgehen wird aufgrund der erfolgten Vorauswahl der Empfänger *selektives Broadcasting* (auch *Multicasting*) genannt.

Durch das Konzept Message Server plus vom Werkzeug gesteuertes selektives Broadcasting ist es in Field möglich, Werkzeuge sehr flexibel zu einer Umgebung zusammenzustellen. Ein Werkzeug, das bestimmte Nachrichten zur Verfügung stellt, kann ausgetauscht werden, ohne daß irgendein anderes Werkzeug dadurch verändert werden muß. Zusätzliche Werkzeuge können in die Umgebung eingebracht werden und existierende Funktionalität nutzen, ohne daß Änderungen bei den Funktionalität erbringenden Werkzeugen notwendig sind.

Auf den Ideen von Field sind verschiedene andere Umgebungen aufgebaut und weiterentwickelt worden, wie z.B. *SoftBench* (z.B. [Cag90]) auf der Basis des *Broadcast Message Server* (*BMS*) oder *SparcWorks* (z.B. [Sun93]) auf der Basis von *ToolTalk* (z.B. [JH94]). So führt ToolTalk beispielsweise neben einer Art *Objektbegriff* auch noch *Kontexte* ein, die zusätzlich begrenzen, an welches Werkzeug eine Nachricht ausgeliefert wird. Der Objektbegriff erlaubt es, Zugriffsmethoden auf Daten zu definieren. Die Zugriffsmethoden werden durch Werkzeuge realisiert. Damit ist es dann möglich, virtuell Nachrichten an Objekte zu versenden, ohne zu wissen, wie die Implementierung der mit der Nachricht verbundenen Methoden aussieht. Insbesondere ist es auch möglich, Objekte auf der Basis von Daten zu definieren, die feingranularer als z.B. eine Datei sind.

Auch in den Konzepten des Projekts *ESF* (*European Software Factory)* werden Nachrichten als Integrationsbasis für Werkzeuge benutzt. Werkzeuge sind eine Menge von *Services* (siehe [FNO92]), wobei ein Service als „an atomic operation the user cannot interrupt once it has started“ definiert ist. Diese Definition soll Konzepte zulassen, die es möglich machen, wiederverwendbare Softwarekomponenten zu erstellen und diese nach Bedarf in verschiedenen Kombinationen zusammenzusetzen. Dazu wird der *SoftwareBus* eingeführt, der als Vermittler den Austausch von Nachrichten in der ESF-Architektur unterstützt. Die Konzepte aus ESF beinhalten auch Ideen zur Steuerung der Werkzeuge durch eine Prozeßsteuerungskomponente (vgl. [FO91], [FNO92]). Der Fokus liegt dabei auf der Definition geeigneter Mechanismen zur Prozeßkontrolle und nicht auf der Vermeidung von Inkonsistenzen zwischen Modell- und Prozeßausführung.

Der letzte funktionsorientierte Ansatz, der in diesem Kapitel vorgestellt wird, ist die Common Object Request Broker Architecture (kurz CORBA, vgl. z.B. [OMG98g]), ein Bestandteil der *Object Management Architecture* (kurz *OMA*, vgl. [OMG97]). Die bisher betrachteten Ansätze setzen Nachrichten dazu ein, Werkzeuge mit Werkzeugen kommunizieren zu lassen. Es handelte sich also um funktionsorientierte Ansätze in dem Sinne, daß kein direkter Bezug auf die Daten mehr genommen wird. Einzige Ausnahme ist ToolTalk, das mit seinem einfachen Objektbegriff erlaubt, eine Kapselung von Daten so vorzunehmen, daß die Menge der Funktionen, die auf den Daten arbeiten sollen, durch die Schnittstelle des Objekts gegeben sind. CORBA ist eine Architektur, die als Kern auch eine Vermittlungsinstanz für Nachrichten hat, aber als Sender und Empfänger von Nachrichten Objekte einführt. Der Ansatz geht weiter als die ToolTalk-Option, Objekte zu definieren, da in ToolTalk der Objektbegriff ein Konzept neben anderen ist, während es in einem CORBA basierten System ausschließlich Objekte gibt. Nachrichten werden nicht als unabhängige Elemente verschickt, sondern entsprechen Methodenaufrufen auf Objekten. Die Schnittstellen der Objekte werden dazu in einer speziellen Spezifikationssprache, der Interface Definition Language (kurz IDL, [OMG98e]), notiert.

Im Unterschied zu den bisher beschriebenen Ansätzen ist mit CORBA keine konkrete Implementierung verbunden. Es kann verschiedene Implementierungen geben, die durch ein spezielles Protokoll gemeinsam einsetzbar sind. Auch mit IDL spezifizierte Objektschnittstellen sind zunächst unabhängig von Programmiersprachen. IDL-Spezifikationen werden durch Generatoren in konkrete Programmiersprachen überführt (vgl. [OMG98g], Kapitel 19 - 24 für Abbildungen von IDL auf C, C++, Smalltalk, COBOL, Ada, Java), so daß die Low-Level-Techniken der Verbindung dem Entwickler verborgen bleiben können. Implementierungen in verschiedenen Programmiersprachen können gemeinsam eingesetzt werden. Auf diese Weise wird bezogen auf die Implementierung von neuen Objekten und deren Einbindung in bestehende Umgebungen schon eine weitgehende Flexibilität erreicht. Um diese zu steigern, sind zusätzliche Möglichkeiten vorgesehen, konkrete Aufrufe einer Methode erst zur Laufzeit zu formulieren. Dazu ist es möglich, die Spezifikation von Methoden in einem Repository abzulegen, aus dem diese Information dann abrufbar ist (vgl. [OMG98b], [OMG98c], [OMG98h]). Eine gute Ergänzung sind auch die als weitere Komponente der OMA spezifizierten *CORBAservices* ([OMG98a]). Dabei handelt es sich um allgemeine Dienste, die in einer auf CORBA aufgebauten Umgebung benötigt werden. Beispielsweise kann ein *Naming Service* ([OMG98d]) benutzt werden, der es erlaubt, Objekte durch Namen zu benennen und deren aktuelle Objektidentifizierer über den Namen wiederzufinden. Interessant ist im Zusammenhang dieser Arbeit auch der *Persistent Object Service* ([OMG98f]), der es ermöglicht, die mit einem Objekt verbundenen Daten persistent abzulegen. Als dritte Komponente der OMA sind die Common Facilities zu erwähnen, die benutzernahe Dienste, die in jeder Umgebung konfiguriert werden müssen, realisieren. Beispiele für benutzernahe Dienste sind Druckmöglichkeiten oder E-Mail-Dienste.

Die vorgestellten Ansätze haben ohne zusätzliche Hilfsmittel alle den Nachteil, daß die Werkzeuge a priori in Bezug auf den jeweiligen Ansatz implementiert sein müssen, um die angebotenen Möglichkeiten nutzen zu können. Dies widerspricht der Anforderung zur *Flexibilität der Werkzeugwahl*. Unter dem Aspekt der Wiederverwendbarkeit gemäß der Anforderung zur *Wiederverwendbarkeit der Anbindung* sind die vorgestellten Ansätze als Basis durchaus verwendbar. Field, SoftBench und SparcWorks bieten eine Infrastruktur, auf der applikationsspezifische Implementierungen (z.B. für PSEU) aufgesetzt werden können (zum Einsatz von Message Servern in SEU siehe auch [Bro92]). In ESF wird genau dieser Ansatz verfolgt, indem eine Art Prozeßwerkzeug die Arbeit des Entwicklers steuern und dann die geeigneten Werkzeuge aufrufen soll. Integration von nicht a priori integrierten Werkzeugen wird dabei nicht ausreichend betrachtet. Als weiterer Schwachpunkt ist für die bisher genannten Ansätze die starke Funktionsorientierung zu nennen, die keine geeignete Verbindung der Daten mit Zugriffsfunktionen darauf zuläßt. In diesem Punkt hat CORBA Vorteile. In einer PSEU, deren Aufgaben sich um Objekte (die zu erstellenden Dokumente) gruppieren, bei der aber trotzdem aus den in Kapitel 5.1.1 genannten Gründen nicht auf einen datenorientierten Ansatz zurückgegriffen werden kann, ist dieser Ansatz als eine mögliche Basis weiterführender Konzepte zur Integration von Werkzeugen von besonderem Interesse. Auch für CORBA gilt, daß es sich um Middleware handelt, zu deren Einsatz Programmieraufwand nötig ist. Dies schränkt die Nutzbarkeit für diese Arbeit ohne Ergänzung ein, da die Anforderung zur *Wiederverwendbarkeit der Anbindung* eine einfache Wiederverwendung fordert. Trotzdem ist CORBA von den vorgestellten Ansätzen der am besten geeignete.

5.1.2.2 Envelopes

Eine Technik, die dazu gedacht ist, Werkzeuge nachträglich zu integrieren, sind sogenannte *Envelopes* (auch *Wrapper*), wie sie u.a. von Kaiser, Feiler und Popovich in [KFP88] eingeführt werden (Beispiele für Anwendungen siehe u.a. [VK95], [SPA95b], [Ger94]). Ein Envelope legt eine Schale, einen Umschlag (Envelope), um das jeweils zu integrierende Werkzeug. Der Sinn dieser Schale ist, auf der einen Seite die vorhandene Schnittstelle des eingekapselten Werkzeugs unverändert zu nutzen und auf der anderen Seite einem Werkzeug, das mit dem eingekapselten Werkzeug integriert werden soll, eine saubere, integrierbare Schnittstelle anzubieten. Die jeweilige Schale wird somit direkt abgestimmt mit dem eingekapselten Werkzeug integriert und nimmt dann – als eigentliche Funktionalität – eine Schnittstellenumsetzung der einzelnen Schnittstellen aufeinander vor. Wie weitreichend die Integration der Werkzeuge mit diesem Verfahren ist, hängt von den zu integrierenden Werkzeugen ab. Diese bestimmen, wieviel ihrer Funktionalität sie nach außen geben bzw. wieviel Zugriff auf ihre Daten sie gewähren. Im Verhältnis zu den bisher vorgestellten Ansätzen handelt es bei Envelopes um eine andere Qualität der Technik. Es geht nicht darum, einen Weg zu definieren, über den sich Werkzeuge austauschen können. Stattdessen sollen die vorhandenen Techniken so angepaßt werden, daß damit eine Integration möglich ist. Envelopes zielen damit direkt auf a posteriori Integration. Die oben vorgestellten Ansätze erlauben zwar zum Teil a posteriori Integration, binden diese aber an die Verwendung einer bestimmten Implementierung als Integrationsmechanismus. Envelopes sind per se zunächst von der konkreten Implementierung unabhängig, da sie nur eine Technik auf eine andere abbilden. Der Preis, der für diese Flexibilität gezahlt werden muß, ist der zusätzliche Implementierungsaufwand, um den Envelope zu erstellen. Dieser summiert sich noch zusätzlich, wenn ein Werkzeug paarweise mit mehreren anderen integriert werden soll (in heterogenen PSEU der Normalfall: PE mit Werkzeug1, PE mit Werkzeug2, ...), da Envelopes ohne zusätzliche Maßnahmen spezielle Implementierungen für jeweils eine Integration sind.

Insbesondere der letzte Punkt hat dazu geführt, daß über Möglichkeiten nachgedacht wurde, die das Konzept auf bestimmte Problemfelder / Integrationsmechanismen eingrenzen und für diese Bereiche dann wiederverwendbare Lösungen unterstützen bzw. die Erstellung von Envelopes vereinfachen. Für SoftBench (siehe Kapitel 5.1.2.1) sind z.B. zwei konkrete Möglichkeiten geschaffen worden, die Erstellung von Envelopes zu unterstützen und damit die Integration von Werkzeugen in SoftBench zu erleichtern. Sie werden unter dem Namen *Encapsulator* zusammengefaßt ([Fro90], [Hew91]). Zum einen handelt es sich um eine Art Programmiersprache, die spezielle Konstrukte zur Integration von Unix-Werkzeugen, die nicht für die Verwendung in SoftBench implementiert wurden, enthält. Zum anderen wird eine Bibliothek angeboten, die die Nutzung der speziellen Konstrukte zur Integration von verschiedenen Programmiersprachen aus erlaubt. Über die Erfahrungen eines Experiments, auf dieser Technik eine Umgebung aufzubauen, wird in [BMZ+93] berichtet. In einer Fortsetzung dieses Experiments ist in einem zweiten Schritt auch untersucht worden, inwieweit Integrationsmechanismen auf der Basis von Message Servern einfach gegeneinander austauschbar sind ([ZB93]). Gerle beschreibt in [Ger94] ein Experiment, in dem er untersucht, inwieweit Verfahren entwickelt werden können, so daß verschiedene Message Server (hier: BMS und ToolTalk) gemeinsam in einer Umgebung benutzt werden können.

Envelopes schaffen eine gute Ausgangsbasis, um die Anforderung zur *Flexibilität der Werkzeugwahl* realisieren zu können. Durch die Flexibilität, die sie bieten, können auch verschiedene Integrationstechniken miteinander verbunden werden. Es ist sogar denkbar, durch Programmierung eines aufwendigen Envelopes, datenorientierte mit funktionsorientierten Techniken in Einklang zu bringen. Für den Einsatz in einer heterogenen PSEU, bei der die Anforderung zur *Wiederverwendbarkeit der Anbindung* erfüllt ist, reichen Envelopes alleine deshalb noch nicht aus, weil sie für jedes Werkzeug und möglicherweise für jede neue Umgebung aufgrund eines geänderten Prozesses neu implementiert werden müssen.

5.1.3 Zusammenfassung

In den vorangegangenen Kapiteln sind verschiedene Herangehensweisen und Techniken vorgestellt und eingestuft worden, die zur Integration voneinander unabhängig entwickelter Werkzeuge genutzt werden können. Dabei wurde festgestellt, daß keine der Techniken geeignet ist, die Prozeßausführung und die Modellausführung in einer PSEU zu integrieren. CORBA ist aufgrund seiner Konzeption, die eine lose Kopplung von Werkzeugen und Prozeßsteuerung erlaubt, besonders interessant als Basis für diese Arbeit. Auf CORBA basierend wird eine applikationsspezifische Schicht aufgebaut, die die speziellen Bedürfnisse aus den Anforderungen zur *Abbildung des Zustandsmodells*, zur *Übermittlung von Feedback*, zur *Übermittlung von Zustandsinformation*, zur *Zustandsbestimmung nach Werkzeugabbruch*, zur *Zustandsbestimmung nach PE-Abbruch* und zur *Auswertung von Feedback* befriedigt. Envelopes sind eine weitere Technik, die im Umfeld heterogener PSEU hilft, Probleme zu lösen. Hier wird überlegt, wie der notwendige Implementierungsaufwand zur Anbindung von Werkzeugen auf der Basis von Envelopes reduziert werden kann.

Es wurden keine Arbeiten betrachtet, die sich zwar im Umfeld der Integration von Werkzeugen in eine SEU bewegen, aber für diese Arbeit weniger relevante Aspekte beleuchten. In [BFW92], [BWF93] und [Wal92] wird z.B. teilweise ein Überblick darüber gegeben, was im Bereich der Integration im Zusammenhang mit SEU geleistet worden ist. Es wird insbesondere auf die Bedeutung der Verbindung des Softwareentwicklungsprozesses hingewiesen. Der Kern der Arbeiten ist aber Konfigurationsmanagement in SEU, was für diese Arbeit kein relevanter Aspekt ist.

5.2 Existierende PML/PSEU

Im vorangegangenen Kapitel 5.1 wurden allgemeine Ansätze für Integrationstechniken auf ihre Verwendbarkeit im Kontext einer PSEU untersucht. In diesem Kapitel werden mehrere PML und PSEU unter dem Aspekt der Anforderungen aus Kapitel 4 betrachtet. Die Auswahl der Beispiele berücksichtigt verschiedene Paradigmen (petrinetzbasiert, regelbasiert), ist aber ansonsten willkürlich.

Die folgenden Kapitel enthalten immer drei Teile. Im ersten wird die jeweils betrachtete PML eingeführt und dabei gezeigt, welche Möglichkeiten der Modellierung existieren. Im zweiten Teil wird eine diese PML verarbeitende PSEU vorgestellt. In dieser Betrachtung wird ein besonderes Augenmerk auf die Einbindung von Werkzeugen gelegt. Im dritten Teil wird analysiert, inwieweit die PML und die zugehörige PSEU das Anforderungsprofil aus Kapitel 4 erfüllen. Die Ergebnisse werden am Ende jedes Beispiels tabellarisch zusammengefaßt.

5.2.1 FunSoft und Melmac

Zur Modellierung und Ausführung von Softwareprozessen sind an der Universität Dortmund die Prozeßmodellierungssprache FunSoft und die PSEU Melmac entwickelt worden (vgl. z. B. [Gru91b, S. 49-124], [Dei93]). In FunSoft erstellte Prozeßmodelle, sogenannte FunSoft-Netze, können mit der Umgebung Melmac verwaltet und ausgeführt werden.

5.2.1.1 FunSoft

FunSoft ist eine auf Petri-Netzen basierende Prozeßmodellierungssprache. Die entstehenden Prozeßmodelle werden auch *FunSoft-Netze* genannt. Ein FunSoft-Netz besteht aus den folgenden Elementen (vgl. z.B. [Gru91b, S.192]):

- Aus einem Petri-Netz(S, T, F) mit *S* als Menge von *Stellen, T* als Menge von *Transitionen* und *F* als Menge von *Kanten*. Stellen werden durch Kreise repräsentiert und Transitionen durch Rechtecke. Kanten verbinden Stellen und Transitionen und werden durch Pfeile dargestellt.
- Aus einer Menge *J* von *Jobs*. Die Funktion T_j bindet Jobs an Transitionen. Ein Job zeichnet sich durch ein Ein-/Ausgabeverhalten, eine Vorbedingung, einen Aktionsteil und eine Nachbedingung aus. Der Job wird genau dann ausgeführt, wenn die zugehörige Transition schaltet.
- Aus einer Menge *O* von Definitionen zu *Objekttypen*. Die Funktion S_t bindet Objekttypen an Stellen. Dadurch wird ausgedrückt, daß Stellen nur mit Token (= Objekten) des entsprechenden Objekttyps belegt werden dürfen.
- Aus einer Menge von *Prädikaten*, die Vorbedingungen für das Schalten einer Transition beschreiben. Sie beziehen sich auf Objekte und werden durch die Funktion T_p an Transitionen gebunden.
- Aus einer *initialen Markierung* M_0, die die Anfangsbelegung des Petri-Netzes mit Token, d.h. Objekten im Prozeß, vorgibt. In der initialen Markierung sind die durch die Funktion S_t definierten Restriktionen zu beachten. Die an einer Stelle anliegenden Objekte unterliegen einer Ordnung, um verschiedene Zugriffsreihenfolgen (Stack, Queue, ...) realisieren zu können.

Eine Aktivität wie das Editieren eines Quelltextes oder die Teilnahme an einem Meeting wird durch einen Job in das Netz eingebracht. Die Realisierung eines auf einem Computer ausführbaren Jobs erfolgt durch Software wie Programme, Shell-Skripte oder auch eingekapselte Programme.

Transitionen können ganze Teilnetze repräsentieren. Dies ist ein Strukturierungsmittel, das z.B. dazu benutzt werden kann, einzelne Aktivitäten eines beschriebenen Prozesses auf verschiedenen Abstraktionsebenen zu modellieren.

In FunSoft-Netzen können sowohl Stellen als auch Transitionen und Kanten *Attribute* zugeordnet werden, die es erlauben, die Komplexität der Netze zu reduzieren. Ein Beispiel für solche Attribute ist das Attribut zu Stellen *access kind*. Dieses kann die Werte LIFO, FIFO oder RANDOM annehmen, was beschreibt, wie die Token abgearbeitet werden. Ein weiteres Beispiel bildet der *Typ* der Kanten. Dieses Attribut kann beispielsweise die Werte IN oder CO enthalten. Diese Werte beschreiben dann Datenfluß-

kanten von einer Stelle zu einer Transition. Dabei wird eine Kante vom Typ IN so interpretiert, daß vom Schalten der Transition betroffene Token verbraucht, d.h. in der Stelle gelöscht werden. Im Falle eines CO-Typen werden die betroffenen Token stattdessen kopiert.

Abbildung 5.1 zeigt ein Beispiel für einen *Software-Prozeß*. Der Prozeß modelliert den Design-Editieren-Übersetzen-Binden-Zyklus.

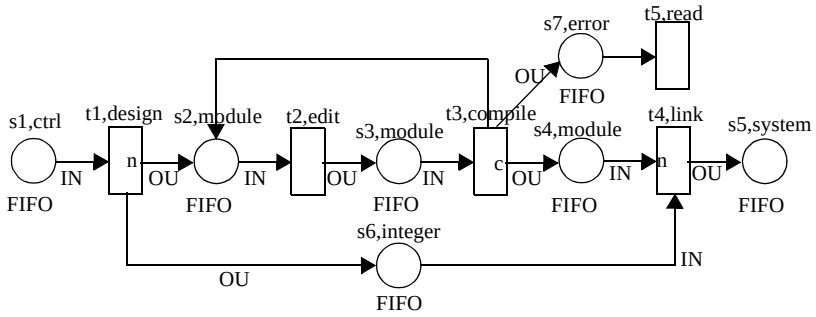


Abbildung 5.1 Ausschnitt aus einem FunSoft Prozeßmodell (aus [Gru91b], S. 194)

Die Netze werden durch ein kompliziertes Schaltverhalten von Transitionen sehr unübersichtlich. Daher ist die Beschreibung zusätzlicher Attribute zu den Transitionen vorgesehen, um deren Komplexität zu reduzieren. Die im Beispiel gezeigten und durch *n* oder *c* im Rechteck visualisierten Attributwerte lassen sich wie folgt erklären. Das n spezifiziert, daß n Token gelesen oder geschrieben werden müssen. Das c spezifiziert, daß ein komplexes Ausgangsschaltverhalten vorliegt und je nach Resultat des Jobs ein Schreiben von Token in eine oder mehrere der angegebenen Stellen erfolgt.

5.2.1.2 Melmac

Melmac setzt zur Spezifikation der Prozeßmodelle FunSoft-Netze ein und bietet Möglichkeiten zur Simulation und Analyse von Prozeßmodellen. Um die Spezifikation von Prozeßmodellen zu vereinfachen, unterstützt Melmac fünf verschiedene Sichten, die nach dem Prinzip des Separation-of-Concerns jeweils einen Teilaspekt der Prozeßmodellierung betrachten.

- Der *Structural View* erlaubt die Spezifikation der zuvor beschriebenen FunSoft-Netze mit den dazugehörigen Objekttypen.
- Der *Project Management View* erlaubt die Angabe von Informationen zu personellen Ressourcen, Terminen und Kosten. Wesentlicher Punkt ist dabei die Zuordnung von Personen zu ebenfalls im Project Management View definierten Rollen. Zeiten und Kosten werden entsprechend den Aktivitäten zugeordnet.
- Der *Profile View* legt für jede Aktivität auf der Basis einer festen Menge von Kriterien fest, welche Kenntnisse zu ihrer Durchführung benötigt werden. Auf der anderen Seite wird für jeden Entwickler ebenso festgelegt, welche Kenntnisse er besitzt. Ein Entwickler kann dann eine Aktivität ausführen, wenn sein persönliches Profil zumindest die für die Aktivität definierten Kenntnisse umfaßt.
- Der *Change View* dient zur Kennzeichnung der Teile eines Software-Prozesses, die geändert werden müssen. Die Änderung selbst wird nicht in diesem View durchgeführt.

- Der *Service View* beschreibt letztendlich die Einbettung von Werkzeugen in das Prozeßmodell.

Der Service View baut auf einer textuellen Beschreibung der Services auf, die bei der Durchführung einer Aktivität aufgerufen werden. Ein *Service* ist eine Abstraktion von Jobs auf der FunSoft-Ebene. Services werden in einer speziellen Sprache spezifiziert und können dann u.U. in C selbst programmiert werden, sofern es sich nicht um einen einfachen externen Aufruf handelt. In Abbildung 5.2 ist ein Beispiel einer Servicebeschreibung dargestellt.

```
BEGIN SERVICE MODULE LaTeX_Processing

EXPORT

LaTeX_doc_print( IN doc: LaTeXDocType)
    INDEP doc

LaTeX_doc_processing( IN doc: LaTeXDocType, OUT doc: LaTeXDocType)
    INDEP doc
    OUTDEP doc

<weitere Services>


BODY

TOOL SERVICE LaTeX_doc_print( IN doc: LaTeXDocType)
    INDEP doc
    INFORMAL DESCRIPTION:
        This service converts a LaTeX-File into Postscript and prints it to the default printer
    EXECUTION CODE
    <geeigneter C-Code>

<weitere TOOL SERVICE-Abschnitte>

END SERVICE MODULE LaTeX_Processing
```

Abbildung 5.2 Ein Beispiel für eine Servicebeschreibung in Melmac (vgl. [Dei93])

5.2.1.3 Problemlösungen in FunSoft und Melmac

Aktivitäten werden in FunSoft/Melmac durch Transitionen/Jobs repräsentiert. Für sie gilt, daß ihre Signatur auf der Modellebene durch die Formulierung der Objekttypen in den Stellen des Vor- bzw. Nachbereichs einer Transition spezifiziert ist. Eingabedokumente entsprechen den Objekttypen des Vorbereichs, Ausgabedokumente entsprechen den Objekttypen des Nachbereichs. Auf der Werkzeugebene wird durch die Services eine zu den Typen passende Implementierung bereitgestellt. Die Schnittstelle sagt aber nichts über das durch den Service gekapselte Werkzeug aus. In der Abbildung 5.2 wird z.B. ein Service `LaTeX_doc_print` beschrieben. Der Service hat ein Eingabedokument vom Type `LaTeXDocType` und keine Ausgabedokumente. Was sich hinter `<geeigneter C-Code>` verbirgt, ist nur dem C-Code zu entnehmen. Insbesondere liegt keine Spezifikation des gekapselten Werkzeugs vor, so daß nicht sicher davon ausgegangen werden kann, daß die Service-Schnittstelle mit der Schnittstelle eines aufgerufenen Werkzeugs identisch ist. Das einzige, was sicher gesagt werden kann, ist, daß der Service in der Lage ist, mit den von der Prozeßsteuerung gelieferten Eingabedokumenten das Werkzeug aufzurufen. Bei den Ausgabedokumenten kann es durchaus sein, daß mehr Dokumente erzeugt werden, als durch die Aktivität definiert wurde. Dies ist z.B. ein Problem, wenn für den Entwickler wichtige Information nicht in ein persistentes Dokument ausgegeben wird. Wie in Kapitel 4 festgestellt, können solche Ausgaben aber auch als Dokument interpretiert werden, so daß ohne deren Berücksich-

tigung wichtige Informationen wegfallen. Da in diesem Sinne keine Spezifikation der Werkzeuge existiert, ist die Anforderung zur *Spezifikation der Signatur* nur durch die PML erfüllt. Damit ist die Erfüllung der Anforderung zur *Relation der Signaturen* unmöglich.

Die Anforderung zur *Spezifikation des Leistungsumfangs* fordert von der PML, daß sie den Leistungsumfang der Aktivitäten (hier also der Transitionen/Jobs) und Werkzeuge spezifiziert. FunSoft spezifiziert den Leistungsumfang der Transitionen/Jobs. Dies wird durch die Typisierung der Token im Vor- und Nachbereich der jeweiligen Transition und die Definition, wie die Transition Token aus dem Vor- und Nachbereich behandelt (erzeugt, kopiert, ...), erreicht. Bezüglich der Werkzeuge gilt der schon bei der Anforderung zur *Spezifikation der Signatur* beschriebene Mangel, daß keine Spezifikation der Werkzeuge erstellt wird: „No formal specification of the functionality associated with services is given. This is only given by the semantics of the tool fulfilling services [...]“ ([Dei93]). Wird also ein Editor in einem Service gekapselt, so ist zum Zeitpunkt der Instanziierung unklar, ob es sich um einen „normalen“ Editor mit einer Funktionalität handelt, die zu weitreichend ist und damit möglicherweise den Prozeßablauf stört. Die Anforderung zur *Spezifikation des Leistungsumfangs* ist nur in dem Teil erfüllt, der die PML betrifft. Damit kann auch die Anforderung zur *Abbildung des Leistungsumfangs* nicht mehr erfüllt werden. Entsprechend kann die Umgebung keine Unterstützung zur Vermeidung versehentlicher Fehler aus einem unangepaßten Werkzeug bieten, d.h. die Anforderung zur *Kontrolle des Leistungsumfangs* kann nicht erfüllt werden.

Aggregationen sind im Structural View von Melmac als eine der beiden Möglichkeiten vorgesehen, strukturierte Dokumenttypen zu erstellen. Auf einzelne Komponenten einer Aggregation kann nur durch explizite Dekomposition der Aggregation zugegriffen werden. Dieser Dekomposition muß eine explizite Komposition im Prozeßablauf folgen, wenn wieder auf das Aggregat zugegriffen werden soll. Melmac bietet keine Möglichkeiten an, um auf Probleme, wie sie im Abschnitt *Aggregationen* (in Kapitel 4.1.2) beschrieben worden sind, einzugehen. Das Zitat aus [Dei93] im vorangegangenen Absatz belegt, daß bewußt auf Detailkenntnisse bezüglich der eingesetzten Werkzeuge verzichtet wird. Im Falle der Aggregationen verhindert das den Einsatz von Werkzeugen, wie sie im oben genannten Kapitel aufgezeigt wurden. Eine Unterstützung dieser Abweichungen vom spezifizierten Modell durch Melmac ist explizit ausgeschlossen. Eine Implementierung der Services, die Aufrufe auf das Aggregat z.B. auf die Komponenten abbildet, ist aufgrund der dazu vorher notwendigen Dekomposition unmöglich. Bezüglich der Anforderung zur *Implementierung modellierter Aggregationen* bedeutet das, daß Inkonsistenzen zwischen modellierten Aggregationen und ihrer Implementierung nicht möglich sind.

Für die Spezifikation des Zustandsmodells der Aktivitäten (Transitionen) aus dem Prozeßmodell und die Spezifikation des Zustandsmodells der Werkzeuge (in den Services gekapselt) gilt das gleiche wie beim Leistungsumfang und der Signatur. Auf der Seite des Modells ist eine Spezifikation durch das FunSoft-Netz gegeben. Dazu werden die Stellen des jeweiligen FunSoft-Netzes als Zustände interpretiert. Die Werkzeuge haben keine Spezifikation. Entsprechend gilt für die Unterstützung zur Laufzeit, daß Abweichungen vom vorgesehen Modell im Code des Services versteckt behandelt werden. Die Anforderung zur *Spezifikation des Zustandsmodells* wird also nur vom Modell und nicht von den Werkzeugen abgedeckt. Die Anforderung zur *Abbildung des Zustands-*

modells ist teilweise erfüllt, da es durch geschickte Programmierung des Service möglich ist, die geforderte Zustandsabbildung vorzunehmen. Eine Spezifikation der Abbildung ist aber nicht vorgesehen. Die Anforderung zur *Übermittlung von Zustandsinformation*, also daß auf jeden Fall die Prozeßsteuerung nach einer Aktivität über eine Zustandsänderung des bearbeiteten Dokuments informiert werden muß, ist erfüllt. Da die Dokumente über die Schnittstelle des Service an die Prozeßsteuerung in Form von Token in Stellen geschrieben werden, ist ein solches Feedback sichergestellt.

Auch beim Feedback über Änderungen zur Laufzeit einer Aktivität kümmert sich Melmac nicht um die eingesetzten Werkzeuge. Die Services werden mit einer zum Prozeßmodell passenden Schnittstelle programmiert. Die Implementierung enthält einen Zugriffsmechanismus auf das Prozeßmodell, so daß, wenn Ausgaben vom Service erwartet werden, diese an das Prozeßmodell weitergegeben werden. Eine Unterstützung für die Anbindung der Werkzeuge ist nicht vorgesehen. Im Service können die durch die Programmiersprache gegebenen Programmiermöglichkeiten genutzt werden, um mit einem gekapselten Werkzeug Informationen auszutauschen. Ist in der Schnittstelle des Service zur Prozeßsteuerung die Möglichkeit vorgesehen, Änderungen zur Laufzeit zu publizieren, hat man einen aufwendig zu erstellenden, aber funktionierenden Mechanismus zur Meldung prozeßrelevanter Änderungen. Dieses Vorgehen ist deswegen nicht befriedigend, weil der gleiche Aufwand bei einer erneuten Verwendung des gleichen Werkzeugs wieder betrieben werden muß, wenn nicht zufällig alle Parameter und sonstigen Benennungen genauso gewählt worden sind. Dazu kommt, daß bei Werkzeugen, die Informationen nicht auf der Basis eines Kommunikationsprotokolls weitergeben, unterstützende Maßnahmen wie z.B. Nachfragedialoge in jedem Service einzeln implementiert werden müssen. Der Zeitpunkt des Feedbacks ist fest definiert und liegt direkt nach Ausführung des Service. Die Anforderung zur *Übermittlung von Feedback* ist damit unbefriedigend abgedeckt. Damit verbunden ist auch die Anforderung zur *Auswertung von Feedback* nicht abgedeckt, da keine speziellen Mechanismen für diese Aufgabe vorgesehen sind.

Die Angabe von Parametern für den Aufruf eines Werkzeugs, die nicht aus dem Prozeßmodell abgeleitet werden, ist kein Problem. Da die Werkzeuge in den Services einzeln gekapselt werden, können einem Aufruf beliebige Parameter mitgegeben werden. Die Anforderung zur *Spezifikation der prozeßunabhängigen Eingaben* ist daher abgedeckt.

Im Bereich der Laufzeitfehler (Anforderungen zur *Zustandsbestimmung nach Werkzeugabbruch* und zur *Zustandsbestimmung nach PE-Abbruch*) und deren Erkennung zur Laufzeit (Anforderung zur *Erkennung von Werkzeugabbrüchen*) bietet Melmac keine Konzepte an. Bezüglich der Erkennung von Abbrüchen könnte noch geeigneter C-Code in jeden einzelnen Service implementiert werden. Allerdings würde dies nur bedingt Nutzen bringen, da seitens der Prozeßsteuerung keine Elemente enthalten sind, um auf die Rückmeldungen eines solchen Codestücks einzugehen.

Als Integrationstechnik stützt sich Melmac auf Envelopes ab. Services sind eine Form von Envelopes, deren Schnittstelle auf die Objekttypen aus dem Prozeßmodell abgestimmt formal beschrieben wird. Diese Technik ist sehr flexibel, wenn es um die Anbindung verschiedener Werkzeuge geht (vgl. Kapitel 5.1.2.2). So wie Envelopes in Melmac aufgebaut sind, ergeben sich zwei Kritikpunkte: In Melmac werden die Services als Implementierung eines Jobs aufgebaut. Sie werden in jedem neuen Prozeß neu aufgesetzt. Eine Wiederverwendung findet also höchstens über Copy-Paste Vor-

gänge statt, d.h. über Codevervielfältigung. Dies erzeugt einen hohen Aufwand für die Einbindung von Werkzeugen, der zudem wiederholt auftritt. Dies entspricht nicht der Anforderung zur *Wiederverwendbarkeit der Anbindung*. Der zweite Kritikpunkt ist, daß die Einbindung auf C-Programme beschränkt ist. Damit wird im günstigsten Fall das Einbinden von Werkzeugen erschwert, die über eine andere Technik einfacher anzubinden wären. Sollen z.B. auf ToolTalk basierende Programme eingebunden werden, dann erfordert eine Einbindung über C einen Aufwand, um die geeignete Infrastruktur aufzubauen, der durch die Verwendung von Konzepten, die mehr Flexibilität bieten, reduziert werden könnte. Die Anforderung zur *Flexibilität der Werkzeugwahl* ist damit nur unzureichend abgedeckt.

Einen Überblick über die Bewertung von FunSoft und Melmac nach den Anforderungen aus Kapitel 4 gibt die folgende Tabelle. Darin bedeutet ein +, daß die jeweilige

Konzeptebenen	*Spezifikation der Signatur*	*Relation der Signaturen*	*Spezifikation des Leistungsumfangs*	*Abbildung des Leistungsumfangs*	*Implementierung modellierter Aggregationen*	*Spezifikation des Zustandsmodells*	*Abbildung des Zustandsmodells*	*Spezifikation der prozeßunabhängigen Eingaben*	*Übermittlung von Feedback*	*Übermittlung von Zustandsinformation*	*Auswertung von Feedback*	*Kontrolle des Leistungsumfangs*	*Zustandsbestimmung nach Werkzeugabbruch*	*Zustandsbestimmung nach PE-Abbruch*	*Flexibilität der Werkzeugwahl*	*Wiederverwendbarkeit der Anbindung*	*Erkennung von Werkzeugabbrüchen*
FunSoft	O		O			O		+									
Melmac							O		O	+							
Services															O		
gesamt	O	-	O	-	-	O	O	+	O	+	-	-	-	-	O	-	-

+: Anforderung wird vollständig erfüllt
-: Anforderung ist nicht betrachtet (nur in der Zeile „gesamt“)
O: Ebene liefert Beitrag zur Erfüllung der Anforderung, Anforderung nicht vollständig abgedeckt
kein Eintrag: kein Beitrag zu dieser Anforderung (nicht in der Zeile „gesamt“)
: betrifft diese Ebene nicht

Anforderung durch Melmac bzw. FunSoft erfüllt ist. Ein Kreis bedeutet, daß ein Ansatz vorhanden ist, der ausgebaut oder verändert werden müßte, um die Anforderung zu erfüllen. Ein Minus besagt, daß für diese Anforderung keine Konzepte vorgesehen sind. Freigelassene Felder sagen aus, daß zur betrachteten Anforderung keine Konzepte im jeweiligen Bereich angeboten werden. Ein Minus (-) in der Zusammenfassung sagt dann aus, daß nirgends in FunSoft bzw. Melmac Konzepte zur Unterstüt-

zung der betrachteten Anforderung enthalten sind, also die einzelnen Felder freigelassen wurden. Ein Plus (+) in der Zusammenfassung gibt es, wenn für die jeweilige Anforderung ein + aufgezeigt wurde oder sich mehrere Kreise (O) so ergänzen, daß die Anforderung erfüllt wird.

5.2.2 EMSL und Marvel

Marvel ist eine PSEU, die als Prototyp an der Columbia University (NY) entwickelt worden ist (vgl. z.B. [KFP88], [Bar92], zu Teilaspekten auch [GK91], [VK95]). In Marvel werden Modelle ausgeführt, die in *MSL* (Marvel Strategy Language) bzw. *EMSL* (Extended MSL) formuliert werden.

5.2.2.1 EMSL

EMSL ist eine regelbasierte Prozeßmodellierungssprache. Prozesse werden in EMSL durch zwei zusammenhängende Modelle beschrieben:

- ein *Datenmodell* und
- ein *Prozeßmodell*.

Das Datenmodell beschreibt die im Prozeß auftretenden *Objekte*. Eigenschaften der Objekte werden in beschreibenden *Attributen* spezifiziert. Das Datenmodell besteht aus einer Art *Klassenhierarchie*, in der auch Mehrfachvererbung vorkommen darf. Zwischen den einzelnen Klassen können *Beziehungen* eingefügt werden. Auch Aggregationen können in Form einfacher Mengen formuliert werden. Jede Instanz einer Klasse wird mit einem Verzeichnis assoziiert. Die Struktur der Verzeichnisse ergibt sich aus den Beziehungen zwischen den Klasseninstanzen.

Abbildung 5.3 zeigt ein Beispiel aus einer Datenmodellbeschreibung zusammen mit der dazugehörigen Objekthierarchie. Die Objekte darin stehen zueinander in einer part-of-Beziehung.

Die Klassendefinition besteht aus dem Bezeichner der Klasse, gefolgt von einer Liste ihrer Superklassen. Im Anschluß daran ist die Definition der Attribute zu finden. Attribute werden durch einen Namen, ihren Typ sowie optional durch die Angabe eines Wertes beschrieben (vgl. Abbildung 5.3).

Die Attribute werden in vier Gruppen aufgeteilt:

1. *Statusattribute* zur Aufnahme der zur Verwaltung des Objekts benötigten Informationen (z.B. status und purpose),
2. *Datenattribute* zur Aufnahme des Namens der Datei, in der sich der Objektinhalt befindet (z.B. contents und afile),
3. *strukturelle Attribute* zur Definition der Objekthierarchie (z.B. modules und libraries) und
4. *Linkattribute* zur Definition typisierter Links zwischen zwei Objekten (z.B. libs und includes).

Das *Prozeßmodell* spezifiziert für die im Datenmodell definierten Objekte die *Aktivitäten*, die auf diesen ausgeführt werden können, sowie deren *Koordination*. Das Prozeß-

```
PROGRAM :: superclass ENTITY;
        modules : set_of MODULE;
        libraries : set_of LIB;
        includes : set_of INCLUDE;
        status : (built, notBuilt, error, none) = none;
end
REVERSABLE :: superclass ENTITY;
        locker : user;
        purpose : string;
        reservation-status : (checkedOut,
                        available, none) = none;
end
FILE :: superclass REVERSABLE;
        contents : text;
end
HFILE :: superclass FILE;
        contents : text = ".h";
end
```

```
CFILE :: superclass FILE;
        contents : text = ".c";
        includes : set_of link HFILE;
        status : (notCompiled, ..., initial) = initial;
        test-status : (tested, notTested,
                        failed) = notTested;
end
INCLUDE :: superclass ENTITY;
        includes : set_of INCLUDE;
        hfiles : set_of HFILE;
end
LIB :: superclass ENTITY;
        afile : binary = ".a";
end
MODULE :: superclass REVERSABLE;
        cfiles : set_of CFILE;
        modules : set_of MODULE;
        libs : set_of link LIB;
end
```

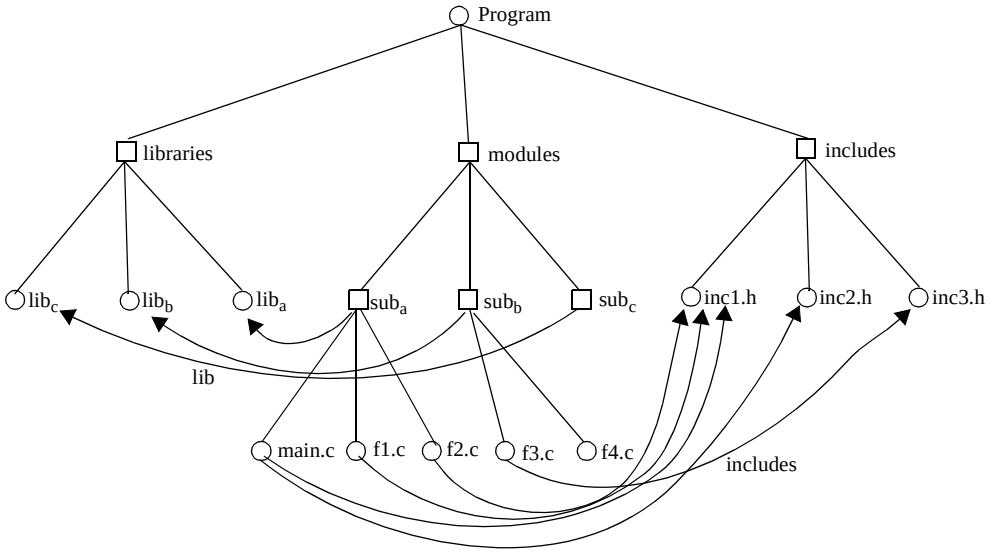


Abbildung 5.3 Beispiel für ein Datenmodell für Marvel (vgl. [Bar92])

modell basiert auf *Regeln*. Diese legen fest, wie die Ausführungsreihenfolge von Aktivitäten durch die Entwickler aussieht und ob das automatische Abarbeiten von Aktivitäten möglich ist.

Regeln im Prozeßmodell bestehen aus einer *Vorbedingung*, einem *Aktionsteil* und einer *Nachbedingung*. Vorbedingungen enthalten einen logischen Ausdruck, der zur Ausführung der Regel erfüllt sein muß. Die einzelnen Teile einer Vorbedingung beziehen sich auf von der Datenbank verwaltete Objekte bzw. deren Attribute.

Der Aktionsteil repräsentiert die Aktivitäten, die letztlich die einzelnen Entwicklungsschritte ausmachen. Die Aktivitäten werden in *autonome Aktivitäten*, d.h. solche, die ohne Beteiligung eines Software-Entwicklers ausgeführt werden können (z.B. ein Compiler), und *interagierende Aktivitäten*, d.h. solche, die eine menschliche Interaktion erfordern (z.B. ein Editor), unterteilt. In beiden Fällen wird der jeweiligen Aktivität ein Werkzeug zur Repräsentation in der Prozeßausführung zugeordnet.

Wird ein Werkzeug ausgeführt, das mit einer Aktivität assoziiert ist, so ergeben sich Konsequenzen, die die Projektinstanzen verändern. Die Werkzeugausführung verän-

dert damit die Attribute der Objekte der Modellausführung. Die Konsequenzen werden in der Nachbedingung spezifiziert.

Ein Beispiel für eine Regel ist in Abbildung 5.4 zu sehen. Die dort gezeigte Regel beschreibt, daß für die Übersetzung einer Instanz der Klasse `CFILE` die folgende Vorbedingung erfüllt sein muß: Entweder das `CFILE` oder aber ein zum `CFILE` gelinktes `HFILE` ist seit der letzten Übersetzung editiert worden. Falls diese Bedingung erfüllt ist, wird die Aktivität `compile` aufgerufen, wobei die entsprechenden `CFILE`- und `HFILE`-Inhalte als Parameter übergeben werden. Die Attribute, die von der Aktivität `compile` geschrieben werden, sind hinter dem Schlüsselwort `output` angegeben. In der Nachbedingung wird spezifiziert, daß die Attribute `status` und `object_timestamp` des `CFILE` neu gesetzt werden müssen, falls die Übersetzung erfolgreich beendet wird. Falls die Übersetzung hingegen fehlschlägt, wird dem `status`-Attribut der Wert `Error` zugewiesen.

```
compile [?f:CFILE]:
        # If C source file has been edited but not yet compiled,
        # or if any of the header files it is linked to has been
        # changed after the last compilation of the CFILE,
        # then it can be recompiled

(bind (?h to_all HFILE suchthat (linkto [?f.includes ?h])));

(exists ?h):
    (or (?f.status = NotCompiled) (?h.timestamp > ?f.object_timestamp))

{ compile ?f.contents ?h.contents „-g“;
    output: ?f.object_code ?f.error_msg }

(and (?f.status = Compiled)
    (?f.object_timestamp = CurrentTime));

(?f.status = Error);
```

Abbildung 5.4 Beispiel für eine Regel eines Prozeßmodells für Marvel (vgl. [Bar92])

Regeln können durch sogenannte *Strategien* (*strategies*) strukturiert werden. Strategien sind spezielle Regelmengen, die auf unterschiedliche Aufgaben ausgerichtet sind. Eine einzelne Strategie bietet eine Sicht auf einen konkreten Teil des Projekts. Strategien können z.B. rollenspezifisch sein und einem Projektmanager andere Regelmengen zur Verfügung stellen als etwa einem Programmierer (vgl. [KFP88], S. 45).

Die Einbettung von Werkzeugen erfolgt durch *Envelopes*, welche den Austausch von Parametern zwischen Prozeß und Werkzeug realisieren. In [GK91] und [Val94] bzw. [VK95] werden Verbesserungen des ursprünglich recht einfachen Mechanismus zur Kapselung von Werkzeugen zur Benutzung in Marvel vorgeschlagen (zur Technik von Envelopes vgl. auch Anhang 5.1.2.2). Envelopes werden danach in einer speziellen Sprache (Shell Envelope Language, SEL) geschrieben. Diese erzwingt eine klare Typisierung der Eingangsparameter im Sinne der EMSL-Objekte. Darüber hinaus erlaubt die Sprache die Definition von Ausgangsparametern, die über den UNIX-üblichen einzelnen Ganzzahl-Wert hinausgehen. SEL erlaubt allerdings nur die Kapselung von Werkzeugen, die zwischen Aufruf und Beendigung keine weiteren Informationen von der Umgebung benötigen. Entsprechend können SEL-gekapselte Werkzeuge auch keine Informationen zur Laufzeit an die Umgebung weitergeben. Die in [VK95] beschriebenen Erweiterungen gehen das Problem verschiedener Werkzeugtypen an, indem für einzelne Werkzeugtypen eine spezielle Unterstützung vorgesehen wird. So werden z.B. Werkzeuge unterstützt, für die ein Server nur genau einmal aufgerufen

wird und die Werkzeuginstanzen mehrerer Benutzer als Client sich diesen Server teilen können.

5.2.2.2 Marvel

Zur Eingabe von *Marvel*-Programmen stehen keine speziellen Editoren zur Verfügung. Zudem existieren bisher keine Werkzeuge zur Simulation und Analyse spezifizierter *Software-Prozesse*. Die Ausführung erfolgt durch einen eigens für Marvel entwickelten Interpreter, welcher die persistente Ablage der Daten, basierend auf der *Unix*-Dateistruktur sowie der „Gnu-Version des *Unix*-DBM byte managers“, unterstützt.

5.2.2.3 Problemlösungen in EMSL und Marvel

Wie für FunSoft und Melmac wird in diesem Kapitel wieder untersucht, welche Anforderungen aus Kapitel 4 für EMSL und Marvel erfüllt sind.

In EMSL werden Aktivitäten durch Aktionsteile der Regeln repräsentiert. Ihre Signatur (vgl. Kapitel 4.1.1) wird auf der Modellebene durch typisierte Parameter spezifiziert, die in den Aktionsteilen angegeben sind. In der Regel aus Abbildung 5.4 gibt es im Aktionsteil beispielsweise sowohl Eingabe- als auch Ausgabedokumente. `f.contents` (vom Typ `CFILE`) und `h.contents` (Menge mit Elementen vom Typ `HFILE`) sind die Eingabedokumente; die Ausgabedokumente, die nach dem Schlüsselwort `output` genannt werden, sind `f.object_code` und `f.error_msg`. Auf der Werkzeugebene wird durch die Envelopes eine zu den Typen passende Implementierung bereitgestellt. Wie schon bei den Melmac Envelopes sagt diese Schnittstelle aber nichts über das durch den Envelope gekapselte Werkzeug aus. Auch hier liegt keine Spezifikation des gekapselten Werkzeugs vor, so daß nicht sicher ist, ob die Envelope-Schnittstelle identisch mit der Schnittstelle eines aufgerufenen Werkzeugs ist. Die aus dieser Situation resultierenden Probleme wurden im Kontext von Melmac in Kapitel 5.2.1.3 dargestellt. Wie dort ist demnach die Anforderung zur *Spezifikation der Signatur* nur durch die PML erfüllt. Die Anforderung zur *Relation der Signaturen* kann entsprechend auch von EMSL und Marvel nicht erfüllt werden.

Die Anforderung zur *Spezifikation des Leistungsumfangs* fordert von der PML, daß sie den Leistungsumfang der Aktivitäten (Aktionsteile der Regeln) und Werkzeuge spezifiziert. EMSL spezifiziert den Leistungsumfang der Aktionsteile insofern, als dort angegeben wird, welche Dokumente und Beziehungen wie zu lesen und zu schreiben sind. Da für die Werkzeuge keine Spezifikation erstellt wird (siehe oben), wird die Anforderung zur *Spezifikation des Leistungsumfangs* nur für die Aktivitäten durch die PML erfüllt. Da ein Abgleich des Leistungsumfangs von Aktivität und Werkzeug nicht vorgesehen ist, wird die Anforderung zur *Abbildung des Leistungsumfangs* nicht erfüllt. Weiterhin bietet auch die Umgebung keine Unterstützung zur Vermeidung versehentlicher Fehler aus einem unangepaßten Werkzeug, womit auch die Anforderung zur *Kontrolle des Leistungsumfangs* nicht erfüllt ist.

Bezüglich der Anforderung zur *Implementierung modellierter Aggregationen* ist festzustellen, daß Marvel keine Aggregationen zur Strukturierung von Dokumenten kennt. Dadurch kann es auch nicht zu den in Abschnitt *Aggregationen* (in Kapitel 4.1.2) beschriebenen Problemen kommen. Andererseits sind Aggregationen wichtige Hilfsmittel, um a) den semantischen Gehalt des Prozeßmodells zu erhöhen und b) unnötige

Überprüfungen, wie z.B. die in Kapitel 4.1.2.1 beschriebenen, zu vermeiden. In diesem Sinne ist die Anforderung zur *Implementierung modellierter Aggregationen* so zu interpretieren, daß zunächst einmal ein Aggregationskonzept in der benutzten PML vorhanden sein sollte. Dies ist nicht erfüllt.

Für die Aktivitäten ist die Spezifikation des Zustandsmodells in der Nachbedingung des Aktionsteils der Regel enthalten, die die Aktivität repräsentiert. Die Werkzeuge werden außer durch die Angabe ihres Aufrufs nicht spezifiziert. Entsprechend gilt für die Unterstützung zur Laufzeit, daß Abweichungen vom vorgesehenen Modell im Code des Envelope behandelt werden. Die Anforderung zur *Spezifikation des Zustandsmodells* ist also nur vom Modell und nicht von den Werkzeugen abgedeckt. Die Anforderung zur *Abbildung des Zustandsmodells* ist nur bedingt erfüllt, da es durch geschickte Programmierung eines Envelopes möglich ist, die geforderte Zustandsabbildung vorzunehmen. Eine Spezifikation der Abbildung ist aber nicht vorgesehen. Die Anforderung zur *Übermittlung von Zustandsinformation*, die fordert, daß die Prozeßsteuerung über den Zustand der einzelnen Dokumente nach deren Ausführung informiert werden muß, ist bei der Definition der Envelopes mit SEL erfüllt.

Feedback wird am Ende der Ausführung des Envelopes durch Veränderung der Objektattribute gegeben. Dies ist relativ unflexibel, da nicht die Möglichkeit besteht, Feedback schon zur Laufzeit zu geben und von der Seite der Prozeßsteuerung gegebenenfalls darauf zu reagieren. Für die Anbindung an die Werkzeuge gilt, daß SEL auf UNIX-Kommandozeilenwerkzeuge abgestimmt ist. Über die normalen Aufrufmechanismen aus einer Shell-Sprache heraus ist keine Unterstützung für die Anbindung der Werkzeuge vorgesehen. Die Anforderung zur *Übermittlung von Feedback* ist für die Werkzeuge, die unterstützt werden, abgedeckt (vgl. Bemerkungen zur Anforderung zur *Flexibilität der Werkzeugwahl* weiter unten). Die Anforderung zur *Auswertung von Feedback* ist nicht erfüllt, da die durchgeführten Änderungen nicht auf ihre Prozeßkonformität überprüft werden.

Die Angabe von Parametern für den Aufruf eines Werkzeugs, die nicht aus dem Prozeßmodell abgeleitet werden, ist problemlos möglich. Ergänzende Parameter werden direkt mit in den Aktionsteilen angegeben. Die Anforderung zur *Spezifikation der prozeßunabhängigen Eingaben* ist daher abgedeckt.

Keine Konzepte bietet Marvel im Bereich der Laufzeitfehler (Anforderung zur *Zustandsbestimmung nach Werkzeugabbruch* und Anforderung zur *Zustandsbestimmung nach PE-Abbruch*) und deren Erkennung zur Laufzeit (Anforderung zur *Erkennung von Werkzeugabbrüchen*).

Als Integrationstechnik stützt sich Marvel auf Envelopes ab. Im Marvel-Kontext sind Envelopes auch das erste Mal eingesetzt worden, um heterogene PSEU zusammenstellen zu können. So wie Envelopes in Marvel aufgebaut sind, ergibt sich allerdings eine zu enge Kopplung von Prozeß und eingesetztem Werkzeug. Einerseits könnten die Envelopes nämlich allgemeiner geschrieben werden, was sich durch die Übergabe erweiterter Parameter aus den Regeln anbieten würde. Damit wären sie leichter wiederverwendbar. Das führt andererseits aber dazu, daß aufrufspezifische Informationen im Prozeß notiert werden. Wird das gekapselte Werkzeug getauscht, muß also zwangsläufig (bis auf Sonderfälle) die Prozeßdefinition verändert werden. Die Reduzierung der einsetzbaren Werkzeuge auf UNIX-Kommandozeilenwerkzeuge widerspricht zudem der Anforderung zur *Wiederverwendbarkeit der Anbindung*, führt aber aufgrund der

geringen Komplexität zu einfachen Envelopes. Die Bindung der Envelopes an die Typen des Datenmodells, dessen Dokumente sie bearbeiten sollen, relativiert die oben erwähnte Unabhängigkeit der Envelopes unter dem Aspekt der Wiederverwendung. Da diese Typen nicht von Prozeßmodell zu Prozeßmodell gleich sind, ist darin eine direkte Kopplung von Envelope und Prozeß festgelegt. In dieser Implementierung sind demnach die Anforderung zur *Flexibilität der Werkzeugwahl* und die Anforderung zur *Wiederverwendbarkeit der Anbindung* nur teilweise erfüllt.

Der Vergleich mit den Anforderungen aus Kapitel 4 wird in der folgenden Tabelle zusammengefaßt. Die Ergebnisse sind wie im vorangegangenen Kapitel notiert.

Konzeptebenen	*Spezifikation der Signatur*	*Relation der Signaturen*	*Spezifikation des Leistungsumfangs*	*Abbildung des Leistungsumfangs*	*Implementierung modellierter Aggregationen*	*Spezifikation des Zustandsmodells*	*Abbildung des Zustandsmodells*	*Spezifikation der prozeßunabhängigen Eingaben*	*Übermittlung von Feedback*	*Übermittlung von Zustandsinformation*	*Auswertung von Feedback*	*Kontrolle des Leistungsumfangs*	*Zustandsbestimmung nach Werkzeugabbruch*	*Zustandsbestimmung nach PE-Abbruch*	*Flexibilität der Werkzeugwahl*	*Wiederverwendbarkeit der Anbindung*	*Erkennung von Werkzeugabbrüchen*
MSL	O		O			O		+	+								
Marvel							O			+							
SEL-Envelopes															O	O	
gesamt	O	-	O	-	-	O	O	+	+	+	-	-	-	-	O	O	-

+: Anforderung wird vollständig erfüllt
-: Anforderung ist nicht betrachtet (nur in der Zeile „gesamt“)
O: Ebene liefert Beitrag zur Erfüllung der Anforderung, Anforderung nicht vollständig abgedeckt
kein Eintrag: kein Beitrag zu dieser Anforderung (nicht in der Zeile „gesamt“)
: betrifft diese Ebene nicht

5.2.3 SLANG und SPADE

SPADE ist eine PSEU, die aus Projekten bei CEFRIEL und der Politecnico Milano hervorgegangen ist (vgl. [BBFL94], [SPA95a], Kapitel 6, zu einzelnen Aspekten [SPA95b]). SLANG (SPADE Language) ist die Prozeßmodellierungssprache zur Formulierung von Prozeßmodellen für SPADE (vgl. [SPA95a], [BFG93]).

5.2.3.1 SLANG

Die Prozeßmodellierungssprache *SLANG* ist eine petrinetzbasierte Sprache. Ein in SLANG formuliertes Prozeßmodell besteht aus zwei Teilen ([SPA95a]):

- aus den *Prozeßtypen* und
- aus den *Prozeßaktivitäten*.

Die *Prozeßtypen* sind eine Menge von Abstrakten Datentypen (ADT), die in einer objektorientierten Generalisierungshierarchie angeordnet sind. Die ADT-Definitionen benutzen das objektorientierte Typsystem, das SLANG als Basis anbietet. Sie beschreiben die gesamten im Prozeßmodell vorkommenden Daten.

Atomare Typen des SLANG-Typsystems sind `integer`, `char`, `boolean`, `string` und `bits`. Auf diesen Typen aufbauend können komplexere *strukturierte Typen* durch die Konstruktoren `tuple`, `set`, `unique set` und `list` aufgebaut werden. In der nächsten Stufe können unter Benutzung von atomaren und strukturierten Typen dann abstrakte Datentypen (SLANG ADT) definiert werden. Diese stützen sich auf den Konstrukten ab, die im objektorientierten Datenbanksystem *O2* als Konstruktor von Klassen vorgesehen sind. O2 wird zur Ablage von SLANG-Modellen verwendet. Die Elemente, aus denen ein solcher Konstruktor aufgebaut wird, sind:

- ein eindeutiger Bezeichner der Klasse,
- die Menge der Klassen, von denen die Klasse erbt,
- eine Typbeschreibung und
- eine Liste von Methoden.

Die Spezifikation eines SLANG ADT wird in Abbildung 5.5 gezeigt.

```
class Engineer inherit Person
type tuple (knownLanguages: set (string))
method public knowsLanguage (language: string): boolean {
    boolean knows;
    knows = language in self->knownLanguages;
    return knows; }
end;
```

Abbildung 5.5 Beispiel für einen SLANG ADT (vgl. [SPA95a])

Weiterhin gibt es neben diesen benutzerdefinierten Typen auch noch einzelne durch die Sprache vordefinierte und damit vom spezifischen Prozeßmodell unabhängige Typen. Instanzen aller Typen (benutzer- und sprachdefinierte) bilden, wie in FunSoft-Netzen, die in den Aktivitäten laufenden Token.

Die *Prozeßaktivitäten* sind entsprechend eine Menge von Aktivitätsdefinitionen, wobei jede Aktivität durch ein Petri-Netz-Fragment repräsentiert wird. Eine Aktivität ist definiert als 5-Tupel *Activity = (P, T, A, I, L)*. Dabei gilt:

- *P* ist eine Menge der Stellen des Petri-Netzes,
- *T* ist eine Menge von Transitionen,
- *A* ist die Menge der im Netz enthaltenen Kanten,

- *I* ist eine Menge von Aktivitätsaufrufen und
- *L* ist eine Menge von Links.

Jede Stelle hat einen Bezeichner, der innerhalb einer Aktivität eindeutig ist, und einen Typ. Eine Stelle kann nur solche Token aufnehmen, die mit ihrem Typ oder einem Subtypen davon übereinstimmen.

Bei den Stellen werden *normale Stellen* und *Benutzerstellen* unterschieden. Normale Stellen verändern die in ihnen aufgenommenen Token nur dadurch, daß eine mit ihnen in Verbindung stehende Transition feuert. Sie werden durch einen einfachen Kreis symbolisiert. Benutzerstellen hingegen reagieren auf Ereignisse der Umgebung, wie z.B. Benutzereingaben. Benutzerstellen werden durch einen Doppelkreis symbolisiert. Sie dürfen nur in der Eingangsmenge einer Transition verwendet werden.

Transitionen repräsentieren Ereignisse im Petri-Netz. Transitionen werden in zwei Gruppen getrennt. Sogenannte *white transitions* bilden Ereignisse ab, die innerhalb der Modellausführung entstehen. *Black transitions* stehen umgekehrt für Ereignisse, in die die Prozeßausführung involviert ist. Sie stellen die Werkzeugaufrufe dar.

Die Ausführung von Transitionen ist atomar in dem Sinne, daß außerhalb der Transition keine Zustandsinformation aus der Ausführung der Transition bekannt gemacht wird.

Jede Beschreibung einer Transition beinhaltet einen sogenannten *Guard* und eine *Action*. Der Guard ist ein boolescher Ausdruck auf der Basis der Token in den Stellen der Eingangsmenge der Transitionen. Ist der Ausdruck wahr, kann die Transition feuern. Das Feuern besteht dann in der Ausführung der Transition, die an der Action spezifiziert ist. Diese beschreibt, wie die erzeugte Menge der Token sich aus den Token der Eingangsmenge berechnet. An black transitions werden spezielle Actions spezifiziert, die unter anderem die Ausführung des Werkzeugs beschreiben.

In SLANG werden Kanten in drei Gruppen unterteilt. *Normale Kanten* entsprechen den üblichen in normalen Petri-Netzen verwendeten Kanten. Sie werden als nicht unterbrochene Linie dargestellt. Bidirektionale Kanten erlauben eine vereinfachte Schreibweise für eine Stelle, die sowohl in der Eingangs- als auch in der Ausgangsmenge einer Transition liegt.

Nur-Lesende Kanten beschreiben, daß eine Transition aus einer Stelle in ihrer Eingangsmenge ein Token liest, ohne daß es aus der Stelle entfernt wird. Diese Kantenform wird durch eine gestrichelte Linie repräsentiert.

Eine *Überschreibende Kante* bewirkt, daß alle Token ihrer Ausgangsmenge durch die durch das Feuern der Transition erzeugten Token überschrieben werden. Eine solche Kante wird durch eine doppelte Spitze dargestellt.

Aktivitäten können andere Aktivitäten aufrufen. Die damit verbundene implizite Hierarchie von Aktivitäten kann als Strukturierungsmittel zur Reduzierung der Komplexität von Prozeßmodellen genutzt werden. Aktivitätsaufrufe realisieren dieses Strukturierungsmittel. Ein Aktivitätsaufruf besteht auf dieser Ebene aus einer Schnittstellenbeschreibung der aufzurufenden Aktivität, die dann in die aktuelle Aktivität eingebunden wird.

Links sind spezielle Kanten, die Stellen mit Aktivitätsaufrufen verbinden.

Die Anbindung an Werkzeuge im Rahmen von black transitions erfolgt in deren Action-Teil. Dieser wird in einen Prolog- und einen Epilog-Teil unterteilt. Der Prolog-Teil enthält alle notwendigen Vorbereitungen für den Werkzeugaufruf. Insbesondere wird eine vordefinierte Variable mit dem Namen extAction so gesetzt, daß sie einen String enthält, der den Aufruf des externen Aufrufs in der Art beschreibt, wie er auch auf der Kommandozeile angegeben würde. Spezielle Werkzeuge, die wie SLANG auf der Datenbank O2 arbeiten, können als solche gekennzeichnet werden und können dann noch zusätzliche Parameter in Form von O2-Objekten übernehmen.

5.2.3.2 SPADE

SPADE ist eine Umgebung, in der SLANG-Prozeßmodelle erstellt, analysiert und ausgeführt werden können. Sie umfaßt dazu u.a. einen Editor, um SLANG-Modelle zu erstellen, und einen Interpreter, um SLANG-Modelle auszuführen.

Zur Anbindung der Werkzeuge wird, wie oben beschrieben, der Aufruf des Werkzeugs angegeben. Eine spezielle Komponente der Umgebung, das SPADE-Communication Interface, erlaubt es zusätzlich, auch sogenannte servicebasierte Werkzeuge aufzurufen. Bei diesen werden zur Laufzeit Teilfunktionalitäten über eine bestimmte Schnittstelle auf der Basis von Nachrichten aufgerufen und deren Ergebnisse ebenso wie beim Aufruf auf der Basis von Nachrichten ausgewertet (zur Technik von Nachrichten vgl. Kapitel 5.1.2.1). SPADE bietet dazu eine ToolTalk- und eine DEC Fuse-Anbindung an. Auf dieser Basis können einzelne Services eines externen Werkzeugs wie ein Werkzeug behandelt und im SLANG-Netz beschrieben werden. Die Atomarität eines externen Werkzeugs wird dadurch teilweise aufgehoben.

5.2.3.3 Problemlösungen in SLANG und SPADE

SPADE ist wie FunSoft ein Ansatz, der auf Petri-Netzen basiert. Die Konzepte in den Bereichen, die die Spezifikation von Aktivitäten betreffen, ähneln denen von FunSoft. Die Spezifikation von Signatur, Leistungsumfang und Zustandsmodell im Prozeßmodell sind ebenso wie bei FunSoft gegeben. Zur Begründung können die Betrachtungen aus Kapitel 5.2.1.3 herangezogen werden.

Für die Werkzeugseite gilt genau wie in den vorangegangenen PSEU-Beispielen, daß keine Spezifikation der benutzten Werkzeuge eingesetzt wird, um zu überprüfen, ob ein Werkzeug mit einer Aktivität (black transition) zusammenpaßt. Damit sind die Anforderungen zur *Spezifikation der Signatur*, zur *Spezifikation des Leistungsumfangs* und zur *Spezifikation des Zustandsmodells*, wie schon in den anderen Ansätzen, nur teilweise erfüllt. Auch für die Abbildungen aus den Anforderungen zur *Relation der Signaturen*, zur *Abbildung des Leistungsumfangs* und zur *Abbildung des Zustandsmodells* ergeben sich keine neuen Aspekte. Die ersten beiden Anforderungen sind also nicht erfüllt (Begründungen siehe vorangegangene Kapitel). Die Anforderung zur *Abbildung des Zustandsmodells* ist teilweise erfüllt. Entsprechend kann die Umgebung keine Unterstützung zur Vermeidung versehentlicher Fehler aus einem unangepaßten Werkzeug bieten, d.h. die Anforderung zur *Kontrolle des Leistungsumfangs* kann nicht erfüllt werden.

Die Möglichkeiten, Aggregationen zu modellieren, sind durch die Verwendung der O2-Typen gegeben. Es werden darauf aber keine Konzepte aufgesetzt, die die Anbindung von Werkzeugen im Sinne der Anforderung zur *Implementierung modellierter Aggregationen* umsetzen. Die Anforderung zur *Implementierung modellierter Aggregationen* ist somit nicht erfüllt.

Feedback wird in der Implementierung der black transitions von SPADE durch den Epilog-Teil realisiert, der einen Return-Wert zurückgeben kann. Dieser Wert ist im Fall eines normalen UNIX-Kommandozeilen-Werkzeugs der Rückgabewert des Werkzeugs oder leer. Änderungen prozeßrelevanter Daten werden nicht bemerkt. Black-Box-Tools können auch noch Parameter im benutzten Repository (O2) übergeben werden. Dabei kommt es aber zu den in Kapitel 5.1.1.1 identifizierten Problemen. Mit diesen Mechanismen ist insbesondere im Fall der Verwendung von üblichen UNIX-Werkzeugen kein Feedback sicherzustellen. Die Anforderung zur *Übermittlung von Feedback* kann also nicht abgedeckt werden. Damit verbunden ist auch die Anforderung zur *Auswertung von Feedback* nicht abgedeckt, da keine speziellen Mechanismen für diese Aufgabe vorgesehen sind. Im Gegensatz dazu reicht der Epilog, um sicherzustellen, daß auf jeden Fall eine Information über den Zustand gegeben werden kann, auch wenn sie nicht durch das Werkzeug gegeben werden sollte. Die Anforderung zur *Übermittlung von Zustandsinformation* ist damit erfüllt.

Die Angabe von Parametern für den Aufruf eines Werkzeugs, die nicht aus dem Prozeßmodell abgeleitet werden, ist problemlos möglich. Da die Werkzeuge direkt in den black transitions aufgerufen werden, können dabei beliebige Parameter mitgegeben werden. Die Anforderung zur *Spezifikation der prozeßunabhängigen Eingaben* ist daher abgedeckt.

Im Bereich der Laufzeitfehler (Anforderungen zur *Zustandsbestimmung nach Werkzeugabbruch* und zur *Zustandsbestimmung nach PE-Abbruch*) und deren Erkennung zur Laufzeit (Anforderung zur *Erkennung von Werkzeugabbrüchen*) bietet SPADE keine Konzepte an.

Die zur Integration verwendeten black transitions basieren auf dem Envelope-Konzept. Die hier vorhandene Variante ist im Vergleich die ausdrucksschwächste. Während Melmac vollständige C-Programme als Envelope einsetzen kann und Marvel sich auf der SEL abstützt, können in black transitions wenige vordefinierte Parameter gesetzt, ein Aufruf ausgeführt und vordefinierte Antwortparameter ausgewertet werden. Dieses System funktioniert nur, wenn viel Aufwand in die Auswahl bereits passender Werkzeuge gelegt wird. SPADE ist wiederum variabler in dem Sinn, daß verschiedene Anbindungskonzepte unterstützt werden: UNIX-Kommandozeilenwerkzeuge oder DEC FUSE bzw. ToolTalk basierte Werkzeuge, die jeweils auf O2 aufsetzen können oder nicht. Wie in Kapitel 5.1 schon ausgeführt wurde, sichert diese Form der Variabilität nicht die Flexibilität, viele verschiedene Techniken einbinden zu können. Die Anforderung zur *Flexibilität der Werkzeugwahl* ist somit teilweise erfüllt. Eine Wiederverwendbarkeit der Anbindung von Werkzeugen ist in SPADE nicht ersichtlich. Die Anforderung zur *Wiederverwendbarkeit der Anbindung* kann mit den in SPADE vorgesehenen Mitteln nicht abgedeckt werden.

Die Bewertung wird in der folgenden Tabelle zusammengefaßt. Die Ergebnisse sind wie in den vorangegangenen Kapiteln notiert.

Konzeptebenen	*Spezifikation der Signatur*	*Relation der Signaturen*	*Spezifikation des Leistungsumfangs*	*Abbildung des Leistungsumfangs*	*Implementierung modellierter Aggregationen*	*Spezifikation des Zustandsmodells*	*Abbildung des Zustandsmodells*	*Spezifikation der prozeßunabhängigen Eingaben*	*Übermittlung von Feedback*	*Übermittlung von Zustandsinformation*	*Auswertung von Feedback*	*Kontrolle des Leistungsumfangs*	*Zustandsbestimmung nach Werkzeugabbruch*	*Zustandsbestimmung nach PE-Abbruch*	*Flexibilität der Werkzeugwahl*	*Wiederverwendbarkeit der Anbindung*	*Erkennung von Werkzeugabbrüchen*
SLANG	O		O			O		+									
SPADE							O			+							
black transitions															O		
gesamt	O	-	O	-	-	O	O	+	-	+	-	-	-	-	O	-	-

+: Anforderung wird vollständig erfüllt
-: Anforderung ist nicht betrachtet (nur in der Zeile „gesamt“)
O: Ebene liefert Beitrag zur Erfüllung der Anforderung, Anforderung nicht vollständig abgedeckt
kein Eintrag: kein Beitrag zu dieser Anforderung (nicht in der Zeile „gesamt“)
: betrifft diese Ebene nicht

5.2.4 PRO-ART

PRO-ART (Process and RepOsitory-based Approach to Requirements Traceability) ist eine PSEU, die speziell auf den Teilaspekt der Anforderungsanalyse zugeschnitten ist. PRO-ART ist an der RWTH Aachen entwickelt worden.

5.2.4.1 PRO-ART-Adaption von SLANG

PRO-ART nutzt als Prozeßmodellierungssprache eine angepaßte Version von SLANG (vgl. Kapitel 5.2.3.1). Dabei ist die Sprache in ihrer Form beibehalten worden. Da das Ziel in PRO-ART allerdings ist, die Modellausführung und die Prozeßausführung direkt miteinander zu integrieren, sind einzelne Abstraktionen eingebaut worden. Diese erlauben es, den Prozeß, der inhärent in Werkzeugen enthalten ist, im Modell mit abzubilden und somit eine Art gemeinsamen Prozeß von Projektebene und eingesetzten Werkzeugen zu erhalten. In PRO-ART werden dazu im Unterschied zu SLANG alle Stellen in drei Gruppen untergliedert, die speziell diesem Zweck genügen.

5.2.4.2 PRO-ART

Die Umgebung PRO-ART besteht zunächst einmal aus dem bereits in SPADE benutzten SLANG-Interpreter. Dieser ist an die veränderte Werkzeuganbindung angepaßt worden, indem er nun Black-Box-Werkzeuge ansprechen kann, die eine sehr enge Integration mit seinem eigenen Datenmodell haben. Diese enge Integration wird darüber erreicht, daß ein spezielles Protokoll implementiert ist, dem alle Werkzeuge in PRO-ART folgen, um jeden im Werkzeug durchgeführten Schritt direkt mit der Prozeßausführungskomponente, hier also dem SLANG-Interpreter, abzustimmen.

Weiterhin gehören zur Umgebung mehrere Editoren, mit denen das Zusammenspiel von Werkzeugen und SLANG-Interpreter, d.h. also das Zusammenspiel von Prozeß- und Modellausführung, exemplarisch gezeigt wird.

5.2.4.3 Problemlösungen in PRO-ART

Die Art und Weise, wie in PRO-ART Werkzeuge eingebunden werden, ist im Sinne der Probleme aus Kapitel 4 sehr effektiv. Dabei muß man allerdings im Auge haben, daß es sich um eine a priori Integration mit speziell für das implementierte Protokoll entworfenen Werkzeugen handelt. Für eine solche Umgebung wurde bereits in der Motivation dargelegt, daß die Integrationsqualität besser ist als im a posteriori-Fall einer heterogenen PSEU. Es wurde aber auch schon festgestellt, daß insbesondere bei Software-Entwicklungsumgebungen, die nicht sehr stabil bezüglich der ablaufenden Prozesse und Werkzeuge sind, eine solche a priori Anbindung nicht die Flexibilitätsanforderungen an eine PSEU erfüllt. Ein solcher Anbindungsmechanismus ist ein sehr gutes Beispiel dafür, wie eine enge Kopplung von Prozeß- und Modellausführung aussehen kann, kann aber nicht als alleiniger Mechanismus existieren.

Für eine solche Implementierung paßt auch der Satz von Anforderungen aus Kapitel 4 nicht. Die Anforderungen sind darauf ausgerichtet, in einem Umfeld von Werkzeugen, die nicht speziell für die Verwendung in einer PSEU implementiert wurden, das Zusammenstellen einer PSEU zu ermöglichen. Diese Vorbedingung ist hier nicht erfüllt. Es wird daher an dieser Stelle für PRO-ART auf einen Einzelabgleich mit den Anforderungen aus Kapitel 4 verzichtet.

5.2.5 ESCAPE und Merlin

ESCAPE ist eine PML, die an der Universität Dortmund entwickelt worden ist ([Jun95]). Merlin ist eine PSEU, die in der Lage ist, in ESCAPE formulierte Modelle auszuführen. Auch sie wurde an der Universität Dortmund entwickelt. Auf ESCAPE und Merlin soll an dieser Stelle ausführlicher als auf die anderen Sprachen/PSEU eingegangen werden, da im folgenden Kapitel ein möglicher Lösungsvorschlag für die Probleme aus Kapitel 4 im Umfeld von ESCAPE und Merlin betrachtet wird.

Im folgenden wird die Darstellung von ESCAPE anhand der drei in ESCAPE enthaltenen Teilmodelle vorgenommen. Bei Merlin wird die aktuelle Architektur erläutert sowie die Benutzungsschnittstelle beschrieben. Abschließend werden dann in diesem Kapitel die Schwächen von ESCAPE bzw. Merlin zusammengefaßt.

5.2.5.1 ESCAPE

ESCAPE (EER-Models and Statecharts Combined for an Advanced Process Engineering) ist der Name einer graphischen PML, die auf der Object Modeling Technique (OMT, vgl. [RBP+91]) aufsetzt. Das bedeutet, daß bei der Erstellung von ESCAPE die Sprachmittel, die OMT bietet, im einzelnen daraufhin überprüft wurden, inwieweit sie in einer speziell zur Beschreibung von Prozeßmodellen gedachten Sprache Sinn machen, überflüssig sind oder umgekehrt nicht ausreichen. Im Gegensatz zu OMT werden die einzelnen Konzepte formal beschrieben, so daß die Bedeutung der einzelnen Sprachelemente definiert ist.

Den Ansatz einer allgemein gehaltenen Beschreibungssprache zur Modellierung objektorientierter Systeme auf die Modellierung von Prozeßmodellen anzuwenden, macht deswegen Sinn, weil Prozeßmodellierung als eine Art der Programmierung betrachtet werden kann (vgl. [Ost87]). Durch die Verwendung von OMT als Basis greift ESCAPE diese Idee auf und bindet dadurch gleichzeitig die entstehenden Prozeßbeschreibungen an das objektorientierte Paradigma.

Bei der folgenden Betrachtung fällt auf, daß ESCAPE die Aufteilung in einzelne Modelle beibehalten hat. Jedes für sich genommen beschreibt einen bestimmten Aspekt der Prozeßmodellierung (Prinzip des „Separation of Concerns“, vgl. Abbildung 5.6). Das *Objektmodell* und das *Koordinationsmodell* wurden direkt übernommen. Das dritte Teilmodell der OMT, das Datenflußmodell, ist eliminiert worden, da in ESCAPE keine Datenflüsse im Prozeß modelliert werden. Stattdessen ist ein neues Modell, das sogenannte *Organisationsmodell* hinzugekommen, das es erlaubt, die notwendigen Zuordnungen von Aktivitäten zu den Rollen, die sie ausführen, zu spezifizieren.

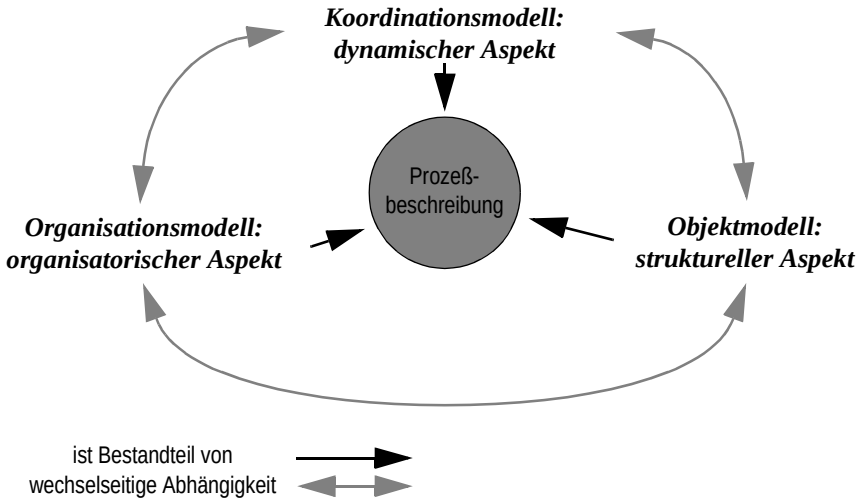


Abbildung 5.6 Die Teilmodelle von ESCAPE (vgl. [Jun95], S. 24)

In den folgenden Kapiteln werden die Modelle jeweils kurz erläutert. Die formale Beschreibung wird nicht aufgegriffen. Für diese wird auf die entsprechende Arbeit von Junkermann, [Jun95], Kapitel 6 und 7 verwiesen.

Objektmodell

Den zentralen Bestandteil einer Prozeßspezifikation mit ESCAPE bildet das *Objektmodell*. Hier geht ESCAPE den gleichen Weg wie die Autoren der OMT, die begründen, daß zunächst beschrieben werden muß, welche Objekte bzw. Klassen es gibt, bevor deren Dynamik beschrieben wird. Auf den Kontext einer PML übertragen bedeutet dies, daß zunächst klar sein muß, welche Elemente in einem konkreten Prozeßmodell enthalten sind, bevor darüber nachgedacht werden kann, diese zu koordinieren und zu steuern. Aus der Sicht von ESCAPE handelt es sich bei diesen Elementen um die Dokumenttypen, die im Zuge der Erstellung von Software nach dem mit ESCAPE zu spezifizierenden Entwicklungsprozeß erstellt werden sollen.

Wie OMT benutzt ESCAPE als Mittel der Darstellung im Objektmodell erweiterte Entity-Relationship-Modelle (EER-Modelle). Dieser Notationstyp entstammt ursprünglich dem Bereich der Datenbankmodellierung und hatte das Ziel, Daten einer Datenbank und ihre Zusammenhänge darzustellen. Diesen Grundgedanken haben OMT und andere Sprachen aufgegriffen und haben die Konzepte aus der ER-Modellierung auf ihren Bedarf angepaßt. Bei OMT und damit auch bei ESCAPE umfaßt diese Anpassung zwei verschiedene Bereiche. Der erste ist die Erweiterung der ursprünglich recht knapp gehaltenen sprachlichen Möglichkeiten der originalen ER-Notation (vgl. [Che76]). Zu diesem Zweck werden zusätzliche Sprachmittel hinzugefügt, die dann weitergehende Zusammenhänge beleuchten. Noch wichtiger ist aber, daß die ursprünglich ausschließlich Daten beschreibende Notation für das objektorientierte Paradigma erweitert wird. Das bedeutet konkret, daß den Daten Methodenbeschreibungen hinzugefügt werden, die den Zugriff auf die Daten beschreiben und somit objektorientierte Klassen darstellen. Trotzdem bleibt die Beschreibung der Zusammenhänge auf der Ebene des Objektmodells zunächst eine rein datenbezogene.

In Abbildung 5.7 ist in einem Beispiel dargestellt, wie die durch das Objektmodell repräsentierten strukturellen Aspekte modelliert werden. Das Beispielmodell ist im Sinne der ESCAPE Syntax nicht vollständig. Die Blätter des durch die Vererbungsbeziehung aufgespannten Baums - ESCAPE läßt nur Einfachvererbung zu - spezifizieren die durch den modellierten Software-Entwicklungsprozeß zu unterstützenden Dokumenttypen. Diese entsprechen also Klassen einer herkömmlichen OMT-Hierarchie.

Wie im Beispiel zu sehen ist, werden alle Dokumenttypen durch Spezialisierung aus einer (in ESCAPE invariant festgeschriebenen) Wurzelklasse abgeleitet. Vererbt werden dabei Attribute, Methoden und Assoziationen der Superklassen. Die Methoden heißen in ESCAPE Aktivitäten, um dem Anwendungsbereich Prozeßmodellierung sprachlich näher zu sein. In der Terminologie von ESCAPE sind alle Nicht-Blattklassen sogenannte abstrakte Dokumenttypen, alle Blattklassen konkrete Dokumenttypen. Die Bedeutung von abstrakt ist die im objektorientierten Sinne übliche, d.h. daß solche Klassen zur Laufzeit nicht instanziiert werden, sondern zur Schnittstellenspezifikation unter Ausnutzung von Vererbung benutzt werden. Durch diese Begriffswahl wird ausgedrückt, daß die Spezialisierung als reines Strukturierungsmittel verstanden wird und tatsächlich im realen, durchzuführenden Prozeß nur Dokumente der Blattklassen existent sind.

Im Beispiel werden mehrere konkrete Klassen spezifiziert (Pflichtenheft, Architektur, Quelltextdokument-Schnittstelle, Quelltextdokument-Implementierung, Objektdatei, Programm), die sich, z.T. über Zwischenstufen, aus der abstrakten Wurzelklasse ablei-

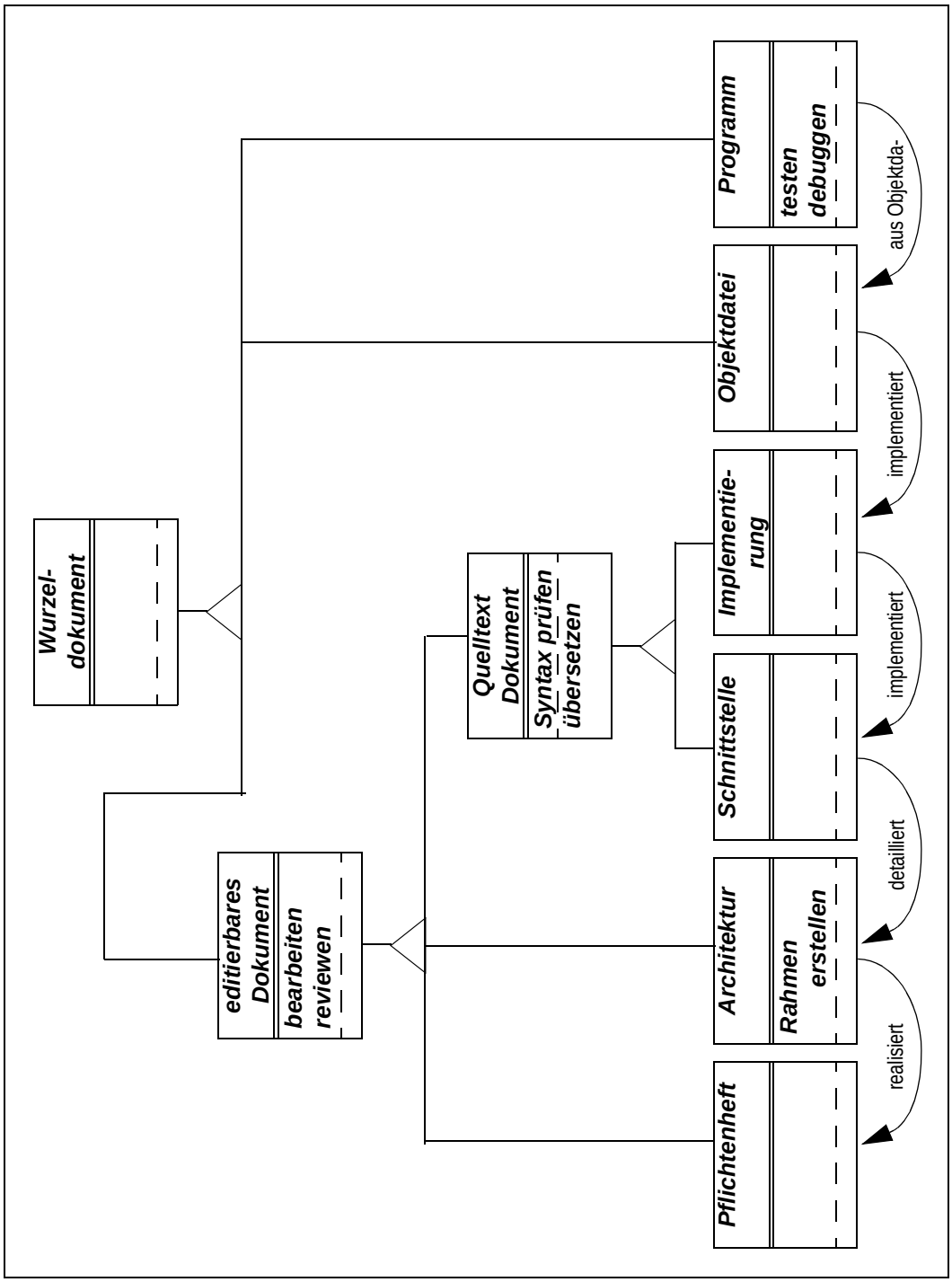


Abbildung 5.7 Beispielausschnitt aus einem ESCAPE-Objektmodell

ten. Die editierbaren Dokumente erben die Aktivitäten, die auf ihnen auszuführen sind, von der Klasse `editierbares Dokument`. Somit gilt beispielsweise, daß auch auf einem `Architektur`-Dokument die Aktivitäten `bearbeiten` und `reviewen` ausgeführt werden können. Die Assoziationen deuten an, daß es Beziehungen zwischen den einzelnen Klassen gibt. So wird durch die Assoziation `realisiert` zwischen den Klassen `Architektur` und `Pflichtenheft` ausgedrückt, daß sich ein Architektur-Dokument auf ein Pflichtenheft bezieht. Dies ist für die Beschreibung der Dynamik von Bedeutung, da z.B. das Architekturdokument erneut überarbeitet werden sollte, falls sich das Pflichtenheft ändert. Den ESCAPE-Assoziationen kann dabei

nicht entnommen werden, wie oft ein Dokument eine Assoziation eingehen kann oder muß. Dem Modell kann nicht entnommen werden, ob z.B. ein Schnittstellendokument nur durch genau ein Implementierungsdokument codiert wird oder ob es auch mehrere davon zu einer Schnittstelle geben darf, die beispielsweise jeweils nur Teile der Schnittstelle realisieren.

Bei den Aktivitäten ist noch eine Unterscheidung wichtig, die dem Modell entnommen werden kann und später im Koordinationsmodell benutzt wird. Im Beispielausschnitt aus einem Objektmodell, der in Abbildung 5.7 dargestellt ist, ist die Liste der Aktivitäten eines Dokumenttypen durch eine gestrichelte Linie in zwei Teile getrennt. Die im oberen Teil genannten Aktivitäten sind die nach ESCAPE Nomenklatur sogenannten *interaktiven Aktivitäten*. Die Aktivitäten im unteren Teil werden als *automatische Aktivitäten* bezeichnet. Interaktive Aktivitäten sind dabei solche, die explizit von einem Benutzer eingeleitet werden müssen. Für diese muß im Organisationsmodell demnach eine verantwortliche Rolle eingetragen sein. Umgekehrt sind automatische Aktivitäten solche, die vollständig ohne Benutzerinteraktion auskommen. Der Ausführungszeitpunkt einer automatischen Aktivität läßt sich aus dem Projektzustand, insbesondere dem Zustand des betroffenen Dokuments direkt ableiten. Diese Aktivitäten können also vollständig automatisch in den Prozeßablauf eingebaut werden, wodurch sich eine Reduzierung des Aufwands für die Entwicklung ergibt. Natürlich muß in der Realisierung solcher Aktivitäten sichergestellt sein, daß Aktivitäten, die ein Ergebnis produzieren, dieses geeignet an die Prozeßmaschine zur weiteren Verarbeitung zurückgeben können.

Die Menge der im Objektmodell spezifizierten Dokumenttypen und Aktivitäten gibt nun den Rahmen vor, auf dem Dynamik in das Prozeßmodell gebracht wird. Sie bilden die Basis für das Koordinationsmodell, das im folgenden Kapitel dargestellt wird.

Koordinationsmodell

Zur Modellierung der dynamischen Aspekte des zu spezifizierenden Software-Entwicklungsprozesses bietet das Objektmodell keine Mittel. Die notwendige Ergänzung der Beschreibung wird in einem eigenen Modell, dem *Koordinationsmodell* eingebracht.

Im Koordinationsmodell gibt es für jeden konkreten Dokumenttyp, der im Objektmodell definiert wurde, ein Zustandsmodell. Die Darstellung der Zustandsmodelle basiert auf den StateCharts von Harel. Die Wahl der Harel´schen Notation zur Darstellung der Dynamik trägt dem Rechnung, daß Prozeßmodelle typischerweise keine rein sequentielle Abfolge von Aktivitäten beschreiben, sondern auf interne und externe Abhängigkeiten reagieren. Software-Entwicklungsprozesse sind demnach ein Spezialfall *reaktiver Systeme*, die sich genau durch das Fehlen vorherbestimmbarer, sequentieller Abfolgen und die Reaktion auf Ereignisse charakterisieren.

StateCharts sind im wesentlichen Zustandsübergangsdiagramme, die allerdings um Strukturierungskonzepte, um Parallelität und um Kommunikation zwischen den verschiedenen parallelen Zustandsfolgen erweitert worden sind. Sie sind allerdings ursprünglich zur Beschreibung konkreter Systeme, d.h. einzelner Instanzen von Objekten gedacht gewesen. Da Prozeßmodelle Prozesse auf der Typebene beschreiben und vom konkreten Projekt und damit auch von konkreten Dokumentinstanzen unabhängig

sind, sind die Sprachmittel in ESCAPE angepaßt worden. Zu den genauen Änderungen und ihren Konsequenzen wird auf die entsprechende Arbeit von Junkermann ([Jun95]) verwiesen.

Das Koordinationsmodell besteht aus genau einem AND-Zustand. Ein solcher Zustand besteht wiederum aus mehreren parallelen Zustandsfolgen, die zusammengenommen den Gesamtzustand des Systems beschreiben. Die einzelnen, parallelen Anteile des AND-Zustands sind die Zustandsmodelle, die das Verhalten der jeweiligen Dokumenttypen beschreiben. In Abbildung 5.8 ist ein Beispielausschnitt aus einem Koordinationsmodell abgebildet.

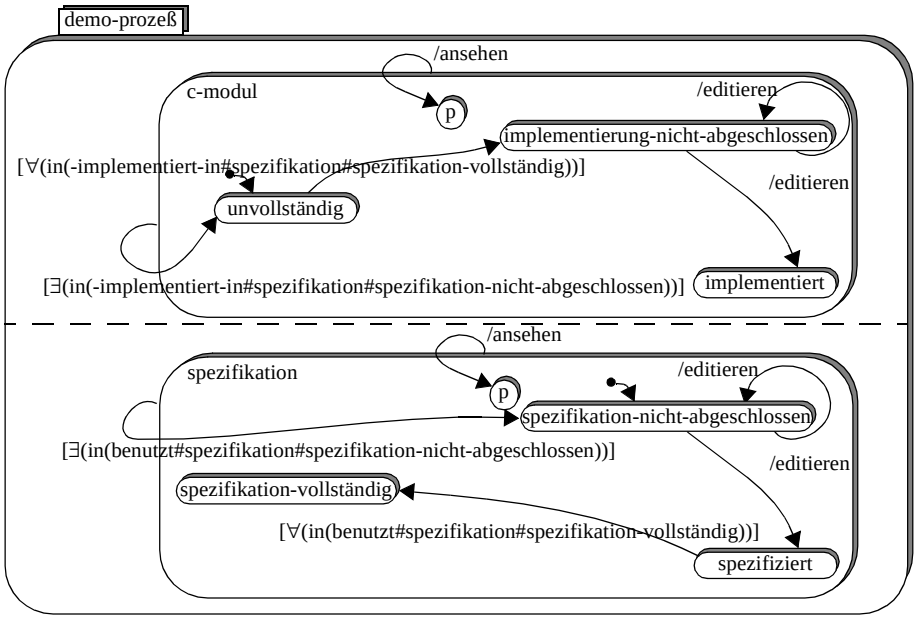


Abbildung 5.8 Beispielausschnitt aus einem ESCAPE-Koordinationsmodell

Im Beispiel sind mögliche Zustandsabfolgen für die zwei Dokumenttypen c-modul und spezifikation dargestellt. Die gestrichelte Linie unterteilt die beiden Teilzustände des AND-Zustands, die die Dokumenttypen beschreiben. Die Knoten entsprechen den Zuständen, die sie verbindenden Kanten den Übergängen. Die Zustände werden je nach gewünschter Prozeßabfolge definiert. Hier ist also der Punkt, an dem festgelegt werden kann, wie ein Dokument im definierten Prozeß eingebunden ist. Den letzten Zusammenhalt und die Reihenfolgen in der Dokumentbearbeitung ergeben sich allerdings erst durch die definierten Transitionen.

Die Transitionen zwischen den einzelnen Zuständen des Zustandsmodells eines Dokumenttyps stellen die oben angesprochenen internen oder externen Abhängigkeiten dar. Diese werden auch als Ereignisse bezeichnet. Ereignisse können entweder die Durchführung einer auf dem Dokument definierten Aktivität (aus dem Objektmodell) oder ein internes Ereignis aus der Ablaufkontrolle der Prozeßmaschine sein. Letztere können dann auftreten, wenn eine Abhängigkeit zwischen zwei Dokumenttypen definiert wird, die durch eine im Objektmodell definierte Assoziation verbunden sind. Diese Abhängigkeiten referenzieren die Instanzebene und können so aussehen, daß mindestens ein oder alle durch den Assoziationstyp mit dem Ausgangsdokument in Verbindung stehenden Dokumente in einem bestimmten Zustand sind. Um welchen Typ von Transistion es sich handelt, wird im Diagramm durch das daran notierte Label ausge-

drückt, das entweder einen Aktivitätsnamen referenziert oder aber die zu überprüfende Beziehung und den Überprüfungstyp festlegt.

Mit den Informationen, die im Objekt- und Koordinationsmodell festgelegt sind, ist der Software-Entwicklungsprozeß schon recht gut beschrieben. Was nun noch fehlt, ist die Einbindung in die betrieblichen Abläufe, so daß auch eine Personenzuordnung erfolgen kann.

Organisationsmodell

Der Aspekt der Anbindung an Personen, die dann die Softwareentwicklung tatsächlich durchführen, wird durch das Organisationsmodell geleistet. Dabei ist die Formulierung „Anbindung an Personen“ genaugenommen falsch, da auch das Organisationsmodell auf der Prozeßmodellebene, d.h. auf der Typebene bleibt und keine konkreten Personen benennt.

Die Typebene wird in diesem Kontext durch die Angabe von Rollen realisiert. Die Durchführung einer Aktivität wird an eine Rolle gebunden, so daß die durchführende Person im Organisationsmodell offen bleibt. Beispiele für Rollen sind z.B. Analytiker, Entwickler etc.

Im Koordinationsmodell wird beispielsweise eingetragen, daß ein Quelltext-Interface durch ein Review gehen muß, bevor es in den Zustand `abgeschlossen` kommen kann. Im Organisationsmodell kann die Aktivität `review` nun der Rolle `Entwickler` zugeordnet werden, so daß die Durchführung der Aktivität nur noch Personen aufgetragen werden kann, die im Projekt die Rolle `Entwickler` einnehmen. Umgekehrt formuliert kann über diesen Mechanismus sichergestellt werden, daß ein Minimum an Zugriffsschutz gewährleistet ist, da eine Aktivität nur noch solchen Rolleninhabern angeboten wird, die dafür vorgesehen sind. Auf diese Weise wird sichergestellt, daß z.B. das Review der Architektur von den davon betroffenen Entwicklern vorgenommen wird und nicht von einer beliebigen Person.

5.2.5.2 Merlin

Merlin bildet den Kern einer Entwicklungsumgebung, die auf einer Modellausführungskomponente basiert. Zur Erstellung der Modelle ist Merlin ein eigenständiger Editor vorgeschaltet, der die Eingabe von ESCAPE-Modellen ermöglicht und diese - soweit möglich - auf syntaktische und statisch semantische Fehler überprüft. Ist ein Modell nach Meinung des Modellierers vollständig, so kann er es in eine ausführbare Variante übersetzen, die dann von Merlin benutzt werden kann. In einem relativ unkomfortablen Verfahren wird dann die ausführbare Version des Modells durch initiale Informationen ergänzt, so daß diese ergänzte Version in Merlin (also einem konkreten Projekt) eingesetzt werden kann. Zu diesen Informationen gehören die Spezifikation von Informationen des Organisationsmodells und die initiale Definition von Dokumentinstanzen.

Merlin umfaßt im engeren Sinne eine Prozeßmaschine und zusätzlich mehrere Front-End-Komponenten, die den Zugang zum aktuellen Prozeßzustand für die jeweiligen Prozeßbeteiligten erlauben. Zu diesen Komponenten kommen noch die im einzelnen eingesetzten Werkzeuge der Ausführungsebene hinzu. Diese Anbindung wird bisher

manuell durchgeführt. An genau dieser Stelle sollen daher die in Kapitel 6 beschriebenen Mechanismen ansetzen, um eine angemessene Anbindung von Ausführungsebene und Modellausführung zu gewährleisten. Zunächst soll aber die existierende Merlinversion beschrieben werden.

Benutzersicht

Der Benutzer von Merlin, typischerweise ein mit Aufgaben in einer laufenden Softwareentwicklung betrauter Entwickler, sieht Merlin als ein Paket von drei Front-En-Komponenten, die den Zugriff auf die die Modellausführung steuernde Prozeßmaschine erlauben. Diese Komponenten sind über eine Nachrichtenschnittstelle, die ein eigens für diesen Zweck definiertes Protokoll realisiert, miteinander verbunden (vgl. Abschnitt *Architektur* in Kapitel 5.2.5.2). Im Übersichtsbild stellt sich das Zusammenspiel wie in Abbildung 5.9 dar.

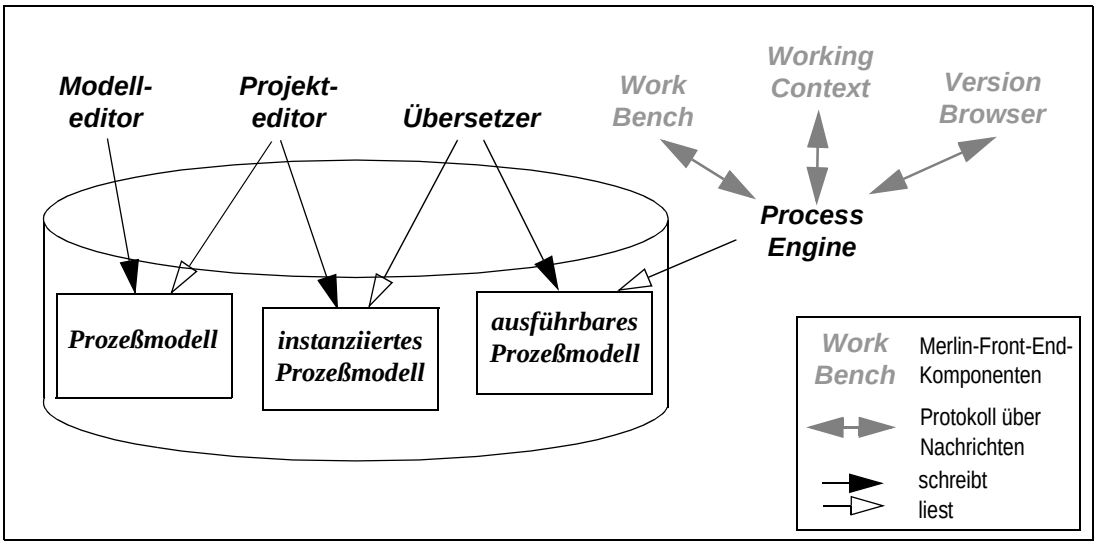


Abbildung 5.9 Übersicht über die Merlin-Komponenten und ihr Zusammenspiel

Die sogenannte *Work Bench* ist für einen Benutzer der Umgebung der Einstieg in die Arbeit. Da die von der Merlin-Prozeßmaschine ausgeführten ESCAPE-Modelle über die Instanziierung des Organisationsmodells eine Bindung von Aktivitäten und Zugriffsrechten an Personen in bestimmten Rollen haben, muß sich jeder Benutzer der Umgebung hier mit seinem Namen und einem Paßwort einloggen. Je Benutzer ist zur Laufzeit entsprechend eine Work Bench aktiv. Hat der Benutzer sich eingeloggt, so kann er von der Work Bench aus die Prozeßmaschine anweisen, weitere Front-End-Komponenten (im Moment also Working Context oder Version Browser) zu öffnen. Diese sind dann projekt- und rollenspezifisch, so daß der Benutzer zu ihrem Start aus der Liste der Projekte, an denen er beteiligt ist, und nach Auswahl eines Projekts aus der Liste der Rollen, die er in diesem Projekt erfüllt, auswählen muß, welche Tätigkeiten er durchführen möchte. In Abbildung 5.10 ist die Work Bench in ihrer derzeitigen Realisierung dargestellt.

Wenn der Benutzer diese Informationen eingegeben hat, kann er den *Working Context* öffnen. Dieser ist in Merlin die Schnittstelle zur Durchführung einzelner Aktivitäten auf Dokumenten eines Projekts. In Abbildung 5.11 ist ein Beispiel dargestellt, wie der Working Context zur Laufzeit für einen konkreten Benutzer aussieht.

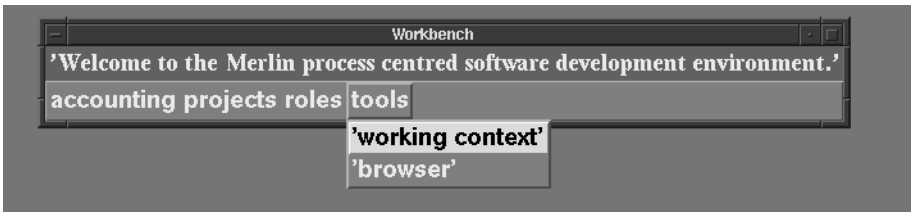


Abbildung 5.10 Work Bench in Merlin

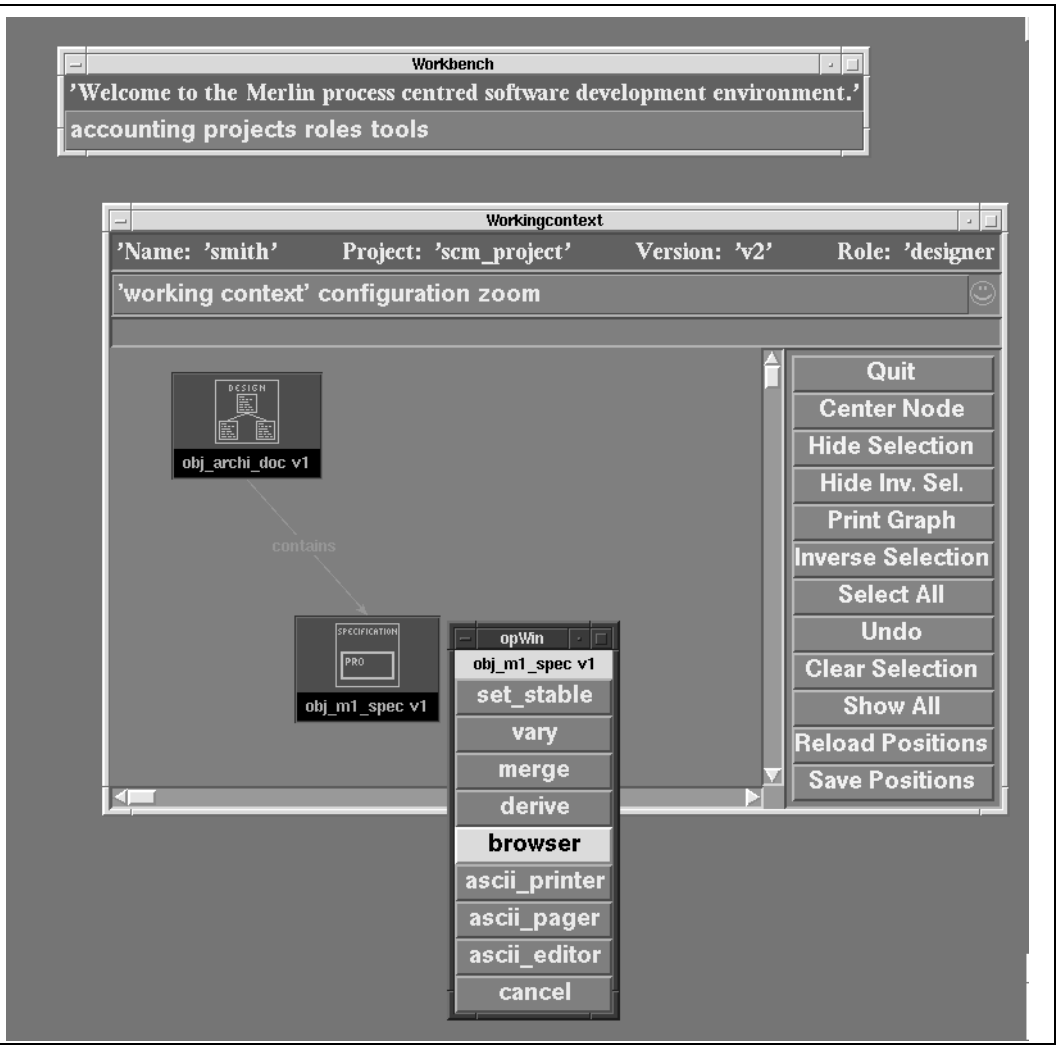


Abbildung 5.11 Working Context in Merlin

Neben der vom Benutzer unabhängigen Werkzeugschnittstelle sind dort graphisch die Dokumente dargestellt, die der Benutzer in seiner momentanen Rolle zur Bearbeitung anstehen hat bzw. die er zu ihrer Bearbeitung benötigt. Weiterhin werden die Abhängigkeiten (Instanzen von ESCAPE-Assoziationen) zwischen den Dokumenten aufgezeigt. Der Benutzer kann nun aus dieser Menge von Dokumenten eines herausgreifen und wiederum aus der Menge der darauf angebotenen Aktivitäten eine zur Ausführung auswählen. Die angebotenen Aktivitäten sind eine auf die Rolle des Entwicklers bezogene Teilmenge aller auf dem Dokument(typ) definierten Aktivitäten. Bei der Instanziierung des Prozeßmodells wird für jede Aktivität eine Repräsentation durch ein

Werkzeug angegeben. Entsprechend wird nun das jeweilige Werkzeug nach Auswahl der Aktivität gestartet.

Der Benutzer kann dann das Dokument in der gewünschten Form bearbeiten, wobei es in dieser Zeit keine Rückmeldung über den Inhalt seiner Tätigkeit gibt. Als Abschluß einer solchen Aktivität muß der Benutzer den neuen Status des von ihm bearbeiteten Dokuments setzen. Damit wird die Anbindung an das Koordinationsmodell des zugrunde liegenden ESCAPE-Modells geschaffen, dessen mögliche Folgestati in dieser Auswahl angeboten werden.

Als Folge einer solchen manuell durchgeführten „Weiterschaltung“ des Zustandsmodells kann es zu weiteren Werkzeugaufrufen kommen, falls für den neuen Zustand die Ausführung einer automatischen Aktivität definiert worden ist. Es kann auch zu weiteren Zustandsänderungen innerhalb des Projekts kommen, falls ein anderes Dokument von dem gerade geänderten durch eine Beziehung abhängt, die im Koordinationsmodell definiert worden ist.

In der aktuellen Fassung von Merlin, die allerdings schon auf einer Überarbeitung von ESCAPE, *ESCAPE+* (vgl. [NSS96]) aufsetzt, wird die bereits erwähnte dritte Front-End-Komponente, der *Version Browser*, angeboten. Die Implementierung der Process Engine, die in der Lage ist, die neueren ESCAPE+-Modelle auszuführen, kennt zusätzlich zu den Dokumenten auch noch Versionen von Dokumenten. Die Hierarchie von vorgegebenen Klassen ist dabei vom einfachen Wurzeldokument auf weitere Klassen verbreitert worden, wodurch eine sinnvolle Versionierung je nach Art des Dokumenttyps erlaubt wird. Der Version Browser erlaubt es nun, sich die Versionshistorie eines Dokuments anzusehen und daraus die aktuell für die Entwicklung zu verwendende Version zu selektieren. Diese Funktionalität hat natürlich nicht nur für Einzelversionen eine Auswirkung, sondern auch für ganze Konfigurationen, da jede Version ihre Assoziationen zu bestimmten Versionen anderer Dokumente aufbaut. Somit muß beachtet werden, wie denn die gewählte Version eines Dokuments mit den anderen gewählten Versionen der anderen Dokumente zusammenpaßt.

Architektur

Nachdem im vorangegangenen Kapitel gezeigt wurde, wie Merlin für den Benutzer aussieht, soll nun die zugrundeliegende Architektur dargestellt werden. Die in Abbildung 5.9 gezeigte Übersicht über die Komponenten von Merlin und deren Zusammenspiel ist nur eine sehr grobe Skizze gewesen. In diesem Kapitel soll die Architektur detaillierter betrachtet werden.

Die in Abbildung 5.9 dargestellten Komponenten bilden gewissermaßen den Kern von Merlin, der unabhängig von Prozeßmodellen, Projektinstanziierungen und gewählten Werkzeugen ist. Allerdings ist das Zusammenspiel der Komponenten nur auf der Ebene der Kommunikationsbeziehungen so, wie es in der Abbildung dargestellt wurde. Auf der Architekturebene existiert als weitere Komponente der Nachrichten-Server, der für die Verteilung der Nachrichten, die letztlich das eigentliche Protokoll bilden, verantwortlich ist. Ebenso kommt eine Datenbank hinzu, die es der Prozeßmaschine erlaubt, den Projektzustand persistent zu halten. Die Transaktionsmechanismen der Datenbank werden dazu genutzt, die Arbeit verschiedener Benutzer miteinander zu koordinieren. Dazu wird je Entwickler eine PE geöffnet, die ihre Information über den

aktuellen Stand des Projekts aus einer zentralen Datenbank bezieht und Änderungen dort einträgt. Nimmt eine PE-Instanz Änderungen vor, die auch andere Entwickler betreffen könnten, so informiert sie alle anderen aktiven PE-Instanzen darüber, so daß diese ihre Informationen neu einlesen können. Damit ergibt sich die in Abbildung 5.12 dargestellte Architektur.

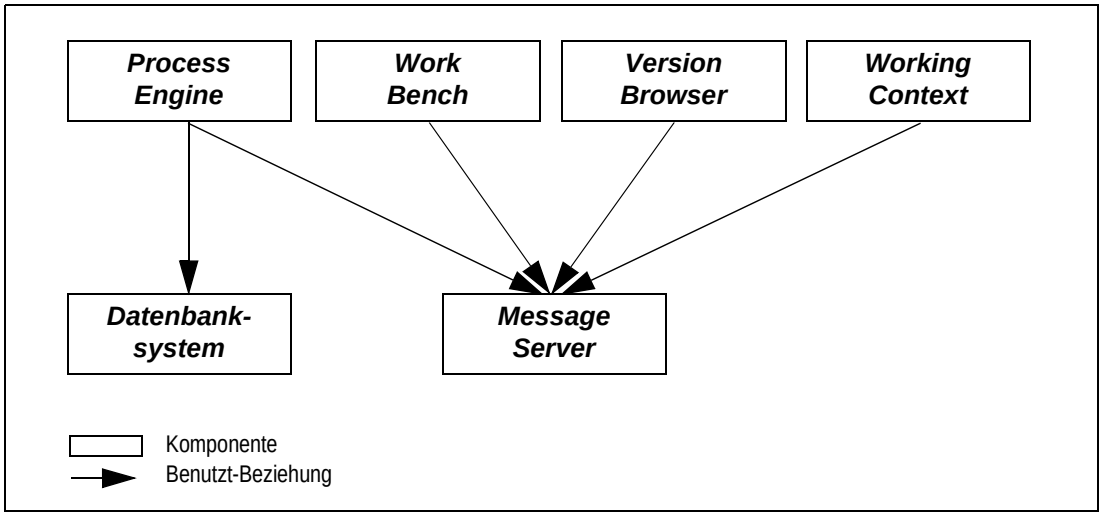


Abbildung 5.12 Darstellung der Architektur um die Kernkomponenten von Merlin

In diese Architektur müssen nun noch die Werkzeuge und die von ihnen bearbeiteten Dokumente eingepaßt werden. In Merlin werden alle automatischen Werkzeuge von der Prozeßmaschine aufgerufen. Für die interaktiven Werkzeuge gilt, daß sie vom Working Context aus aufgerufen werden, d.h. vom Benutzer. Der Working Context fragt wiederum vor der Ausführung des Werkzeugs bei der Prozeßmaschine an, um sich von dort die Information geben zu lassen, welches Werkzeug als Instanziierung einer gewählten Aktivität ausgeführt werden soll. Die durch die Werkzeuge ergänzte Architektur ist in Abbildung 5.13 dargestellt.

Die Abbildung der Aktivitäten auf die Werkzeuge wird im Rahmen der Projektinstanziierung auf einer rudimentären Basis und für jede Aktivität und jedes Werkzeug einzeln manuell durchgeführt. Diese wird sehr spezifisch auf die Plattform zugeschnitten, auf der Merlin im Moment implementiert ist. Faktisch werden Kommandozeilen für Aufrufe programmiert, die dann als Aktivität aufgerufen werden. In diesen Kommandozeilen können einzelne Parameter als Platzhalter übergeben werden, für die hart verdrahtet ist, wie sie implementiert werden müssen. Da die Dokumente entsprechend von jedem Werkzeug einzeln verwaltet werden, gibt es keine Kontrollmöglichkeiten, die über den Inhalt der durchgeführten Aktivitäten Auskunft geben können. Die einzigen Feedbackmöglichkeiten, die vorgesehen sind, nutzen die Unix-spezifische Art der Rückgabe eines ganzzahligen Ergebniscodes bzw. die im Zusammenhang der Benutzungsschnittstelle erwähnte Abfrage eines Folgestatus für das Dokument. Damit könnten selbst intelligentere Werkzeuge nur sehr eingeschränkt zur Projektsteuerung auf der Basis der Modellausführung eingesetzt werden.

In einem Projekt an der Universität Dortmund (Groupie, vgl. [ES94]) wurde versucht, ein solches Feedback zu ermöglichen. Die Durchführung sah so aus, daß die Änderungen vom Werkzeug in einem der PE und dem Werkzeug vorher bekannten Format in einer Datei abgespeichert wurden. Diese Datei wurde nach der Beendigung dieses spe-

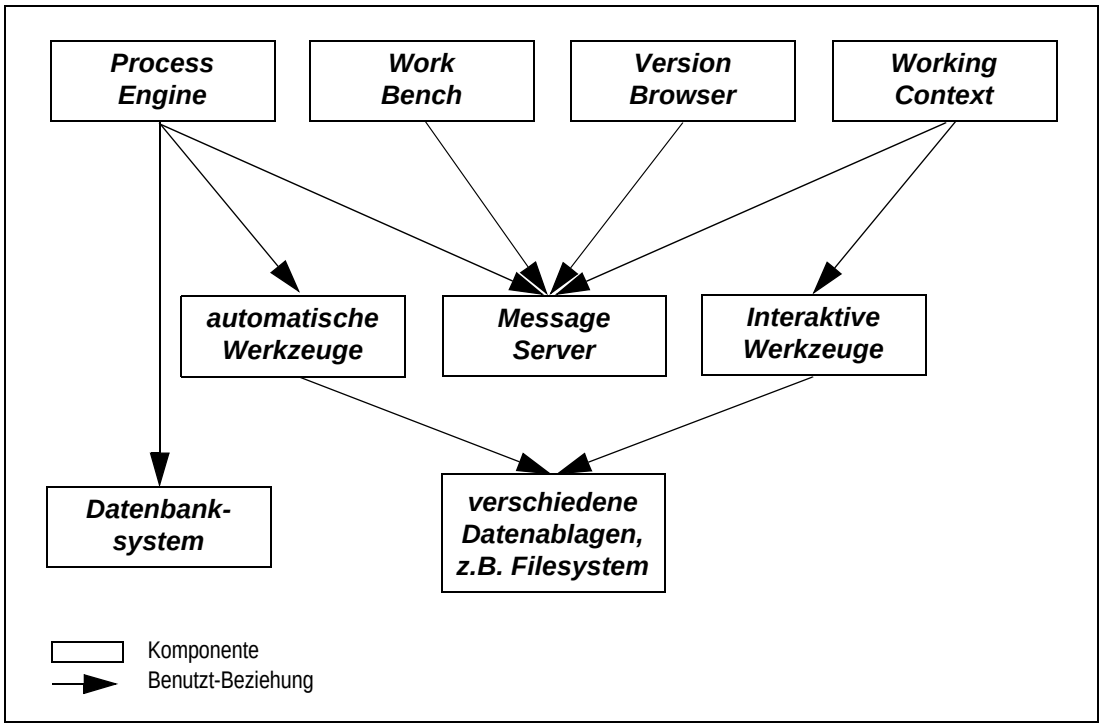


Abbildung 5.13 Skizze einer auf Merlin basierenden Software-Entwicklungsumgebung

ziellen Werkzeugs von der PE gelesen, und die darin verzeichneten Änderungen wurden durchgeführt.

Fehlerträchtig ist der Weg, wie Dokumente zur Laufzeit erzeugt werden. Es wird für jede Aktivität, die weitere Dokumente erstellt, verlangt, daß sie die Anzahl der erzeugten Dokumente festlegt. Mit dieser Einschränkung kann dann statisch vorherbestimmt werden, welche Dokumente im Verlaufe des Projekts automatisch erzeugt werden. Auf dieser Kenntnis baut auch die Vergabe von physikalischen Namen auf. Zur Erzeugung mehrerer Dokumente werden z.T. Makefiles verwendet, in denen die Dokumentreferenzen hart verdrahtet werden. Dadurch entstehen unnötige und fehlerträchtige Abhängigkeiten.

Kann man sich hierbei noch auf den Standpunkt stellen, daß solche Abhängigkeiten letztlich vermeidbar wären, wenn stattdessen ein wenig trickreicher mit den Makefiles umgegangen würde, so bleibt in jedem Falle das Problem, daß keine Aktivitäten spezifiziert werden können, die eine vorher unbekannte Anzahl von Dokumenten erzeugen. Diese dynamische Anzahl von Dokumenten tritt aber immer dann auf, wenn ein Werkzeug dazu eingesetzt wird, den weiteren Projektverlauf zu planen, d.h. wenn nicht der gesamte Projektverlauf bei Beginn des Projekts bekannt ist und wenn durch die eingesetzten Werkzeuge weitere Dokumente angelegt werden sollen.

5.2.5.3 Problemlösungen in ESCAPE und Merlin

Generell gilt auch für ESCAPE/Merlin, daß keine Werkzeugspezifkation benutzt wird, um die Anbindung von Werkzeugen vorzunehmen bzw. die Werkzeuge zu beschreiben. Die dadurch verursachten Einschränkungen bezüglich der Anforderungen aus Kapitel 4, die in den vorangegangenen Kapiteln herausgearbeitet worden sind, gelten hier demnach analog.

Für die Signatur der Aktivitäten (vgl. Kapitel 4.1) gilt, daß sie auf der Ebene von ESCAPE-Modellen nur teilweise beschrieben wird. Es wird bei einer Aktivität nicht formuliert, welche Dokumente, außer dem, auf dem die Aktivität aufgerufen wurde, benutzt werden. Dies ist eine gute Abstraktionsebene für die Formulierung von Dokumenten, macht aber die Anbindung von Werkzeugen schwierig. Die Anforderung zur *Spezifikation der Signatur* ist demnach auch auf der Modellseite nur bedingt erfüllt.

Die Anforderung zur *Spezifikation des Leistungsumfangs* fordert von der PML, daß sie den Leistungsumfang der Aktivitäten spezifiziert. ESCAPE kann den Leistungsumfang nicht weitergehend spezifizieren, als es die Signatur zuläßt, da ein Teil dieser Spezifikation mit der Signatur zusammenhängt. Spezifziert wird allerdings, ob es sich bei einer Aktivität um eine automatisch ablaufende Aktivität handelt oder ob die Aktivität interaktiv mit einem Entwickler abläuft. Die Anforderung zur *Spezifikation des Leistungsumfangs* ist daher in dem Teil, der die PML betrifft, nur bedingt erfüllt. Demnach kann Merlin auch keine Unterstützung zur Vermeidung versehentlicher Fehler aus einem unangepaßten Werkzeug bieten, d.h. die Anforderung zur *Kontrolle des Leistungsumfangs* kann durch eine solche Maßnahme nicht erfüllt werden. Allerdings basiert der Working Context auf einem Agenda-Prinzip, das es erlaubt, die ungültigen Aktivitäten vor dem Aufruf auszublenden. Dadurch ist eine einfache Kontrolle schon vor dem Aufruf der Werkzeuge möglich.

Bezüglich der Anforderung zur *Implementierung modellierter Aggregationen* ist festzustellen, daß Merlin keine Aggregationen zur Strukturierung von Dokumenten kennt. Dadurch kann es natürlich nicht zu den in Abschnitt *Aggregationen* (in Kapitel 4.1.2) beschriebenen Problemen kommen. Im Zusammenhang mit Marvel ist allerdings schon auf die Notwendigkeit eines Aggregationskonzepts hingewiesen worden. In diesem Sinne ist die Anforderung zur *Implementierung modellierter Aggregationen* so zu interpretieren, daß ein Aggregationskonzept in der benutzten PML vorhanden sein soll, was bei ESCAPE nicht der Fall ist.

Bezüglich des Zustandsmodells ist in ESCAPE eine detaillierte Spezifikation durch das Koordinationsmodell gegeben. Die Anforderung zur *Spezifikation des Zustandsmodells* ist damit vom Prozeßmodell abgedeckt. Die Anforderung zur *Abbildung des Zustandsmodells* ist bedingt erfüllt, da es durch geschickte Programmierung der Instanziierung des Prozeßmodells möglich ist, die geforderte Zustandsabbildung vorzunehmen. Eine Spezifikation der Abbildung ist aber nicht vorgesehen. Die Anforderung zur *Übermittlung von Zustandsinformation*, d.h. daß die Prozeßsteuerung auf jeden Fall über den Zustand der einzelnen Dokumente nach der Ausführung eines Werkzeugs informiert werden muß, ist erfüllt. Im Falle interaktiver Werkzeuge, deren Ergebnis sich häufig nicht automatisch bestimmen läßt, wird dazu automatisch eine Abfrage durchgeführt.

Feedback aus den Werkzeugen heraus ist in Merlin zunächst nicht vorgesehen. Dies liegt daran, daß von Prozessen ausgegangen wurde, deren Dokumente sich aufgrund der Spezifikationen vordefinieren lassen, womit eine dynamische Veränderung an Bedeutung verliert. Diese Annahme gilt nicht für den allgemeinen Fall, so daß in diesem Sinn die Anforderung zur *Übermittlung von Feedback* nicht erfüllt ist. Als direkte Konsequenz kann auch die Anforderung zur *Auswertung von Feedback* nicht abgedeckt werden.

Die Angabe von Parametern für den Aufruf eines Werkzeugs, die nicht aus dem Prozeßmodell abgeleitet werden, ist problemlos möglich. Ergänzende Parameter werden direkt mit in den Aufrufen der Werkzeuge aus den Envelopes angegeben. Die Anforderung zur *Spezifikation der prozeßunabhängigen Eingaben* ist daher abgedeckt.

Im Bereich der Laufzeitfehler (Anforderungen zur *Zustandsbestimmung nach Werkzeugabbruch* und zur *Zustandsbestimmung nach PE-Abbruch*) und deren Erkennung zur Laufzeit (Anforderung zur *Erkennung von Werkzeugabbrüchen*) bietet Merlin keine Konzepte an.

Als Integrationstechnik stützt sich Merlin auf Envelopes ab. Envelopes sind bei Merlin einfache Shell-Skripte, die UNIX-Kommandozeilenaufrufe beinhalten. Ein solches Konzept ist in den vorangegangen Kapiteln schon beurteilt worden.

Die Bewertung wird in der folgenden Tabelle zusammengefaßt. Die Ergebnisse sind wie in den vorangegangenen Kapiteln notiert.

Konzeptebenen	*Spezifikation der Signatur*	*Relation der Signaturen*	*Spezifikation des Leistungsumfangs*	*Abbildung des Leistungsumfangs*	*Implementierung modellierter Aggregationen*	*Spezifikation des Zustandsmodells*	*Abbildung des Zustandsmodells*	*Spezifikation der prozeßunabhängigen Eingaben*	*Übermittlung von Feedback*	*Übermittlung von Zustandsinformation*	*Auswertung von Feedback*	*Kontrolle des Leistungsumfangs*	*Zustandsbestimmung nach Werkzeugabbruch*	*Zustandsbestimmung nach PE-Abbruch*	*Flexibilität der Werkzeugwahl*	*Wiederverwendbarkeit der Anbindung*	*Erkennung von Werkzeugabbrüchen*
ESCAPE	O		O			O											
Merlin							O	+		+		O					
Merlin-Envelopes															O	O	
gesamt	O	-	O	-	-	O	O	+	-	+	-	O	-	-	O	O	-

+: Anforderung wird vollständig erfüllt
-: Anforderung ist nicht betrachtet (nur in der Zeile „gesamt“)
O: Ebene liefert Beitrag zur Erfüllung der Anforderung, Anforderung nicht vollständig abgedeckt
kein Eintrag: kein Beitrag zu dieser Anforderung (nicht in der Zeile „gesamt“)
: betrifft diese Ebene nicht

5.3 Zusammenfassung

In diesem Kapitel sind existierende Ansätze für Anbindungstechniken und existierende PSEU vorgestellt worden.

Die vorgestellten Anbindungstechniken lassen sich in daten- und funktionsorientierte Ansätze unterteilen. Für die datenorientierten Ansätze ist gezeigt worden, daß sie ungeeignet sind, um in einer PSEU als Kommunikationsbasis zwischen den Werkzeugen und der Prozeßmaschine eingesetzt zu werden. Unter den funktionsorientierten Ansätzen sind mit den auf Nachrichten basierenden Ansätzen solche, die flexibel genug sind, um in heterogenen PSEU eingesetzt zu werden. Insbesondere CORBA ist als Basis einer Kommunikation einer solchen Umgebung gut geeignet. Um die Anbindung zu gewährleisten, muß durch Envelopes noch eine Schnittstellenumsetzung vorgenommen werden.

In existierenden PSEU werden nur wenige der Anforderungen aus Kapitel 4 erfüllt. Als Basis der Anbindung von Werkzeugen in eine PSEU werden Envelopes in verschiedenen Facetten benutzt. Diese sind in der Art, wie sie in diesen Ansätzen verwendet werden, zu unflexibel, da sie nicht einfach in einer neuen Umgebung wiederverwendet werden können oder aber zu ausdrucksschwach sind.

Im nächsten Kapitel soll einer der Ansätze benutzt werden, um die Basis für eine flexible, heterogene PSEU aufzubauen. Dazu wird aus den obigen Ansätzen die Kombination aus ESCAPE und Merlin ausgewählt. Diese Auswahl ist in dem Sinne willkürlich, daß die weiteren Betrachtungen auch mit jedem der anderen Ansätze angestellt werden könnten. Am Beispiel von ESCAPE und Merlin wird die Anwendung der entwickelten Konzepte gezeigt, mit denen Werkzeuge flexibel um eine Prozeßmaschine zu einer PSEU zusammengestellt werden.

6

TOOL INTEGRATION CONCEPT - MODELLIERUNG

In den nun folgenden zwei Kapiteln wird das Tool Integration Concept (TIC) eingeführt. TIC bildet einen Rahmen, mit dem Werkzeuge in eine heterogene PSEU integriert werden können. TIC besteht dabei aus verschiedenen Konzepten, die in einer PML bzw. PSEU enthalten sein müssen, um Werkzeuge so integrieren zu können, daß sie sich von einer PE aus steuern lassen. Kapitel 6 und 7 bieten dabei einen Leitfaden, mit dem PML und PSEU auf der Basis der Konzepte aus TIC angepaßt werden können. Die Verwendung dieser Konzepte wird durch ein Beispiel mit der PML ESCAPE und der PSEU Merlin illustriert.

Im ersten Schritt wird ein Prozeßmodell erstellt, das den Anforderungen aus Kapitel 4 genügt. Die dafür in TIC vorgesehenen Konzepte zur Beschreibung der Semantik von Aktivitäten durch eine PML werden in Kapitel 6.1 definiert. Die Spezifikation der Semantik der Aktivitäten hat nicht zum Ziel vollständig zu sein. Es wird nur soweit spezifiziert, wie es nötig ist, um die Integration von Werkzeugen in die jeweilige PSEU und eine Steuerung der Werkzeuge durch diese PSEU zu ermöglichen. Eine vollständige Beschreibung der Semantik ist nicht erforderlich, da die Komplexität unnötig erhöht würde, wobei kein weiterer Nutzen für die Anforderungen aus Kapitel 4 entstünde.

Der zweite Schritt, der unabhängig von der Erstellung eines Prozeßmodells durchgeführt wird und spezifisch für jede Arbeitsumgebung umgesetzt werden muß, ist die Beschreibung der Semantik der zu integrierenden Werkzeuge. Hierfür werden in Kapitel 6.2 neue Sprachelemente eingeführt. Die Semantikbeschreibung der Werkzeuge ist wie die der Aktivitäten auf die Anteile reduziert, die zur Integration der Werkzeuge und deren Steuerung durch die PE notwendig sind. Dabei werden die Sprachmittel so aufgebaut, daß eine Abbildung der Aktivitäten auf die Werkzeuge möglichst einfach wird.

Der dritte Schritt auf dem Weg zu einer heterogenen PSEU umfaßt eine Abbildung der Aktivitäten auf die Werkzeuge. In diesem Schritt wird eine sogenannte *Umgebungsspezifikation* erstellt, die die Semantikbeschreibungen von Aktivitäten auf die der

Werkzeuge abbildet und damit eine konkrete Umgebung spezifiziert. Die Umgebungsspezifikation wird in Kapitel 6.3 eingeführt.

Für die beispielhafte Umsetzung der TIC Konzepte in ESCAPE werden Sprachkonstrukte vorgestellt, die Teil von *ESCAPE+* sind, einer Erweiterung von ESCAPE. In dieser Arbeit werden nur die Teile von ESCAPE+ angesprochen, die benötigt werden, um Konzepte zur Anbindung von Werkzeugen in Merlin zu entwickeln, die den Anforderungen aus Kapitel 4 genügen. Dazu gehört auch die Ergänzung eines Aggregationskonzepts, das im Rahmen des Prozeßmodells benötigt wird, aber bisher in ESCAPE nicht vorgesehen war (Kapitel 6.1.2). ESCAPE+ enthält noch andere Erweiterungen, deren Inhalt und Zweck in [Sac99] nachzulesen sind. Die formale Beschreibung von ESCAPE (vgl. [Jun95], Kap. 5) wird für die in dieser Arbeit entwickelten Erweiterungen ergänzt.

6.1 Erstellung eines Prozeßmodells

Zentraler Aspekt des Verhaltens einer Aktivität sind die Veränderungen an der Menge und den Inhalten der im Prozeß bearbeiteten Dokumente oder an deren Beziehungen. Das Schreiben oder Lesen eines Dokuments bzw. einer Beziehung durch eine Aktivität wird als *Zugriff* bezeichnet.

Sollen die Anforderungen aus Kapitel 4 zur Spezifikation der Aktivitäten erfüllbar sein, so müssen für eine Aktivität mindestens die folgenden Teile ihrer Semantik spezifiziert werden können:

- Name der Aktivität
- Dokumenttyp D, auf dem die Aktivität aufgerufen wird
- Zugriffe
  - Dokumenttyp d oder Beziehungstyp b, auf den zugegriffen werden soll
  - Beziehungspfad von D zu d oder b
  - Zugriffstyp (lesend, verändernd, erzeugend, löschend)
  - Multiplizität (Anzahl der potentiell betroffenen Dokumente oder Beziehungen)
- Fortpflanzung der Aktivitäten in einer Aggregation
- Ausführungstyp der Aktivität (interaktiv, automatisch)
- Zustandsmodell der Aktivität

In Kapitel 5 wurde gezeigt, daß die verschiedenen vorgestellten PML und PSEU schon für einzelne dieser Teile Spezifikationsmöglichkeiten bieten. Umgekehrt sind für andere Teile noch gar keine oder nicht ausreichende Spezifikationsmöglichkeiten vorhanden und wenn, dann nur in einzelnen Umgebungen. Zugriffstypen, Beziehungspfade, Zugriffe auf Beziehungen, Multiplizitäten, Aggregationen und Fortpflanzung von Aktivitäten in Aggregationen sind Konzepte, die häufig fehlen. In diesem Kapitel wird gezeigt, wie die Konzepte aussehen können, die die Spezifikationen vervollständigen. Eine Zuordnung, welche Konzepte welche Anforderungen erfüllen (bzw. Mängel aus Kapitel 5 beheben), wird in den einzelnen Teilkapiteln vorgenommen und am Ende von Kapitel 7 in einer Übersichtstabelle zusammengefaßt.

6.1.1 Zugriffe

In den folgenden zwei Kapiteln werden *Zugriffe* und *Multiplizitäten* als neue Konzepte eingeführt. Mit ihnen und mit dem *Ausführungstyp* einer Aktivität (im Beispiel ESCAPE interaktiv bzw. automatisch, Kapitel 5.2.5.1, *Objektmodell*) ist es möglich, den Leistungsumfang einer Aktivität so darzustellen, wie es in der Anforderung zur *Spezifikation des Leistungsumfangs* (Kapitel 4.1.2) vorgesehen ist.

6.1.1.1 Zugriffstypen und Beziehungspfade

Zugriff: Das Schreiben oder Lesen eines Dokuments oder einer Beziehung wird als Zugriff bezeichnet.

Die Spezifikation eines Zugriffs (auf Dokumente) wird so aufgebaut, daß zunächst der Name der Aktivität sowie der Name des Dokuments, auf dem die Aktivität aufgerufen wird, angegeben werden.

Auswirkungen einer Aktivität auf die Dokumente ihrer Signatur werden im Leistungsumfang über Zugriffe und damit für jeden Dokumenttyp einzeln spezifiziert. Dazu wird die Spezifikation eines Zugriffs auf einen Dokumenttyp oder eine Beziehung durch einen *Zugriffstyp* ergänzt.

Zugriffstyp: Ein Zugriffstyp spezifiziert die Art der Auswirkungen eines Zugriffs auf ein Dokument.

Um die Anforderungen aus Kapitel 4 zu erfüllen, reichen vier Zugriffstypen auf Dokumente aus, die von den oben genannten Effekten auf das zugegriffene Dokument abgeleitet sind. Es handelt sich um die folgenden Zugriffstypen:

- *lesend* (auch: *USES*): die Aktivität läßt das Dokument unverändert
- *verändernd* (auch: *UPDATES*): das zugegriffene Dokument kann (aber muß nicht) verändert werden
- *erzeugend* (auch: *CREATES*): die Aktivität erzeugt Dokumente des angegebenen Dokumenttyps
- *löschend* (auch: *DELETES*): das betroffene Dokument bzw. die betroffenen Dokumente werden während der Ausführung der Aktivität gelöscht

Hinzu kommen zwei weitere Zugriffstypen, um Zugriffe auf Beziehungen beschreiben zu können:

- *verbindend* (auch: *CONNECTS*): die Aktivität führt eine zusätzliche Beziehung ein
- *trennend* (auch: *DISCONNECTS*): die Aktivität darf eine Beziehung löschen

Eine Aktivität darf nur auf Dokumente und Beziehungen zugreifen, die mit dem Dokument, auf dem sie aufgerufen wurde, in Beziehung stehen. Die Zugriffe werden unter Angabe der entsprechenden Beziehungstypen, auf die zugegriffen wird bzw. die die zugegriffenen Dokumenttypen in Beziehung setzen, spezifiziert. Bei der Erzeugung eines Dokuments (CREATES) wird angegeben, welche Beziehung zu dem neuen Dokument eingeführt wird. Beim Löschen eines Dokuments (DELETES) werden entsprechend alle Beziehungen dorthin entfernt.

In Abbildung 6.1 ist ein kurzes Beispiel für die Formulierung von Zugriffen einer Aktivität gegeben. Darin greift die Aktivität `compile` lesend auf Dokumente des Typs `C-Header` zu, die durch eine `includes`-Beziehung mit dem Dokument, auf dem `compile` aufgerufen wurde (im Beispiel nicht angegeben), verbunden sind. Weiterhin werden Dokumente des Typs `LIBRARY` gelesen, zu denen es eine `uses_lib`-Beziehung gibt. Die Aktivität erzeugt im Laufe ihrer Durchführung Dokumente des Typs `OBJECT`, die dann mit dem Dokument, auf dem `compile` aufgerufen wurde, durch eine `has_object-` Beziehung verbunden werden. Da bei der Spezifikation des Lei-

```
compile
USES +includes#C-HEADER;
CREATES +has_object#OBJECT;
USES +uses_lib#LIBRARY;
```

Abbildung 6.1 Beispiel für die Darstellung von Zugriffen auf Dokumente in ESCAPE+-Syntax

stungsumfangs nur Typen (Dokumenttypen und Beziehungstypen) angegeben werden, können zur Laufzeit mehrere Dokumentinstanzen mit den zugehörigen Beziehungen angesprochen sein. Auf wieviele dieser Dokumentinstanzen durch die Aktivität zugegriffen wird, wird durch die Angabe von Multiplizitäten festgelegt (vgl. Kapitel *Multiplizitäten*).

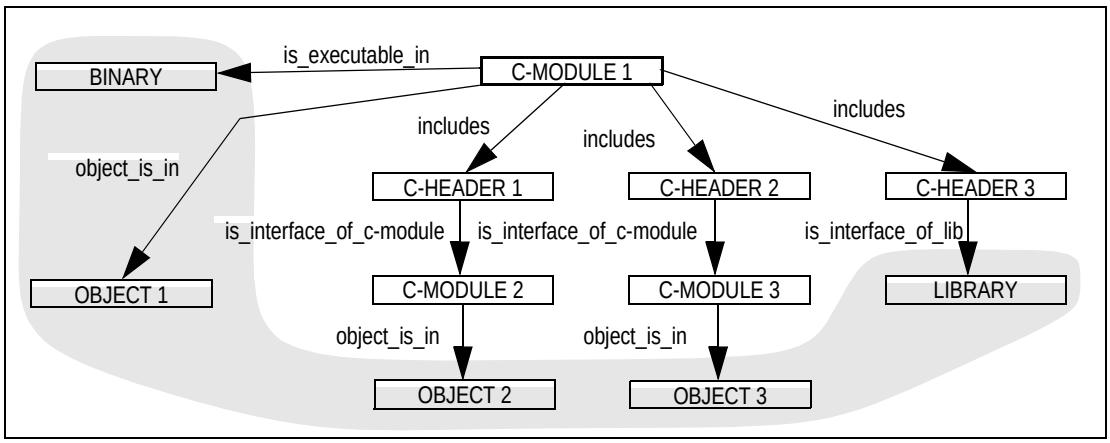
Um zuzulassen, daß ein Dokument, auf dem eine Aktivität aufgerufen wurde, nicht mit jedem zugegriffenen Dokument direkt in Verbindung stehen muß, wird definiert, daß in den Zugriffen anstelle einer einzelnen Beziehung sogenannte *Beziehungspfade* verwendet werden.

Beziehungspfad: Ein Beziehungspfad beschreibt eine Verbindung zwischen zwei Dokumenten, die auch über mehrere Beziehungen hinweg laufen kann.

Es muß sich um eine durchgehende Verkettung von Beziehungen handeln, die von dem Dokumenttyp, auf dem die Aktivität spezifiziert ist, zu dem Dokumenttyp führt, der von der Aktivität zugegriffen wird. Das am Ende des Beziehungspfades stehende Dokument ist dasjenige, auf das die Aktivität zugreift. In Abbildung 6.2 werden Beispiele für Beziehungspfade angegeben. Der obere Teil der Abbildung zeigt die zugrundeliegende Struktur des Beispiels.

Der untere Teil der Abbildung 6.2 zeigt die Definition der Aktivität `link`, die auf dem Dokumenttyp `C-MODULE` arbeitet. Der grau hinterlegte Teil der Abbildung faßt dabei die Dokumente zusammen, auf die die Aktivität zugreift. Wird die Aktivität auf dem Dokument `C-MODULE 1` aufgerufen, so wird das ausführbare Programm `BINARY` erzeugt. Zur Erzeugung werden die Objektdateien aller benutzten C-Module und die benutzten Bibliotheken (`OBJECT 2`, `OBJECT 3`, `LIBRARY`) benötigt. Diese werden indirekt über die bei der Übersetzung benutzten `C-HEADER` angesprochen, von denen aus entweder über die C-Module zu den Objektdateien oder zu den Bibliotheken navigiert wird. Als letzte Komponente wird dann noch die Objektdatei von `C-MODULE 1`, `OBJECT 1`, benötigt.

Da jede Aktivität wenigstens das Dokument bearbeitet, auf dem sie aufgerufen wird, und da für die Spezifikation des Leistungsumfangs gerade gefordert wurde, daß das zugegriffene Dokument mit dem Ausgangsdokument in Beziehung stehen muß, wird eine spezielle Beziehung benötigt. Diese beschreibt einen Beziehungspfad der Länge 1




```
link
USES +object_is_in#OBJECT;
CREATES +is_executable_in#BINARY;
USES +includes#C-HEADER+is_interface_of_c-module#C-MODULE
                +object_is_in#OBJECT;
USES +includes#C-HEADER-is_interface_of_lib#LIBRARY
```

Abbildung 6.2 Beispiel für die Benutzung von Beziehungspfaden (ESCAPE+-Syntax)

und genügt einer besonderen Semantik. Jeder Dokumenttyp steht mit sich selbst durch die Beziehung *SELF* in Verbindung, ohne daß dies im Prozeßmodell spezifiziert werden muß. Diese Beziehung verbindet jedes Dokument (d.h. jede Dokumentinstanz) mit sich selbst. Sie darf in keinem Beziehungspfad auftreten, der weitere Beziehungen enthält. Außerdem kann die Beziehung SELF nicht im Zusammenhang mit Dokumentzugriffen des Typs CREATE verwendet werden.

Um anzugeben, welche Beziehung gelöscht werden soll, benutzt man einen Beziehungspfad der Länge 1. Soll eine Beziehung neu hinzugefügt werden, so werden der Beziehungstyp und der in Beziehung zu setzende Dokumenttyp ebenfalls in einem Beziehungspfad der Länge 1 festgelegt. Bezogen auf die SELF-Beziehung gilt, daß CONNECTS- und DISCONNECTS-Zugriffe nicht auf diese Beziehung angewendet werden dürfen. Dies ist dadurch begründet, daß SELF eine Beziehung ist, die nicht angelegt oder gelöscht werden kann, sondern stattdessen immer existiert.

In Abbildung 6.3 ist ein kurzes Beispiel für die Formulierung von Zugriffen auf Beziehungen einer Aktivität gegeben. Darin erzeugt die Aktivität `edit` Beziehungen des Typs `includes` zu Dokumenten des Typs `C-Header` und darf solche auch löschen.

```
edit
CONNECTS +includes#C-HEADER;
DISCONNECTS +includes#C-HEADER;
```

Abbildung 6.3 Beispiel für die Darstellung von Zugriffen auf Beziehungen

In der gewählten Beispielsprache ESCAPE+ müssen die Zugriffsbeschreibungen, wie sie gerade eingeführt wurden, noch nachgezogen werden, da in ESCAPE keine entsprechenden Sprachmittel vorhanden sind. Die Beispiele benutzen eine konkrete Syntax, die auch als Basis für eine Beschreibung von Zugriffen in ESCAPE+ benutzt wird. In diesem Fall wird den Namen der Aktivitäten, die in der Spezifikation der Doku-

menttypen aufgelistet werden, jeweils eine solche Beschreibung der Zugriffe hinzugefügt. Die Grammatik der konkreten Syntax kann in [Sac99] nachgelesen werden.

Die ESCAPE zugrundeliegende Formalisierung muß bei einer Erweiterung der Sprachmittel ebenfalls geeignet fortgeführt werden. Der Name einer Aktivität und der Dokumenttyp, auf dem sie aufgerufen wird, sind in ESCAPE bereits definiert. Zugriffe sind in ESCAPE nicht bekannt (vgl. [Jun95], Kap.5.1). Die Zugriffstypen und Beziehungspfade und darauf aufbauend die Spezifikation der Zugriffe auf Dokumenttypen, werden daher wie folgt eingeführt:

$$\begin{gathered} \text{K: endliche, nichtleere Menge von Klassenbezeichnern} \\ \text{B: endliche Menge von Assoziationsbezeichnern} \\ \text{AK: endliche Menge von Aktivitätsbezeichnern} \\ B_{Assoziation}\text{: Assoziationsrelation, } B_{Assoziation} \subseteq K \times B \times K \\ K_{Aktivität}\text{: Aktivitätsrelation, } K_{Aktivität} \subseteq K \times AK \end{gathered} \quad \text{(EQ 1)}$$

$$\begin{gathered} ZA_K\text{: Menge der Zugriffstypen auf Klassen,} \\ ZA_K = \{ \text{ USES, UPDATES, CREATES, DELETES } \} \end{gathered} \quad \text{(EQ 2)}$$

$$\begin{gathered} \text{P: Beziehungspfade,} \\ P \subseteq B_{Assoziation} \times ... \times B_{Assoziation} \cup \text{SELF} \end{gathered} \quad \text{(EQ 3)}$$

$$\begin{gathered} Z_K\text{: Menge der Dokumentenzugriffe einer Aktivität, } Z_K \subseteq K_{Aktivität} \times ZA_K \times P \\ Z_K = \{((k, a), za, ((k_1, b_1, k_2), ..., (k_{2n-1}, b_n, k_{2n}))) \mid (k, a) \in K_{Aktivität} \wedge za \in ZA_K \wedge \\ ((k_1, b_1, k_2), ..., (k_{2n-1}, b_n, k_{2n})) \in P \wedge ((k = k_1 \wedge k_2 = k_3) \vee (k = k_2 \wedge k_1 = k_3)) \\ k_{2i} = k_{2(i+1)-1} \text{ für } 1<i<n\} \cup \\ \{((k, a), za, \text{SELF}) \mid (k, a) \in K_{Aktivität} \wedge za \in ZA_K\} \end{gathered} \quad \text{(EQ 4)}$$

$$\begin{gathered} ZA_B\text{: Menge der Zugriffstypen auf Relationen,} \\ ZA_B = \{ \text{ CONNECTS, DISCONNECTS } \} \end{gathered} \quad \text{(EQ 5)}$$

$$\begin{gathered} Z_B\text{: Menge der Beziehungszugriffe, } Z_B \subseteq K_{Aktivität} \times ZA_B \times B_{Assoziation} \\ Z_B = \{((k, a), za, (k1, b, k2)) \mid (k, a) \in K_{Aktivität} \wedge za \in ZA_B \wedge \\ (k1, b, k2) \in B_{Assoziation} \wedge (k = k_1 \vee k = k_2)\} \end{gathered} \quad \text{(EQ 6)}$$

$$ZA\text{: Menge aller Zugriffstypen, } ZA = ZA_K \cup ZA_B \quad \text{(EQ 7)}$$

$$ZU\text{: Menge aller Zugriffe, } ZU = Z_K \cup Z_B \quad \text{(EQ 8)}$$

Zugriffe bieten durch die explizite Formulierung des Leistungsumfangs der Aktivitäten auch die Möglichkeit, die Korrektheit des Organisationsmodells weitergehend, als es bisher in ESCAPE möglich war, zu überprüfen. So kann beispielsweise kontrolliert werden, ob die Zugriffsrechte auf einen Dokumenttyp, die den Rollen zugestanden werden, zu den in dem jeweils referenzierten Zustand möglichen Aktivitäten passen. Es sollte sichergestellt sein, daß eine Rolle zumindest eine Aktivität angeboten bekommt.

6.1.1.2 Multiplizitäten

Um den Leistungsumfang einer Aktivität vollständig zu spezifizieren, fehlt eine Angabe dazu, welche Teilmenge der durch einen Zugriff erreichbaren Dokumentinstanzen angesprochen wird. Dazu wird das Konzept der *Multiplizität* eines Zugriffs eingeführt, das in ähnlicher Form auch schon in FunSoft (vgl. Kapitel 5.2.1) angeboten wird.

Multiplizität: In einer Multiplizität wird auf der Typebene spezifiziert, auf wieviele der in Frage kommenden Dokumentinstanzen oder Beziehungen ein Zugriff Auswirkungen hat.

Jede Spezifikation eines Zugriffs wird durch genau eine Multiplizität verfeinert. Es werden drei Multiplizitätsarten eingeführt. Eine *ALL-Multiplizität* gibt an, daß alle Instanzen, die von dem durch diese Multiplizität beschriebenen Zugriff spezifiziert werden, von der Aktivität betroffen sind. Die *ONE-Multiplizität* beschreibt, daß es sich um genau ein Element aus der Menge aller zugreifbaren Instanzen handelt, das von der Aktivität betroffen ist. Die *SOME-Multiplizität* gibt an, daß aus der Menge der potentiell zugreifbaren Instanzen eine Teilmenge ausgewählt werden muß. Damit beinhaltet die SOME-Multiplizität die ALL- und die ONE-Multiplizität. Kann der Modellierer zur Modellierungszeit keine genauen Angaben machen, wie sich die Aktivität verhalten wird, so verwendet er die SOME-Multiplizität, da die beiden anderen Fälle durch Auswahl aller bzw. nur eines Elements erreicht werden können. Die ALL-Multiplizität hat den Vorteil, daß zur Laufzeit kein Auswahldialog mit dem Entwickler nötig ist. ONE- und SOME-Multiplizitäten erzwingen dagegen einen Dialog und können daher nicht in Zugriffen automatischer Aktivitäten verwendet werden.

Für die Umsetzung des Multiplizitätskonzepts in der Prozeßmaschine fehlt nun noch die Beschreibung der Ausführungssemantik. Abbildung 6.4 zeigt, welche Zugriffstypen in Kombination mit welchen Multiplizitäten eingesetzt werden dürfen.

Beziehung (nicht SELF)	**ONE**	**SOME**	**ALL**
CREATES, CONNECTS	0..1	0..n	–
UPDATES, USES, DISCONNECTS, DELETES	0..1	0..n, n ≤ Anzahl der verbundenen Dokumente	n, n = Anzahl der verbundenen Dokumente

Abbildung 6.4 Anzahl von Dokumenten oder Beziehungen, auf die zugegriffen wird

Der Tabelle kann entnommen werden, welche Kombinationen von Zugriffstypen, Multiplizitäten und in den Zugriffspfaden spezifizierten Beziehungen miteinander sinnvoll kombinierbar sind. Nur diese Kombinationen dürfen in Modellen verwendet werden. Da die Zugriffe CREATES, CONNECTS und DISCONNECTS für die SELF-Beziehung bereits ausgeschlossen wurden und für die restlichen Zugriffe nur die Multiplizität ONE sinnvoll ist, ist die SELF-Beziehung automatisch mit der ONE-Multiplizität ausgestattet. Sie erscheint deshalb auch nicht in Abbildung 6.4.

Das kurze Beispiel für die Formulierung von Zugriffen einer Aktivität auf Dokumente aus Abbildung 6.3 wird nun in Abbildung 6.5 erweitert. Die Aktivität `compile` greift in diesem Beispiel lesend auf alle durch eine `includes`-Kante verbundenen Dokumente des Typs `C-Header` zu und erzeugt im Laufe ihrer Durchführung genau ein Dokument des Typs `OBJECT`, das dann mit dem aktuellen Dokument durch eine `has_object` Beziehung verbunden wird.

compile
USES ALL + includes C-HEADER;
CREATES ONE + has_object OBJECT;
USES ALL + uses_lib LIBRARY;

Abbildung 6.5 Beispiel für die Verwendung von Multiplizitäten

6.1.2 Aggregierte Dokumente

Um die Anforderung zur *Implementierung modellierter Aggregationen* (Kapitel 4.1.2.1) erfüllen zu können, ist es zunächst einmal nötig, daß Aggregationen in der PML modellierbar sind. Da dies in der verwendeten Beispiel-PML ESCAPE nicht der Fall ist, wird ESCAPE in einem ersten Schritt so erweitert, daß Aggregationen modelliert werden können. Das in diesem Kapitel eingeführte Modellierungskonzept gehört damit nicht direkt zu TIC, ist aber nötig, um die TIC-Konzepte aufsetzen zu können. Das folgende Kapitel 6.1.3 zeigt dann, wie Aktivitäten auf Aggregationen spezifiziert werden.

Um strukturierte Dokumenttypen realisieren zu können, werden zwei Aggregationstypen eingeführt. Der erste Typ ist so definiert, daß ein Aggregat nicht ohne seine Komponenten existieren kann und umgekehrt. Sobald ein Aggregat dieses Typs angelegt wird, müssen auch seine Komponenten angelegt werden. Diese Form der Aggregation soll als *statische Aggregation* bezeichnet werden. Durch die Erzeugung einer statischen Aggregation in einem Zugriff wird implizit auch die Erzeugung der Komponenten mitspezifiziert.

Bezogen auf das Beispielszenario aus Kapitel 3 könnte das Pflichtenheft durch das Mittel der statischen Aggregation spezifiziert werden, um auszudrücken, daß ein `Pflichtenheft` immer aus den gleichen, festen Kapiteln zu bestehen hat (`Systembeschreibung`, `Funktionsbeschreibung` und `Benutzungsschnittstelle`). Weiterhin würde ausgedrückt, daß jede einzelne der drei Komponenten nur mit den anderen beiden zusammen, d.h. also im Kontext des Pflichtenhefts sinnvoll ist.

Der zweite Typ einer Aggregation sieht vor, daß ein Aggregat und Dokumente, die Komponenten des Aggregats sein können, unabhängig voneinander existieren können. Es wird also zunächst ein Aggregatdokument ohne Komponenten erzeugt. Die Komponenten werden separat erzeugt und dem Aggregat hinzugefügt. Das Hinzufügen wird durch Einfügen von Beziehungen eines speziellen, vordefinierten Beziehungstyps realisiert. Diese Form der Aggregation soll *dynamische Aggregation* genannt werden.

Ein Beispiel für den sinnvollen Einsatz einer dynamischen Aggregation ist der Dokumenttyp `Testdokument` (vgl. Kapitel 3). Das Dokument soll so aufgebaut sein, daß es für jeden Test eine eigene `Testbeschreibung` enthält. Zu Beginn des Teilprozesses, der sich mit dem Softwaretest beschäftigt, wird ein leeres `Testdokument` erzeugt. Es ist bei der Erzeugung keine Aussage über die Anzahl der am Ende des Testprozesses enthaltenen Testbeschreibungen möglich. Damit kann im Prozeß frei festgelegt werden, ob eine Testdokumentation alle Testergebnisse zusammenfaßt, je Testbeschreibung ein eigenes Testdokument erstellt oder je Testbeschreibung und Testlauf ein eigenes Testdokument erstellt wird (im Falle von Regressionstests). Wei-

terhin wird für eine dynamische Aggregation vom Prozeß festgelegt, ob Komponenten auch ohne ihr Aggregat existieren können. Im Falle des Testdokuments macht es dann Sinn, Testbeschreibungen von Testdokumenten zu trennen, wenn diese wiederverwendet werden, z.B. bei einem Folgerelease erneut zu einem Testdokument zusammengestellt werden sollen. Die Trennung von Dokumenten des Typs `Ergebnisdokumentation` vom konkreten Testdokument ist nicht sinnvoll.

Zur formalen Einbindung des Aggregationskonzepts in ESCAPE+ werden die folgenden Festlegungen getroffen:

$$\text{H: Hierarchierelation, } H \subset K \times K \text{ (vgl. [Jun95])} \quad \text{(EQ 9)}$$

$$E_{Aggregation}\text{:Aggregationsrelation, } E_{Aggregation} \subseteq K \times B \times K \quad \text{(EQ 10)}$$

$$\kappa: E_{Aggregation} \rightarrow \{ONE, ZEROMORE\}$$
$$\kappa(e_{Aggregation}) := ONE, \text{ wenn die Kardinalität der Aggregationsbeziehung (1,1) ist}$$
$$\kappa(e_{Aggregation}) := ZEROMORE, \text{ wenn die Kardinalität der Aggregationsbeziehung (0,n) ist} \quad \text{(EQ 11)}$$

Für die formale Einbindung des Aggregationskonzepts soll gelten:
$$k, k_1, ..., k_n \in K$$
$$ak, ak_1, ..., ak_n \in AK$$
$$b, b_1, ..., b_n \in B$$
zur Definition K, AK, B siehe (EQ 1) (EQ 12)

Es werden nun zunächst drei Hilfsprädikate definiert:

$$Abstrakt(k) \Leftrightarrow \exists k_{sub} \in K: (H(k, k_{sub})) \quad \text{(EQ 13)}$$

$$Konkret(k) \Leftrightarrow \neg Abstrakt(k) \quad \text{(EQ 14)}$$

$$Sub^*(k) = \{k_{sub} \in K | H(k, k_{sub}) \vee k = k_{sub}\} \quad \text{(EQ 15)}$$

Darauf aufbauend wird das Aggregationskonzept durch die folgenden Prädikate formalisiert:

Jede Komponente ist genau einem Aggregat zugeordnet:
$$E_{Aggregation}(k_{agg1}, b_1, k_{kompo}) \Rightarrow \neg\exists k \in K: (E_{Aggregation}(k_{agg2}, b_2, k_{kompo}) \wedge k_{agg2} \neq k_{agg1}) \quad \text{(EQ 16)}$$

Vererbung der Aggregationsbeziehung bei dynamischer Aggregation:
$$E_{Aggregation}(k_{agg}, b_1, k_{kompo}) \wedge \kappa((k_{agg}, b_1, k_{kompo})) = ZEROMORE$$
$$\Rightarrow \forall k_{subagg} \in Sub^*(k_{agg}): \forall k_{subkompo} \in Sub^*(k_{kompo}): E_{Aggregation}(k_{subagg}, b_1, k_{subkompo}) \quad \text{(EQ 17)}$$

Vererbung der Aggregationsbeziehung bei statischer Aggregation:
$$E_{Aggregation}(k_{agg}, b_1, k_{kompo}) \wedge \kappa((k_{agg}, b_1, k_{kompo})) = ONE$$
$$\Rightarrow (Abstrakt(k_{kompo}) \Rightarrow Abstrakt(k_{agg})) \quad \text{(EQ 18)}$$

Vererbung der Aggregationsbeziehung bei statischer Aggregation und konkreter Komponente:
$$E_{Aggregation}(k_{agg}, b_1, k_{kompo}) \wedge Konkret(k_{kompo}) \wedge \kappa((k_{agg}, b_1, k_{kompo})) = ONE$$
$$\Rightarrow \forall k_{subagg} \in Sub^*(k_{agg}): (H(k_{agg}, k_{subagg}) \Rightarrow E_{Aggregation}(k_{subagg}, b_1, k_{kompo})) \quad \text{(EQ 19)}$$

Vererbung der Aggregationsbeziehung bei statischer Aggregation und
abstrakter Komponente und abstraktem Aggregat:
Wenn $E_{Aggregation}(k_{agg}, b_1, k_{kompo}) \wedge \kappa((k_{agg}, b_1, k_{kompo})) = ONE \wedge$
$Abstrakt(k_{kompo}) \wedge Abstrakt(k_{agg})$
dann muß durch Redefinition von b_1 gelten
$$\forall k_{subagg} \in Sub^*(k_{agg}): \exists k_{subkompo} \in Sub^*(k_{kompo}): E_{Aggregation}(k_{subagg}, b_1, k_{subkompo}) \wedge$$
$$(\neg\exists k_{subkompo2} \in Sub^*(k_{kompo}): k_{subkompo2} \neq k_{subkompo} \wedge E_{Aggregation}(k_{subagg}, b_1, k_{subkompo2})) \quad \text{(EQ 20)}$$

Verbot von Zyklen ausschließlich statischer Aggregationsbeziehungen:
Sei

$$\text{STAT}(k_{agg}, k_{kompo}) \begin{cases} \text{TRUE wenn:} & \exists b \in B: E_{Aggregation}(k_{agg}, b, k_{kompo}) \wedge \\ & \kappa((k_{agg}, b, k_{kompo})) = \text{ONE} \\ \text{TRUE wenn:} & \exists k \in K: \text{STAT}(k_{agg}, k) \wedge \text{STAT}(k, k_{kompo}) \\ \text{FALSE sonst} & \end{cases}$$

dann muß gelten: $\forall k \in K: \neg\text{STAT}(k, k)$ (EQ 21)

Die Sichtweise, daß eine Aggregation entweder statischer oder dynamischer Natur ist, ist relativ unflexibel. So möchte man häufig auch ausdrücken können, daß einzelne Komponenten dynamisch hinzugefügt werden können, obwohl andere statisch zur Erzeugungszeit des Aggregats angelegt werden. Um dies zu erreichen, wird der Typ einer Aggregation je Komponente definiert. So kann eine Aggregation mit einer Kombination aus statisch zugehörigen und dynamisch hinzugefügten Komponenten gebildet werden. Als Beispiel für den sinnvollen Einsatz dieses Konzepts soll es im Szenario aus Kapitel 3 zusätzlich möglich sein, dem Pflichtenheft Anhänge hinzuzufügen. So wäre ein Anhang des Typs `Designdokument` an das Pflichtenheft möglich, in dem erste Gedanken zum späteren Design abgelegt sind und dort als Basis dienen können.

Durch die Aggregationsbeziehungen ergibt sich in ESCAPE+ eine Hierarchie von Dokumenttypen. Diese ist orthogonal zu der aus ESCAPE übernommenen Vererbungshierarchie aus abstrakten und konkreten Dokumenttypen. In Abbildung 6.6 ist ein Ausschnitt aus einem ESCAPE+-Objektmodell für das Beispielszenario (Kapitel 3) dargestellt. Es handelt sich um den Teil, der den Dokumenttyp `Pflichtenheft` betrifft. (Anmerkung: Die exakte Syntax in ESCAPE+ entspricht nicht der Grafik, da ESCAPE+ Aggregationen gesondert beschreibt. Die hier benutzte vereinfachte Darstellung soll das Verständnis der vorgestellten Konzepte erleichtern. Die exakte Syntax kann in der vollständigen Darstellung von ESCAPE+ in [Sac99] nachgelesen werden.)

Die Modellierung in Abbildung 6.6 stellt dar, daß sowohl das `Pflichtenheft` selbst, wie auch die Dokumenttypen `Systembeschreibung`, `Funktionsbeschreibung` und `Benutzungsschnittstelle` editierbar (durch die geerbte Aktivität `bearbeiten` mit dem Default-Zugriff `USES:SELF`) sind. Weiterhin wird durch das Symbol der Raute (vgl. Aggregation in OMT [RBP+91]) in der konkreten Syntax ausgedrückt, daß das Pflichtenheft ein Aggregat aus den anderen drei Dokumenttypen ist. Es wird spezifiziert, daß es sich im Sinn der oben beschriebenen Aggregationstypen bei allen Komponenten um eine statische Aggregation handelt, d.h. daß mit der Erzeugung eines Pflichtenhefts auch die Erzeugung von jeweils genau einer der drei Komponenten des Aggregats einhergeht. Eine statische Aggregation wird syntaktisch durch eine 1 an der Aggregationsbeziehung ausgedrückt. Soll eine dynamische Aggregation eingeführt werden, so wird die Anzahl der Komponenten nicht angegeben, wodurch eine beliebige Anzahl erlaubt wird.

Es ist für eine vollständige Betrachtung notwendig zu überlegen, was passiert, wenn Aggregationsbeziehungen unter Beteiligung von abstrakten Dokumenttypen eingeführt werden. Für das Beispiel im Kontext dieser Arbeit kann dieser Teil zunächst beiseite gelassen werden. Zur genaueren Betrachtung des Problems sei auf [Kur97], Kap. 7.2.2 und bezüglich der Vererbung von Aggregationsbeziehungen auf [Sac99] verwiesen.

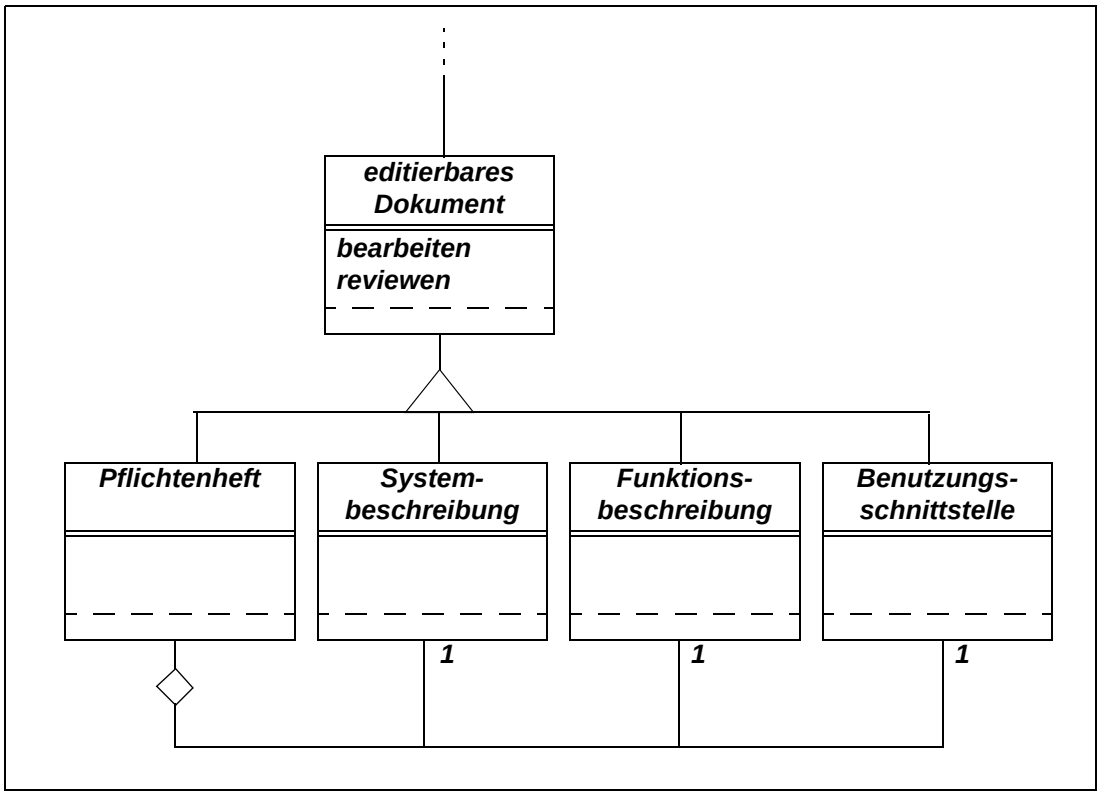


Abbildung 6.6 Beispiel für Aggregationen in ESCAPE+

6.1.3 Aktivitäten auf aggregierten Dokumenten

In einer Aggregation gelten zwischen den Aktivitäten spezielle Abhängigkeiten, die in TIC zur Erfüllung der Anforderung zur *Implementierung modellierter Aggregationen* berücksichtigt werden müssen.

Betrachtet man z.B. Aktivitäten, die ein Aggregat erzeugen bzw. löschen, das mit Komponenten in einer statischen Aggregationsbeziehung steht, so müssen diese Aktivitäten auch die Komponenten erzeugen bzw. löschen. Das bedeutet, daß die Ausführung bestimmter Aktivitäten auf dem Aggregat mit Aktivitäten auf den Komponenten gekoppelt werden muß.

Um eine solche Kopplung von Aktivitäten eines Aggregats und denen seiner Komponenten im Modell formulieren zu können, wird das Konzept der *Fortpflanzung* einer Aktivität eingeführt.

Fortpflanzung von Aktivitäten: Ruft eine vom Benutzer ausgelöste Aktivität ihrerseits weitere Aktivitäten auf, so wird dies als Fortpflanzung von Aktivitäten bezeichnet.

Im obigen Beispiel wird eine Aktivität des Aggregats auf Aktivitäten der Komponenten fortgepflanzt. Dieser Typ der Fortpflanzung wird als *aggregatbestimmte Fortpflanzung* bezeichnet.

Aggregatbestimmte Fortpflanzung: In einer aggregatbestimmten Fortpflanzung wird eine fortgepflanzte Aktivität so ausgeführt, daß die gleichnamigen Aktivitäten auf den

Komponenten des Aggregats in einer nichtdeterministischen Reihenfolge vor der Aktivität auf dem Aggregat ausgeführt werden.

Die zweite Form der Fortpflanzung wird als *komponentenbestimmte Fortpflanzung* bezeichnet.

Komponentenbestimmte Fortpflanzung: In einer komponentenbestimmten Fortpflanzung ist die Ausführung einer fortgepflanzten Aktivität so definiert, daß zunächst die gleichnamige Aktivität auf dem Aggregat und dann die ursprünglich aufgerufene Aktivität ausgeführt werden.

Wird in einer statischen Aggregation die Erzeugung einer Komponente angestoßen, so ist dies eine Situation, in der komponentenbestimmte Fortpflanzung benötigt wird, da die Komponente per Definition nur im Zusammenhang ihres Aggregats existieren kann. In diesem Fall muß also die Erzeugung des gesamten Aggregats angestoßen werden. Dies wird durch die komponentenbestimmte Aggregation erreicht. Wird also beispielsweise die Erzeugung der `Funktionsbeschreibung` des Pflichtenhefts aus dem Beispiel aus Kapitel 3 angestoßen, so muß im gleichen Zug das Aggregat Pflichtenheft erzeugt werden. Die Erzeugung des Aggregats führt nun dazu, daß eine aggregatbestimmte Fortpflanzung initiiert wird. Daher bewirkt die Erzeugung der `Funktionsbeschreibung` die Erzeugung des gesamten `Pflichtenhefts`, also auch der Komponenten `Systembeschreibung` und `Benutzungsschnittstelle`.

Formal kann angenommen werden, daß die Abarbeitung der verschiedenen fortgepflanzten Aktivitäten nichtdeterministisch passiert. Als Vorbedingung für die Ausführung einer fortgepflanzten Aktivität gilt, daß sich alle von dieser Aktivität betroffenen Dokumente in einem Bearbeitungszustand befinden, in dem die Anwendung der Aktivität erlaubt ist. Da nur die auslösende Funktion aufgerufen wird, könnte (durch Fortpflanzung) eine Aktivität auf einem Dokument aufgerufen werden, das nicht in einem Zustand ist, in dem diese Aktivität aufgerufen werden darf. Daher muß kontrolliert werden, ob eine auslösende Aktivität aufgrund der Zustände der betroffenen Dokumente alle fortgepflanzten Aktivitäten ausführen darf. Nur falls dies stimmt, darf die Aktivität auch ausgeführt werden.

In der Beispiel-PML ESCAPE kann das Fortpflanzungskonzept wie folgt formal eingebunden werden (vgl. auch die Festlegungen in Gleichung (EQ 12)):

$$\beta: K_{Aktivität} \rightarrow \{INTER, AUTO\} \text{ (vgl. [Jun95])}$$
$$\beta(k_{Aktivität}) := INTER, \text{ genau dann wenn } k_{Aktivität} \text{ eine interaktive Aktivität ist}$$
$$\beta(k_{Aktivität}) := AUTO, \text{ genau dann wenn } k_{Aktivität} \text{ eine automatische Aktivität ist} \quad \text{(EQ 22)}$$

$$F_{Fortpflanzung}: \text{Fortpflanzungsrelation}, F_{Fortpflanzung} \subseteq AK \times (K \times K) \quad \text{(EQ 23)}$$

$$\text{Fortpflanzung darf nur zwischen Aggregat und Komponente spezifiziert werden}$$
$$F_{Fortpflanzung}(ak, k_{agg}, k_{kompo}) \Rightarrow E_{Aggregation}(k_{agg}, b, k_{kompo}) \quad \text{(EQ 24)}$$

$$\text{Fortgepflanzte Aktivitäten müssen für Aggregat und Komponente definiert sein}$$
$$F_{Fortpflanzung}(ak, k_{agg}, k_{kompo}) \Rightarrow K_{Aktivität}(k_{agg}, ak) \wedge K_{Aktivität}(k_{kompo}, ak) \quad \text{(EQ 25)}$$

$$\text{Fortgepflanzte Aktivitäten müssen interaktive Aktivitäten sein}$$
$$F_{Fortpflanzung}(ak, k_{agg}, k_{kompo}) \wedge K_{Aktivität}(k_{agg}, ak) \wedge K_{Aktivität}(k_{kompo}, ak)$$
$$\Rightarrow \beta((k_{agg}, ak)) = INTER \wedge \beta((k_{kompo}, ak)) = INTER \quad \text{(EQ 26)}$$

Die konkrete Syntax in ESCAPE+ ist ein Pfeil vom Aggregat zu den Komponenten bei aggregatbestimmter Fortpflanzung bzw. ein Pfeil von den Komponenten zum Aggregat bei komponentenbestimmter Aggregation. An dem jeweiligen Pfeil wird ein Aktivitätenname aus der Schnittmenge der Aktivitätenbezeichner von Aggregat und Komponenten notiert. Wird die Abbildung 6.6 um Fortpflanzungen erweitert, so verändert sich das dortige Beispiel wie in Abbildung 6.7 gezeigt.

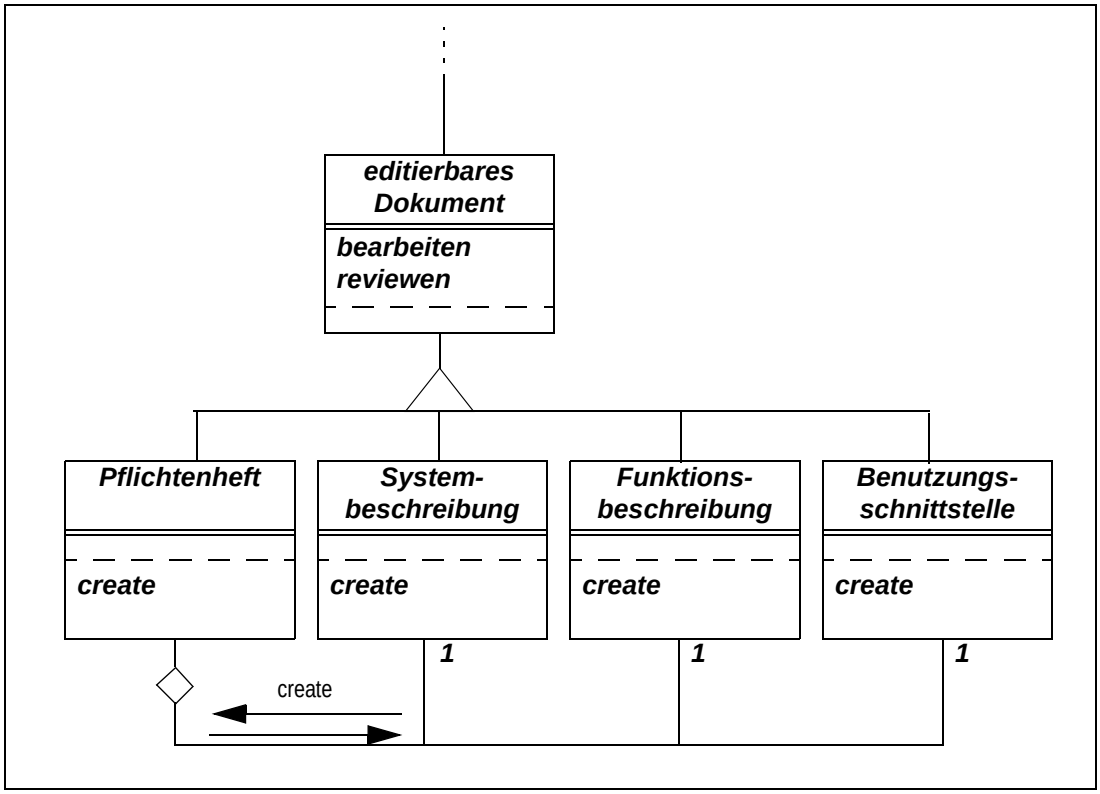


Abbildung 6.7 Beispiel für Fortpflanzung in ESCAPE+

Mit den vorgestellten Konzepten ist das Instrumentarium zur Spezifikation der Aktivitäten vollständig. Es wurde am Beispiel von ESCAPE gezeigt, wie diese Konzepte in eine PML eingebracht werden können. Dabei ist das Konzept der Aggregation ergänzt worden, um die Modellierungsmöglichkeiten von ESCAPE an die Anforderungen anzupassen.

6.2 Spezifikation der Werkzeuge

Das Ziel der Werkzeugspezifikation, die den zweiten Schritt der Erstellung einer heterogenen PSEU auf der Basis von TIC bildet, ist zum einen, eine Abbildung der Spezifikation der Werkzeuge auf die Spezifikation der Aktivitäten zu ermöglichen (vgl. Anforderung zur *Relation der Signaturen*, Anforderung zur *Abbildung des Leistungsumfangs*, Anforderung zur *Abbildung des Zustandsmodells*). Zum anderen soll durch die Spezifikation unterstützt werden, daß eine PSEU so flexibel aufgebaut werden kann, daß sich Werkzeuge leicht austauschen lassen (Anforderung zur *Flexibilität der Werkzeugwahl*, Anforderung zur *Wiederverwendbarkeit der Anbindung*).

Dazu wird in Kapitel 6.2.1 das Konzept *logischer Werkzeuge* eingeführt. Dabei handelt es sich um Schnittstellenbeschreibungen, hinter denen sich mehrere verschiedene Werkzeuge verbergen können. Wann sich Werkzeuge zu einem logischen Werkzeug

zusammenfassen lassen, wird in den Kapiteln 6.2.2 und 6.2.3 anhand dort definierter Kriterien festgelegt.

Die Spezifikation erfolgt dann konkret mit der in Kapitel 6.2.4 eingeführten Sprache zur Beschreibung logischer Werkzeuge.

6.2.1 Logische Werkzeuge

Es liegt auf der Hand, daß eine Spezifikation, die nur ein einzelnes Werkzeug beschreibt, bei der Erstellung einer PSEU sehr unflexibel ist. Wird z.B. die Modellierung auf einen `Editor A` abgestimmt, so muß für einen `Editor B`, der keine funktionalen Unterschiede zu `Editor A` aufweist, eine eigene Spezifikation erstellt, ein eigener Aufruf spezifiziert oder möglicherweise sogar die Modellierung der PSEU geändert werden. Ein solches Vorgehen widerspricht der geforderten Flexibilität einer heterogenen PSEU.

Um dem zu begegnen, abstrahiert TIC bei der Spezifikation der Werkzeuge von den konkreten Werkzeugen und führt ähnlich der Spezifikation der Aktivitäten eine abstrakte Beschreibung für Werkzeuge ein. Mit einer solchen Beschreibung ist es möglich, ähnliche Werkzeuge zu einer Art Werkzeugfamilie zusammenzufassen.

Logisches Werkzeug: Eine Schnittstellenbeschreibung, die eine Werkzeugfamilie durch eine gemeinsame Spezifikation von Signatur, Leistungsumfang und Zustandsmodell repräsentiert, wird als *logisches Werkzeug* bezeichnet.

Der Nutzen logischer Werkzeuge liegt auf der Hand: Können die Editoren aus dem obigen Beispiel gemeinsam beschrieben werden, so ist ein Wechsel des Werkzeugs innerhalb der gleichen Familie nicht sichtbar, da auf der Ebene des Prozeßmodells nur das logische Werkzeug referenziert wird. Die Referenzen auf die konkreten Werkzeuge entfallen hier vollständig.

Eine PSEU wird auf der Basis logischer Werkzeuge zusammengestellt, die dann im jeweiligen Umfeld konkret instanziiert werden. Verschiedene Umfelder können z.B. unterschiedliche Plattformen sein, oder es handelt sich z.B. um Rechner mit der gleichen Betriebssystemplattform, aber z.B. der `Editor A`, der auf den Rechnern des Entwicklungsstandortes `A` benutzt wird, ist auf den Rechnern des Entwicklungsstandortes `B` nicht verfügbar. In Abbildung 6.8 wird der Sachverhalt in einem Beispiel skizziert.

Der Software-Entwicklungsprozeß aus dem Szenario aus Kapitel 3, das dem Beispiel von Abbildung 6.8 zugrunde liegt, ist auf mehrere Standorte verteilt. Diese Standorte und teilweise sogar die einzelnen Benutzer sind nun frei, die jeweilige Implementierung von `Editor` an das lokale Arbeitsumfeld anzupassen. An `Standort A` wird von `Benutzer 1` als Implementierung von `Editor` `wordpad` verwendet, `Benutzer 2` an `Standort B` benutzt stattdessen als Editor `notepad`. Zuhause auf seinem privaten Computer verwendet `Benutzer 1` unter `Linux` den `emacs`.

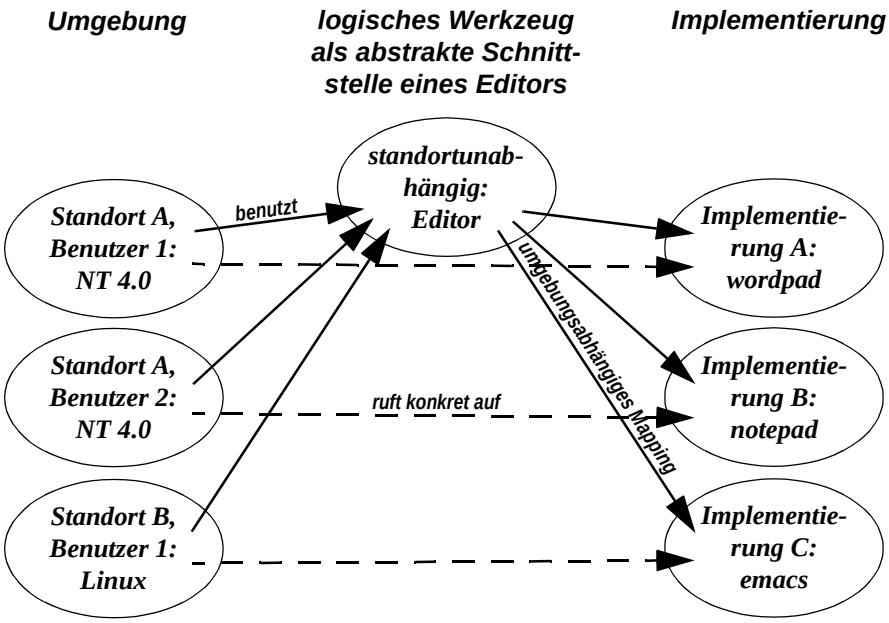


Abbildung 6.8 Zuordnung von Aktivitäten zu log. Werkzeugen

6.2.2 Signatur und Leistungsumfang

In der Signatur eines logischen Werkzeugs werden im Unterschied zur Signatur von Aktivitäten zwei Typen von Eingaben unterschieden:

Primäreingaben: Die einem Werkzeug übergebenen Referenzen auf Ein- oder Ausgabedokumente werden als *Primäreingaben* bezeichnet.

Sekundäreingaben: Eingaben, die prozeßunabhängige Informationen (vgl. Kapitel 4.1.4) übergeben, werden als *Sekundäreingabe* bezeichnet.

Primäreingaben entsprechen den Eingaben der Aktivitäten auf der Werkzeugebene.

Bietet ein logisches Werkzeug die Eingabe von Sekundäreingaben an, so bedeutet das, daß das Werkzeug seine Aufgabe auf verschiedene Weisen erfüllen kann. Als Beispiel dient ein Compiler, der den Objektcode entweder mit oder ohne Debug-Informationen erzeugen kann. Die gültigen Werte für eine Sekundäreingabe listen demnach die verfügbaren Ausführungsalternativen auf. Bezogen auf das Beispiel des Compilers wäre dort entsprechend eine Sekundäreingabe mit den Werten `debug` oder `no_debug` denkbar. Die möglichen Werte werden durch die Angabe von Aufzählungstypen definiert.

Für Primäreingaben werden Typen spezifiziert, die Formate für die physikalische Ablage der Dokumente festlegen. Damit ist gewährleistet, daß Dokumente von einem beliebigen Werkzeug, das durch das gleiche logische Werkzeug repräsentiert wird, weiter bearbeitet werden können.

Sollen Werkzeuge durch ein logisches Werkzeug abstrahiert werden, so muß für jedes einzelne Werkzeug gelten: Seine Eingaben bilden eine Teilmenge der durch das logische Werkzeug spezifizierten Eingaben. Dieses Kriterium bezieht sich sowohl auf die

Primär- als auch auf die Sekundäreingaben. Es ist dann möglich, dem Werkzeug alle Eingaben des logischen Werkzeugs zu übergeben und die nicht benötigten Anteile unbenutzt zu lassen.

Eingaben: Werkzeuge, die durch ein gemeinsames logisches Werkzeug repräsentiert werden, werden mit denselben Primär- und Sekundäreingaben aufgerufen, wobei ein Werkzeug auch nur eine Teilmenge der Eingaben verwenden kann.

Für die Ausgaben gilt analog zu den Primäreingaben, daß alle durch ein logisches Werkzeug repräsentierten Werkzeuge die gleichen Ausgabedokumente haben müssen. Damit ist verbunden, daß der Leistungsumfang im Sinne erzeugter, gelöschter, veränderter und gelesener Dokumente identisch sein muß. Auch die vom logischen Werkzeug veränderten Beziehungen müssen sich auf die gleichen zu verbindenden Typen von Dokumenten beziehen.

Ausgaben und Leistungsumfang: Werkzeuge, die durch ein gemeinsames logisches Werkzeug repräsentiert werden, erzeugen dieselben Ausgaben und Wirkungen, d.h. sie lesen bzw. verändern die gleichen Eingabedokumente und erzeugen bzw. löschen die gleichen Dokumente und Beziehungen.

Anzumerken ist, daß die obigen Kriterien insbesondere bedeuten, daß die in der Signatur verwendeten physikalischen Datentypen in allen gemeinsam repräsentierten Werkzeugen übereinstimmen müssen.

Wird ein neues Dokument erzeugt, so ist es wichtig, daß sein Typ vorher festgelegt ist. Bei der Vergabe des konkreten Dokumentnamens kommt es darauf an, ob der Name beim Entwickler erfragt werden muß oder nicht. Falls der Name durch die Prozeßsteuerungskomponente festgelegt oder vom Werkzeug generiert werden kann, muß mit dem Entwickler kein Dialog geführt werden, d.h. solche Werkzeuge können eine automatische Aktivität repräsentieren. Wenn der Name jedoch beim Entwickler erfragt werden muß, kann das Werkzeug nur zur Implementierung einer interaktiven Aktivität eingesetzt werden.

Die vorangegangenen Bemerkungen resultieren in folgendem Kriterium:

Referenzen: Werkzeuge, die durch ein gemeinsames logisches Werkzeug repräsentiert werden, besitzen die gleichen Fähigkeiten zur Vergabe von Referenzen auf Dokumente der Prozeßausführung.

6.2.3 Zustandsmodell

Zur Spezifikation der Werkzeuge gehört auch die Spezifikation des Zustandsmodells (vgl. Anforderung zur *Spezifikation des Zustandsmodells* in Kapitel 4.1.3). Werte, die von den Werkzeugen als Information über ihren Verlauf zurückgegeben werden, sind deren *Rückgabewerte*. In logischen Werkzeugen werden diese Rückgabewerte in Form von symbolischen Werten berücksichtigt, die konkreten Implementierungen abstrahieren. Dabei gilt, daß der Ausgangszustand des Dokuments aus Sicht des Werkzeugs unbekannt ist. Nach der Bearbeitung durch das Werkzeug geben dessen Rückgaben Auskunft über den Verlauf der Bearbeitung.

Symbolische Rückgaben: Die durch ein logisches Werkzeug spezifizierten Rückgaben werden als *symbolische Rückgaben* bezeichnet. Diese beschreiben das Zustandsmodell eines Werkzeugs, das von einem unbekannten Ausgangszustand in einen durch den Rückgabewert bestimmten Folgezustand führt (vgl. Abbildung 6.9). Der Zustand des Dokuments nach der Bearbeitung muß dann in einen Zustand des Prozeßmodells abgebildet werden.

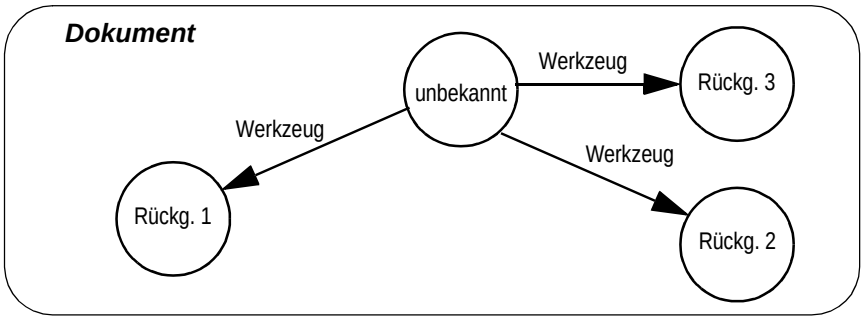


Abbildung 6.9 Allgemeines Zustandsmodell für Werkzeuge

Es wäre durchaus möglich, auch den Ausgangszustand eines Werkzeugs näher zu spezifizieren. Dadurch würde sich möglicherweise die Einsetzbarkeit eines Werkzeugs als Implementierung von Aktivitäten klarer definieren. Da man beim Konfigurieren einer Umgebung aber den Abgleich, ob Prozeßmodellzustand und Werkzeugzustand zueinander passen, manuell durchführen muß, ergibt sich kaum ein Vorteil.

Ein weiteres Kriterium, das zwei durch das gleiche logische Werkzeug repräsentierte Werkzeuge beschreibt, ist daher:

Rückgabewerte: Die Rückgabewerte von Werkzeugen, die durch ein gemeinsames logisches Werkzeug repräsentiert werden, können surjektiv auf einen gemeinsamen Satz symbolischer Rückgabewerte abgebildet werden.

Werden dabei mehrere verschiedene Rückgabewerte auf den gleichen symbolischen Wert abgebildet, so verliert man Informationen, die auf der Ebene des Werkzeugs vorhanden waren. Kennt das logische Werkzeug `Compiler` z.B. den symbolischen Rückgabewert `fehlerhaft_übersetzt`, und ein Werkzeug kennt den Unterschied von `übersetzt_mit_Warnung`, `übersetzt_mit_Fehlern`, `fataler_Fehler`, so geht die Differenzierung verloren, wenn die konkreten Rückgaben auf die symbolische Rückgabe abgebildet werden. Bei der Erstellung eines logischen Werkzeugs muß also sehr darauf geachtet werden, daß die Schnittstelle auch an dieser Stelle eine ausreichende Breite besitzt. Sind in einem logischen Werkzeug mehr symbolische Rückgaben vorgesehen, als durch ein Werkzeug angeboten werden, so ist eine Repräsentation durch das logische Werkzeug nicht mehr möglich. Der Grund ist, daß einzelne symbolische Rückgaben keiner Werkzeugrückgabe entsprechen und damit später in der Abbildung auf das Zustandsmodell der Aktivitäten nicht mehr alle Übergänge sichergestellt werden können.

6.2.4 Spezifikation logischer Werkzeuge

Die Sprache, in der die logischen Werkzeuge spezifiziert werden können, wird in diesem Kapitel eingeführt. Ein logisches Werkzeug enthält die folgenden Bestandteile (die Reihenfolge der Auflistung entspricht nicht der konkreten Syntax):

Name des logischen Werkzeugs

Liste von Übergabeparametern
Dokumentparameter (Primäreingaben, Ausgaben und Leistungsumfang bezogen auf Dokumente):
1. Teil: Variablenname + Datentyp
2. Teil: Bearbeitungsart (USES, UPDATES, ...)
Beziehungsparameter (Leistungsumfang bezogen auf Beziehungen):
1. Teil: Variablenname + Datentyp
2. Teil: Bearbeitungsart (CONNECTS, DISCONNECTS)
3. Teil: Beziehungstyp
Sekundäreingaben:
1. Teil: Variablenname + Datentyp

Menge von Rückgabewerten

In Abbildung 6.10 ist ein Beispiel dargestellt, anhand dessen der Aufbau eines logischen Werkzeugs erläutert wird. Die vollständige Grammatik der Spezifikationssprache für logische Werkzeuge ist im Anhang B zu finden.

Der *Name eines logischen Werkzeugs* ist der Name, unter dem ein Werkzeug als Realisierung einer Aktivität aufrufbar ist. In Abbildung 6.10 ist das logische Werkzeug `compile` spezifiziert worden.

Die *Übergabeparameter eines logischen Werkzeugs* enthalten mindestens den Namen einer Variable, in der der jeweilige Parameterwert übergeben wird, und den Typ des Parameters. Diese Angaben werden je nach Art des Parameters durch weitere ergänzt, die in der Folge für jede Parameterart einzeln erläutert werden.

In Dokumentparametern steht der Variablenname für einen Dokumentnamen. Der Typ des Dokuments ist in der Prozeßausführung bekannt und legt eine Dokument-Datenstruktur fest, in der das betroffene Dokument physikalisch abgelegt ist oder werden soll. Der im Beispiel in Abbildung 6.10 verwendete Dokumenttyp `UNIX-ASCII` steht z.B. für ein als unstrukturierter Text in ASCII-Codierung abgelegtes Dokument. Entsprechend handelt es sich beim Typ `OBJECT` um einen Typ, der ein strukturiert abgelegtes Dokument beschreibt. Die Struktur ist dabei die einer Binärdatei, die einen linkfähigen Objektcode beinhaltet. Wie die Struktur aussieht, die sich hinter einem gewählten Namen verbirgt, ist aus der Schnittstelle des logischen Werkzeugs nicht zu erkennen. Auf diesen Aspekt wird im Kapitel 6.3 noch eingegangen.

Die Angabe des Typs eines Parameters kann durch das Schlüsselwort SET OF ergänzt werden. Damit können Mengen von Dokumenten übergeben werden, deren genaue Elementzahl nicht bekannt ist. Im Beispiel wird dieses Hilfsmittel u.a. dazu verwendet, der logischen Abstraktion eines C-Compilers die benutzten Include-Dateien zu übergeben, von denen natürlich bei der Definition des logischen Werkzeugs nicht bekannt ist, wieviele tatsächlich benutzt werden.

Ergänzt werden Dokumentparameter durch Schlüsselwörter, die beschreiben, wie ein logisches Werkzeug den jeweiligen Parameter bearbeiten wird. Es gibt Entsprechungen für alle Dokument-Zugriffstypen (USES, UPDATES, CREATES, DELETES). Das bedeutet, daß die Werkzeuge, die durch ein logisches Werkzeug repräsentiert wer-

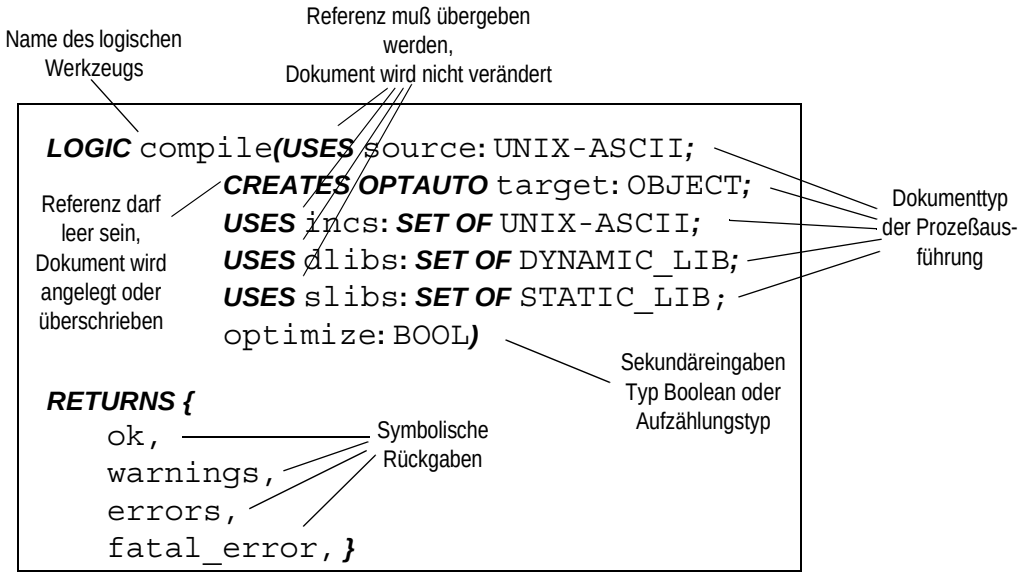


Abbildung 6.10 Beispiel für die Beschreibung logischer Werkzeuge

den, einmalig sehr genau auf ihr Verhalten untersucht werden müssen. Kapitel 7.2.3 geht darauf ein, wie diese Zusicherung ausgenutzt wird.

Soll ein Dokument neu angelegt werden, so ist durch den Dokumenttyp zwar festgelegt, in welchem Ausgabeformat das Dokument nach der Erzeugung vorliegen soll, wie aber der physikalische Name für das Dokument (der Name, der der Prozeßausführung bekannt ist) zu vergeben ist, ist noch nicht klar. Durch die Zusätze AUTO, OPTAUTO und OPTUSER zu dem Schlüsselwort CREATES werden hierfür weitere Spezifizierungsmöglichkeiten geschaffen. Keine Ergänzung bedeutet zunächst einmal, daß der Aufrufer einen Namen vorgeben muß. OPTAUTO definiert, daß entweder durch den Aufrufer ein Name vorgegeben ist oder daß das Werkzeug diesen automatisch erzeugt. Mit OPTUSER wird spezifiziert, daß entweder ein Name vorgegeben ist oder daß der Entwickler diesen manuell vergibt. Damit macht OPTUSER nur bei logischen Werkzeugen Sinn, die als Realisierung benutzerinitiierter Werkzeuge vorgesehen sind. AUTO steht für die automatische Erzeugung der Referenz durch das Werkzeug.

Beziehungsparameter betreffen Beziehungen, die zwischen dem Dokument, das hauptsächlich bearbeitet wird, und anderen Dokumenten vorgesehen sind und vom logischen Werkzeug erzeugt oder gelöscht werden können. Für Beziehungsparameter werden entsprechende Bearbeitungsarten für die Parameter vorgesehen (CONNECTS, DISCONNECTS). Die hier verwendbaren Schlüsselwörter entsprechen den auf den Beziehungszugriffen definierten Möglichkeiten, so daß die spätere Abbildung der Aktivitäten auf die logischen Werkzeuge vereinfacht wird. Für Beziehungsparameter muß auch noch der Typ der Relation spezifiziert werden, die gelöscht bzw. neu erzeugt werden soll. Ein Beispielauszug aus der Spezifikation eines logischen Werkzeugs, in dem auch Beziehungen verändert werden, ist in Abbildung 6.11 zu sehen.

Für Sekundäreingaben gibt es die Möglichkeit, einen vordefinierten Typ zu verwenden oder aber selbst einen Aufzählungstyp zu definieren, der die erlaubten Werte für den

```
LOGIC bearbeiten(USES source: UNIX-ASCII;
                 CONNECTS incs: SET OF UNIX-ASCII REL includes;
                 DISCONNECTS incs: SET OF UNIX-ASCII REL includes)
...
```

Abbildung 6.11 Beispielauszug aus einem logischen Werkzeug, das Beziehungen verändert

Parameter enthält. Als vordefinierter Typ ist nur der Typ Boolean vorgesehen, der durch das Schlüsselwort BOOL repräsentiert wird (vgl. Abbildung 6.10).

Allen logischen Werkzeugen gemein ist die Ausgabe einer Menge von symbolischen Rückgabewerten. Diese werden durch die Angabe eines Aufzählungstyps als Rückgabe des logischen Werkzeugs aufgelistet.

Da spezifiziert ist, welche Wirkung ein logisches Werkzeug auf die Dokument- bzw. Beziehungsparameter hat (im Sinn der Zugriffstypen), wird es für jedes Ein- bzw. Ausgabedokument möglich zu überprüfen, ob der Leistungsumfang des Werkzeugs mit dem der zu realisierenden Aktivität übereinstimmt. Dazu muß es eine Abbildung des fraglichen logischen Werkzeugs auf die zu implementierende Aktivität geben. Falls dies nicht der Fall ist, kann das Werkzeug nicht als Implementierung der Aktivität eingesetzt werden.

Mit der Spezifikation der logischen Werkzeuge sind nun die drei Anforderungen aus Kapitel 4.1, die die Spezifikation betreffen, erfüllt (Anforderungen zur *Spezifikation der Signatur*, zur *Spezifikation des Leistungsumfangs* und zur *Spezifikation des Zustandsmodells*). Im folgenden wird nun gezeigt, daß sich die im Prozeßmodell spezifizierten Aktivitäten auf die logischen Werkzeuge abbilden lassen. Dazu wird geprüft, ob es Relationen und Abbildungen in der Form gibt, wie sie in den Anforderung zur *Relation der Signaturen*, *Abbildung des Leistungsumfangs* und *Abbildung des Zustandsmodells* festgelegt sind. Auch die *Spezifikation der prozeßunabhängigen Eingaben* ist noch nicht erfüllt und wird erst in Kapitel 6.3.3 vervollständigt. Wie die Relation und die Abbildung im einzelnen aussehen, wird im folgenden Kapitel erläutert.

6.3 Umgebungsspezifikation

In diesem Kapitel wird dargestellt, wie die Modellierungskonzepte aus Kapitel 6.1 mit den logischen Werkzeugen in Beziehung gesetzt werden können. Dazu werden die jeweiligen Konzepte des Prozeßmodells und die logischen Werkzeuge aufeinander abgebildet.

Umgebungsspezifikation: Die Beschreibung der Abbildung von Aktivitäten auf logische Werkzeuge und umgekehrt wird als *Umgebungsspezifikation* bezeichnet und definiert durch die Abbildung eine prozeßspezifische, heterogene PSEU.

Sie definiert damit die Instanziierung einer PSEU, wie sie in Kapitel 4.1 eingeführt worden ist. Durch die Umgebungsspezifikation werden die Anforderungen zur *Relation der Signaturen*, zur *Abbildung des Leistungsumfangs* und zur *Abbildung des Zustandsmodells* sowie die Anforderung zur *Spezifikation der prozeßunabhängigen Eingaben* abgedeckt.

6.3.1 Dokumentklassen

Dokumentklasse: Eine Zusammenfassung aller Abbildungsvorschriften von Aktivitäten auf logische Werkzeuge, die alle Vorschriften enthält, die sich auf einen Dokumenttyp beziehen, heißt *Dokumentklasse*.

Eine Umgebungsspezifikation kann vollständig in Dokumentklassen aufgeteilt werden. Die Dokumentklassen strukturieren eine Umgebungsspezifikation. Sie besitzen für jede Aktivität, die im korrespondierenden Dokumenttypen spezifiziert wird, genau eine Abbildungsvorschrift, in der die Abbildung der Aktivität auf ein logisches Werkzeug realisiert wird.

Methode: Eine einzelne Abbildungsvorschrift in einer Dokumentklasse heißt *Methode*.

Die „Implementierung“ dieser Methoden stützt sich auf den logischen Werkzeugen ab, indem die Methoden logische Werkzeuge aufrufen. Dieses Vorgehen ist in Abbildung 6.12 dargestellt.

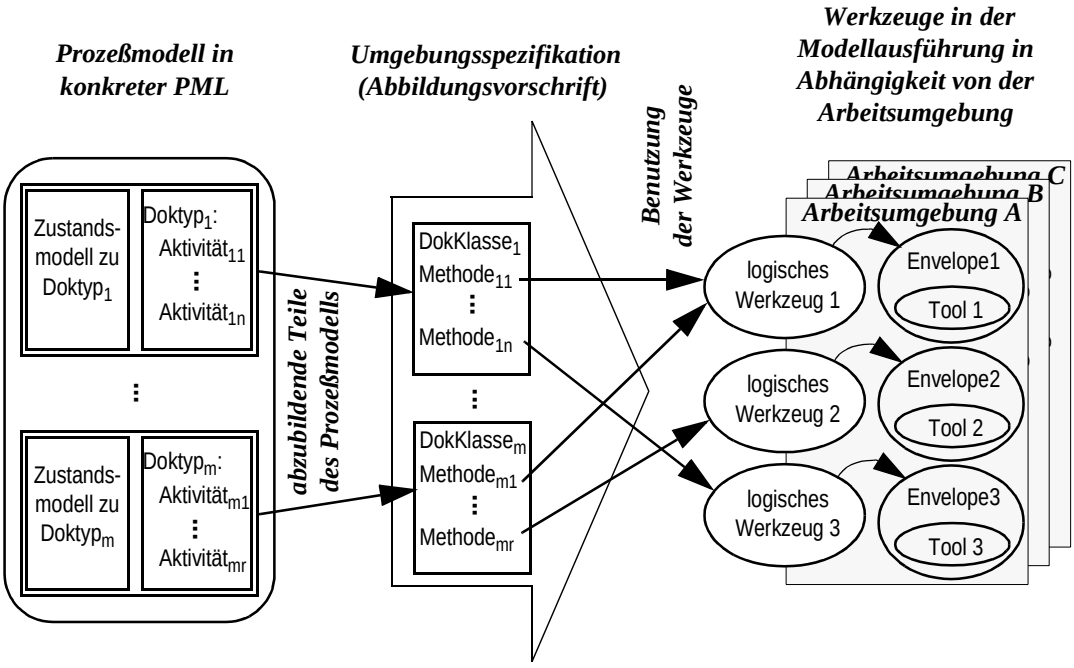


Abbildung 6.12 Zusammenhang von Prozeßmodell, Umgebungsspezifikation und logischen Werkzeugen

Abbildung 6.13 zeigt auf der rechten Seite ein Beispiel für eine Dokumentklasse, deren Methoden noch nicht formuliert sind. Der Name der Dokumentklasse und die Namen ihrer Methoden werden aus den Dokumenttypen des Prozeßmodells bzw. aus deren Aktivitäten abgeleitet. Der gezeigte Dokumenttyp ist der Dokumenttyp `Systembeschreibung` aus dem Beispiel in Kapitel 3. Diese und die folgenden Beispiele sind syntaktisch auf der Basis von ESCAPE+ formuliert worden*. Generell kann auf der lin-

* Die Verwendung von Spezifikationssprachen, die aus Allgemeinheitsgründen englisch formuliert sind, zusammen mit dem deutsch gehaltenen Beispiel aus Kapitel 3 führt zu einer Mischung der Sprachen, die an dieser Stelle nicht vermieden werden kann.

ken Seite des Bildes aber natürlich jede im Sinne von Kapitel 6.1 definierte Sprache verwendet werden.

Die vollständige Syntax für die Umgebungsspezifikation kann in Anhang A nachgelesen werden.

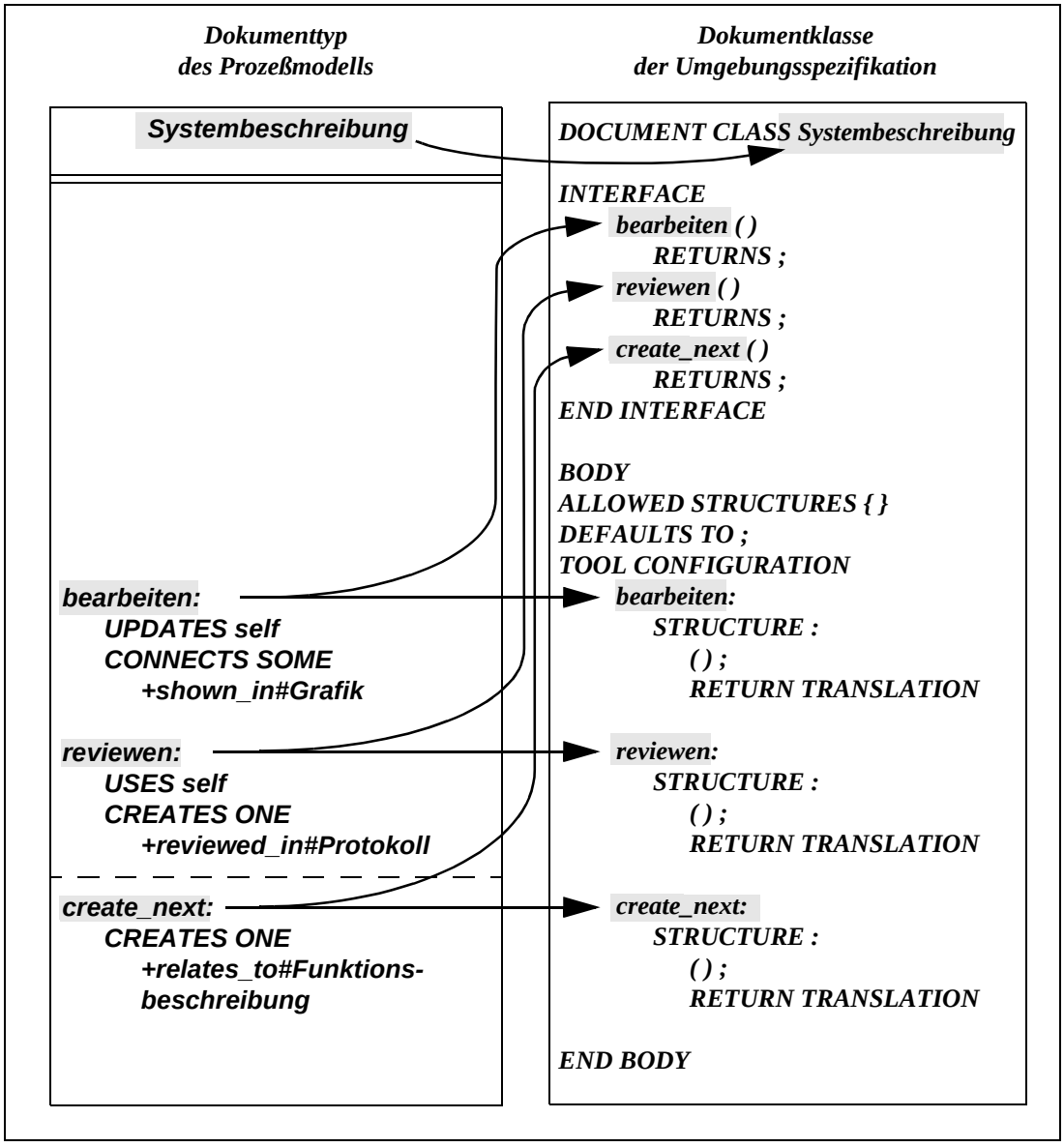


Abbildung 6.13 Dokumentklassen: Ableitung der Benennung

In den Dokumentklassen wird die abstrakte (Schnittstellen-) Sicht von der konkreten (Implementierungs-) Sicht getrennt. Diese Abschnitte sind in der konkreten Syntax durch die Schlüsselwörter *INTERFACE* beziehungsweise *BODY* gekennzeichnet.

Der im Beispiel dargestellte Dokumenttyp `Systembeschreibung` legt drei Aktivitäten fest, die auf ihm aufgerufen werden können (linke Seite des Bildes). Die Namen des Dokumenttyps und seiner Aktivitäten (im Bild grau hinterlegt) legen die Namen einer Dokumentklasse der Umgebungsspezifikation und der darin spezifizierten Methoden fest (grau hinterlegte Informationen auf der rechten Seite des Bildes). Die Definition der Dokumentklasse nutzt aus, daß die Konzepte zur Beschreibung von

Aktivitäten und logischen Werkzeugen ähnlich definiert sind. Es gibt größtenteils direkte Entsprechungen von Konzepten, weswegen die Formulierung der Umgebungsspezifikation auch als Erweiterung einer bestehenden Spezifikation eines Prozeßmodells betrachtet werden kann.

6.3.2 Abzubildende Anteile des Prozeßmodells

6.3.2.1 Ausführungstyp

Eine Information, die für die Werkzeuganbindung bedeutend ist, ist der *Ausführungstyp* der Aktivität, der festlegt ob eine Aktivität automatisch oder interaktiv ausgeführt wird. Diese Information wird direkt aus dem Prozeßmodell, in dem sie nach Kapitel 6.1 enthalten ist, in die Abbildung auf die logischen Werkzeuge eingebracht (vgl. Abbildung 6.14). Darin stehen die Schlüsselwörter *AUTO* für automatische Werkzeuge und *USER INTERACTIVE* für benutzerinitiierte Aktivitäten mit notwendigem Dialog mit dem Benutzer.

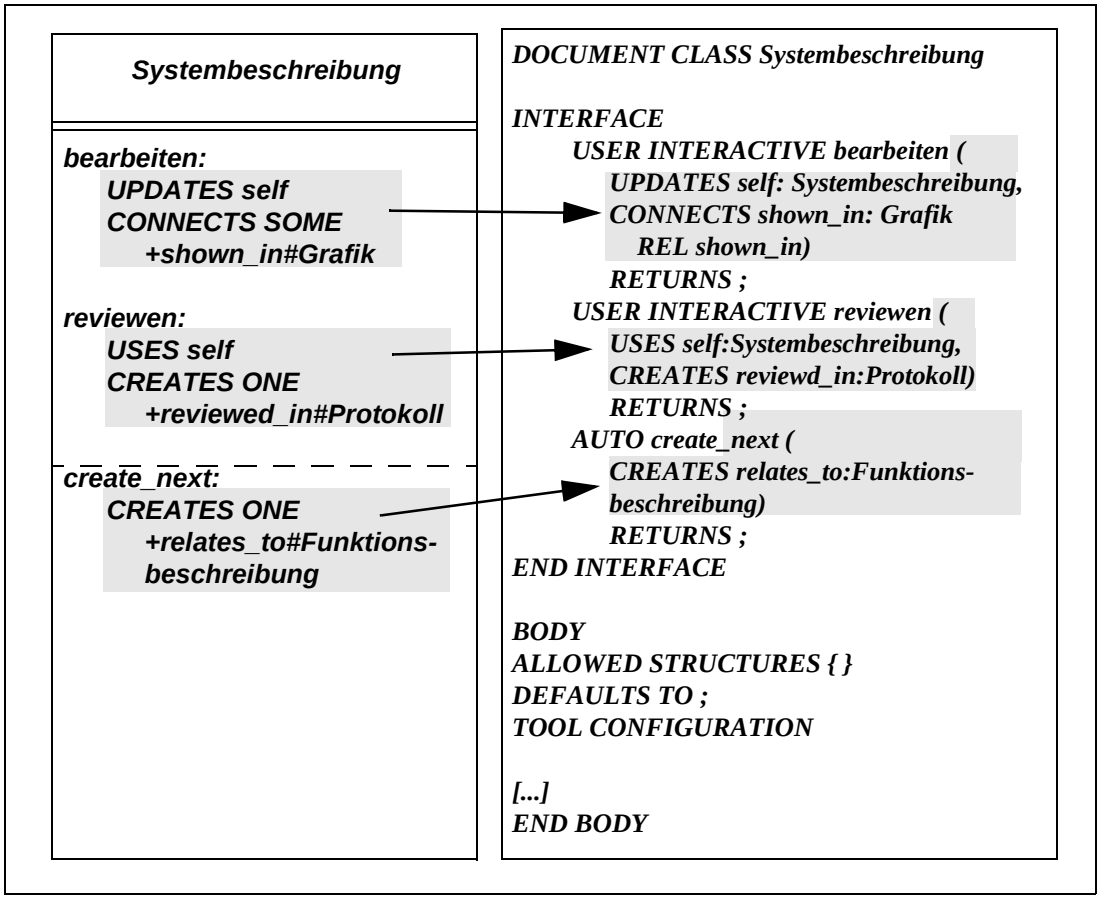


Abbildung 6.14 Übersetzung der Prozeßmodellinformationen in die Umgebungsspezifikation, Zugriffe

6.3.2.2 Zugriffe

Um den Leistungsumfang der Aktivitäten vollständig auf die logischen Werkzeuge abbilden zu können, müssen zusätzlich zu den Ausführungstypen der Aktivitäten auch

deren Zugriffe in die Umgebungsspezifikation übernommen werden. Die Zugriffe können als Schnittstellenbeschreibungen der Aktivitäten auf der Ebene des Prozeßmodells verstanden werden (vgl. Kapitel 6.2.2). Zugriffe werden daher in der Umgebungsspezifikation als Parameter der Methoden eingetragen. Dazu werden die entsprechenden Schlüsselwörter aus der Spezifikation der Aktivitäten in die Umgebungsspezifikation übernommen. Der Parametername und der Parametertyp werden aus den verwendeten Beziehungspfaden abgeleitet. Ein Beispiel dafür, wie die Zugriffe in die Umgebungsspezifikation eingebracht werden, ist in Abbildung 6.14 (auf Seite 100) dargestellt.

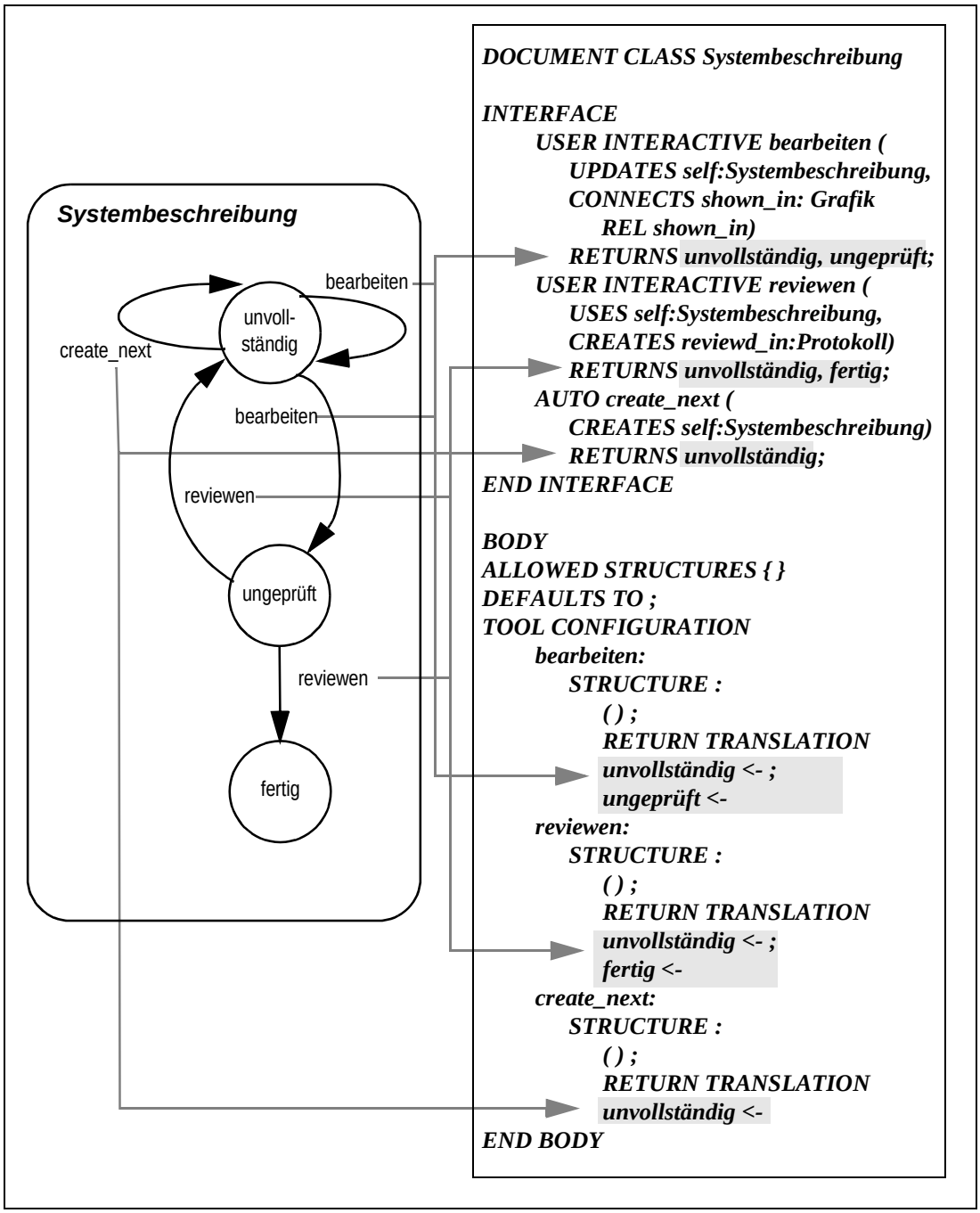


Abbildung 6.15 Übersetzung der Prozeßmodellinformationen in die Umgebungsspezifikation, Bearbeitungsstände der Dokumente

6.3.2.3 Zustandsmodell

Im INTERFACE-Abschnitt der Umgebungsspezifikation werden nun noch die durch die Aktivitäten erreichbaren Prozeßmodellzustände ergänzt. Die Bearbeitungsstände werden in der Umgebungsspezifikation dazu benötigt, die tatsächlichen Rückgaben der Werkzeuge auf die Modellzustände der Dokumente abzubilden. In der PML ESCAPE+ wird die Zustandsinformation der Dokumente beispielsweise aus dem Koordinationsmodell gezogen. Dort sind die jeweiligen Bearbeitungszustände eines Dokumenttyp nach der Durchführung einer Aktivität direkt aufgelistet. Abbildung 6.15 (auf Seite 101) zeigt diesen Teil der Umgebungsspezifikation und die Abbildung eines ESCAPE+-Zustandsmodells in die Umgebungsspezifikation.

Mit diesen Informationen ist das Prozeßmodell bezogen auf die Umgebung zunächst vollständig ausgewertet. Alle weiteren Informationen sind Teil der eigentlichen Instanziierung der Umgebung und werden manuell bei der Erstellung der Umgebungsspezifikation hinzugefügt.

6.3.3 Abbildung von Aktivitäten auf logische Werkzeuge

In diesem Kapitel wird die Umgebungsspezifikation verfeinert und vervollständigt. Dazu werden Vorschriften angegeben, die die aus dem Prozeßmodell in die Umgebungsspezifikation übernommenen Konzepte auf logische Werkzeuge abbilden. In Abbildung 6.16 wird das Beispiel aus Kapitel 6.3.2 aufgegriffen. Dort wird dargestellt, welche Positionen noch zur Vervollständigung der Umgebungsspezifikation benötigt werden.

Im Beispiel aus Abbildung 6.16 (auf Seite 103) kann man sehen, daß sich das Interface der Dokumentklassen zur Beschreibung der Anbindung an die Prozeßmaschine vollständig aus dem ESCAPE+-Prozeßmodell ableiten läßt. Dies ist eine notwendige Voraussetzung dafür, daß eine Prozeßmaschine unabhängig von der eingesetzten Umgebung und insbesondere unabhängig vom ausgeführten Prozeßmodell realisiert werden kann.

Im Rumpf der Dokumentklasse werden die Abbildungsvorschriften für die Signatur, den Leistungsumfang und das Zustandsmodell konkret formuliert.

6.3.3.1 Abbildung von Dokumenttypen auf Datenstrukturen

Um den Übergang von Dokumenttypen der Modellausführung zu Datentypen der Prozeßausführung (vgl. Kapitel 6.2) zu ermöglichen, wird zunächst jeder Dokumentklasse und damit implizit jedem Dokumenttypen eine *Default-Struktur* zugeordnet. Dabei handelt es sich um einen Datentyp, der in der Prozeßausführung bekannt ist und der die Struktur der abgelegten Dokumente dieses Dokumenttyps bestimmt. Bei der Auswahl einer Default-Struktur kommen daher nur solche Datentypen in Frage, für die ein geeignetes logisches Werkzeug vorhanden ist.

Die Benutzung weiterer logischer Werkzeuge wird durch die Angabe alternativer Datentypen (ALLOWED STRUCTURES) ermöglicht. Falls alternative Datentypen zugelassen werden, muß für jeden alternativen Datentyp eine Transformation der Default-Struktur in die Struktur des alternativen Typs sowie eine Umkehrtransforma-

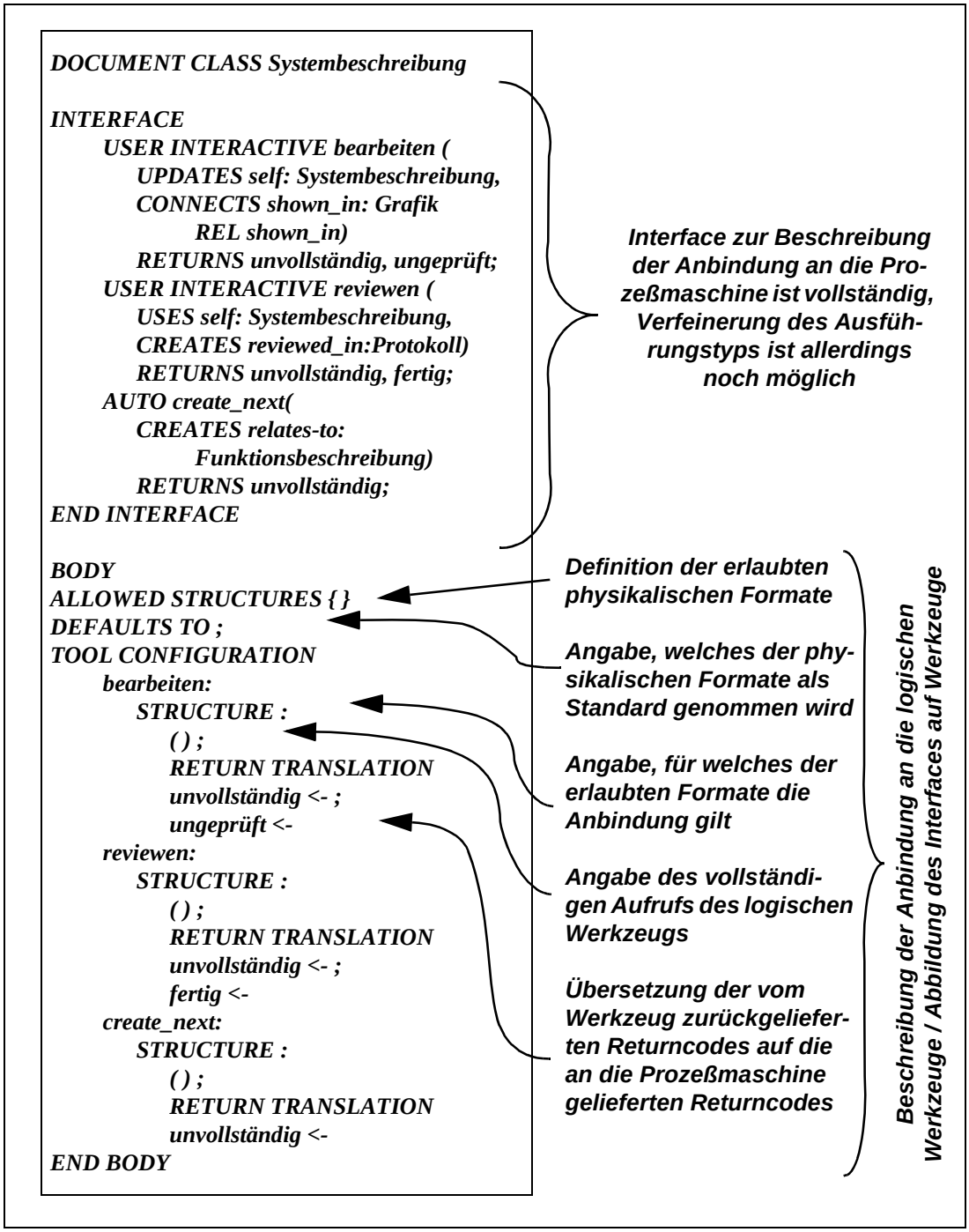


Abbildung 6.16 Im Prozeßmodell identifizierte Konzepte zur Abbildung auf logische Werkzeuge

tion spezifiziert werden. Dies wird durch die Angabe logischer Werkzeuge ermöglicht, deren Implementierungen die Transformation durchführen können. Somit wird den Transformationen jeweils ein logisches Werkzeug mit zwei Parametern zugeordnet, wobei der eine Parameter ein Dokument im Ausgangstyp und der zweite Parameter ein Dokument im Zieltyp ist.

Soll auf einem Dokument nun ein Werkzeug verwendet werden, das dieses Dokument nicht in dessen Default-Struktur erwartet, so kann die entsprechende Transformations-

möglichkeit aufgerufen werden. Ebenso wird mit einer möglichen Rückgabe des veränderten Dokuments vom logischen Werkzeug verfahren.

Für dieses Vorgehen muß sichergestellt sein, daß die Transformation eine umkehrbare Funktion ist, deren Umkehrung durch die Rücktransformation beschrieben ist. Ist diese Bedingung nicht erfüllt, so kann das Dokument nicht verlustfrei in die alternative Datenstruktur und zurück transformiert werden.

Wird keine Veränderung des Ausgangsdokuments vorgenommen (das Ausgangsdokument wird nur gelesen), so kann auf die Rücktransformation verzichtet werden. Ein Beispiel, in dem eine Rücktransformation kaum möglich wäre, aber auch nicht benötigt wird, ist die Umwandlung eines strukturierten Dokuments (d.h. kein „Plain Text“) in ein Ausgabeformat auf der Basis einer Bitmap. Die Umwandlung in ein solches Format würde z.B. für den Druck auf einem Nadeldrucker vorgenommen werden. In einem solchen Fall kann also ein alternativer Dokumenttyp zugelassen werden, für den nur eine „Hin-“Transformation existiert. Wird ein solcher Typ eingeführt, so geht das daher mit der Einschränkung einher, daß nur solche logischen Werkzeuge auf dem Ausgangsdokument aufgerufen werden können, die es in der betreffenden alternativen Datenstruktur übergeben bekommen und auf das Dokument selbst nur lesend zugreifen.

Bezogen auf die Anforderungen aus Kapitel 4 wird mit den Informationen zur Datenstruktur ein Teil der Anforderungen zur *Relation der Signaturen* erfüllt. Zur vollständigen Erfüllung wird allerdings noch die Formulierung des vollständigen Aufrufs des logischen Werkzeugs mit den Parametern der Methode benötigt.

6.3.3.2 Aufruf der logischen Werkzeuge

Um aus den Methoden der Dokumentklassen heraus logische Werkzeuge aufzurufen, muß der Spezifizierer der PSEU diesen Aufruf explizit angeben. Das bedeutet zum einen, daß er den Aufruf des jeweiligen logischen Werkzeugs mit den Parametern der dadurch implementierten Methode beschreibt und dadurch die Abbildung von Signatur und Leistungsumfang der Aktivität auf ein logisches Werkzeug vervollständigt. Das bedeutet zum anderen, daß er die Sekundärparameter des logischen Werkzeugs mit konkreten Werten zum Aufruf besetzt. So könnte der Aufruf eines logischen Werkzeugs in der Methode `reviewen` aus dem Beispiel lauten: `inspect (reviewed_in, self)`. Dabei ist `inspect` der Name eines aufgrund seiner Spezifikation geeigneten logischen Werkzeugs. Der Typ des Parameters `self` ist dabei durch die Festlegung der Datenstruktur definiert (vgl. oben). Der Typ des Parameters `reviewed_in` ergibt sich durch die in der Dokumentklasse `Protokoll` festgelegte Datenstruktur.

Mit diesen Festlegungen sind die Anforderungen zur *Relation der Signaturen* und zur *Abbildung des Leistungsumfangs* sowie die Anforderung zur *Spezifikation der prozeßunabhängigen Eingaben* vollständig abgedeckt.

6.3.3.3 Auswertung der symbolischen Rückgaben

Die symbolischen Rückgaben der logischen Werkzeuge werden auf die Zustände des Zustandsmodells der Aktivität abgebildet. Jede Methode einer Dokumentklasse der

Umgebungsspezifikation gibt an die Modellausführung ein Element eines Aufzählungstyps zurück, der aus der Menge der möglichen Zustände besteht. Mit dieser Information kann die Modellausführung aktualisiert werden. Es wird dabei vorausgesetzt, daß die Methode nach der Ausführung den Bearbeitungsstand des Dokuments kennt, auf dem sie aufgerufen wurde. Um dies sicherzustellen, können als Implementierung einer automatisch (d.h. ohne Benutzerinteraktion) von der Prozeßsteuerungskomponente aufgerufenen Methode nur solche logischen Werkzeuge benutzt werden, die Rückgaben liefern, auf deren Basis sich eindeutig der erreichte Bearbeitungsstand ablesen läßt (Anforderung zur *Übermittlung von Zustandsinformation*).

Die nötige Abbildung von symbolischen Rückgaben der logischen Werkzeuge auf Zustände des Prozeßmodells wird in einem Abschnitt des Rumpfes jeder Methode spezifiziert. Als Beispiel sei im Prozeßmodell ein Dokumenttyp `Quelltext` mit einer Aktivität `übersetzen` definiert. Die Aktivität überführt ein Dokument dieses Dokumenttyps vom Bearbeitungsstand `zu_übersetzen` in einen der Bearbeitungsstände `übersetzt`, `übersetzt_mit_Fehlern` oder `übersetzt_mit_Warnungen`. Als Implementierung wird ein logisches Werkzeug `compile` mit vier symbolischen Rückgaben (`ok`, `warnings`, `errors` und `fatal_error`) vorgesehen. Es wird daraufhin manuell vom Spezifizierenden der Umgebung eine Zuordnung der Rückgabewerte vorgenommen (vgl. Abbildung 6.17).

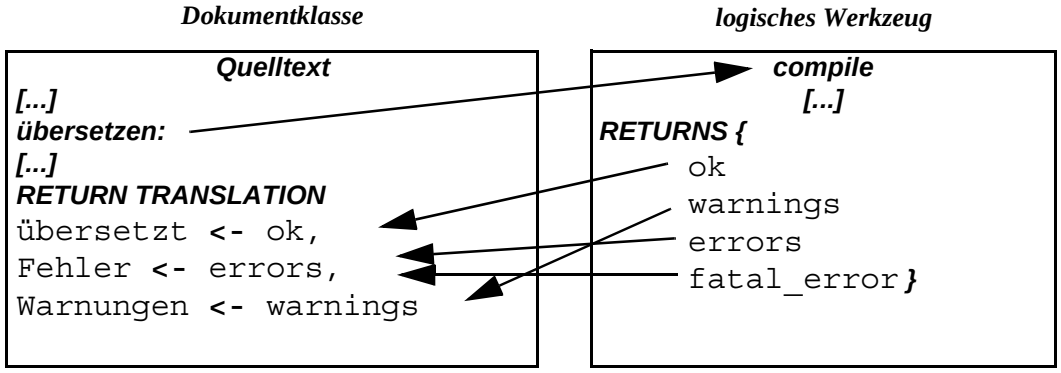


Abbildung 6.17 Vorgehen zur Spezifikation von Feedback am Ende einer Aktivität

Im Beispiel werden zwei Typen von Fehlern unterschieden, die durch die Rückgaben `errors` bzw. `fatal_error` angezeigt werden. `compile` gibt `errors` dann zurück, wenn es Übersetzungsfehler gegeben hat, aufgrund derer die Übersetzung abgebrochen wird. `fatal_error` wird im Gegensatz dazu immer dann zurückgegeben, wenn es nicht wirklich mit der Übersetzung zusammenhängende Fehler gab (z.B. „Festplatte voll“ oder „Zugriff auf ein Dokument verwehrt“).

Damit ist auch die Anforderung zur *Abbildung des Zustandsmodells* erfüllt und die Umgebungsspezifikation vollständig. Der Rumpf der Klasse `Systembeschreibung` sieht dann vollständig wie in Abbildung 6.18 gezeigt aus.

```
DOCUMENT CLASS Systembeschreibung

INTERFACE
[...]
END INTERFACE

BODY
ALLOWED STRUCTURES {UNIX-ASCII}
DEFAULTS TO UNIX-ASCII;
TOOL CONFIGURATION
    bearbeiten:
        STRUCTURE: UNIX-ASCII
            edit( self, shown_in ) ;
            RETURN TRANSLATION
            unvollständig <- ASK;
            ungeprüft <- ASK
    reviewen:
        STRUCTURE: UNIX-ASCII
            inspect( reviewed_in, self ) ;
            RETURN TRANSLATION
            unvollständig <- ASK;
            fertig <- ASK
    create_next:
        STRUCTURE: UNIX-ASCII
            create-unix-ascii( relates-to ) ;
            RETURN TRANSLATION
            unvollständig <- ok
END BODY
```

Abbildung 6.18 Vollständiger Body einer Dokumentklasse

6.3.3.4 Verfeinerung des Ausführungstyps von Aktivitäten

Bei der Abbildung auf logische Werkzeuge müssen mehr als die beiden im Prozeßmodell verwendeten Ausführungstypen von Aktivitäten (automatisch und interaktiv) unterschieden werden. Welche Ausführungstypen für Werkzeuge denkbar sind, ist in Abbildung 6.19 aufgelistet. Die Regeln R_1 und R_2 beschreiben direkte Abhängigkeiten zwischen den Spalteneinträgen.

Vier verschiedene Werkzeugtypen sind denkbar. Im Prozeßmodell, d.h. auf der Ebene von ESCAPE+, ist zunächst nur interessant, ob ein Werkzeug durch den Benutzer gestartet wird oder nicht. Der dabei verwendete Begriff interaktiver Aktivitäten meint dabei nur, daß der Aufruf des Werkzeugs manuell durch einen Benutzer erfolgen muß (Spalten „Initiierung manuell/automatisch“ in Abbildung 6.19). Zur Anbindung an die Modellausführung muß darüberhinaus noch bekannt sein, wie der nach der Ausführung der Aktivität erreichte Bearbeitungsstand an die Prozeßsteuerungskomponente weitergegeben werden soll (Spalte „Bestimmung des Bearbeitungsstands“ in Abbildung 6.19). Hierbei lassen sich bei den manuell initiierten Werkzeugen zwei Typen unterscheiden: Werkzeuge, die trotz manueller Initiierung in der Lage sind, den erreichten

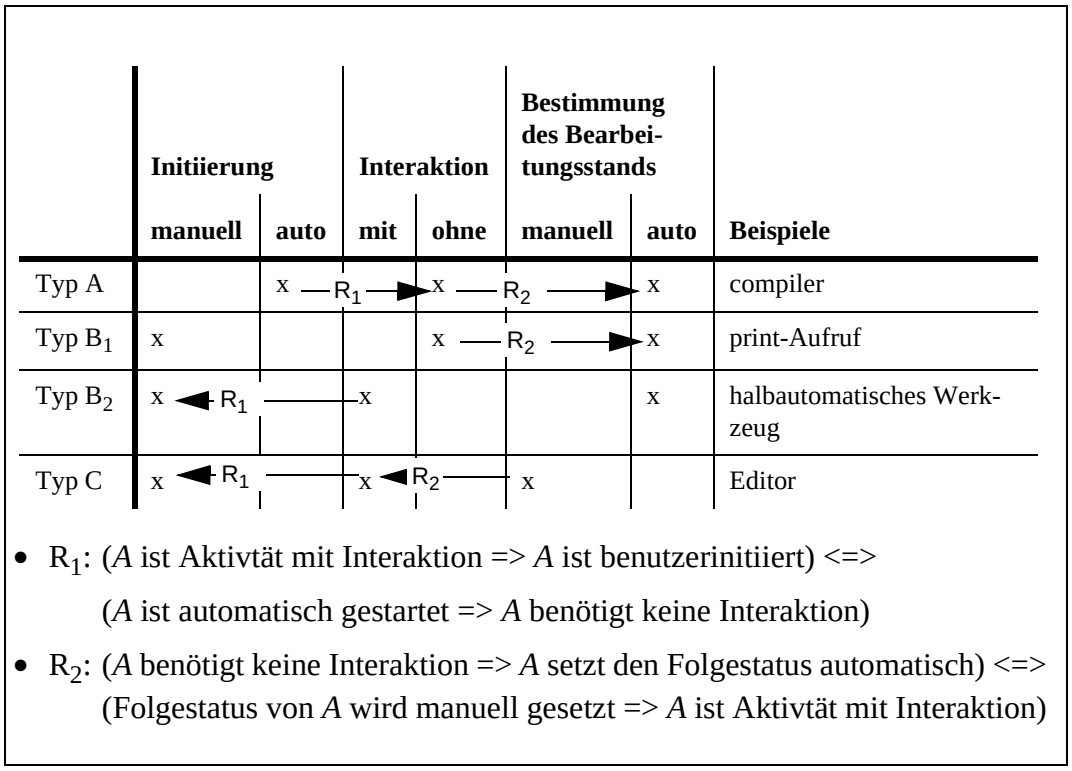


Abbildung 6.19 Ausführungstypen von Werkzeugen

Bearbeitungsstand automatisch zu bestimmen (Typen B_1 und B_2) und solche, bei denen in jedem Fall der Benutzer gefragt werden muß (Typ C).

Für die Verfeinerung der Dokumentklassen wird zunächst für jede interaktive Aktivität des ESCAPE+-Prozeßmodells angenommen, daß der erreichte Bearbeitungsstand nur beim Benutzer erfragt werden kann. Die Typen B_1 und B_2 werden also zunächst wie Typ C behandelt. Dies ist insofern eine korrekte und sichere Lösung, als im ungünstigsten Fall ein Entwickler nach dem nach der Bearbeitung aktuellen Folgezustand des bearbeiteten Dokuments gefragt würde, obwohl die Aktivität den Zustand auch automatisch bestimmt. Der umgekehrte Fall, daß der Entwickler nicht gefragt wird, obwohl es unbedingt notwendig wäre, und die daraus entstehenden Probleme, treten nicht auf. Bei Bedarf und falls möglich, kann dann bei benutzerinitiierten Aktivitäten der Ausführungstyp nachträglich angepaßt werden (vgl. Kapitel 6.3.3).

6.4 Zusammenfassung der Modellierung

In den vorangegangenen Kapiteln wurde beschrieben, welche Konzepte im Tool Integration Concept verwendet werden, um die an ein Prozeßmodell angepaßte Instanziierung einer heterogenen PSEU durchzuführen. In Kapitel 6.1.1 und Kapitel 6.2.2 wird sichergestellt, daß Signatur und Leistungsumfang sowohl für Aktivitäten wie auch für Werkzeuge spezifiziert sind (Anforderung zur *Spezifikation der Signatur*, Anforderung zur *Spezifikation des Leistungsumfangs*). Kapitel 6.1.2 und Kapitel 6.1.3 betreffen inhaltlich die Anforderung zur *Implementierung modellierter Aggregationen*, indem zunächst ein Aggregationskonzept eingeführt worden ist, um dann darauf einzugehen, wie die Anforderung erfüllt werden kann. Um die Anforderung zur *Implementierung modellierter Aggregationen* vollständig zu erfüllen, fehlt noch ein weiterer Teil, der im Kapitel 7 beschrieben wird. Die Anforderung zur *Spezifikation des Zustandsmodells*

der Aktivitäten ist nicht im Detail behandelt worden, da diese Anforderung auch schon in existierenden PML in der Regel bereits abgedeckt ist (für ESCAPE vgl. z.B. Kapitel 5.2.5.1, *Koordinationsmodell*). Die Spezifikation der Zustandsmodelle der Werkzeuge ist in Kapitel 6.2.3 bzw. 6.2.4 ergänzt worden, wodurch die Anforderung vollständig abgedeckt ist. Durch die Umgebungsspezifikation (Kapitel 6.3) wird sichergestellt, daß sich die Aktivitäten wie in Kapitel 4 gefordert auf die Werkzeuge abbilden lassen (Anforderungen zur *Relation der Signaturen*, zur *Abbildung des Leistungsumfangs*, zur *Abbildung des Zustandsmodells* und zur *Spezifikation der prozeßunabhängigen Eingaben*). Die weiteren Anforderungen aus Kapitel 4 werden nicht auf der Modellierungsebene abgedeckt. Konzepte dazu werden auf der Ebene der Implementierung von Werkzeugen und der Prozeßmaschine eingeführt. Diese Ebene wird im nächsten Kapitel betrachtet. Die durch die in diesem Kapitel abgedeckten Anforderungen werden in der Abbildung 6.20 in einer tabellarischen Übersicht gezeigt.

Konzeptebenen	*Spezifikation der Signatur*	*Relation der Signaturen*	*Spezifikation des Leistungsumfangs*	*Abbildung des Leistungsumfangs*	*Implementierung modellierter Aggregationen*	*Spezifikation des Zustandsmodells*	*Abbildung des Zustandsmodells*	*Spezifikation der prozeßunabhängigen Eingaben*	*Übermittlung von Feedback*	*Übermittlung von Zustandsinformation*	*Auswertung von Feedback*	*Kontrolle des Leistungsumfangs*	*Zustandsbestimmung nach Werkzeugabbruch*	*Zustandsbestimmung nach PE-Abbruch*	*Flexibilität der Werkzeugwahl*	*Wiederverwendbarkeit der Anbindung*	*Erkennung von Werkzeugabbrüchen*
ESCAPE+ und Umgebungs-spezifikation	+	+	+	+	O	+	+	+									

+: Anforderung wird vollständig erfüllt
-: Anforderung ist nicht betrachtet (nur in der Zeile „zusammengefaßt“)
O: Ebene liefert Beitrag zur Erfüllung der Anforderung, Anforderung nicht vollständig abgedeckt
kein Eintrag: kein Beitrag zu dieser Anforderung (nicht in der Zeile „zusammengefaßt“)

Abbildung 6.20 Abgleich mit den Anforderungen

7

TOOL INTEGRATION CONCEPT - IMPLEMENTIERUNG

Die im Kapitel 6 beschriebene TIC-Modellierung wird nun auf die ausführbare Ebene heruntergebrochen. Konkret wird also gezeigt, wie die vorgestellte Abbildung von Aktivitäten auf Werkzeuge verwendet wird, um den Aufruf einer Aktivität in der Modellausführung in einen Werkzeugaufruf in der Prozeßausführung umzusetzen. Damit werden in diesem Kapitel die Anforderungen aus Kapitel 4.2 und Kapitel 4.3 bearbeitet.

Das Kapitel gliedert sich in vier Teile. In Kapitel 7.1 wird zunächst gezeigt, wie die konkrete Anbindung von Werkzeugen an eine PSEU auf der technischen Ebene aussieht. Für die Beispielumsetzung mit Merlin wird darin außerdem dargestellt, wie Aktivitäten auf Aggregationen aufgerufen werden und ob sich dadurch Veränderungen der Anbindung ergeben.

Im Kapitel 7.2 wird beschrieben, wie die Abstimmung von Prozeßsteuerungskomponente und Werkzeugen realisiert wird.

Im Kapitel 7.3 werden Recoverymöglichkeiten für die Rekonstruktion der Daten einer PE nach Abbrüchen von Werkzeugen bzw. der PE selbst entwickelt, und es wird gezeigt, wie solche Abbrüche bemerkt werden können.

Am Schluß wird in Kapitel 7.4 die durch TIC festgelegte Rahmenarchitektur beschrieben. Es wird dann dargestellt, wie eine auf der Rahmenarchitektur basierende Struktur von Merlin aussieht.

7.1 Integrationstechnik

In diesem Kapitel wird gezeigt, wie sich die Anbindung von Werkzeugen so realisieren läßt, daß eine möglichst große Freiheit bei der Wahl der Werkzeuge gewährleistet ist (Anforderung zur *Flexibilität der Werkzeugwahl*). Um den Aufwand zur Integration von Werkzeugen zu reduzieren, sollen Werkzeuganbindungen wiederverwendbar sein, das heißt, daß die Anbindung so durchgeführt werden muß, daß sie an anderer Stelle

ohne Zusatzaufwand wieder benutzt werden kann (Anforderung zur *Wiederverwendbarkeit der Anbindung*).

7.1.1 Flexibilität der Anbindung

Mit den logischen Werkzeugen wurde eine Schnittstellenbeschreibung für Werkzeuge eingeführt. Zwei Werkzeuge, die den Bedingungen genügen, die in Kapitel 6.2 genannt sind, und sich „nur“ im konkreten Aufruf unterscheiden, sind für einen Benutzer des logischen Werkzeugs nicht unterscheidbar.

Um die Anforderung zur *Flexibilität der Werkzeugwahl* erfüllen zu können, werden die durch die logischen Werkzeuge gegebenen abstrakten Schnittstellen auf konkrete Werkzeugaufrufe abgebildet. Der Grundgedanke dazu ist das in Kapitel 5.1.2.2 erläuterte Konzept der Envelopes. Das anzubindende Werkzeug wird mit einem Envelope versehen, der dafür sorgt, daß die Verbindung aus Envelope und Werkzeug nach außen das Verhalten aufweist, das durch das logische Werkzeug vorgegeben ist.

Da die Envelopes das Element der Architektur sind, in dem die Vielfalt der Anbindungstechniken aufgefangen wird, bleibt die Erstellung eines Envelopes eine sehr spezifische, für jeden Fall im Detail neu aufgebaute Lösung. Die Erstellung von Envelopes läßt sich in diesem Sinne nur durch unterstützende Elemente wie z.B. Bibliotheken vereinfachen. Im Umfang von SoftBench, einer (nicht prozeßgesteuerten) erweiterbaren Software-Entwicklungsumgebung, ist beispielsweise eine Bibliothek enthalten, die spezielle Funktionalitäten anbietet, um UNIX-Werkzeuge mit einem textuellen Front-End oder einem cursorbasierten Front-End einzukapseln. Entsprechende Hilfestellungen lassen sich auch für eine auf TIC basierende PSEU anbieten, so daß der Aufwand zur Erstellung eines Envelopes reduziert wird.

In Abbildung 7.1 ist das Zusammenspiel der genannten Konzepte in einer Übersicht dargestellt.

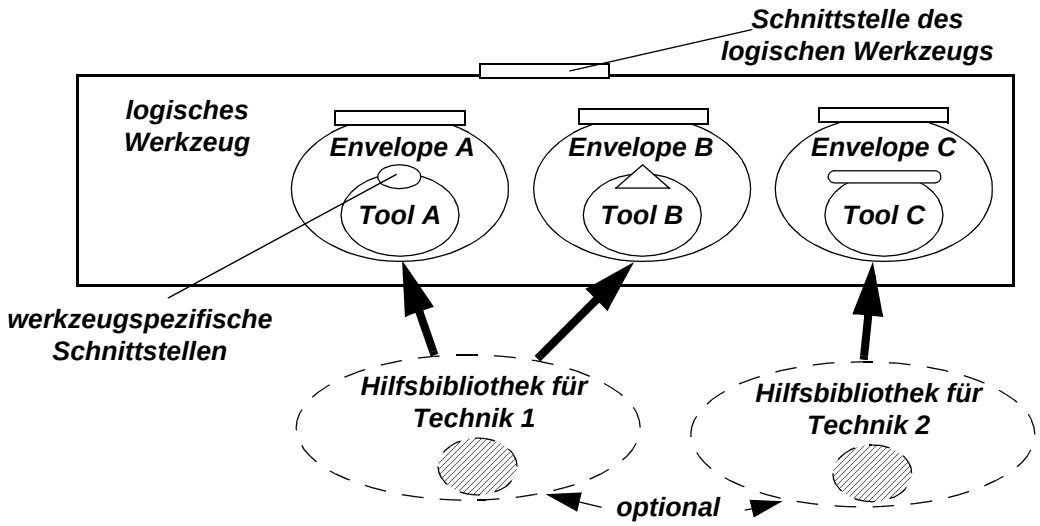


Abbildung 7.1 Gesamtzusammenhang der verschiedenen Komponenten der Kapselung

Jedes Werkzeug aus der Abbildung ist durch einen werkzeugspezifischen Envelope an die durch ein logisches Werkzeug spezifizierte Schnittstelle angepaßt. In der Abbildung sind die drei Werkzeuge verschiedene Implementierungen des gleichen logischen Werkzeugs. Zwei der Werkzeuge folgen einer gemeinsamen Anbindungstechnik. Es

sind beispielsweise UNIX-Kommandozeilen-Werkzeuge mit einer einfachen Ausgabe auf die Standardausgabe. Das dritte Werkzeug ist anders aufgebaut. Es ist beispielsweise ein UNIX-Werkzeug, das ToolTalk-Nachrichten empfängt und versendet.

Für beide Techniken (Kommandozeilen, ToolTalk) gibt es im Beispiel Hilfsbibliotheken, die von den Envelopes benutzt werden. Da letztlich alle Envelopes auf der Basis einer gemeinsamen Technik die Schnittstelle implementieren, die durch das logische Werkzeug zur Benutzung in einer TIC-basierten PSEU definiert wird, können alle drei Envelopes als Implementierung des logischen Werkzeugs eingesetzt und gegeneinander ausgetauscht werden.

In Abbildung 7.2 ist ein Auszug aus einem Beispiel-Envelope zu sehen, der unter Linux den Aufruf eines Editors, hier des vi, kapselt (Methode StartEdit). Der Editor ist ein Black-Box-Werkzeug, und die Implementierung des Envelopes ist ohne den Einsatz von Bibliotheken vorgenommen worden. Die Implementierung erfolgte mit C++ unter Nutzung von ILU, einer CORBA ähnlichen Middleware-Implementierung (zu den Unterschieden zu CORBA vgl. z.B. [ILU00]).

Die Methode StartEdit ist im Beispiel im Detail zu sehen. Sie bekommt durch das aufrufende Anbindungsobjekt zwei Variablen übergeben, von denen die erste den Verbindungsstatus zurückmeldet und die zweite mit der Dokumenten-Id eine Referenz auf das zu bearbeitende Dokument übergibt. Mit der Dokumenten-Id wird eine Verbindung zum ILU-Server, über den Nachrichten an den Aufrufer zurückgeschickt werden, aufgebaut. Unter Ausnutzung von UNIX-üblichen Mechanismen wird nun der vi-Prozeß gestartet und überwacht. Falls das Starten des vi geklappt hat, wird an das aufrufende Anbindungsobjekt eine Bestätigungsmeldung zurückgeschickt, die Teil des Protokolls ist.

Der Aufwand zur Erstellung von Envelopes wird in TIC dadurch relativiert, daß die Envelopes von der Anbindung an eine PSEU getrennt werden. Dadurch tritt die Erstellung des Envelopes genau einmal auf, während die Erstellung der Umgebungsspezifikation ein immer wiederkehrender Aufwand ist.

Die Aufgaben der Envelopes bei der Anbindung von Werkzeugen sind

- Abstraktion von Werkzeugdetails und
- Realisierung einer einem logischen Werkzeug entsprechenden Schnittstelle.

Wie diese Aufgaben realisiert werden, wird im folgenden näher beschrieben.

7.1.1.1 Abstraktion von Werkzeugdetails

Genügen zwei Werkzeuge dem gleichen logischen Werkzeug, so sind Werkzeugdetails die Teile, in denen sich die Werkzeuge unterscheiden. Beispiele für Werkzeugdetails sind die Reihenfolge der übergebenen Parameter, die Verwendung verschiedener Rückgabewerte für dieselbe Information etc.

Die Realisierung dieser werkzeugunabhängigen Parameterübergabe erfolgt durch eine Abbildung der Aufrufparameter des logischen Werkzeugs auf die Parameter des Werkzeugs. Dies gilt nicht nur für den Werkzeugaufruf, sondern auch für die sonstige zur Laufzeit des Werkzeugs durchgeführte Kommunikation zwischen Werkzeug und

```
// verschiedene Includes

// Prototypen
typedef void (*sighandler_t)(int);
void handle_user_stop(int param);

// globale Variable zur Speicherung der Dokument-Id
edit_T_DOCUMENT glob_docid;

// methoden
void EDIT_derived::StartEdit(editStatus *_status, edit_T_DOCUMENT docid) {
        sighandler_t handlerInstalled;

        DOCUMENT_TYPEStatus *status = new DOCUMENT_TYPEStatus;
        DOCUMENT_TYPE_T_DOCUMENT_TYPE* handle;
        glob_docid = docid;

        handle = (DOCUMENT_TYPE_T_DOCUMENT_TYPE *) iluObject::Lookup(„MyServer“,
                        docid, DOCUMENT_TYPE_T_DOCUMENT_TYPE::ILUClassRecord);
        if (handle == NULL) {
                cerr << „Unable to import server!\n“; exit(1); }

        // Eine Kopie des Prozesses als Kindprozess anlegen und die Prozess-ID
        // merken
        child_pid = fork();
        if (child_pid == 0) {
                // Dieser Prozess ist der Kindprozess. Durch das Werkzeug ersetzen!
                // Die Prozess-ID bleibt dabei gleich, so dass der Vaterprozess des
                // Werkzeugs den Prozess ueberwachen kann.
                execl(„/usr/bin/X11/xterm“, „xterm“,“-e“, „/usr/bin/vi“,docid,NULL); }
        else if (child_pid < 0) {
                // Ich bin der Vaterprozess und es ist beim Starten etwas schief gelaufen!
                cerr << „Es hat einen Systemfehler beim Starten des Werkzeugs gegeben\n“;
                // HIER MUSS DIE RUECKMELDUNG AN DAS DOKUMENTOBJEKT HIN!
                // z.B. dokument.StoppedEdit(...) }
        else {
                // Ich bin der Vaterprozess
                // Rueckmeldung an das Dokumentobjekt, dass das Werkzeug gestartet wurde
                handlerInstalled = signal(SIGCHLD, handle_user_stop);
                if (handlerInstalled == SIG_ERR) {
                        cerr << „Problems with SIGCHLD-handler!\n“; }
                status->returnCode = DOCUMENT_TYPEReply_Success;
                cerr << „Envelope Edit has started vi on „ << docid << „!\n“;
                handle->Started(status, „edit“, docid) ; }
        }
void EDIT_derived::StopEdit(editStatus *_status, edit_T_DOCUMENT docid) {
        // Hier können in einem nicht Black-Box-Envelope noch
        // Änderungsmeldungen abgesetzt werden
        // beende das Werkzeug
        kill(child_pid, SIGQUIT);
        cerr << „Werkzeug beendet\n“; }

// Weitere Methoden, z.B. im Falle eines White-Box-Werkzeugs Methoden
// zur Durchführung des Control-Protokolls
```

Abbildung 7.2 Beispiel für Envelope um vi unter Linux

Anbindungsobjekt und umgekehrt. An dieser Stelle können auch Hilfsbibliotheken angewendet werden, wie sie oben im Zusammenhang mit SoftBench beschrieben wurden.

Welche Konzepte der logischen Werkzeuge auf welche der physikalischen Schnittstelle abgebildet werden müssen und welche Hilfen für die daraus resultierenden Envelopes angeboten werden können, muß für jeden Typ von Anbindungstechnik einzeln festgelegt werden. Ist dies einmal erfolgt, so wird die Erstellung eines weiteren Envelopes im Aufwand deutlich reduziert. Der Aufwand, eine neue PSEU zusammenzustellen, verringert sich, je mehr Hilfsbibliotheken entstehen.

7.1.1.2 Realisierung einer geeigneten Schnittstelle

Die Bereitstellung einer für die Integration geeigneten Schnittstelle ist die gemeinsame Idee aller Ansätze, die Envelopes benutzen (vgl. die entsprechenden PSEU in Kapitel 5.2). Durch einen Envelope wird eine nicht passende Schnittstelle eines Werkzeugs so umgesetzt, daß das Werkzeug mit anderen Werkzeugen integriert werden kann. In TIC ist „integriert werden können“ dadurch definiert, daß ein Envelope die Schnittstelle eines logischen Werkzeugs implementiert und die Benutzung einer vorgegebenen Anbindungstechnik und eines der darauf vorgesehenen Protokolle (vgl. Kapitel 7.2.1.2) festlegt. Im Unterschied zu den Ansätzen aus Kapitel 5.2 wird diese Anpassung nicht in jeder Umgebung neu durchgeführt. Stattdessen wird einmal ein logisches Werkzeug erstellt, das eine in allen relevanten Umgebungen (= TIC-basierte PSEU) benutzbare Schnittstelle bereitstellt. Dadurch sind Werkzeuge, die durch ein gemeinsames logisches Werkzeug repräsentiert werden, nicht mehr unterscheidbar und damit austauschbar.

Die Anforderung zur *Flexibilität der Werkzeugwahl* kann durch die oben beschriebenen Maßnahmen erfüllt werden, da durch die Verwendung von Envelopes das größtmögliche Maß von Flexibilität erzielt wird. Die Möglichkeiten sind insbesondere deshalb ausreichend, da durch dieses Verfahren zusammen mit den Konzepten aus Kapitel 7.2.1 Black-Box-Werkzeuge in eine PSEU integriert werden können.

7.1.2 Wiederverwendbarkeit der Anbindung

Bei der Einführung von Envelopes wurde noch nicht ausgeführt, welche Maßnahmen nötig sind, um sicherzustellen, daß die möglicherweise aufwendig zu erstellenden Envelopes beim Wechsel in eine andere Umgebung nicht umgeschrieben werden müssen. Solche Konzepte sind aber notwendig, um die Anforderung zur *Wiederverwendbarkeit der Anbindung* erfüllen zu können.

In einem ersten Schritt wird festgelegt, die Aufrufe von Werkzeugen nicht aus Kernkomponenten der verwendeten Prozeßsteuerungskomponente heraus auszuführen. Die Konfiguration der Werkzeuge, d.h. die Information zur Anbindung der Aktivitäten an die Werkzeuge, wird ausschließlich in den Instanzen der Dokumentklassen verwaltet. Die Instanzen der Dokumentklassen werden als *Anbindungsobjekte* bezeichnet. Die Aufgabe der Prozeßmaschine beim Aufruf von Werkzeugen wird durch diese Maßnahme darauf reduziert, die Methoden der Dokumentklassen, die den Aktivitäten entsprechen, anzusprechen. Der Hauptvorteil dieses Vorgehens ist die Entkopplung der Aufruftechnik von der Prozeßsteuerungskomponente.

Dieses Vorgehen ist mit der Verwendung eines Treibers bei der Anbindung von Peripherie an einen Rechner vergleichbar. Dabei ruft das Betriebssystem unabhängig von der tatsächlich angebundenen Komponente eine vermittelnde Komponente, den Treiber, auf. Dieser übernimmt dann die Umsetzung der Anbindung auf der physikalischen Ebene. Solange die Schnittstelle zwischen Treiber und Betriebssystem konstant bleibt, können verschiedene Peripheriebausteine durch einfachen Austausch der Treiber angebunden werden.

In der Beispielumgebung Merlin können in der existierenden Architektur genau solche Werkzeuge aufgerufen werden, für die speziell ein Aufrufmechanismus (Anbindungstechnik und zugehöriges Protokoll) implementiert ist. Der Einsatz eines Werkzeugs, das einen neuen Aufrufmechanismus erfordert, würde die Änderung der Implementierung der aufrufenden Merlinkomponenten (PE, WC) nötig machen. Nach den oben definierten Regeln wird dem Prozeßmodell entnommen, welcher Dokumenttyp welche Aktivitäten besitzt, welche Methoden welcher Dokumentklasse entsprechen und welche Parameter die Methode beim Aufruf benötigt. Daraus wird der an das jeweilige Anbindungsobjekt abgesandte Aufruf erzeugt, der sich auf einem gemeinsam definierten Protokoll abstützt. Die eigentliche Anbindung des Werkzeugs erfolgt durch Umsetzung des Aufrufs auf der Basis der Abbildungsvorschriften aus der Umgebungsspezifikation in den Anbindungsobjekten und den Envelopes.

Der Algorithmus zum Aufruf der Aktivitäten ist in der beschriebenen Architektur in der Prozeßmaschine „hart verdrahtet“. Er nutzt die im Modell eingebrachten Zugriffspfade aus, um die Parameter für die Aufrufe der Methoden der Anbindungsobjekte zu bestimmen. Ein Zugriffspfad stellt eine Navigationsanweisung durch die zwischen den Dokumenten aufgebauten Assoziationen dar. Daher können zur Laufzeit eines Projekts die Assoziationen, die konkret von einem Dokument ausgehen, benutzt werden, um die Parametrisierung einer Aktivität durchzuführen. Die Prozeßmaschine unterscheidet dabei zwischen den verschiedenen Zugriffstypen, um zu erkennen, ob möglicherweise weitere Aktionen notwendig werden. USES- und UPDATES-Zugriffe werden ausschließlich zur Bestimmung der Dokumente benutzt, die von der Aktivität betroffen sind. Weitere Aktionen müssen nicht durchgeführt werden. Bei einem DELETES-Zugriff wird zusätzlich die verbindende Beziehung zum zu löschenden Dokument entfernt. Entsprechend muß bei einer Aktivität mit CREATES-Zugriff die notwendige Beziehung eingefügt werden. DISCONNECTS und CONNECTS Zugriffe werden ebenso ausgewertet, d.h. daß z.B. eine Beziehung, die durch einen CONNECTS-Zugriff beschrieben wird, nach der Beendigung der Aktivität auf der Modellebene eingefügt wird.

In den Dokumentklassen bzw. zur Laufzeit in den Anbindungsobjekten muß die Umsetzung der Aufrufe der abstrakten Aktivitäten auf die konkreten Werkzeuge erfolgen (vgl. Kapitel 7.1).

In Abbildung 7.3 wird der Zusammenhang der zu erstellenden Schnittstelle mit den Envelopes zunächst ohne die Ebene der Anbindungsobjekte dargestellt. Eine ähnliche Realisierung findet sich in Marvel (vgl. [VK95]). Hier müssen mit jedem Umgebungswechsel alle Envelopes erneuert werden (vgl. auch Kapitel 5.2.2.3), was bei häufigem Wechsel von Umgebungen schnell sehr aufwendig wird.

In TIC wird dieser Aufwand dadurch reduziert, daß ein Envelope für ein Werkzeug nur ein einziges Mal geschrieben wird. Um dies tun zu können, werden die Dokumentklas-

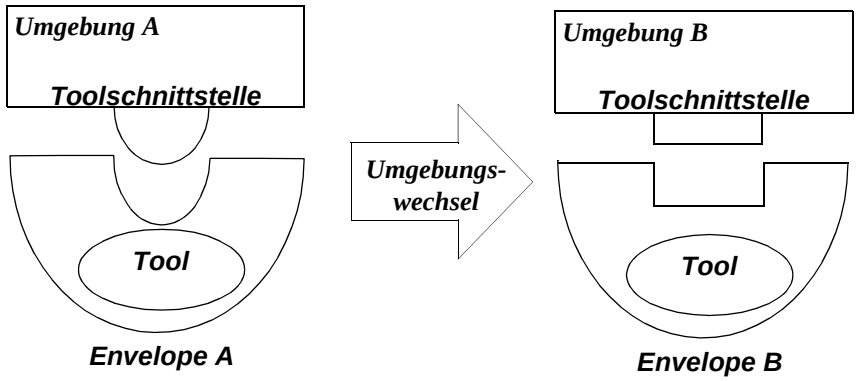


Abbildung 7.3 Werkzeugintegration in verschiedenen Umgebungen ohne TIC

sen aus der Umgebungsspezifikation und dem Prozeßmodell derart generiert, daß die Nachrichten der Prozeßmaschine in Nachrichten an die Werkzeuge umgesetzt werden. Zur Laufzeit übernehmen dann die Instanzen der Dokumentklassen die Rolle eines Dokuments, auf dem Aktivitäten aufgerufen werden. Typischerweise werden daher die Dokumentklassen in einer objektorientierten Programmiersprache vorliegen. Der manuell für die Definition der Dokumentklassen zu leistende Aufwand (Erstellen der Umgebungsspezifikation, vgl. Kapitel 6.3.3) ist marginal im Verhältnis zum Erstellungsaufwand für einen neuen Envelope. Liegen bereits Werkzeuge mit passendem Envelope als wiederverwendbare Komponenten vor, so sinkt der Gesamtaufwand zur Erstellung einer prozeßspezifischen PSEU. Dieses Prinzip ist in Abbildung 7.4 dargestellt.

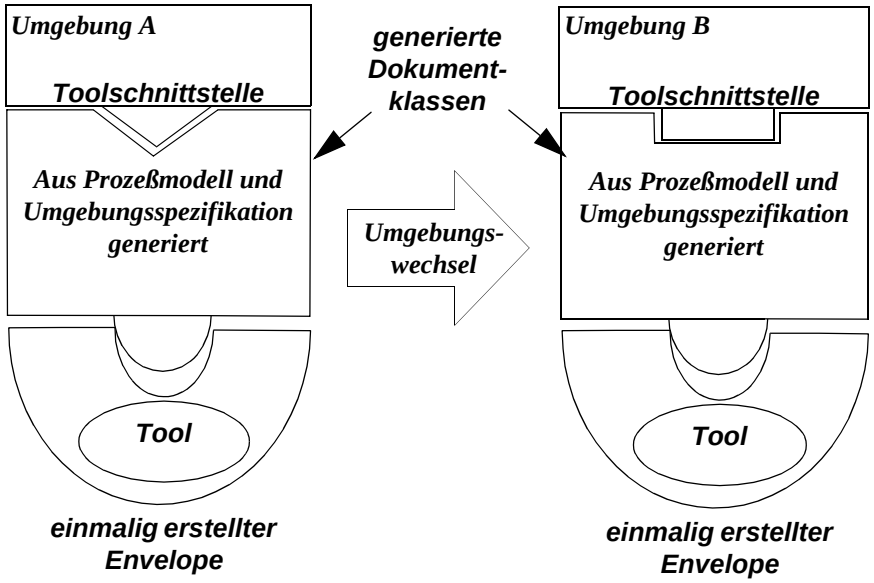


Abbildung 7.4 Werkzeugintegration in verschiedenen Umgebungen mit TIC

Die oben geforderte Eingrenzung des Aufwands wird also durch die Generierung der Dokumentklassen aus der Umgebungsspezifikation und dem Prozeßmodell, für das eine PSEU erstellt werden soll, erreicht.

Damit ist die Anforderung zur *Wiederverwendbarkeit der Anbindung* erfüllt.

Die Betrachtung der Integrationstechnik ist damit abgeschlossen. Es wurde insbesondere dargestellt, wie Aktivitäten aufgerufen werden und wie der Aufruf einer Aktivität auf den eines Werkzeugs abgebildet wird. Noch nicht behandelt wurde der Aufruf von Aktivitäten auf Aggregationen. Dies wird im nächsten Kapitel gesondert betrachtet.

7.1.3 Aggregationen

Für Aktivitäten, die auf Aggregationen aufgerufen werden, muß sichergestellt sein, daß sie die Abhängigkeiten beachten, die für die Ausführung einer Aktivität auf einem Teil der Aggregation (Aggregat oder Komponente) bestehen. Ist dies gewährleistet, so ergeben sich keine Anforderungen an die Anbindung der die Aktivitäten implementierenden Werkzeuge über die bei den „normalen“ Aktivitäten hinaus. In Teilen der vorgestellten Beispielumgebungen waren Aggregationskonzepte bereits vorgesehen, und die im obigen Sinne korrekte Behandlung von Aktivitäten ist bereits sichergestellt. Welche Auswirkungen das Fehlen eines Aggregationskonzepts im Zuge des Aufbaus einer auf TIC basierenden Umgebung hat, kann am Beispiel von Merlin gut gezeigt werden.

Für die Anforderung zur *Implementierung modellierter Aggregationen* müssen auch in Merlin Konzepte zur Bearbeitung von Aggregationen ergänzt werden. Insbesondere muß (neben weiteren Komponenten) die Prozeßsteuerung geeignet erweitert werden. Dies betrifft die Implementierung der auf Aggregationen definierten Aktivitäten bzw. die Implementierung der Fortpflanzung von Aktivitäten auf Aggregationen.

Die Semantik der Aktivitäten auf Aggregationen sieht vor, fortgepflanzte Aktivitäten nichtdeterministisch auszuführen (Kapitel 6.1.3). Die Erweiterung der PE wird in einer Abwandlung so vorgenommen, daß bei der Ausführung von Aktivitäten von der Merlin-Prozeßmaschine die Menge der potentiell parallel auszuführenden Aktivitäten sequentiell abgearbeitet wird. Bei der Sequentialisierung von Aktivitäten auf einem Aggregat muß bereits von der Prozeßsteuerungskomponente geprüft worden sein, daß der Folgezustand der in der Aggregation liegenden Dokumente determiniert ist. Das bedeutet, daß keiner der Folgezustände eines der betroffenen Dokumente durch die Reihenfolge der sequentialisierten Einzelausführungen der Aktivität auf den Komponenten beeinflußt wird. Dies kann z.B. dadurch sichergestellt werden, daß kontrolliert wird, daß keine Aktivität auf einer der Komponenten eine andere Komponente in ihrem Bearbeitungsstand beeinflußt.

Durch diese Bedingung kann aber nicht sichergestellt werden, daß sich das Gesamtsystem von Dokumenten deterministisch verhält. Es könnten nämlich Aktivitäten zweier Komponenten auf ein anderes Dokument Auswirkungen haben, wobei der resultierende Bearbeitungsstand des dritten Dokuments nicht eindeutig ist. Um diese Situation zu verhindern, muß sichergestellt werden, daß die Ausführung zweier fortgepflanzter Aktivitäten nicht auf das gleiche Dokument Auswirkungen hat oder aber in genau einen resultierenden Bearbeitungsstand führt.

Werden diese Bedingungen so in der Prozeßsteuerungskomponente realisiert, daß die Aktivitäten auf einer Aggregation überprüft werden, so können die diese Aktivitäten implementierenden Werkzeuge wie andere Werkzeuge auch eingebunden werden. Mit den Modellierungskonzepten aus Kapitel 6.1.2 und der Bereitstellung der entsprechen-

den Ausführungskonzepte auf der Ebene der Prozeßsteuerung ist auch die Anforderung zur *Implementierung modellierter Aggregationen* erfüllt.

7.2 Abstimmung zwischen PE und Werkzeugen

Nachdem auf der Ebene der Integrationstechnik die Anbindung von Werkzeugen gewährleistet ist, wird nun die inhaltliche Zusammenarbeit der Werkzeuge mit der Prozeßsteuerungskomponente angesprochen. Die dazu eingeführten Konzepte betreffen die Anforderungen zur *Übermittlung von Feedback*, zur *Übermittlung von Zustandsinformation*, zur *Auswertung von Feedback* und zur *Kontrolle des Leistungsumfangs*.

7.2.1 Übermittlung von Feedback

Aus der Anforderung zur *Übermittlung von Feedback* ergeben sich zwei Fragestellungen: 1. Wie kann sichergestellt werden, daß die Prozeßsteuerungskomponente unabhängig vom Werkzeug über Änderungen informiert wird, und 2. wie wird die Änderungsinformation ausgetauscht? Um diese Fragen beantworten zu können, müssen zunächst die Werkzeuge danach unterschieden werden, welche Fähigkeiten zum Übermitteln von Feedback sie mitbringen. Jedem der identifizierten Typen wird dann jeweils ein angepaßtes Protokoll zur Übermittlung von Feedback zwischen den Werkzeugen und den Anbindungsobjekten zugeordnet.

7.2.1.1 Werkzeugtypen

Unter dem Aspekt der Möglichkeiten zur Kommunikation eines Werkzeugs mit einer PE lassen sich die folgenden drei Werkzeugtypen unterscheiden:

- *Black-Box-Werkzeuge*
- *Grey-Box-Werkzeuge*
- *White-Box-Werkzeuge*

Black-Box-Werkzeuge geben kein Feedback über Änderungen, die von ihnen an bearbeiteten Daten vorgenommen werden. Außerdem können sie auch über Veränderungen ihrer Umwelt (in diesem Kontext also der Modellausführung) nicht informiert werden, da sie keine Möglichkeiten zum Empfang solcher Informationen anbieten. Ohne Hilfen erfüllen Black-Box-Werkzeuge Anforderungen wie die Anforderung zur *Übermittlung von Feedback* oder die Anforderung zur *Übermittlung von Zustandsinformation* damit nicht. Das Problem bei den Werkzeugen dieses Typs besteht darin, durch diesen Informationsmangel hervorgerufene Inkonsistenzen zwischen Modell- und Prozeßausführung zu erkennen und daraufhin zu beseitigen oder bei Bedarf auch als Notiz an den betroffenen Dokumenten in der Modellausführung zu vermerken.

White-Box-Werkzeuge sind das Gegenteil der Black-Box-Werkzeuge. Sie leiten alle Informationen weiter, die dazu benötigt werden, um die Prozeßsteuerung zu jedem Zeitpunkt über Änderungen auf dem laufenden zu halten. Außerdem können sie Informationen empfangen, die Veränderungen der Modellausführung betreffen und die die Ausführung des White-Box-Werkzeugs beeinflussen, und sie können auf diese Informationen reagieren.

Grey-Box-Werkzeuge umfassen die gesamte Palette der Werkzeuge, die zwischen White-Box-Werkzeugen und Black-Box-Werkzeugen liegen. Ein Beispiel sind Werkzeuge, die Änderungen, die im Verlauf der Werkzeugausführung durchgeführt werden, sammeln und am Ende der Laufzeit des Werkzeugs geschlossen an die Prozeßsteuerung weitergeben.

Die drei Werkzeugtypen können nicht scharf voneinander getrennt werden. Ein Beispiel dafür ist ein White-Box-Werkzeug, das auch die Möglichkeit bietet, die Abstimmung von Änderungen kontrolliert zu reduzieren. Damit wird eine lokale Sicht des Entwicklers unterstützt, die unnötige Aktualisierungen der Modellausführung vermeidet. In dieser lokalen Sicht kann der Entwickler solange Änderungen vornehmen und auch widerrufen, bis die vorgenommenen Änderungen aus seiner Sicht stabil sind. Die Änderungen werden gesammelt an die Prozeßsteuerungskomponente weitergegeben und mit ihr abgestimmt (vgl. z.B. [ES94]). Das Werkzeug ist somit in der Lage, die Modellausführung jederzeit mit der Entwicklungstätigkeit zu aktualisieren, kann aber auch so konfiguriert werden, daß nicht bei jeder Änderung sondern nur zu bestimmten Zeitpunkten eine Aktualisierung vorgenommen wird. Die Prozeßmaschine wird auch bei diesem Vorgehen baldmöglichst über alle prozeßrelevanten Änderungen informiert, da die Änderungen, sobald sie stabil sind, weitergegeben werden. Falls ein Entwickler die Weitergabe herauszögert, verhält sich das Werkzeug wie ein Grey-Box-Werkzeug. Die Einstufung in White-Box- oder Grey-Box-Werkzeug hängt damit vom einzelnen Einsatz und nicht generell vom Werkzeug ab.

7.2.1.2 Unterstützung und Protokoll für Feedback

Welcher der drei Typen vorliegt, wird dynamisch, anhand der vom Werkzeug übermittelten Nachrichten entschieden. Wird vom Werkzeug die Nachricht übermittelt, daß das Werkzeug gestartet ist, so wird eine Beschreibung des Werkzeugs mitgeliefert. Einem Envelope, der ein Werkzeug kapselt, wird mitgegeben, inwieweit das Werkzeug in der Lage ist, a) Änderungen mitzuteilen, b) auf Nachrichten zu reagieren und c) sich an die durch das logische Werkzeug vorgegebenen Einschränkungen zu halten.

In Abbildung 7.5 wird dargestellt, wie das Protokoll aussieht, wenn aus den Anbindungsobjekten heraus Black-Box-Werkzeuge aufgerufen werden. Wurde ein Black-Box-Werkzeug gestartet, so ist bei dessen Beendigung nicht klar, ob Änderungen durchgeführt worden sind. Dieser Mangel an Information wird durch das Anbindungsobjekt dadurch ausgeglichen, daß als minimale Unterstützung zur Konsistenzsicherung das Anbindungsobjekt in einem Dialog beim Entwickler nachfragt, welche Änderungen tatsächlich durchgeführt worden sind. In der Abbildung wird der Aufruf des Dialogs durch die Nachricht `Query_Activities` repräsentiert. Die erfragten Änderungen werden durch die Nachricht `Sub_Activities` zurückgegeben. Eine Beispielimplementierung eines entsprechenden Dialogs ist in [Kur97] angegeben.

Das Argument, daß dieses Vorgehen als Nachbearbeitung für den einzelnen Entwickler aufwendig ist, stimmt für sich genommen. Allerdings ist dieser Aufwand notwendig, um korrekte Daten zur Prozeßsteuerung in der Modellausführung zu erhalten. Die automatische Nachfrage durch das Anbindungsobjekt stellt für ein Black-Box-Werkzeug sicher, daß auf jeden Fall an die Korrektur gedacht wird. Der Aufwand ist für den Entwickler nur dadurch zu vermeiden, daß zusätzliche Analysewerkzeuge benutzt werden, um die durchgeführten Änderungen automatisch bestimmen zu können. Diese

können dann in einem Envelope direkt nachgeschaltet werden, wodurch das Black-Box-Werkzeug zu einem Grey-Box-Werkzeug transformiert wird. Alternativ können ansonsten nur intelligente Werkzeuge benutzt werden, die die gleiche Information schon zur Laufzeit des Werkzeugs erzeugen, d.h. also Grey-Box- oder White-Box-Werkzeuge. Aus dem Anbindungsobjekt heraus wird dann das Feedback erzeugt, das in der Anforderung zur *Übermittlung von Feedback* gefordert ist.

Im Änderungsdialog können auch Hilfen gegeben werden, die ohne den Bezug zur ändernden Aktivität verloren gehen würden. So kann z.B. angezeigt werden, welche Dokumente und Beziehungen vor dem Aufruf der Aktivität bestanden, um dem Entwickler schneller sichtbar zu machen, welche Komponenten des Dokuments von ihm bearbeitet wurden. Weiterhin kann dem Entwickler sofort mitgeteilt werden, wenn er Daten geändert hat, die er nach den Vorgaben des Prozeßmodells nicht hätte ändern dürfen. Zu diesem Zweck wird in der Schnittstelle der Prozeßsteuerungskomponente zwischen einer *Änderungsanfrage* und einer *Änderungsmeldung* unterschieden.

Änderungsanfrage: Eine Änderungsanfrage ist eine Nachricht an die PE, die eine gewünschte Datenänderung durch das anfragende Werkzeug beschreibt. Es wird eine Antwort von der Prozeßsteuerungskomponente erwartet, aufgrund derer die Änderung tatsächlich durchgeführt oder verworfen wird.

Änderungsmeldung: Eine Änderungsmeldung ist eine Nachricht an die PE, die feststellt, daß in der Prozeßausführung eine Änderung durchgeführt wurde. Es wird keine Antwort erwartet. Die PE hat keine direkte Einflußmöglichkeit.

Die Veränderungen, die durch ein Black-Box-Werkzeug durchgeführt werden, werden durch die Anbindungsobjekte immer in Änderungsmeldungen weitergegeben. Weitergehend wird das Protokoll zur Anbindung von Black-Box-Werkzeugen in [Kur97] unter dem Namen *Atomic-Modell* behandelt.

Für die Prozeßsteuerungskomponente hat dieses Vorgehen den Effekt, daß sie keine speziellen Maßnahmen für die Verwaltung von Dokumenten, die durch ein Black-Box-Werkzeug bearbeitet wurden, ergreifen muß. Das jeweilige Anbindungsobjekt gibt die eingegebenen Änderungen als Änderungsmeldung weiter. Die Prozeßsteuerungskomponente verhält sich genau so, als ob sie die Ergebnisse eines Grey-Box-Werkzeugs mitgeteilt bekäme.

In Abbildung 7.6 wird ein weiteres Protokoll gezeigt, das bei der Anbindung von Grey-Box-Werkzeugen Anwendung findet (vgl. [Kur97], Protokoll zum *Observer-Modell*). Im Gegensatz zum Protokoll für Black-Box-Werkzeuge werden Informationen über die Aktionen des Anwenders schon zur Laufzeit des Werkzeugs weitergegeben (siehe Nachricht `Subactivity_Notification` in der Abbildung 7.6). Es werden allerdings keine Reaktionen darauf angenommen. Dadurch ist auf der Seite der Modellausführung nicht unterscheidbar, ob es sich um ein Grey-Box- oder ein Black-Box-Werkzeug handelt.

Für White-Box-Werkzeuge kann das Protokoll für Grey-Box-Werkzeuge noch verfeinert werden (vgl. [Kur97], Protokoll zum *Control-Modell*). Die Verfeinerung besteht darin, dem Werkzeug direkt nach der Mitteilung einer Aktion ein Feedback zu geben. Das Werkzeug wartet dieses ab, um in jedem Falle prozeßkonform zu bleiben. Dieses Protokoll ist in Abbildung 7.7 dargestellt. Bei der Anfrage des Werkzeugs wird im

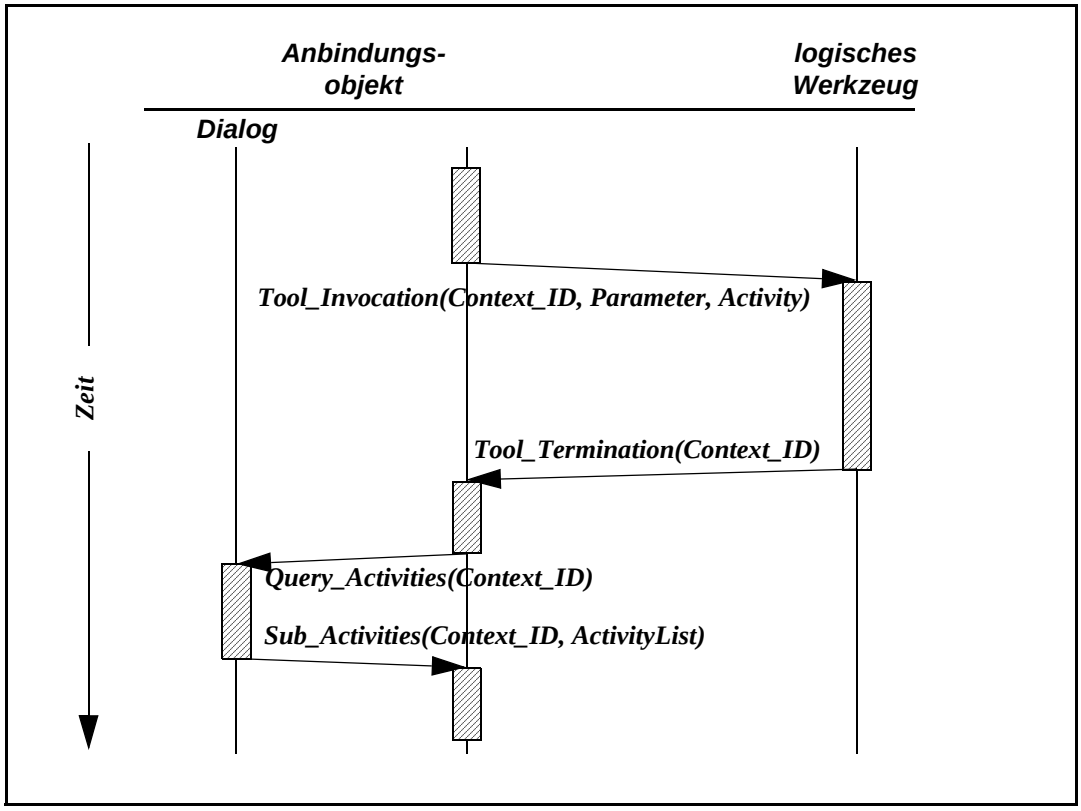


Abbildung 7.5 Schematische Darstellung der Anbindung von Black-Box-Werkzeugen

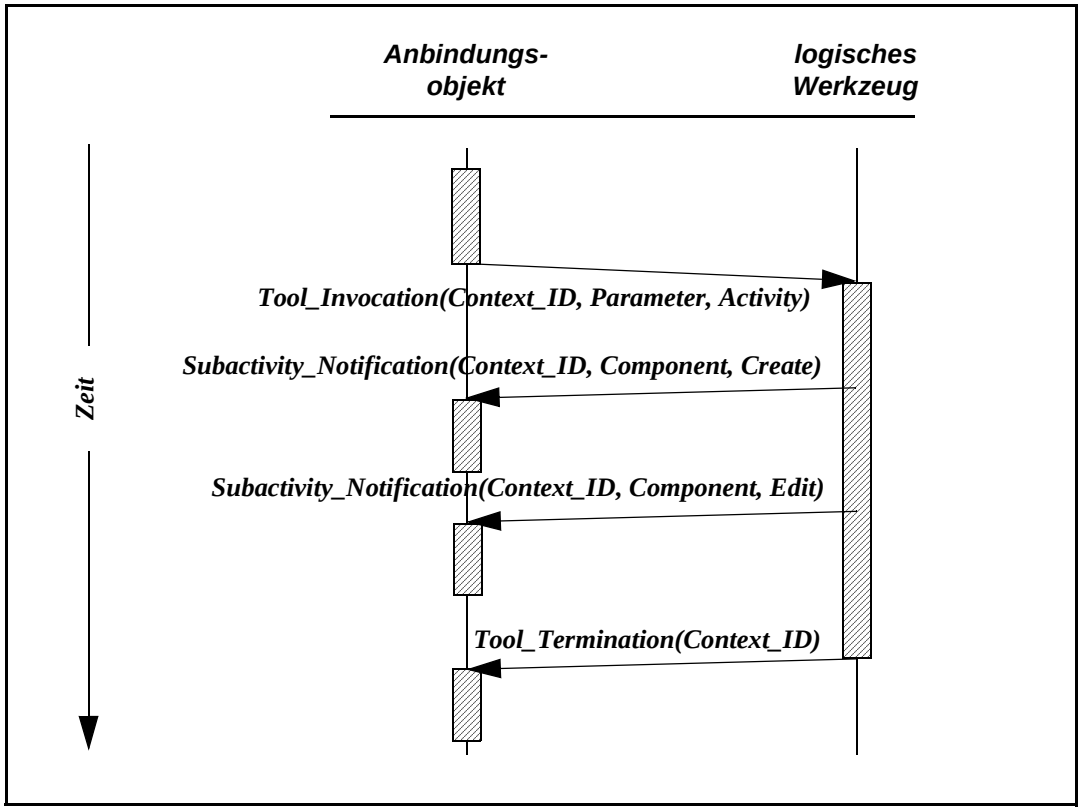


Abbildung 7.6 Schematische Darstellung der Anbindung von Grey-Box-Werkzeugen

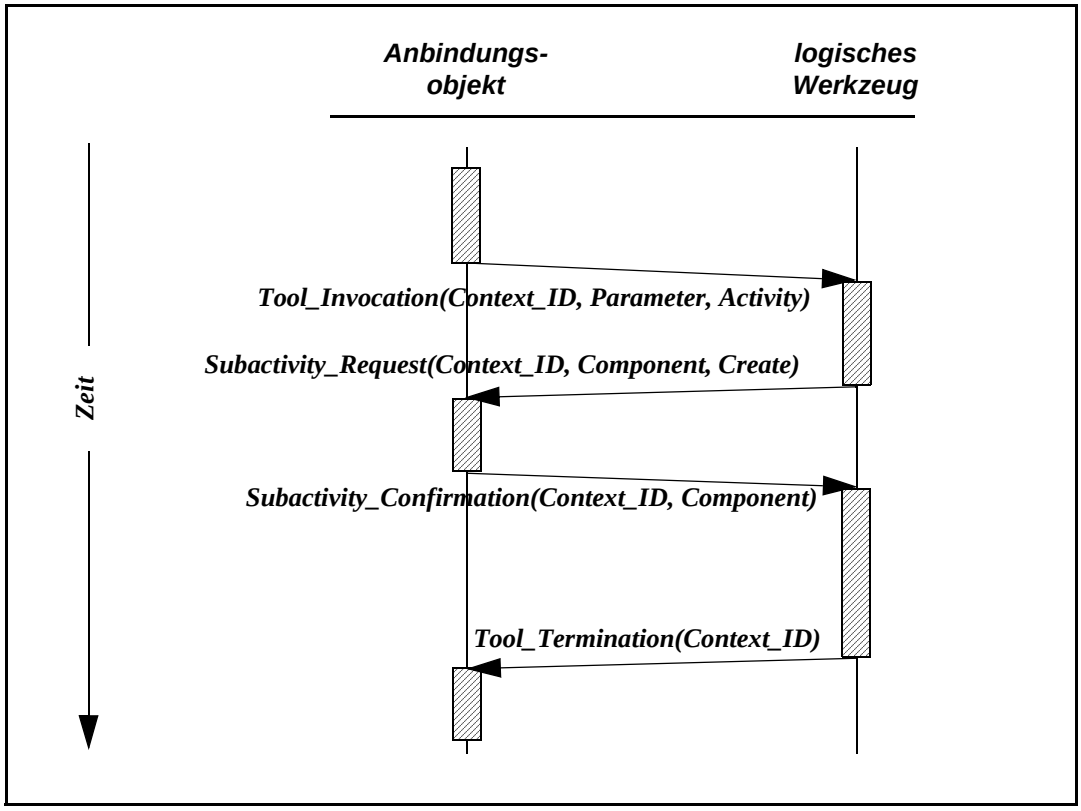


Abbildung 7.7 Schematische Darstellung der Anbindung von White-Box-Werkzeugen

Gegensatz zur `Subactivity_Notification` aus dem Protokoll zur Anbindung von Grey-Box-Werkzeugen das Nachrichtenpaar `Subactivity_Request` / `Subactivity_Confirmation` verwendet, um die Abstimmung zwischen Werkzeug und Prozeßsteuerung durchzuführen.

Mit den drei vorgestellten Protokollen kann sichergestellt werden, daß Feedback gegeben wird und damit die Anforderung zur *Übermittlung von Feedback* erfüllt ist. Im nächsten Zug muß nun sichergestellt werden, daß die Prozeßsteuerung nach der Beendigung eines Werkzeugs darüber informiert wird, ob sich der Zustand des Dokuments, auf dem das Werkzeug aufgerufen wurde, geändert hat.

7.2.2 Bestimmung des Rückgabewerts

Direkt nach der Beendigung einer Aktivität muß ausgewertet werden, ob ein Rückgabewert vom implementierenden Werkzeug gemeldet wurde und ob dieser in der Umgebungsspezifikation auf einen gültigen Folgestatus des zugrundeliegenden Dokuments abgebildet worden ist (Anforderung zur *Übermittlung von Zustandsinformation*). Bei dieser Auswertung müssen mehrere, verschieden zu behandelnde Fälle erfaßt werden.

- Das Werkzeug liefert einen verwertbaren Rückgabewert (vgl. Kapitel 6.1 und Kapitel 6.3). Der zurückgelieferte Rückgabewert ist definiert und kann direkt auf einen Folgezustand abgebildet werden. Dieser kann dann der PE gemeldet werden.
- Das Werkzeug kann den Zustand des bearbeiteten Dokuments nicht automatisch ableiten (z.B. Freitext editieren). Ein Rückgabewert wird daher vom Werkzeug nicht erwartet. Um einen Folgestatus an die PE melden zu können, müssen die mög-

lichen Statuswerte dem Benutzer zur Auswahl angeboten werden. Die Anbindungsobjekte kommunizieren also an dieser Stelle direkt mit dem Benutzer. Der im Dialog erfragte Status kann dann an die PE weitergegeben werden.

- Unabhängig vom Typ des Werkzeugs kann vom Envelope bzw. vom Anbindungsobjekt auch der Abbruch des Werkzeugs in unerwarteter Form gemeldet werden. In diesem Fall wird dieser spezielle Status direkt an die Prozeßmaschine zurückgeliefert, damit diese mit Hilfe des implementierten Recovery-Mechanismus (siehe Kapitel 7.3.2) den Folgezustand des Dokuments feststellen kann.
- Taucht ein in der Umgebungsspezifikation nicht vorgesehener Wert als Rückgabe auf, so wird ein vordefinierter Status UNKOWN vom Anbindungsobjekt an die Prozeßmaschine gemeldet, der diese auch zu Recovery-Maßnahmen auffordert. Offensichtlich kann dieser Fall nur auftreten, falls es sich um einen Spezifikationsfehler handelt.

Ist der Folgezustand bestimmt, und sind alle Aktionen im Zuge dieser Methode beendet, so wird der Zustand durch die Nachricht

finish_activity(dname, dtyp, methodenname, status)

vom Anbindungsobjekt an die Prozeßmaschine gemeldet. Der Name des bearbeiteten Dokuments `dname`, sein Dokumenttyp `dtyp` und der Name der aufgerufenen Methode `methodenname` bezeichnen darin eindeutig die Aktivität, die beendet wurde. Damit ist gewährleistet, daß der Folgestatus eines Dokuments nach seiner Bearbeitung an die Prozeßsteuerungskomponente übermittelt wird und damit die Anforderung zur *Übermittlung von Zustandsinformation* abgedeckt ist.

7.2.3 Sicherung der Prozeßkonformität

Nachdem sichergestellt wurde, daß die Prozeßsteuerungskomponente alle Informationen zur Durchführung ihrer Steuerungsaufgabe erhält, wird in diesem Kapitel untersucht, wie diese Informationen im Rahmen der Prozeßsteuerung benutzt werden können. Durch die vorgestellten Konzepte wird erreicht, daß die Anforderungen zur *Auswertung von Feedback* und zur *Kontrolle des Leistungsumfangs* abgedeckt werden können.

7.2.3.1 Auswertung des Feedbacks

Feedback von den Werkzeugen über Änderungen in den Dokumenten muß auf die Gültigkeit der Änderungen überprüft werden. Die Kontrollen, die dabei vorzunehmen sind, betreffen sowohl den aktuellen Projektzustand als auch die Überprüfung auf Prozeßmodellkonformität. Es muß geprüft werden, ob die Änderung im Modell vorgesehen war. Dies läßt sich durch einen Abgleich mit den im Prozeßmodell spezifizierten Zugriffen bestimmen. Führt z.B. ein Werkzeug eine Beziehung zwischen zwei Dokumenten ein, so muß ein entsprechender CONNECTS-Zugriff im Modell vorgesehen sein. Ist ein solcher Zugriff nicht vorgesehen, ist die Änderung nicht modellkonform.

Ist die Änderung konform zum Prozeßmodell, so muß weiterhin überprüft werden, ob sich durch die Ausführung der Änderung innerhalb der Modellausführung (bzw. innerhalb der Prozeßausführung) Inkonsistenzen ergeben. So kann es beim Hinzufügen einer Dokumentinstanz z.B. zu Namenskonflikten mit einer bereits definierten anderen

Instanz kommen. Dies gilt z.B. in Merlin, da Merlin einen globalen Namensraum besitzt (vgl. [Jun95], Kap. 5.1).

Ist die Änderung aus einem der oben genannten Gründe nicht erlaubt, so wird im Falle einer Änderungsanfrage die Anfrage zurückgewiesen. Im Falle einer Änderungsmeldung bleibt die Änderung bis zu einer expliziten Korrektur durch den Entwickler bestehen. Dadurch entsteht die Notwendigkeit, für die Prozeßsteuerungskomponente die betroffenen Dokumentinstanzen als inkonsistent und nicht prozeßkonform zu markieren und möglicherweise ein Logging über die vorgenommenen, nicht gültigen Änderungen zu führen. Das Dokument kann dann durch sukzessive Beseitigung der Problemstellen wieder in einen konsistenten Zustand überführt werden. Werden solche nicht erlaubten Änderungen bemerkt, so verbleibt das betroffene Dokument im Sinne der Modellausführung in jedem Fall im Ausgangszustand.

Mit diesen Maßnahmen wird erreicht, daß von den Werkzeugen geliefertes Feedback tatsächlich ausgewertet wird bzw. ausgewertet werden kann und dadurch die Anforderung zur *Auswertung von Feedback* abgedeckt wird. Damit bleibt nur noch zu überprüfen, wie die Einhaltung der Vorgaben durch die Spezifikation der Aktivitäten und durch die logischen Werkzeuge auch über die Feedbackmechanismen hinaus kontrolliert werden können.

7.2.3.2 Kontrollen außerhalb der Feedbackmechanismen

Sowohl vor als auch nach dem Aufruf eines Werkzeugs können Maßnahmen initiiert werden, die sicherstellen, daß der Leistungsumfang einer Aktivität nicht durch das sie implementierende Werkzeug verletzt wird.

Unter Verwendung der definierten Zugriffe kann festgelegt werden, welche der Aktivitäten zu einem bestimmten Zeitpunkt welchem Anwender angeboten werden dürfen. Um dieses konsequent zu tun, müssen die vorhandenen Mechanismen unter Umständen erweitert werden.

Beispielsweise wird in ESCAPE im Organisationsmodell spezifiziert, welche Rollen welche Aktivitäten in welchen Zuständen eines Dokuments anwenden dürfen (vgl. [Jun95, Kap.4]). Dabei beinhaltet ein ESCAPE-Prozeßmodell allerdings keine Information darüber, ob eine Aktivität lesend oder schreibend auf ein Dokument zugreift. Dadurch kann nur über die geeignete Definition der Zustandsmodelle der Dokumente sichergestellt werden, daß dem Benutzer nur Aktivitäten durch die Prozeßsteuerungskomponente angeboten werden, die wie vorgesehen auf Dokumente zugreifen.

Ein einfaches Beispiel für diesen Effekt kann mit dem Ausschnitt aus dem Organisationsmodell in Abbildung 7.8 gezeigt werden. In der Abbildung wird für den Dokumenttyp C-Modul spezifiziert, daß ein Entwickler mit einem C-Modul auch die dazugehörige Spezifikation sieht. Diese darf dabei nur gelesen werden. Es ist aber keine Unterscheidung möglich ist, ob die Aktivitäten, die auf einer Spezifikation definiert sind, diese lesen oder schreiben. Demnach werden dem Entwickler alle Aktivitäten angeboten, die in diesem Zustand der Spezifikation möglich sind. Nur wenn diese Aktivitäten alle nur lesend auf die Spezifikation zugreifen, ist sichergestellt, daß keine Verletzung des Prozeßmodells auftreten kann. Dies ist aber nur dann möglich, wenn das Zustandsmodell der Spezifikation entsprechend definiert wurde, was erzwingt, daß eine Spezifikation und ein darauf basierendes C-Modul strikt sequentiell entwickelt

Rolle	zentrale Dokumentklassen	lesbare Dokumentklassen	Zustand
Entwickler	C-Modul		Implementierung-nicht-abgeschlossen
		Spezifikation	Spezifikation-vorhanden
		Fehler-Report	Fehler

Abbildung 7.8 Ausschnitt aus einem Organisationsmodell

werden, da eine parallele Bearbeitung der Dokumente schreibende Zugriffe auch z.B. für das C-Modul erlauben würde. Diese Einschränkung ist unerwünscht.

Mit Hilfe der Zugriffe aus TIC, wie sie weiter vorne auch für ESCAPE+-Modelle definiert wurden, können in Merlin die Vorgaben aus dem Organisationsmodell so umgesetzt werden, daß die erlaubten Aktivitäten genauer bestimmt werden können. Eine Liste von Aktivitäten im Working Context, die auf solche Aktivitäten eingeschränkt ist, die nur lesend zugreifen, vermeidet Aufrufe von Aktivitäten, die eine Veränderung des betreffenden Dokumenttyps (im Beispiel also der Spezifikation) ermöglichen.

Auch die Durchführung von Aktivitäten (vgl. Problembeschreibung in Kapitel 4.2.1.3) läßt sich nun besser kontrollieren. In den Zugriffspfaden wurde definiert, welche Auswirkung die Ausführung einer Aktivität auf andere Dokumente hat. Vor dem Aufruf der eigentlichen Aktivität können damit die betroffenen Dokumente bestimmt werden. Nun kann kontrolliert werden, ob die potentiell durch die beschriebenen Pfade erreichbaren Dokumente auch in einem Zustand sind, auf den eine Rolle zugreifen darf. Verändert beispielsweise eine Aktivität außer dem Dokument, auf dem sie direkt aufgerufen wurde, andere Dokumente (in den Zugriffen/Zugriffspfaden genannt), so müssen diese in dem Zustand, in dem sie sich gerade befinden, auch geeignet (d.h. schreibbar, für Merlin im Organisationsmodell in der Spalte `zentrale Dokumentklassen` genannt) zugeordnet worden sein.

Nach Beendigung eines Werkzeugs können in einzelnen Fällen noch weitere Kontrollen ergänzt werden. So kann für Dokumente, die nur lesend zugegriffen werden durften, kontrolliert werden, ob sie identisch mit dem Dokument vor dem Aufruf sind. In einer ohne zentrales Repository zusammengestellten PSEU kann jedoch nicht mit vertretbarem Aufwand überprüft werden, ob ein Werkzeug noch auf weitere Dokumente zugegriffen hat, die nicht hätten zugegriffen werden dürfen.

Damit zeigt sich, daß noch weitere Kontrollmöglichkeiten in speziellen Konstellationen denkbar sind. Die hier beschriebenen Maßnahmen decken allerdings schon so viele der möglichen Fehlerfälle ab, daß die Anforderung zur *Kontrolle des Leistungsumfangs* ausreichend abgedeckt wird und nur im Bedarfsfall mit erhöhtem Aufwand über weitere Maßnahmen nachgedacht werden muß.

7.2.4 Kommunikation der Dokumentklassen

Jede der generierten Dokumentklassen besitzt eine Kommunikationsbeziehung mit der eingesetzten Prozeßsteuerungskomponente. In der Abbildung 7.9 sind die durch TIC eingebrachten Kommunikationsbeziehungen einer auf TIC basierenden PSEU dargestellt.

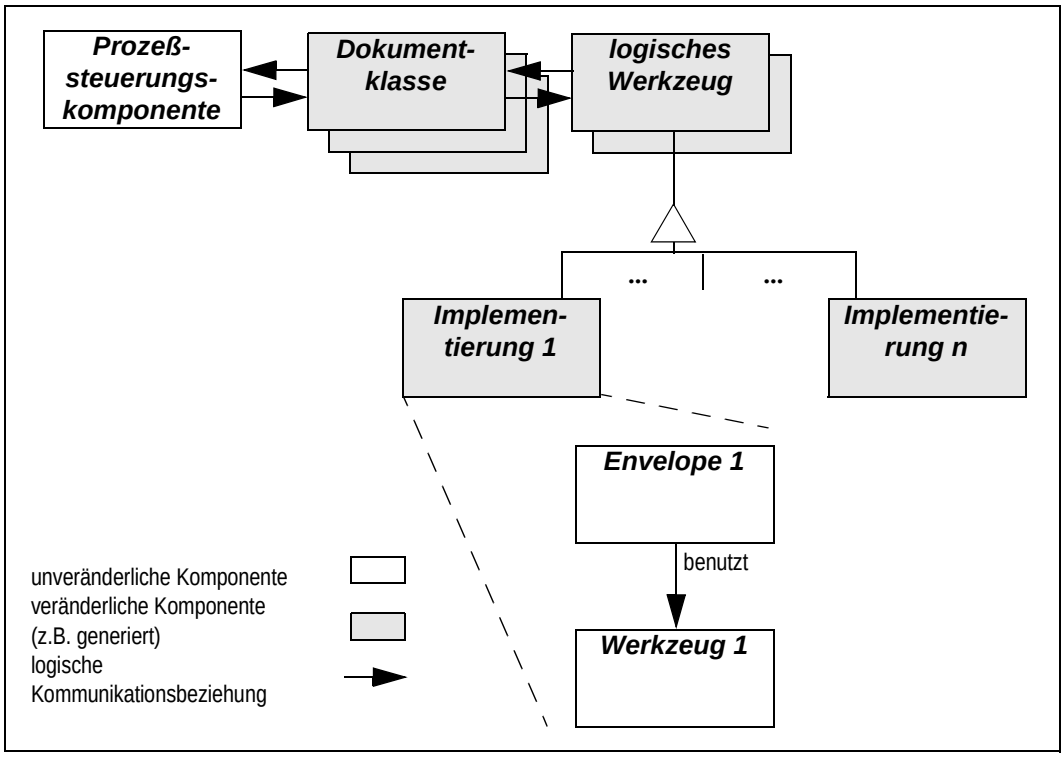


Abbildung 7.9 Kommunikationsbeziehungen in PSEU, die auf TIC basieren

Die Kommunikation zwischen der Prozeßsteuerungskomponente und den Anbindungsobjekten erfolgt auf der Basis eines weiter unten beschriebenen Protokolls. Die Kommunikationsbeziehung ist dabei beidseitig, da aufgrund der asynchronen Kommunikation zwischen den Kommunikationsteilnehmern sowohl die Prozeßmaschine als auch die Anbindungsobjekte Initiator der Kommunikation sein können. Auch zwischen Envelopes und Dokumentklassen sind die Kommunikationsbeziehungen beidseitig.

Die Kommunikation mit den Dokumentklassen basiert auf CORBA-Nachrichten. Ein Ausschnitt aus der Beschreibung der Schnittstelle von Dokumentklassen in ISL-Syntax ist in der Abbildung 7.10 zu sehen. ISL ist eine der IDL ähnliche Beschreibungssprache, die in der CORBA-Implementierung ILU verwendet wird. Im Beispiel setzt die (ILU-)Klasse `DocumentClass` den Aufruf von Aktivitäten von der PE (`call_method`) in die Aufrufe der einzelnen Dokumentklassen (z.B. `Systembeschreibung.start_edit`) um. Die Schnittstelle der Klasse `DocumentClass` ist konstant. Die Implementierung dieser Klasse und die ISL-Beschreibungen wie auch deren Implementierungen werden generiert (vgl. Kapitel 7.2.5).

Die Methoden der generierten Dokumentklassen sind auf das Protokoll zwischen PE und Anbindungsobjekten bzw. zwischen Anbindungsobjekten und Werkzeugen abgestimmt. Jede Methode entspricht einer Nachricht in Richtung des Aufrufs. Ruft also die PE (mit dem Umweg über `DocumentClass`) die Methode `start_edit` auf, so entspricht dies einer Nachricht an ein Anbindungsobjekt der Dokumentklasse `Systembeschreibung` mit dem Inhalt, daß das Werkzeug, das die Aktivität `edit` implementiert, aufgerufen werden soll.

```
INTERFACE BASIC_TYPES
IMPORTS ilu END;

TYPE STRING = ilu.CString;

TYPE CALLMODE = ENUMERATION
            START,
            STOP,
            HALT,
            CHANGE_Ack,
            CHANGE_NAck
END;

TYPE METHOD_ID = RECORD
            document_class: STRING,
            method_name: STRING
END;

TYPE LIST_OF_STRING = SHORT SEQUENCE OF STRING;
```

```
INTERFACE DOCUMENT_CLASS
IMPORTS ilu, BASIC_TYPES END;

TYPE DocumentClass = OBJECT
            /* Methoden zum Aufruf aus der PE */
            METHODS
               ASYNCHRONOUS call_method(
                    IN method_to_call: BASIC_TYPES.METHOD_ID,
                    IN mode: BASIC_TYPES.CALLMODE,
                    IN text: BASIC_TYPES.STRING /* Kommentar */,
                    INOUT params: LIST_OF_STRING /* Params */),
            /* Methoden zum Aufruf aus dem Werkzeug */
            ...
            END;
```

```
INTERFACE Systembeschreibung
IMPORTS ilu, BASIC_TYPES, DOCUMENT_CLASS END;

TYPE Systembeschreibung = OBJECT
            METHODS
               ASYNCHRONOUS start_edit(
                    INOUT self: BASIC_TYPES.STRING,
                    INOUT shown_in: BASIC_TYPES.STRING,
                    IN possible_states: LIST_OF_STRING),
               /* Weitere Methoden */
               ...
```

Abbildung 7.10 ISL-Interface von Dokumentklassen

Um ein generisches Protokoll für eine Kommunikation, die die Integration der Werkzeuge realisiert, gewährleisten zu können, muß für jedes änderbare prozeßrelevante Datum eine Möglichkeit für Feedback geschaffen werden. Somit muß ein Protokoll zwischen Werkzeugen und Dokumentklassen, das entwicklungsbedingte Änderungen behandelt, Nachrichten über Änderungen von Dokumenten und Beziehungen zwischen Dokumenten enthalten. Diese betreffen das Anlegen und das Löschen von Dokumentinstanzen sowie das Ändern ihrer Attribute, d.h. beispielsweise ihres Namens. Ebenso gehört dazu das Einfügen und das Löschen von Beziehungen zwischen Dokumentinstanzen.

Der Parameter `mode` der oben vorgestellten Methode `call_method` beschreibt die Anforderung an die aufgerufene Methode. Der Typ `BASIC_TYPES.CALLMODE` (siehe Abbildung 7.10) legt die Schlüsselwörter fest, mit denen von der Prozeßmaschine der Start einer Aktivität bzw. deren vorzeitige Beendigung nachgefragt werden kann. Die Modi `STOP` und `HALT` unterscheiden sich dabei in der Auswirkung. `STOP` fragt lediglich eine möglichst baldige Beendigung nach, was im einfachsten Fall durch ein Dialogfenster mit einem entsprechenden Hinweis realisiert werden kann. Die Beendigung wird nicht wie im Modus `HALT` erzwungen. So kann z.B. einem Benutzer mitgeteilt werden, daß sich die Datenbasis des Projekts geändert hat und er nicht mehr auf einem aktuellen Bestand arbeitet und eine Aktualisierung des Datenbestandes die Beendigung der aktuellen Arbeit voraussetzt. Der Parameter `TEXT` könnte je nach Implementierung dazu verwendet werden, z.B. eine Begründung für die `STOP`-Anforderung mitzusenden, die dann von der Dokumentklasseninstanz angezeigt oder an das Werkzeug weitergegeben werden kann.

Die Modi `CHANGE_Ack` und `CHANGE_NAck` dienen dazu, Änderungsanfragen positiv oder negativ zu quittieren.

Die von der Prozeßmaschine abgesetzten Aufrufe werden in den Anbindungsobjekten entsprechend der Angaben in der Umgebungsspezifikation in Aufrufe logischer Werkzeuge umgesetzt. Auf der Basis der Umgebungsspezifikation kann dann auch entschieden werden, ob zur Durchführung des Aufrufs eine Transformation des physikalischen Dokuments in einen anderen physikalischen Dokumenttyp notwendig ist, z.B. von einem Textdokument in ein PostScript-Dokument.

Das Protokoll zwischen den Envelopes und den Anbindungsobjekten ist ähnlich generisch wie das zwischen der Prozeßsteuerungskomponente und den Anbindungsobjekten. Es werden verschiedene Standardnachrichten definiert, deren Benutzung teilweise vom Werkzeugtyp abhängt (vgl. Kapitel 7.2.1). Die Funktionalitäten eines Envelopes sind

- Start,
- Stop,
- Halt,
- Änderungsanfrage angenommen,
- Änderungsanfrage abgelehnt,
- Änderung durchführen.

Umgekehrt nimmt ein Werkzeug maximal die folgenden Funktionalitäten der Anbindungsobjekte in Anspruch:

- Start bestätigen,
- Stop bestätigen,
- benutzer-/werkzeuginitiierten Stop melden,
- Änderungsanfrage stellen,
- Änderungsmitteilung machen,
- Mitteilung über Persistenz einer angenommenen Änderung an PE weitergeben.

Zu bemerken ist, daß die Funktion zum Start den gesamten Komplex der Auswertung der übergebenen Parameter sowie den eigentlichen Aufruf des Werkzeugs und das Initialisieren des Envelopes verkapselt, so daß später z.B. Anfragen an das richtige Anbindungsobjekt gestellt werden können.

7.2.5 Generierung von Dokumentklassen

Der Schluß des Kapitel 7.2 beschäftigt sich damit, wie Dokumentklassen generiert werden können. Als Basis für die Generierung dienen die Umgebungsspezifikation und das Prozeßmodell, aus denen abgeleitet wird, welche Methoden durch eine Dokumentklasse angeboten und implementiert werden müssen. Zur Implementierung der Dokumentklassen werden Komponenten benutzt, die mit den Basisinformationen parametrisiert werden (z.B. welcher Werkzeugzustand auf welchen Modellzustand abgebildet werden muß). Die Komponenten korrelieren dabei direkt mit den Aufgaben, die für Dokumentklassen identifiziert wurden. Die Aufgaben umfassen dabei die folgenden Aspekte:

- Abbildung der Aufrufe von Aktivitäten auf Aufrufe der Werkzeuge
- Verwaltung notwendiger Informationen zur Abbildung von Nachrichten von den Werkzeugen an die Prozeßmaschine
- Durchführen eines Dialogs zum Feststellen vorgenommener Änderungen nach Beendigung von Black-Box-Werkzeugen
- Abfrage des Zustands des bearbeiteten Dokuments bei allen Werkzeugen, die nicht in der Lage sind, diesen automatisch zu liefern
- Abbildung eines vom Werkzeug automatisch bestimmten Zustands auf die Werte des Prozeßmodells
- Abbildung der Nachrichten von der Prozeßmaschine an die Anbindungsobjekte auf Nachrichten von den Anbindungsobjekten an das Werkzeug

Auf Kommunikationskomponenten wurde im vorangegangenen Kapitel sowie im Kapitel 7.2.1 schon eingegangen. Die nötigen Abfragen nach Beendigung eines Werkzeugs (Status und Änderungen) werden in [Kur97] vertiefend behandelt. Auf die Umsetzung der anderen Komponenten wird an dieser Stelle nicht weiter eingegangen, da es sich nicht um sehr aufwendig zu realisierende Aufgaben handelt.

Bei der Generierung ist zu beachten, daß die Abfrage des Zustands nur dann benötigt wird, wenn der repräsentierte Dokumenttyp Aktivitäten enthält, die in der Umgebungs-

spezifikation mit dem Typ USER INTERACTIVE spezifiziert sind. Hat er keine solchen Aktivitäten, so lassen sich alle Bearbeitungszwischenstände automatisch ableiten und die Abfrage des Bearbeitungsstatus ist überflüssig. Die bei der automatischen Abbildung von Rückgabewerten oder der manuellen Abfrage eines Bearbeitungsstandes vorgesehenen Zuordnungen bzw. Werte werden der Umgebungsspezifikation entnommen und in die Dokumentklassen mit hineingeneriert.

Alle anderen Komponenten werden in jedem Fall in die Anbindungsobjekte hineingeneriert.

7.3 Behandlung von PE- und Werkzeugabbrüchen

Nicht nur durch Datenänderungen kann es zu Situationen kommen, die zu einer Prozeßsteuerung führen. Auch Abbrüche von PE oder Werkzeugen können Probleme auslösen. Daher wird TIC im letzten Schritt um Konzepte erweitert, die es einer PSEU ermöglichen, Abbrüche zunächst einmal zu erkennen (Anforderung zur *Erkennung von Werkzeugabbrüchen*), um dann kontrolliert wieder aufsetzen zu können.

7.3.1 Erkennung von Werkzeugabbrüchen

In Kapitel 4.3.2 wurden zwei Typen von Werkzeugabbrüchen unterschieden: Kontrollierte und unkontrollierte Abbrüche. Zur Erkennung kontrollierter Werkzeugabbrüche sind keine Maßnahmen nötig, da in diesen Fällen noch eine Information über den Abbruch vom Werkzeug an die PE zurückgeliefert wird. Im Falle unkontrollierter Abbrüche, bei denen das abgebrochene Werkzeug keine Information mehr an die Prozeßsteuerung schickt bzw. schicken kann, muß die Prozeßsteuerung den Abbruch anders erkennen. Dazu werden im folgenden drei Maßnahmen entwickelt, die in verschiedenen Fällen greifen.

Erkennung auf Envelope-Ebene: Auf der Seite der Werkzeuge kann zunächst ausgenutzt werden, daß die Werkzeuge durch einen Envelope eingekapselt sind. Der Envelope kann nun so geschrieben werden, daß er den Abbruch des aufgerufenen Werkzeugs bemerkt. Dazu werden betriebssystemnahe Mechanismen, wie z.B. unter UNIX die Kontrolle von Prozeßkennungen, verwendet (vgl. Beispiel in Abbildung 7.2). In allen Fällen, in denen ein Envelope den Abbruch des von ihm gestarteten Werkzeugs bemerkt, wird durch ihn eine entsprechende Zustandsmeldung durchgeführt. Diese Fälle können dann wie kontrollierte Abbrüche behandelt werden.

Erkennung auf Ebene der Prozeßsteuerung bei interaktivem Werkzeug: Ist auch eine Fehlermeldung durch den Envelope nicht mehr möglich, beispielsweise wegen eines Rechnerausfalls, so werden Abbrüche durch die Prozeßsteuerung auf der Ebene der Aktivitäten festgestellt. Um dies zu ermöglichen, wird von der Prozeßsteuerung mitprotokolliert, welche Aktivitäten gestartet worden sind, und der Benutzer erhält zusätzlich die Möglichkeit, den Abbruch einer Aktivität bei der Prozeßsteuerung zu melden.

Erkennung auf Ebene der Prozeßsteuerung bei automatischem Werkzeug: Wird ein Abbruch bei einem automatischen Werkzeug nicht durch seinen Envelope gemeldet, so merkt die Prozeßsteuerung den Abbruch durch die Kontrolle ihrer Daten. Die Prozeßsteuerung stellt dabei fest, daß eine Aktivität noch nicht regulär beendet wurde, wenn keine abschließende Zustandsmeldung gesendet wird. In diesem Fall kann aktiv kon-

trolliert werden, ob das zugehörige Anbindungsobjekt noch existiert und ob die Aktivität wirklich noch nicht beendet wurde oder aber ob ein Abbruch vorliegt. Es ist eine Frage der Implementierung, wann die Überprüfung der Daten auf einen möglichen Abbruch hin durchgeführt wird.

Handelt es sich um einen gemeinsamen Abbruch von Werkzeug und Prozeßsteuerung, so kann die Prozeßsteuerung den Abbruch beim nächsten Start feststellen. Es wird dann zu Beginn der Ausführung kontrolliert, ob noch Aktivitäten als laufend eingetragen sind. Dies ist ein sicheres Zeichen dafür, daß die Prozeßsteuerung abgebrochen wurde. Maßnahmen können sich analog zu dem oben ausgeführten Fall anschließen.

Werden diese Maßnahmen in einer TIC-basierten Umgebung implementiert, so geht die Information über Werkzeugabbrüche der Prozeßsteuerung nicht mehr verloren. Die Prozeßsteuerung kann also in jedem Fall geeignet reagieren (vgl. Kapitel 7.3.2). Die Anforderung zur *Erkennung von Werkzeugabbrüchen* ist damit erfüllt. Im einem weiteren Schritt muß dafür gesorgt werden, daß Fehler im Prozeßablauf durch Abbrüche erkannt und korrigiert werden.

7.3.2 Recovery nach Abbrüchen

Die Fortsetzung eines Prozesses nach Fehlern der Laufzeitumgebung wird durch die Anforderungen zur *Zustandsbestimmung nach Werkzeugabbruch* und zur *Zustandsbestimmung nach PE-Abbruch* angesprochen. Um diese Anforderungen in einer PSEU abdecken zu können, werden Recovery-Maßnahmen benötigt, wenn ein Abbruch erkannt wird. Merlin ist ein gutes Beispiel hierfür, weil systembedingte Unterbrechungen von ESCAPE+ nicht vorgesehen sind (vgl. Kapitel 4.2.2). Dies liegt daran, daß die Aktivitäten im Modell als Zustandsübergänge von vernachlässigbarer Zeitdauer angesehen werden. Der Ablauf einer Aktivität findet in diesem Sinne immer im Ausgangszustand der Aktivität statt. Erst die Beendigung der Aktivität löst mit der Bekanntgabe des zu wählenden Folgezustandes den tatsächlichen Übergang aus. Diese Sichtweise entspricht allerdings nicht unbedingt den Abläufen in der Prozeßausführung.

Wird entweder die Modellausführung oder aber eines der Werkzeuge ungeplant unterbrochen, so führt dies dazu, daß für die Daten, die der Modellausführung zugrunde liegen, nicht mehr sichergestellt ist, daß sie konsistent zu den Daten der Prozeßausführung sind. Weiterhin könnten auch die nicht prozeßrelevanten Daten korrumpiert worden sein. Beide Aspekte müssen bei der Definition von Recovery-Mechanismen berücksichtigt werden. Damit stellt sich die Aufgabe, nach einer ungeplanten Unterbrechung einer der Komponenten zu erkennen, welche der obigen Fehlersituationen aufgetreten sind, um sie dann wieder zu bereinigen. Für Recovery-Mechanismen in einer PSEU wird meistens die Hilfe eines Benutzers benötigt, da die Granularität der Daten in der Modellausführung in der Regel nicht der kleinsten Änderungseinheit der Prozeßausführung entspricht. Nachfolgend werden zwei Konzepte eingeführt, mit deren Hilfe ein Wiederaufsetzen nach einem Abbruch möglich ist.

Die erste Maßnahme betrifft die bis zum Zeitpunkt eines Abbruchs durchgeführten Änderungen an Dokumenten. Sie nutzt die weiter oben vorgestellten Feedback-Protokolle (vgl. Kapitel 7.2.1.2) aus. Die beschriebenen Protokolle werden dahingehend erweitert, daß nach der Meldung, daß eine Änderung persistent gemacht wurde, eine weitere Bestätigung als Abmeldung dieser Änderung an die Prozeßsteuerung gesendet

wird. Erst danach wird die Änderung von der Prozeßsteuerung vorbehaltlos übernommen. Kommt es nun zu einem Absturz des Werkzeugs, bevor eine solche Bestätigung angekommen ist, so muß für alle noch nicht bestätigten Änderungen manuell kontrolliert werden, ob sie wirklich schon durchgeführt wurden. Falls tatsächlich Änderungen verloren gingen, so erhält ein Entwickler eine gute Hilfestellung zur Rekonstruktion der verlorenen Daten. Die Änderungen können erneut durchgeführt oder aber zurückgenommen werden. Falls der Abbruch des Werkzeugs erfolgte, nachdem eine Änderung persistent gemacht worden ist und bevor die entsprechende Meldung an die Prozeßsteuerung geschickt wurde, bestätigt der Entwickler die Änderungen, und sie werden in die Modellausführung übernommen.

Als zweite Maßnahme werden Recovery-Möglichkeiten implementiert, die den Status von Dokumenten wieder herstellen, die zum Zeitpunkt eines Absturzes bearbeitetet wurden. Das Ergebnis einer Aktivität, deren implementierendes Werkzeug während seiner Laufzeit unterbrochen wird, ist abhängig von der konkreten Aktivität und vom Zeitpunkt, zu dem der Abbruch erfolgte. Das Dokument, auf dem die Aktivität aufgerufen wurde, kann entweder im Ausgangszustand, in einem der Folgezustände oder in einem undefinierten Zustand sein.

Im Umfeld einer PSEU besteht das Problem, einen Aufsetzpunkt im Prozeßablauf festzulegen, an dem der Prozeß nach Abbruch einer Aktivität fortgesetzt werden kann. Die Prozeßausführung muß also das Wiederaufsetzen nach einem Aktivitätsabbruch erlauben. Das bedeutet, daß Zustände für die durch einen Abbruch betroffenen Dokumente gesetzt werden müssen, die dem Status der Dokumente beim Abbruch entsprechen.

Die im einzelnen zu untersuchenden Fälle ergeben sich im wesentlichen durch drei Unterscheidungskriterien. Zu unterscheiden ist, a) ob es sich um eine Unterbrechung der Prozeßmaschine, eines Werkzeugs oder sogar beider Teile handelt, b) welchen Typ (Ausführungstyp, vgl. Abbildung 6.19) die Aktivität hat und c) ob es sich um eine ausschließlich lesende oder eine (auch) schreibende Aktivität handelt. Diese drei Aspekte sind orthogonal zueinander.

Zur Definition des Recovery-Vorgehens wird eine spezielle Funktion *Recover* eingeführt, durch die sich das Verhalten der Prozeßsteuerungskomponente nach einem Abbruch formal beschreiben läßt. Diese Funktion muß sicherstellen, daß jeder Abbruch der PE oder eines Werkzeugs während der Projektlaufzeit für ein bearbeitetes Dokument wieder in einen definierten Zustand mündet.

Am Beispiel Merlin kann gut gezeigt werden, wie die Funktion aussieht. Sie wird in Ergänzung der bereits eingeführten Formalismen aus Kapitel 6.1 definiert. Gleichung (EQ 27) bis Gleichung (EQ 31) enthalten zunächst die zu ESCAPE zugehörigen Teile, die keinen unvorhergesehenen Ausgang zulassen. (vgl. [Jun95], Kap. 5.2):

$$\text{Zb: endliche, nichtleere Menge von Bezeichnern für Zustände und Platzhalter} \qquad \text{(EQ 27)}$$

$$\text{Z: Zustandsrelation, } Z \subset Zb \times Zb \qquad \text{(EQ 28)}$$

$$\text{P: Platzhalterrelation, } P \subset Zb \times Zb \qquad \text{(EQ 29)}$$

$$\text{L: endliche Menge von Labeln} \qquad \text{(EQ 30)}$$

$$\text{T: Übergangsrelation, } T \subset Z \times L \times \{Z \cup P\} \qquad \text{(EQ 31)}$$

Im folgenden wird nun der undefinierte Zustand (als Zeichen $\perp$) als mögliches Ergebnis eines Übergangs zugelassen und Recover so definiert, daß es einen solchen Zustand wieder zurück in den definierten Bereich überführt.

$$Z^{\perp} \subset Zb \times \{Zb \cup \{\perp\}\}, \perp \text{ ist undefinierter Zustand} \quad \text{(EQ 32)}$$

$$P^{\perp} \subset Zb \times \{Zb \cup \{\perp\}\}, \perp \text{ ist undefinierter Zustand} \quad \text{(EQ 33)}$$

$$T^{\perp}\text{: erweiterte Übergangsrelation: } T^{\perp} \subset Z \times L \times \{Z^{\perp} \cup P^{\perp}\}, T \subset T^{\perp} \quad \text{(EQ 34)}$$

$$\text{Recover: } T^{\perp} \rightarrow T \quad \text{(EQ 35)}$$

$$\text{Recover: } t \rightarrow t' \text{ mit } \begin{matrix} t' \equiv t \text{ für } t \in T \\ t' \in T \text{ sonst} \end{matrix} \quad \text{(EQ 36)}$$

Die neu definierte erweiterte Übergangsrelation $T^{\perp}$ erlaubt für jede Transition (und damit für jede Aktivität) neben den definierten Folgezuständen auch den Übergang in den undefinierten Zustand. Die Funktion Recover bildet ein Element aus der Menge der $T^{\perp}$ (Transitionen mit definiertem oder undefiniertem Ausgang) auf ein Element der Menge T (Transitionen mit definiertem Ausgang) ab. Diese Abbildung ist so aufgebaut, daß gültige Transitionen (Elemente aus T) unverändert auf sich selbst abgebildet werden. Die durch einen Abbruch entstehenden Transitionen mit dem undefinierten Zustand als Folgezustand (Elemente aus $T^{\perp}\backslash T$) werden wiederum auf eine der gültigen Transitionen abgebildet. Es ist nicht auszuschließen, daß nach einem Abbruch der gültige Folgezustand identisch mit dem Ausgangszustand ist, selbst wenn dieser Übergang im zugrundeliegenden Prozeß nicht vorgesehen ist. Es müssen daher für jeden Zustand auch Übergänge auf sich selbst definiert werden. Dazu wird die Aktivität `recover` modelliert, die in den Ausgangszustand zurück führt. Dies wird auf der formalen Ebene durch eine entsprechende Transition in T repräsentiert (vgl. Abbildung 7.11). Die Aktivität recover verändert ein Dokument nicht.

$$t_{recover} = (z, recover, z),\ z \in Z,\ recover \in L$$

Abbildung 7.11 Transition mit der Aktivität `recover`

Das Besondere an der Aktivität recover ist, daß sie im Prozeßmodell nicht explizit angegeben werden muß, da das Prozeßmodell sonst unnötig aufgebläht würde. Abbildung 7.12 zeigt einen Ausschnitt aus einem ESCAPE-Prozeßmodell, wie er sich mit vollständiger Angabe der Aktivität recover darstellen würde.

Die Aktivität recover kann in Merlin direkt in ein ausführbares Modell eingebracht werden. Bei der Generierung des Projekts aus dem Prozeßmodell werden für Merlin PROLOG-Fakten erzeugt, die von dem PROLOG-Programm, das die Merlin Prozeßsteuerung implementiert, benutzt werden. Entsprechend wird auch die Aktivität recover in Fakten übersetzt (vgl. Abbildung 7.13), die bei Bedarf durch die Regeln der Prozeßsteuerung interpretiert werden können.

Mit Hilfe der Funktion Recover kann der Folgezustand eines Dokuments nach dem Abbruch einer Aktivität bestimmt werden. Dabei muß Verschiedenes beachtet werden. Da die Prozeßmaschine beim Aufsetzen nach einer Unterbrechung nicht entscheiden kann, ob möglicherweise Daten der Prozeßausführung korrumpiert worden sind, gilt, daß ein bearbeitetes Dokument aus einer solchen Aktivität heraus nicht ohne Kontrolle

Statechart Objekttyp C-Modul – Auszug

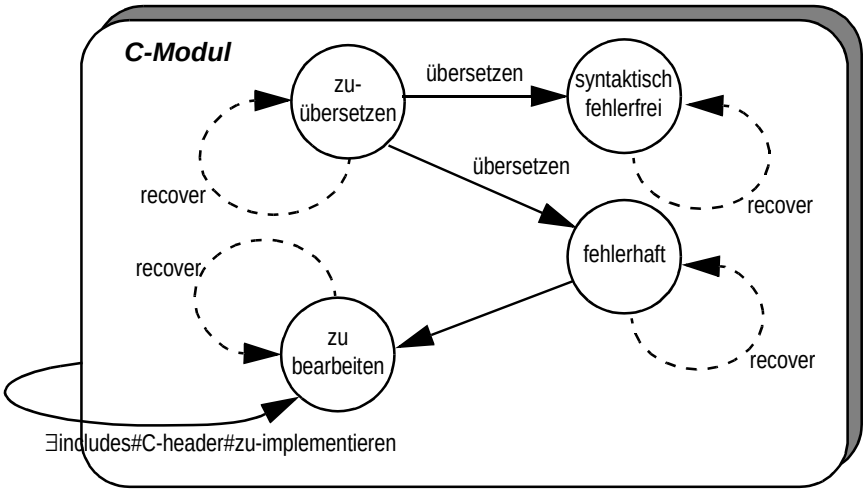


Abbildung 7.12 Ausschnitt aus einem ESCAPE-Prozeßmodell mit vollständiger Angabe der Aktivität `recover`

object_class_recover(ProcessName, DocumentClass, ActivityName, PreviousState, UndefState).

z.B.
object_class_recover(demoProcess, C-Modul, recover, zu-übersetzen, zu-übersetzen).

Abbildung 7.13 Merlin-Fakten zur Beschreibung der Recover-Funktion

des Benutzers in einen geänderten Zustand überführt werden darf. Dies ist wichtig, da eine solche Kontrolle auch bei automatischen Aktivitäten durchgeführt werden muß. Bei der Ausführung einer automatischen Aktivität muß das bearbeitete Dokument nicht unbedingt in einem Arbeitsbereich liegen, d.h. es muß aktuell keinem Bearbeiter zugeordnet sein. In einem solchen Fall muß die letzte Rolle bestimmt werden, die das Dokument vor der Ausführung der automatischen Aktivität interaktiv bearbeitet hat. In Merlin wird dazu in einem Projektfaktum abgelegt, wer der letzte Bearbeiter war und in welcher Rolle er das Dokument bearbeitet hat. Ist der letzte Bearbeiter gefunden, so muß diesem das Dokument erneut vorgelegt werden. Er muß nun entscheiden, wie die Bearbeitung fortgesetzt werden soll.

War die Aktivität nur lesend, so liegen die betroffenen Dokumente auf jeden Fall unverändert vor. Es ist jedoch nicht sicher, ob und mit welchem Folgezustand die Aktivität abgeschlossen wurde. Da ein erneutes Lesen nie zu Problemen führt, wird in diesem Fall generell der Ausgangzustand als Folgezustand gewählt. Dies führt bei einer automatischen Aktivität zu ihrer erneuten Ausführung.

Kann die Aktivität Dokumente auch verändern, so könnten die betroffenen Dokumente sogar in einem in sich inkonsistenten Zustand hinterlassen worden sein. Die Daten können aber auch ganz korrekt vorliegen, so daß ein Fortschreiben der Dokumente in den entsprechenden Folgezustand unbedingt erfolgen muß. Um eine Kontrolle zu gewährleisten, wird bei interaktiven Aktivitäten der Benutzer befragt. Dieser hat nach

der Kontrolle der betroffenen Dokumente die Möglichkeit, einen Folgezustand anzugeben oder mit Hilfe anderer angebotener Mechanismen (in Merlin z.B. das Versionsverwaltungssystem aus [Sac99]) den Zustand vor dem Aufruf der Aktivität wiederherzustellen.

Durch die beschriebenen Konzepte werden die Anforderungen zur *Zustandsbestimmung nach Werkzeugabbruch* und zur *Zustandsbestimmung nach PE-Abbruch* erfüllt, da sowohl bei einem Werkzeugabbruch wie auch bei einem PE-Abbruch der Folgezustand eines Dokuments durch Anwendung der Funktion Recover und durch Einbeziehung des Benutzers bestimmt werden kann. Die Umsetzung der entsprechenden Funktionalität wird in der Process Engine vorgenommen.

7.4 Architektur

7.4.1 TIC-Rahmenarchitektur

7.4.1.1 Komponenten der Architektur

Auf TIC basierende PSEU sind in eine gemeinsame Rahmenarchitektur eingebettet. Die Rahmenarchitektur kennt dabei Teile, die für alle TIC basierenden Architekturen gleich sind, und Teile, die sich für jede PSEU unterscheiden. Die einzelnen Komponenten der Architektur korrespondieren mit den in den Kapiteln 6 und 7 vorgestellten Elementen von TIC.

Die erste Ebene der Rahmenarchitektur enthält die Komponenten, mit denen ein ausführbares Prozeßmodell erstellt wird. Sie sind abhängig von der eingesetzten Prozeßmodellierungssprache und damit PSEU-spezifisch.

1. Mit dem sogenannten *Modelleditor* wird ein *Prozeßmodell* erstellt. Der Modelleditor erlaubt die Eingabe eines Prozeßmodells in der jeweils gewählten PML. Bei der Eingabe überprüft er die Syntax sowie die statische Semantik des eingegebenen Modells. Ein Editor, der diese Überprüfungen nicht vornimmt, kann auch eingesetzt werden, läßt aber entsprechenden Spielraum für Fehler.
2. Dieses Prozeßmodell wird dann bis auf die Aktivitäten für ein konkretes Projekt instanziiert (*instanziiertes Prozeßmodell*). Dazu wird ein Editor eingesetzt, der das Prozeßmodell aus dem ersten Schritt lesen kann und der es erlaubt, das Prozeßmodell zu instanziieren (*Projekteditor*).
3. Das instanziierte Prozeßmodell wird von einem *Übersetzer* in eine für die PE ausführbare Form gebracht. Das *ausführbare Modell* ist nun die Basis der Prozeßausführung durch die PE (*PE mit Recovery + Kernkomponenten*). Diese Übersetzung wird allerdings nur dann benötigt, wenn die vorgeschalteten Editoren nicht schon ein durch die PE ausführbares Modell erzeugen. Falls sie dieses tun, so ist das instanziierte Prozeßmodell offensichtlich identisch mit dem ausführbaren Modell. Zur Vereinfachung soll für diesen Fall gelten, daß der Übersetzer eine Kopie des instanziierten Modells für die Ausführung erzeugt.

Während der erste und zweite Schritt zur Modellierung gehören (vgl. Kapitel 6.1), bildet der dritte Schritt den Übergang zur Implementierung (vgl. Kapitel 7).

Die Architekturkomponenten, die diese Ebene realisieren, sind in der Abbildung 7.14 dargestellt. Dabei ist es unerheblich, ob in der konkreten Architektur der Modelleditor

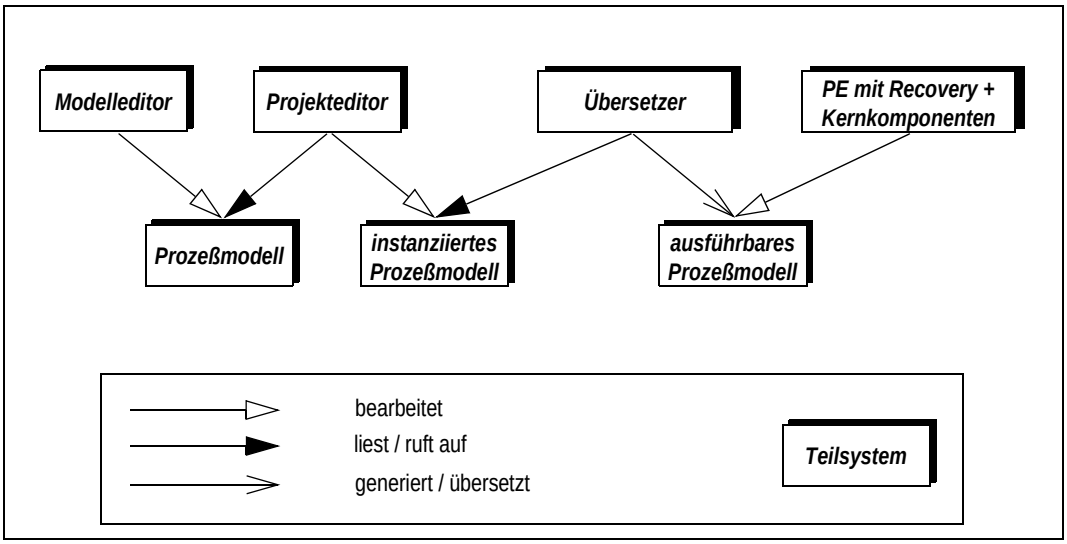


Abbildung 7.14 Komponenten der TIC-Rahmenarchitektur - Prozeßmodellierung

z.B. durch verschiedene Einzeleditoren realisiert oder das Modell möglicherweise in einzelne Teile getrennt wird. Diese Teile werden daher in der Architektur als Teilsysteme dargestellt, die sich beliebig zusammensetzen dürfen.

Auf einer weiteren Ebene der Architektur werden die zur Beschreibung und zum Aufruf der Werkzeuge notwendigen Schnittstellen eingeführt (vgl. Kapitel 6.2). Die Werkzeugschnittstellen werden unabhängig von einer konkreten PSEU formuliert. Die Komponenten dieser Ebene sind entsprechend nicht PSEU-spezifisch.

1. Im ersten Schritt werden die *logischen Werkzeuge* erstellt. Der dazu notwendige Editor (*Editor logische Werkzeuge*) überprüft, wie die bisherigen Editoren, die Syntax und die statische Semantik (siehe Anhang B) der Eingabe.
2. Die Implementierung eines logischen Werkzeugs besteht in der Regel aus einem *Envelope* und einem gekapselten *Werkzeug*. Die Envelopes werden mit dem sogenannten *Envelope-Editor* erstellt. Als Spezialfall ist auch denkbar, daß einzelne Werkzeuge so geschrieben sind, daß eine zusätzliche Kapselung nicht nötig ist. Der in der Architektur vorgesehene Envelope könnte dann wegfallen. Zur Vereinfachung wird angenommen, daß es sich um einen Envelope handelt, der die Kommunikation mit dem gekapselten Werkzeug unverändert durchreicht.

Da die logischen Werkzeuge die abstrakte Beschreibung der Envelopes und der durch sie gekapselten Werkzeuge bilden, sind sie plattformunabhängige Komponenten der Architektur. Envelopes und Werkzeuge selbst sind zwar feste Bestandteile der Rahmenarchitektur, unterscheiden sich aber möglicherweise von Plattform zu Plattform in der konkreten Architektur der Umgebung. Die Verwendung eines logischen Werkzeugs kommt in dieser Betrachtung einer polymorphen Variable in einer objektorientierten Sprache gleich, wobei die Envelopes/Werkzeuge den Spezialisierungen entsprechen, die der polymorphen Variable zugewiesen werden dürfen (z.B. plattformabhängig). Jedes benutzte logische Werkzeug ist in diesem Sinn ein polymorpher, abstrakter Platzhalter.

In der Rahmenarchitektur werden logische Werkzeuge, Envelopes und Werkzeuge zusammen mit den sie bearbeitenden Editoren direkt als Komponenten übernommen. In einer konkreten Architektur gibt es natürlich mehrere logische Werkzeuge, Envelopes und Werkzeuge, die abhängig von der konkreten Umgebung Abhängigkeiten aufweisen. Davon wird in der vorliegenden Darstellung der Rahmenarchitektur durch die Angabe allgemeingültiger Abhängigkeiten zwischen den Komponenten abstrahiert, Envelopes rufen also generell Werkzeuge auf etc. (siehe Abbildung 7.15).

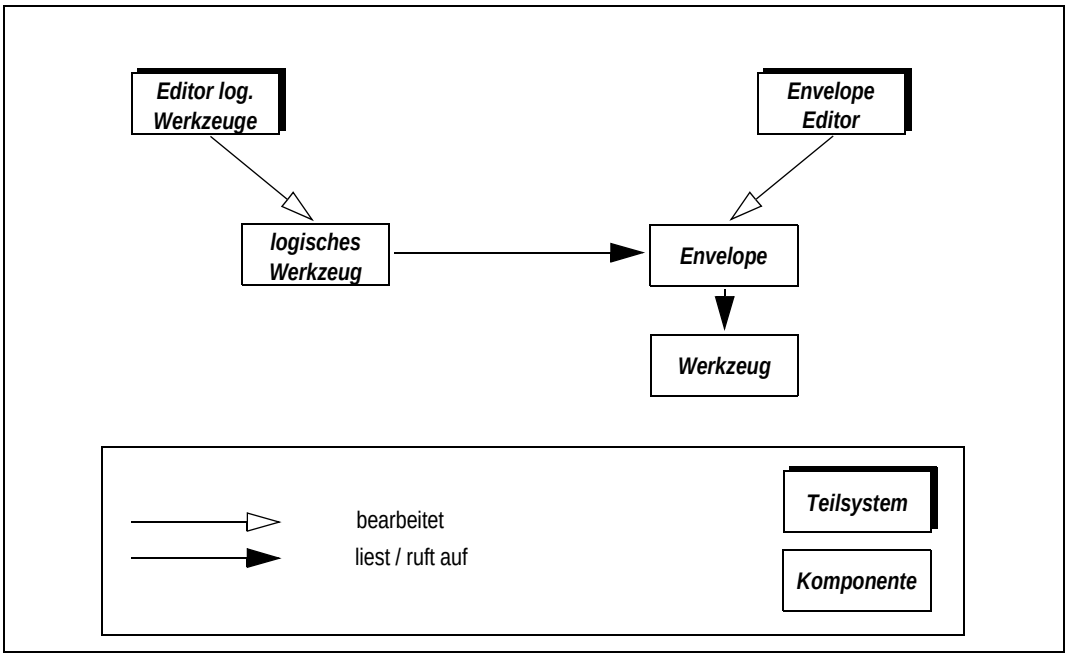


Abbildung 7.15 Komponenten der TIC-Rahmenarchitektur - Werkzeugspezifikation

Die dritte Ebene der Rahmenarchitektur liegt zwischen den beiden bisher vorgestellten. Diese Ebene enthält Komponenten, die zur Verbindung von Aktivitäten und Werkzeugen sowohl bei der Modellierung wie auch zur Laufzeit benötigt werden. In dieser Ebene gibt es PSEU-spezifische wie auch allgemeine Komponenten.

1. Zur weiteren Spezifikation der Umgebung wird ein Editor (*Umgebungseditor*) eingeführt, der die Informationen aus dem Prozeßmodell extrahiert, die die Aktivitäten betreffen. Diese Informationen können dann mit dem Editor zu einer vollständigen *Umgebungsspezifikation* verfeinert werden. In der Umgebungsspezifikation werden die logischen Werkzeuge benutzt, um die Implementierung der Aktivitäten festzulegen. Die Umgebungsspezifikation wird unabhängig von einer konkreten PSEU formuliert. Der Editor ist PSEU-spezifisch, da er die Prozeßmodelle liest, er besitzt aber eine nicht von der PSEU abhängige Schnittstelle zur Umgebungsspezifkation. Die Komponenten zur Abbildung der Aktivitäten auf die Werkzeuge und umgekehrt sind damit vollständig.
2. Nun muß die konkrete Umgebung noch durch Aufrufe von Werkzeugen, die die Implementierung der Aktivitäten bilden, realisiert werden. Die *Dokumentklassen* werden aus der Umgebungsspezifikation durch einen *Dokumentklassengenerator* erzeugt, so daß eine weitere manuelle Tätigkeit hier nicht notwendig ist (vgl. Kapitel 6.3). Diese Komponenten sind offensichtlich auch unabhängig von einer konkreten PSEU.

Dieser Architekturteil ist in Abbildung 7.16 dargestellt. Die Instanzen der Dokumentklassen (Anbindungsobjekte) realisieren die wesentlichen Teile der in Kapitel 7 dargestellten Anbindungsfunktion zwischen PE und Werkzeugen zur Laufzeit. Daher wird der Aufbau der Dokumentklassen später noch detaillierter betrachtet.

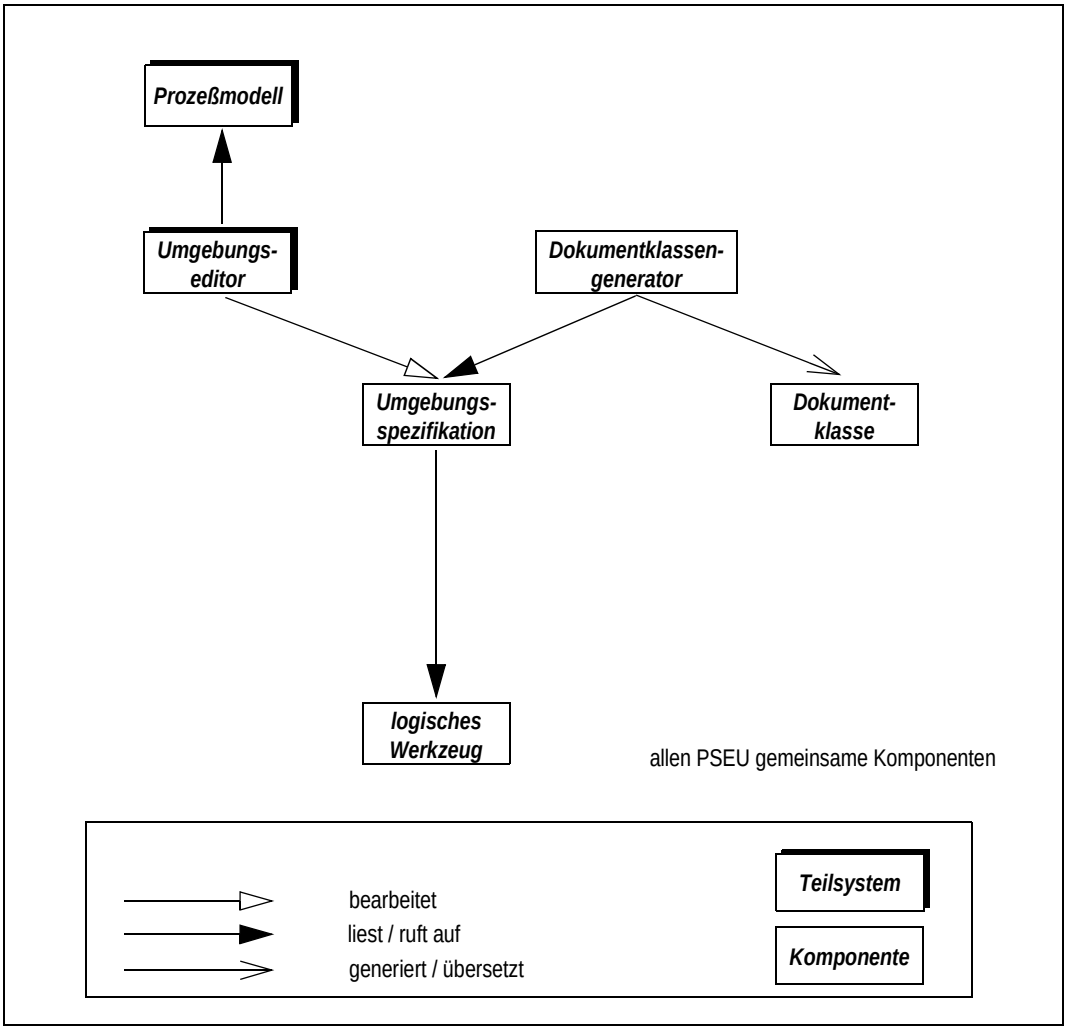


Abbildung 7.16 Komponenten der TIC-Rahmenarchitektur - Umgebungspezifikation

Die eingeführten Ebenen zur Gliederung der Architektur werden als *Prozeßebene*, *Werkzeugebene* und *Anbindungsebene* angesprochen. Das Gesamtbild der vorgestellten Architektur und die Gliederungsebenen sind in Abbildung 7.17 zu sehen. Die Unterteilung in zwei Hälften (gepunktete, senkrechte Linie) zeigt, wo strukturell die Komponenten der Modellierung (vgl. Kapitel 6) und der Implementierung (vgl. Kapitel 7.1 - 7.3) zu sehen sind. Hier sind zusätzlich die Kommunikationsbeziehungen eingetragen, die die Zusammenarbeit zwischen der PE, den Dokumentklassen und den Envelopes realisieren.

7.4.1.2 Aufbau der generierten Dokumentklassen

Zentrale Komponente der Architektur sind die Dokumentklassen bzw. zur Laufzeit ihre Instanzen. Dabei ist es wesentlich, daß sie erstens generiert werden und dadurch die manuelle Programmierung solcher Anbindungskomponenten unnötig machen und zweitens trotzdem eine gute Anpassung an eine konkrete, flexibel anpaßbare Umge-

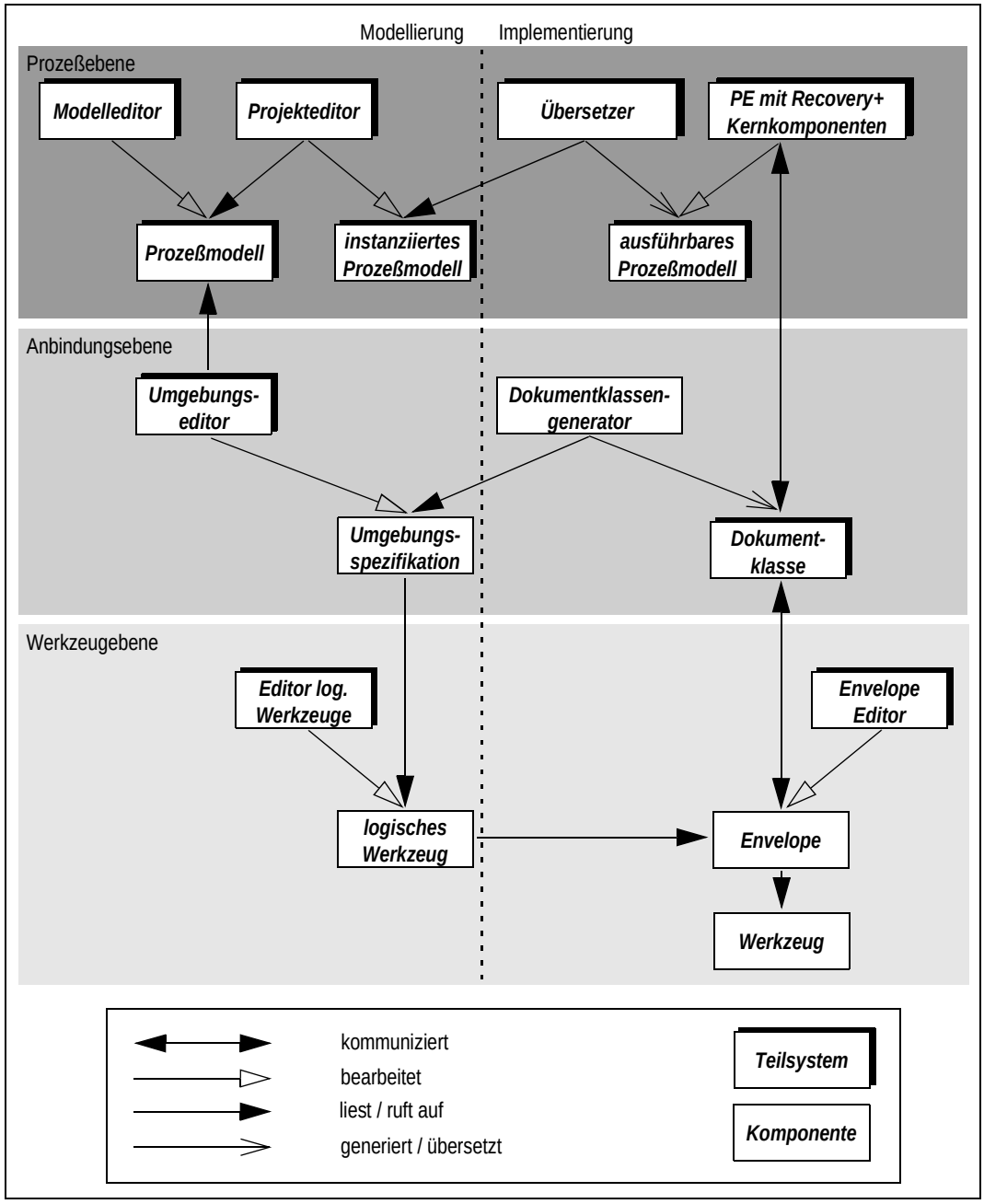


Abbildung 7.17 Komponenten der TIC-Rahmenarchitektur - Gesamtzusammenhang

bung erlauben. Die Generierung hat wiederum die Vorteile reduzierten Aufwands bei gleichzeitig reduzierten Fehlermöglichkeiten. In den Kapiteln 7.1 - 7.3 sind die Funktionsblöcke, die in einer Dokumentklasse abgehandelt werden, beschrieben worden. Diese müssen sich im Aufbau der erzeugten Dokumentklassen nun wiederfinden.

In Kapitel 7.2.5 sind die Aufgaben der Dokumentklassen identifiziert worden. Hauptaufgabe ist die Vermittlung von Aufrufen von Aktivitäten im Prozeßverlauf an die Werkzeuge, die die jeweiligen Aktivitäten realisieren. Dazu werden Komponenten erzeugt, die sich um a) die konkrete Realisierung der Kommunikation in Richtung PE, b) die konkrete Realisierung der Kommunikation in Richtung Envelope und c) das in der Umgebungsspezifikation beschriebene Mapping von PE-Aufruf auf Envelope-Schnittstelle kümmern. Auf dem Rückweg bei der Beendigung eines Werkzeugs ist

wiederum die Abbildung der Statusmeldung des Werkzeugs auf einen gültigen Status des Prozeßmodells zu leisten.

Wird eine Nachricht von der PE an ein Werkzeug gesendet, so wird diese zunächst durch das entsprechende Modul (*Kommunikation PE-Tool*) angenommen um festzustellen, nach welchem Protokoll (vgl. Kapitel 7.2.1) die Kommunikation durchgeführt werden muß. Dazu wird im Modul *Werkzeugtyp-Erkennung* gespeichert, um welchen Werkzeugtyp es sich handelt. Gesetzt wird diese Information beim Empfangen der Antwort auf den Aufruf des Werkzeugs. Der Aufruf selbst wird aus dem Modul `Envelope-Aufruf` heraus gesteuert, da zu diesem Zeitpunkt die Zuordnung zu einem Protokoll noch nicht möglich ist.

Zu einem späteren Zeitpunkt in der Kommunikation kann ein Protokoll ausgewählt werden, und die Nachricht wird an das entsprechende Protokoll weitergegeben. Die Protokoll-Module realisieren Zustandsautomaten, so daß ein Anbindungsobjekt über die gesamte Dauer einer Kommunikation existiert und nicht zwischenzeitlich abgebaut werden kann. Nun wird über das Modul *Kommunikation Anbindung Envelope* eine geeignete Nachricht an das Werkzeug weitergesendet. Das Modul `Feedback-Erzeugung` wird vom Black-Box-Protokoll aus aufgerufen, wenn die Beendigung des Werkzeugs durch dessen Envelope gemeldet wird. Dabei werden vorgenommene Änderungen interaktiv beim Benutzer erfragt. Das Status-Mapping schließt dann die Abläufe zur Realisierung einer Aktivität ab. Im gleichnamigen Modul wird entweder aus der Meldung des Envelopes oder aber durch Nachfrage beim Benutzer der Folgestatus des bearbeiteten Dokuments bestimmt und an die PE zurückgeliefert.

Insgesamt stellt sich die Struktur einer Dokumentklasse wie in Abbildung 7.18 dar.

7.4.2 Architektur für Merlin

In diesem Kapitel wird die vorgestellte Rahmenarchitektur beispielhaft auf Merlin angewendet.

Im Grundsatz bleibt die alte Merlin-Architektur erhalten (vgl. Kapitel 5.2.5). Das ESCAPE-Modell, das mit einem geeigneten Editor (wie ProMotor, vgl. [BH95]) entworfen und dann instanziiert wurde, wird in eine für Merlin ausführbare Darstellung (ein Projekt) übersetzt und im Datenbanksystem abgelegt. Das Projekt wird als Basis der Modellausführung für Merlin verwendet. Während die alte Merlin-Architektur im Instanziierungsschritt die Werkzeuge festlegte, die die Aktivitäten realisieren sollten, wird dies nun gesondert in der *Umgebungsinstanziierung* vorgenommen. Die Umgebungsinstanziierung besteht im wesentlichen aus der Formulierung der Umgebungsspezifikation. In der Abbildung 7.19 ist die neue Merlin-Architektur dargestellt. Der in Kapitel 5.2.5 in der Architektur enthaltene Version-Browser ist zur Vereinfachung der Darstellung in dieser Abbildung nicht enthalten, ist aber weiterhin Teil der Architektur.

Die neu hinzugekommenen Teile sind in der Abbildung dunkelgrau hinterlegt. Veränderungen der alten Architektur wurden hellgrau markiert, und die unverändert übernommenen Teile sind weiß dargestellt. Deutlich zu sehen ist die Ergänzung der Architektur um den Anteil, der die Umgebung explizit spezifiziert, und die aus der Spezifikation generierten Dokumentklassen. Im Gegensatz zur alten Merlin-Architektur ist der Komplex der Werkzeuganbindung aus den Kernkomponenten herausgezogen worden und wird nun durch die Dokumentklassen realisiert.

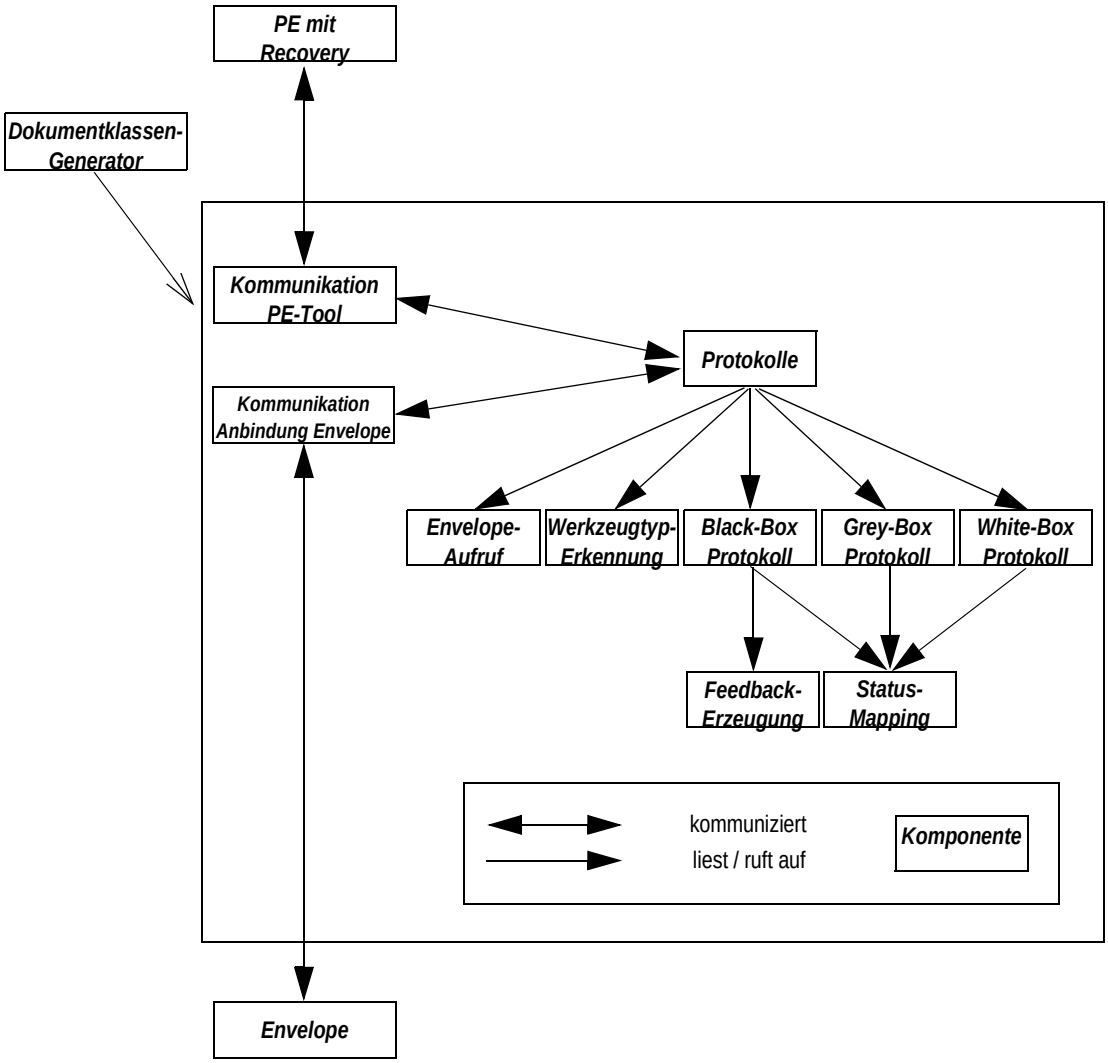


Abbildung 7.18 Die Komponenten einer Dokumentklasse in der TIC-Rahmenarchitektur

Die Zweiteilung von Merlin in Process Engine und Front-End Komponenten, z.B. Work Bench und Working Context als Kernkomponenten der Umgebung, bleibt erhalten. Um diese Komponenten herum werden die Werkzeuge zur vollständigen Merlin-PSEU gruppiert, die sich dadurch von Projekt zu Projekt unterscheiden kann.

Die Anbindung der Werkzeuge wird in den Dokumentklassen realisiert. Die Flexibilität bei der Einbindung verschiedener Werkzeugarten ist durch die Envelopes gewährleistet. Sind die Envelopes nur eine Kapselung von UNIX-Kommandozeilen-Werkzeugen, so erhält der Entwickler nach außen zunächst das gleiche Bild der Umgebung wie in der alten Merlin-Architektur. Erkennbar wird der Übergang auch in diesem Szenario, wenn es um die Erzeugung von Feedback für die PE geht. Hier werden die Entwickler wie beschrieben in den Feedbackprozeß mit eingebunden, wodurch sich der Übergang auf TIC auch nach außen sichtbar niederschlägt.

Wird ein Werkzeug eingebunden, das weiterreichende Feedbackmöglichkeiten als ein UNIX-Kommandozeilen-Werkzeug hat, so kann es problemlos integriert werden, ohne

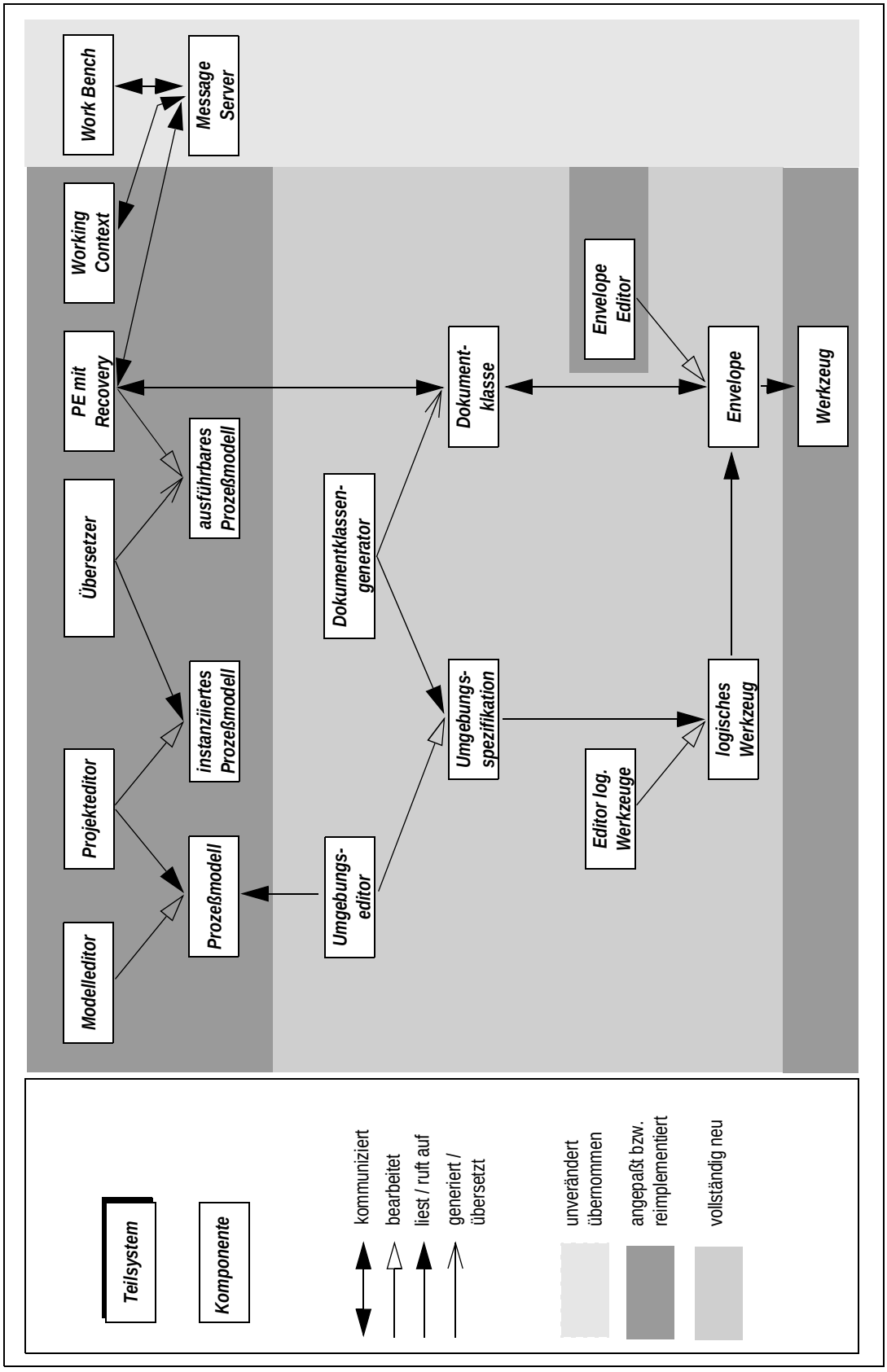


Abbildung 7.19 Merlin-Architektur nach Anpassung an TIC

daß die Implementierung der PE oder einer sonstigen Kernkomponente angepaßt werden muß.

7.5 Zusammenfassung der Implementierung

In Kapitel 7 wurde gezeigt, wie die Konzepte zur Spezifikation des Prozeßmodells und der Instanziierung der Aktivitäten in der Modell- bzw. Prozeßausführung genutzt werden, um eine flexible, heterogene PSEU zusammenzusetzen. Die Konzepte wurden insbesondere so ausgerichtet, daß die in Kapitel 4 gestellten Anforderungen erfüllt werden.

Die Anforderung zur *Übermittlung von Feedback* und die Anforderung zur *Übermittlung von Zustandsinformation* werden beide dadurch sichergestellt, daß die Anbindungsobjekte bei Bedarf interaktiv beim Benutzer die Informationen, die für das PE Feedback notwendig sind, erfragen und das Feedback sicherstellen. Zur Sicherung des Feedbacks werden auch die drei in Kapitel 7.2.1 eingeführten Protokolle zwischen den Werkzeugen und den Anbindungsobjekten unterschieden.

Werden in einer Feedback-Meldung an die PE nicht erlaubte Änderungen übermittelt, so werden die betroffenen Dokumente gekennzeichnet, und der Prozeß wird erst nach einer Korrektur wieder normal fortgesetzt. Damit ist die Anforderung zur *Auswertung von Feedback* erfüllt. Die Anforderung zur *Kontrolle des Leistungsumfangs* wird dadurch abgedeckt, daß die Aktivitäten eingeschränkt werden, die einem Entwickler angeboten werden. Die Einschränkung wird auf der Basis der Zugriffe vorgenommen (siehe Kapitel 7.2.3.2). Weitere Überprüfungen in diesem Sinne sind denkbar.

Der Ausfall eines Werkzeugs läßt sich durch die Trennung von Anbindungsobjekten, Envelopes und Werkzeugen und durch Loggingmaßnahmen feststellen, wodurch die Anforderung zur *Erkennung von Werkzeugabbrüchen* erfüllt ist. Die Folgen von Laufzeitunterbrechungen, die in der Anforderung zur *Zustandsbestimmung nach Werkzeugabbruch* und der Anforderung zur *Zustandsbestimmung nach PE-Abbruch* angesprochen werden, werden durch ein spezielles Recovery-Konzept behandelt.

Die Werkzeuganbindung basiert auf einer Architektur, in der die Schnittstellen der Envelopes zu denen der logischen Werkzeuge passen müssen. Sowohl Envelopes als auch logische Werkzeuge können unabhängig von den konkreten Prozeßmodellen entwickelt werden. Dadurch ist sichergestellt, daß zum einen alle Werkzeuge eingebunden werden können, deren Schnittstelle sich auf das definierte CORBA-Protokoll abbilden läßt (Anforderung zur *Flexibilität der Werkzeugwahl*), und zum anderen die Werkzeuganbindung unabhängig vom konkreten Prozeßmodell wiederverwendet werden kann (Anforderung zur *Wiederverwendbarkeit der Anbindung*). Die Parametrisierung der Werkzeuge ist durch die Definition von Primär- und Sekundäreingaben und durch die Konzepte, um Referenzen auf die Dokumente der Prozeßausführung zu erzeugen, sichergestellt. Dies erfüllt die Anforderung zur *Spezifikation der prozeßunabhängigen Eingaben*.

Durch die Anwendung der Konzepte aus Kapitel 6 und Kapitel 7 sind die Anforderungen aus dem Kapitel 4 für die an TIC angepasste Merlin-Version abgedeckt (vgl. Abbildung 7.20). Die in Kapitel 5 für das Beispiel ESCAPE+ und Merlin identifizierten Probleme sind damit beseitigt.

Konzeptebenen	*Spezifikation der Signatur*	*Relation der Signaturen*	*Spezifikation des Leistungsumfangs*	*Abbildung des Leistungsumfangs*	*Implementierung modellierter Aggregationen*	*Spezifikation des Zustandsmodells*	*Abbildung des Zustandsmodells*	*Spezifikation der prozeßunabhängigen Eingaben*	*Übermittlung von Feedback*	*Übermittlung von Zustandsinformation*	*Auswertung von Feedback*	*Kontrolle des Leistungsumfangs*	*Zustandsbestimmung nach Werkzeugabbruch*	*Zustandsbestimmung nach PE-Abbruch*	*Flexibilität der Werkzeugwahl*	*Wiederverwendbarkeit der Anbindung*	*Erkennung von Werkzeugabbrüchen*
ESCAPE+ und Umgebungs-spezifikation	+	+	+	+	O	+	+	+									
Merlin^{TIC} mit Werkzeuganbin-dung					O				+	+	+	+	+	+	+	+	+
zusammen-gefaßt	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+

+: Anforderung wird vollständig erfüllt
-: Anforderung ist nicht betrachtet (nur in der Zeile „zusammengefaßt“)
O: Ebene liefert Beitrag zur Erfüllung der Anforderung, Anforderung nicht vollständig abgedeckt
kein Eintrag: kein Beitrag zu dieser Anforderung (nicht in der Zeile „zusammengefaßt“)

Abbildung 7.20 Abgleich mit den Anforderungen

8

ZUSAMMENFASSUNG

8.1 Zusammenfassung der Ergebnisse

Sollen PSEU in der Wahl der verwendeten Entwicklungswerkzeuge flexibel sein, ohne die Prozeßsteuerungsaufgabe aufzugeben, so müssen die eingesetzten Werkzeuge mit der Prozeßsteuerungskomponente integriert werden. In dieser Arbeit wurde diskutiert, wie PSEU so flexibel aus Werkzeugen zusammengestellt werden können, daß die Integration der Werkzeuge durchführbar wird, ohne dabei die Werkzeuge selbst anzupassen oder die Abstimmung mit der Prozeßmaschine aufzugeben.

Als Ergebnis einer detaillierten Untersuchung der zur Durchführung der Prozeßsteuerungsaufgabe nötigen Funktionalitäten wurde ein Katalog mit Anforderungen entwikkelt, die an eine PSEU und die darin ausgeführte PML gestellt werden, um eine heterogene Werkzeuglandschaft an eine Prozeßsteuerung anzubinden. Darauf basierend wurde das Tool Integration Concept (TIC) definiert, ein Rahmen von Konzepten, deren Implementierung die Erfüllung der Anforderungen sicherstellt. Weiterhin ist eine allgemeine Architektur als Teil von TIC entwickelt worden, so daß sich die Anpassung oder auch die Neuentwicklung einer PSEU an diesen Vorgaben orientieren kann. Wie die in dieser Arbeit entwickelten Konzepte zusammenhängen und in wel-

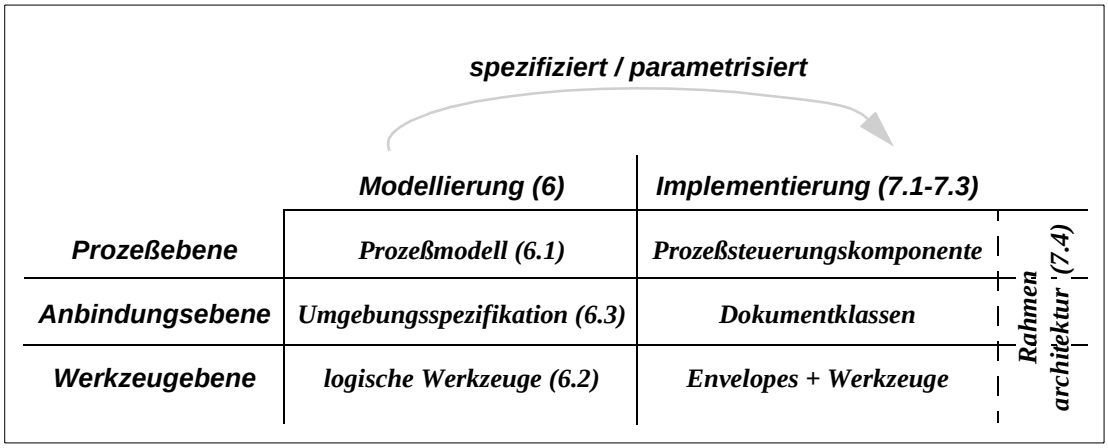


Abbildung 8.1 Spezifikation und Realisierung einer anwendungsspezifischen PSEU mit TIC

chen Kapiteln sie entwickelt wurden, ist in Abbildung 8.1 dargestellt. Eine vollständige Übersicht, welche Anforderungen in welchen Kapiteln definiert und erfüllt werden, ist in Abbildung 8.2 am Schluß dieses Kapitels zu finden.

Im einzelnen zeigt Abbildung 8.1, daß die in TIC eingeführten Modellierungskonzepte auf drei Ebenen verteilt sind. Auf der ersten Ebene liegt die Modellierung im *Prozeßmodell*. Hier werden Modelle erstellt, mit denen insbesondere das Verhalten der Aktivitäten spezifiziert werden kann. Auf der dritten Ebene werden *logische Werkzeuge* als neues Konzept eingeführt. Sie beschreiben die Funktionalität der Werkzeuge so, daß ein Abgleich mit den Aktivitäten möglich wird. Ebene zwei bildet das Bindeglied zwischen Ebene eins und Ebene drei. Hier wird eine Abbildung von den Aktivitäten auf die logischen Werkzeuge angegeben, die *Umgebungsspezifikation*. Weiterhin wurde eine allgemeine *Architektur* für eine PSEU definiert, deren Komponenten die Modellierungsanteile der drei Ebenen verwenden, um die Integration der Werkzeuge in eine PSEU zu realisieren.

Auf der Modellierungsebene geht es zunächst darum, die Aktvitäten brauchbar zu spezifizieren. Zu den wichtigsten Konzepten, die an dieser Stelle neu eingeführt wurden, gehören die *Zugriffe auf Dokumente und Beziehungen*, die durch *Zugriffspfade, Zugriffstypen* und *Multiplizitäten* erweitert werden. Außerdem wurden Konzepte zur *Fortpflanzung von Aktivitäten* auf Aggregationen ergänzt. Mit den Zugriffen auf Dokumente läßt sich zunächst einmal angeben, auf welche Dokumente eine Aktivität Auswirkungen hat. Wird auf verschiedene Dokumente durch die gleiche Aktivität zugegriffen, dann wird die Abhängigkeit zwischen den Dokumenten explizit in den *Zugriffspfaden* spezifiziert. Weiterhin wurden zur Spezifikation von Zugriffen *Zugriffstypen* definiert, auf deren Basis es möglich ist, zu entscheiden, inwieweit eine Aktivität prozeßrelevante Informationen erzeugt und Feedback nötig ist. Die Verwendung der Zugriffstypen, insbesondere CREATES und DELETES, unterstützt zudem ein vertieftes Verständnis des vom Prozeßmodellierer vorgesehenen Prozeßablaufs und erlaubt damit auch Kontrollen über die Korrektheit des Prozeßmodells. Diese Konzepte werden durch die *Zugriffe auf Beziehungen* ergänzt. Mit ihnen ist es möglich, Beziehungen zwischen Dokumenten zur Laufzeit hinzuzufügen oder zu löschen. Mit den *Multiplizitäten*, die zu einem Zugriff angegeben werden, ist es möglich, die Menge der durch einen Zugriff betroffenen Dokumente oder Beziehungen genauer zu beschreiben.

Da die Daten, auf die von einer Aktivität zugegriffen wird, häufig strukturiert vorliegen, wurde gefordert, daß sich Aggregationen modellieren lassen. Greifen Aktivitäten auf Aggregationen zu, so kann sich dies bei verschiedenen Aggregationstypen verschieden auswirken. Um diese Unterschiede modellieren zu können, wurden die *aggregatbestimmte Fortpflanzung* und die *komponentenbestimmte Fortpflanzung* eingeführt.

Ein Problem bereits existierender PML/PSEU ist, daß unabhängig vom Grad der Spezifikation der Aktivitäten in der jeweiligen PML keine Spezifikation der einzusetzenden Werkzeuge vorlag, aufgrund derer entschieden werden konnte, ob ein Werkzeug als Implementierung einer Aktivität geeignet ist oder nicht. Eine solche Spezifikation wurde durch die *logischen Werkzeuge* geschaffen.

Das Verbindungselement zwischen Aktivitäten und Entwicklungswerkzeugen bildet die *Umgebungsspezifikation*. In der Umgebungsspezifikation wird eine Abbildung der spezifizierten Aktivitäten auf die spezifizierten logischen Werkzeuge vorgenommen,

wodurch die zur Prozeßsteuerung notwendige Verbindung zwischen Aktivitäten und Werkzeugen hergestellt wird. Um die Integration von Werkzeugen in eine PSEU sicherstellen zu können, ist keine vollständige Spezifikation der Semantik von Aktivitäten oder Werkzeugen notwendig. Die Spezifikation der Aktivitäten und der Werkzeuge ist in diesem Sinn nicht vollständig, sondern stellt „nur“ die Integration der Werkzeuge sicher.

Die Implementierung der Konzepte basiert darauf, daß aus der Umgebungsspezifikation die *Dokumentklassen* generiert werden. Die Instanzen der Dokumentklassen, die *Anbindungsobjekte*, dienen als Vermittler zwischen der Prozeßsteuerungskomponente und den Werkzeugen. Die Anbindungsobjekte übernehmen weiterhin die Aufgabe, Informationen bezüglich prozeßrelevanter Daten und Statusveränderungen der Dokumente bei Bedarf nachzufragen und an die Prozeßsteuerungskomponente weiterzugeben. Dazu sind drei *Werkzeugtypen* identifiziert worden, denen jeweils ein spezielles *Protokoll* zur Kommunikation mit den Anbindungsobjekten zugeordnet wurde. Auf dieser Basis können die Anbindungsobjekte bei Bedarf weitere Informationen erfragen. Dadurch wurde sichergestellt, daß eine Prozeßsteuerungskomponente genügend Informationen erhält, um ihre Aufgaben erfüllen zu können.

Die Kommunikation mit den (nicht angepaßten) Werkzeugen wird durch die Kapselung in *Envelopes* sichergestellt. Diese sind speziell auf das jeweilige Werkzeug angepaßt. Die Anpassung muß jedoch nur ein einziges Mal erfolgen und kann dann in weiteren Umgebungen wiederverwendet werden. Die Schnittstelle jedes Envelopes wird durch ein logisches Werkzeug definiert. Implementieren zwei Envelopes das gleiche logische Werkzeug, so ist es nun möglich, sie ohne weitere Änderungen der Umgebung gegeneinander auszutauschen.

In einer realen Umgebung kann es immer zu unvorhergesehenen Ausfällen (z.B. durch Stromausfall) kommen. Für diesen Fall wurden *Recoverymechanismen* eingeführt, mit denen es möglich ist, nach Ausfällen von Werkzeugen oder der Prozeßsteuerungskomponent ein laufendes Projekt korrekt wieder aufzunehmen.

Die eingeführten Konzepte sind für die PML/PSEU-Kombination ESCAPE/Merlin beispielhaft realisiert worden.

8.2 Offene Punkte und Erweiterungsoptionen

Die Implementierung der Konzepte für das Beispiel ESCAPE/Merlin verdeutlichte, daß notwendige Änderungen in den PSEU-spezifischen Anteilen speziell auf jede Zielumgebung zugeschnitten werden. Damit sind zwar die Konzepte vorgegeben, die konkreten Implementierungen unterscheiden sich aber. Hier wäre es wünschenswert zu überprüfen, welche Bedingungen erfüllt sein müssen, damit Implementierungen der TIC-Konzepte einfacher übernommen werden können.

Eine weitergehende Unterstützung kann durch eine Umgebung zur Erstellung von Umgebungsspezifikationen und logischen Werkzeugen erreicht werden. Eine solche Umgebung hätte dann mindestens die Aufgabe, die Konsistenz zwischen den formulierten Aufrufen in der Umgebungsspezifikation und den tatsächlich vorhandenen logischen Werkzeugen sicherzustellen.

Ein Aspekt, der über diese Arbeit hinausgeht, sind die Auswirkungen einer Schemaevolution, d.h. einer Veränderung des Modells während der Durchführungszeit eines Projekts, auf eine existierende Umgebungsspezifikation. Vereinfachend kann sicher angenommen werden, daß ein Projekt in der Regel auf der Basis des bei seinem Start verwendeten Prozesses durchgeführt wird. Allerdings ist dies eine Bedingung, die nicht hält, wenn z.B. projektbegleitend Optimierungsmaßnahmen durchgeführt werden. Es wäre dann zu klären, welche Konsequenzen sich für die Spezifikation der Umgebung ergeben, falls solche Änderungen ermöglicht werden sollen.

Weitere Bereiche zur Klärung ergeben sich im Umfeld der Werkzeuge. Bei diesen wurde in dieser Arbeit angenommen, daß es sich generell um nicht vorbereitete Werkzeuge handelt. Sollen nun Werkzeuge so gebaut werden, daß sie die TIC-Konzepte direkt unterstützen, so muß geklärt werden, wie die Architektur solcher Werkzeuge aussehen müßte.

Wird diese Überlegung weitergeführt, so stellt sich die Frage, ob die Werkzeuge auch eine abgespeckte Prozeßmaschine zur Steuerung ihres Verhaltens (z.B. zur Einhaltung von Zugriffsrechten) enthalten könnten. In einem Konzept offener Umgebungen macht dies aber nur Sinn, wenn sich daraus, bezogen auf einen Einzelprozeß, keine proprietären Werkzeuge ergeben, sondern z.B. beim Start der Werkzeuge ein Stück Prozeßmodell mit der Prozeßmaschine zur Interpretation im Werkzeug ausgetauscht wird. Sinn eines solchen Vorgehens wäre es, die zwischen der Prozeßsteuerungskomponente und den Werkzeugen stattfindende Kommunikation zu reduzieren.

8.3 Bewertung der Ergebnisse

Ziel dieser Arbeit ist es, die Integration von Werkzeugen in eine heterogene PSEU so zu ermöglichen, daß die Erstellung und Anpassung von flexiblen, heterogenen PSEU unterstützt und vereinfacht wird. Dieses Ziel wurde erreicht, indem

- ein Anforderungsprofil an eine heterogene PSEU, die die Einhaltung von Entwicklungsprozessen gewährleistet und die Koordination großer Entwicklungsteams unterstützt, erstellt wurde,
- ein Satz von Konzepten entwickelt wurde, der die Einhaltung dieser Anforderungen sicherstellt,
- eine Architektur aufgebaut wurde, die durch gezielten Einsatz von Spezifikationselementen und Automatismen (z.B. Generierung von PSEU-Teilen) eine im Vorbereitungsaufwand möglichst günstige, aber trotzdem klar definierte Umgebungserstellung erlaubt.

Der Aufwand bei der Erstellung einer TIC-basierten PSEU ist sicher nicht zu unterschätzen. Zusätzlicher Aufwand entsteht durch die Anpassung einer aktuell existierenden PSEU sowie durch die erweiterte Spezifikation. Die Vorteile dieser Lösung sind, im Verhältnis zu den in der Arbeit aufgezeigten Problemen ohne ein solches Vorgehen, unter den in Kapitel 2.2 genannten Rahmenbedingungen aber so bedeutend, daß sie den zusätzlichen Aufwand rechtfertigen.

Diese Arbeit leistet einen wichtigen Beitrag dazu, den Einsatz prozeßgesteuerter Software-Entwicklungsumgebungen praxisrelevanter zu gestalten.

Abbildung 8.2 Abgleich Anforderungsdefinition und Anforderungslösung

LITERATUR

[ABGM92] P. Armenise, S. Bandinelli, C. Ghezzi und A. Morzenti. Software Processes Representation Languages: Survey and Assessment. In *Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering*, Seite 455–462, Los Alamitos, California, 1992. IEEE Computer Society Press. Abgedruckt in Garg und Jazayeri [GJ96].

[ACF95] V. Ambriola, G. A. Cignoni und Ch. Fernström. Current Issues on Integration. In Schäfer [Sch95], Seite 197–199. Nordwijkerhout, Niederlande, April 1995.

[Bar92] N. S. Barghouti. Concurrency Control in Rule-Based Software Development Environments. Technical Report CUCS-001-92, Columbia University, 1992.

[BBFL94] S. Bandinelli, M. Braga, A. Fuggetta und L. Lavazza. The Architecture of the SPADE-1 Process–Centered SEE. In Warboys [War94], Seite 15–30. LNCS 772.

[Ber92] A.-J. Berre. COOP — An Object Oriented Framework for Systems Integration. In P. A. Ng, C. V. Ramamoorthy, L. C. Seifert und R. T. Yeh, Hrsg., *Proceedings of the Second International Conference on Systems Integration*, Seite 104–113, Morristown, New Jersey, Juni 1992. IEEE, IEEE Computer Society Press, Los Alamitos, California.

[BESS96] N. Barghouti, W. Emmerich, W. Schäfer und A. Skarra. Information Management in Process-Centered Software Engineering Environments. In Fuggetta und Wolf [FW96], Kapitel 3, Seite 53–87.

[BFG93] S. Bandinelli, A. Fuggetta und S. Grigolli. Process Modeling in-the-large with SLANG. In *IEEE Transactions on Software Engineering*, 12(Dezember 1993):1128–1144, 1993. IEEE Computer Society Press.

[BFW92] A. W. Brown, P. H. Feiler und K. C. Wallnau. Understanding Integration in a Software Development Environment. Technical Report CMU/SEI-91-TR-31, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, Januar 1992.

[BGR+94] R. F. Bruynooghe, R. M. Greenwood, I. Robertson, J. Sa und B. C. Warboys. PADM: Towards a Total Process Modelling System. In Finkelstein et al. [FKN94], Kapitel 12, Seite 293–334.

[BH95] S. Berger und J. Hohmann. ProMotor - Entwurf und Implementierung eines Editors zur graphischen Eingabe von Software-Prozeßmodellen. Diplomarbeit, Universität Dortmund, Fachbereich Informatik. 1995.

[BM91] A. W. Brown und J. A. McDermid. On Integration and Reuse in a Software Development Environment. In Long [Lon91], Seite 171–194.

[BM92] A. W. Brown und J. A. McDermid. Learning from IPSE's Mistakes. *IEEE Software*, Seite 23–28, März 1992.

[BMZ+93] A. W. Brown, E. J. Morris, Paul F. Zarella, F. W. Long und W. M. Caldwell. Experiences with a Federated Environment Testbed. In I. Sommerville und M. Paul, Hrsg., *Software Engineering — ESEC '93*. Springer-Verlag, September 1993. LNCS 717, 4th European Software Engineering Conference, Garmisch–Partenkirchen, Deutschland, 13.-17. September, 1993.

[Bro92] A. W. Brown. Control Integration Through Message Passing in a Software Development Environment. Technical Report CMU/SEI-92-TR-35, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, Dezember 1992.

[BS86] R. Bahlke und G. Snelting. The PSG System — From Language Definitions to Interactive Programming Environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.

[BWF93] A. W. Brown, K. C. Wallnau und P. H. Feiler. Understanding Integration in a Software Development Environment: Issues and Illustrations. *Journal of Systems Integration*, 3(3/4):303–329, September 1993.

[Cag90] M. R. Cagan. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, 41(3):36–47, Juni 1990.

[CDI92] EIA CDIF, Technical Committee. Introduction to CDIF - The CASE Data Interchange Format Standards, April 1992.

[Che76] P. P.-S. Chen. The Entity–Relationship Model — Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[CLJ91] R. Conradi, C. Liu und M. L. Jaccheri. Process Modeling Paradigms: An Evaluation. In *Proceedings of the 7th International Software Process Workshop*, Seite 51–53, Los Alamitos, California, Oktober 1991. IEEE, IEEE Computer Society Press. Yountville, California, USA.

[Coa96] Workflow Management Coalition. Terminology & Glossary. WFMC Document WFMC-TC-1011, Workflow Management Coalition, Avenue Marcel Thiry 204, 1200 Brussels, Belgien, Juni 1996. Ausgabe 2.0.

[Dei93] W. Deiters. *A View Based Approach to Software Process Management*. Dissertation, Universität Dortmund, 1993. Erschienen als Forschungsbericht Nr. 467/1993, FB Informatik, Universität Dortmund.

[DF94] M. Dowson und Ch. Fernström. Towards Requirements for Enactment Mechanisms. In Warboys [War94], Seite 90–106. LNCS 772.

[DGHKL84] V. Donzeau-Gouge, G. Huet, G. Kahn und B. Lang. Programming Environments Based on Structured Editors: The Mentor Experience. In D. R. Barstow, H. E. Shrobe und E. Sandewall, Hrsg., *Interactive Programming Environments*, Kapitel 7, Seite 128–140. McGraw-Hill, New York, 1984.

[DGSZ94] G. Dinkhoff, V. Gruhn, A. Saalmann und M. Zielonka. Business Process Modeling in the Workflow Management Environment LEU. In P. Loucopoulos, Hrsg. *Proceedings of the 13th International Conference on the Entity-Relationship Approach*, Seite 46–63. Springer Verlag, Dezember 1994. LNCS 881.

[Dow91] M. Dowson, Hrsg. *Proc. of the 1st Int. Conference on the Software Process*. IEEE Computer Society Press, Oktober 1991.

[Dow93] M. Dowson. Software Process Themes and Issues. In *Proc. of the 2nd Int. Conference on the Software Process* [ICS93], Seite 54–62.

[ECMA92] ECMA European Computer Manufacturers Association. Introducing PCTE+, August 1992. (ECMA/TC33/89/48).

[EF96] W. Emmerich und A. Finkelstein. Do Process-Centred Environments Deserve Process-Centred Tools. In C. Montangero, Hrsg., *Proceedings of the 5th European Workshop, EWSPT '96*, Seite 75–81. Springer Verlag, Oktober 1996. Nancy, Frankreich, Oktober 1996.

[EG94] G. Engels und L. Groenewegen. SOCCA: Specifications of Coordinated and Cooperative Activities. In Finkelstein et al. [FKN94], Kapitel 4, Seite 71–101.

[Emm95] W. Emmerich. *Tool Construction for Process-centred Software Development Environments based on Object Database Systems* . Dissertation, Universität–GH Paderborn, FB Mathematik - Informatik, AG Softwaretechnik, D-33095 Paderborn, September 1995.

[ES89] G. Engels und W. Schäfer. *Programmentwicklungsumgebungen, Konzepte und Realisierung*. Stuttgart. Teubner. 1989.

[ES94] W. Emmerich und W. Schäfer. Groupie - An Environment supporting Group-Oriented Architecture Developement. SWT Memo Nr. 71, Internes Memorandum des Lehrstuhls Software-Technologie, Fachbereich Informatik, Universität Dortmund. 1994.

[Fer93b] Ch. Fernström. State models and protocols in process-centred environments. In W. Schäfer, Hrsg., *Proceedings of the 8th International Software Process Workshop*, Seite 72–77, Los Alamitos, California, März 1993. IEEE, IEEE Computer Society Press. Wadern, Germany.

[FH93] P. H. Feiler und W. S. Humphrey. Software Process Development and Enactment: Concepts and Definitions. In *Proc. of the 2nd Int. Conference on the Software Process* [ICS93], Seite 28–40.

[FKN94] A. Finkelstein, J. Kramer und B. Nuseibeh, Hrsg. *Software Process Modelling and Technology*. Advanced Software Development Series. Research Studies Press Ltd., Taunton, Somerset, England, 1994.

[FNO92] Ch. Fernström, K. H. Närfelt und L. Ohlsson. Software Factory Principles, Architecture, and Experiments. *IEEE Software*, Seite 36–44, März 1992.

[FO91] Ch. Fernström und Lennart Ohlsson. Integration Needs in Process Enacted Environments. In Dowson [Dow91], Seite 142–158.

[Fro90] B. D. Fromme. HP Encapsulator: Bridging the Generation Gap. Hewlett-Packard Journal, Juni 1990.

[FS93] R. Fehling und W. Schäfer. OPUS: Konzept und Werkzeug für die Unterstützung verteilter, modularer Software-Entwicklung. In *Proceedings der Softwaretechnik '93*, Seite 73–80. GI-FG 2.1.1, November 1993.

[Fug93] A. Fuggetta. A Classification of CASE Technology. *Computer*, 26(12):25–38, Dezember 1993.

[FW96] A. Fuggetta und A. Wolf, Hrsg. *Software Process*. Trends in Software. Wiley, 1996.

[Ger94] St. Gerle. DoBench Communication Server - Ein Werkzeug für den Nachrichtenaustausch zwischen Entwicklungsumgebungen. Diplomarbeit, Universität Dortmund, Lehrstuhl Softwaretechnik, April 1994.

[GHJV94] E. Gamma, R. Helm, R. Johnson und J. Vlissides. *Design Patterns*. Professional Computing Series. Addison-Wesley, 1994.

[GJ96] P. Garg und M. Jazayeri. Process-Centered Software Engineering Environments: A Grand Tour. In Fuggetta und Wolf [FW96], Kapitel 2, Seite 25–52.

[GK91] M. A. Gisi und G. E. Kaiser. Extending a Tool Integration Language. In Dowson [Dow91], Seite 218–227.

[GM89] C. Ghezzi und J. A. McDermid, Hrsg. *ESEC 89, Proceedings*. Springer Verlag, September 1989. 2nd European Software Engineering Conference, Warwick, 11.-15. Septem-

ber, 1989, LNCS 387.

[Gru91a] V. Gruhn. The Software Process Management Environment MELMAC. In *Proceedings of the 1st European Workshop on Software Process Modeling, Milan, Italy*, Seite 191–201. A.I.C.A. Press, Mai 1991.

[Gru91b] V. Gruhn. Validation and Verification of Software Process Models. Dissertation, Universität Dortmund, Juni 1991. Erschienen als Technischer Bericht 394/91.

[Gru95] V. Gruhn. Business Process Modeling and Workflow Management. *International Journal of Cooperative Information Systems*, 4(2 & 3):145–164, 1995.

[Hew91] Hewlett-Packard. *SoftBench Encapsulator: Programmer's Reference*, 1 (b2606-90001) edition, Dezember 1991.

[HJKW96] P. Heimann, G. Joeris, C.-A. Krapp und B. Westfechtel. DYNAMITE: Dynamic Task Nets for Software Process Management. In *18th International Conference on Software Engineering*, Seite 331–341. IEEE Computer Society Press, März 1996.

[ICS93] *Proc. of the 2nd Int. Conference on the Software Process*. IEEE Computer Society Press, Februar 1993.

[Imb91] M. Imber. The CASE Data Interchange Format (CDIF) standards. In Long [Lon91], Seite 457–474.

[ILU00] B. Janssen, M. Spreitzer, D. Larner und Ch. Jacobi. ILU Reference Manual, ILU Version 2.0. Parc Xerox, 2000.

[JH94] A. M. Julienne und B. Holtz. *ToolTalk & OpenProtocols*. SunSoft, 1994.

[JPSW94] G. Junkermann, B. Peuschel, W. Schäfer und S. Wolf. MERLIN: Supporting Cooperation in Software Development Through a Knowledge–Based Environment. In Finkelstein et al. [FKN94], Kapitel 5, Seite 103–129.

[Jun95] G. Junkermann. *ESCAPE — Eine graphische Sprache zur Spezifikation von Software–Prozessen* . Dissertation, Universität Dortmund, FB Informatik, LS Softwaretechnik, D-44221 Dortmund, September 1995.

[Kel93] U. Kelter. Integrationsrahmen für Software–Entwicklungsumgebungen. *Informatik Spektrum*, 16(5):281–285, Oktober 1993.

[KFP88] G. E. Kaiser, P. H. Feiler und St. S. Popovich. Intelligent Assistance for Software Development and Maintenance. *IEEE Software*, Seite 40–49, 1988.

[KPBS93] G. E. Kaiser, S. Popovich und I. Z. Ben-Shaul. A Bi-Level Language for Software Process Modeling. In *Proceedings of the 15th International Conference on Software Engineering, Baltimore, Maryland*, Seite 132–143. IEEE Press, Mai 1993.

[KR84] B. W. Kerninghan und R. P. Ritchie. *The UNIX Programming Environment*. Prentice-Hall, Englewood Cliffs, 1984.

[Kur97] M. Kurth. *Spezifikation von Integration durch Nachrichten*. Diplomarbeit, Universität-GH Paderborn, Mai 1997.

[Lef95] M. Lefering. *Integrationswerkzeuge in einer Softwareentwicklungsumgebung*. Verlag Shaker, 1995. Dissertation, RWTH Aachen.

[Lon91] F. Long, Hrsg. *Software Engineering Environments: Volume 3*, Aberystwyth, UK, März 1991. Elllis Horwood.

[Lon93] J. Lonchamp. A Structured Conceptual and Terminological Framework for Software Process Engineering. In *Proc. of the 2nd Int. Conference on the Software Process* [ICS93], Seite 41–53.

[Lon94] J. Lonchamp. An Assessment Exercise. In Finkelstein et al. [FKN94], Kapitel 13, Seite 335–356.

[LV89] M. Lacroix und M. Vanhoedenaghe. Tool Integration in an Open Environment. In Ghezzi und McDermid [GM89], Seite 311–323. 2nd European Software Engineering Conference, Warwick, 11.-15. September, 1989. LNCS 387.

[Mar90] C. D. Marlin. The MultiView Integrated Programming Environment Project. University of Adelaide, Februar 1990.

[Mey91] S. Meyers. Difficulties in Integrating Multiview Development Systems. *IEEE Software*, 8(1):49–57, Januar 1991.

[Mon94] C. Montangero. The "Process in the Tool Syndrome": Is it Becoming Worse? In C. Ghezzi, Hrsg., *Proceedings of the 9th International Software Process Workshop*, Seite 53–56, Los Alamitos, California, Oktober 1994. IEEE, IEEE Computer Society Press. Airlie, Virginia, USA.

[Mon96] Carlo Montangero (Ed.): Software Process Technology, 5th European Workshop, EWSPT '96, Nancy, France, 9.-11. Oktober 1996, Tagungsband. Lecture Notes in Computer Science, Vol. 1149, Springer, 1996

[MPMH93] C. Marlin, B. Peuschel, M. McCarthy und J. Harvey. Multiview-Merlin: An experiment in tool integration. In *Proceedings of the Software Engineering Environments Conference 1993*, Seite 35–48, Reading, UK, Juli 1993. IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, California.

[Nag93] M. Nagl. Software–Entwicklungsumgebungen: Einordnung und zukünftige Entwicklungslinien. *Informatik Spektrum*, 16(5):273–280, Oktober 1993.

[Nag96] M. Nagl, Hrsg. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, Band 1170 der *Lecture Notes in Computer Science*. Springer Verlag, Heidelberg, Germany, 1996.

[NE93] NIST and ECMA. Reference Model for Frameworks of Software Engineering Environments. ECMA Technical Report / NIST Special Publication TR/55 / 500-211, ECMA and NIST, Juni 1993. NIST ISEE Working Group and ECMA TC33 Task Group on the Reference Model, Edition 3.

[NSS96] Olaf Neumann, Sabine Sachweh, Wilhelm Schäfer: A High-Level Object-Oriented Specification Language for Configuration Management and Tool Integration. in [Mon96], Seite 137-143.

[OMG97] Object Management Group OMG. *A Discussion of the Object Management Architecture*. OMG, Januar 1997.

[OMG98a] Object Management Group OMG. *CORBAservices: Common Object Services Specification*. OMG, März 1995. Aktualisiert 21. Dezember 1998.

[OMG98b] Object Management Group OMG. Dynamic Invocation Interface. In *The Common Object Request Broker: Architecture and Specification* [OMG98g], Kapitel 5. Revision 2.2.

[OMG98c] Object Management Group OMG. Dynamic Skeleton Interface. In *The Common Object Request Broker: Architecture and Specification* [OMG98g], Kapitel 6. Revision 2.2.

[OMG98d] Object Management Group OMG. Naming Service Specification. In *CORBAservices: Common Object Services Specification* [OMG98a], Kapitel 3.Aktualisiert 21. Dezember 1998.

[OMG98e] Object Management Group OMG. OMG IDL Syntax and Semantics. In *The Common Object Request Broker: Architecture and Specification* [OMG98g], Kapitel 3. Revision 2.2.

[OMG98f] Object Management Group OMG. Persistent Object Service Specification. In *CORBAservices: Common Object Services Specification* [OMG98a], Kapitel 5. Aktualisiert 21. Dezember 1998.

[OMG98g] Object Management Group OMG. *The Common Object Request Broker: Architecture and Specification*. OMG, Juli 1998. Revision 2.2.

[OMG98h] Object Management Group OMG. The Interface Repository. In *The Common Object Request Broker: Architecture and Specification* [OMG98g], Kapitel 8. Revision 2.2.

[Ost87] L. Osterweil. Software Processes are Software too. In *Proceedings of the 9th International Conference on Software Engineering, ICSE 87*, Seite 2–13, New York, N.Y., 1987. ACM Press.

[PB92] A. Plutino und P. Beyssac. PCTE Foundation and CDIF Meta Meta Model: A Comparison. Technical Note ECMA_TC33/TGRM/92/29, ECMA/TC33/93/10, Dezember 1992.

[Poh96] K. Pohl. *Process-Centerd Requirements Engineering*. Advanced Software Development Series. Research Studies Press, Taunton Somerset, England, 1996.

[Rei88] S. P. Reiss. Integration Mechanisms in the FIELD Environment. Technical Report CS-88-18, Brown Unversity, Providence, Rhode Island, Oktober 1988.

[Rei90a] S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, Seite 57–66, Juli 1990.

[Rei90b] S. P. Reiss. Interacting with the FIELD environment. *Software-Practice and Experience*, 20(S1):89–115, Juni 1990.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy und W. Lorenson. *Object Oriented Modeling and Design*. Prentice-Hall, 1991.

[Sac99] S. Sachweh. *KOKOS - Ein kooperatives Konfigurationsmanagementsystem*. Universität-GH Paderborn, 1999. Dissertation, Universität-GH Paderborn.

[SB93] D. Schefstroem und G. van den Broek, Hrsg. *Tool Integration*. Addison-Wesley, 1993.

[Sch95] W. Schäfer, Hrsg. *Proceedings of the 4th European Workshop, EWSPT '95*. Springer Verlag, April 1995. Nordwijkerhout, Niederlande, April 1995.

[Sch97] T. Schmal. Ein Ansatz zum Workflow-Management in Krankenhaus-Informationssystemen. In W. Hasselbring, Hrsg., *Erfolgsfaktor Softwaretechnik für die Entwicklung von Krankenhausinformationssystemen*, Seite 65–73, Münster, Februar 1997. GMDS-Projektgruppe und GI-Arbeitskreis "Methoden und Werkzeuge für das Management von Krankenhausinformationssystemen", Krehl Verlag.

[Som92a] I. Sommerville. Part 3, Programming Techniques and Tools. In *Software Engineering* [Som92b].

[Som92b] I. Sommerville. *Software Engineering*. Addison-Wesley, edition 4, 1992.

[SPA95a] CEFRIEL – Politecnico di Milano. *SLANG Process Modeling Language Reference Manual Version 3.0*, 1995.

[SPA95b] CEFRIEL – Politecnico di Milano. *SPADE-1 Tool Developer Manual Version 3.0*, 1995.

[Sun93] SunPro. *SparcWorks Tutorial*. Sun, 1993. Version 3.0 für Solaris.

[SW94] R. A. Snowdown und B. C. Warboys. An Introduction to Process-Centred Environments. In Finkelstein et al. [FKN94], Kapitel 1, Seite 1–8.

[TN92] I. Thomas und B. A. Nejmeh. Definitions of Tool Integration for Environments. *IEEE Software*, Seite 29–35, März 1992.

[TSY+88] R. N. Taylor, R. W. Selby, M. Young, F. C. Belz, L. A. Clarce, J. C. Wileden, L. Osterweil und A. L. Wolf. Foundations of the Arcadia Environment Architecture. *ACM SIGSOFT Software Engineering Notes*, 13(5):1–13, 1988. Proc. of the 4th ACM SIGSOFT Symposium on Software Development Environments, Irvine, California.

[Val94] G. Valetto. Expanding the Repertoire of Process-based Tool Integration. Technical Report CUCS-004-94, Columbia University, Department of Computer Science, New York,

NY10027, Februar 1994.

[VK95] G. Valetto und G. Kaiser. Enveloping "Persistent" Tools for a Process-Centered Environment. In Schäfer [Sch95], Seite 200–204. Nordwijkerhout, Niederlande, April 1995.

[VL95] A. I. Verkamo und G. Lindén. Problems in Interfacing Tools of Different Development Environments. In *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, Seite 429–437, University of Pittsburgh, Juni 1995. Knowledge Systems Institute.

[Wag95] M. Wagener. Generierung von prozeßgesteuerten Software-Entwicklungs-Werkzeugen. Diplomarbeit, Lehrstuhl Softwaretechnik, Universität Dortmund, November 1995.

[Wal92] K. C. Wallnau. Issues and Techniques of CASE Integration with Configuration Management. Technical Report CMU/SEI-92-TR-5, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, März 1992.

[War94] B. Warboys, Hrsg. *Proceedings of the 3rd European Workshop on Software Process Technology EWSPT 94*, Villard de Lans (Grenoble), Februar 1994. ESPRIT-BRA Promoter, Springer. LNCS 772.

[Was89] A. I. Wassermann. Tool Integration in Software Engineering Environments. In F. Long, Hrsg., *Proceedings of the International Workshop on Environments*, Seite 137–149, Chinon, Frankreich, September 1989. Springer-Verlag. erschienen als Lecture Notes in Computer Science, Vol. 467.

[XML98] W3C XML Working Group. *Extensible Markup Language (XML) 1.0*. REC-xml-19980120, W3C, Februar 1998.

[Yan92] Y. Yang. *Tool Interfaces for Software Development*. PhD Thesis, University of Queensland, Queensland, Australia, Juni 1992.

[ZB93] P. F. Zarrella und A. W. Brown. Replacing the Message Service Component in an Integration Framework. Technical Report CMU/SEI-94-TR-17, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, USA, 1993.

[Zün95] A. Zündorf. *PROgrammierte GRaphErsetzungsSysteme (Spezifikation, Implementierung und Anwendung einer integrierten Entwicklungsumgebung*. Dissertation, RWTH Aachen, Lehrstuhl für Informatik III, 1995.

A

SYNTAX DER UMGEBUNGSSPEZIFIKATIONSPRACHE

In diesem Anhang wird die vollständige Syntax der in Kapitel 6.3 eingeführten Umgebungsspezifikation dargestellt. In Anhang A.1 ist die Syntax in einer EBNF-Notation formuliert.

A.1 Syntax in EBNF

A.1.1 Konventionen und Meta-Symbole der verwendeten EBNF

Die Syntax ist in BNF formuliert. In der BNF werden folgende Meta-Symbole und Konventionen verwendet:

- ::= : Das Symbol trennt die linke und die rechte Seite einer Produktion.

  Beispiel: state-list ::= <state> <state-list-cont>.

- . : Das Symbol schließt jede einzelne Produktion ab.

  Beispiel: state-list ::= <state> <state-list-cont>.

- ‘ ’: Die Symbole begrenzen ein einzelnes von der Grammatik erwartetes Zeichen (Terminalsymbol).

  Beispiel: ‘,’

- Sonstige Terminal-Symbole werden durch Verwendung von Großbuchstaben gekennzeichnet und entsprechen i.d.R. den erwarteten Schlüsselwörtern.

  Beispiel: DOCUMENT TYPE

- < > : Die Symbole begrenzen Non-Terminal-Symbole auf der rechten Seite einer Produktion.

  Beispiel: state-list ::= <state> <state-list-cont>.

- Der Präfix opt- eines Non-Terminals bezeichnet ein an der Verwendungsstelle optionales Symbol.

  Beispiel: method ::= <method-name> ‘(’ <opt-parameter-list> ‘)’.

- Optionale Non-Terminal-Symbole sind immer mit der folgenden Struktur definiert:
  opt-non-terminal ::= /* empty */ | <non-terminal>.
  Beispiel opt-parameter-list ::= /* empty */ | <parameter-list>.
- Der Suffix -list bezeichnet ein Non-Terminal-Symbol, das eine Liste ist.
  Beispiel: parameter-list ::= parameter parameter-list-cont.
- Der Suffix -cont bezeichnet die (optionale) Fortsetzung einer Liste.
  Beispiel: parameter-list ::= parameter parameter-list-cont.
- Listen werden immer aus einem Listenkopf und einer Listenfortsetzung definiert.
  Beispiel: parameter-list ::= parameter parameter-list-cont.

A.1.2 Kontextfreie Syntax

env-spec ::= <document-class>
<opt-document-class>.

opt-document-class::=
/ empty */* **|**
<opt-document-class>.

document-class ::= DOCUMENT CLASS <classname>
<interface-section>
<body-section>.

classname ::= <Identifier>.

interface-section::= INTERFACE
<method-list>
END INTERFACE.

method-list ::= <method>
<method-list-cont>.

method-list-cont ::= / empty */* **|**
<method-list>.

method ::= <method-type>
<method-name> '(' <opt_method_parameter_list> ')'
RETURNS <next-state> ';'.

method-type ::= USER INTERACTIVE **|**
USER AUTO **|**
AUTO.

next-state ::= <state-list> **|**
<generic-state>.

state-list ::= <state>
<state-list-cont>.

state-list-cont ::= / empty */ |*
‘,’ <state>
<state-list-cont>.

state ::= <Identifier>.

method-name ::= <Identifier>.

opt-method-parameter-list ::=
/ empty */ |*
<method-parameter-list>.

method-parameter-list ::=
<method-parameter> <method-parameter-list-cont>.

method-parameter-list-cont :=
/ empty */*
‘,’ <method-parameter-list-cont>.

method-parameter::=<dok-param> | <rel-param>.

dok-param ::= <dok-usage> <parameter-name> ‘:’
<parameter-type>.

dok-usage ::= CREATES |
USES |
UPDATES |
DELETES.

parameter-name::= <Identifier>.

parameter-type ::= <opt-set-descriptor> <Identifier>.

opt-set-descriptor::=/ empty */ |*
<set-descriptor>.

set-descriptor ::= SET OF.

rel-param ::= <rel-usage> <parameter-name> ‘:’
<parameter-type> REL reltype.

rel-usage ::= CONNECTS |
DISCONNECTS.

reltype ::= <Identifier>.

body-section ::= BODY
ALLOWED STRUCTURES ‘{’ <structure-list> ‘}’
<defaults-section>
TOOL CONFIGURATION <tool-configuration-list>
END BODY.

structure-list ::= <structure> <structure-list-cont>.

structure-list-cont ::=
/ empty */ |*
',' <structure> <structure-list-cont>.

defaults-section ::= DEFAULTS TO <structure> ';'.

structure ::= <Identifier>.

tool-configuration-list ::=
<tool-configuration> <tool-configuration-list-cont>.

tool-configuration-list-cont ::=
/ empty */ |*
<tool-configuration> <tool-configuration-list-cont>.

tool-configuration ::=
<method-name> ':' <tool-per-structure-configuration-list>.

tool-per-structure-configuration-list ::=
<tool-per-structure-configuration>
<tool-per-structure-configuration-list-cont>.

tool-per-structure-configuration-list-cont ::=
/ empty */ |*
<tool-per-structure-configuration>
<tool-per-structure-configuration-list-cont>.

tool-per-structure-configuration ::=
STRUCTURE <structure> ':'
<tool-interface>
RETURN TRANSLATION <translation-list>.

translation-list ::= <translation> <translation-list-cont>.

translation-list-cont ::=
/ empty */ |*
','
<translation> <translation-list-cont>.

translation ::= <post-execution-state> '<-' <tool-return-list>.

tool-return-list ::=
<tool-return> <tool-return-list-cont>.

tool-return-list-cont ::=
/ empty */ |*
',' <tool-return> <tool-return-list-cont>.

tool-return ::= <Identifier>.

tool-interface ::= <tool-name> '(' opt-tool-parameter-list ')' ';'.

tool-name ::= <Identifier>.

opt-tool-parameter-list ::=
/ empty */ |*
<tool-parameter-list>.

tool-parameter-list ::=
<tool-parameter> <tool-parameter-list-cont>.

tool-parameter-list-cont :=
/ empty */ |*
',' <tool-parameter-list-cont>.

tool-parameter ::= <parameter-name> ':' <parameter-type>.

post-execution-state ::=
<next-state-inquiry> |
<generic-state> |
<error-state>.

generic-state ::= UNCHANGED.

next-state-inquiry ::=

ASK.

error-state ::= UNKNOWN |
ABORT.

A.2 Kontextsensitive Einschränkungen

In diesem Kapitel sollen textuell kontextsensitive Einschränkungen der in Anhang A.1.2 vorgestellten Syntax beschrieben werden. Dabei können im wesentlichen drei Typen von Einschränkungen unterschieden werden:

a) Einschränkungen, die sich bereits durch Einschränkungen auf dem Prozeßmodell ergeben und somit durch die Umsetzung eines korrekten Prozeßmodells bereits eingehalten werden sollten,

b) Einschränkungen durch Abgleich mit den verwendeten (logischen) Werkzeugen, und

c) Einschränkungen auf der Umgebungsspezifikation selbst.

Die folgenden Einschränkungen sind jeweils einem dieser drei Typen zugehörig.

A.2.1 Einschränkungen durch das Prozeßmodell

Ein Beispiel für Einschränkungen, die sich aus dem Prozeßmodell auf die Umgebungsspezifikation fortsetzen, ist die folgende Regel.

- Methoden des <method-type> AUTO dürfen als <next-state> nicht UNCHANGED eingetragen haben.

  Begründung: Würde eine Aktivität das Ergebnis unchanged liefern, so wäre das Dokument, auf dem die Aktivität aufgerufen wurde, nach der Ausführung der Aktivität im gleichen Zustand wie vor dem Aufruf. Falls es sich um eine automatisch ausgelöste Aktivität handelte, käme es zu einer klassischen Endlosschleife, eine immer wiederkehrende Ausführung der gleichen Aktivität (vgl. [Jun95], Kapitel 7.1.2, Fehler KoM_{F-7})

Da sich diese Einschränkungen auf dem Prozeßmodell selbst ergeben, sind sie in den in [Jun95] angegebenen Einschränkungen enthalten. Entsprechend sollten sie bereits durch den Prozeßmodelleditor überprüft und abgedeckt sein. Sie werden daher hier nicht weiter behandelt.

A.2.2 Einschränkungen durch die logischen Werkzeuge

- <tool-name> muß ein existierendes logisches Werkzeug beschreiben.

  Begründung: Offensichtlich kann es sich nur dann um eine gültige Realisierung einer Aktivität handeln, wenn das dazu zu konfigurierende Tool durch ein gültiges logisches Werkzeug repräsentiert wird.

- Die <tool-parameter-list> muß zu einem logischen Werkzeug passen, das den Namen <tool-name> hat.

  Begründung: Damit ein Mapping der Aktivität auf das logische Werkzeug vorgenommen werden kann, ist es offensichtlich nötig, daß es sich um ein passendes Werkzeug handelt.

- Die <tool-return-list> darf nur Rückgaben des gewählten logischen Werkzeugs enthalten.

A.2.3 Einschränkungen auf der Umgebungsspezifikation

- Jeder <method-name>, der in der <interface-section> beschrieben worden ist, muß auch in der <tool-configuration-list> der <body-section> spezifiziert werden.

  Begründung: Offensichtlich soll jede im Interface eines Dokumenttyps genannte Methode auch realisiert sein. Daher muß sie auch eine Abbildung auf einen Werkzeugaufruf haben.

- Jede <tool-configuration> in der <tool-configuration-list> muß eine <tool-per-structure> in der entsprechenden Liste haben, die als <structure> die in der <default-section> spezifizierte <structure> hat.

  Begründung: Jede Methode muß auf das Dokument, für das sie definiert wurde, anwendbar sein. Da das Dokument standardmäßig in der durch den DEFAULTS TO-Eintrag spezifizierten Datenstruktur abgelegt wird, muß jede Aktivität entsprechend anwendbar sein. Dies kann auch durch eine in der <transformation>-Produktion definerte Übersetzung des default-Formats in das Zielformat sichergestellt sein. Dann darf es sich aber nur um Methoden handeln, die nur lesend auf dem Objekt arbeiten, um sicherstellen zu können, daß die Rücktransformation immer möglich (in diesem Fall = nicht nötig) ist.

- Jeder in der <next-state>-Produktion erwähnte Zustand muß in einer <translation-list> der entsprechenden Methode genau einmal vorkommen. Die Angabe von ASK in der <translation-list> steht semantisch für das Erfragen des Folgezustands nach Beendigung des Werkzeugs und somit für alle in der <next-state>-Produktion erwähnten Folgezustände.

  Begründung: Um das zugrundeliegende Prozeßmodell erfüllen zu können, ist es notwendig, daß jeder vom Prozeßmodell vorgegebene Folgezustand auch erreicht werden kann. Das ist nur möglich, wenn die möglichen Endzustände des aufgerufenen Werkzeugs in irgendeiner Form auf alle vom Prozeßmodell vorgesehenen Folgezustände verteilt werden. Würde ein einzelner Zustand bei dieser Abbildung ausgelassen, so würden die für das Prozeßmodell erfolgten Analysen nicht mehr sicher gelten. Beispielsweise könnten Zyklen dadurch eintreten, daß nur einer von zwei im Prozeßmodell vorgesehen Zuständen zurückgegeben wird.

- Es dürfen ausschließlich in der <next-state>-Produktion genannte Zustände in einer <translation-list> der entsprechenden Methode vorkommen. Ausnahme sind die <next-state-inquiry> und die <error-state> Zustände. Wird die <next-state-inquiry>-Produktion verwendet, so darf kein anderer der in der <next-state>-Produktion genannten Zustände in der gleichen <translation-list> vorkommen. Es können somit zusätzlich nur die Zustände der <error-state>-Produktion eingebracht werden.

  Begründung: Es macht keinen Sinn, andere Zustände an die Prozeßmaschine zurückzugeben als die vom Prozeßmodell vorgegebenen. Daher ist die Liste der Folgezustände, die sinnvoll in einer Abbildung der Rückgabewerte auf Folgezustände vorkommen können, auch genau auf diese beschränkt. Daß dabei auch alle vorkommen müssen, ist bereits in einer früheren Regel festgeschrieben worden. Da das Schlüsselwort ASK semantisch alle vom Prozeßmodell vorgesehenen Statuswerte repräsentiert und jeder Zustandswert nur einmal in der Abbildungsliste auftreten darf, ist auch klar, daß, wenn ASK benutzt wird, zusätzlich nur noch die Zustände UNKNOWN und ABORT verwendet werden können. Dabei kann der Zustand UNKNOWN nur in wenigen Fällen wirklich sinnvoll eingesetzt werden, da er inhärent besagt, daß der Rückgabewert des Werkzeugs nicht spezifiziert war. Eine Benutzung ist also nur dann sinnvoll, wenn während der Spezifikation erreicht werden soll, daß einzelne Rückgabewerte des Werkzeugs die Behandlung eines unbekannten Rückgabewertes erhalten, z.B. wenn das Werkzeug schon zurückliefert, daß eine unbekannte Fehlersituation eingetreten ist.

- <next-state-inquiry> dürfen ausschließlich in <translation-list> von Methoden des <method-type> USER INTERACTIVE vorkommen.

  Begründung: Offensichtlich kann der Folgezustand nur beim Entwickler nachgefragt werden, wenn es sich beim aufgerufenen Werkzeug um ein interaktives Werkzeug handelt, da der Ausführungszeitpunkt bei automatischen Aktivitäten nicht eindeutig vorherbestimmbar ist und das Dokument im Ausgangszustand der automatischen Aktivität nicht unbedingt dem Arbeitsbereich eines Entwicklers zugeordnet ist. Weiterhin ist es ebenso klar, daß der Entwickler nur dann eine Entscheidung treffen kann, wenn er das Ergebnis der Aktivität begutachten kann. Eine Nachfrage nach dem Folgezustand macht also keinen Sinn, wenn es sich um eine benutzerinitiierte, aber nicht wirklich interaktive Aktivität handelt. Bei diesen muß sich der Folgezustand eindeutig auf der Basis des Rückgabewertes des Werkzeugs ermitteln

lassen. Es bleiben nur die benutzerinitiierten interaktiven Aktivitäten übrig, bei denen i.d.R. nicht automatisch entschieden werden kann, welcher Folgezustand erreicht wird, sofern mehrere mögliche Folgezustände spezifiziert wurden.

B

SYNTAX DER SPEZIFIKATION LOGISCHER WERKZEUGE

In diesem Kapitel wird die kontextfreie Syntax der Sprache zur Spezifikation der logischen Werkzeuge vorgestellt.

B.1 Konventionen und Meta-Symbole der verwendeten EBNF

Die Syntax ist in BNF formuliert. In der BNF werden folgende Meta-Symbole und Konventionen verwendet:

- ::= : Das Symbol trennt die linke und die rechte Seite einer Produktion.

  Beispiel: state-list ::= <state> <state-list-cont>.

- . : Das Symbol schließt jede einzelne Produktion ab.

  Beispiel: state-list ::= <state> <state-list-cont>.

- ‘ ’: Die Symbole begrenzen ein einzelnes von der Grammatik erwartetes Zeichen (Terminalsymbol).

  Beispiel: ‘,’

- Sonstige Terminal-Symbole werden durch Verwendung von Großbuchstaben gekennzeichnet und entsprechen i.d.R. den erwarteten Schlüsselwörtern.

  Beispiel: DOCUMENT TYPE

- < > : Die Symbole begrenzen Non-Terminal-Symbole auf der rechten Seite einer Produktion.

  Beispiel: state-list ::= <state> <state-list-cont>.

- Der Präfix opt- eines Non-Terminals bezeichnet ein an der Verwendungsstelle optionales Symbol.

  Beispiel: method ::= <method-name> ‘(’ <opt-parameter-list> ‘)’.

- Optionale Non-Terminal-Symbole sind immer mit der folgenden Struktur definiert:

  opt-non-terminal ::= /* empty */ | <non-terminal>.

Beispiel: opt-parameter-list ::= /* empty */ | <parameter-list>.

- Der Suffix -list bezeichnet ein Non-Terminal-Symbol, das eine Liste ist.
  Beispiel: parameter-list ::= parameter parameter-list-cont.
- Der Suffix -cont bezeichnet die (optionale) Fortsetzung einer Liste.
  Beispiel: parameter-list ::= parameter parameter-list-cont.
- Listen werden immer aus einem Listenkopf und einer Listenfortsetzung definiert.
  Beispiel: parameter-list ::= parameter parameter-list-cont.

B.2 Kontextfreie Syntax

LogicTool ::= LOGIC <tool-name> '('param-list')'
RETURNS '{' <symbolic-return-list> '}'.

tool-name ::= <identifier>.

param-list ::= param opt-param-list-cont.

param ::= doc-param | rel-param | secondary-input.

doc-param ::= <doc-effect> <param-name> ':' <doc-type>.

doc-effect ::= <create-effect> | <other-effect>.

create-effect ::= CREATES <opt-reference-type>.

opt-reference-type::=
/ empty */ |*
<reference-type>.

reference-type ::= AUTO | OPTAUTO | OPTUSER.

other-effect ::= USES | UPDATES | DELETES.

param-name ::= <identifier>.

doc-type ::= <identifier>.

rel-param ::= <rel-effect> <param-name> ':' <doc-type> REL <reltype>.

rel-effect ::= CONNECTS | DISCONNECTS.

rel-type ::= <identifier>.

secondary-input ::= <param-name> ':' secondary-input-type.

secondary-input-type::=
<standard-type> | <enum-type-list>.

standard-type ::= BOOL.

enum-type-list ::= ‘{’ <enum-type> <opt-enum-type-list-cont> ‘}’.

enum-type ::= <identifier>.

opt-enum-type-list-cont::=
/ empty */ |*
‘,’ <enum-type> <opt-enum-type-list-cont>.

opt-param-list-cont::=
/ empty */ |*
‘;’ <param> <opt-param-list-cont>.

symbolic-return-list::=
<symbolic-return-value> <opt-symbolic-return-list-cont>.

symbolic-return-value::=
<identifier>.

opt-symbolic-return-list-cont::=
/ empty */ |*
‘,’<symbolic-return-value> <opt-symbolic-return-list-cont>.

C

MODELLE ZUR BESCHREIBUNG VON SOFTWAREINTEGRATION

Es hat verschiedene Versuche gegeben, Integration im Zusammenhang der Werkzeugintegration zu definieren oder zu beschreiben. In diesem Anhang sollen Modelle betrachtet werden, die zu beschreiben versuchen, was Integration meint, welcher Grad an Integration möglich ist und was die Kosten von Integration sind. Allen Ansätzen gemeinsam ist, daß der Grad der Integration, sofern man davon sprechen will, nicht gemessen werden kann und somit letztendlich subjektiv ist. Die betrachteten Modelle

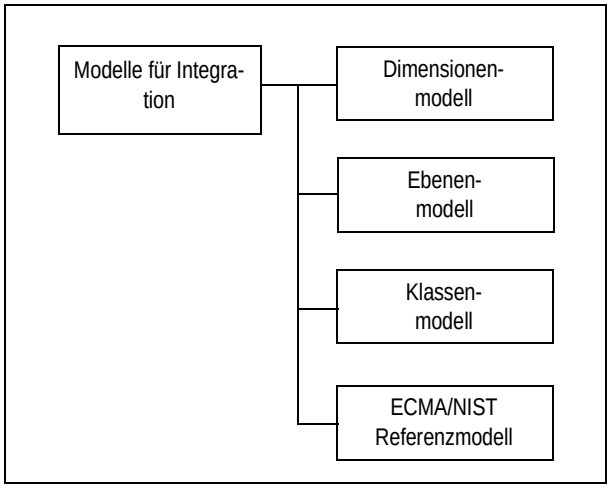


Abbildung C.1 Struktur im Anhang C

und die Struktur des Anhangs sind in Abbildung C.1 dargestellt. Begonnen wird mit dem am weitesten verbreiteten Modell, das Integration in verschiedene Dimensionen unterteilt.

C.1 Dimensionenmodell

Das am weitesten verbreitete Modell wird Wassermann ([Was89]) zugeschrieben und ist von Thomas und Nedjhmeh in [TN92] verfeinert worden. Die meisten Veröffentlichungen, die sich mit Integration beschäftigen, greifen dieses Modell als Basis auf.

Auch in Standardwerken zur Softwaretechnik wird es benutzt (siehe z.B. [Som92a]). Dabei wird sehr selten darauf hingewiesen, daß dieses Modell einige Schwächen aufweist. Aufgrund seiner Verbreitung soll der Besprechung dieses Modells relativ viel Raum eingeräumt werden. Dabei werden dann in Anhang C.1.2 auch die Probleme ausführlich erörtert.

C.1.1 Darstellung des Modells

Das Modell beschreibt Integration von Werkzeugen auf einer abstrakten Ebene und macht keine Vorgaben bezüglich ihrer Realisierung. Es trennt Integration in fünf verschiedene, nach Wassermann zueinander orthogonale, *Dimensionen* auf. Diese Dimensionen sind

1. *Oberflächenintegration*,
2. *Kontrollintegration*,
3. *Datenintegration*,
4. *Prozeßintegration* und
5. *Plattformintegration*.

Im Papier von Wassermann sind diese Begriffe mit sehr wenig Kontextinformation eingeführt. Ein tiefgreifendes Verständnis, was sich im Detail hinter den einzelnen Begriffen verbirgt, wird dadurch erschwert.

Thomas und Nedjmeh greifen das Modell auf und füllen es mit Leben [TN92]. Die Dimension *Plattformintegration* beschreibt, daß verschiedene Betriebssystem- und Hardwareplattformen miteinander eine Form des Austausches von Informationen auf Daten- oder sogar Funktionsebene haben müssen, damit darauf aufbauende Mechanismen diese zur Integration auf der Applikationsebene nutzen können. Allerdings erlaubt eine solche Low-Level-Integration zwar die Verteilung der zu integrierenden Werkzeuge, aber die Problematik der Integration von Werkzeugen ist durch das Angebot der Low-Level-Funktionen nicht einfacher geworden. Weiterhin gilt, daß die Integrationsmechanismen zwischen Werkzeugen im lokalen Bereich in der Regel ebenso im verteilten Fall funktionieren. Erst wenn die Verteilung einer Applikation den Netztyp wechselt, d.h. z.B. vom LAN zum WAN, zieht sie möglicherweise eine Anpassung der Integrationsmechanismen nach sich. Die Autoren stellen aufgrund dieser Überlegungen fest, daß der so definierte Aspekt der Plattformintegration die Problematik der Integration von Werkzeugen zwar beeinflussen kann, aber nicht direkt mit ihr zusammenhängt. Plattformintegration wird daher bei Thomas und Nedjmeh nicht weiter betrachtet.

Für die restlichen vier Bereiche bauen sie die Bedeutung in Form von zugehörigen Teilbereichen aus. Eine Übersicht über die Bereiche und deren Teilbereiche wird in Abbildung C.2 gegeben.

Unter *Oberflächenintegration* ist danach eine sowohl in Optik, Bedienungsphilosophie als auch der Eingabe globaler Daten durchgängige Oberfläche zu verstehen. Der einfachste Grad von Oberflächenintegration wird erreicht, wenn bei verschiedenen Werkzeugen ein gemeinsamer Standard für das Aussehen der Oberfläche verwendet wird, z.B. Motif für X-Window Applikationen. Etwas weiter gehen Werkzeuge, die die glei-

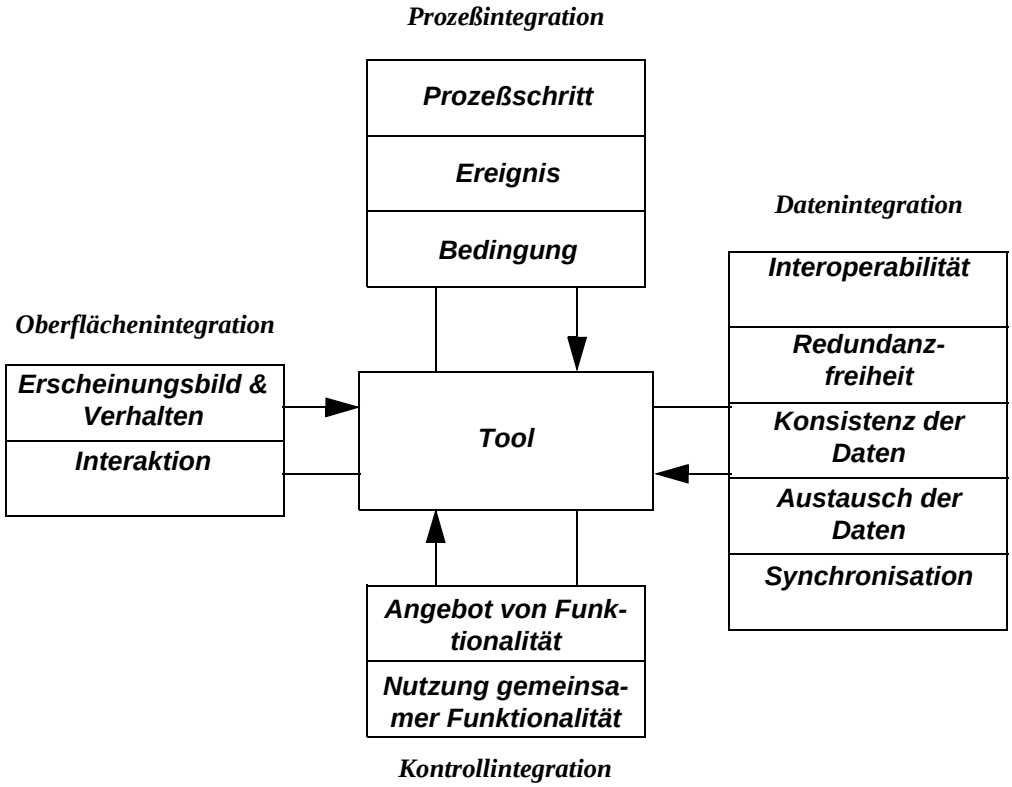


Abbildung C.2 Dimensionen der Integration von Werkzeugen nach [TN92]

che Bedienungsphilosophie, z.B. in der Anordnung der applikationsspezifischen Menüs, haben. Ein Beispiel für unterschiedliche Bedienungsphilosophien ist die Unterteilung von grafischen Editoren in solche, die zuerst das zu bearbeitende Objekt auswählen und anschließend darauf eine Operation aufrufen, und andere, die zuerst die Operation auswählen und dann dazu passend ein Objekt selektieren. Können zwei Werkzeuge, die gleiche Parameter benötigen, so integriert werden, dass Parameter, die beim Aufruf des einen angegeben wurden, bei einem Aufruf des zweiten nicht mehr abgefragt werden, ist in diesem Sinne der höchste Grad an Oberflächenintegration erreicht. Das kann z.B. durch den Aufruf des als zweites gestarteten Werkzeugs durch das zuerst gestartete Werkzeug erreicht werden. Dabei könnte z.B. der Name eines gemeinsam bearbeiteten Objekts mit übergeben werden.

Kontrollintegration beschreibt den Bereich gemeinsam nutzbarer bzw. genutzter Funktionalität. Damit ist gemeint, daß im Idealfall jede irgendwie von einer Umgebung zu erbringende Leistung von nur genau einem Werkzeug bereitgestellt wird. Benötigen mehrere Werkzeuge die gleiche Funktionalität, so wird diese von einem der Werkzeuge realisiert und für die anderen Werkzeuge über eine Schnittstelle zur Mitbenutzung bereitgestellt. In dieser Situation können zwei Aspekte der Integration unterschieden werden: Erstens stellt sich die Frage, inwieweit in einem Werkzeug realisierte Funktionalität von diesem Werkzeug über Schnittstellen nach aussen gegeben wird. Zweitens ist von Interesse, inwieweit Funktionalität von außen von einem Werkzeug genutzt wird, ohne die gleiche Funktionalität intern selbst zu implementieren. Der Grad der Integration beschreibt in diesem Sinne das Zusammenspiel der Werkzeuge untereinander. Schwache Integration bedeutet viel mehrfach implementierte Funktionalität und wenig gemeinsam genutzte Funktionalität bei teilweise gleichem Funktio-

nalitätsbedarf. Enge Integration bedeutet im Gegenzug ein System von miteinander über Schnittstellen vernetzten Werkzeugen, die gemeinsam eine Aufgabe erfüllen. An dieser Stelle verschwimmt der Begriff der Einheit ‘Werkzeug’.

Datenintegration ist der Bereich, der die von den Werkzeugen bearbeiteten Daten betrifft. Zu integrierende Werkzeuge werden zusammengestellt, um eine gemeinsame Aufgabe zu realisieren. Jedes Werkzeug arbeitet dabei auf einem Teil der gesamten Daten im Rahmen dieser Aufgabe. Die Daten, die von verschiedenen Werkzeugen erzeugt und/oder bearbeitet werden, nehmen dabei mindestens teilweise auf die gleichen Fakten und Informationen Bezug. Damit muß sichergestellt sein, daß die überlappenden Daten konsistent zueinander gehalten werden (*Datenkonsistenz)*. Es handelt sich um eine etwas allgemeinere Formulierung des bekannten Problems der Konsistenzsicherung in Datenbanken. Ist die interne Struktur eines Objekts bekannt und hat dieses Objekt Abhängigkeiten zu anderen Objekten, so ist es eine Frage der Integration der bearbeitenden Werkzeuge, ob und wie gut Inkonsistenzen erkannt und vermieden werden können. Interessant ist, daß unter dieser Sichtweise sogar der Integrationsgrad eines Werkzeugs mit sich selbst untersucht werden kann. Wird beispielsweise mit einem Editor ein Objekt bearbeitet, dessen innere Struktur dem Editor bekannt ist und die weiterhin Konsistenzabhängigkeiten zu anderen Objekten mit der gleichen Struktur aufweist, die auch entsprechend mit dem gleichen Editor bearbeitet werden, so kann die Datenkonsistenz auch hier verletzt werden, wenn der Editor keine geeigneten Mechanismen zur Vermeidung oder Kontrolle unterstützt.

Auch bei der *Redundanzfreiheit* handelt es sich um eine Eigenschaft, die im Bereich von Datenbanken betrachtet wird. Dabei geht es darum, ob Daten, die in verschiedenen Kontexten von verschiedenen Werkzeugen angelegt und bearbeitet werden, mehrfach gespeichert werden oder ob durch enges Zusammenarbeiten Daten trotz verschiedener Kontexte und verschiedener Werkzeuge nur genau einmal gespeichert werden. Redundanzfreiheit bezieht sich auf persistente, d.h. über die Laufzeit des erzeugenden Werkzeugs hinaus auf einem Sekundärmedium abgelegte Daten. Der Begriff der *Interoperabilität* beschreibt wiederum die gleiche Eigenschaft für sogenannte transiente Daten, d.h. Daten, die nur zur Laufzeit aufgebaut und benötigt werden. Ein Beispiel ist die gemeinsame Nutzung eines abstrakten Syntaxgraphen (ASG) von einem syntaxgestützten Editor und einem Compiler. Der Editor würde den ASG zur Laufzeit aufbauen. Vom Editor aus könnte dann ein spezialisierter Compiler aufgerufen werden, der, statt diesen ASG erneut für die eigenen Zwecke aufzubauen, den bestehenden ASG nutzt und möglicherweise ergänzt.

Die letzten beiden Teilbereiche der Datenintegration beschreiben die Fähigkeit zweier Werkzeuge, Daten auszutauschen. Auch hierbei wird unterschieden, ob es sich um persistente oder transiente Daten handelt. Die Fähigkeit, persistente Daten auszutauschen, wird als *Datenaustausch* bezeichnet. Bei transienten Daten wird von *Synchronisation* gesprochen. Ein Beispiel für Synchronisation ist die gemeinsame Nutzung eines Datums von einem Editor und einem Debugger, das anzeigt, in welcher Quellzeile welches Quelltextes sich der Debugger während der Ausführung eines aus mehreren Quelltexten erzeugten Programms befindet. Damit wird ermöglicht, daß die aktuelle Zeile im Editor parallel zur Programmausführung im Editor verändert wird.

Den letzten großen Bereich macht die sogenannte *Prozeßintegration* aus. Diese soll beschreiben, wie verschiedene Werkzeuge in Ausführungsreihenfolgen eingebunden sind, die im Rahmen der Gesamtaufgabe beachtet werden müssen. Der Begriff darf

nicht mit dem in Kapitel 2.1.2 definierten, gleichnamigen Begriff verwechselt werden, da er im Kontext dieses Modells auf die Integration zweier Werkzeuge untereinander bezogen und nicht in Beziehung zu einem Prozeß gesetzt wird.

Prozeßintegration erstreckt sich auf die drei Elemente *Prozeßschritt*, *Prozeßereignis* und *Prozeßbedingung*. Ein Prozeßschritt ist ein Abschnitt eines Prozesses, der ein Ergebnis hervorbringt. Zur Durchführung eines Prozeßschrittes kann es nötig sein, verschiedene Werkzeuge einzusetzen, die dann jedes für sich einen kleinen Teil des vollständigen Prozeßschrittes erledigen. Unter diesem Aspekt werden Werkzeuge dann als gut integriert bezeichnet, wenn die Ausführung des einen die Durchführung des anderen gut unterstützt und vorbereitet bzw. ausnutzt. Dabei gibt es natürlich Werkzeuge, für die es irrelevant ist, diese Beziehung aufzuzeigen, da sie letztlich keinen Berührungspunkt miteinander im Prozeß haben. Der Begriff Prozeßereignis bezieht sich darauf, daß im Prozeßverlauf Ereignisse auftreten können, die wiederum Folgeaktionen auslösen können. Gut integriert unter diesem Aspekt sind zwei Werkzeuge dann, wenn sie mit einer gemeinsamen Sicht auf Ereignisse arbeiten. Dabei ist die Menge der Ereignisse angesprochen, aber auch das gemeinsame Verständnis, wer Ereignisse produziert und wer sie annimmt. Der dritte Bereich betrifft die Prozeßbedingungen. Diese beschränken in irgendeiner Weise den Prozeß. Dabei gilt dann, daß zwei Werkzeuge gut im Hinblick auf Prozeßbedingungen integriert sind, wenn sie die gleichen Bedingungen unterstützen und einhalten.

C.1.2 Kritik

Datenintegration wird auf das Problem, Daten zwischen verschiedenen Anwendungen auszutauschen, reduziert. Dabei wird hauptsächlich die technische Seite des Problems betrachtet und die Idee eines Repositories mit Standardschnittstelle entwickelt. Durch diese eingeschränkte Sichtweise werden aber wesentliche Punkte ignoriert. Zum einen ergibt sich das Problem eines Standards an sich. Gibt es mehrere herstellerabhängige Standards, so ist insbesondere das Ziel der herstellerunabhängigen Integration von Werkzeugen wieder in weite Ferne gerückt und das eigentliche Ziel verfehlt.

Zum anderen wird ignoriert, daß der sinnvolle Austausch von Daten nur dann möglich ist, wenn zuerst über die Struktur und Granularität, d.h. die Art der Daten nachgedacht wird. So macht es z.B. keinen Sinn, über die Verwendung eines über das normale Dateisystem hinausgehenden Repositories als Austauschmedium für Daten nachzudenken, solange die darin abgelegten Dokumente nicht feiner strukturiert sind als z.B. in einem Dateisystem. In diesem Fall bringt der Einsatz eines Repositories keinen Vorteil, da im Vergleich zur Ablage im normalen Dateisystem keine zusätzlichen Möglichkeiten entstehen. Insbesondere können keine zusätzlichen Konsistenzkontrollen durchgeführt werden.

Vor diesem Hintergrund sind unstrukturiert arbeitende Texteditoren, wie sie durchaus üblich und gängig sind, vollkommen ungeeignet, da sie nicht einmal Integration zwischen zwei von ihnen selbst angelegten Objekten erlauben. Die Diskussion von Integration von Daten ist also zuerst eine Diskussion von Strukturierung der Daten und damit von Implementierung der einzelnen Werkzeuge und erst danach eine Frage des Datenaustausches zwischen geeignet aufgebauten Werkzeugen.

Für den Begriff der Kontrollintegration gilt die gleiche Schwäche, die auch schon für den Begriff der Datenintegration angemerkt wurde. Im wesentlichen wird geprüft, welche Integrationsmechanismen (im Sprachgebrauch von [Fug93] treffend „enabling technology“ genannt) vorhanden sind. Es wird kaum darüber nachgedacht, was die genauen Ziele und die Voraussetzungen auf der Werkzeugseite sind. Daß sich die Werkzeughersteller zuerst darauf einigen müssen, was sinnvoll auszutauschende Informationen und Ereignisse sind, bevor die Werkzeuge Integrationsmechanismen nutzen können, wird nicht berücksichtigt.

Die obigen Schwächen zeigen sich auch in der Einführung einer „mathematischen Funktion“ in [Was89] zur Zuordnung einer Art Integrationstyp zu einem Werkzeug. Die Definition dieser Funktion legt nahe, daß Werkzeuge mit gleichen Parametern, d.h. mit dem gleichen Integrationstyp, einfacher integriert werden könnten als andere, bzw. sie sogar schon integriert seien. Auf den ersten Blick mag dies auch so scheinen, allerdings werden dabei die oben genannten Probleme außer acht gelassen. Zur Integration sind nicht in erster Linie die Basistechniken, die sich in den Parametern der Funktion widerspiegeln, interessant, sondern die Verständigung über Struktur und kontextsensitive Semantik. Damit ist eine Integration von Werkzeugen in einem der fraglichen Bereiche möglicherweise technisch gesehen einfacher, wenn in allen beteiligten Werkzeugen auch tatsächlich die gleiche Basistechnologie verwendet und angeboten wird. Die zur Benutzung der angebotenen Technologie notwendigen Absprachen sind aber Voraussetzung, um überhaupt eine Integration erreichen zu können. Die eingeführte Bewertungsfunktion sagt also letztendlich kaum etwas über die tatsächliche Integrationsfähigkeit oder den Integrationsgrad von Werkzeugen aus.

C.1.3 Zusammenfassung

Faßt man die Darstellung aus Modell und Kritik zusammen, so kann man sagen, daß im Dimensionenmodell zu sehr die Basistechniken und zu wenig die konzeptionelle Ebene der Integration zur Definition der verschiedenen Dimensionen führen. Wassermann konzentriert sich auf die Lösung der technischen Probleme, bevor sichergestellt ist, daß durch geeignete Vorarbeit auf konzeptioneller Ebene diese Techniken auch sinnvoll eingesetzt werden können. Insbesondere mangelt es an einer Analyse und Identifikation der durch die einzusetzenden bzw. zu entwickelnden Techniken zu lösenden Probleme. Das führt auch dazu, daß nicht betrachtet wird, wie die einzelnen Dimensionen zusammenhängen.

In Bezug auf diese Arbeit und die in Kapitel 4 beschriebenen Probleme kann festgestellt werden, daß sich das Dimensionenmodell auf einer zu abstrakten Ebene befindet, um bei der Suche nach Problemlösungen verwendet werden zu können. Es kann höchstens einer sehr allgemeinen Betrachtung von Integration dienen. Umgekehrt kann festgestellt werden, daß sich die beschriebenen Problemlösungen in den Bereichen Kontroll-, Daten- und Prozeßintegration befinden. Oberflächenintegration kann im Kontext einer im Prinzip heterogenen Umgebung nicht im primären Fokus liegen.

C.2 Ebenenmodell

C.2.1 Darstellung des Modells

Ein anderer Ansatz, Integration von Werkzeugen zu beschreiben, ist am Software Engineering Institute der Carnegie Mellon University entwickelt worden ([BM91], [Bro92], [BM92]). Eingebettet ist das Gesamtmodell in ein Schema aus drei Dimensionen, die sich aber wesentlich von denen des Dimensionenmodells unterscheiden. Die eingeführten Dimensionen sind

- die *Werkzeugintegration*,
- die *Prozeßintegration* und
- die *Managementintegration*.

Die Dimension Werkzeugintegration umfaßt den Aspekt der Zusammenarbeit mehrerer Werkzeuge (vornehmlich unter dem Aspekt gemeinsam zu bearbeitender Daten) auf technischer Ebene, die Dimension Prozeßintegration umfaßt (im Unterschied zur in Kapitel 2.1.2 eingeführten Definition) ausschließlich den Aspekt der Festlegung der Reihenfolge der Ausführung verschiedener Werkzeuge und die Managementdimension umfaßt den Bereich der Überwachung und Kontrolle des real ausgeführten Prozesses.

Bei der Betrachtung der Dimensionen fällt auf, daß PSEU im wesentlichen alle drei Dimensionen unterstützen müssen. Der Bereich der Prozeßintegration kann dabei vollständig durch die Modellausführung geleistet werden, da das Prozeßmodell festlegt, in welcher Reihenfolge Aktivitäten aufgerufen werden. Dabei wird natürlich vorausgesetzt, daß Aktivitäten nicht an der Modellausführung und damit am Modell vorbei aufgerufen werden können. Die beiden Bereiche Werkzeugintegration und Managementintegration umfassen zusammengenommen ungefähr den in dieser Arbeit als Prozeßintegration bezeichneten Bereich. Dieser Unterschied läßt sich dadurch erklären, daß in dieser Arbeit das Problem der Spezifikation eines Prozeßmodells und damit der Reihenfolge von Aktivitäten in der Modellausführung im wesentlichen als in anderen Arbeiten gelöst betrachtet wird und stattdessen die Integration der Werkzeuge in den Prozeß mit den genannten Problemen gleichgesetzt werden muß, d.h. mit der Auswahl geeigneter Werkzeuge und Synchronisation der Ausführung dieser Werkzeuge mit der Modellausführung.

In der Beschreibung des Ebenenmodells gehen die Autoren vornehmlich auf den Beireich der Werkzeugintegration ein. Darin werden fünf *Integrationsebenen* unterschieden, die die semantische Qualität der Integration bzw. der Integrationsmechanismen beschreiben sollen. Diese Ebenen sind die Basis der Benennung des Modells in diesem Kapitel und werden folgendermaßen benannt:

- *Carrier Level*
- *Lexical Level*
- *Syntactic Level*
- *Semantic Level*
- *Method Level*

Die Ebenen beschreiben verschiedenen Grade der Integration. Das *Carrier Level* beschreibt die schwächste Form der Integration. Auf dieser Ebene werden unstrukturierte Bytefolgen zwischen den auf diese Weise integrierten Werkzeugen ausgetauscht. Unix-Pipes sind ein Beispiel für die Realisierung dieser Integrationsform. Die Daten werden dabei i.d.R. als "Paket" betrachtet, das nur komplett bearbeitet wird. Die nächste Ebene, das *Lexical Level* beschreibt Integration, die auf einer einfachen lexikalischen Ebene abgewickelt wird. Im Gegensatz zum Carrier Level kennen die beteiligten Werkzeuge also schon Bedeutung des Dateninhalts in einer sehr einfachen Weise. So kann z.B. festgelegt werden, daß, wenn eine Datenzeile mit einem Punkt beginnt, danach ein Schlüsselwort folgt, das Informationen über die Bearbeitung liefert. Jedes einzelne Werkzeug kennt nur die von ihm selbst bearbeiteten Schlüsselworte. Insbesondere werden keine allgemeinen syntaktischen Regeln für die Daten festgelegt. Das ist auch der Unterschied zum *Syntactic Level*, in dem sich die ausgetauschten Daten durch eine Struktur kennzeichnen, die in allen Werkzeugen bekannt ist und genutzt werden kann. Werkzeuge, die auf gemeinsamen, graphbasierten Datenstrukturen aufsetzen, sind ein Beispiel für diese Ebene der Integration. Dieser Grad der Integration wird im *Semantic Level* dadurch gesteigert, daß nicht nur die Struktur der Daten vordefiniert wird, sondern auch mögliche Operationen auf den Daten und ihre Auswirkungen allen Werkzeugen bekannt sind. Werden objektorientierte Datenbanken benutzt, um verschiedene Werkzeuge miteinander zu integrieren, so ist i.d.R. durch die Schemadefinition mit Datenanteil und darauf definierten Methoden das Semantic Level erreicht. Das *Method Level* beschreibt zum Schluß den höchsten Integrationsgrad. Auf der Ebene des Method Level besitzen die einzelnen integrierten Werkzeuge auch noch Wissen über den Softwareprozess (mindestens einen Ausschnitt daraus), in dem sie eingesetzt werden (können). Ein Werkzeug, das mit einem anderen auf dieser Ebene integriert ist, muß also wissen, ob und unter welchen Umständen / zu welchen Zeitpunkten in der Bearbeitung es das andere Werkzeug aufrufen darf oder sogar muß, um dem Software-Lebenszyklus zu folgen. Auf dieser Ebene ergibt sich also eine Überschneidung mit dem Begriff der Prozeßintegration, die darauf zurückzuführen ist, daß die Werkzeuge zwar Prozeßwissen über die sie betreffenden Zusammenhänge haben, aber nicht über den Gesamtprozeß.

C.2.2 Kritik und Zusammenfassung

Betrachtet man das Ebenenmodell unter dem Aspekt, wie es die in dieser Arbeit anzusprechenden Bereiche beschreibt, so kann festgestellt werden, daß es eine eingeschränkte Sicht der für die Integration von Werkzeugen in eine PSEU wichtigen Punkte hat. Trotz der sich vom Dimensionenmodell unterscheidenden Sichtweise stellt das Modell Integration von Werkzeugen untereinander in den Vordergrund, was auch davon getrieben ist, daß nicht von der Existenz einer Prozeßsteuerungskomponente ausgegangen wird. Diese Sichtweise resultiert in der Notwendigkeit, Prozeßwissen in den Werkzeugen zu implementieren und dadurch Prozeßbestandteile hart zu verdrahten. Das widerspricht der notwendigen Flexibilität bei der Erstellung einer PSEU. Wie im Dimensionenmodell ist auch zu beobachten, daß der Begriff Prozeßintegration nur mit der Ausführungsreihenfolge von Werkzeugen in Verbindung gebracht wird. Der Prozeß selbst wird also vornehmlich als die Beschreibung des Softwarelebenszyklus und Prozeßintegration als die Einordnung der Werkzeuge in diesen Prozeß betrachtet. Diese Sichtweise paßt nicht mit der inzwischen üblichen sehr viel weitergehenden Sichtweise eines Prozesses überein, die auch Dokumentrelationen, Rechte auf Dokumenten u.ä. als zum Prozeß gehörig betrachtet. Unter dieser Sichtweise gehört die im

Ebenenmodell gesonderte Dimension Managementintegration in den für diese Arbeit definierten Begriff der Prozeßintegration hinein. Insbesondere ist unter diesem Aspekt die Synchronisation der Modellausführung mit der Prozeßausführung ein Teil der Prozeßintegration.

C.3 Klassenmodell

C.3.1 Darstellung des Modells

Der dritte Ansatz eines Integrationsmodells, der hier vorgestellt werden soll, ist das Ergebnis einer Dissertation an der University of Queensland, Australien (Yun Yang, [Yan92]). Auch diese Arbeit geht davon aus, daß die Integration von Werkzeugen nicht in die von Wassermann vorgesehenen, unabhängig existierenden Integrationsebenen unterteilt werden kann. Entsprechend können diese Ebenen auch nicht getrennt untersucht werden. Stattdessen geht Yang davon aus, daß Integration in einer Art gesamtheitlichen Ansatz zu kategorisieren ist.

Es werden dazu drei Klassen von Integration definiert. Das zugehörige Modell wird in dieser Arbeit als *Klassenmodell* bezeichnet. Die Integrationsklassen fassen einzelne Anteile der Integrationsdimensionen des Dimensionenmodells bzw. der Werkzeugintegrationsebenen des Ebenenmodells zusammen. Die Klassen werden kurz als *Klasse 1 Integration*, *Klasse 2 Integration* und *Klasse 3 Integration* bezeichnet. Im folgenden werden diese Klassen kurz erläutert.

Die Definition der verschiedenen Integrationsklassen basiert auf den im Dimensionenmodell von Wassermann eingeführten Integrationsdimensionen. Zwei davon, Plattformintegration und Prozeßintegration, werden von Yang als nicht wesentlich für die Erörterung von Integration eingestuft und deshalb nicht näher erläutert. Prozeßintegration wird dabei von der Betrachtung ausgenommen, weil angenommen wird, daß sie keine wesentlichen Veränderungen der Integrationstechniken zur Folge hat. Die verbleibenden drei Integrationsdimensionen werden genauer betrachtet, als es Wassermann in [Was89] getan hat, um auf dieser Untersuchung die drei Integrationsklassen aufzubauen. Anschließend werden Ziele für die jeweiligen Integrationsdimensionen formuliert. Da sich die Definition der Integrationsklassen ausschließlich auf die Dimensionen Kontroll- und Datenintegration bezieht, wird die in [Yan92] vorgenommene Ausformulierung der Oberflächenintegration an dieser Stelle nicht weiter aufgegriffen. Die Ergebnisse für die Kontroll- und Datenintegration werden kurz zusammengefaßt, um dann die Integrationsklassen definieren zu können.

Als Ziele der Kontrollintegration arbeitet Yang im wesentlichen drei Punkte heraus. Erstens strebt Yang an, den Benutzer von expliziten Aufrufen zu befreien, wenn es sich um die Hintereinanderausführung von Werkzeugen handelt. Dieser Punkt entspricht dem, was sich im Ebenenmodell hinter dem Begriff der Prozeßintegration verbirgt und umfaßt insbesondere die mit einer Prozeßsteuerungskomponente notwendige Integration. Das zweite Ziel ist die Erhöhung der Performance der Softwareentwicklung, die mit einer SEU ausgeführt wird, durch eine Parallelisierung einzelner Schritte. Darunter ist zu verstehen, daß, während der Entwickler noch seinen Editor geöffnet hat, z.B. schon die Übersetzung eines Teils des von ihm bearbeiteten Textes denkbar ist, sofern sich eine sinnvolle Gliederung finden läßt. Das dritte Ziel ist ein sehr allgemeines. Es besagt, daß durch Kontrollintegration Architekturen ermöglicht werden, in denen neue

Funktionalitäten aus vorhandenen zusammengesetzt werden können. Dadurch soll Wiederverwendung vereinfacht und der Bau von neuen, monolithischen Werkzeugen unnötig gemacht werden.

Im Bereich der Datenintegration werden vier verschiedene Ziele herausgearbeitet. Erstens soll ein Datenaustausch von sogenannten primären oder sekundären Daten zwischen den zu integrierenden Werkzeugen möglich sein. Primäre Daten sind dabei solche, die der eigentliche Gegenstand der gemeinsamen Bearbeitung sind, während sekundäre Daten z.B. die Ausgaben eines der Werkzeuge während der Bearbeitung sind. Zweitens soll der Austausch von geeignet strukturierten Daten unterstützt werden. Das dritte Ziel ist dann die Unterstützung einer geeigneten Granularität im Datenaustausch. Damit ist gemeint, daß z.B. Unix-Pipes nur als Datenaustauschtechnik geeignet sind, wenn es sich bei den Daten um unstrukturierte Daten (d.h. Bytefolgen) und bei den Werkzeugen um byteorientierte Werkzeuge handelt, da Unix-Pipes nur strukturlose Bytefolgen übertragen. Als viertes und letztes Ziel ist es notwendig, eine geeignete Koordination von Kontrolle und Datenaustausch zwischen den zu integrierenden Werkzeugen zu erzielen, so daß es möglich ist, Daten auch über mehrere Werkzeuge hinweg konsistent zu halten.

Klasse 1 Integration beschreibt aufbauend auf den gerade erläuterten Integrationsdimensionen, daß eine Verbesserung der Produktivität bei der Nutzung der integrierten Werkzeuge dadurch erreicht wird, daß “user-generated delay” reduziert wird. Das bedeutet, daß z.B. Benutzereingaben zwischen Werkzeugaufrufen durch Vermeidung von replizierten Eingaben reduziert werden. Diese Klasse wird dadurch beschrieben, daß sie das erste Ziel der Kontrollintegration und das erste bzw. das vierte Ziel der Datenintegration verfolgt. Klasse 2 Integration beschreibt die Vermeidung von “tool-generated delay”. Damit ist gemeint, daß Werkzeuge durch geschickte Verzahnung und Parallelisierung ihrer Ausführung die Wartezeiten für den Benutzer reduzieren können. Diese Klasse basiert auf der Verfolgung des Ziels zwei der Kontrollintegration und der Ziele zwei und drei der Datenintegration. Klasse 3 Integration hat feingranulare Synchronisation und trotzdem eine Unabhängigkeit der Werkzeuge zum Ziel. Sie basiert auf dem Ziel drei der Kontrollintegration und dem Ziel vier der Datenintegration.

C.3.2 Kritik und Zusammenfassung

Im Hinblick auf die vorliegende Arbeit ist an Yangs Ansatz besonders wichtig, daß durch die Prozeßintegration keine zusätzlichen Anforderungen an die Integrationstechniken gestellt werden. Dabei wird ignoriert, daß es im Bereich von Softwareprozessen Elemente gibt, die nicht über die normale Abstimmung von Daten oder Kontrolle im obigen Sinne angesprochen werden. Ein Beispiel ist die Vergabe von Rechten für einzelne Dokumente, insbesondere wenn diese als Komponenten eines zusammengesetzten Dokuments definiert sind. Hierbei muß das bearbeitende Werkzeug mit beschreibenden Informationen (z.B. den Rechten oder der Menge der Komponenten, die der Aufrufer verändern darf) versorgt werden. Legt man die im Klassenmodell definierten Begriffe weit genug aus, so kann man diese Informationen natürlich auch in die Klasse der sekundären Daten fassen und somit sagen, daß dieser Aspekt auch abgedeckt ist. In dieser Arbeit wird dieser Standpunkt nicht geteilt. Der Grund ist, daß gerade die von Yang als *Kontrollwerkzeuge* bezeichnete Klasse von Werkzeugen, zu

der auch Prozeßsteuerungskomponenten zählen, besondere Anforderungen an die Zusammenarbeit von Werkzeugen stellen.

Erwähnenswert ist auch noch der Unterschied zum Dimensionenmodell. Im Klassenmodell werden die gleichen Parameter betrachtet, nach denen Integration von Werkzeugen einzuordnen ist. Im Gegensatz zum Dimensionenmodell wird aber ein Bezug der einzelnen Integrationsdimensionen in Form der Integrationsklassen hergestellt. Wassermann geht davon aus, daß jede einzelne Integrationsdimension orthogonal zu den anderen ist und somit unabhängig von den anderen graduell von keiner Integration bis zu sehr starker Integration verändert werden kann. In dem durch die Dimensionen aufgespannten Raum ergeben sich beliebig viele Punkte, die verschiedene Formen von Integration beschreiben. Yang beschreibt mit seinen Integrationsklassen konkrete Anforderungen und gliedert somit alle bei Wassermann denkbaren Möglichkeiten in seine drei Integrationsklassen. Dadurch ergibt sich ein Rahmen, der eine Einordnung von Integrationstechniken erlaubt und das Dimensionenmodell handhabbar macht.

C.4 ECMA/NIST Referenzmodell

C.4.1 Darstellung des Modells

Ein konkreteres Modell, das die in einer SEU auftretenden Bereiche herausarbeitet, ist das von der ECMA und dem NIST gemeinsam entwickelte und in [NE93] vorgestellte *Referenzmodell*. Referenzmodell bedeutet dabei im Gegensatz zu Framework, daß Bestandteile von (P)SEU identifiziert werden, ohne daß durch das Modell eine bestimmte Architektur impliziert oder vorgeschrieben wird. Eine schematisierte Darstellung des Modells ist in Abbildung C.3 zu sehen.

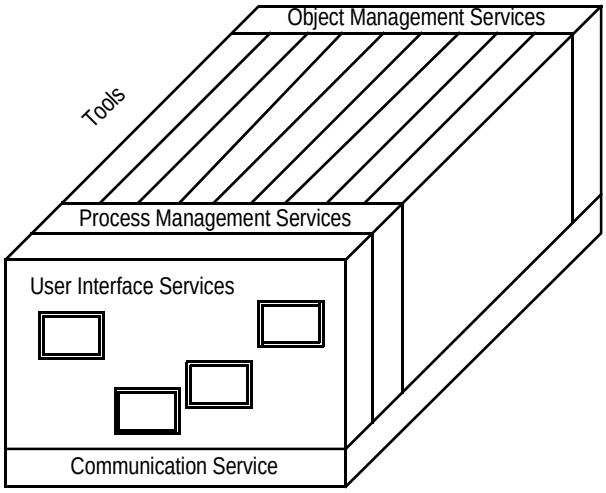


Abbildung C.3 Struktur des ECMA/NIST Referenzmodells nach [NE93**]**

Das Modell soll neben anderen Zielen eine Beschreibung und einen Vergleich von existierenden Frameworks ermöglichen, eine Basis zur Identifikation von Standards (im Umfeld von SEU) bieten und Möglichkeiten zur Integration und Interoperabilität von Werkzeugen beschreiben (vgl. [NE93], Kapitel 1.3). Die Bereiche, in denen die zu lösenden Probleme liegen, werden als Integration und Prozeßunterstützung identifiziert

(vgl. [NE93], Kapitel 3). Der Bereich der Integration wird dabei auf der Basis der fünf Integrationsdimensionen des Dimensionenmodells gegliedert. Die Dimension der Prozeßintegration wird - leicht abweichend von den anderen bisher vorgestellten Definitionen - als die Möglichkeit des Zugriffs auf Umgebungsfunktionalität auf der Basis eines ausführbaren Entwicklungsprozesses eingeführt. Diese Sichtweise läßt wie andere Definitionen die Problematik der Synchronisation von Modellausführung und Prozeßausführung unberücksichtigt.

Das Modell selbst, so wie es in Abbildung C.3 dargestellt ist, beinhaltet fünf große Bereiche, aus denen eine SEU zusammengesetzt wird.

- *Object Management Services*

  Die Object Management Services dienen der Definition, der Speicherung, der Wartung, dem Management und dem Zugriff auf die Objekte der Software-Entwicklungsumgebung und Relationen zwischen ihnen.

- *Process Management Services*

  Diese Gruppe von Services hat den Zweck der Verwaltung und computerbasierten Ausführung eines den gesamten Software-Lebenszyklus beschreibenden Softwareprozesses. Der in dieser Arbeit zu untersuchende Bereich erstreckt sich über drei der identifizierten Teilbereiche dieser Gruppe.

  - Der *Process Development Service* bietet Möglichkeiten, einen Prozeß zu definieren und zu instanziieren,
  - der *Process Enactment Service* erlaubt die Ausführung des spezifizierten Prozesses und
  - der *Process Monitoring Service* beinhaltet Möglichkeiten, den laufenden Prozeß zu überwachen.

- *Communication Services*

  Die Communication Services haben den Zweck, die notwendige Kommunikation zwischen den einzelnen Komponenten einer SEU zu befriedigen. Im Zusammenhang mit dem Process Monitoring Service sind dabei insbesondere der *Message Service* und der *Event Notification Service* als Teile zu nennen.

- *User Interface Services*

  Die User Interface Services sind als Basis einer durchgängigen Benutzungsschnittstelle für eine SEU aus eng zusammenarbeitenden Komponenten vorgesehen.

C.4.2 Kritik und Zusammenfassung

Das ECMA/NIST Referenzmodell gibt eine wertvolle Hilfe für die Einstufung von SEU und berücksichtigt auch Prozeßsteuerungsfragen. Umgekehrt ist es auf einem Abstraktionslevel gehalten, das es nicht erlaubt, konkrete Konsequenzen für eine zu erstellende SEU abzuleiten. Dies ist nur bedingt ein negativer Kritikpunkt am Modell, da es mit genau dem Anspruch erstellt worden ist, unabhängig von konkreten Architekturen und Implementierungen zu sein. Allerdings folgt daraus, daß das Modell in einem Grenzbereich von rein konzeptionellen Modellen zu Frameworks liegt. Einzelne Frameworks ordnen sich zwar entlang des Referenzmodells ein (vgl. z.B. COOP, [Ber92]), für diese Arbeit können aber daraus keine konkreten Hinweise auf die Lösung der in Kapitel 4 angesprochenen Probleme gezogen werden.

D

ANWENDUNGEN ALLGEMEINER INTEGRATIONSTECHNIKEN

D.1 Multiview-Ansätze

Die in diesem Kapitel betrachteten Integrationsansätze, die als *Multiview-Ansätze* zusammengefaßt werden sollen, befassen sich mit einem Teilaspekt der Integration. Es geht darum, daß eine Änderung von Daten, die von verschiedenen Werkzeugen parallel bearbeitet und angezeigt werden, in allen Vorkommen gleichzeitig konsistent durchgeführt wird. Dieses Problem hat direkte Bezüge zur vorliegenden Arbeit, wenn die prozeßrelevanten Daten einer PSEU als Daten betrachtet werden, die die Werkzeuge und die Prozeßmaschine parallel bearbeiten.

Meyers beschäftigt sich in [Mey91] damit, welche Techniken prinzipiell denkbar sind, um Integration derart zu erlauben, daß eine gleichzeitige Sichtbarkeit von Datenänderungen in verschiedenen Werkzeugen gewährleistet ist. Er unterscheidet fünf Vorgehensweisen:

- "Shared file systems": Es handelt sich um die im Kapitel 5.1.1.1 vorgestellte Integrationslösung auf der Basis eines normalen Dateisystems.
- "Selective broadcasting": Es handelt sich um die im Kapitel 5.1.2.1 vorgestellte Lösung auf der Basis eines Nachrichtensystems.
- "Simple database": Es handelt sich um eine Integration auf der Basis einer gemeinsam genutzten Datenbank. Sie ist der in Kapitel 5.1.1.1 vorgestellten Datenbanklösung ähnlich, unterscheidet sich aber insofern, als jedes Werkzeug sein eigenes Schema definiert, auf das dann von anderen Werkzeugen zugegriffen werden kann.
- "View oriented database": Hierbei handelt es sich um eine hier nicht vorgestellte Verfeinerung des einfachen Datenbankansatzes. Jedes Werkzeug definiert eine eigene *Sicht*, die dafür sorgt, daß nicht interessante Daten ausgeblendet und die gemeinsamen Daten aus der Sicht des bearbeitenden Werkzeugs definiert werden.
- "Canonical Representation": Diese Lösung basiert auf einer gemeinsamen Datendefinition aller in der Umgebung arbeitenden Werkzeuge und im Idealfall aller noch kommenden Werkzeuge. Es ist offensichtlich, daß dieses Ziel bestenfalls teilweise

erreicht werden kann. Auf der Basis von Datenbanken handelt es sich dann um den in Kapitel 5.1.1.1 vorgestellten, von PCTE verfolgten Ansatz. Beispiele dazu (die nicht auf PCTE basieren) werden auch in [ES89], [Nag96], [Lef95], [Emm95], [Mar90] und ([FS93]) beschrieben.

Meyers faßt für jeden dieser Ansätze die Vor- und Nachteile zusammen. Er kommt zu dem Ergebnis, daß die Ansätze, die eine einfache a posteriori Integration erlauben, inhärent unsicher sind, wenn es um das gesteckte Ziel der Konsistenz geht. Umgekehrt sind die Ansätze, die von der Anlage her Konsistenz gewährleisten, inhärent mit schwieriger a posteriori Integration verbunden.

Die für die Canonical Representation genannten Beispiele integrieren ihre Daten durch mehr oder weniger umfangreiche, globale Datendefinitionen auf der Basis attributierter abstrakter Syntaxgraphen. Dazu werden geeignete Graphen definiert, die sich aus der Syntax der Sprachen ableiten, die in den jeweiligen Werkzeugen eingesetzt werden. Die einzelnen Werkzeuge lesen dann den Teil der Daten, der sie interessiert, und transformieren ihn in eine geeignete Darstellung. In [GHJV94] wird eine generelle Implementierungsstrategie für das Multiview-Problem vorgeschlagen, das *Observer-Pattern*

Yang ([Yan92]) betrachtet das Problem unter veränderten Voraussetzungen. Er geht davon aus, daß es eine Menge von unabhängig voneinander arbeitenden Werkzeugen gibt, deren Daten auch nicht miteinander in der beschriebenen Form integriert werden sollen oder können. Da die Werkzeuge im Sinne eines Multiview-Ansatzes trotzdem miteinander integriert werden sollen, entwickelt er einen geänderten a priori Integrationsansatz. Er entwirft eine gemeinsame Ausgabestruktur, die allen Werkzeugen bekannt sein muß. Die Werkzeuge koordinieren sich dann gegenseitig auf der Basis der Ausgabestruktur, so daß gemeinsame Daten in diese Struktur geschrieben und auch von dort gelesen werden. Die Ausgabe auf den Bildschirm erfolgt über ein gemeinsames Front-End-Werkzeug, das in der Lage ist, die Daten geeignet in verschiedenen Ausgabesichten darzustellen.

Der von Yang gewählte Ansatz einer Integration ist im Rahmen dieser Arbeit nicht verwendbar, da er ganz speziell implementierte Werkzeuge benötigt. Der Problempunkt a priori Integration wird durch die besondere Behandlung der Ausgaben in einem einzigen Front-End noch intensiviert. Wäre es möglich, eine Menge von Bausteinen zu implementieren, die sowohl die Funktionalität der Umgebung abdecken als auch eine Prozeßsteuerungskomponente umfassen würde, wäre diese Lösung aber weitreichender als die anderen in Frage kommenden a priori Ansätze, da sie sofort ein einheitliches Front-End der entstehenden PSEU implizieren würde.

Die Betrachtung von MultiView ist für diese Arbeit besonders interessant, da im Rahmen dieses Projektes auch Experimente zur Integration der MultiView-Umgebung mit einer Prozeßmaschine vorgenommen wurden ([MPMH93]). Weder MultiView (auf der Prozeßausführungsseite) noch Merlin (auf der Modellausführungsseite) waren so ausgestattet, daß sie ohne weiteres miteinander integriert werden konnten. Sie waren so implementiert, daß sie zu jedem Zeitpunkt die volle Kontrolle über die eigene Umgebung besaßen. Das Problem, diese beiden Komponenten miteinander zu integrieren, wurde durch Eingriffe in die jeweiligen Quelltexte, die beim Experiment vorlagen, gelöst. Eine Integration ohne Änderung der Werkzeuge schien aufgrund der mangelnden Vorbereitung der Werkzeuge nicht sinnvoll. Die vorgenommene Integrationslösung ist eine nachrichtenbasierte Lösung. Dadurch, daß von beiden Systemen die

Quelltexte zugreifbar waren, konnte ein spezialisiertes Protokoll implementiert werden. Dieses Protokoll wird über einen sogenannten Adapter vermittelt.

D.2 Werkzeuggeneratoren und Meta-SEU

A priori Integration besitzt den Reiz, durch geeignete Implementierung der zu integrierenden Werkzeuge neue, bisher nicht benutzte Konzepte einführen und deren spezielle Eigenschaften optimal ausnutzen zu können. Ein ganz wesentlicher Nachteil dieses Ansatzes ist, daß dieses Vorgehen zur Erstellung einer SEU erfordert, daß sukzessive alle darin benötigten Werkzeuge unter Verwendung der neuen Konzepte reimplementiert werden müssen. Das ist um so aufwendiger, je umfangreicher die Integrationsmechanismen sind. Diese Situation u.a. hat dazu geführt, daß in verschiedenen Kontexten *Werkzeuggeneratoren* (z.B. *GENESIS* [Emm95], *PROGRESS* [Zün95], vgl. auch Kap. 6.6 in [Emm95] für weitere Arbeiten) entstanden sind. Diese erlauben es, Werkzeuge auf der Basis einer *Werkzeugspezifikationssprache* zu beschreiben. Die Spezifikation wird dann einem Generator übergeben, der daraus ein ablauffähiges Programm erstellt. Dieses Vorgehen vereinfacht die Erstellung von Werkzeugen im Vergleich zu einer Programmierung von Hand. Ziel dieses Vorgehens ist u.a., daß Werkzeuge, von denen keine gut integrierte Version vorliegt, einfach (re-)implementiert werden können und auf diese Weise die jeweils unterstützten (Integrations-)Konzepte in allen benutzten Werkzeugen gleichermaßen vorhanden sind.

Die existierenden Ansätze haben in der Regel das Ziel, Integration und feingranulare Bearbeitung von Daten zu unterstützen. Es wird meist durch Definition einer geeignet aufgebauten Datenstruktur erreicht (z.B. *attributierte abstrakte Syntaxgraphen*). Die Unterstützung der Nutzung gemeinsamer Funktionalität, im Umfeld dieser Arbeit z.B. die Unterstützung der Synchronisation von Modellausführung und Prozeßausführung, wird nicht betrachtet. In [Emm95] wird das erste Mal ein Ansatz vorgestellt, der auch eine Integration der generierten Werkzeuge auf funktionaler Ebene und insbesondere mit einer Prozeßsteuerungskomponente vorsieht. Dieses Konzept ist in [Wag95] im Detail ausgeführt. Als Grundidee werden mehrere *Nachrichtentypen* definiert. Ihre Semantik ist dabei von der vom Werkzeug bearbeiteten Datenstruktur abhängig und wird in der Werkzeugspezifikation festgelegt. Daraus werden zur Laufzeit des generierten Programms konkrete Nachrichten abgeleitet, mit deren Hilfe die Synchronisation des Werkzeugs mit der Prozeßsteuerungskomponente (sowohl Feedback als auch Unterstützung) vorgenommen wird.

Für die in dieser Arbeit anstehenden Probleme ist diese Lösung zur Definition integrierbarer Werkzeuge ein erster guter Ansatz. Die in [Wag95] beschriebene Vorgehensweise läßt weiterhin Integrationsprobleme offen. In erster Linie sind dabei zwei Punkte zu nennen. Zum einen ist die Definition der erwähnten Nachrichtentypen zu sehr an der Werkzeugseite orientiert. Es wird vorausgesetzt, daß ein Werkzeug immer auf genau einem Dokument arbeitet. Dadurch ist es z.B. unmöglich, Werkzeuge zu definieren, die auf aggregierten Dokumenten arbeiten. Wird ein verändertes Prozeßmodell vorgegeben, das gegenüber einem vorherigen Modell auf einer feineren oder gröberen Granularität arbeitet, bleibt daher nur die Möglichkeit, sofort wieder neue Werkzeuge zu definieren, die 1:1 zu der im Modell vorgesehenen Granularität passen. Ebenso ist keine Verwendung anderer von einer Prozeßmaschine verwalteten und für eine Synchronisation benötigten Entitäten möglich. Der zweite Punkt ist nicht dem konkreten Konzept anzulasten, sondern ist generellerer Natur. Die generierten Werk-

zeuge stützen sich auf festen Mechanismen ab, wie z.B. einem konkreten *Nachrichtenmechanismus*. Sollen die Werkzeuge dann mit einer Prozeßsteuerungskomponente integriert werden, so ist es notwendig, den gleichen Mechanismus dort auch zur Verfügung zu haben oder andere Integrationstechniken zu verwenden. Dieses Problem besteht, weil die Werkzeuggeneratoren nicht dazu vorgesehen sind, z.B. auch eine Prozeßsteuerungskomponente zu erzeugen und damit implizit dieses Problem der Anbindung zu lösen. Entsprechend gibt es auch Probleme auf der Ebene der Interpretation von Nachrichten, da z.B. Benennungen von Dokumenttypen und Relationen, die in einem verwendeten Prozeßmodell definiert wurden, nicht unbedingt mit den von den Werkzeugen verwendeten übereinstimmen. An dieser Stelle ist eine Verbindungsschicht zwischen den Werkzeugen der Prozeßausführung und der Prozeßsteuerungskomponente der Modellausführung notwendig, wobei die vorgestellten Konzepte einen guten Ansatzpunkt für die Anbindung der Werkzeuge bieten.

Die Werkzeuggeneratoren zugrundeliegende Idee, eine spezialisierte Erstellung bestimmter Software zu unterstützen, ist in anderen Arbeiten in ein noch umfassenderes Konzept eingebettet worden. Werkzeuggeneratoren erzeugen einzelne Werkzeuge mit der Absicht, sie dadurch, daß die hineingenerierten Integrationsmechanismen ausgenutzt werden, leicht zusammensetzen zu können. *Meta-SEU,* wie sie z.B. in [Nag93] benannt werden (z.B. *PSG* [BS86], *Mentor* [DGHKL84] und *IPSEN* [Nag96]), dienen im nächsten Schritt dazu, ganze Umgebungen zu erstellen. Der Name Meta-SEU ergibt sich daraus, daß es sich selbst um Software-Entwicklungsumgebungen handelt, die insbesondere dafür gebaut sind, die Entwicklung eines speziellen Softwaretyps, nämlich SEU, zu unterstützen. Da ein Teil einer zu erstellenden SEU aus den jeweiligen für die Teilaufgaben der Softwareentwicklung zuständigen Werkzeugen besteht, können Werzeuggeneratoren sinnvoller Bestandteil einer Meta-SEU sein. Prototypen, wie die oben genannten PSG und Mentor, entsprechen dabei nicht der Definition einer (Meta-) SEU nach Kapitel 2.1, da sie nicht den gesamten Lebenszyklus der mit ihrer Hilfe zu erstellenden Software abdecken. Im diesem Sinne sind solche Umgebungen Werkzeuge (wenn auch mit weitreichender Funktionalität).

Meta-SEU haben im wesentlichen die inzwischen mehrfach angeführten Vorteile eines a priori Ansatzes, d.h. die Möglichkeit verschiedenste Integrationsmechanismen als Basismechanismen einzuführen und durchgängig zu nutzen. Sie erreichen dadurch einen hohen Integrationsgrad für die erstellten SEU. Umgekehrt wird aber ein Nachteil des a priori Ansatzes noch evidenter. Soll mit einer Meta-SEU wirklich eine Umgebung erstellt werden, die vollständig von den Vorteilen einer a priori Definition der Integrationsmechanismen Gebrauch macht, so ist es notwendig, sämtliche Werkzeuge, die im Rahmen des Softwarelebenszyklus eingesetzt werden, für diese zu erstellende Umgebung neu zu spezifizieren und zu erzeugen. Dies ist ein Aufwand, der möglicherweise vertretbar ist, wenn es sich um eine Redefinition einer bestehenden Umgebung handelt. Wie aber schon in der Einleitung bemerkt wurde, ist kaum eine Firma wirklich in der Lage, den Aufwand der Entwicklung von Werkzeugen “für jede Gelegenheit” zu unterstützen. Dadurch wird die initiale Erstellung einer SEU zum Problem. Insbesondere Speziallösungen von kleineren Firmen, die nur kleinere Segmente der Software-Entwicklung abdecken, können im Rahmen einer solchen SEU nur dann optimal integriert eingesetzt werden, wenn sich Umgebungsersteller und kleine Firma auf eine Art kleinen Standard oder möglicherweise die gemeinsame Verwendung einer Meta-SEU geeinigt haben. Für den Anwender, der für seine Entwicklung die für sich optimale Kombination von Werkzeugen schaffen möchte und dabei u.U. seine Priorität auf die

Funktionalität der Bausteine und nicht auf deren optimale Integration legt, ist dieser Ansatz damit eine wesentliche Einschränkung. Wird ein solcher Ansatz allerdings in Kombination mit einem Konzept zur a posteriori Integration benutzt, so erlaubt er die Flexibilität, die der Anwender benötigt.

Wichtig ist in diesem Zusammenhang insbesondere, daß keine der existierenden Meta-SEU Konzepte zur Integration der in der Umgebung enthaltenen Werkzeuge mit einer Prozeßsteuerungskomponente enthalten. Die unterstützte Integration liegt also im Bereich der Interprodukt-Integration und nicht im Bereich der Prozeßintegration. Auch in diesem Bereich besteht also, sobald eine PSEU statt einer SEU betrieben werden soll, die bereits angesprochene Lücke zwischen der Modellierung und der Modellausführung und Prozeßausführung.

LISTE DER ABBILDUNGEN

INDEX

T