Fakultät Informatik
Institut für Systemarchitektur
Professur Rechnernetze

SAP Research
CEC Dresden

Diplomarbeit

Erweiterung der jBPM Workflow-Engine um ad-hoc Funktionalitäten

eingereicht von:

Mathias Staab
geb. 17.10.1981

August 2009

Hochschullehrer:	Prof. Dr. Alexander Schill
Betreuer:	Dr.-Ing. Iris Braun
	Dr.-Ing. Thomas Springer
Externer Betreuer:	Dipl.-Inf. Christian Sell

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik, Institut für Systemarchitektur, Lehrstuhl Rechnernetze

AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

Name, Vorname: Staab, Mathias
Studiengang: Medieninformatik
Matr.-Nr.: 2951877

Thema: „Erweiterung der jBPM Workflow-Engine um ad-hoc Funktionalitäten "

HINTERGRUND

Workflow Management Systeme (WfMS) dienen der Modellierung, Ausführung und Verwaltung von Workflows. Zur Modellierung verfügen WFMS meist über graphische Workflow-Modellierer, mit deren Hilfe zur Designzeit ein Workflow-Modell erstellt werden kann. Zur Ausführung wird dieses Modell an die Workflow Engine des WfMS weitergeleitet. Diese erstellt daraus eine Instanz und führt selbige in der Ausführungszeit aus.

Durch die Unterscheidung von Design- und Ausführungszeit entsteht das Problem, dass Workflows zur Anpassung zunächst angehalten werden müssen, da sie nur zur Designzeit modelliert und damit angepasst werden können. Anpassungen zur Laufzeit werden aktuell nur sehr bedingt unterstützt. So ermöglichen beispielsweise die sogenannten „flexiblen WfMS " die Anpassung des Workflows zur Laufzeit, beschränken sich dabei jedoch nur auf eine Menge von zur Designzeit modellierter Operationen. Diese können jedoch nur auf einen bestimmten Teil des Workflows angewendet werden. Möglich ist auch die Modellierung von alternativen Pfaden, die von der Engine zur Laufzeit auf Basis von Kontextinformationen ausgewählt werden können. Diese Ansätze sind ein erster Schritt, ermöglichen aber beispielsweise die flexible Reaktion auf unvorhersehbare Ereignisse nicht! Es kann aktuell nur das angepasst werden, was bereits modelliert wurde. Aus diesem Grund soll in der Arbeit ein Konzept entwickelt werden, mit dem beliebige Workflows ad-hoc zur Laufzeit und ohne die obigen Beschränkungen modifiziert werden können. Dabei sind die Zustände der einzelnen Workflow-Aktivitäten zu berücksichtigen. So soll es beispielsweise nicht möglich sein, bereits beendete oder gerade ausgeführte Aktivitäten zu löschen oder zu modifizieren.

ZIELSTELLUNG

Ziel der Arbeit ist die Entwicklung eines Konzeptes, das die ac-hoc Anpassung von Workflows zur Laufzeit und auf Basis von Statusinformationen ermöglicht. Dazu sollen zunächst die Anforderungen die an ein solches Konzept bestehen analysiert und der Stand der Technik bezüglich dieses Themas recherchiert werden. Daraufhin soll basierend auf den ermittelten Anforderungen ein Konzept zur ac-hoc Anpassung von Workflows erstellt und anhand eines Beispielszenarios aus dem Katastrophenmanagement prototypisch implementiert werden. Eine Validation und Selbsteinschätzung des eigenen Ansatzes soll am Ende der Arbeitet erfolgen.

SCHWERPUNKTE

- Analyse der Anforderungen an ein Konzept zur Unterstützung von ad-hoc Funktionalitäten durch die jBPM Workflow Engine
- Recherche des Standes der Technik bezüglich existierender Ansätze in diesem Bereich
- Entwicklung eines Konzeptes zur Unterstützung von ad-hoc Funktionalitäten durch die jBPM Workflow Engine
- Prototypische Implementierung des Konzepts
- Validierung des entwickelten Konzepte
- Selbsteinschätzung der Arbeit

Betreuer:	Iris Braun, TU Dresden
Externer Betreuer:	Dipl.-Inf. Christian Sell, SAP AG
Verantwortlicher Hochschullehrer:	Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
Institut:	Institut für Systemarchitektur
Lehrstuhl:	Rechnernetze
Beginn am:	01.02.2009
Einzureichen am:	30.07.2009

Unterschrift des verantwortlichen Hochschullehrers

Verteiler: 1 x HSL, 1 x Betreuer, 1 x Student

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tage dem Prüfungsausschuss der Fakultät Informatik eingereichte Diplomarbeit zum Thema:

Erweiterung der jBPM Workflow-Engine um ad-hoc Funktionalitäten

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 31.08.09

Mathias Staab.

Inhaltsverzeichnis

1 Einleitung

Workflow Management Systeme (WfMS)[1] werden heute bereits sehr oft zur Automatisierung von Geschäftsprozessen in zahlreichen Unternehmen eingesetzt. Mit Hilfe des Workflow Managements wird es ermöglicht, komplexe interne oder externe Abläufe in Unternehmen durch informationstechnische Konzepte zu beschreiben und umzusetzen.

Das Workflow Management kann als eine Grundlage der Entwicklung von Unternehmenssoftware des letzten Jahrzehnts betrachtet werden. Es ist heutzutage kaum noch ein (größeres) Unternehmen vorstellbar, in dem keine Software zum Workflow Management zum Einsatz kommt. WfMS gehören gewissermaßen zur Standardeinrichtung eines jeden Unternehmens. Entsprechend groß ist auch die Vielfalt der erhältlichen kommerziellen Systeme zum Workflow Management. Neben den proprietären Lösungen existieren auch eine Vielzahl von Open-Source Systemen, von denen eines (jBPM) in dieser Arbeit zum Einsatz kommen wird. Grundsätzlich sind jedoch die meisten dieser Systeme ähnlich aufgebaut, was durch die Standardisierungsmaßnahmen verschiedener Organisationen ermöglicht wurde. Dies ist deshalb relevant, da oft Prozesse unterstützt werden sollen, die über Unternehmensgrenzen hinweg gehen und somit verschiedene Systeme miteinander verknüpfen müssen.

1.1 Problemanalyse

Allerdings wurden durch die Verbreitung der Systeme für das Workflow und Prozessmanagement zunehmend weitere Anwendungsgebiete ersichtlich, in denen sich eine Einbindung von Workflow Management Systemen anbietet. Die Vorteile eines Workflow Management Systems lassen sich auch in Bereichen erzielen, die nichts oder nur sehr wenig mit der ursprünglichen Idee der Automatisierung von Geschäftsprozessen zu tun haben. Ein Beispiel für ein solches Anwendungsgebiet ist das Katastrophenmanagement. Auch im Katastrophenmanagement gibt es Arbeitsabläufe, die verwaltet werden müssen. Allerdings unterscheiden sich diese Arbeitsabläufe stark von den Aktivitäten, die in einem „normalen“ Geschäftsprozess in Unternehmen ablaufen. Da die üblichen WfMS auf den Einsatz in Unternehmen und zur Abbildung von Geschäftsprozessen zugeschnitten sind, ergeben sich unterschiedliche Herausforderungen, wenn ein solches System in einem anderen Kontext (z.B. dem Katastrophenmanagement) zum Einsatz kommen soll.

Genau dies soll im Rahmen dieser Arbeit geschehen. Das Forschungsprojekt SoKNOS[2] hat sich die Aufgabe gestellt, „Konzepte zu entwickeln und zu erforschen, die staatliche Organe, Unternehmen und andere Organisationen im Bereich öffentlicher Sicherheit wirksam unterstützen.“ [20] Zu diesem Themenbereich gehört unter anderem die Unterstützung der Einsatzkräfte (Feuerwehr, Technisches Hilfswerk, Rettungsdienste, ...) im Katastrophenfall. Hierbei wird eine Serviceorientierte Architektur (SOA) entwickelt, die unter anderem auch eine Workflow-Engine enthält. Diese Workflow-Engine ist dafür zuständig, den Ablauf eines

1 Workflow Management System: System zur Verwaltung von Arbeitsabläufen

2 SoKNOS – **S**ervice-**o**rientierte Archite**K**turen zur Unterstützung von **N**etzwerken im Rahmen **O**effentlicher **S**icherheit – `http://www.soknos.de/`

Einsatzes zu koordinieren. Hier kommt jedoch eine Einschränkung der üblichen Workflow-Engines zum Tragen, die den Einsatz in diesem sehr kritischen Umfeld stark einschränkt.

In allen herkömmlichen WfMS existiert eine strikte Trennung von „Designzeit“ und „Laufzeit“. Während der Designzeit werden „Workflow Modeling Tools“ dazu eingesetzt, eine Prozess Definition zu erstellen. Diese Prozess Definition ist Grundlage für den bearbeitenden Workflow, da hier die relevanten Aktivitäten und deren zeitliche Abfolge festgelegt werden. Während der Laufzeit wird diese zuvor erstellte Prozess Definition abgearbeitet, indem aus der Definition eine Prozess Instanz generiert und diese durch eine Workflow Engine ausgeführt wird. Näheres zu den Begriffen und der Vorgehensweise kann in Kapitel 2.1 nachgelesen werden.

Dieses Vorgehen bewirkt, dass keine Änderungen am definierten Workflow während der Ausführung vorgenommen werden können. Im Kontext einer herkömmlichen Unternehmensumgebung ist dies nicht weiter problematisch, da hier nicht mit Änderungen zu rechnen sind. Stattdessen werden die Prozesse vorher ausführlich geplant und sollen auch genau so ablaufen. Im Anwendungsgebiet Katastrophenmanagement lässt sich dieses Vorgehen allerdings nicht verwirklichen. Hier ist durchaus damit zu rechnen, dass sich Änderungen ergeben, da nicht alle denkbaren Alternativen eines Einsatzes bereits zur „Designzeit“ bekannt sind. Als Beispiel ist in Abbildung 1.1 ein vereinfachter Einsatzplan zu erkennen, in dem das Vorgehen bei der Evakuierung von Personen aus einem Katastrophengebiet beschrieben wird. Dieser Einsatzplan könnte zur Designzeit in eine adäquate Prozess Definition zur Abarbeitung in einem WfMS übertragen werden.

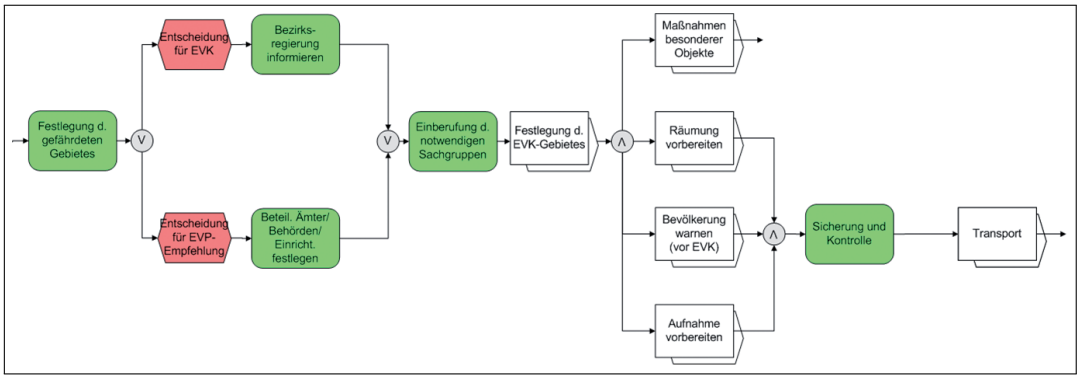

Abbildung 1.1: Einsatzplan zur Evakuierung von Personen

In Abbildung 1.2 ist nun jedoch zu erkennen, dass sich der ursprüngliche Plan nicht genau so durchführen lässt. Eine zusätzliche Information wurde erst während des Einsatzes ersichtlich. Auf Grund einer erhöhte Giftgaskonzentration über dem Evakuierungsgebiet muss parallel zum ursprünglich geplanten Vorgehen eine Dekontamination des betroffenen Gebietes durchgeführt werden.

Dieser neue Kontext konnte zur Designzeit nicht im Einsatzplan — also der Prozess Definition — berücksichtigt werden. Solche unvorhergesehenen Ereignisse sind während eines Einsatzes im Katastrophenmanagement an der Tagesordnung. Aus diesem Grund muss eine flexible Anpassung der Workflow Definition zur Laufzeit möglich sein. Es ist der Ansatzpunkt dieser Arbeit, Funktionalitäten für eine solche flexible „ad-hoc“ Adaption für die verwendete Workflow-Engine zu entwerfen.

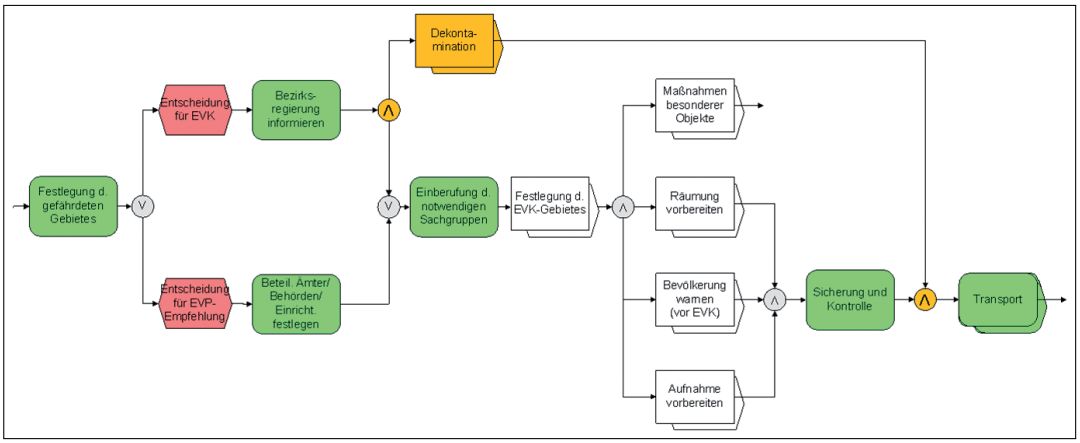

Abbildung 1.2: Einsatzplan zur Evakuierung von Personen - Angepasst

1.2 Zielstellung

Durch die Architektur der Workflow-Engine im ursprünglichen Zustand würde eine nötige Anpassung bedeuten, dass der Workflow angehalten werden muss, um Änderungen an der Definition durchführen zu können. Dies ist jedoch in einem zeitkritischen Umfeld wie dem Katastrophenmanagement keine Option. Mit den „out-of-the-box“ Lösungen der üblichen Workflow Management Systeme sind keine flexiblen Änderungen zur Laufzeit möglich — aus diesem Grund muss die verwendete Workflow-Engine um die benötigten Funktionalitäten zur ad-hoc Anpassung während der Laufzeit erweitert werden. Da die im SoKNOS System eingesetzte Workflow-Engine bereits fest steht — die jBPM Workflow-Engine — kommt diese auch dieser Arbeit zum Einsatz.

Ziel ist es, dass während der Laufzeit „ad-hoc“ Anpassungen am Workflow vorgenommen werden können. Hierfür müssen unterschiedliche Operationen angeboten werden, mit denen sich die für den Anwendungsbereich relevanten Adaptionen durchführen lassen.

1.3 Übersicht über die Arbeit

Nach der Einleitung werden in Kapitel 2 zuerst die Grundlagen des Workflow Managements allgemein erläutert. Hier werden alle nötigen Begriffe und Konzepte des Workflow Managements vorgestellt. Im Anschluss werden die Grundlagen der jBPM Workflow-Engine erläutert, auch hier kommen die relevanten Begriffe und Konzepte zur Sprache. In Kapitel 3 werden die Anforderungen an das zu entwickelnde Konzept zur Unterstützung von ad-hoc Funktionalitäten durch die jBPM Workflow Engine definiert, darauf folgend werden in Kapitel 4 einige verwandten Arbeiten aus dem Bereich vorgestellt. Die Ansätze werden primär nach ihrer Relevanz im Anwendungsgebiet bewertet. Das Kapitel 5 widmet sich dem Entwurf der ad-hoc Funktionalitäten. Insbesondere wird die Repräsentation des Status beschrieben und die relevanten Korrektheits- und Konsistenzkriterien werden definiert. Im Anschluss werden die eigentlichen Operationen vorgestellt. Das Kapitel 6 widmet sich der Validierung und Bewertung des entworfenen Ansatzes und Kapitel 7 gibt eine Zusammenfassung über die Arbeit, sowie einen Ausblick auf mögliche zukünftige Entwicklungen.

2 Grundlagen

Um eine Workflow Engine (bzw. die jBPM Workflow-Engine im Speziellen) um Funktionalitäten zur ad-hoc Anpassung erweitern zu können, ist es zunächst wichtig die Grundlagen der WfMS zu betrachten und die zugehörigen begrifflichen Konventionen zu definieren.

Da auf diesem Gebiet eine sehr große Anzahl von unterschiedlichen Herstellern und Forschungsgruppen arbeiten, entwickelten sich im Lauf der Zeit viele verschiedene Architekturen und Bezeichnungen für deren Komponenten. Aus diesem Grund wurden verschiedene Organisationen gegründet, die sich der Standardisierung im Bereich der WfMS annahmen. Eine Übersicht über die vorhandenen Standardisierungs-Organisationen ist bei Mendling u. a. [12] zu finden. In dieser Arbeit wird primär die Terminologie der Workflow Management Coalition (WfMC)[1] zum Einsatz kommen und auf die relevanten Thematiken übertragen, da sich diese als Quasi-Industriestandard durchgesetzt hat. In Abbildung 2.1 sind die wichtigsten von der WfMC definierten Bezeichnungen dargestellt.

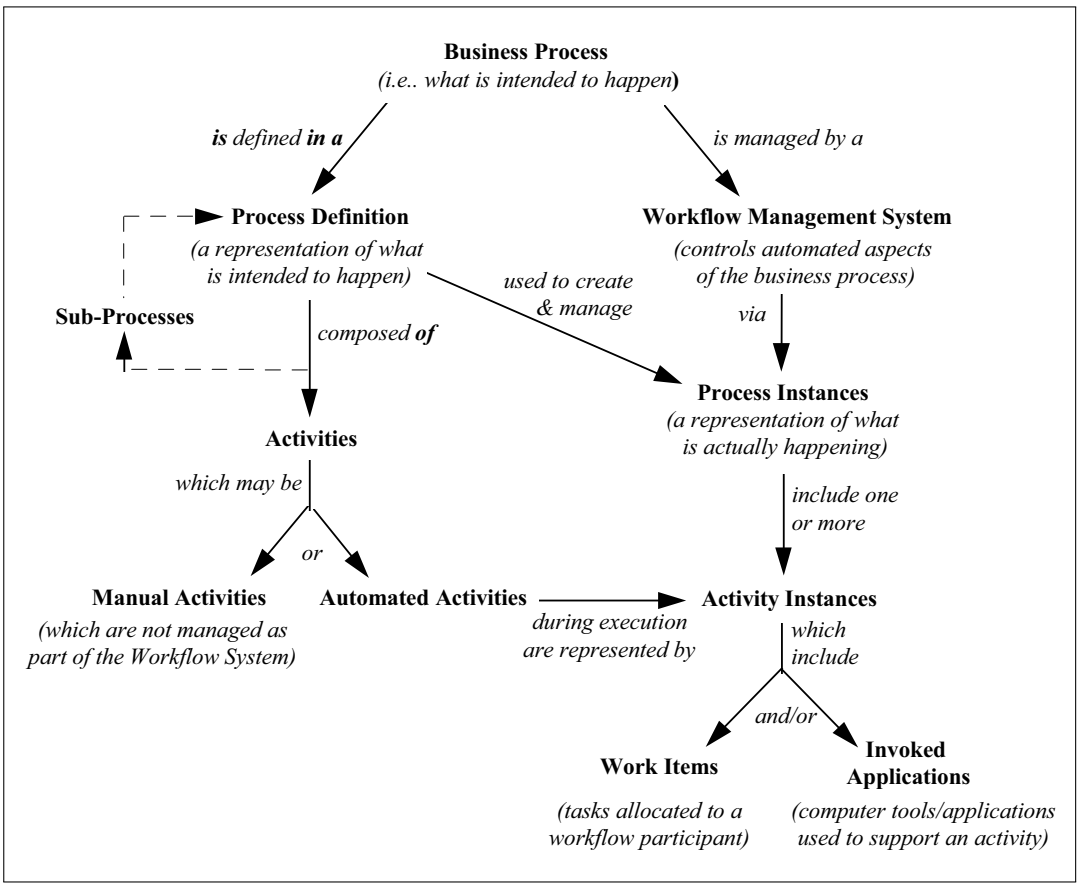

Abbildung 2.1: Verhältnisse zwischen den Grundbegriffen [24]

1 Workflow Management Coalition: `http://www.wfmc.org/`

In den folgenden Abschnitten werden zuerst die nötigen Begriffe definiert und gleichzeitig der grundsätzliche Aufbau von Workflow Management Systemen erläutert. Im Anschluss folgt eine nähere Betrachtung der jBPM Workflow-Engine im Besonderen, wobei die wichtigsten Komponenten und grundlegenden Konzepte genauer vorgestellt werden.

2.1 Workflow Management

Das den folgenden Definitionen zugrunde liegende Dokument [24] wurde im Jahr 1999 durch die WfMC veröffentlicht und deckt alle gängigen Begriffe aus dem Bereich der Workflow Systeme ab. Die ursprüngliche Bezeichnung „Workflow Management“ wurde allerdings vor einiger Zeit durch das Business Process Management (BPM) ersetzt [siehe 21]. Es gibt jedoch durchaus einen Unterschied zwischen Workflows und Business Processes (Geschäftsprozessen). In der Literatur werden WfMS und Business Process Management Systeme (BPMS) nicht weiter differenziert — beide Begriffe bezeichnen die gleichen Konzepte, welche in den folgenden Abschnitten näher erläutert werden.

2.1.1 Grundlegende Konzepte

Jedem Workflow liegt ein entsprechender *Geschäftsprozess* zugrunde. Ein Geschäftsprozess ist eine Menge miteinander verknüpfter Arbeitsvorgänge, die gemeinsam ein definiertes Ziel erreichen sollen. Üblicherweise geschieht dies im Kontext einer Organisationsstruktur, die die Rollen und Verhältnisse der beteiligten Entitäten zueinander festlegt. Ein solcher Prozess kann wiederholt durchlaufen werden.

Ein Geschäftsprozess muss nicht zwangsläufig durch computergesteuerte Anlagen unterstützt bzw. ausgeführt werden. Falls dies jedoch der Fall ist, spricht man von einem *Workflow*. Ein Workflow ist die (computergestützte) Automatisierung eines Geschäftsprozesses. Im Verlauf des Workflows werden Dokumente, Informationen oder Aufgaben nach definierten Regeln zwischen den Bearbeitern vermittelt.

Um die Automatisierung eines Geschäftsprozesses vornehmen zu können, muss zuerst eine Repräsentation des Prozesses erstellt werden, die von einem WfMS interpretiert werden kann. Die so entstandene *Prozess Definition* besteht aus einem Netz von Aktivitäten und deren Verhältnisse zueinander. Zudem beinhaltet eine Prozess Definition die Bedingungen zum Start bzw. der Terminierung des Prozesses und Informationen über die einzelnen Aktivitäten (wie zugeordnete Bearbeiter, verknüpfte Anwendungen und Daten, etc.). Es existieren unterschiedliche Ansätze, wie eine Prozess Definition erstellt und verwaltet werden kann.

Üblicherweise wird mittels graphischer Editoren eine zum Geschäftsprozess passende Prozess Definition modelliert. Die fertige Prozess Definition wird in einer Datei oder Datenbank gespeichert und von einem WfMS abgerufen, um die Instanzen des Prozesses zu erzeugen. In einem klassischen WfMS sind keine Änderungen an der entsprechenden Prozess Definition mehr möglich, sobald eine Instanz des Prozesses gestartet wurde.

Als *Aktivität* wird hierbei ein einzelner Arbeitsvorgang bezeichnet, der einen logischen Schritt innerhalb des Prozesses festlegt. Im Bereich der Workflow Systeme bezeichnet der Begriff immer eine automatisierte Aktivität, die menschliche oder maschinelle Ressourcen zur Prozessausführung benötigt. In diesem Fall kann eine Aktivität aus einem oder mehreren *Work Items* bestehen, die — falls menschliche Ressourcen benötigt werden — einem oder mehreren *Bearbeitern* zugeordnet werden.

Bearbeiter sind Ressourcen, die Arbeitsvorgänge durchführen. Ein solcher Arbeitsvorgang wird durch eine *Aktivitätsinstanz* — die Repräsentation einer Aktivität während der Ausführung eines Prozesses — beschrieben.

Dieser Arbeitsvorgang wird üblicherweise durch eine oder mehrere Work Items repräsentiert, die dem Bearbeiter durch seine *Worklist* zugewiesen werden. Die Worklist ist somit ein Verzeichnis der Work Items, die mit einem bestimmten Bearbeiter verknüpft sind (oder ggf. einer Gruppe von Bearbeitern, die eine gemeinsame Worklist besitzen).

Soll der Prozess ausgeführt werden, generiert die Workflow Engine aus der Prozess Definition eine *Prozessinstanz*. Die Prozessinstanz verkörpert somit den definierten Prozess zur Laufzeit und beinhaltet alle mit diesem Vorgang verknüpften Daten. Des weiteren besitzt jede Prozessinstanz Schnittstellen, über die der entsprechende Prozess eindeutig angesprochen werden kann.

Das zugrunde liegende Software-System, welches Workflows sowohl definiert und erstellt als auch deren Ausführung überwacht, wird als *Workflow Management System* bezeichnet. Die eigentliche Ausführung erfolgt auf einer oder mehreren Workflow Engines. Das System interpretiert die Prozess Definition und interagiert mit den Bearbeitern; außerdem steuert es die Aufrufe von externen (IT) Anwendungen. Der Zugriff auf das System erfolgt durch eine entsprechende Workflow Anwendung.

2.1.2 Der Prozess

Da es *Prozesse* sind, die durch ein Workflow Management System bearbeitet werden, ist es nötig einige Begrifflichkeiten aus dem Bereich der Prozesse zu definieren. Grundsätzlich ist der Prozess in einem WfMS die formalisierte Ansicht eines Geschäftsprozesses, der durch eine aufeinander abgestimmte Menge von Aktivitäten (parallel oder sequentiell vernetzt) repräsentiert wird.

Möchte man einen Teil des Gesamtprozesses abgrenzen, um zum Beispiel einen komplexen Prozess grob zu strukturieren, kommt ein sogenannter *Sub Prozess* zum Einsatz. Ein Sub Prozess ist ein Prozess, der innerhalb eines anderen (initiierenden) Prozesses (oder Sub Prozesses) ausgeführt oder aufgerufen wird und Teil des (initiierenden) Gesamtprozesses ist. In der Regel sind mehrere Ebenen von Sub Prozesse möglich. Übertragen auf den Anwendungsbereich der Workflows spricht man hierbei auch von einem Sub-Workflow.

Wichtig für die Bearbeitung eines Prozesses ist immer auch der aktuelle Status der beteiligten Entitäten, sowie der Status des Prozesses im Allgemeinen. Vor allem für die ad-hoc Anpassung eines Workflows ist es notwendig, genaue Informationen über diesen Status zu verwalten. Hierbei kommen die Konzepte des *Prozess-Status* und des *Aktivitätsstatus* zum Einsatz. Diese sind die Repräsentation der internen Bedingungen, die den Status einer Prozessinstanz bzw. Aktivitätsinstanz zu einem bestimmten Zeitpunkt definieren. Als *Zustandsübergang* wird hierbei die Bewegung von einem internen Zustand (einer Prozess- oder Aktivitätsinstanz) zu einem Anderen bezeichnet, die die Veränderung im Status des Workflows widerspiegelt — zum Beispiel der Start einer bestimmten Aktivität. Zustandsübergänge können unter anderem durch externe Ereignisse, API Aufrufe oder Ablaufentscheidungen der Workflow Engine ausgelöst werden.

2.1.3 Aufbau und Struktur

Um die vielfältigen Ausprägungen von Geschäftsprozessen in einer Prozess Definition abbilden zu können, sind verschiedene strukturelle Konstrukte notwendig. Hierbei geht es primär um Vorgänge die parallel oder sequentiell ablaufen und daher auch so modelliert werden müssen.

Die Modellierung erfolgt durch spezielle Übergänge zwischen den einzelnen Aktivitäten. Sogenannte *Transitionen* stellen die Verbindung der Aktivitäten zueinander her (siehe Abbildung 2.2. Wenn während der Ausführung einer Prozessinstanz eine Aktivität beendet ist, wird der Pfad der Ausführung über die — mit der Aktivitäten verknüpften — Transition geleitet und startet somit die anschließende Aktivität. Dies ist natürlich nur im einfachsten Fall, einer simplen Sequenz von Aktivitäten, möglich. Komplexere Konstruktionen wie beispielsweise parallele oder bedingte Verzweigungen sind jedoch auch mittels Transitionen und zugehörigen *Übergangsbedingungen* modellierbar. Übergangsbedingungen sind logische Ausdrücke, die von einer Workflow Engine ausgewertet werden können, um die Abfolge bei der Ausführung von Aktivitäten zu bestimmen.

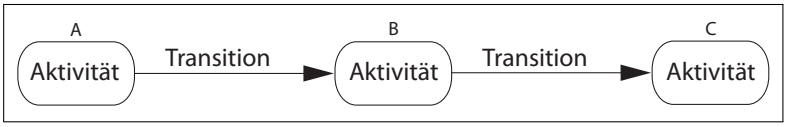

Abbildung 2.2: Struktur: Sequenz aus drei Aktivitäten

Der einfachste Fall der Strukturierung eines Prozesses ist die *Sequenz* — vgl. Abbildung 2.2. Eine Sequenz ist der Teil eines Prozesses, in dem mehrere Aktivitäten sequentiell ausgeführt werden — hier $(A \rightarrow B \rightarrow C)$. Im Gegensatz dazu ist eine *Nebenläufigkeit* der Abschnitt, in dem zwei oder mehrere Aktivitäten im Workflow parallel ausgeführt werden. Üblicherweise wird diese Funktionalität durch Parallele Verzweigung und folgende Synchronisierung realisiert.

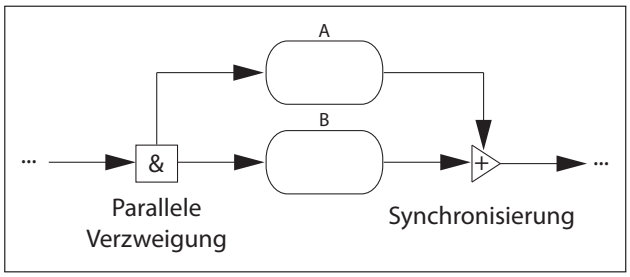

Abbildung 2.3: Struktur: Parallele Verzweigung, Nebenläufigkeit

Eine *Parallele Verzweigung* oder UND-Verzweigung (&) ist eine Stelle im Workflow, an der ein einzelner Pfad der Ausführung (bzw. Kontroll-Thread) in zwei oder mehrere Zweige (bzw. Threads) aufgeteilt wird — siehe Abbildung 2.3. Diese Zweige werden parallel durchlaufen, was die gleichzeitige Ausführung mehrerer Aktivitäten — also eine Nebenläufigkeit von $(A \wedge B)$ — ermöglicht. Am Ende jeder parallelen Ausführung von Aktivitäten müssen die entsprechenden Pfade wieder zusammengeführt werden. Hierbei spricht man von einer *Synchronisierung* (+) — verschiedene parallel laufende Zweige des Workflows verschmelzen an dieser Stelle wieder zu einem einzelnen. Die Umsetzung erfolgt meist in der Art, dass die Ausführung an der Stelle der Synchronisierung so lange wartet, bis alle parallelen Aktivitäten abgeschlossen wurden und somit als nächster Schritt wieder sequentiell gearbeitet werden kann.

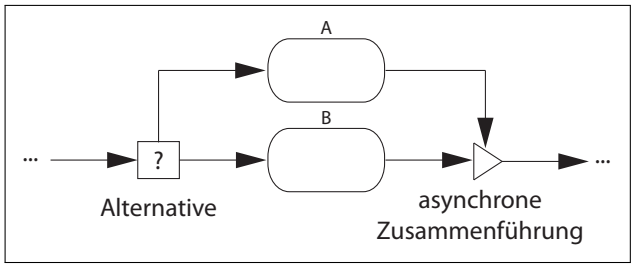

Abbildung 2.4: Struktur: Alternative

Im Gegensatz zur UND-Verzweigung, bei der bedingungslos alle verknüpften Pfade ausgeführt werden, steht die ODER-Verzweigung bzw. *Alternative Verzweigung* — siehe Abbildung 2.4, die alternative Ausführung der Aktivitäten ($A \vee B$). Wenn sich ein Workflow in mehrere alternative Pfade verzweigt, muss entschieden werden welcher dieser Zweige ausgeführt werden soll. Die ODER-Verzweigung ist die Stelle, an der diese Entscheidung getroffen wird. Hierfür werden meist unterschiedliche Bedingungen (in Abbildung 2.4 mit „?“ gekennzeichnet) ausgewertet, die den jeweils richtigen Workflow-Zweig zurückliefern. Analog zur Parallelen Verzweigung muss auch in diesem Fall der entsprechende Workflow-Abschnitt wieder abgeschlossen werden. Hierbei kommt eine *Asynchrone Zusammenführung* zum Einsatz. In diesem Fall werden zwei oder mehrere Alternativen des Workflows wieder in einen einzelnen gemeinsamen Pfad zusammengeführt. Es ist keine Synchronisierung notwendig, da die Aktivitäten nicht parallel ausgeführt wurden.

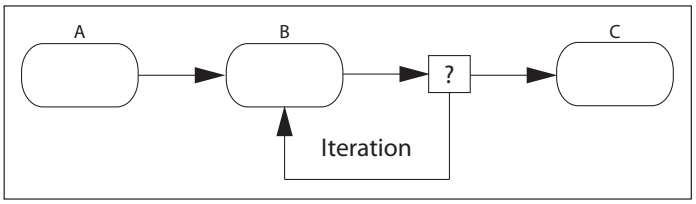

Abbildung 2.5: Struktur: Iteration

Weiterhin ist es möglich, dass eine Aktivität oder ein Sub-Workflow beliebig oft wiederholt werden soll. Diese *Iteration* erstellt einen Zyklus von Aktivitäten im Workflow. So lange eine bestimmte Bedingung erfüllt bzw. nicht erfüllt ist, wird die periodische Wiederholung des umschlossenen Abschnittes erzwungen (siehe Abbildung 2.5). Hier wird die Aktivität (B) so lange wiederholt, wie die Bedingung (?) erfüllt ist ($A \rightarrow B, B, B, B \ldots \rightarrow C$).

2.1.4 Weitergehende Konzepte

Im breiten Feld des Workflow Management existieren noch diverse weitergehende Konzepte, die auch und vor allem für die Konzeption der ad-hoc Funktionalitäten von Bedeutung sind. Von grundlegender Relevanz ist der Umstand, dass in klassischen Workflow Systemen die beiden Zeitspannen der *Geschäftsprozessmodellierung* und der *Vorgangsbearbeitung* als grundsätzlich unterschiedlich und sich nicht überschneidend definiert werden. Die Geschäftsprozessmodellierung ist hierbei die Zeitspanne, in der die Beschreibung eines Prozesses definiert oder modifiziert wird — also die Erstellung der Prozess Definition. Die Vorgangsbearbeitung wiederum ist die Zeitspanne, in der der Prozess in Betrieb ist und Prozessinstanzen erstellt und/oder verwaltet werden. Die unterschiedlichen Phasen des Workflow Management sind in Abbildung 2.6 erkennbar. Oft werden auch die Begriffe „Designzeit“ und „Laufzeit“ als

Synonym für die beiden Phasen genutzt. Um ad-hoc Funktionalitäten zu ermöglichen, ist es unabdingbar, diese beiden ursprünglich getrennten Zeitspannen zu verschmelzen. Die eigentliche Natur einer ad-hoc Anpassung ist es, dass Aspekte der Geschäftsprozessmodellierung *während* der Vorgangsbearbeitung durchgeführt werden können — also die Prozess Definition im laufenden Betrieb des Prozesses Änderungen erfährt.

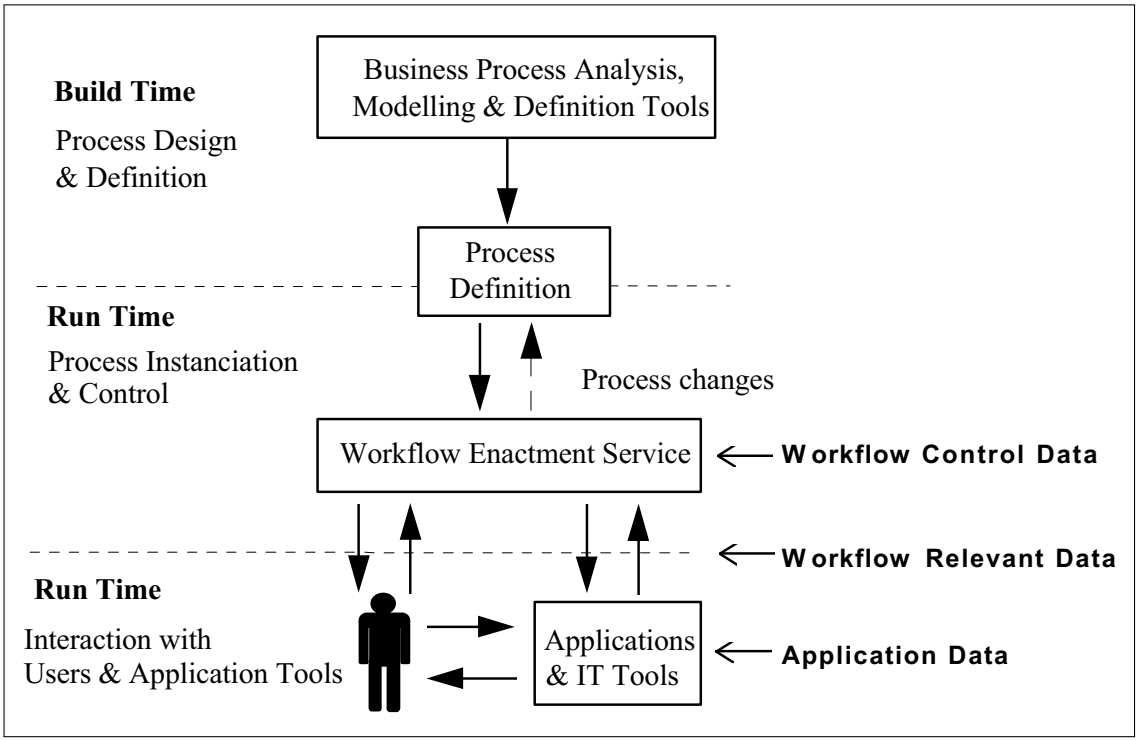

Abbildung 2.6: Phasen und Datentypen im Workflow Management [24]

Hierbei muss es möglich gemacht werden, dass die Workflow Anwendung auf Daten zugreifen und diese verändern kann, die während der Ausführung üblicherweise nur der Engine zur Verfügung stehen. Bei klassischen Workflow Systemen hat die Workflow Anwendung nur Zugriff auf *Workflow Relevant Data*. Das sind die Daten, die von einem WfMS genutzt werden, um mögliche Zustandsübergänge einer Prozessinstanz zu bestimmen — beispielsweise innerhalb von Übergangsbedingungen oder bei der Zuweisung von Bearbeitern. Diese Daten können auch für andere Aktivitäten oder Prozessinstanzen zugänglich gemacht werden, die zum Beispiel interne Ablaufentscheidungen davon abhängig machen. Darüber hinaus können auf diesem Weg Bezugsgrößen zwischen unterschiedlichen Aktivitäten übermittelt werden. Diese Daten können — wie oben angedeutet — von einer Workflow Anwendung oder Workflow-Engine problemlos verändert werden. Das dynamische Verhalten der Workflow Relevant Data zur Laufzeit wird auch als *Datenfluss* bezeichnet.

Im Gegensatz dazu stehen die *Workflow Control Data*. Dies sind die Daten, die von einem WfMS bzw. einer Workflow-Engine intern verwaltet werden. Im klassischen Fall sind sie nicht für Workflow Anwendungen zugänglich. Sie repräsentieren den dynamischen Status des Workflow Systems und der Prozessinstanzen — beispielsweise Statusinformationen über jede Prozessinstanz oder den Status von Aktivitätsinstanzen. Zudem ist die eigentliche Struktur des Workflow Schemas Teil der Workflow Control Data, das Verhalten dieser Daten zur Laufzeit wird auch als *Kontrollfluss* bezeichnet. In Abbildung 2.6 werden auch die unterschiedlichen Datentypen im Workflow Management verdeutlicht.

Um nun zur Laufzeit Änderungen am Aufbau des Prozesses zu ermöglichen ist es unabdingbar, dass die entsprechende Anwendung Zugriff auf alle für die Anpassung relevanten Daten — also auch Workflow Control Data — hat. Nur so ist es möglich zu überprüfen, welche Änderungen überhaupt grundsätzlich zulässig sind. Zudem müssen bei einer erfolgten ad-hoc Anpassung diese Daten natürlich aktualisiert werden.

Ein weiterer wichtiger Ansatz in Workflow Systemen und für ad-hoc Anpassungen ist das Konzept der *Events*. Als Events bezeichnet man das Auftreten einer bestimmten (internen oder externen) Bedingung, die eine oder mehrere Aktionen des WfMS auslöst. Ein Ereignis besteht aus zwei Elementen: Ein Trigger — die Erkennung des Eintritts von vordefinierten Umständen — und eine vordefinierte Action, die darauf folgt. Zum Beispiel kann das Eintreffen einer bestimmten E-Mail die Weiterführung einer zuvor wartenden Aktivität auslösen. Außerdem können intern auftretende Events beispielsweise dazu verwendet werden, den aktuellen Status der beteiligten Entitäten zu bestimmen.

2.2 jBPM Workflow-Engine

Im folgenden Abschnitt werden zuerst die jBPM Workflow-Engine vorgestellt und daraufhin allgemeine Informationen zum Projekt und der Implementierung gegeben. Im Anschluss wird die Struktur eines Workflows in jBPM beschrieben und die relevanten Bestandteile der Engine bzw. der Workflow-Beschreibungssprache werden aufgezeigt.

2.2.1 Allgemeines

Die jBPM Workflow-Engine ist ein in Java geschriebenes Open Source Projekt und steht unter der LGPL[2]. Die Entwicklung findet aktiv bereits seit Januar 2003 statt. Seit 2004 wird das Projekt unter dem Dach von JBoss[3] weiter entwickelt und im Rahmen der JBoss Enterprise Middleware Suite (JEMS) angeboten. Der ebenfalls enthaltene JBoss Application-Server ist üblicherweise die J2EE Plattform, auf der die jBPM Workflow-Engine ausgeführt wird. Grundsätzlich ist die Engine jedoch auf jedem beliebigen J2EE Application-Server lauffähig, bzw. auch ganz ohne Application-Server einsatzfähig.

Für diese Arbeit ist die Version 3.1.1 relevant (neuste stabile Version: 3.2), aktuell findet jedoch bereits die Entwicklung der Version 4.0 statt. Diese befindet sich allerdings noch im Beta Stadium. Die Version 3.1.1 kommt deshalb zum Einsatz, da diese auch im SoKNOS System eingesetzt wird und die im Rahmen dieser Arbeit getroffenen Erweiterungen in das System übertragen werden sollen.

Mit dem Sprung auf Version 4.0 wird das Konzept der Process Virtual Machine (PVM) als universelle Workflow-Engine eingeführt, die in der Lage ist, fast jede beliebige Workflow-Beschreibungssprache auszuführen [vgl. 5, 9]. Der Begriff PVM kam bereits in früheren Versionen zur Sprache, allerdings wurde das Konzept noch nicht vollständig umgesetzt. Auch die Version 3.1 ist daher bereits in der Lage, unterschiedliche Workflow Sprachen (z.B. BPEL[4] oder Pageflow — vgl. Abbildung 2.7) auszuführen, die Implementierung steckt allerdings noch in den Kinderschuhen. Daher ist der Einsatz der speziell auf jBPM zugeschnittenen Sprache jBPM Process Definition Language (jPDL) vorteilhaft und wird auch im Laufe dieser Arbeit

2 GNU Lesser General Pulic License: `http://www.gnu.org/licenses/lgpl.html`
3 JBoss Community: `http://www.jboss.org/`
4 BPEL: WS-Business Process Execution Language, XML basierte Workflow Sprache

angenommen. Die Proprietäre Sprache jPDL baut auf der angesprochenen PVM auf und ist klein, flexibel und leicht erweiterbar. Es ist möglich mittels jPDL einen Zustandsautomaten vollständig abzubilden. Die meisten Anforderungen an Geschäftsprozesse können also umgesetzt werden. Allerdings werden die resultierenden XML Dateien relativ schnell unübersichtlich, außerdem ist der graphische Editor (zumindest bis Version 3.2) teilweise fehlerbehaftet und noch verbesserungswürdig. Weitere Informationen zum Aufbau und der Struktur von jPDL werden in den folgenden Abschnitten gegeben.

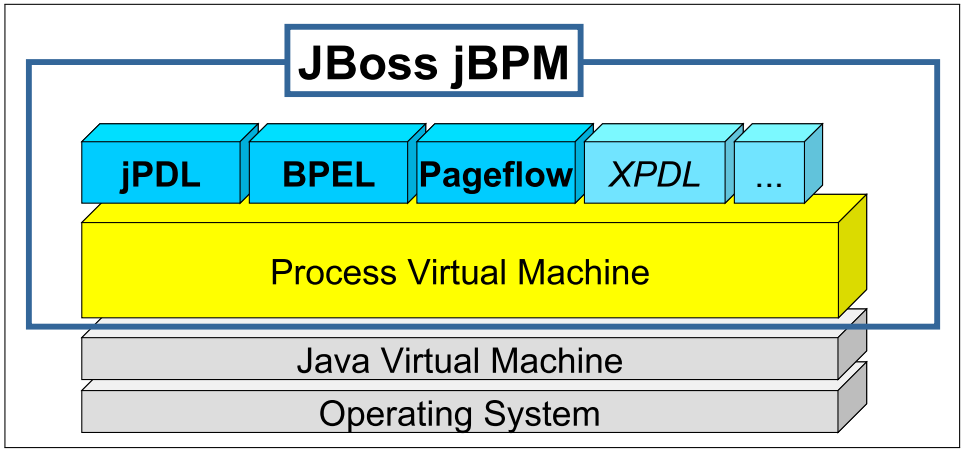

Abbildung 2.7: jBPM, die PVM und jPDL, [4]

Der Kern der jBPM Engine ist die Implementierung eines Zustandsautomaten in Java. Dieser Kern wird durch gewöhnliche Java Objekte (POJO) aufgebaut. Die durch die Beschreibungssprache übergebenen Prozesse werden also intern durch Java-Modelle repräsentiert. Durch die Einbindung des Hibernate Frameworks[5] ist es möglich unterschiedlichste Datenbanken einzubinden, außerdem wird so auch die Persistenz der Prozesse gewährleistet. Die Verwendung von Hibernate und den damit verbundenen Cashing Mechanismen kann jedoch zu Problemen bei der ad-hoc Adaption eines Prozesses führen, näheres dazu in Kapitel 6.

Grundsätzlich wird die jBPM Engine von den beteiligten Entwicklern so klein und flexibel wie möglich gehalten und ist stark auf Erweiterungen ausgelegt. Durch dieses Vorgehen wird der Fokus in der Entwicklung eher auf die Stabilität als auf neue Features gelegt. Ein Vorteil von jBPM ist zudem die große und aktive Community, zusätzlich ist kommerzieller Support (JBoss Enterprise) möglich.

2.2.2 Struktur

Die Struktur der jBPM Workflow-Engine lässt sich in Definitions Daten — Konzepte die primär für die Geschäftsprozessmodellierung wichtig sind — und Laufzeit Daten — Konzepte die während der Vorgangsbearbeitung zum Tragen kommen — unterteilen. In Tabelle 2.1 ist eine kurze Übersicht über diese Konzepte dargestellt. Die in Abschnitt 2.1 beschriebenen allgemeinen Vorgaben der WfMC werden weitgehend durch die jBPM Workflow-Engine umgesetzt. In den folgenden Abschnitten wird näher auf die Lösungen eingegangen, die von jBPM eingesetzt werden. Grundlage für die Ausführungen ist der jBPM jPDL User Guide 3.2.3 [9].

5 Hibernate: Open-Source-Persistenz für Java, `http://www.hibernate.org/`

Definitions Daten	Laufzeit Daten
Node	Token
Transition	Signal
Action	Process Instance

Tabelle 2.1: Struktur von jBPM

2.2.3 Definitions Daten

Nodes

Innerhalb von jBPM erfolgt die Repräsentation von Aktivitäten durch das Konzept der Nodes, also Knoten. Diese Implementierung hat zwei Hauptaufgaben:

- Die Ausführung von gewöhnlichem Java Code (Die Funktion der Node)
- Den Fortschritt in der Vorgangsbearbeitung propagieren (Konzept: Token)

Die Art der Propagierung kann unterschiedliche Formen annehmen. Das Signal, welches die Weiterführung des Prozesses auslöst, kann sowohl intern auftreten, als auch von einer externen Quelle an den Knoten übermittelt werden. Außerdem kann der Token — der Pfad der Ausführung — sowohl über einen der definierten Übergänge (Transitionen) weitergeführt, als auch im Rahmen einer parallelen Verzweigung (Fork-Node) aufgeteilt werden. Durch diesen Vorgang entstehen zusätzliche Tokens bzw. Ausführungspfade, die durch Synchronisierung (Join-Node) wieder beendet werden. Der primäre Pfad der Ausführung wird durch den ersten Knoten (Start-Node) initialisiert und durch den letzten Knoten (End-Node) abgeschlossen und terminiert.

Ein Nachteil dieser Implementierung in Bezug auf die zukünftige ad-hoc Adaption ist, dass Asynchrone Zusammenführungen keinen expliziten Node-Typen besitzen. Die Architektur einer ProcessDefinition mittels jPDL sieht vor, dass eine Node mit mehreren eingehenden Transitionen automatisch eine Asynchrone Zusammenführung darstellt — wenn sie keine Join-Node (Synchronisierung) ist. Dies ist deshalb problematisch, weil eine Adaptionsoperation auf die Unterscheidung zwischen Aktivitäten und anderen strukturellen Konstruktionen angewiesen ist.

Generell können Nodes die Laufzeitstruktur der Prozessinstanz verändern — also Tokens erstellen und beenden, Tokens in andere Nodes platzieren und Tokens über Transitionen weiterleiten. Es ist zu erkennen, dass die in Abschnitt 2.1.3 vorgestellten strukturellen Bestandteile wie Aktivitäten, Verzweigungen und Synchronisierungen durch unterschiedliche Ausprägungen der Klasse Node in jBPM repräsentiert werden. Die Node hat deshalb zwei unterschiedliche Aufgaben. Zum einen bildet sie (zusammen mit den Transitionen) die Architektur der ProcessDefinition. Zum anderen implementiert sie sowohl das Konzept der Aktivitäten als auch die Konzepte der Verzweigungen. Diese Vermischung ist die Grundlage des *Graph Oriented Programming (GOP)* [9, Kapitel 4] und hat zur Folge, dass eine Node nicht zwangsläufig eine Aktivität sein muss, jede Aktivität aber immer eine Node ist.

Transitionen

Transitionen (Übergänge) stellen die Verbindung zwischen den unterschiedlichen Knoten her. Jede Transition hat jeweils genau einen Quell- und Ziel-Knoten. Jeder Transition kann optional ein eindeutiger Namen zugewiesen werden, was jedoch zur Notwendigkeit wird falls

aus einem Knoten mehr als eine Transition hervorgeht. Dies ist vor allem bei parallelen und bedingten Verzweigungen (Fork-Node und Decision-Node) der Fall. Im Kontext der bedingten Verzweigung kommt ein weiteres strukturelles Konzept zum Tragen — die Übergangsbedingung oder Transition Condition. Hierbei wird bei jBPM zwischen zwei Methoden der Modellierung unterschieden. Entweder wird die Entscheidung, über welche Transition die Vorgangsbearbeitung weiter laufen soll, intern durch in der Prozessdefinition festgelegte Spezifikationen oder durch externe Teilnehmer gefällt.

Im Falle der internen Bearbeitung gibt es wiederum unterschiedliche Möglichkeiten die Entscheidung vorzunehmen. Die erste Möglichkeit ist es, an jeder Transition entsprechende Übergangsbedingungen zu modellieren. Die erste Transition, an der die entsprechende `condition` den Wert `true` zurück gibt, wird ausgelöst. Die anderen Möglichkeiten verwenden verschiedene Funktionen (z.B. `DecisionHandler`), welche den entsprechenden eindeutigen Namen der zu wählenden Transition liefern sollen. Gibt es keine Entscheidung, wird standardmäßig immer die erste Transition der Liste ausgewählt.

```
1  <process-definition>
2          <start-state>
3                  <transition to='phase one' />
4          </start-state>
5          <state name='phase one'>
6                  <transition to='phase two' />
7          </state>
8          <state name='phase two'>
9                  <transition to='phase three' />
10         </state>
11         <state name='phase three'>
12                 <transition to='end' />
13         </state>
14         <end-state name='end' />
15 </process-definition>
```

Listing 2.1: Einfache ProcessDefinition mit jBPM-jPDL

Actions

Um zusätzliche Funktionalitäten zum modellierten Geschäftsprozess hinzufügen zu können, ermöglicht es die jBPM Engine gewöhnlichen Java Code in einer Action zu kapseln, welche bei Auftritt eines bestimmten Ereignisses ausgeführt werden soll. Eine Action kann entweder direkt in eine Node platziert oder auch allgemein mit einem Event (Ereignis) verknüpft werden. Im ersten Fall ist die entsprechende Action auch für die Weiterführung des Pfades der Ausführung (Propagierung des Token) verantwortlich.

Actions, die mit einem Event verknüpft sind, werden ausgeführt sobald das spezifizierte Ereignis auftritt. (vgl. Entwurfsmuster: Observer) Primäre Ereignisse, mit denen Actions verknüpft werden können, sind beispielsweise der Eintritt des Pfades der Ausführung in einen Knoten (`node-enter`) oder die Aktivierung eines Übergangs (`take-transition`).

Grundsätzlich betrachtet sind Actions die vorrangige Methode um zusätzliche technische Details zu einer Prozessdefinition hinzuzufügen, die in der graphischen Repräsentation des Prozesses versteckt bleiben. Diese technischen Details sind zwar für die allgemeine Beschreibung des Geschäftsprozesses durch einen Graphen nur von untergeordneter Bedeutung, erweisen sich jedoch wesentlich für die technische Umsetzung von Prozess-Eigenschaften.

2.2.4 Laufzeit Daten

Token

Das Konzept des Token (Marke) repräsentiert — wie bereits in Abschnitt 2.2.3 angesprochen — den eigentlichen Pfad der Ausführung in der Vorgangsbearbeitung bzw. das Laufzeit Konzept der jBPM Workflow-Engine. Grob betrachtet besteht der Token aus einem Zeiger, der immer auf den gerade aktuellen Knoten verweist und im Verlauf der Vorgangsbearbeitung so dem Pfad der Ausführung folgt. Der Token ist deshalb das Hauptobjekt im Kontext der Ausführung eines Prozesses durch jBPM.

Diesem Konzept liegt der Ansatz der Tokens (auch Marken, Zeichen) in Petri-Netzen[6] zugrunde. Wie in Petri-Netzen auch, wandert der Token während der Ausführung zwischen unterschiedlichen Knoten, die durch Transitionen verbunden sind. Wenn der Token in einen Knoten eintritt, wird dieser ausgeführt.

Der Knoten selbst bzw. die damit verknüpfte Action ist daraufhin dafür verantwortlich, dass der Token den Knoten auch wieder verlässt und somit die Ausführung des Prozesses fortgesetzt wird. Hierfür besitzt der Token eine Schnittstelle, über welche er das Signal zur Weiterführung entgegen nimmt. Wenn der Token auf eine parallele Verzweigung trifft, so wird für jeden Pfad der Verzweigung ein sogenannter Child-Token erstellt. Die folgende Synchronisierung „sammelt“ diese Child-Tokens wieder und fügt sie im Root-Token zusammen.

Durch diesen Ansatz ist es möglich, das Verhalten der Tokens als Grundlage für implizite Informationen zum Status der Vorgangsbearbeitung heranzuziehen. Die unterschiedlichen Zustände der beteiligten Knoten sind immer davon abhängig, ob und wie sie mit einem Token in Verbindung gekommen sind.

Signal

Um die Vorgangsbearbeitung schrittweise weiterzuführen, muss dem Token signalisiert werden das er zur nächsten Phase übergehen soll. Dies geschieht durch eine Methode (`token.signal`) des Tokens, die aufgerufen wird sobald sich der Zustand der Vorgangsbearbeitung entsprechend ändern soll. Hierbei können sowohl einzelne eventuell vorkommende Tokens im speziellen, als auch die Prozessinstanz allgemein angesprochen werden. Im ersten Fall wird der — für den gewählten Token — aktuelle Pfad der Ausführung weitergeführt, zum Beispiel der Zweig einer parallelen Verzweigung. Im zweiten Fall wird das Signal von der Engine an den Haupt-Pfad und somit den Wurzel-Token delegiert, der dann durch den Übergang ebenfalls die nächste Phase des Prozesses startet.

Es ist außerdem möglich, der Signal Methode den Namen der Transition zu übergeben, die ausgewählt werden soll — falls mehrere Transitionen von einem Knoten abzweigen. Durch diese Herangehensweise ist es zum Beispiel möglich bedingte Verzweigungen zu modellieren. Wird kein Name angegeben, fällt die Auswahl automatisch auf die Standard-Transition des aktuellen Knotens.

ProcessInstance

Wie bereits in Abschnitt 2.1.1 beschrieben, repräsentiert eine Prozessinstanz die einzelne Ausführung eines durch die Prozessdefinition beschriebenen Prozesses. Dieses Konzept wird durch die jBPM Workflow-Engine ebenfalls umgesetzt.

6 Grundlagen von und Begriffsdefinitionen zu Petri-Netzen in [14]

Die ProcessInstance beinhaltet alle Daten, die während der Ausführung eines Prozesses benötigt werden. Dazu gehört vor allem der aktuelle Zustand der Vorgangsbearbeitung — also die Anzahl und die Aufenthaltsorte der vorhandenen Tokens. Im Verlaufe der Ausführung beinhaltet die ProcessInstance einen Baum von Tokens, die die vorhandenen Ausführungspfade repräsentieren.

Die übliche Vorgehensweise bei jBPM ist es, Prozessdefinitionen in einer Datenbank zu speichern. Es wird jedoch Caching[7] Mechanismus angewendet, um Ladezeiten zur Laufzeit zu verhindern. Dieses Caching ist allerdings nur deshalb möglich, weil sich im klassischen Fall die Prozessdefinition zur Laufzeit nicht ändert. Dies ist jedoch bei ad-hoc Anpassungen nicht der Fall. Hier soll es möglich sein, die Anpassungen *gerade* zur Laufzeit vorzunehmen.

Da von jBPM aus der ProcessDefinition zur Laufzeit eine ProcessInstance erstellt wird, müssen die angestrebten Anpassungen direkt in der ProcessInstance vorgenommen und gegebenenfalls (bei permanenten Änderungen) zurück in die ProcessDefinition übertragen werden.

2.3 Zusammenfassung

Die meisten Vorgaben der WfMC werden durch die Implementation der jBPM Workflow-Engine direkt umgesetzt. Um die Korrektheit und Konsistenz bei der Adaption eines Workflows zur Laufzeit zu gewährleisten, wird eine explizite Repräsentation der Zustände beteiligter Entitäten benötigt. Die Engine bietet in der vorliegenden Version keine solchen Repräsentationen. Aus diesem Grund muss eine Erweiterung stattfinden. Hierauf wird in Abschnitt 5.1 eingegangen.

Ein wichtiger Unterschied zwischen dem Referenzmodell der WfMC und der Implementierung durch jBPM ist das Konzept der Aktivität. Grundsätzlich werden Aktivitäten von jBPM durch Nodes gekapselt, die allerdings noch weitergehende Aufgaben besitzen. Im Zusammenspiel mit Transitions und Tokens, sowie einer grob an Petri-Netzen orientierten Architektur, sorgen sie für den Aufbau des Graphen der ProcessDefinition. Hierbei kommt die für jBPM entwickelte Implementationstechnik GOP zum Einsatz, welche den Aufbau und die Datenstruktur der ProcessDefinition bestimmt.

Zusammenfassend kann man aus der vorgestellten Struktur der jBPM Engine schließen, dass für die nötige ad-hoc Anpassung zur Laufzeit vor allem das Objekt der ProcessInstance eine große Rolle spielt. Die ProcessInstance kapselt alle nötigen Daten, die entweder angepasst werden sollen oder für die Adaption nötig sind. Der Entwurf von Funktionalitäten, die diese Anpassungen an einer ProcessInstance vornehmen können, wird in Kapitel 5 thematisiert.

7 Cache = Pufferspeicher

3 Anforderungen

In diesem Kapitel werden die Anforderungen analysiert, die an ein Konzept zur Unterstützung von ad-hoc Funktionalitäten durch die jBPM Workflow-Engine gestellt werden. Die Anforderungen lassen sich in Anforderungen an die Workflow-Engine und Anforderungen an den Adaptionsdienst unterteilen. Die Anforderungen an die Engine wurden teilweise bereits durch die Auswahl der jBPM Workflow-Engine in der Aufgabenstellung eingeschränkt. Die Erfüllung der Anforderungen an die Engine ist jedoch Grundlage für eine erfolgreiche Erweiterung um den gewünschten Adaptionsdienst, der selbst auch eigenen Anforderungen unterliegt.

3.1 Anforderungen an die Workflow-Engine

A1.1 Bereitstellung von Schnittstellen Um überhaupt Anpassungen an einer Prozessinstanz zur Laufzeit vornehmen zu können, muss das eingesetzte WfMS sowohl lesenden als auch schreibenden Zugriff auf die Daten der Prozessinstanz zur Laufzeit ermöglichen. Hierfür müssen Schnittstellen bereit gestellt werden, auf die ein Adaptionsdienst zugreifen kann um die Anpassungen vorzunehmen. Da die Auswahl der Workflow-Engine bereits durch die Aufgabenstellung getroffen wurde, müssen die angebotenen Schnittstellen der jBPM Workflow-Engine eingesetzt werden.

A1.2 Repräsentation von Zustandsinformationen Ein korrekter Zugriff auf die Daten der Prozessinstanz kann nur dann erfolgen, wenn Informationen über den aktuellen Status der beteiligten Entitäten vorliegen. Zum Beispiel darf keine Aktivität entfernt werden, die sich zu diesem Zeitpunkt gerade in Ausführung befindet. Auch ist es nicht wünschenswert, Teilbereiche des Workflows zu editieren, die bereits ausgeführt wurden. Dies kann nur durch explizite Zustandsinformationen ermöglicht werden. Die durch die Aufgabenstellung vorgegebene jBPM Workflow-Engine bietet keine solchen Zustandsinformationen an, aus diesem Grund muss eine entsprechende Erweiterung der Engine vorgenommen werden. Hierbei müssen sowohl die möglichen Zustände als auch die Bedingungen für Zustandsübergänge explizit beschrieben und umgesetzt werden.

A1.3 Sicherung der Konsistenz Wenn die Ausführung des Workflows in einen Bereich eintritt, der zu diesem Zeitpunkt gerade vom Adaptionsdienst angepasst wird, können Inkonsistenzen in der Workflow Repräsentation auftreten. Aus diesem Grund muss gewährleistet werden, dass kein gleichzeitiger Zugriff von Workflow Ausführung und Adaptionsdienst auf die betroffenen Komponenten des Workflows stattfinden kann.

3.2 Anforderungen an den Adaptionsdienst

A2.1 Anpassung der Workflow Instanz zur Laufzeit Um ad-hoc Funktionalitäten in einem WfMS zu ermöglichen, müssen strukturelle Anpassungsoperationen bereit gestellt werden.

Die Operationen sollen sich zur Adaption des Kontrollflusses einer laufenden Prozessinstanz einsetzen lassen. Die Form der Anpassung muss vereinfacht stattfinden. Der Benutzer soll eine Anpassung nicht durch die dazu notwendigen atomaren Operationen spezifizieren, sondern es müssen vorgefertigte Anpassungsmuster angeboten werden die eine Adaption in ihrer Gesamtheit durchführen. Aus einer Menge von grundlegenden Mustern können dann komplexeren Operationen zusammengesetzt werden. Die grundsätzlich benötigten Muster sind:

1. Aktivität einfügen
    - Sequentielles Einfügen
    - Paralleles Einfügen
    - Alternatives Einfügen
2. Aktivität entfernen

A2.2 Sicherstellung der Korrektheit Bei der Anpassung von Workflows zur Laufzeit muss die strukturelle Korrektheit des Schemas auch nach der Adaption weiterhin gewährleistet sein. Die Anwendung einer Adaptionsoperation darf also nicht dazu führen, dass als Ergebnis ein nicht-korrektes Workflow Schema entsteht. Außerdem darf durch die Anwendung einer Adaption kein nicht-korrekter Zustand des Workflows entstehen. Dies betrifft vor allem mögliche Deadlocks. Ein Deadlock kann zum Beispiel dann auftreten, wenn die Synchronisierung mehrerer paralleler Pfade nicht vollendet werden kann, weil eine Adaptionsoperation einen dieser Pfade unterbrochen oder gestört hat. Aus diesem Grund muss jede beabsichtigte Adaption dahingehend kontrolliert werden, ob sie die Bedingungen der Korrektheit verletzt. Wenn dies der Fall ist, ist die Adaption abzuweisen. Grundlage für Kontrolle der Bedingungen sind unter anderem die Status-Informationen der beteiligten Entitäten, wie sie in A1.2 gefordert werden.

A2.3 Bereitstellung einer Benutzerschnittstelle Um mit dem Benutzer interagieren zu können, muss eine Benutzerschnittstelle bereit gestellt werden. Durch diese Benutzerschnittstelle wird die Art der Anpassung ausgewählt und der Ort der Anpassung im Schema festgelegt. Daraufhin soll eine Rückmeldung darüber erfolgen, ob die Adaption (bzgl. den Kriterien der Korrektheit) möglich ist. Letztendlich soll erkennbar sein, ob die Änderung erfolgreich war oder der alte Zustand des Workflows wiederhergestellt wurde.

4 Verwandte Arbeiten

Auf dem Gebiet der Workflow Systeme gibt es eine große Anzahl von Forschungsansätzen, die sich durch grundlegend unterschiedliche Herangehensweisen an das Problem der „Flexibilität“ auszeichnen. Schon die unterschiedlichen Bezeichnungen wie dynamische, adaptive, flexible oder ac-hoc Workflows zeigt das große Spektrum der vorhandenen Ansätze. Van der Aalst und Jablonski [2] unterscheiden hierbei die großen Teilbereiche der „Exceptions“, des „ad-hoc Workflow“ und „Dynamic Change“. Letzteres bezieht sich primär auf die Handhabung von „Workflow Evolution“, also der Migration und Versionierung von Prozess Definitionen. Diese kommt im folgenden nur kurz zur Sprache, da sie für die Aufgabenstellung dieser Arbeit nicht relevant ist. Im Anwendungsfeld des Katastrophenmanagements sind Adaptionen am Workflow Schema immer temporär. Die unerwarteten Ereignisse, die zu einer Adaption führen können, sind bei der nächsten Ausführung des Workflows nicht mehr relevant. Es ist daher nicht nötig, die verwendeten Prozess Definitionen zu versionieren. Weitere Konzepte werden im Rahmen des Kapitels jedoch näher auf ihre Tauglichkeit im Anwendungsfeld untersucht und kurz bewertet.

4.1 Grundlegende Ansätze

Die folgenden Arbeiten stellen keine funktionsfähigen dynamischen Workflow Systeme vor, sondern beschreiben grundlegende Herangehensweisen an das Problem der Flexibilität und Adaption.

4.1.1 Herausforderungen der Adaption

Beschreibung

Das Paper von van der Aalst und Jablonski [2] beschreibt sehr umfassend die Herausforderungen und Probleme, die bei der Anpassung von Workflows zur Laufzeit auftreten können. Die Autoren unterscheiden hierbei zwischen *„vorübergehenden“* und *„evolutionären“* Änderungen an der Workflow Struktur. Um die möglichen Fehler strukturieren zu können, wird eine Unterteilung in fünf unterschiedliche Workflow Aspekte (Abb. 4.1) vorgeschlagen. Fehler können daher entweder unter genau einem Aspekt auftreten, oder sich über mehrere Aspekte erstrecken. Um die potentiellen Probleme noch weiter eingrenzen zu können, wird außerdem zwischen *temporären* oder *dauerhaften*, sowie *syntaktischen* und *semantischen* Fehlern unterschieden.

Jeder der definieren Aspekte ist verantwortlich für bestimmte grundlegende Konzepte des Workflow Managements. Im *Prozess-Aspekt* werden z.B. die Prozess Definitionen erstellt und verwaltet, während sich der *Informations-Aspekt* um die beteiligten Workflow Daten (vgl. Abschnitt 2.1.4) kümmert. Der *Ausführungs-Aspekt* beschreibt die elementaren Operationen, die im Prozess-Aspekt verwendet werden um die Daten des Informations-Aspektes zu bearbeiten. Die nötige Verbindung zwischen den Aspekten stellt deshalb der *Integrations-Aspekt*

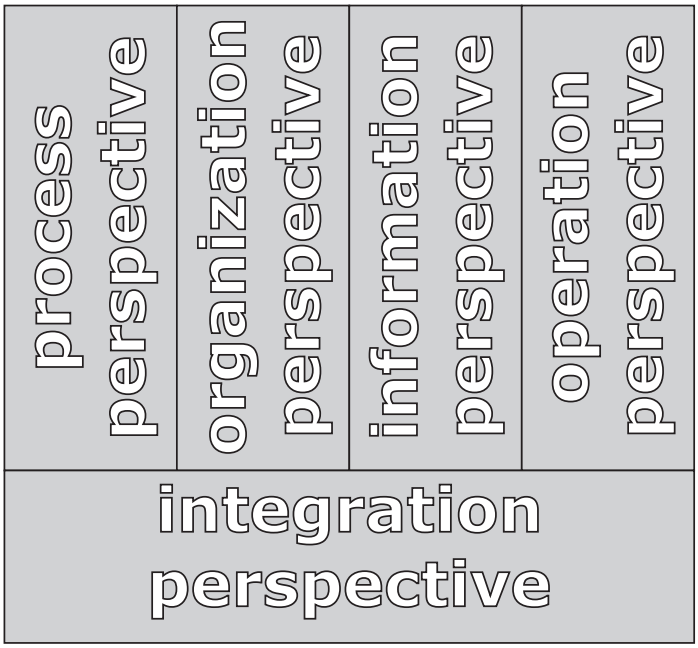

Abbildung 4.1: Fünf Workflow Aspekte nach [2]

her. Die Architektur der meisten vorhandenen WfMS beruht auf der Annahme, dass die Inhalte der fünf Aspekte bereits zur Designzeit bekannt sind und sich kaum ändern. Daraus resultieren die bereits in Kapitel 1 angesprochenen Problematiken, da es unter bestimmten Bedingungen doch sehr häufig zu Änderungen während der Laufzeit kommen kann.

Von den Autoren wird nicht ausführlich die Lösung eines speziellen Problems genannt, sondern eine umfassende Strukturierung der möglichen Änderungen und den daraus resultierenden Problemen bzw. Fehlern vorgeschlagen.

Notwendige Änderungen an einem Workflow klassifizieren van der Aalst und Jablonski nach sechs Kriterien:

1. *Was ist der Grund der Änderung?*
2. *Was ist die Auswirkung der Änderung?*
3. *Welche Aspekte sind betroffen?*
4. *Welche Änderungen sind notwendig?*
5. *Wann sind die Änderungen erlaubt?*
6. *Wie werden existierende Instanzen behandelt?*

Die möglichen Probleme, die auf Grund einer Änderung auftauchen können, können ebenso klassifiziert werden. Hierbei ist die Unterscheidung nach der Art des Fehlers (syntaktisch oder semantisch), der Dauer des Fehlers (temporär oder dauerhaft) und dem Umfang des Fehlers (betrifft einen oder mehrere Aspekte) möglich.

Syntaktische Korrektheit ist hierbei unabhängig vom Kontext des Workflows. Sie betrifft die Struktur der Prozess Definition und das dynamische Ablaufverhalten. Diese Fehler können ohne nähere Kenntnis des Kontextes und der Umgebung entdeckt und vermieden werden. Eingangsparameter einer Aktivität müssen z.B. zwingend vor deren Ausführung

vorhanden sein. Mögliche Deadlocks[1] sind außerdem Beispiele für syntaktische Fehler, die auf jeden Fall vermieden werden müssen, da sie die Terminierbarkeit des Workflows unterbinden.

Semantische Korrektheit betrifft den Kontext der Anwendung. Um diese Fehler vermeiden zu können, ist die genaue Kenntnis der Anwendungsumgebung nötig. Zum Beispiel kann die Errichtung eines Damms erst dann stattfinden, nachdem die Sandsäcke angeliefert wurden.

Ein temporärer Fehler wird von einer Änderung ausgelöst, die nur in der aktuellen Instanz syntaktische und semantische Fehler auslöst. Neue Instanzen sind von der Änderung nicht betroffen.

Da die möglichen Probleme sehr umfangreich sind, wird von den Autoren vorgeschlagen Änderungen so oft es geht zu vermeiden. Falls doch Änderungen auftreten, sollte das System darauf vorbereitet sein. Aus diesem Grund werden die beiden Konzepte „flexiblity by selection“ und „flexiblity by adaption“ vorgestellt. Im ersten Fall werden neue Konstrukte zur Workflow Definitions Sprache hinzugefügt. Zum Beispiel „SEQUENCE (A,B,C)“, welches der Semantik „führe A,B und C sequentiell aber in unbestimmter Reihenfolge aus“ entspricht. Somit könnten bereits einige Änderungen an der Prozess Definition zur Laufzeit vermieden werden. Da aber niemals alle möglichen Änderungen durch solche Konstrukte abgedeckt werden können, kommt das zusätzlich zweite Konzept zum Einsatz. Hierbei geht es primär um dauerhafte Änderungen und einer resultierenden „Varianten und Versionen“ von Workflow Definitionen. Die „flexiblity by adaption“ steht vor dem Problem, auf welche Workflow Instanzen eine Adaption angewendet werden soll. Hierbei unterscheiden auch Heinl u. a. [8] — die die beiden Konzepte vorschlagen — zwischen zukünftigen Instanzen, Instanzen die den zu adaptierenden Bereich noch nicht erreicht haben und einer Menge bekannter (vordefinierter) Instanzen.

Aufbauend wird in [2] das Konzept der *„Workflow Inheritance“* vorgestellt, bei dem neue Versionen einer Prozess Definition von den vorhergehenden Version bestimmte Teile „erben“ können. Dies ist vergleichbar mit dem Konzept der Vererbung in der objektorientierten Programmierung.

Bewertung

Es wird deutlich, dass sich im begrenzten Umfeld des Katastrophenmanagements und dem relevanten Anwendungsfeld die Kriterien der Probleme und Fehler stark eingrenzen lassen. Der Grund für eine Adaption wird hier immer durch eine externe Entwicklung verursacht und hat nur temporäre Auswirkungen. Die Änderungen sind nicht vorhersehbar oder gar planbar, für folgende Instanzen könnten bereits neue externe Bedingungen eine Rolle spielen. Der Kontext eines Workflows kann bei jeder Ausführung ein anderer sein. Deshalb betrifft eine Adaption hier auch immer nur genau eine Instanz, alle anderen existierenden Instanzen müssen gesondert betrachtet werden. Grundsätzlich sind jedoch alle vorhandenen Aspekte und Änderungen zu berücksichtigen und die Adaptionen müssen zu jeder Zeit (also „on-the-fly“) möglich sein. Außerdem ist es nicht möglich auf semantische Korrektheit zu prüfen. Da jedoch angenommen werden kann, dass der Anwender der ad-hoc Funktionalität eingehende Kenntnis über den Kontext der Änderung besitzt, wird diese Verantwortung auf den Benutzer

1 Deadlock: Zustand einer Verklemmung, bei dem Prozesse auf ein Ereignis warten, dass nur von einem Prozess aus dieser Menge ausgelöst werden kann. Die weitere Ausführung der Prozesse ist so nicht mehr möglich.

(z.B. Einsatzleiter) übertragen. Die Syntaktische Korrektheit muss jedoch zu jedem Zeitpunkt gegeben sein und wird deshalb vom System überprüft, wenn eine Adaption stattfinden soll.

Da die auftretenden Änderungen jedoch immer temporärer Natur sind, ist das vorgeschlagene Konzept der Workflow Inheritance für die nötigen ad-hoc Funktionalitäten nicht praktikabel. Außerdem gehen die Autoren nicht näher auf die verwendeten Anpassungsmöglichkeiten oder die Repräsentation des Status in ihrem Modell ein. Die Klassifizierung der Probleme und möglichen Fehler ist jedoch durchaus hilfreich für die Entwicklung der ad-hoc Funktionalitäten. Die Abgrenzung der überhaupt relevanten Adaptionen schränkt die Anforderungen an nötige Funktionen ein und umschreibt bereits grob die entsprechende Fehlerbehandlung.

4.1.2 Flexibilität und Adaption

Beschreibung

In den Arbeiten von Narendra [15][16] wird eine Drei-Schichten-Systemarchitektur für Adaptives Workflow Management vorgestellt. Die vorgestellte Architektur baut stark auf bereits bestehende Forschungsarbeiten im Bereich der (adaptiven) Workflows auf und integriert diese in einem eigenen Workflow Modell. Das *OpenPM* Process Model von Hewlett-Packard[2] dient als Grundlage für die Repräsentation des Workflow Graphen. Um diesen Graphen anzupassen — also zu transformieren — verwendet Narendea die Theorie der „Graph Queries and Graph Transformations“, die im Rahmen des *DYNAMITE* Projekts[3] entwickelt wurde. Außerdem wird zur Überprüfung der Korrektheit und Konsistenz das *ADEPT* Process Model eingesetzt, welches auch in Abschnitt 4.2.4 vorgestellt wird.

Der Ansatz ist, dass mittels OpenPM ein grundlegendes graph-basiertes Workflow Modell erstellt wird und alle dynamischen Änderungen an diesem Modell als Graphen Transformationen betrachtet werden. Diese Transformationen werden durch Mechanismen von DYNAMITE in entsprechende Anfragen (Graph Queries) übersetzt und mittels der Algorithmen des ADEPT Projektes vor der Ausführung auf ihre Korrektheit und Konsistenz überprüft.

Narendra definiert vier grundlegende Design-Ziele [vgl. 15], die ein adaptives WfMS erfüllen sollte:

- *Dynamische Änderungen des Workflow Modells*
  Workflow Instanzen und Workflow Schemata (Prozess Definitionen) sollen während der Ausführung einfach anpassbar sein
- *Modularität des Systems*
  Die einzelnen Komponenten des Systems sollen austauschbar sein
- *Anpassbarkeit*
  Das System soll mit einer hohen Zahl von Nutzern skalieren
- *Geschwindigkeit*
  Benutzer-Interaktionen dürfen keine übermäßigen Wartezeiten verursachen

Um diese Ziele zu erfüllen, wurde eine stark modularisierte drei-Schichten Architektur entwickelt, welche in Abbildung 4.2 zu erkennen ist. Es ist anzumerken, dass hierbei auch auf die

2 HP Workflow Research: Past, Present, and Future
`http://www.hpl.hp.com/techreports/97/HPL-97-105.html`
3 P. Heimann u.a. - DYNAMITE: Dynamic Task Nets for Software Process Management, Proc. 18th ICSE, 331-341

Migration der Workflow Schemata und die Integration in die eigentliche Workflow Planung zur Designzeit Wert gelegt wurde. Relevant für diese Arbeit ist jedoch nur die Instanz-Ebene mit den Komponenten „Wokflow Change Verifier“ und „Workflow Change Model“, da im hier relevanten Szenario des Katastrophenmanagements primär temporäre Änderungen an der Prozessinstanz eine Rolle spielen, die nicht wieder in die Prozess Definition zurückgeschrieben werden müssen.

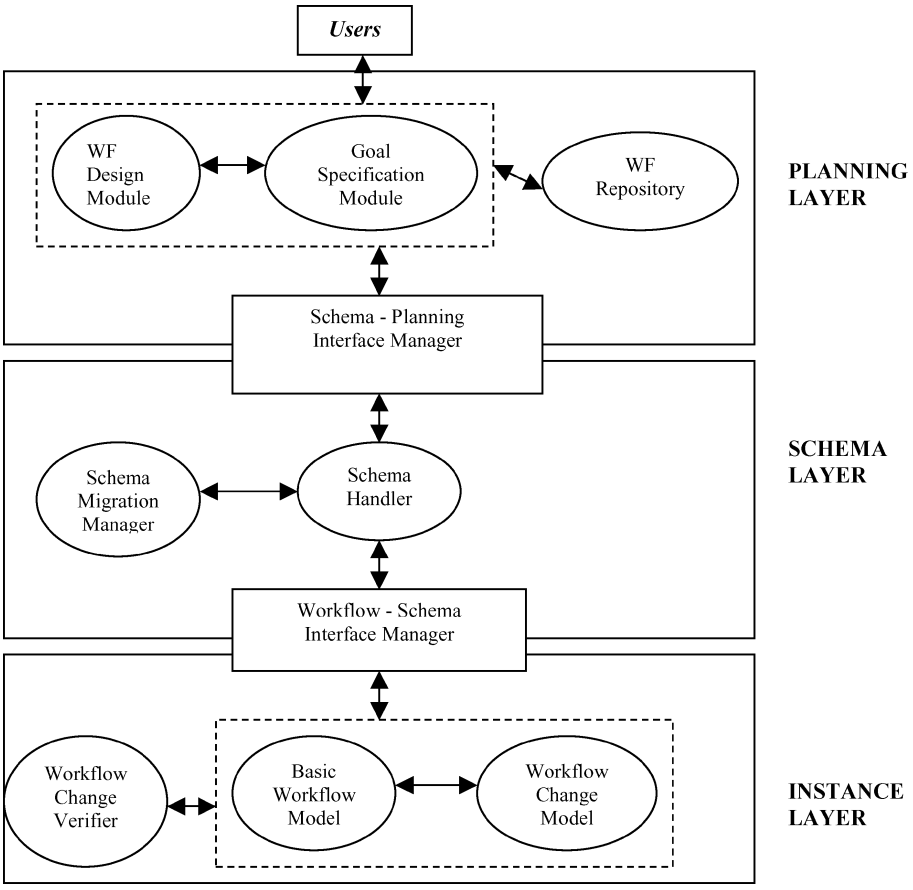

Abbildung 4.2: Drei-Schichten Adaptive Workflow Architektur, Narendra [16]

Um die Adaption des Workflow Modells zu ermöglichen, unterteilt Narendra die vorhandenen Knoten in *„work nodes“* bzw. *„rule nodes“* und führt spezielle *„sync-edges“* ein. Für jeden Knoten werden zudem fünf Statusoptionen und entsprechende Übergangsbedingungen definiert, die sich ebenfalls an den Vorgaben von ADEPT orientieren. Die Konzepte der „sync-edges“ und die Statusoptionen werden in Abschnitt 4.2.4 weiter vertieft.

Der Ablauf einer dynamischen Workflow Änderung läuft bei Narendra in drei Schritten ab:

1. Die gewünschte Änderung wird als Graphen Transformation spezifiziert
2. Die Ausführbarkeit der Transformation wird mittels Graph Anfragen verifiziert und das Ergebnis an den Nutzer gemeldet
3. Falls die Transformation möglich ist, wird sie ausgeführt und der Nutzer darüber informiert

Jede Graphen Transformation folgt hierbei einer speziellen Sequenz von Operationen. Zuerst wird der relevante Sub-Graph selektiert (Start und End-Knoten). Im Anschluss wird der

Aufbau des neuen Sub-Graphen definiert, der den alten Sub-Graphen ersetzen soll. Daraufhin werden die grundlegenden Operationen ermittelt, die nötig sind um den alten Sub-Graphen in den neuen zu überführen. Diese Operationen bestehen primär aus dem Hinzufügen oder Entfernen von Knoten bzw. Transitionen.

Die Anfragen, die entscheiden ob eine gewünschte Änderung möglich ist, werden vom System als sogenannte „graph queries“ ausgeführt. Hierbei wird zuerst die *syntaktische* Korrektheit und Konsistenz der Transformation bestimmt. Hierzu zählt zum Beispiel, dass alle Eingabe-Parameter eines Knotens vor dessen Ausführung vorhanden sein müssen. Außerdem wird die *semantische* Korrektheit und Konsistenz der Transformation überprüft, indem die Ablaufregeln in den „rule-nodes“ überprüft werden.

Nach der eigentlichen Adaption des Workflows kommt die Schema-Ebene zum Einsatz, die sich darum kümmert das neu entstandene Workflow Schema in das bestehende zu übertragen bzw. die Versionierung der vorhandenen Schemata zu verwalten.

Bewertung

Die vorgeschlagene Architektur zieht ihre Vorteile primär aus der Modularisierung des Systems. Durch diesen Ansatz ist es möglich, verbesserte Teilkomponenten schnell zu integrieren. Ein weiterer Vorteil ist die Integration des Schema Migration Managers, der unterschiedliche Möglichkeiten der Migration und Versionierung besitzt. Die Migration ist jedoch für die Zielstellung dieser Arbeit eher unbedeutend. Ein wichtiger Punkt ist die Beschreibung der Zustandsinformationen, die jedoch ebenso wie die Kriterien für Korrektheit und Konsistenz relativ unverändert aus dem Konzept von ADEPT übernommen wurden. Prinzipiell sollte die vorgeschlagene Architektur alle relevanten Anpassungsmöglichkeiten unterstützen. Der Autor geht allerdings nicht genauer auf die Adaptionen und entsprechenden Algorithmen ein, so dass eine präzise Bewertung ohne weiteres nicht möglich ist.

Vor allem die Architektur der Instanz Ebene und die Ablaufbeschreibung der dynamischen Workflow Änderungen sind für den Entwurf der ad-hoc Funktionalitäten hilfreich. Sie werden deshalb als Anhaltspunkte in Kapitel 5 dienen.

4.1.3 Exceptions

Beschreibung

Der Begriff *Exception*[4] beschreibt eine „Abweichung von der normalen — zu erwartenden — Ausführung eines Prozesses“ [18]. Er stammt ursprünglich aus dem Gebiet der Software Architektur. Das Konzept entstand, weil es sehr schwierig ist alle möglichen unvorhersehbaren Ereignisse, die bei der Ausführung eines Programms auftreten können, zu kategorisieren. Um mit den auftretenden Exceptions umgehen zu können, werden „Exception Handler“[5] eingesetzt. Sie enthalten Anweisungen dafür, wie die Auswirkungen der entsprechenden Ereignisse zu handhaben sind.

Da auch bei der Ausführung eines Workflows Abweichungen vom erwarteten Pfad der Ausführung auftreten, ist es naheliegend den Begriff Exception auch in diesem Anwendungsgebiet zu gebrauchen. Russell u. a. [18] veröffentlichten in ihrem Paper erstmals sogenannte „Workflow

4 Ausnahme
5 Ausnahmen-Behandlung

Exception Patterns“, die eine grundsätzliche Kategorisierung der möglichen Exceptions während einer Vorgangsbearbeitung ermöglichen. Es werden zudem die Probleme untersucht, die zu Exceptions während der Ausführung eines Workflows führen können und unterschiedliche Wege zur Behandlung dieser Ausnahmen aufgezeigt. Auf dieser Grundlage wird ein Framework vorgestellt, welches in der Form von unterschiedlichen Patterns[6] zur Fehlerbehandlung vorliegt.

Es wird angenommen, dass eine Exception ein „unterscheidbares, erkennbares Ereignis“ ist, „das zu einem spezifischen Punkt während der Laufzeit eines Workflows auftritt und mit einem einzigartigen Work Item verknüpft ist.“ [18]

Um die Behandlung dieser Ausnahmen zu ermöglichen, müssen drei primäre Eigenschaften betrachtet werden. Wie (1) das betroffene Work Item behandelt werden soll, wie (2) die restlichen Work Items der Instanz behandelt werden sollen und (3) welche Aktionen zur Wiederherstellung ausgeführt werden müssen, um die Auswirkungen der Exception zu behandeln.

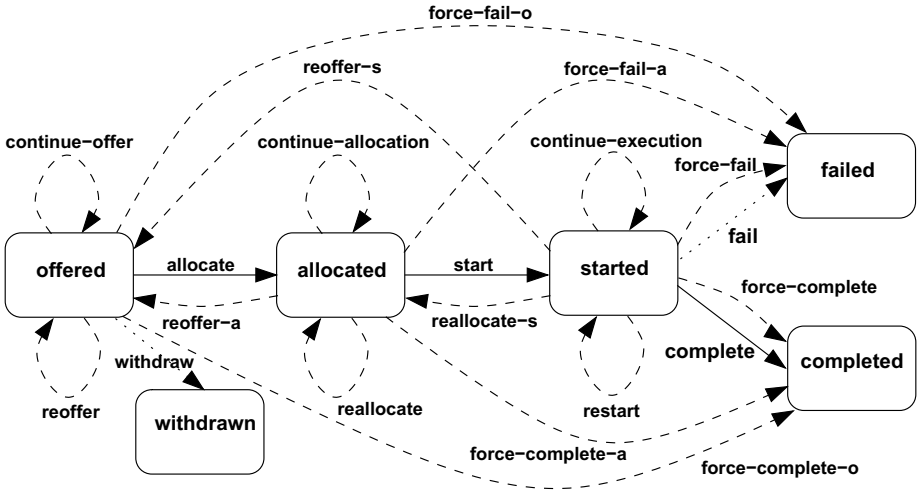

Abbildung 4.3: Status und Exception Handling [18]

Da es nur möglich ist, Handler für *erwartete* Exceptions anzubieten, ist es nötig alle möglichen Ereignisse — die erkannt und behandelt werden können — zu bestimmen. Die Autoren teilen diese Ereignisse in fünf Gruppen ein: Work Item Failure, Deadline Expiry, Resource Unavailability, External Trigger und Constraint Violation. „Work Item Failure“ ist beispielsweise die Unfähigkeit eines Work Items zur weiteren Ausführung. Dies könnte durch einen Benutzer-Abbruch oder aber auch durch Hardware bzw. Software Fehler ausgelöst werden. Die Behandlung der Exceptions muss auf zwei Ebenen stattfinden: Work Item Ebene und Instanz Ebene. In Abbildung 4.3 sind die Fehlerbehandlungen auf Work Item Ebene als gestrichelte Pfeile zu erkennen, während die durchgehenden Pfeile die Zustandsübergänge darstellen. Auf Instanz Ebene wird zum einen entschieden, was mit der aktuellen Instanz bzw. den aktiven Instanzen des Schemas bei Auftritt einer Exception geschieht. Zum anderen wird die entsprechende Aktion zur Wiederherstellung ausgewählt. Daraus ergibt sich eine dreigeteilte Strategie zur Ausnahmen-Behandlung.

6 Patterns = Muster

Bewertung

Die Behandlung von Exceptions ist nur dann möglich, wenn diese bereits im Vorfeld definierbar sind. Nur dann kann ein entsprechender Exception Handler implementiert werden. Dies ist das Hauptproblem dieses Ansatzes, da vor allem im relevanten Anwendungsfeld mit *unerwarteten* Ereignissen zu rechnen ist. Der Einsatz von Exception Handlern für dynamische Workflows ist nur dann möglich, wenn sich die Ereignisse in einem bestimmten Rahmen bewegen, z.B. die üblichen Ausnahmen die während der Vorgangsbearbeitung eines klassischen Workflows im Unternehmensumfeld auftreten können. Hier zeigt der Vorschlag auch die größten Vorteile, da sich diese Ereignisse klar klassifizieren lassen und somit eine sichere Behandlung der Fehler möglich ist. Im Umfeld des Katastrophenmanagements ist dies aber nicht praktikabel und wäre höchstens als *zusätzliche Absicherung* im Fehlerfall denkbar.

Die Operationen die von Russell u. a. zur Behandlung von Exceptions eingesetzt werden, unterscheiden sich zudem stark von den Operationen die für eine „echte“ Anpassung von Workflows zur Laufzeit nötig sind. Zumindest Operationen wie „Aktivität hinzufügen/entfernen“ müssen von einem System mit ad-hoc Funktionalitäten laut Anforderungen bereitgestellt werden. Die in Abbildung 4.3 vorgestellten Operationen sind jedoch typisch für die Behandlung von Exceptions und für ad-hoc Funktionalitäten nicht zu gebrauchen.

Außerdem ist das Konzept mit der Ausrichtung auf Fehler bei der Bearbeitung von Work Items relativ begrenzt einsetzbar, allerdings wäre eine Ausweitung auf Aktivitäten und Sub-Workflows durchaus denkbar. Die Anzahl an möglichen Patterns könnte dann jedoch relativ schnell unüberschaubar werden.

4.1.4 Ad-hoc Workflow

Beschreibung

Als ad-hoc Workflow werden solche Workflows bezeichnet, die vom Benutzer des Systems „ad-hoc“ direkt vor dem Start erstellt werden. Von van der Aalst und Voorhoeve [3] wird diese Art von Workflows zum ersten mal erwähnt und beschrieben. Zum Zeitpunkt der Veröffentlichung wurden Workflow Systeme primär zum Einsatz in der Industrieproduktion entworfen und waren entsprechend starr und unflexibel. Als Gegensatz dazu nennen die Autoren die Groupware Systeme, die wenig strukturierte kooperative Arbeiten unterstützen. Jedoch gerade durch diese Unstrukturiertheit sind Groupware Systeme sehr schwer zu unterstützen und überwachen.

Um Prozesse zu unterstützen, die zwischen diesen beiden Extremen angesiedelt sind, wird das Konzept der „ad-hoc Workflows“ eingeführt. Hierbei sollen sogenannte „template processes“ (Prozessvorlagen) als Grundlage für einen (ad-hoc) definierten detaillierten Workflow dienen. Da hierbei auch gleichzeitige Prozesse ablaufen können und es bei einer Anpassung der Struktur zu den bereits beschriebenen Problemen kommt, schlagen die Autoren die Verwendung der Terminologie aus dem Bereich der Petri-Netze vor. Der Aufbau von Petri-Netzen, sowie deren Korrektheit und Terminierbarkeit wurde bereits umfangreich erforscht und ist auch für Laien relativ einfach verständlich. Wie von van der Aalst [1] erläutert, lassen sich Workflow Prozesse durch eine Klasse von Petri-Netzen beschreiben, die auch als „WF nets“ bezeichnet werden.

Der Vorteil hierbei ist, dass sich bei einer Repräsentation durch Petri-Netze beispielsweise die Erreichbarkeit bestimmter Zustände des Prozesses (z.B. der Endzustand) durch bekannte

Algorithmen überprüfen lässt. Außerdem lassen sich Petri-Netze hierarchisch in sogenannte „subnets“ gliedern, was dem Aufbau von Prozess Definitionen mit Sub Prozessen entspricht. Die subnets sind auch für die vorgeschlagenen Prozessvorlagen von Wichtigkeit, da es das Prinzip der ad-hoc Modellierung ist, bestimmte Bereiche des Netzes durch andere zu ersetzen. Wenn hierbei immer ein subnet getauscht wird, lässt sich die Korrektheit der Operation relativ einfach überprüfen.

Die ständige Anpassung und Neudefinition bei ad-hoc Workflows kann nur dann erfolgreich sein, wenn das zugrunde liegende Prozess Modell die Anpassungen verifizieren kann. Es wird vorgeschlagen, besonderen Wert auf das sogenannte „Safe WorkFlow Net (SWF)“ zu legen. Diese Netze unterliegen strengeren Vorschriften und sind deshalb garantiert terminierbar, beinhalten also keine Deadlocks oder ähnliche Probleme. Sowohl Prozessvorlagen als auch die gewünschten Adaptionen müssen als SWF vorliegen. Somit wird gewährleistet, dass die nötigen Operationen zur Anpassung fehlerfrei sind.

Mögliche Operationen auf den SWF Netzen sind:

- Refinement
  Eine Transition in einem SWF wird durch ein neues SWF subnet ersetzt
- Reduction & Extension
  Das Entfernen bzw. Hinzufügen von Knoten unter bestimmten Voraussetzungen
- Andsplit & Orsplit
  Teilung eines Platzes bzw. einer Transition — Verzweigungen
- Iterate
  Eine neue Transition wird eingefügt, die mit ihrem Ursprung verbunden ist — Schleife

Eine Verknüpfung dieser Operationen soll es ermöglichen, die Templates so zu adaptieren, dass ad-hoc ein gewünschtes Resultat entsteht. In Abbildung 4.4 ist die sequentielle Ausführung unterschiedlicher Operationen zu erkennen, die ein sehr einfachen Process Template in ein komplexeres Workflow Netz umwandeln.

Bewertung

Der große Unterschied zwischen „echten“ ad-hoc Workflows und Workflows, die zur Laufzeit ad-hoc adaptiert werden sollen, ist der Ausgangspunkt der Adaption. Bei ad-hoc Workflows existiert nur eine möglichst einfache Prozessvorlage, die durch Umformung und Erweiterung in ein funktionsfähiges Workflow Schema umgewandelt wird. Nach dieser ad-hoc Erstellung der Prozess Definition wird der Workflow gestartet. Die ad-hoc Adaption von Workflows zur Laufzeit auf der anderen Seite geht davon aus, dass bereits ein voll funktionsfähiges (komplexes) Workflow Schema vorliegt. In diesem Schema sollen bestimmte Teilbereiche ad-hoc angepasst werden, nachdem der eigentliche Workflow schon gestartet wurde.

Der Ansatz der ad-hoc Workflows kommt der gewünschten Funktionalität für das Katastrophenmanagement trotzdem bereits relativ nahe. Die Umsetzung erfolgt bei van der Aalst und Voorhoeve [3] allerdings durch eine Darstellung der Prozess Definition durch native Petri-Netze. Dies erleichtert zwar die Überprüfung der Operationen auf Korrektheit und Konsistenz, schränkt jedoch die Möglichkeiten der Modellierung ein. Nicht jedes beliebige Workflow Schema lässt sich 1:1 durch ein entsprechendes Petri-Netz darstellen. Wäre dies der Fall, könnte man mit dem Ansatz des ad-hoc Workflow so gut wie alle gewünschten Adaptionen zur Laufzeit ausführen und sich bei der Überprüfung der Korrektheit auf bereits bestehende Algorithmen aus dem Bereich der Petri-Netze verlassen.

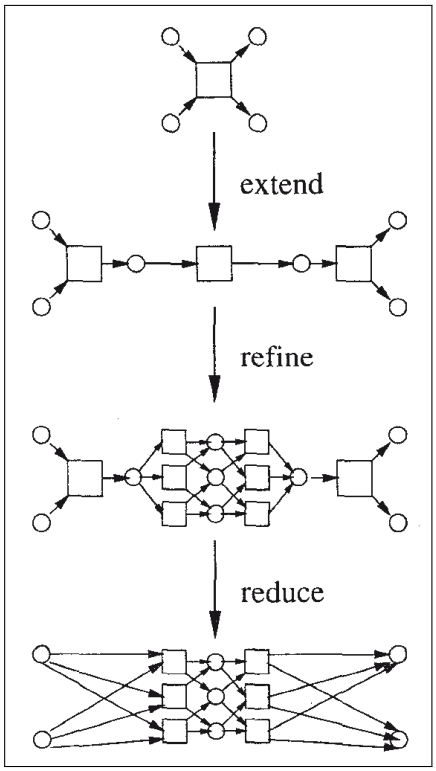

Abbildung 4.4: Ad-hoc Workflow, Adaption von Petri-Netzen [3]

Der Aufbau einer Prozess Definition in jBPM ist wie beschrieben stark an die Struktur von Petri-Netzen angelehnt. Die Nodes, Token und Transitionen sind allesamt ebenfalls Bestandteile in Petri-Netzen. Aus diesem Grund lassen sich einige der vorgeschlagenen Operationen mit leichten Anpassungen auch in jBPM realisieren. Die Operationen sind bereits ein gutes Grundgerüst für die Adaption zur Laufzeit, allerdings müssen sie für den relevanten Anwendungsfeld noch um einige Aspekte erweitert werden.

4.2 Workflow Systeme mit Adaptionsmöglichkeiten

In den letzten Jahren wurden immer häufiger funktionsfähige Workflow Systeme vorgestellt, die unter anderem auch Möglichkeiten zur Anpassung des Workflow Schemas während der Laufzeit bereit stellen. In den folgenden Abschnitten werden einige Systeme präsentiert, die meist aus Forschungsarbeiten heraus entstanden sind und primär auf Flexibilität bzw. Adaptionsmöglichkeiten Wert legen.

4.2.1 Endeavors Workflow System

Beschreibung

Das Endeavors Workflow System wurde im Rahmen von Forschungsarbeiten für die US Air Force entwickelt. In Kammer u. a. [10] wird im Besonderen auf die Techniken eingegangen, die dynamische und adaptive Workflows im System ermöglichen sollen.

Um Änderungen an einem Workflow zu kategorisieren, wird primär das Konzept der Exceptions eingesetzt, allerdings in einer etwas allgemeineren Definition. Zum einen sind dies

Ereignisse, die gar nicht im System modelliert wurden („unexpected exceptions“). Zum anderen jedoch auch Abweichungen vom normalen Ablauf, für die eine Behandlung vorgesehen wurde („expected exceptions“ oder „variations“). Da vor allem die unerwarteten Exceptions interessant sind, wird keine strikte Koppelung von Exception und zugehörigem Handler benötigt.

Um die Ausnahmen weiter zu kategorisieren, wird eine Trennung zwischen den „unterschiedlichen Stufen der Bedeutsamkeit“ eingeführt. Exceptions, die vom System toleriert bzw. ignoriert werden können, werden als „Rauschen“ bezeichnet. „Spezifische“ Exceptions sind nur für die betroffene Prozessinstanz von Bedeutung, während „evolutionäre“ Exceptions eine Änderung der zugrunde liegenden Prozess Definition bewirken müssen. Als die allgemeinen Ziele ihres adaptiven Workflow Systems schildern Kammer u. a. die Erkennung, Vermeidung und Behandlung von Exceptions. Bei der Behandlung wird hierbei zwischen den unterschiedlichen Bedeutungen der Ausnahmen unterschieden. Interessant ist vor allem die Behandlung von *unerwarteten spezifischen* Exceptions, da Ausnahmen dieser Art den gewünschten ad-hoc Anpassungen zur Laufzeit entsprechen.

Der wichtigste Teil der Funktionalitäten für adaptive Workflows ist die dynamische Erstellung und Änderung von Prozess Definitionen. Bei Endeavors kommen hierfür die Konzepte des „late binding of resources“ und die „ just-in-time execution“ zum Einsatz. Ersteres erlaubt die Erledigung von Aktivitäten mit genau den Ressourcen, die zum Zeitpunkt der Ausführung vorhanden sind. Ein Beispiel hierfür ist die Trennung von Daten und Verhalten eines Objektes. Die „ just-in-time execution“ wird auch als „on-the-fly“ Workflow Erstellung bezeichnet und beschreibt in etwa die Eigenschaften der ad-hoc Workflows aus Abschnitt 4.1.4. Die komplette Definition eines Workflows wird erst dann erstellt, wenn sie auch wirklich benötigt wird. Es wird jedoch nur kurz erwähnt, dass zur Umsetzung des dynamischen Verhaltens zusätzliche Funktionen zur Überprüfung der Konsistenz und Korrektheit bzw. zur Fehlerbehebung unabdingbar sind. Eine nähere Beschreibung der eingesetzten Konzepte ist nicht vorhanden.

Als weitere Funktionalität wird unter anderem die „Reflexion“ genannt. Ein reflexiver Prozess kann während der Ausführung auf sich selbst zugreifen und sich so neu modellieren. Der Zugriff wird über Statusinformationen und Rückmeldungen über den Fortschritt des Prozesses gesteuert. Reflexive Prozesse sollten zu jeder Zeit die eigenen Prozessfragmente manipulieren können.

Um diesen Funktionsumfang zu gewährleisten, wurde für das Endeavors Workflow System (vgl. Abbildung 4.5) ein objektorientiertes dynamisches Prozess Modell entwickelt. Die Definition und Spezifikation von *Prozess Artefakten*, *Aktivitäten* und verwendeten *Ressourcen* erfolgt hierbei nach einem durchgehend objektorientierten Schema. Durch dieses Vorgehen ist auch eine weitere Spezifikation in Subkategorien möglich. Zum Beispiel ist die Kategorie *text-editor* eine Spezifizierung von *Ressource*. Das Verhalten der Prozess Objekte wird durch Handler realisiert, die zur Laufzeit geladen und an Objekte gebunden werden können. Durch bereitgestellte Schnittstellen (die „Endeavors Open API“) sind Handler in der Lage, reflexiv auf den Status des Workflows zuzugreifen. So ist es möglich, Komponenten des Prozesses zu analysieren und zu optimieren.

In einer Prozess Definition werden Aktivitäten mittels „control flow“, „data flow“ und „resource flow“ Bindungen verknüpft. Diese Objekte und Bindungen können dynamisch angepasst werden, um auf Änderungen im Workflow oder der Umgebung zu reagieren. Außerdem können bestimmte Benutzer die zugrunde liegende Prozess Definition einer Instanz oder auch spezifische Instanzen direkt anpassen. Eine angepasste Instanz kann wiederum auch als neue Prozess Definition weiterverwendet werden.

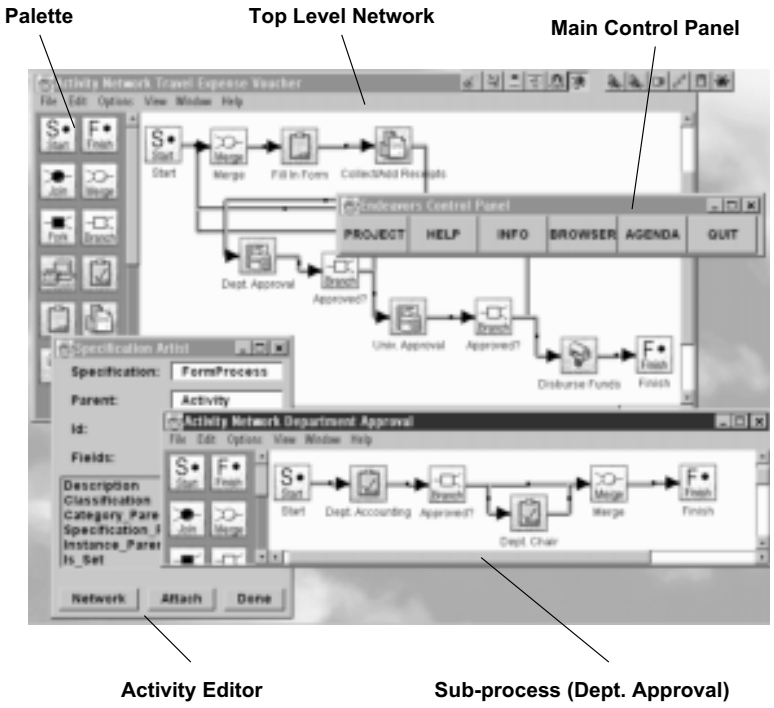

Abbildung 4.5: Endeavors Workflow System [10]

Bewertung

Endeavors beinhaltet noch einige zusätzliche Funktionen, die zur Unterstützung von dynamischen und adaptiven Workflows geeignet sind. Die wichtigsten Eigenschaften wurden allerdings in diesem Abschnitt vorgestellt. Hierbei ist vor allem die dynamische Anbindung von Ressourcen zur Laufzeit, die Reflexion des Objektmodells und die direkte Adaption durch Benutzer hervorzuheben.

Durch diese Funktionalitäten werden umfangreiche Adaptionen eines Workflows zur Laufzeit ermöglicht. Die Autoren haben erfolgreich ein System entwickelt und implementiert, welches die geforderten Ziele für adaptive Workflows erfüllt. Einige Teilbereiche waren zum Zeitpunkt der Veröffentlichung noch als „Prototyp“ oder „Future Work“ gekennzeichnet. Allerdings war die Implementierung der genannten Komponenten bereits abgeschlossen.

Was jedoch nicht weiter zur Sprache kommt, ist die Überprüfung gewünschter Änderungen auf syntaktische oder semantische Korrektheit. Es findet sich leider kein Hinweis darauf, wie bzw. ob dies in Endeavors implementiert wurde. Durch dieses Manko ist ist es nicht ohne weiteres möglich, die gewünschten Anforderungen an ad-hoc Funktionalitäten zur Laufzeit komplett in Endeavors zu überprüfen.

Es finden sich jedoch einige interessante Konzepte und Anregungen im entwickelten System. Vor allem die späte Bindung von Ressourcen dürfte einige Probleme der Laufzeit-Adaption vereinfachen. Allerdings ist es nicht ohne weiteres möglich, dieses Konzept auch auf anderen Workflow-Engines (wie z.B. jBPM) anzuwenden. Eine Engine muss von Grund auf eine Architektur aufweisen, die die späte Bindung von Ressourcen oder auch Reflexion ermöglicht. Dies als zusätzliche Funktionalität hinzuzufügen ist nicht möglich, wenn nicht ein gewisser Grundstein bereits vorhanden ist.

4.2.2 PoliFlow

Beschreibung

Das Verbundprojekt PoliFlow beschäftigte sich mit dem Einsatz von Workflow Systemen in der öffentlichen Verwaltung. In Siebert [19] wird die daraus hervorgegangene Konzeption eines Adaptiven Workflow Systems (AWS) vorgestellt. Dieses System soll vor allem in der Lage sein „unstrukturierte Vorgänge“ zu unterstützen, bei denen noch nicht alle zur Ausführung erforderlichen Details während der Geschäftsprozessmodellierung bekannt sind. Im Laufe des Projektes wurde das SWATS[7] Workflow System entwickelt, welches die vorgestellten Konzeptionen umsetzen soll.

Das so entstandene Referenzmodell für AWS ist in Abbildung 4.6 zu erkennen. Das Konzept basiert zum einen auf der Vermeidung oder Automatisierung von Adaptionen — zum anderen sollen allerdings auch manuelle Anpassungen von Workflow Instanzen zur Laufzeit ermöglicht werden, die durch verschiedene Kontrollmechanismen eingeschränkt werden können. Die Kontrolle der syntaktischen und semantischen Korrektheit wird hierbei ergänzt durch die Definition von Anpassungsrechten. Da in der Anwendungsumgebung beliebige Benutzer den Workflow anpassen können, sollten manuelle Adaptionen über einen graphischen Workflow-Editor durchgeführt werden.

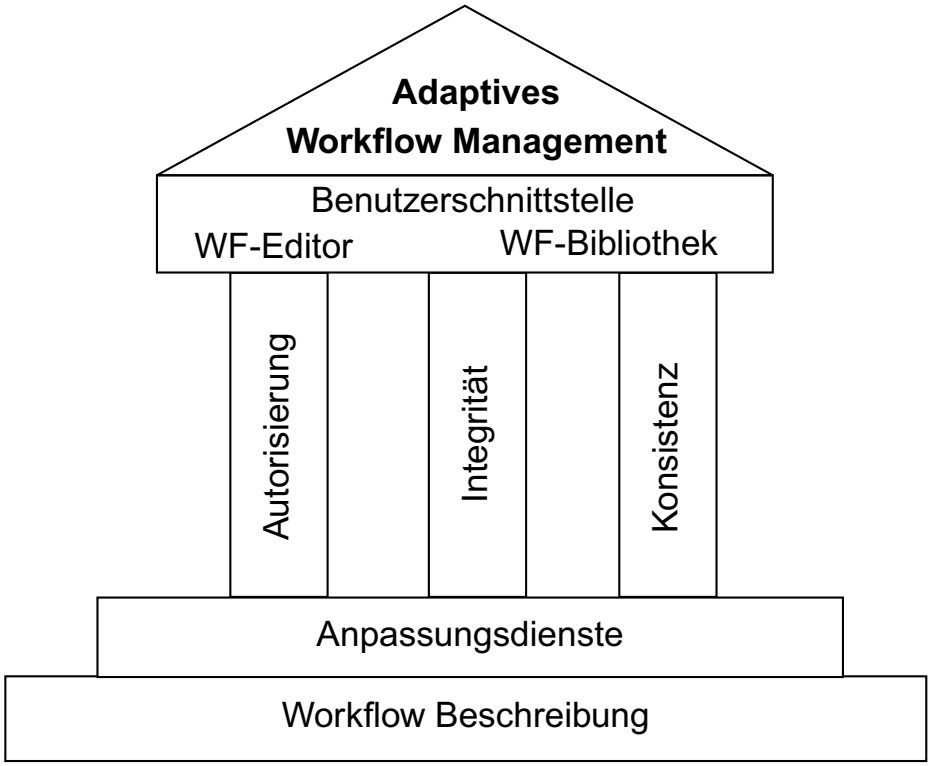

Abbildung 4.6: Adaptives Workflow Management bei PoliFlow [19]

Relevant für diese Arbeit sind die Funktionalitäten der manuellen Workflow Anpassung. Hierbei wird eine Menge von vordefinierten Änderungen durch einen „Anpassungsdienst“ zur Verfügung gestellt. Dieser kann einerseits für die Adaption bei unvorhergesehenen Ereignissen eingesetzt werden. Andererseits ist es auch möglich den Dienst bewusst einzusetzen. Zum

7 SWATS: Stuttgarter Workflow- und Telekooperationssystem

Beispiel wird das Schema eines komplexen Workflows nur bis zu einer bestimmten Stelle explizit definiert, um später während der Ausführung auf spezifische Eigenheiten mit gezielten Anpassungen zu reagieren.

Siebert stellt auch fest, dass die Auswirkungen einer Anpassung sehr schnell komplex und für den normalen Benutzer unüberschaubar werden können. Aus diesem Grund werden zusätzliche Kontrollmechanismen eingeführt, die vermeiden sollen dass: [vgl. 19]

- Benutzer unerwünschte/unerlaubte Änderungen durchführen (Anpassungsrechte)
- Die syntaktische Korrektheit durch manuellen Anpassungen gefährdet wird
- Die semantische Korrektheit durch unerwünschte Auswirkungen gefährdet wird

Die Anpassungskontrolle beinhaltet also zum einen die Autorisation des Benutzers und zum anderen die Überprüfung der Korrektheit. Da die Konsistenz- und Integritätsprüfung zur Laufzeit nicht trivial ist, werden Mechanismen bereitgestellt um diese effizient zu gestalten. Hierfür müssen alle Auswirkungen einer Anpassung berechenbar sein und die Bedingungen der betroffenen Objekte müssen wieder ausgewertet werden. Um entscheiden zu können, welche Anwender welche Anpassungsrechte besitzt, wurde ein neuer Aspekt des Workflow Modells definiert. Hierbei ist es von Bedeutung „welche Person unter welchen Umständen welche Anpassungen durchführen dürfen“. Außerdem werden bestimmten Maßnahmen an die Anpassung gebunden, wie z.B. die Benachrichtigung betroffener Benutzer. Ein Anpassungsrecht wird deshalb bei Siebert als „Quadrupel(Anpassung, Aktor, Zustand, Maßnahme)“ definiert.

Bewertung

Wichtig ist bei PoliFlow zum einen die Konzeption der Architektur für adaptive Workflows in allgemeinen. Hierbei wird bereits berücksichtigt, dass eine umfassende Kontrolle der gewünschten Anpassungen unabdingbar ist. Zum anderen ist es eben diese definierte Anpassungskontrolle, die auch für die Konzeption der ad-hoc Funktionalitäten mit jBPM zu beachten ist. Die Kontrolle der Autorisation und Korrektheit sind von grundlegender Bedeutung bei jeglicher Adaption.

Als negativ ist jedoch zu beurteilen, dass wieder nicht näher auf die Mechanismen zur Überprüfung der Korrektheit und Konsistenz eingegangen wurde. Es wird zwar allgemein gefordert, Auswirkungen einer Änderung berechnen zu können — wie dies umgesetzt werden kann ist allerdings nicht weiter ausgeführt. Genauer wurde beschrieben, wie die Autorisation der Benutzer erreicht werden kann. Allerdings ist es im Umfeld des Katastrophenmanagements etwas einfacher, da nur der Einsatzleiter eine Anpassung am Workflow vornehmen darf und dieser Person eine gewisse Kompetenz im Zusammenhang mit dem betroffenen Prozess unterstellt werden kann.

4.2.3 AgentWork

Beschreibung

Das WfMS AgentWork [13] wurde entwickelt um *automatisierte* Workflow Adaptionen zu ermöglichen. Hierfür wurde ein Ansatz gewählt, der auf der Definition von speziellen Regeln beruht. Diese Regeln spezifizieren mögliche Exceptions und folglich nötige Workflow Adaptionen. Im besonderen wird es durch die Betrachtung temporärer Zusammenhänge während der Vorgangsbearbeitung ermöglicht, „voraussagend“ auf denkbare Fehler einzugehen.

Müller u. a. [13] unterscheiden bei der automatisierten Adaption von Workflows zwischen zwei Herangehensweisen: reagierende und voraussagende Adaption. Wenn Bereiche eines Workflows von einem „logischen Fehler“ betroffen sind, passt die voraussagende Adaption den Bereich bereits im Voraus an, sobald der Fehler erkennbar ist. Die reagierende Adaption muss dann zum Einsatz kommen, wenn eine voraussagende nicht möglich war und die Ausführung der betroffeneren Bereiche unmittelbar bevor steht.

Um dies zu ermöglichen, wird ein sogenanntes Event/Condition/Action (ECA) regel-basiertes Modell eingesetzt. Diese Regeln sollen automatisch logische Fehler entdecken und nötige Adaptionen ausführen. Hierbei ist immer ein entsprechendes Zeitintervall von Bedeutung, da nur durch temporäre Logik voraussagende Adaptionen denkbar sind. Außerdem wird durch „workflow estimation algorithms“ festgestellt, welcher Teil eines Workflow betroffen ist und angepasst werden muss. Für die eigentliche Anpassung werden umfangreiche Operatoren bereit gestellt, die sowohl den Kontrollfluss (incl. der Struktur des Workflows) als auch — wenn nötig — den Datenfluss anpassen können. Ein Mechanismus zur Überwachung der Adaptionen ist ebenfalls vorhanden.

In Abbildung 4.7 ist die drei-Schichten Architektur des Workflow Systems zu erkennen. Die Adaptions-Schicht besteht hierbei aus drei Komponenten, die als „Agenten“ bezeichnet werden. Der „Event Monitoring Agent“ entscheidet, welche Ereignisse relevante Fehler nach sich ziehen können. Der „Adaptation Agent“ nimmt die eigentliche Adaption am Workflow vor, während der „Workflow Monitoring Agent“ überprüft ob die (zeitlichen) Annahmen des Adaptation Agents der Wirklichkeit entsprechen.

Das Workflow Modell von AgentWork dient zum einen als Grundlage für die Prozess Definition, zum anderen basieren auch die ECA Regeln auf diesem gemeinsamen Schema. Der Aufbau ist vergleichbar mit anderen Systemen. Als Besonderheit werden jedoch auch hier wieder „synchronization edges“ angebracht. Da das ADEPTflex System [17] in der Workflow Ebene von AgentWork zum Einsatz kommt, wurden entsprechende Funktionalitäten übernommen. Auch die von ADEPTflex bekannten „data flow edges“ kommen zum Einsatz, um die Korrektheit des Datenflusses gewährleisten zu können. Zur Beschreibung der synchronization und data flow edges siehe Abschnitt 4.2.4.

Um die eigentliche Adaption durchführen zu können, werden strukturelle „Control Flow Operatoren“ angeboten. Diese Operatoren verändern die Struktur des spezifischen Workflows und werden durch in den Regeln definierte (allgemein gültige) „Control Actions“ aufgerufen. Beispiele für solche Operatoren sind „drop-node“, „add-node“ und „add-node-loop“. Weitere Operatoren lassen sich durch Kombinationen erstellen. Die Operatoren besitzen bestimmte Entscheidungskriterien, wie in dem speziellen Fall eine Adaption durchgeführt werden kann. Zum Beispiel lässt sich „drop-node“ dahingehend unterscheiden, dass falls die gewünschte Aktivität in einer Sequenz steht, sie einfach entfernt werden kann. Wenn sie jedoch Teil einer parallelen Verzweigung und nur mit einem zusätzlichen Zweig vorhanden ist, so kann der komplette Block der parallelen Verzweigung entfernt und durch eine Sequenz ersetzt werden.

Bewertung

Der große Unterschied zwischen AgentWork und anderen adaptiven Workflow Systemen, ist die Möglichkeit der *automatisierten* Adaption. Durch die ECA Regeln können Anpassungen am Workflow vorgenommen und mögliche Fehlerquellen *voraussagend* gefunden werden. Hierfür ist allerdings umfassendes Wissen über den zeitlichen Kontext des Workflows eine grundlegende Voraussetzung. Dies ist im Anwendugsfall des Katastrophenmanagements nur

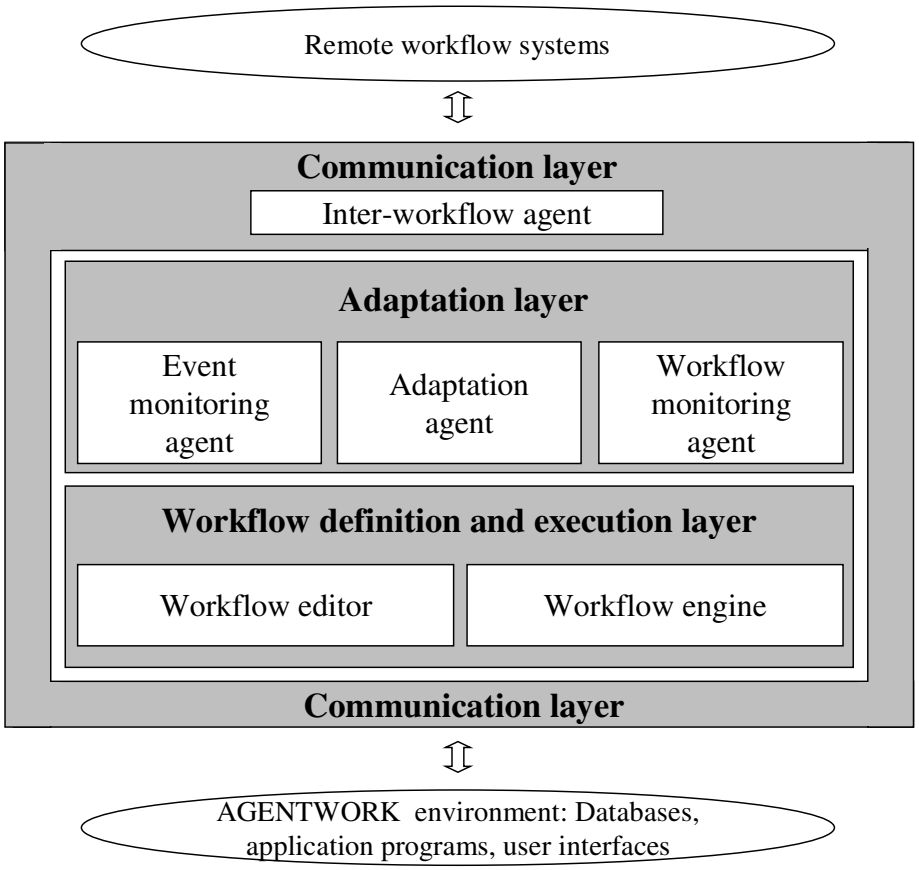

Abbildung 4.7: AgentWork Architektur [13]

schwierig zu erreichen, da die Dauer einer bestimmten Aktivität je nach Aufwand zeitlich sehr variabel sein kann.

Die angebotenen Operatoren zur Workflow Adaption sind allerdings sehr umfassend und können alle gewünschten Anpassungen vornehmen. Durch das zugrunde liegende ADEPTflex System können die Anpassungen wie „drop-node“ an Instanzen zur Laufzeit vorgenommen und die Korrektheit bzw. Konsistenz überprüft werden. Die zusätzliche Funktionalität der Automatisierung ist für die angesprochenen Anwendungsfelder (Krebspatientenversorgung) durchaus hilfreich, jedoch nicht auf beliebige Gebiete übertragbar.

4.2.4 ADEPTflex

Beschreibung

Die Arbeit von Reichert und Dadam [17] wurde bereits weiter oben erwähnt, da sie als grundlegend im Bereich der adaptiven Workflow Systeme angesehen werden kann. Die Autoren haben es sich zur Aufgabe gemacht, die Flexibilität von Workflow Systemen zu erhöhen. Die entwickelte Workflow-Engine ADEPT, mit der für Adaption zuständigen Komponente ADEPTflex, ermöglicht daher strukturelle Veränderungen einer Workflow Definition zur Laufzeit. Hierfür wurde eine vollständige Menge von Anpassungsoperationen definiert, die vor allem die strukturelle Korrektheit und Konsistenz des Workflows berücksichtigen.

Bei ADEPT besteht ein Workflow Graph aus Knoten und Kanten, wobei die Nodes Aktivitäten und die Kanten Verhältnisse der Aktivitäten und beinhalteten Daten zueinander repräsentieren. So entsteht ein gerichteter strukturierter Graph mit je genau einem Start- und Endknoten. Prinzipiell ist der Aufbau also vergleichbar mit dem Aufbau einer Prozess Definition, wie sie in Abschnitt 2.2 dargestellt wurde. Als strukturelle Bausteine gibt es hierbei z.B. „sequential execution“, „parallel processing“ und „conditional routing“. Außerdem werden Schleifen als symmetrische Blöcke mit einzigartigem Start- und Endknoten definiert.

Zusätzlich werden jedoch unterschiedliche Konzepte vorgestellt, die den Workflow Graphen erweitern und ihn um für die Adaption nötige Funktionalitäten bereichern. Eine „failure edge“ verbindet einen Knoten n_f mit einem vorhergehenden Knoten n_s. Falls zur Laufzeit die Aktivität in n_f fehlschlägt, kann so ein „Rollback“ der Zustände aller Knoten bis n_s erfolgen. Um die Synchronisierung von Aktivitäten in unterschiedlichen Zweigen einer parallelen Verzweigung zu ermöglichen, werden die sogenannten „synchronization edges“ eingeführt. Eine „soft-sync-edge“ zwischen zwei Knoten n_1 und n_2 bewirkt z.B. eine Verzögerung von n_2. Die Aktivität kann nur dann ausgeführt werden, falls n_1 entweder bereits beendet wurde oder gar nicht mehr gestartet werden kann. Des Weiteren wird ein „data flow schema“ des Workflows erstellt. In diesem Schema sind die Abhängigkeiten von Daten der Aktivitäten zueinander festgelegt. Ausgabeparameter eines Knotens können z.B. Eingabeparameter eines anderen Knotens darstellen — dieses Verhältnis muss bekannt sein, wenn man eine Adaption vornehmen möchte. Durch die Überprüfung von Eigenschaften dieses Schemas ist es möglich, die Korrektheit des Datenflusses zu überprüfen.

Ein weiterer ausschlaggebender Punkt für die strukturellen Anpassung eines Workflows ist der Status der vorhandenen Aktivitäten. Reichert und Dadam definieren hierbei folgende Statusoptionen für die Knoten: NOT_ACTIVATED, ACTIVATED, RUNNING, COMPLETED, FAILED, SKIPPED. Zudem erhalten die Kanten des Graphen die Statusoptionen: NOT_SIGNALED, FALSE_SIGNALED, TRUE_SIGNALED. Der Status aller Knoten und Kanten sowie die beinhalteten Datenwerte bilden gemeinsam den Status der Prozessinstanz. Die Autoren definieren zusätzlich die Bedingungen die erfüllt sein müssen, damit ein bestimmter Status eingenommen werden kann.

Bei den Kriterien für die Korrektheit und Konsistenz wird zwischen *Kontrollfluss* und *Datenfluss* unterschieden. Um die Korrektheit des Kontrollflusses zu gewährleisten, muss jeder Knoten der Definition vom Start aus erreichbar sein und von jedem Knoten aus der Endknoten erreichbar sein. Für nicht-zyklische Workflow Graphen ist dieses Kriterium grundsätzlich erfüllt, allerdings können Probleme bei Schleifen auftreten.

Die Überprüfung der Korrektheit des Datenflusses stellt eine größere Herausforderung dar. Zum Beispiel muss gewährleistet sein, dass alle Eingabeparameter eines Knotens vorhanden sind, bevor die enthaltene Aktion ausgeführt werden kann. Dies wird bei ADEPT durch das data flow Schema bestimmt. Zudem dürfen z.B. Aktivitäten in unterschiedlichen Zweigen einer parallelen Verzweigung nur dann Schreibzugriff auf dasselbe Datenelement erlangen, wenn sie durch eine sync-edge synchronisiert sind. Das Hinzufügen und Entfernen von Knoten kann alle diese Bedingungen verletzten.

Das Hauptaugenmerk beim Entwurf der Operationen zur dynamischen strukturellen Änderung eines Workflows mit ADEPTflex liegt daher auf der Überprüfung der Korrektheit und Konsistenz. Das Resultat einer Änderung muss immer ein syntaktisch korrektes Workflow Schema in einem „erlaubten“ Zustand sein. Hierfür werden Operationen zum Hinzufügen und Entfernen von Aktivitäten oder ganzen Teilbereichen definiert. Weiterhin soll die Parallelisierung bzw. Serialisierung von Abschnitten ermöglicht oder einfach nur eine Aktivität übersprungen werden.

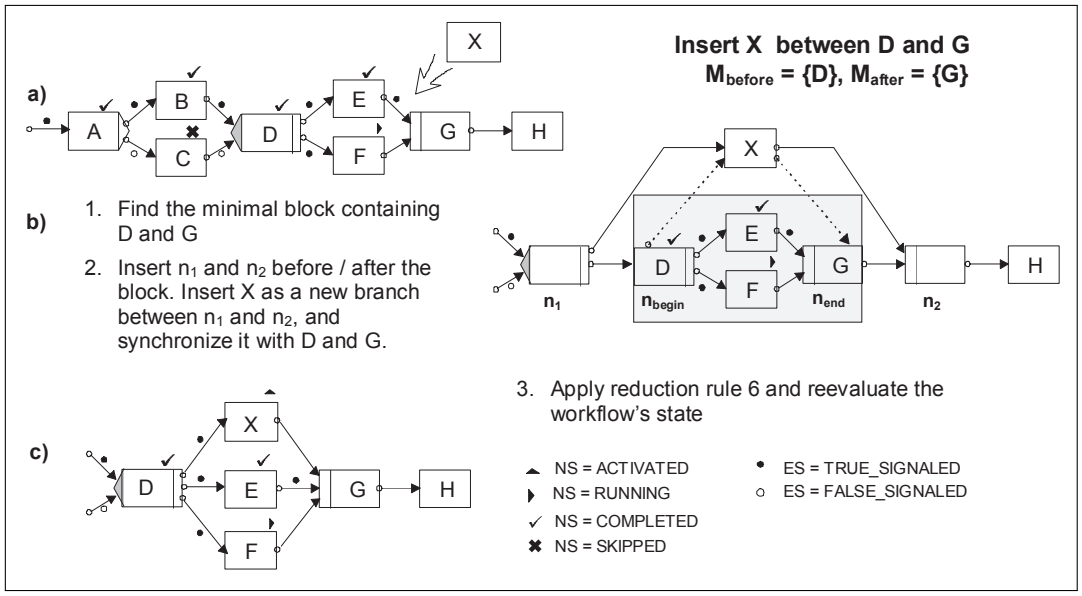

Abbildung 4.8: Aktivität einfügen mit ADEPTflex [17]

Als Beispiel wird die „insert operation“ näher erläutert. Hierbei wird in mehreren Schritten eine Graphen-Substitution ausgeführt, um die gewünschte Änderung zu erreichen. Der Ablauf der Operation ist in Abbildung 4.8 zu erkennen. Es ist zu sehen, dass durch die Allgemeingültigkeit der Operation bestimmte Umwege in Kauf genommen werden müssen. So muss im Anschluss an das eigentliche Einfügen des Knotens der Graph noch „reduziert“ werden, da im Laufe der Operation einige Hilfskonstruke („NULL Nodes“) hinzugefügt wurden.

Bewertung

ADEPTflex beschreibt ein umfassendes formales Fundament für die Unterstützung von Adaptionen an Workflow Instanzen zur Laufzeit. In diesem Zusammenhang wurde eine eigene Workflow-Engine entwickelt, die die geforderten Zusätze bei der strukturellen Beschreibung umsetzt. Durch diese Zusätze wie „sync-edges“ und das „data flow schema“ werden die nötigen Korrektheits- und Konsistenzkriterien erst überprüfbar.

Das ist auch der Nachteil des Ansatzes, da keine bestehende Workflow-Engine einfach um diese geforderten Eigenschaften erweitert werden kann. Die Architektur einer grundsätzlich neuen Engine, wie sie für ADEPT entwickelt wurde, beinhaltet natürlich die gewünschten strukturellen Zusätze. Das fertige System ist aus diesem Grund sehr vielversprechend, weil alle denkbaren Adaptionsmöglichkeiten unterstützt werden. Minimale Einschränkungen auf Grund der Korrektheits- und Konsistenzsicherung können auch in anderen Systemen nicht verhindert werden.

Vor allem die Repräsentation des Status bei ADEPT wird für die Entwicklung der ad-hoc Funktionalitäten als Grundlage dienen, soweit möglich kommen auch Regeln zur Überprüfung der Korrektheit zum Einsatz. Dies ist allerdings nur dann möglich, wenn keine sync-edges und data flow Informationen benötigt werden, da diese in jBPM nicht vorhanden sind.

4.3 Bewertung

Tabelle 4.1 gibt einen kurzen Überblick über die betrachteten Ansätze und Systeme. Die Bewertung erfolgt nach den Gesichtspunkten der in Kapitel 3 definierten Anforderungen. Ein „+“ bedeutet, die Anforderung wird erfüllt, während ein „−“ die Nichterfüllung repräsentiert. Da teilweise die Kriterien nicht nachprüfbar waren, zeigt das Symbol „*o*“ eine neutrale Bewertung. Da die Architektur von Narendra und das System Agent-Work die Workflow-Engine von ADEPTflex übernommen haben, konnte die Repräsentation des Status und die Überprüfung der Korrektheit nicht voll angerechnet werden.

	A1.2 Status	**A2.1 Adaption**	**A2.2 Korrektheit**	**A2.3 UI**
WF Inheritance	−	*o*	+	+
3-T Narendra	(+)	+	(+)	+
Exceptions	+	−	+	−
ad-hoc WF	−	+	+	+
Edeavors	*o*	+	−	+
Poli-Flow	*o*	+	*o*	*o*
Agent-Work	(+)	+	(+)	+
ADEPTflex	+	+	+	+

Tabelle 4.1: Erfüllung der Anforderungen durch Related Work

5 Entwurf

In diesem Kapitel werden die entworfenen Konzepte vorgestellt, die zur Umsetzung der ad-hoc Funktionalitäten benötigt werden.

Die Aufgaben, die sich bei der Entwicklung von ad-hoc Funktionalitäten für die jBPM Workflow-Engine stellen, lassen sich grob in zwei Teilbereiche untergliedern. Zum einen muss das bestehende System so erweitert werden, dass der Status aller Objekte — die an der Durchführung einer Adaption beteiligt sind — verwaltet werden kann. In A1.2 — Repräsentation von Zustandsinformationen — wurde diese Anforderung beschrieben. Darauf aufbauend werden die eigentlichen Anpassungsoperationen (Anforderung A2.1 — Anpassung der Workflow Instanz zur Laufzeit) entwickelt, für die zuvor jedoch die Kriterien der Korrektheit definiert werden müssen. Zum Schluss des Kapitels wird aufgezeigt, wie die Korrektheit der Anpassungsoperationen sichergestellt werden kann. Diese Anforderung wurde in A2.2 — Sicherstellung der Korrektheit — definiert.

5.1 Status: Repräsentation und Bedingungen

5.1.1 Definition der Zustände

Die jBPM Workflow-Engine wurde im klassischen Sinn implementiert, es wird also eine strikte Trennung von Design- und Laufzeit angenommen. Aus diesem Grund sind explizite Statusinformationen von Aktivitäten und Transitionen kein Teil des Systems. Diese sind erst dann relevant, wenn zur Laufzeit eine Anpassung der Workflowstruktur vorgenommen werden soll. Es darf z.B. kein Zugriff auf eine gerade laufende Aktivität stattfinden. Natürlich ist es über Umwege möglich, den aktuellen Status einer Node in jBPM zumindest teilweise zu bestimmen. Sobald der Token in eine Node eintritt, wird diese ausgeführt. Deshalb ist jede Node, die gerade einen Token beinhaltet, im Status „aktiv“. Es lässt sich allerdings nicht nachprüfen, ob eine Node bereits ausgeführt wurde oder ob sie noch ausgeführt werden wird. Die Engine beschreibt zwar implizit die beiden Zustände „aktiv/inaktiv“, also „Token in Node/Token nicht in Node“, diese Zustände sind allerdings für die geforderten Funktionalitäten nicht ausreichend. Außerdem wird in Anforderung A1.2 eine explizite Zustandsbeschreibung verlangt.

Folgendes Beispiel verdeutlicht die Relevanz weiterer Zustände: Eine Aktivität, die bereits ausgeführt wurde (sich also im Endzustand „completed“ befindet) darf nicht mehr Bestandteil einer Adaption sein. Diese Node kann mit hoher Wahrscheinlichkeit Daten enthalten, die von späteren Aktivitäten benötigt werden. Entfernt man also diese Node, wird die Integrität des laufenden Workflows gefährdet.

Neben „completed“ sind noch zwei weitere Endzustände möglich. Zum einen beschreibt der Zustand „failed“, dass bei der Bearbeitung eines Knotens ein Fehler aufgetreten ist. Zum anderen sollte das Überspringen einer Aktivität den Zustand „skipped“ hervorrufen.

Direkt bei der Erstellung eines Node-Objekts geht dieses in den Status „not_activated“ über und verweilt in diesem Zustand so lange keine Ereignisse auftreten, die den Zustand der enthaltenen Aktivität beeinflussen.

Im Anwendungsfeld des Katastrophenmanagements, insbesondere im relevanten Projekt SoKNOS, wird eine zusätzliche Statusoption für Aktivitäten benötigt. Dieser zusätzliche Status „activated“ wird ins Zustandsmodell eingeführt. Dieser Status wird nicht durch die Engine gesetzt, sondern er beschreibt einen externen Vorgang. Steht die Ausführung einer Aktivität unmittelbar bevor (z.B. die vorhergehende Aktivität einer Sequenz wurde gerade beendet), so wird durch einen zuständigen Einsatzleiter der Status der Aktivität auf „activated“ gesetzt. Daraufhin entscheidet z.B. der zuständige Stabsschef über die weiteren Schritte. Sobald das „OK“ eintrifft, kann die Aktivität ausgeführt werden. Durch diese Vorgehensweise können zum Beispiel auch bestimmte Vorkehrungen — die zur Ausführung der Aktivität nötig sind — bereits im Vorfeld getroffen werden, damit sich die eigentliche Ausführung in diesem zeitkritischen Umfeld nicht unnötig verzögert. Jedoch muss diese Entscheidung immer durch eine externe Partei getroffen werden. Eine interne Bedingungsbeschreibung darüber, wann ein Knoten auf „activated“ gesetzt werden muss, kann nicht im System definiert werden. Allerdings muss der Zustand trotzdem vorhanden sein, damit er gegebenenfalls von außen gesetzt werden kann.

Sobald die Aktivität gestartet wird, wechselt der Status auf „running“. Dies signalisiert, dass die Aktivität gerade bearbeitet wird und der Zugriff von Adaptionsoperationen abzuweisen ist. Sollte ein Prozess angehalten werden, müssen alle gerade laufenden Aktivitäten in den Zustand „suspended“ übergehen, beim Fortsetzen des Prozesses werden die betroffenen Aktivitäten dann wieder in den Ursprungszustand „running“ zurück gesetzt. Wurde eine Aktivität erfolgreich beendet, so wird wie beschrieben der Endzustand „completed“ erreicht. Der Abbruch einer laufenden Aktivität verursacht dementsprechend einen Zustandsübergang nach „failed“.

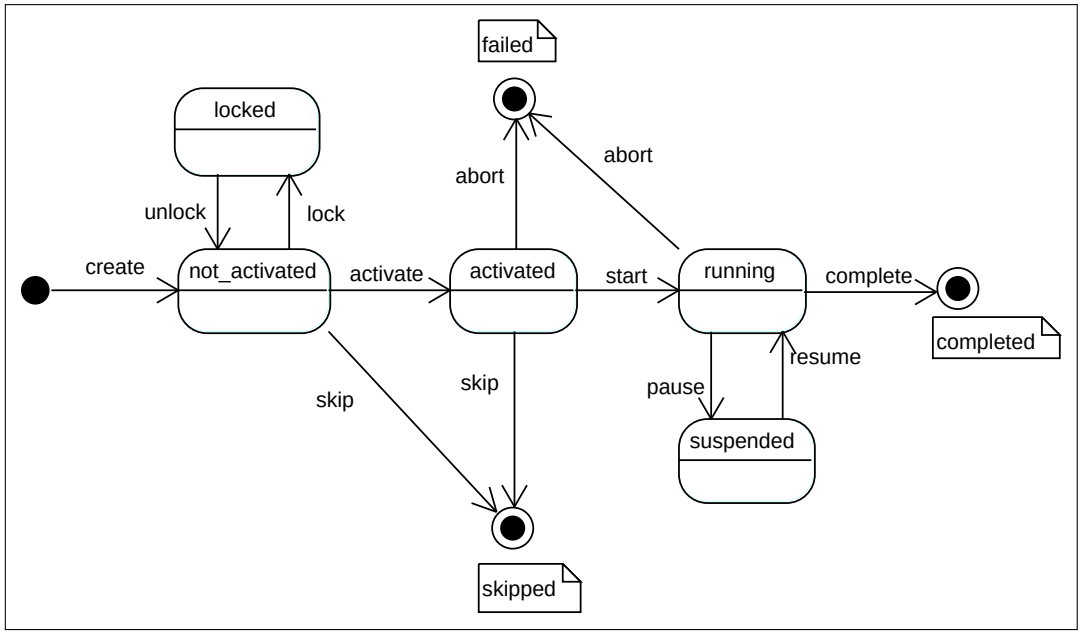

Abbildung 5.1: Zustandsdiagramm einer Aktivität

Speziell zur Unterstützung der ad-hoc Funktionalitäten wurde der Zustand „locked“ eingeführt. Dieser Zustand kommt zum Einsatz, wenn eine Adaptionsoperation den von einer Anpassung betroffenen Bereich markiert. So wird sichergestellt, dass der Pfad der Ausführung nicht in den Bereich einer laufenden Adaption eintritt, während diese ausgeführt wird.

Sobald die Adaption vorgenommen wurde, werden die zuvor gesperrten Aktivitäten wieder auf „not_activated“ zurückgesetzt.

5.1.2 Die jBPM Erweiterung

Es gibt unterschiedliche Herangehensweisen, wie die jBPM Workflow-Engine um die benötigten Funktionalitäten erweitert werden kann. Zum einen ist es denkbar einen „Zustandsmanager“ als neue Komponente zu entwerfen, der die Zustände und Zustandsübergänge aller beteiligten Entitäten verwaltet. Zum anderen ist es möglich, die nötigen Erweiterungen direkt an bereits vorhandenen Klassen der jBPM Engine vorzunehmen. Im folgenden werden beide Möglichkeiten kurz vorgestellt und bewertet.

Zustandsmanager

Die Idee hierbei ist es, den Status-Manager auf Basis eines Event-Listeners zu implementieren. Wie in Abschnitt 2.2 erläutert, besitzt die jBPM Engine ein umfangreiches Event Modell. Ein etwaiger Zustandsmanager könnte die für Zustandsübergänge relevanten Ereignisse überwachen und so den Status der Aktivitäten bestimmen. Hierfür würde vom Status-Manager eine Liste mit allen vorhandenen Nodes und Transitions der entsprechenden ProcessDefinition erstellt und diese um den jeweiligen Zustand ergänzt. Der Zustandsmanager wäre somit allein für die Verwaltung aller Zustände (setzen, abfragen, ...) verantwortlich.

Der große Vorteil dieses Ansatzes, ist die Unabhängigkeit des Zustandsmanagers. Durch die autarke Verwaltung der Zustände in einem „Add-On“ sind keine komplexen Änderungen im ursprünglichen jBPM Quellcode nötig. Die Methoden der Zustandsübergänge sowie die Methoden zur Abfrage des aktuellen Status können komplett in einer eigenständigen Komponente gekapselt werden. Der Zustandsmanager wäre demnach die alleinige Schnittstelle zur Bestimmung des Status.

Der Nachteil hierbei ist allerdings die doppelte Datenhaltung, die durch diesen Ansatz benötigt wird. Der Zustandsmanager muss die Bestandteile jeder ProcessDefinition zusätzlich zur internen jBPM Repräsentation noch einmal selbst vorhalten. Dies erzeugt zum einen unnötigen Overhead, zum anderen verkompliziert es die Adaption des Workflows. Ein weiterer Nachteil dieses Ansatzes ist, dass die Statusinformationen nicht direkt bei den Knoten gespeichert werden, auf die der Anpassungsdienst ohnehin zugreifen muss. Stattdessen erfolgt die Abfrage eines zusätzlichen Objektes. Außerdem besteht die Gefahr, dass nicht alle nötigen Zustandsübergänge durch Events repräsentiert und abgefangen werden können. Um dies zu gewährleisten, müsste die Event Logik im jBPM Quellcode erweitert werden. Wenn jedoch umfangreiche Anpassungen im jBPM Quellcode vorgenommen werden müssen, um einen Zustandsmanager zu ermöglichen, wird der eigentliche Vorteil einer „Add-On Funktionalität“ des Zustandsmanagers stark relativiert.

Erweiterung vorhandener Klassen

Im Gegensatz zum Zustandsmanager greift das nun vorgestellte Konzept von Grund auf in die Architektur von jBPM ein und erweitert bestehende Klassen um die benötigten Methoden bzw. zusätzlichen Klassen zur Zustandsverwaltung. Hierbei ist einleuchtend, dass stark in den bestehenden Quellcode von jBPM eingegriffen werden muss, um die betroffenen Klassen anzupassen. Es entsteht keine zusätzliche Komponente die den Status verwaltet, sondern die

bestehende Implementierung von jBPM wird um die benötigten Funktionalitäten im Kern erweitert.

Die Erweiterung der Engine setzt hierbei an der Klasse `GraphElement` an. Alle relevanten Entitäten, also Node, Transition und ProcessDefinition, erben in der Implementation von jBPM von dieser Klasse. Es wäre deshalb denkbar, ein entsprechendes Entwurfsmuster — das State Pattern [6, Kapitel 5] — einzusetzen.

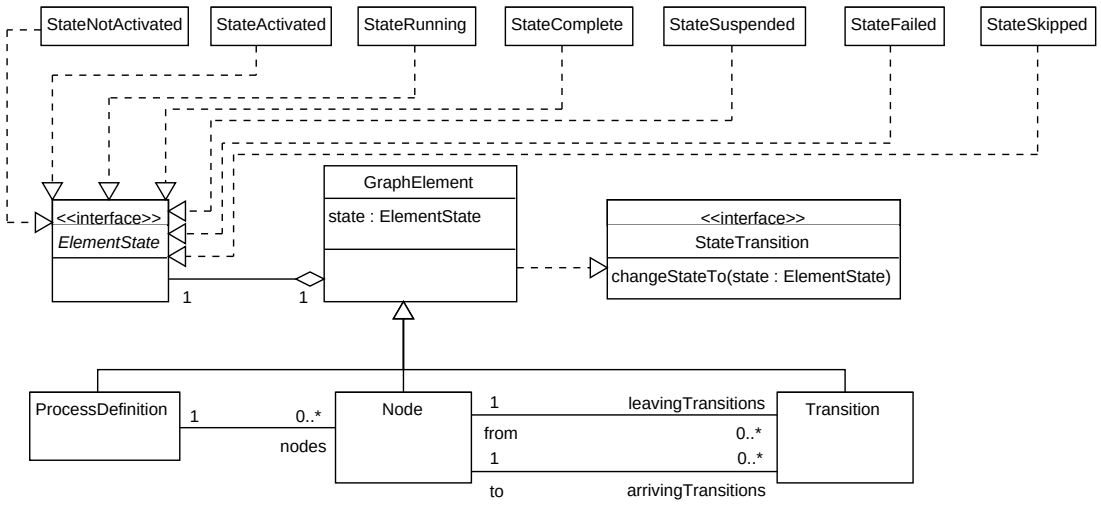

Abbildung 5.2: Klassendiagramm Statusrepräsentation mit State Pattern

Hierfür wird ein neues abstraktes Interface `ElementState` definiert. Die in 5.1.1 definierten Zustände werden durch konkrete Status Klassen (z.B. `StateRunning`) repräsentiert. Die Klasse `GraphElement` fungiert hierbei als Kontext und hält eine Instanz einer solchen konreten Sub-Klasse vor. Erfolgt nun ein Zustandsübergang, so ändert sich diese konkrete Status-Klasse, aus `StateRunning` wird z.B. `StateComplete`. Der Status-Handler bzw. zustandsabhängiges Verhalten wird jeweils komplett in die entsprechende Status-Klasse ausgelagert. Status-Anfragen und sonstige zustandsabhängige Funktionen werden von `GraphElement` an das aktuelle Status-Objekt delegiert. Das Klassendiagramm in Abbildung 5.2 verdeutlicht die Architektur.

Der Vorteil dieser Herangehensweise ist, dass vorhandene Zustände sehr leicht erweiterbar sind. Es muss einfach eine neue konkrete Sub-Klasse von `ElementState` erstellt werden, die den neuen Status repräsentiert. Außerdem ist das Verhalten innerhalb eines Zustands leicht anpassbar, da jede Status-Klasse eben genau das entsprechende Verhalten kapselt. Da die Zustandsübergänge (zumindest teilweise) in die entsprechenden Klassen ausgelagert werden können, sind weniger komplexe Bedingungsanweisungen im ursprünglichen jBPM Code erforderlich.

Nachteilig hierbei ist jedoch die relativ komplexe Implementierung des Entwurfsmusters, die möglicherweise unnötig ist. Die für die ad-hoc Adaption relevante Funktionalität ist kein echtes zustandsbehaftetes Verhalten, sondern der aktuelle Zustand hat keine weitere Aufgabe als genau diesen Zustand widerzuspiegeln. Nur die Abfrage des Zustandes durch die Adaptionsoperationen muss gewährleistet sein.

Aus diesem Grund ist eine einfachere und auf die Anforderungen zugeschnittene Implementierung der Zustände wünschenswert. Um dies zu ermöglichen ist es jedoch unabdingbar, die benötigten Zustandsrepräsentationen und Übergangsbedingungen noch tiefer in die

jBPM Engine einzubinden. Hierfür wird eine neue Klasse `ElementState` hinzugefügt, welche den aktuellen Status repräsentiert. Die Methoden dieser Klasse beschränken sich auf `getStateType` und `setStateType` und ein Attribut `StateType` beschreibt den aktuellen Status. Die Klasse `GraphElement` wird entsprechend erweitert und vererbt diese Eigenschaften auf die relevanten Komponenten `Node`, `Transition` und `ProcessDefinition`. Die eigentlichen Statusänderungen finden in den Implementationen dieser Komponenten statt. Zum Beispiel wird die Methode `enter(ExecutionContext)` der Klasse `Node` um den Aufruf `state.setStateType(ElementState.STATE_RUNNING)` erweitert.

Die Methode `Node.enter()` an sich wird ausgeführt, wenn der Pfad der Ausführung (also der Token) den Knoten erreicht. Entsprechend wird der Status auch in den Methoden anderer Komponenten neu gesetzt. Abbildung 5.3 zeigt das Klassendiagramm dieses Entwurfs.

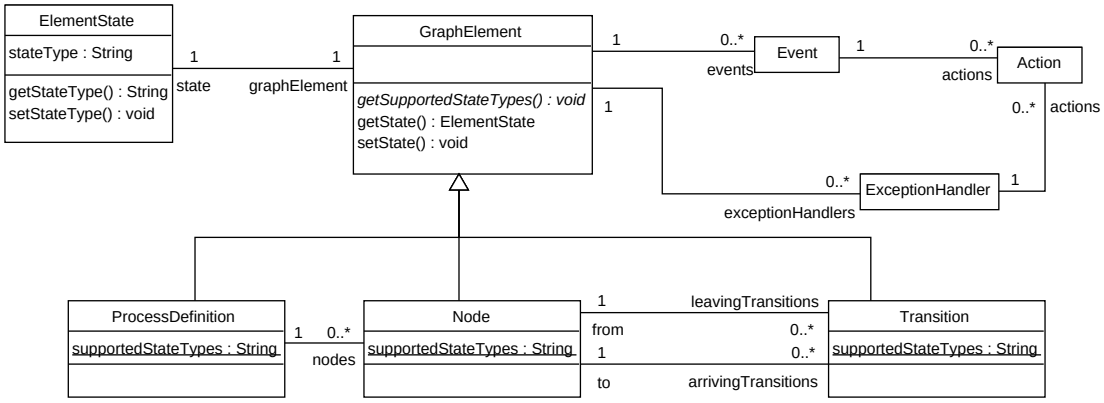

Abbildung 5.3: Klassendiagramm Statusrepräsentation

Dieser Entwurf ist eine einfache Möglichkeit, Zustände in jBPM zu modellieren. Es wird nur das implementiert, was wirklich nötig ist um den aktuellen Zustand von Elementen zu definieren und abzufragen. Es werden keine unnötigen Klassen definiert, die zustandsbehaftetes Verhalten abbilden sollen.

Der große Nachteil hierbei ist, dass ausführliche Anpassungen an unterschiedlichsten Stellen des jBPM Quellcodes vorgenommen werden müssen. Eine Erweiterung oder Änderung der Zustände zieht Code-Anpassungen verteilt über zahlreiche jBPM Klassen nach sich. Außerdem entstehen komplexe Bedingungsanweisungen, wann und wie welcher Zustandsübergang erfolgen soll.

Es wäre denkbar das Konzept noch weiter zu vereinfachen, indem man den aktuellen Zustand direkt durch ein Attribut der Klasse `GraphElement` repräsentiert. Allerdings widerspricht dies dem komponentenbasierten Design der jBPM Workflow-Engine. Aus diesem Grund wurde für die Implementierung der Statusinformationen die in Abbildung 5.3 vorgestellte Architektur gewählt.

Vergleich und Zusammenfassung

Im Vergleich zum oben vorgestellten Zustandsmanager erscheint die Erweiterung vorhandener Klassen wenig praktikabel, da eine spezielle Komponente zur Statusrepräsentation mit minimalen Eingriffen in die vorhandene Architektur grundsätzlich wünschenswert ist. Allerdings

sind die sich durch den Einsatz des Zustandsmanagers ergebenden Nachteile so umfangreich, dass der Einsatz zu viele Probleme aufwerfen würde.

Es stellt sich vor allem die Frage, was mit der Repräsentation des Workflow Schemas im Zustandsmanager geschieht, wenn eine Adaptionsoperation das Schema der Engine anpasst. In diesem Fall ist immer ein zusätzlicher Schritt notwendig, der beide Repräsentationen synchronisiert. Dieses Problem und die doppelte Datenhaltung im Zustandsmanager sind Ausschlusskriterien für dessen Einsatz.

Die in Abbildung 5.3 vorgestellte Architektur ist im Gegensatz dazu genau auf die Anforderungen des Adaptionsdienstes zugeschnitten. Es werden keine unnötigen Funktionalitäten implementiert, die Anpassungen an der jBPM Engine werden so begrenzt wie möglich gehalten. Trotzdem wird eine minimale Komponentenbasierung gewährleistet. Wenn die Repräsentation des Workflow Schemas durch die Anpassungsoperationen geändert wird, werden alle enthaltenen Statusinformationen ebenfalls angepasst. Das Zusammenspiel von Adaptionsdienst und Engine kann so auf die enthaltenen Statusinformationen abgestimmt werden.

5.2 Korrektheits- und Konsistenzkriterien

Die Anpassung von Workflows zur Laufzeit birgt unterschiedliche Risiken. Wie bereits in Kapitel 4.1 beschrieben, lässt sich zum einen die syntaktische und zum anderen die semantische Korrektheit eines Workflows unterscheiden. Die syntaktische Korrektheit wird im Rahmen dieser Arbeit in die beiden Bereiche „strukturelle Korrektheit“ und „erweiterte strukturelle Korrektheit“ aufgeteilt. So entsteht eine dreistufige Betrachtung von Korrektheit:

1. Strukturelle (syntaktische) Korrektheit
2. Erweiterte strukturelle Korrektheit
3. Semantische Korrektheit

Da die semantische Korrektheit nicht ohne ausführliche Kenntnisse über den Kontext sowie die semantische Bedeutung der entsprechenden Aktivitäten überprüft werden kann, werden im Rahmen dieser Arbeit nur die syntaktischen, also strukturellen Kriterien der Korrektheit betrachtet. Die Semantik der adaptierten Prozess Definitionen kann vom Adaptionsdienst nicht erfasst werden, da keinerlei semantische Informationen im Schema abgelegt sind. Die semantische Korrektheit muss demnach extern durch den Bearbeiter sichergestellt werden. Dies ist insofern eine problemlose Annahme, da im Anwendungsbereich ohnehin nur Einsatzleiter mit fundiertem Vorwissen über den Prozess eine Adaption vornehmen können.

Wenn eine Adaption jedoch ohne jegliche Überprüfung durchgeführt wird, werden vermeidbare strukturelle Fehler oder inkonsistente Zustände im betroffenen Workflow Schema auftreten. Aus diesem Grund müssen explizite syntaktische Korrektheits- und Konsistenzkriterien definiert werden, die die Adaptionsoperationen berücksichtigen müssen.

Grundsätzlich muss zu jeder Zeit sichergestellt werden, dass jeder Knoten des Workflow Graphen vom Startknoten aus *erreichbar* ist. Außerdem muss der Endknoten von jedem Knoten des Workflow Graphen aus erreichbar sein. Diese Eigenschaften sind in einem azyklischen Workflow Graphen (der also nur aus Sequenzen und symmetrischen Verzweigungen besteht) per Definition gewährleistet [vgl. 17, Kap. 2.3]. Sobald ein Graph Schleifen enthält, ist dies nicht mehr zwingend der Fall. Grundsätzlich sollen die angebotenen Adaptionsoperationen zumindest den ersten Fall abdecken, das heißt ein korrekter Workflow Graph soll durch die Anpassungsoperation in einen wiederum korrekten Workflow Graphen transformiert werden.

Einen ähnlichen Ansatz der Korrektheit „by Design“ verfolgen Reichert und Dadam [17], Küster u. a. [11] sowie Goedertier und Vanthienen [7].

Im folgenden werden zuerst Kriterien für die grundlegende strukturelle Korrektheit definiert. Hierbei werden die einzelnen strukturellen Konstruktionen getrennt betrachtet. Im Anschluss werden Vorgänge beschrieben, die Fehler der erweiterten strukturellen Korrektheit, wie zum Beispiel Deadlocks, im Workflow auslösen können. Die in Abschnitt 5.1.1 beschriebenen Zustandsinformationen sind unter anderem für die Sicherung der Konsistenz relevant, die bei gleichzeitigen Zugriffen von Engine und Adaptionsdienst auf benötigte Daten auftreten können. Zum Abschluss wird die grundsätzliche Anpassbarkeit von Workflow Graphen basierend auf den enthaltenen Zustandsinformationen diskutiert.

5.2.1 Strukturelle Korrektheit

Die drei grundlegenden Strukturierungselemente Sequenz, Nebenläufigkeit und Alternative unterliegen jeweils speziellen Kriterien der strukturellen Korrektheit.

Sequenzen

S1 Eine Sequenz besteht aus mindestens zwei Aktivitäten
Laut Definition ist eine Sequenz die sequentielle Ausführung mehrerer Aktivitäten (vgl. 2.1.3). Eine Sequenz, die nur aus einer Aktivität besteht, ist also eine Einzelaktivität. Die Unterscheidung ist auch deshalb wichtig, um relevante Anpassungsoperationen in verschachtelten strukturellen Konstruktionen auswählen zu können. Hierbei kommen Prioritätskriterien zum Einsatz die bestimmen welche Anpassungsoperation durchgeführt wird. Diese Kriterien müssen zwischen Einzelaktivität und Sequenz unterscheiden können. Eine Sequenz die nur noch eine Aktivität beinhaltet, wird ab sofort als Einzelaktivität betrachtet. Dabei werden die Eigenschaften des Workflows nicht beeinflusst.

S2 Verkettung von Sequenzen
Eine Sequenz s_1, die der direkte Vorgänger oder Nachfolger einer anderen Sequenz s_2 ist, kann ohne weitere Einflüsse auf den Workflow Graphen aufgelöst werden. Die Aktivitäten der Sequenz s_1 werden am entsprechenden Punkt an die Sequenz s_2 angehängt. Ein solcher Fall kann zum Beispiel dann auftreten, wenn eine von Sequenzen umgebene Nebenläufigkeit aufgelöst und in eine neue Sequenz umgewandelt wird. In diesem Fall entstehen drei aneinandergereihte Sequenzen, die in eine neue abgeschlossene Sequenz überführt werden müssen.

Nebenläufigkeiten

N1 Eine Nebenläufigkeit besteht aus mindestens zwei parallelen Zweigen
Eine Nebenläufigkeit ist per Definition (vgl. 2.1.3) die parallele Ausführung von mindestens zwei Aktivitäten. Aktivitäten können nur dann parallel ausgeführt werden, wenn sie auf unterschiedlichen Pfaden einer parallelen Verzweigung liegen. Eine Nebenläufigkeit mit nur einem Zweig lässt sich also in eine Einzelaktivität bzw. Sequenz umwandeln.

N2 Parallele Verzweigungen werden durch Synchronisierung abgeschlossen
Laut Definition beginnt jede Nebenläufigkeit immer mit einer parallelen Verzweigung und wird durch eine Synchronisierung abgeschlossen (vgl. 2.1.3). Diese beiden strukturellen Elemente können nicht unabhängig von einander auftreten. Außerdem muss jeder

Zweig, der aus einer parallelen Verzweigung hervorgeht, in der selben Synchronisierung enden. Die Zahl der Pfade, die aus der Verzweigung hervorgehen entspricht der Zahl der Pfade die in der Synchronisierung münden.

Alternativen

A1 Eine Alternative besteht aus mindestens zwei Zweigen
Eine Alternative Verzweigung kann nur dann einen Pfad auswählen, wenn mindestens zwei unterschiedliche Zweige zur Auswahl stehen. Eine Alternative mit nur einem Zweig lässt sich also in eine Einzelaktivität oder Sequenz umwandeln.

A2 Alt. Verzweigungen werden durch async. Zusammenführung abgeschlossen
Laut Definition beginnt jede Alternative immer mit einer alternativen Verzweigung und wird durch eine Asynchrone Zusammenführung abgeschlossen (vgl. 2.1.3). Diese beiden strukturellen Elemente können nicht unabhängig von einander auftreten. Außerdem muss jeder Zweig, der aus einer alternativen Verzweigung hervorgeht, in derselben asynchronen Zusammenführung enden. Die Zahl der Pfade, die aus der Verzweigung hervorgehen entspricht der Zahl der Pfade die in der Zusammenführung münden.

5.2.2 Erweiterte strukturelle Korrektheit

Neben den grundlegenden Kriterien der strukturellen Korrektheit für Sequenzen, Nebenläufigkeiten und Alternativen existieren zusätzliche Kriterien der „erweiterten“ strukturellen Korrektheit. Hierbei handelt es sich nicht um „harte“ Fehler, die durch die Definition der strukturellen Workflow Elemente abgedeckt werden (wie die in 5.2.1 beschriebenen Kriterien) — vielmehr handelt es sich um prinzipiell „korrekte“ Workflow Schemata, die allerdings während der Ausführung unter bestimmten Umständen Fehler produzieren. Als erstes Beispiel wird in Abbildung 5.4 ein solcher Fehler vorgestellt. Das hier abgebildete Workflow Schema ist

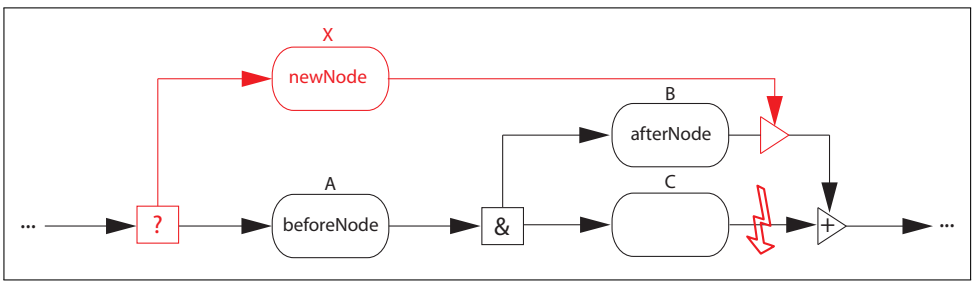

Abbildung 5.4: Erweiterte strukturelle Korrektheit: Deadlock

nach den in Abschnitt 5.2.1 definierten Regeln korrekt. Allerdings wird während der Ausführung des Prozesses mit hoher Wahrscheinlichkeit ein Deadlock auftreten. Ursprünglich war in diesem Fall eine Bearbeitung einer Einzelaktivität gefolgt von einer Nebenläufigkeit geplant $[A \rightarrow (B \wedge C)]$. Wird nun mit der in 5.3.1 definierten Operation „Alternatives Einfügen“ eine neue Aktivität hinzugefügt, muss als `beforeNode` die Aktivität (A) und als `afterNode` die Aktivität (B) gewählt werden, um die Grenzen der alternativen Verzweigung zu definieren. Falls die neue Asynchrone Zusammenführung wie in Abbildung 5.4 direkt im Anschluss an die Aktivität (B) — also vor der Synchronisierung von $(B \wedge C)$ — eingefügt wird, besteht die Gefahr eines Deadlocks. Wird während der Ausführung der neue alternative Pfad (X) ausgewählt, so wird die parallele Verzweigung übersprungen, es werden also (z.B. bei jBPM) keine

Child-Token erstellt (vgl. Abschnitt 2.2.4). Die Synchronisierung wird allerdings ausgeführt und diese befindet sich so lange in einem Wartezustand, bis sie die Child-Token beider Pfade erhalten hat. Da die Pfade jedoch nicht ausgeführt wurden, ergibt sich an dieser Stelle ein Deadlock und der Workflow wird nie in den Endzustand gelangen.

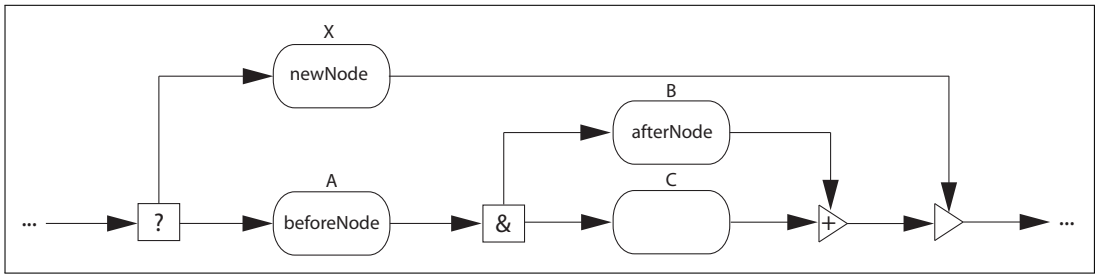

Abbildung 5.5: Erweiterte strukturelle Korrektheit: Deadlock II

Die Lösung für dieses Problem ist in Abbildung 5.5 zu erkennen. Hier wurde darauf geachtet, dass die Asynchrone Zusammenführung nicht Teil eines Zweiges der parallelen Verzweigung ist. Durch dieses Vorgehen wird im Falle der Ausführung von Aktivität (X) die komplette parallele Konstruktion übergangen und es wird kein Deadlock auftreten. Daraus leitet sich die erste Regel zur erweiterten strukturellen Korrektheit ab:

K1 Abgeschlossenheit von Nebenläufigkeiten
Eine Vermischung von Alternativen und Nebenläufigkeiten ist nur dann erlaubt, wenn die strukturellen Bestandteile der Alternative entweder *komplett auf einen Pfad* oder *komplett außerhalb* der Nebenläufigkeit platziert werden. Eine Nebenläufigkeit muss entweder ganz oder gar nicht ausgeführt werden, da eine Synchronisierung immer alle eingehenden Pfade bearbeitet.

Das zweite Beispiel betrifft die spezifische Zuordnung von Aktivitäten zu Pfaden der Ausführung. Wie in Abbildung 5.6 zu sehen ist, wurde hier die Aktivität (B) gleichzeitig auf die Pfade zweier unterschiedlicher Nebenläufigkeiten platziert. Grundsätzlich ist das Schema nach den in Abschnitt 5.2.1 definierten Regeln korrekt (alle benötigten strukturellen Elemente sind vorhanden). Hierbei tritt folgendes Problem auf: Die Aktivität (B) soll in der Konstruktion

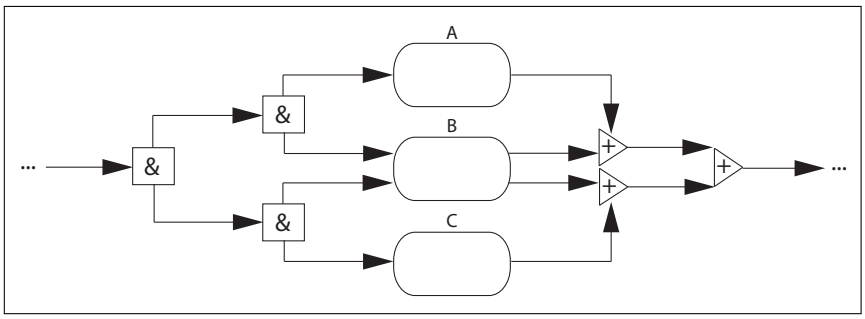

Abbildung 5.6: Erweiterte strukturelle Korrektheit: Deadlock III

$[(A \wedge B) \wedge (B \wedge C)]$ zwei mal ausgeführt werden. Allerdings können nicht beide Child-Token gleichzeitig auf den Knoten der Aktivität zugreifen, hier ist die Konsistenz der enthaltenen Daten nicht mehr gewährleistet. Es entsteht eine Race-Condition[1]. Ein Zweig der äußeren

1 Race-Condition: Kritischer Wettlauf — das Ergebnis einer Operation hängt vom zeitlichen Verhalten beteiligter Einzeloperationen ab

Nebenläufigkeit schlägt also zwangsläufig fehl und die abschließende Synchronisierung produziert einen Deadlock.

Es lassen sich noch andere Beispiele finden, die die Probleme mit Aktivitäten auf unterschiedlichen Pfaden eines Schemas verdeutlichen. Die spezifischen Pfade sind hierbei nicht auf Zweige von Nebenläufigkeiten beschränkt. Es treten ebenfalls Probleme auf, wenn Alternativen mit Nebenläufigkeiten vermischt werden und die Nebenläufigkeit Aktivitäten auf unterschiedlichen Zweigen der Alternative parallelisiert — z.B. in der Konstruktion $[A \vee (A \wedge B)]$. Das Problem wird durch den Einsatz der jBPM Engine noch deutlicher: Ein Knoten, der zwei abgehende Transitionen besitzt, wird automatisch als parallele Verzweigung behandelt. Aus diesem Grund wird die zweite Regel zur erweiterten strukturellen Korrektheit definiert:

K2 Spezifische Zuordnung von Aktivitäten
Jede Aktivität liegt auf genau einem Pfad der Ausführung. Das heißt jede Aktivität besitzt auch nur eine eingehende und eine abgehende Transition. Die Bündelung mehrerer abgehender/eingehender Transitionen ist Verzweigungselementen von Alternativen und Nebenläufigkeiten vorbehalten.

5.2.3 Zugriffskonsistenz und Status

Die in Abschnitt 5.1 definierten Zustände sind zum einen Grundlage für die Sicherung der Konsistenz bei gleichzeitigen Zugriffen von Engine und Adaptionsdienst. Zum anderen werden die Zustandsinformationen benötigt, um Entscheidungen zur grundsätzlichen Anpassbarkeit einer Aktivität zu treffen.

Konsistenz bei gleichzeitigen Zugriffen

Die Anforderung A1.3 verlangt, dass kein gleichzeitiger Zugriff von Workflow-Engine und Adaptionsdienst auf ein Element des Workflow Schemas stattfinden darf. Hierbei muss die Konsistenz von zwei Seiten aus gewährleistet werden.

Einerseits darf die Engine z.B. nicht auf eine Aktivität zugreifen, die gerade vom Adaptionsdienst bearbeitet wird bzw. mit einer laufenden Anpassung in Verbindung steht. Die in Abschnitt 5.1.1 vorgeschlagene Zustandsmodellierung berücksichtigt dieses Problem durch die Einführung des Status „locked“. Auf Seite 48 wurde dieser Ansatz bereits erläutert. Grundsätzlich wird die Ausführung durch diese Vorgehensweise nicht in einen Bereich vordringen können, der gerade von einer Adaption betroffen ist. Um dies zu erreichen, muss die jBPM Engine so erweitert werden, dass der Token nur dann in eine Node eintreten kann, wenn diese sich nicht im Status „locked“ befindet. Hierfür wird die `Signal` Methode des Tokens angepasst, so dass sie den Status der folgenden Knoten abfragt bevor die entsprechende Transition ausgelöst werden kann. Außerdem muss der Adaptionsdienst alle mit einer Anpassung verknüpften Entitäten bestimmen können — *Bereich der Adaption* — und deren Status auf „locked“ setzen, so lange die entsprechende Adaption vorgenommen wird. Der Bereich der Adaption ist grundsätzlich die betroffene Aktivität bzw. strukturelle Konstruktion (Sequenz, Nebenläufigkeit, Alternative). Zusätzlich müssen immer die Vorgänger- und Nachfolgeknoten des direkt betroffenen Konstrukts in der jBPM ProcessDefinition mit markiert werden, da diese im Rahmen der Adaption ebenfalls betroffen sein können (z.B. wenn eine abgehende Transition angepasst werden muss).

Andererseits darf der Adaptionsdienst nicht auf Komponenten des Workflow Schemas zugreifen, die aktuell von der Engine benötigt werden – also gerade von ihr ausgeführt werden.

Hierfür kommen ebenfalls die Statusrepräsentationen aus Abschnitt 5.1 zum Einsatz. Wie bereits in der Anforderung A1.2 festgestellt, ist es an sich undenkbar eine Aktivität zu adaptieren, die sich gerade in Ausführung befindet. Zudem kommen auch die beschriebenen Probleme der Zugriffskonsistenz zum Vorschein. Die Lösung ist die Überprüfung des aktuellen Zustands aller an einer Adaption beteiligter Entitäten (Bereich der Adaption) durch den Adaptionsdienst, bevor die Anpassung freigegeben wird. Nur wenn sich der komplette Bereich der Adaption im Zustand „not_activated“ befindet, kann die gewünschte Anpassung durchgeführt werden. Eine Anpassung wäre prinzipiell, wenn nur Gründe der Zugriffskonsistenz betrachtet werden, in allen Zuständen — außer Status „running“ — möglich. Allerdings wird die Einschränkung auf „not_activated“ gesetzt, aus Gründen die im folgenden Abschnitt näher erläutert werden.

Grundsätzliche Anpassbarkeit

Neben den einerseits rein zur Sicherung der Konsistenz benötigten Einschränkungen, werden andererseits noch Festlegungen zur grundsätzlichen Anpassbarkeit von Komponenten getroffen. Hierbei ist es nötig die restlichen in Abschnitt 5.1.1 definierten Zustände zu betrachten.

Der Zustand „locked“ bedingt eine sofortige Abweisung. Befindet sich eine Entität des Schemas in diesem Zustand, so ist sie bereits Teil einer anderen Adaption und kann von einer neuen Anpassung nicht erfasst werden. Der speziell für das Projekt SoKNOS relevante Status „activated“ (siehe S. 47) beschreibt, dass die Vorbereitungen für eine Aktivität bereits getroffen werden sollen. Deshalb ist auch eine Aktivität im Zustand „activated“ nicht für eine Adaption geeignet, da diese Sekundär-Arbeiten möglicherweise bereits ausgeführt werden. Prinzipiell besitzen Engine und Adaptionsdienst keine weiteren Informationen über den Zustand „activated“, deshalb ist eine Adaptionsanfrage abzuweisen. Der Zustand „suspended“ beschreibt eine pausierte Ausführung. Eine sich in diesem Zustand befindliche Komponente darf nicht adaptiert werden, da die Ausführung zu jedem Zeitpunkt wieder starten kann. Selbst wenn die Adaption beendet worden ist, bevor die Ausführung weiter geht, kann es zu Problemen der Konsistenz kommen, da Engine einen unveränderten Knoten erwartet. Die Zustände „failed“, „skipped“ und „completed“ sind Endzustände. Befindet sich ein Element in einem Endzustand, wurde es bereits ausgeführt bzw. der Pfad der Ausführung ist über dieses Element gelaufen. Deshalb ist es unsinnig, eine Anpassung an einem solchen Element vorzunehmen.

Wie bereits in der Einleitung zu Abschnitt 5.2 beschrieben, lassen sich die Kriterien der Korrektheit und Konsistenz nicht umsetzen, wenn ein Workflow Graph Schleifen (Iterationen) enthält. Aus diesem Grund wird im Rahmen dieser Arbeit auf die Betrachtung von Schleifen verzichtet. Die im Anwendungsbereich der Arbeit betrachteten Workflow Graphen sind also immer *endliche gerichtete schleifenfreie azyklische Graphen*. Zyklische Graphen könnten zwar mit den angebotenen Operationen angepasst werden, die Korrektheit der Adaption allerdings nicht gewährleistet werden.

Prioritätskriterium

Bei der Löschung von strukturellen Elementen muss definiert werden, nach welcher Priorität die entsprechenden Adaptionsoperationen eingesetzt werden sollen. So sind zum Beispiel Verschachtelungen und Vermischungen von Sequenzen und Nebenläufigkeiten in einem Workflow Schema möglich, wie sie in den Abbildungen 5.9 und 5.18 zu erkennen sind. Sobald der Zweig einer Alternative oder Nebenläufigkeit nicht nur aus einer Einzelaktivität, sondern z.B. aus

einer Sequenz besteht, muss hier das Muster „Entfernen aus Sequenz“ zum Einsatz kommen. Nur wenn der Zweig aus einer isolierten Einzelaktivität besteht, kommen die entsprechenden Spezialoperationen zum Tragen, die auch Teile oder komplette Konstruktionen von Nebenläufigkeiten/Alternativen entfernen.

P1 Prioritätskriterium
Beim Entfernen von Aktivitäten werden die angebotenen Muster nach folgender Priorität angewandt: „Entfernen aus Sequenz“ → „Entfernen aus Nebenläufigkeit/Alternative“ → „Entfernen von Einzelaktivitäten“

5.3 Strukturelle Anpassungsoperationen

In [22] und [23] werden 18 sogenannte „change patterns“ vorgestellt, von denen 14 „adaptation patterns“ relevante strukturelle Anpassungen definieren. Diese Muster dienen als Grundlage für die Klassifikation und Erstellung der Adaptionsoperationen in diesem Abschnitt.

Die Muster beschreiben „high-level operations“ — d.h. es werden nicht die atomaren „primitiven“ Schritte (Elementar-Operationen wie z.B. `addNode, addTransition, ...`) definiert, die zur Umsetzung einer speziellen Adaption (Einzel-Operationen wie z.B. Aktivität hinzufügen) notwendig sind. Eine solche „low-level“ Adaption birgt sehr viele Risiken, da durch die Ausführung einzelner atomarer Schritte die Korrektheit des Schemas nicht mehr garantiert werden kann (oder erweiterte strukturelle Inkorrektheiten wie z.B. Deadlocks auftreten können) und diese im Anschluss verifiziert werden muss. Stattdessen wird eine höhere Abstraktionsebene eingeführt und die atomaren Schritte werden zu komplexeren aber kompletten Adaptionsoperation zusammengefasst (siehe Abbildung 5.21). Durch dieses Vorgehen kann die Korrektheit einer Anpassung wesentlich besser gewährleistet werden, indem man die Ausführung der Operationen nur unter bestimmten Bedingungen erlaubt. Eine nähere Betrachtung der Sicherstellung von Korrektheit findet in Abschnitt 5.4 statt.

Aus der Menge der möglichen Adaptionen wurden die für das Anwendungsgebiet der Arbeit relevanten ausgewählt. Die Anpassungen werden mittels der in Abschnitt 2.1.3 eingeführten graphischen Notation vorgestellt und durch entsprechende Abbildungen erläutert. Meist ist ein gewisser Kontext in der Abbildung aus Verständnisgründen nicht zu vermeiden — dieser wird jedoch so gering wie möglich gehalten. Außerdem werden die Schnittstellenparameter der jeweiligen Operationen definiert und kurz tabellarisch aufgelistet.

Zuerst werden grundlegende Adaptionen vorgestellt, die als kleinste Einheiten sinnvoller Anpassungen dienen. Diese Einzel-Operationen können gruppiert und zu neuen (komplexeren) Operationen zusammengesetzt werden.

5.3.1 Aktivität einfügen

Die Operation „Aktivität einfügen“ ist eine der grundlegenden Workflow Adaptionen. Sie wird dann eingesetzt, wenn in einem Bereich des Workflow Schemas (der noch nicht ausgeführt wurde), eine zusätzliche Aktivität benötigt wird. Die betroffene Aktivität wurde zum Zeitpunkt der Erstellung der Prozess Definition nicht eingeplant, ihr Bedarf wird aber im Laufe der Vorgangsbearbeitung ersichtlich.

Zum Beispiel kann es vorkommen, dass ein Gebäude evakuiert werden muss weil es in der Nähe einer Gefahrenquelle steht. Die Existenz von allen gefährdeten Gebäuden kann nicht

in einem universellen Einsatzplan vorhergesehen werden. Deshalb kann diese Aktivität auch nicht in einer entsprechenden Prozess Definition zur Designzeit modelliert werden.

Aus diesem Grund müssen Adaptionsoperationen angeboten werden, die das Hinzufügen von neuen Aktivitäten zur Laufzeit ermöglichen. Es lassen sich hierbei drei Möglichkeiten einer Operation „Aktivität einfügen“ unterscheiden, die jeweils zwei explizite Anwendungsfälle besitzen:

1. Sequentielles Einfügen
   - Sequenzerweiterung
   - Sequenzerzeugung
2. Paralleles Einfügen
   - an Sequenz
   - an Nebenläufigeit
3. Alternatives Einfügen
   - an Sequenz
   - an Alternative

Die daraus resultierenden Adaptionsoperationen werden nun vorgestellt.

Sequentielles Einfügen

Prinzipiell lässt sich das Sequentielle Einfügen in zwei unterschiedliche Anwendungsfälle unterteilen. Die Unterscheidung der beiden Fälle ist nötig, weil das Kriterium für strukturelle Korrektheit S1 dies vorschreibt. Eine Sequenz besteht demnach immer aus mindestens zwei Aktivitäten, also muss zwischen Einzelaktivität und Sequenz unterschieden werden.

Im ersten Fall wird eine bereits vorhandene Sequenz um eine weitere Aktivität erweitert — die Sequenzerweiterung, Abbildung 5.7. Hier wird die ursprüngliche Sequenz der Aktivitäten

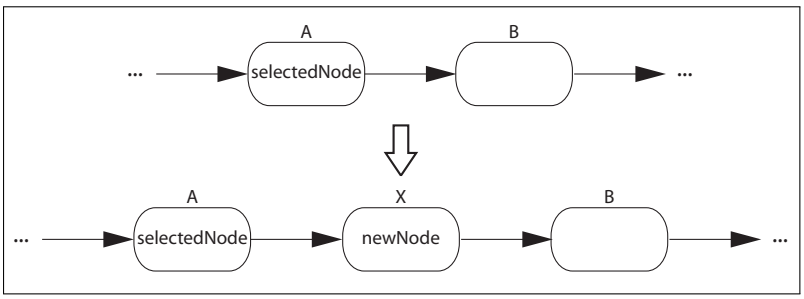

Abbildung 5.7: Sequenzerweiterung

$(A \rightarrow B)$ um eine neue Aktivität (X) erweitert. Aus dieser Operation entsteht die neue Sequenz $(A \rightarrow X \rightarrow B)$.

Im zweiten Fall wird eine einzelne Aktivität, die von anderen strukturellen Konstruktionen umgeben ist, durch das Einfügen einer weiteren Aktivität zum Teil einer neu entstandenen Sequenz — die Sequenzerzeugung, Abbildung 5.8. Bei diesem Beispiel wird nach der ehemaligen Einzelaktivität (A) eine neue Aktivität (X) eingefügt. Aus diesem Vorgang resultiert die neu entstandene Seqeuenz $(A \rightarrow X)$.

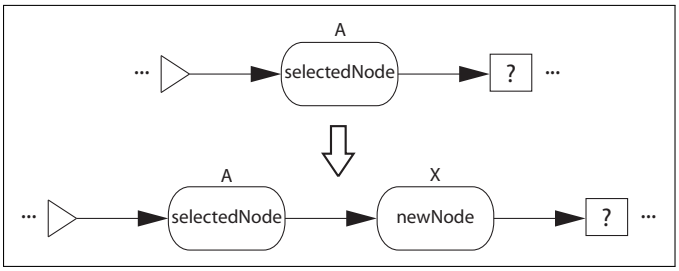

Abbildung 5.8: Sequenzerzeugung

Bei der Operation `sequentialInsert` muss entschieden werden, an welcher Stelle die Erweiterung stattfindet. Die neue Aktivität kann nicht grundsätzlich immer *nach* oder immer *vor* dem gewählten Adaptionspunkt `selectedNode` eingefügt werden. Gerade im zweiten Fall würde dies zu Problemen führen. Um eine universelle Adaptionsoperation zu ermöglichen, müssen die Übergabeparameter neben dem Namen der Aktivität am Adaptionspunkt `selectedNode` auch noch Informationen darüber enthalten, ob vor oder nach der gewählten Aktivität eingefügt werden soll. Die dafür nötigen Parameter der Operation sind in Tabelle 5.1 aufgelistet.

Parameter	**Bedeutung**
`processInstance`	Prozess Instanz, an der die Anpassung vorgenommen werden soll
`newNode`	Neue sequentielle Aktivität
`selectedNode`	Aktivität, die mit neuer Aktivität in direkter Verbindung steht
`pointOfInsert`	Relativer Einfügepunkt

Tabelle 5.1: Schnittstellenparameter für `sequentialInsert`

Paralleles Einfügen

Auch das Parallele Einfügen lässt sich in zwei unterschiedliche Anwendungsfälle unterteilen. Es ist zu unterscheiden, ob eine neue Aktivität parallel zu einer oder mehreren zuvor sequentiell ausgeführten Aktivitäten eingefügt werden soll, oder ob eine bereits bestehende Nebenläufigkeit erweitert werden kann. Hierbei sind die Kriterien der strukturellen Korrektheit für Nebenläufigkeiten N1 und N2 relevant. Außerdem ist das Kriterium für erweiterte strukturelle Korrektheit K2 von Bedeutung.

Im ersten Fall müssen außer der eigentlichen neuen Aktivität auch die nötigen strukturellen Erweiterungen — also Parallele Verzweigung und Synchronisierung — hinzugefügt werden. Die Operation benötigt also sowohl den Startpunkt, als auch den Endpunkt der Nebenläufigkeit als Übergabeparameter. Hierfür müssen die beiden Aktivitäten `beforeNode` und `afterNode` ausgewählt werden, die die Grenzen der gewünschten Verzweigung bilden sollen. Die `beforeNode` ist die Aktivität, *vor* der die Nebenläufigkeit gestartet werden soll. Als `afterNode` wird die Aktivität gewählt, *nach* der die anschließende Synchronisierung stattfinden wird. Für den Fall, dass die neue Aktivität parallel zu nur einer vorhandenen Aktivität eingefügt werden soll, so ist diese vorhandene Aktivität sowohl `beforeNode` als auch `afterNode` zugleich. Dieses „Parallele Einfügen an einer Sequenz“ ist in Abbildung 5.9 zu erkennen. In diesem Beispiel soll die Sequenz der Aktivitäten ($A \rightarrow B \rightarrow C$) so angepasst werden, dass eine neue Aktivität (X) parallel zu der gesamten Sequenz ausgeführt wird. Die entstehende

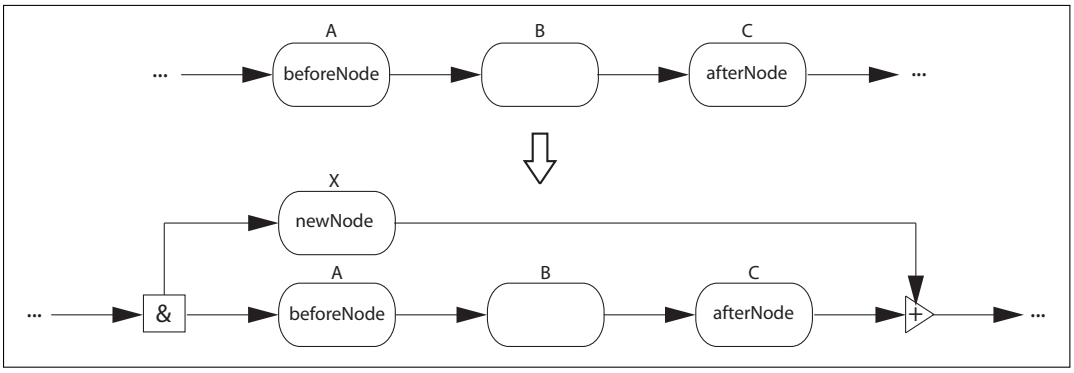

Abbildung 5.9: Paralleles Einfügen an Sequenz

Nebenläufigkeit $[X \wedge (A \rightarrow B \rightarrow C)]$ besitzt somit zwei Zweige, von denen einer der ehemalige Sequenz entspricht.

Soll die neue Aktivität parallel zu einer bereits bestehenden Nebenläufigkeit eingefügt werden, so können die vorhandenen Komponenten der parallelen Verzweigung weiterverwendet werden. Es sind nur zusätzliche Transitionen hinzuzufügen, die die existierende UND-Verzweigung entsprechend erweitern. Die Auswahl der `beforeNode` und `afterNode` fällt hierbei auf die entsprechenden Aktivitäten an den Grenzen der Verzweigung. Abbildung 5.10 verdeutlicht dieses „Parallele Einfügen an Nebenläufigkeit“. Hierbei wird die ehemalige Ne-

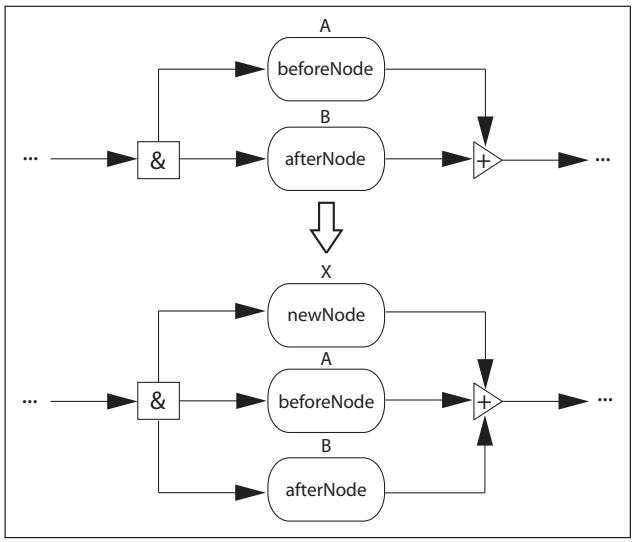

Abbildung 5.10: Paralleles Einfügen an Nebenläufigkeit

benläufigkeit von zwei Aktivitäten $(A \wedge B)$ um einen zusätzlichen parallelen Zweig erweitert. Das Ergebnis ist die parallele Ausführung von drei Aktivitäten $(X \wedge A \wedge B)$.

Eine Adaptionsoperation `parallelInsert` wird nur dann eine neue Nebenläufigkeit erzeugen, wenn der Vorgänger der `beforeNode` keine bestehende UND-Verzweigung und der Nachfolger der `afterNode` keine bestehende Synchronisierung ist. Um eine bestehende Nebenläufigkeit um zusätzliche Aktivitäten zu erweitern, muss daher immer eine Aktivität als Einstiegspunkt gewählt werden, die ein direkter Nachfolger der entsprechenden UND-Verzweigung ist. Durch diese Automatisierung benötigt eine Adaptionsoperation `parallelInsert` keine zusätzlichen

Informationen und kann universell für beide Anwendungsfälle zum Einsatz kommen. Tabelle 5.2 beschreibt die nötigen Schnittstellenparameter.

Parameter	Bedeutung
`processInstance`	Prozess Instanz, an der die Anpassung vorgenommen werden soll
`newNode`	Parallele Aktivität, die eingefügt werden soll
`beforeNode`	Aktivität, vor der die Nebenläufigkeit gestartet werden soll
`afterNode`	Aktivität, nach der die Synchronisierung stattfindet

Tabelle 5.2: Schnittstellenparameter für `parallelInsert`

Alternatives Einfügen

Die Anpassung Alternatives Einfügen lässt sich nach ähnlichen Gesichtspunkten in zwei unterschiedliche Anwendungsfälle unterteilen. Hierbei ist von Bedeutung, ob an der Stelle der gewünschten Adaption bereits eine ODER-Verzweigung vorhanden ist, oder nicht. Hierbei sind die Kriterien der strukturellen Korrektheit für Alternativen A1 und A2 relevant. Zudem ist das Kriterium für erweiterte strukturelle Korrektheit K1 von besonderer Bedeutung.

Falls die neue Aktivität alternativ zu einer oder mehreren vorher sequentiell ausgeführten Aktivitäten eingefügt werden soll, sind entsprechende strukturelle Erweiterungen notwendig. Neben der eigentlichen neuen Aktivität muss dann außerdem die Alternative Verzweigung und zugehörige Asynchrone Zusammenführung erstellt werden. Zudem müssen die Entscheidungskriterien (in Abbildung 5.11 mit „?“ gekennzeichnet) bestimmt werden, die während der Bearbeitung die benötigte Aktivität aus den vorhandenen Alternativen auswählen. Genau wie bei `parallelInsert` muss auch für eine Operation `AlternateInsert` der Start- und Endpunkt der neuen Alternative definiert werden. Hierfür werden wieder die beiden Aktivitäten `beforeNode` und `afterNode` ausgewählt, die die Grenzen der alternativen Verzweigung bilden. Die Bedeutung ist analog zu `parallelInsert`. Auch hier kann eine Einzelaktivität gleichzeitig `beforeNode` und `afterNode` sein und so das alternative Einfügen zu dieser ermöglichen. Das „alternative Einfügen an einer Sequenz“ ist in Abbildung 5.11 dargestellt. In

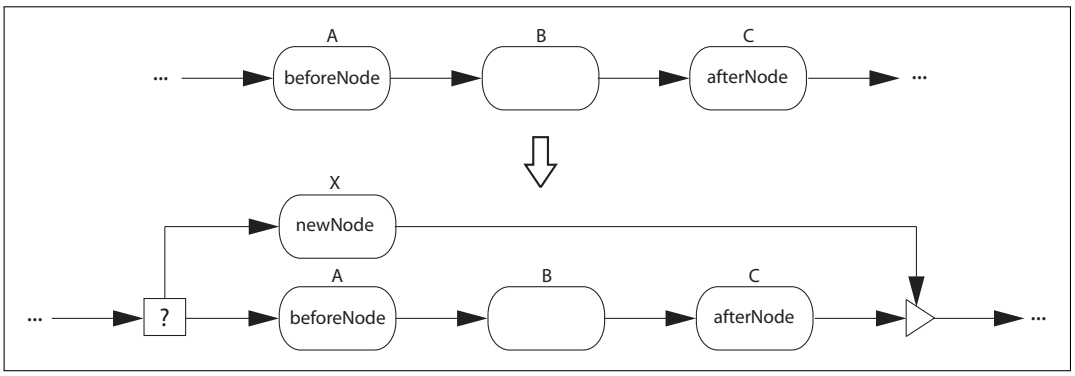

Abbildung 5.11: Alternatives Einfügen an Sequenz

dem Beispiel soll die bestehende Sequenz von Aktivitäten ($A \rightarrow B \rightarrow C$) in Zukunft nur noch alternativ zu einer neuen Aktivität (X) ausgeführt werden. Es werden also die entsprechenden Entscheidungskriterien (?) in die neu entstehende Alternative eingefügt, so dass die neue Struktur der gewünschten Funktionalität $[X \vee (A \rightarrow B \rightarrow C)]$ entspricht.

Das Kriterium für erweiterte strukturelle Korrektheit K1 stellt hierbei besondere Anforderungen. So muss überprüft werden, ob die Aktivitäten `beforeNode` oder `afterNode` Teil des Zweiges einer Nebenläufigkeit (wie in Abbildung 5.4) sind. In diesem Fall muss sichergestellt werden, dass keinerlei Verzweigungselemente der neuen Alternative innerhalb der existierenden Nebenläufigkeit platziert werden.

Ebenfalls möglich ist das Hinzufügen einer neuen Alternative zu einer bereits bestehenden Menge von Alternativen. In diesem Fall kann allerdings die bereits bestehende ODER-Verzweigung nicht ohne weitere Anpassung weiterverwendet werden. Die ursprünglichen Entscheidungskriterien (?) müssen von der Adaptionsoperation in die — um die neue Alternative erweiterten — neuen Entscheidungskriterien (?′) überführt werden. Dafür benötigt die Operation einen entsprechenden Übergabeparamter. Die strukturelle Komponente der asynchronen Zusammenführung kann weiterverwendet werden. Die Auswahl von `beforeNode` und `afterNode` fällt hierbei ebenfalls auf die Aktivitäten, die die Grenze der Verzweigung bilden. Abbildung 5.12 verdeutlicht das Vorgehen dieser Operation. Die ehemalige Alternative aus

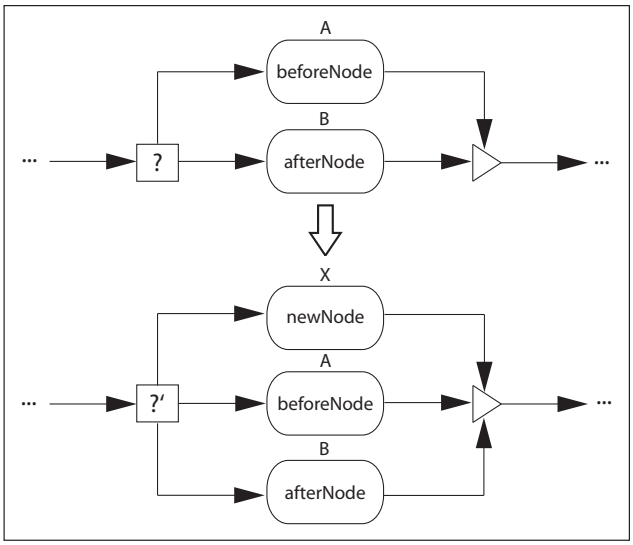

Abbildung 5.12: Alternatives Einfügen an Alternative

zwei Aktivitäten ($A \vee B$) wird um eine weitere alternative Aktivität (X) erweitert. Die bestehenden Entscheidungskriterien (?) werden durch die neuen Kriterien (?′) ersetzt, so dass eine die neue Struktur ($X \vee A \vee B$) entstehen kann.

Die Operation `AlternateInsert` überprüft auch die den Vorgänger der `beforeNode` und den Nachfolger der `afterNode` um zu entscheiden, ob eine neue ODER-Verzweigung erstellt werden muss oder eine bestehende erweitert werden soll. Deshalb muss bei einer Erweiterung von bestehenden Alternativen auch immer die `beforeNode` direkt nach der Verzweigung und analog die `afterNode` vor der Zusammenführung ausgewählt werden. Durch dieses Vorgehen kann eine universelle Operation `AlternateInsert` angeboten werden, was die Bedienbarkeit des Systems vereinfacht. Die entsprechenden Parameter der Operation sind in Tabelle 5.3 zu erkennen.

5.3.2 Aktivität entfernen

Genauso wie es möglich ist, dass neue Aktivitäten hinzugefügt werden sollen, kann es vorkommen, dass bestehende Aktivitäten überflüssig werden und entfernt werden müssen. Die

Parameter	Bedeutung
`processInstance`	Prozess Instanz, an der die Anpassung vorgenommen werden soll
`newNode`	Alternative Aktivität, die eingefügt werden soll
`beforeNode`	Aktivität, vor der die Alternative gestartet werden soll
`afterNode`	Aktivität, nach der die Zusammenführung stattfindet
`newConditions`	Neue Entscheidungskriterien

Tabelle 5.3: Schnittstellenparameter für `alternateInsert`

Operation „Aktivität entfernen“ ist daher eine weitere grundlegende Workflow Adaption. Sie wird dann eingesetzt, wenn eine Aktivität zur Designzeit eingeplant wurde, die zur Laufzeit aber nicht mehr benötigt wird. Eine Ausführung könnte sogar negative Auswirkungen haben und sollte deshalb verhindert werden. In jedem Fall muss die betroffene Aktivität aus dem Workflow entfernt werden.

Zum Beispiel wäre es denkbar, dass die geplante Errichtung eines Sandsackwalls zum Hochwasserschutz übersprungen werden kann, da das Hochwasser (unerwartet) doch nicht in den betroffenen Bereich vorgedrungen ist. Die korrespondierende Aktivität muss deshalb aus dem Workflow entfernt werden, damit keine überflüssigen Arbeiten ausgeführt werden. Natürlich könnten die relevanten Work Items auch direkt nach dem Start der Aktivität vom verantwortlichen Benutzer als erledigt gekennzeichnet werden, ohne die eigentliche „reale Aufgabe“ zu erfüllen. Allerdings sollte eine solche „Umgehung“ des Workflow Schemas immer vermieden werden, da so der eigentliche Sinn eines WfMS untergraben wird und der Einsatz eine Belastung für die Benutzer wird.

Aus diesem Grund ist es nötig, entsprechende Operationen zum Löschen von Aktivitäten zur Verfügung zu stellen. Eine Operation „Aktivität entfernen“ erscheint nicht sehr komplex, wenn der Kontext (die Umgebung) der betroffenen Aktivität nicht betrachtet wird. In diesem Fall könnte die Aktivität einfach aus dem Schema entfernt werden indem man die ehemaligen Vorgänger- bzw. Nachfolge-Aktivitäten mit einer neuen Transition verbindet und die gewählte Aktivität löscht.

Sobald man allerdings den Kontext der zu löschenden Aktivität mit einbezieht, ergeben sich unterschiedliche Probleme der Korrektheit. Wird z.B. die Aktivität einer parallelen Verzweigung entfernt, müssen möglicherweise weitere Schritte ausgeführt werden um die parallele Verzweigung komplett zu entfernen (vgl. Kriterium der strukturellen Korrektheit von Nebenläufigkeiten N1). Aus diesem Grund müssen auch für die Anpassung „Aktivität entfernen“ unterschiedliche Anwendungsfälle betrachtet werden. Hierbei sind folgende Möglichkeiten von Relevanz:

1. Entfernen einer Einzelaktivität
2. Entfernen aus einer Sequenz
   - Sequenzkürzung
   - Sequenzlöschung
3. Entfernen aus einer Nebenläufigkeit
   - Nebenläufigkeit mit ≥ 3 Zweigen
   - Minimale Nebenläufigkeit
4. Entfernen aus einer Alternative

- Alternative mit ≥ 3 Zweigen
- Minimale Alternative

Eine definierte Operation `DeleteNode` lässt sich für alle Anwendungsfälle von „Aktivität entfernen“ einsetzen. Um dies zu gewährleisten muss jedoch im Fall „Entfernen aus einer Alternative“ eine Einschränkung getroffen werden, die im betroffenen Abschnitt noch näher erläutert wird. Die Parameter für `DeleteNode` sind in Tabelle 5.4 aufgelistet.

Parameter	**Bedeutung**
`processInstance`	Prozess Instanz, an der die Anpassung vorgenommen werden soll
`selectedNode`	Aktivität, die entfernt werden soll

Tabelle 5.4: Schnittstellenparameter für `deleteNode`

Entfernen von Einzelaktivitäten

Die einfachste Möglichkeit des Löschens einer Aktivität ist das Entfernen einer Einzelaktivität, Abbildung 5.13. Diese Aktivität darf nicht Teil einer strukturellen Konstruktion wie Sequenz oder Nebenläufigkeit sein. Eine Löschung einer solchen Aktivität wirft deshalb keine weiteren Fragen der Korrektheit auf und kann ohne zusätzliche Bedingungen ausgeführt werden.

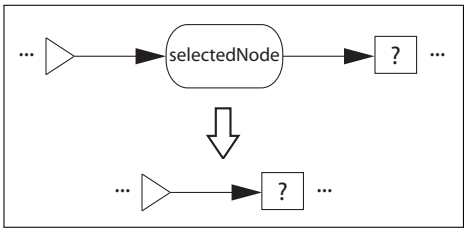

Abbildung 5.13: Entfernen einer Einzelaktivität

Entfernen aus Sequenz

Soll eine Aktivität aus einer Sequenz entfernt werden, muss unterschieden werden ob die Sequenz nach dem Löschen noch vorhanden ist oder nicht. Hierbei ist das Kriterium der Korrektheit für Sequenzen S1 entscheidend.

Eine Sequenz, die mehr als zwei Aktivitäten beinhaltet, wird auch nach der Adaption noch als Sequenz im Workflow vorhanden sein. Diese Operation wird Sequenzkürzung genannt und ist in Abbildung 5.14 zu erkennen. Die abgebildete Sequenz ($A \rightarrow B \rightarrow C$) wird in diesem Fall durch eine kürzere Sequenz ($A \rightarrow C$) ersetzt. Sie kann aber grundsätzlich als Sequenz weiterbestehen, da sie noch aus mindestens zwei Aktivitäten besteht.

Bei der Löschung aus einer minimalen Sequenz (die aus genau zwei Aktivitäten besteht), existiert nach der Adaption jedoch keine Sequenz mehr an der betroffenen Stelle. Die Operation wird deshalb als Sequenzlöschung (siehe Abbildung 5.15) bezeichnet.

Wird aus einer ursprüngliche Sequenz aus zwei Aktivitäten ($A \rightarrow B$) eine Aktivität (B) gelöscht, muss die zurückbleibende Einzelaktivität (A) die ehemalige Sequenz ersetzen. Die

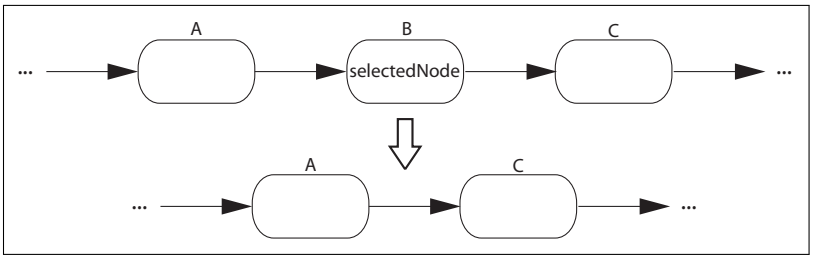

Abbildung 5.14: Sequenzkürzung

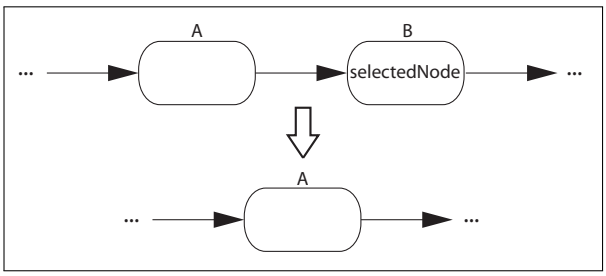

Abbildung 5.15: Sequenzlöschung

Unterscheidung dieses speziellen Anwendungsfalls ist deshalb relevant, weil das Kriterium der Korrektheit für Sequenzen S1 existiert. Dieses Kriterium besagt, dass eine Sequenz aus mindestens zwei Aktivitäten besteht, eine Einzelaktivität also keine Sequenz mehr ist. Eine Unterscheidung von Einzelaktivität und Sequenz ist auch dann wichtig, wenn eine Aktivität aus dem Zweig einer parallelen oder alternativen Verzweigung entfernt werden soll. Besteht der Zweig nur aus einer Einzelaktivität, so muss mit der Aktivität der gesamte Zweig gelöscht werden. Falls der Zweig jedoch aus einer Sequenz oder anderen strukturellen Konstruktionen besteht, so kann er weiterbestehen. (vgl. Prioritätskriterium P1)

Entfernen aus Nebenläufigkeit

Auch die Operation zum Entfernen einer Aktivität, die Teil einer Nebenläufigkeit ist, lässt sich in zwei weitere Anwendungsfälle unterteilen. Es muss entschieden werden, ob durch das Entfernen der Aktivität die Nebenläufigkeit in einen nicht korrekten Zustand übergehen könnte. Besitzt eine Nebenläufigkeit mindestens drei parallele Zweige, so kann die Aktivität auch dann entfernt werden, wenn dadurch der komplette Zweig gelöscht werden muss. Das Kriterium der Korrektheit für Nebenläufigkeiten N1 besagt, dass ein paralleler Ablauf aus mindestens zwei Zweigen bestehen muss — das Kriterium wird in einem solchen Fall nicht verletzt. Wie in Abbildung 5.16 zu erkennen ist, ist ursprünglich die parallele Ausführung der drei Aktivitäten $(A \wedge B \wedge C)$ geplant. Nach dem Löschen der Aktivität (C) können $(A \wedge B)$ weiterhin parallel ausgeführt werden. Die parallele Verzweigung kann also bestehen bleiben.

Problematisch wird es allerdings, wenn betroffene strukturelle Konstruktion eine „minimale Nebenläufigkeit“ ist. Eine Nebenläufigkeit ist dann minimal, wenn sie nur zwei parallele Zweige besitzt. Ein kompletter Zweig muss dann entfernt werden, wenn dieser nur aus einer Einzelaktivität besteht und genau diese Einzelaktivität gelöscht werden soll. Durch das Wegfallen des zweiten Zweiges, wird das Kriterium N1 verletzt. Eine parallele Bearbeitung ist nicht mehr möglich und die beschnittene Synchronisierung kann einen Deadlock verursachen. Falls durch das Entfernen einer Aktivität also ein kompletter Zweig gelöscht wird, muss sowohl die Parallele Verzweigung als auch die Synchronisierung der Nebenläufigkeit mit entfernt

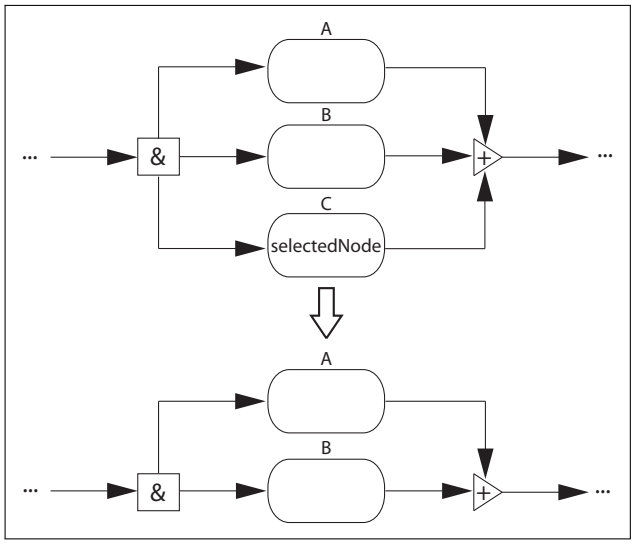

Abbildung 5.16: Entfernen aus Nebenläufigkeit mit ≥ 3 Zweigen

werden. In Abbildung 5.17 ist genau dieser Fall aufgezeigt. Die Aktivitäten $(A \wedge B)$ sollten

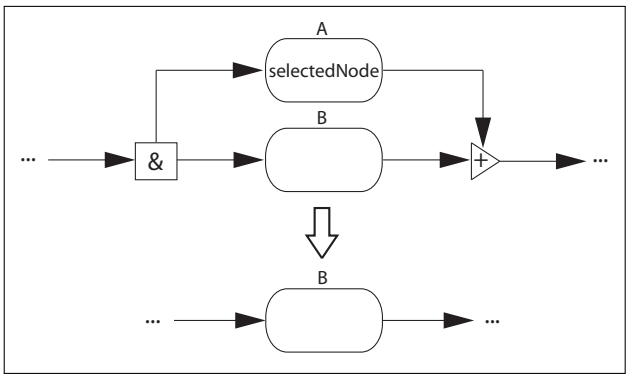

Abbildung 5.17: Entfernen aus minimaler Nebenläufigkeit

ursprünglich parallel zueinander ausgeführt werden. Durch das Löschen der Einzelaktivität (A) bleibt die Aktivität (B) als einziger verbleibender Zweig der Nebenläufigkeit bestehen. Aus diesem Grund müssen alle mit der Nebenläufigkeit verknüpften strukturellen Elemente entfernt werden. Die ehemalige Nebenläufigkeit $(A \wedge B)$ wird so durch die zurückbleibende Einzelaktivität (B) ersetzt.

Allerdings wird nicht jede minimale Nebenläufigkeit aufgelöst, wenn eine Aktivität gelöscht wird. Falls die Aktivität aus einem Zweig entfernt werden soll, der eine Sequenz beinhaltet, so greift laut Prioritätskriterium P1 zuerst das Muster „Entfernen aus Sequenz“ und nicht das Muster „Entfernen aus Nebenläufigkeit“. In Abbildung 5.18 ist dieser Fall zu erkennen. Hier würde nach der Anpassung immer noch eine parallele Bearbeitung von zwei Aktivitäten $(A \wedge B)$ stattfinden können, da die Aktivität (C) laut Prioritätskriterium aus der Sequenz $(B \rightarrow C)$ entfernt wurde.

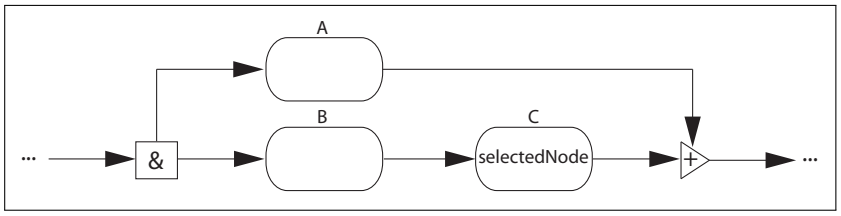

Abbildung 5.18: Entfernen aus minimaler Nebenläufigkeit II

Entfernen aus Alternative

Die Operation „Entfernen aus Alternative“ bietet ähnliche Herausforderungen. Auch hier gibt es wieder zwei grundsätzliche Anwendungsfälle, die das Entfernen von Aktivitäten aus alternativen Verzweigungen betreffen. Es muss ähnlich wie bei der Nebenläufigkeit entschieden werden, ob durch das Löschen einer Aktivität die strukturelle Korrektheit verletzt werden kann. Das Kriterium der Korrektheit für Alternativen A1 besagt, dass eine Alternative mindestens zwei alternative Zweige anbieten muss. Falls eine Alternative also drei Zweige besitzt, wird das Kriterium auch noch nach der Löschung eines dieser Zweige erfüllt. Ein Zweig wird dann gelöscht, wenn er nur aus einer Einzelaktivität besteht die entfernt werden soll. Dieser

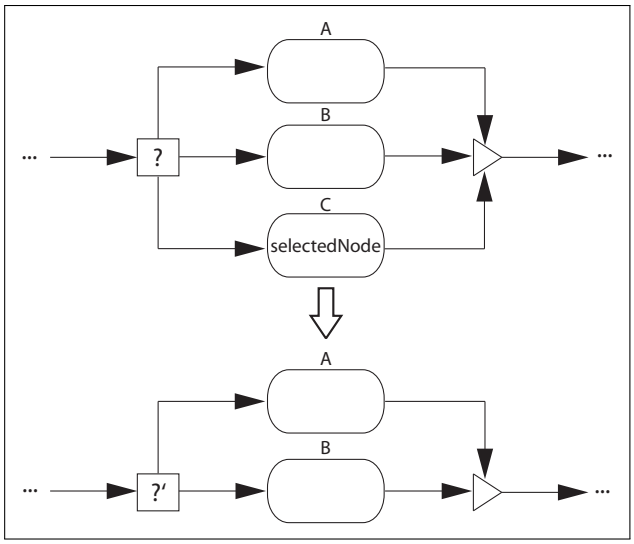

Abbildung 5.19: Entfernen aus Alternative mit ≥ 3 Zweigen

Vorgang ist in Abbildung 5.19 zu erkennen. Hier war die alternative Ausführung von drei Einzelaktivitäten ($A \vee B \vee C$) geplant. Nach der Löschung von Aktivität (C) ist weiterhin eine alternative Auswahl zwischen den zwei noch vorhandenen Aktivitäten ($A \vee B$) möglich. Die Korrektheit wird nicht verletzt. Allerdings ist hierbei zu beachten, dass sich die Entscheidungskriterien (?) in diesem Fall ändern. Eine Entscheidung für den ehemaligen Pfad der gelöschten Aktivität (C) darf nicht mehr getroffen werden, die entsprechenden Anweisungen müssen also im Entscheidungsknoten angepasst werden. Die neuen Entscheidungskriterien ($?'$) werden durch dieses Vorgehen automatisch bestimmt.

Dies ist die Einschränkung einer universellen Operation `deleteNode`. Es wäre denkbar, dass der ursprüngliche Fall für (C) auf einen der beiden anderen Zweige umgelenkt werden soll. Allerdings können durch die automatische Anpassung in `deleteNode` die Entscheidungskriterien nicht im gleichen Schritt angepasst werden. Um die Entscheidungskriterien dennoch

anpassen zu können, muss eine Hilfsoperation „Bedingungen anpassen“ angeboten werden. Die entsprechende Operation kann im Anschluss ausgeführt werden und so die gewünschte Funktionalität sicherstellen. Um die Operation durchführen zu können, muss eine Aktivität ausgewählt werden, die ein direkter Nachfolger der alternativen Verzweigung ist. Auf diesem Weg kann die Adaptionsoperation auf die relevante alternative Verzweigung zugreifen und diese anpassen. Die Schnittstellenbeschreibung der Operation `ChangeConditions` ist in Tabelle 5.5 aufgelistet.

Parameter	**Bedeutung**
`processInstance`	Prozess Instanz, an der die Anpassung vorgenommen werden soll
`selectedNode`	Direkte Nachfolgeaktivität der alternativen Verzweigung
`newConditions`	Neue Entscheidungskriterien

Tabelle 5.5: Schnittstellenparameter für `ChangeConditions`

Ebenso wie bei Nebenläufigkeiten gibt es auch bei Alternativen die minimale Form. Eine Alternative ist dann eine „minimale Alternative“, wenn sie aus genau zwei alternativen Zweigen besteht. Ein kompletter Zweig muss dann entfernt werden, wenn dieser nur aus einer Einzelaktivität besteht und genau diese Einzelaktivität gelöscht werden soll. Falls durch die Adaption so ein kompletter Zweig entfernt wird, wird das Kriterium A1 verletzt. Deshalb müssen in diesem Fall die strukturellen Elemente Alternative Verzweigung und Asynchrone Zusammenführung der Alternative komplett entfernt werden. Durch diesen Schritt werden auch etwaige Entscheidungskriterien mit entfernt. Abbildung 5.20 verdeutlicht diesen An-

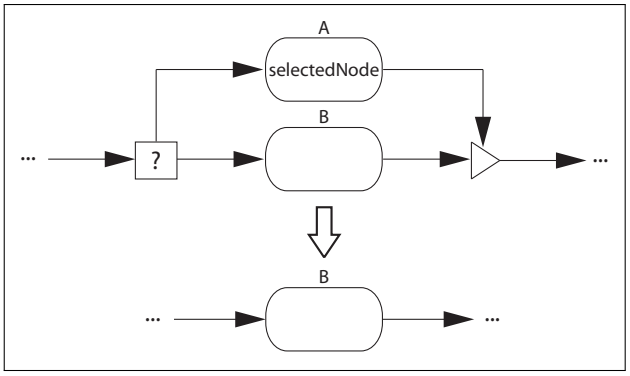

Abbildung 5.20: Entfernen aus minimaler Alternative

wendungsfall. Ursprünglich sollte eine der Aktivitäten $(A \vee B)$ alternativ ausgeführt werden. Durch das Löschen der Einzelaktivität (A) bleibt die Aktivität (B) als einziger Zweig der ehemaligen Alternative bestehen. Um die Korrektheit zu gewährleisten, werden die verknüpften strukturellen Elemente entfernt und die ehemalige Alternative durch die Einzelaktivität (B) ersetzt.

Ebenso wie bei der minimalen Nebenläufigkeit muss auch hier laut Prioritätskriterium P1 unterschieden werden, ob die entfernte Aktivität eines Zweiges der minimalen Alternative eine Einzelaktivität oder Teil einer Sequenz ist. Falls sie Teil einer Sequenz ist, wird laut Prioritätskriterium zuerst die Operation „Entfernen aus Sequenz“ eingesetzt und die Alternative bleibt bestehen. Der Aufbau ist analog zu dem in Abbildung 5.18 beschriebenen. Aus einer alternativen Bearbeitung von $[A \vee (B \rightarrow C)]$ wird durch das Entfernen von (C) aus einer Sequenz $(B \rightarrow C)$ die neue Alternative $(A \vee B)$.

5.3.3 Komposition von Operationen

Die im vorangegangen Teil definierten Muster bzw. Adaptionsoperationen sind die grundlegenden Anpassungsmöglichkeiten, aus denen sich weitere (komplexere) Muster zusammensetzen lassen. Diese Muster stehen auf einer nochmals höheren Stufe der Abstraktion, im Vergleich zu den bereits vorgestellten „high-level“ Einzel-Operationen „Aktivität hinzufügen/entfernen“ (die Bereits von den atomaren Elementar-Operationen abstrahieren). Die so beschriebenen Abstraktionsebenen von Workflow Adaptionen sind in Abbildung 5.21 zu sehen. Die kombinierten Operationen werden allerdings im relevanten Anwendungsfeld vorerst

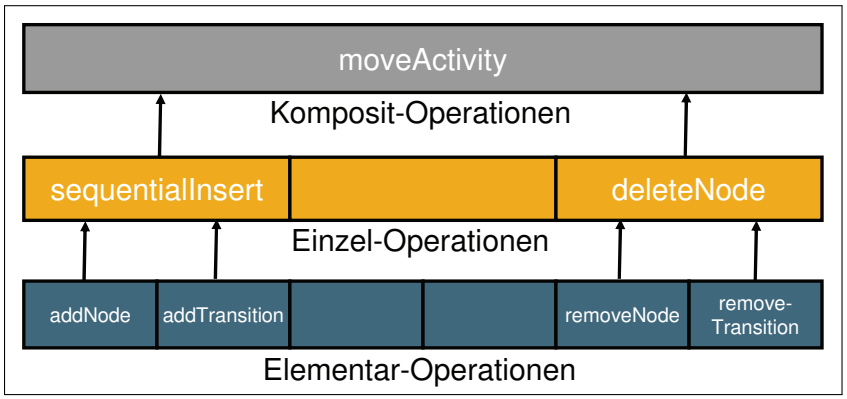

Abbildung 5.21: Abstraktionsebenen der Adaption

nicht zum Einsatz kommen. Die Aufgabe dieser Arbeit ist es, die grundlegenden Möglichkeiten zur Adaption zur Verfügung zu stellen. Deshalb werden die kombinierten Operationen nicht ausführlich behandelt oder implementiert, sondern im folgenden nur kurz vorgestellt. Grundsätzlich sind jedoch alle diese zusammengesetzten Operationen mit den zur Verfügung gestellten Einzel-Operationen realisierbar.

Aktivität verschieben

Muss eine Aktivität innerhalb des Workflows an eine andere Stelle verschoben werden, kommt das Muster „Aktivität verschieben“ zum Einsatz. Unter Verwendung der Einzel-Operationen „Aktivität hinzufügen“ und „Aktivität entfernen“ mit sämtlichen Mischformen von seriell, parallel oder alternativ kann die gewünschte Funktionalität angeboten werden. Das Verschieben einer Aktivität lässt sich grob in folgende Schritte aufteilen:

1. Aktivität auswählen
2. Aktivität entfernen (Zwischenspeicherung der Daten)
3. Neuen Ort der Aktivität auswählen
4. Aktivität am neuen Ort einfügen (unter Verwendung der alten Daten)

Aktivität ersetzen

Soll eine bestehende Aktivität im Workflow durch eine neue Aktivität ersetzt werden, kann das Muster „Aktivität ersetzen“ angeboten werden. In diesem Fall werden wieder die Einzel-Operationen „Aktivität hinzufügen“ und „Aktivität entfernen“ (mit ihren unterschiedlichen Anwendungsfällen) verknüpft. Als Ausnahme ist jedoch der Fall zu betrachten, wenn eine

Aktivität ersetzt werden soll, die Teil einer minimalen Verzweigung ist. In diesem Fall sollte die betroffene strukturelle Konstruktion nicht zuerst entfernt und anschließend wieder hinzugefügt werden. Stattdessen müssen die vorhandenen Elemente der Verzweigung weiter verwendet werden, um keine unnötigen Schritte ausführen zu müssen. Prinzipiell ist das Ersetzen einer Aktivität eine der reibungslosesten Adaptionen, da im Endeffekt nur die betroffenen Transitionen editiert werden müssen. Die Schritte, die hierfür nötig sind, lassen sich vereinfacht so definieren:

1. Alte Aktivität auswählen
2. Alte Aktivität entfernen
3. Neue Aktivität einfügen (unter Verwendung der alten strukturellen Konstruktion)

Aktivität tauschen

Sollen zwei Aktivitäten im Workflow miteinander vertauscht werden, sollte das Muster „Aktivität tauschen“ zum Einsatz kommen. In diesem Fall kann das oben beschriebene Muster „Aktivität verschieben“ wiederum als Grundlage dienen. Prinzipiell ist der Aktivitätentausch nichts anderes, als das Verschieben der betroffenen Aktivitäten an die jeweils entgegengesetzte Ursprungsposition. Ein Aktivitätentausch lässt sich grob in die folgenden Schritte gliedern:

1. Aktivität A auswählen
2. Aktivität B auswählen
3. Aktivitäten entfernen (Zwischenspeicherung der Daten)
4. Aktivitäten an neuen Orten einfügen (unter Verwendung der alten Daten)

Parallelisierung

Als Parallelisierung versteht man den Vorgang, wenn eine zuvor sequentiell ausgeführte Menge von Aktivitäten in Zukunft parallel zu einander stattfinden soll. In diesem Fall müssen die entsprechenden strukturellen Erweiterungen (parallele Verzweigung, Synchronisierung) hinzugefügt und die betroffenen Transitionen angepasst werden. Dabei kann eine abgeänderte Version der Operation für paralleles Einfügen zum Einsatz kommen. Das Muster für die „Parallelisierung“ muss allerdings relativ stark eingeschränkt werden. So ist es nicht möglich, jede beliebige Menge von Aktivitäten zu parallelisieren, ohne die Korrektheit des Schemas zu gefährden. Die Betrachtung dieser Einschränkungen kann jedoch nicht im Rahmen dieser Arbeit stattfinden. Vereinfacht findet eine Parallelisierung in den folgenden Schritten statt:

1. Sequenz von Aktivitäten auswählen
2. Strukturelle Erweiterungen hinzufügen
3. Transitionen anpassen

Umwandlung zu Alternative

Das Muster „Umwandlung zu Alternative“ ist vergleichbar mit der Parallelisierung. Allerdings soll hierbei die bestehende Sequenz nicht in eine parallele Verzweigung, sondern in eine Alternative überführt werden. Das Vorgehen entspricht weitestgehend dem der Parallelisierung — nur die hinzugefügten strukturellen Erweiterungen müssen in diesem Fall auf die Alternative angepasst sein. Die notwendigen Schritte einer Umwandlung zur Alternative sind mit den Schritten der Parallelisierung identisch.

Erweiterung auf Sub Prozesse

Grundsätzlich ist es denkbar, dass die Operationen zur Adaption nicht nur einzelne Aktivitäten, sondern ganze Sub Prozesse anpassen können. Ein abgeschlossener Bereich (Sub Workflow) innerhalb der Prozess Definition könnte also als „Aktivität“ im Sinne der beschriebenen Muster betrachtet werden. Die angebotenen Operationen könnten somit anstatt Aktivitäten komplette Sub Prozessen als Argumente entgegen nehmen. Sub Prozesse können in diesem Fall Teile von Verzweigungen darstellen oder auch Sequenzen bilden. Dabei muss jedoch sichergestellt werden, dass eine Adaptionsoperation nicht auf unterschiedlichen Ebenen eines Schemas agiert.

5.3.4 Ablauf der Adaption

Die beiden Aktivitätsdiagramme in Abbildungen 5.22 und 5.23 beschreiben grob den Ablauf einer Adaption, wie sie im Endsystem abläuft. Das Diagramm 5.22 beschreibt einen Überblick über den Gesamtablauf, während in Diagramm 5.23 der Inhalt der Schrittes „Anpassung vornehmen“ speziell bei einer Sequenzerweiterung dargestellt wird.

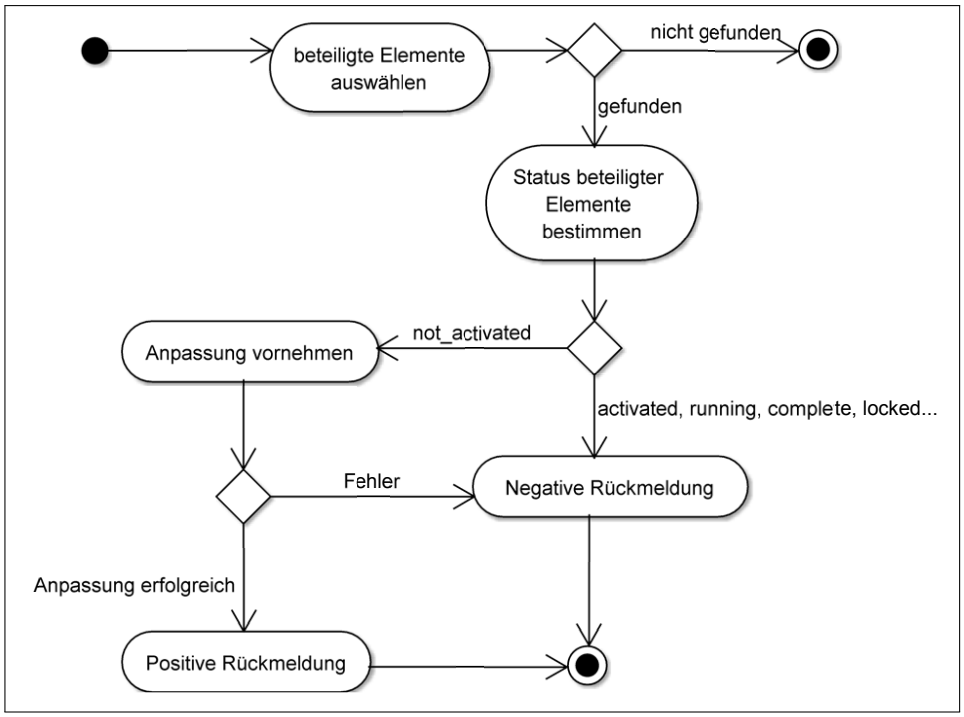

Abbildung 5.22: Aktivitätsdiagramm: Übersicht Adaption

5.4 Sicherstellung von Korrektheit und Konsistenz

Die in Abschnitt 5.2 definierten Kriterien der Korrektheit und Konsistenz dürfen durch die in Abschnitt 5.3 vorgestellten Operationen nicht verletzt werden. Wie bereits in Abschnitt 5.2 beschrieben, ist die gewählte Herangehensweise nicht die nachträgliche Prüfung der Korrektheit. Stattdessen sollen die angebotenen Adaptionsoperationen die Korrektheit „by Design“ gewährleisten. Das heißt, eine Operation kann nur ausgeführt werden, wenn sie Korrektheit

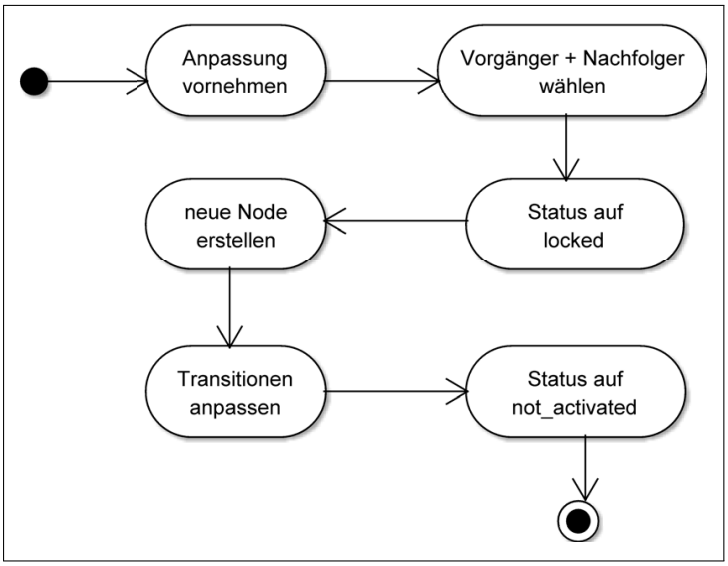

Abbildung 5.23: Aktivitätsdiagramm: Sequenzerweiterung

und die Konsistenz des Workflows nicht gefährdet. Aus diesem Grund müssen die Adaptionsoperationen so definiert werden, dass sie grundsätzlich ein korrektes Workflow Schema nur in ein wiederum korrektes Workflow Schema überführen können. Im folgenden Abschnitt wird mittels Pseudocode eine Formalisierung der benötigten Funktionalitäten vorgestellt.

Außerdem muss sichergestellt werden, dass die Konsistenz der Daten bei gleichzeitigen Zugriffen von Engine und Adaptionsdienst nicht gefährdet wird. In Abschnitt 5.2.3 wurde hierzu bereits der Ansatz geliefert, im Rahmen von Abschnitt 5.4.2 wird zudem erläutert wie der Bereich der Adaption festgelegt wird.

5.4.1 Korrektheit by Design

Bei der Beschreibung der Operationen in Abschnitt 5.3 wurde an den entsprechenden Stellen bereits darauf hingewiesen, wo etwaige Probleme mit der Korrektheit auftreten können. Nun stellt sich die Frage, wie genau die Korrektheit einer Adaptionsoperation überprüft und somit sichergestellt werden kann. Hierfür müssen die relevanten Informationen zum Aufbau des Workflow Schemas von der Workflow-Engine bereitgestellt und vom Adaptionsdienst verarbeitet werden. Die relevanten Informationen sind hier vor allem die Strukturdaten der Prozess Definition — also die interne Repräsentation im `ProcessDefinition` Objekt. Die Klasse `ProcessDefinition` bietet unterschiedliche Schnittstellen an, um den gewünschten Knoten (also die Aktivität) auszuwählen. Die Implementierung der Klasse `Node` bietet entsprechende Methoden an, die es ermöglichen innerhalb einer Prozess Definition zu „navigieren". Der eigentliche Adaptionsdienst benötigt trotzdem einige „Hilfsoperationen", um nicht auf die Elementar-Operationen zurückgreifen zu müssen. Auch können so definierte Hilfsoperationen in unterschiedlichen Adaptionsoperationen eingesetzt werden, um auf die Kriterien der Korrektheit zu prüfen.

Hilfsoperationen

Vorgänger oder Nachfolger eines bestimmten Knotens lassen sich beispielsweise ermitteln, indem die eingehenden bzw. abgehenden Transitionen untersucht werden. Listing 5.1 beschreibt

diesen Vorgang in Pseudocode mit den betroffenen jBPM Methoden. Natürlich können eindeutige Vorgänger oder Nachfolger nur von den Knoten bestimmt werden, die auch nur einen Vorgänger bzw. Nachfolger haben. Aus diesem Grund muss überprüft werden, ob der gewählte Knoten ein „normaler“ Aktivitäts-Knoten ist. Ein Knoten, der eine Verzweigung einleitet oder beendet und somit mehrere Vorgänger/Nachfolger besitzt, muss gesondert behandelt werden. Die Hilfsoperation liefert (falls möglich) die entsprechenden Vorgänger oder Nachfolger Knoten. Wenn der gewählte Knoten eine Verzweigung oder Zusammenführung mehrerer Pfade ist, wird der Knoten selbst wieder zurück gegeben.

```
1  public Node getPredecessor(String selectedNode){
2    Node nodeX = ProcessDefinition.findNode(selectedNode);
3    if (nodeX.getArrivingTransitions().size() = 1){
4     Transition t1 = nodeX.getArrivingTransitions().iterator().next();
5     Node nodeP = t1.getFrom();
6     return nodeP; //Rückgabe des Vorgängerknotens
7    }
8    else {
9     return nodeX; //Gewählter Knoten war Zusammenführung/Synchronisierung
10   }
11 }
12
13 public Node getSucessor(Node selectedNode){
14   Node nodeX = ProcessDefinition.findNode(selectedNode);
15   if (nodeX.getLeavingTransitions().size() = 1){
16    Transition t1 = nodeX.getDefaultLeavingTransition();
17    Node nodeS = t1.getTo();
18    return nodeS; //Rückgabe des Nachfolgeknotens
19   }
20   else {
21    return nodeX; //Gewählter Knoten war Verzweigung
22   }
23 }
```

Listing 5.1: Ermittlung von Vorgänger und Nachfolger mit jBPM

Im Listing ist ebenfalls zu erkennen, dass die Klasse `Transition` die benötigten Methoden anbietet, um die mit der Transition verknüpften Knoten abzufragen. Das Vorhandensein dieser Methoden ist Grundlage für eine erfolgreiche Sicherstellung der Korrektheit einer Adaptionsoperation.

Um die Kriterien der Korrektheit grundsätzlich in der strukturellen Konstruktion einordnen zu können, muss ein Unterschied zwischen Einzelaktivität und Sequenz erkennbar sein. Wie in Kriterium für strukturelle Korrektheit S1 beschrieben, ist es vor allem für das Prioritätskriterium P1 wichtig, um diese Unterscheidung treffen zu können. Kriterium S1 besagt, dass eine Sequenz aus mindestens zwei Aktivitäten besteht. Aus diesem Grund wird eine Hilfsoperation (siehe Listing 5.2) definiert, die Sequenzen von Einzelaktivitäten abgrenzen kann.

Außerdem ist es notwendig eine Entscheidung darüber zu treffen, ob eine gewählte Aktivität Teil einer (parallelen oder alternativen) Verzweigung ist. In diesem Fall muss überprüft werden, ob auf dem betroffenen Pfad der Ausführung eine Verzweigung geöffnet worden ist, die noch nicht geschlossen wurde. Es muss jedoch beachtet werden, dass jBPM keine speziellen Knotenart für Asynchrone Zusammenführungen bereit stellt. In der Syntax von jBPM sind asynchrone Verzweigungen in der Art abzuschließen, indem sich alle von einer `DecisionNode` abgehenden Pfade in einem später folgenden (normalen) Knoten wieder treffen. Es existiert

```
public boolean isPartOfSequence(String selectedNode){
  Node nodeX = ProcessDefinition.findNode(selectedNode);
  while (pred(nodeX).getClass() == Node|State|TaskNode) {
    sequenz[] = [pred(nodeX), nodeX,...];
    nodeX = pred(nodeX);
  }
  while (succ(nodeX).getClass() == Node|State|TaskNode) {
    sequenz[] = [...,nodeX,succ(nodeX)];
    nodeX = succ(nodeX);
  }
  if (sequenz[].length > 0) {
    return true; //Knoten ist Teil einer Sequenz
  }
  else {
    return false; //Knoten ist Einzelaktivität
  }
}
```

Listing 5.2: Bestimmung von Sequenzen mit jBPM

zwar eine Knotenart `Merge`, diese kommt allerdings nicht zum Einsatz, da jeder Knoten implizit ein `Merge` Knoten sein kann, falls er mehrere eingehende Transitionen besitzt. Somit wird als Asynchrone Zusammenführung (OR-Join) jeder Knoten bezeichnet, der mehr als eine eingehende Transition besitzt und gleichzeitig kein `Join` Knoten — also keine Synchronisierung — ist. Aus diesem Grund wird eine Hilfsoperation (Listing 5.3) benötigt, die einen OR-Join Knoten identifizieren kann.

```
public boolean isOrJoin(String selectedNode){
  Node nodeX = ProcessDefinition.findNode(selectedNode);
  if (nodeX.getClass() != Join &&
      nodeX.getArrivingTransitions().size() > 1){
   return true; //Knoten ist async. Zusammenführung
  }
  else {
   return false; //Knoten ist keine async. Zusammenführung
  }
}
```

Listing 5.3: Bestimmung von asynchronen Zusammenführungen mit jBPM

Unter Verwendung der bisher definierten Hilfsoperationen kann eine weitere Hilfsoperation entwickelt werden, die die Entscheidung darüber trifft ob ein gewählter Knoten Teil einer Verzweigung ist. Die Operation durchläuft alle Knoten ab der gewählten Aktivität rückwärts bis zum Start. Hierfür muss eine angepasste Hilfsoperation zur Vorgängerfindung zum Einsatz kommen, die auch einen Vorgänger bei Synchronisierungen/Zusammenführungen zurück liefert. Falls auf diesem Weg mehr Verzweigungen geöffnet als geschlossen wurden, so ist der Knoten selbst Teil einer Verzweigung.

Weiterhin sollte festgestellt werden können, ob eine von der Adaption betroffene Verzweigung eine *minimale Verzweigung* ist, da dies für die einige Kriterien der Korrektheit (vgl. Abschnitt 5.2) ausschlaggebend ist. Um eine Verzweigung umfassend untersuchen zu können, ist die Information über vorhandene Verzweigungen und Synchronisierungen natürlich unabdingbar. Die unter Verwendung der bereits vorgestellten Hilfsoperationen definierte Operation zur Überprüfung auf „minimale Verzweigung“ ist in Listing 5.5 zu erkennen.

```
1  public boolean isPartOfSplit(String selectedNode){
2    Node nodeX = ProcessDefinition.findNode(selectedNode);
3    int openedSplits = 0;
4    int closedSplits = 0;
5
6    while (nodeX.getClass() != StartState){
7     if (nodeX.getClass() == Fork ||
8         nodeX.getClass() == Decision){
9      openedSplits++;
10    }
11    if (isOrJoin(nodeX) == true ||
12        nodeX.getClass() == Join){
13     closedSplits++;
14    }
15    nodeX = pred(nodeX);
16   }
17
18   if (openedSplits > closedSplits){
19    return true; //Knoten ist Teil einer Verzweigung
20   }
21   else {
22    return false; //Knoten ist nicht Teil einer Verzweigung
23   }
24 }
```

Listing 5.4: Prüfung auf Verzweigung mit jBPM

```
1  public boolean isMinimalSplit(String selectedNode){
2    Node nodeX = ProcessDefinition.findNode(selectedNode);
3
4    while (pred(nodeX).getClass() != Fork ||
5           pred(nodeX).getClass() != Decision){
6     nodeX = pred(nodeX);
7    }
8    if (pred(nodeX).getClass() == Fork ||
9        pred(nodeX).getClass() == Decision){
10    Node splitNode = pred(nodeX);
11   }
12   else {
13    return false; //Fehlerhafte ProcessDefinition
14   }
15   if (splitNode.getLeavingTransitions().size() > 2){
16    return false; //Keine minimale Verzweigung
17   }
18   else if (splitNode.getLeavingTransitions().size() = 2){
19    return true; //Minimale Verzweigung
20   }
21   else {
22    return false; //Fehlerhafte ProcessDefinition
23   }
24 }
```

Listing 5.5: Test auf minimale Verzweigungen mit jBPM

Umsetzung

Die eigentlichen Operationen des Adaptionsdienstes — die Anpassungen an der Prozess Instanz vornehmen sollen — greifen auf die oben vorgestellten Hilfsoperationen zu, um in der Prozess Definition zu navigieren und die nötigen Unterscheidungen zwischen Einzelaktivitäten/Sequenzen/Verzweigungen treffen zu können.

Eine Sequenzerweiterung oder Sequenzerzeugung benötigt die Hilfsoperationen zur Bestimmung des Nachfolger- oder Vorgänger-Knotens. Die Vorgehensweise ist in Listing 5.6 dargestellt. Hier sind unter anderem die Elementar-Operationen zu erkennen, die zum Erreichen der gewünschten Funktionalität beitragen.

```
1  public void sequentialInsert(String newNodeName, Node selectedNode, int
       pointOfInsert){
2   Node nodeX = ProcessDefinition.findNode(selectedNode);
3   Node predNode = pred(nodeX);
4   Node succNode = succ(nodeX);
5   Node newNode = new Node(newNodeName);
6
7   if (pointOfInsert == 0){ //Einfügen vor selectedNode
8    Transition t1 = predNode.getDefaultLeavingTransition();
9    Transition t2 = new Transition();
10   t1.setTo(newNode);
11   newNode.addLeavingTransition(t2);
12   nodeX.addArrivingTransition(t2);
13  }
14  else {                    //Einfügen nach selectedNode
15   Transition t1 = nodeX.getDefaultLeavingTransition();
16   Transition t2 = new Transition();
17   t1.setFrom(newNode);
18   nodeX.addLeavingTransition(t2);
19   newNode.addArrivingTransition(t2);
20  }
21 }
```

Listing 5.6: Vorgehensweise bei `sequentialInsert`

Die Operationen `parallelInsert` und `alternateInsert` werden jeweils in zwei Anwendungsfälle unterteilt. Entweder wird an einer Sequenz oder an einer bereits bestehenden Verzweigung eingefügt. Exemplarisch wird in Listing 5.7 die Vorgehensweise der Operation `parallelInsert` vorgestellt.

Die Operation `alternateInsert` läuft prinzipiell analog ab. Allerdings kommt hier das Kriterium für erweiterte strukturelle Korrektheit K1 zum Tragen. Aus diesem Grund muss innerhalb der Operation sichergestellt werden, dass keine Verzweigungselemente auf unterschiedlichen Pfaden der Ausführung platziert werden. In Listing 5.8 wird dargestellt, wie dieser Teil der Operation mittels Elementar-Operationen und Hilfsoperationen realisiert wird. Als Asynchrone Zusammenführung wird mangels jBPM Repräsentation ein normaler Knoten eingefügt, der die eingehenden Transitionen bündelt. Das Listing beschreibt nur einen Ausschnitt der Gesamtoperation für den Fall, dass die `beforeNode` nicht auf der selben parallelen Verzweigung liegt, auf der die `afterNode` liegt. Der umgekehrte Fall muss ebenso behandelt werden.

Die Operation `deleteNode` kommt mit den oben vorgestellten Hilfsoperationen aus und setzt entsprechende Elementar-Operationen ein um gewählte Aktivitäten inklusive der betroffenen

```
1  public void parallelInsert(String newNodeName, String beforeNode,
       String afterNode){
2   Node nodeX = ProcessDefinition.findNode(beforeNode);
3   Node nodeY = ProcessDefinition.findNode(afterNode);
4   Node predNode = pred(nodeX);
5   Node succNode = succ(nodeY);
6   Node newNode = new Node(newNodeName);
7
8   if (predNode.getClass() == Fork &&
9       succNode.getClass() == Join){ //Verzweigung vorhanden
10   Transition t1 = new Transition();
11   Transition t2 = new Transition();
12   predNode.addLeavingTransition(t1);
13   newNode.addArrivingTransition(t1);
14   newNode.addLeavingTransition(t2);
15   succNode.addArrivingTransition(t2);
16  }
17  else { //Verzweigung muss erstellt werden
18   Fork newFork = new Fork();
19   Join newJoin = new Join();
20   predNode.getDefaultLeavingTransition().setTo(newFork);
21   succNode.getArrivingTransition().setFrom(newJoin);
22   Transition t1 = new Transition();
23   Transition t2 = new Transition();
24   newFork.addLeavingTransition(t1);
25   newFork.addLeavingTransition(t2);
26   nodeX.addArrivingTransition(t1);
27   newNode.addArrivingTransition(t2);
28   ... //Rest analog mit Join verknüpft
29  }
30 }
```

Listing 5.7: Vorgehensweise bei `parallelInsert`

Transitionen zu entfernen bzw. diese anzupassen. Einen Sonderfall bilden hier jedoch die minimalen Verzweigungen. Falls eine Aktivität aus einer minimalen Verzweigung entfernt wird, die als Einzelaktivität einen Zweig der Verzweigung dargestellt hat, muss laut den Kriterien A1 und N1 die entsprechende strukturelle Konstruktion entfernt werden. Listing 5.9 gibt eine Übersicht über das Vorgehen in diesem Fall. Der Pseudocode mit jBPM Methoden stellt einen Ausschnitt aus der Operation `deleteNode` dar. Hier sind bereits teilweise die Elementar-Operationen mit aufgeführt, die zur Durchführung der Gesamtoperation nötig sind. Die vereinfacht dargestellte Operation `remove(Node)` besteht in dem Fall aus mehreren Elementar-Operationen zum Entfernen des Knotens und der beteiligten Transitionen, die nicht weiter ausgeführt werden.

```
1  public void alternateInsert(String newNodeName, String beforeNode,
       String afterNode, DecisionCondition newConditions){
2   ...
3   else { //Verzweigung muss erstellt werden
4    if (isPartOfAndSplit(beforeNode) == false &&
5        isPartOfAndSplit(afterNode) == true){
6     Decision newDecision = new Decision();
7     Node newMerge = new Node();
8     ... //Decision vor beforeNode einfügen
9     Node activeNode = succ(afterNode);
10    while (activeNode.getClass() != Join) {
11     activeNode = succ(activeNode);
12    }
13    ...//activeNode ist jetzt der Join Knoten
14    activeNode.getDefaultLeavingTransition().setTo(newMerge);
15    newMerge.addArrivingTransition(t1);
16    t1.setFrom(newNode);
17    newDecision.addLeavingTransition(t2);
18    t2.setTo(newNode);
19    ...//bestehende Transitionen anpassen
20   }
21   ...
22  }
23 }
```

Listing 5.8: Vorgehensweise bei `alternateInsert`

```
1  public void deleteNode(String selectedNode){
2   Node nodeX = ProcessDefinition.findNode(selectedNode);
3   ...
4   if (isPartOfSequence(nodeX) == false){
5    if (isMinimalSplit(nodeX) == true){
6     Node splitNode = pred(nodeX);
7     Node joinNode = succ(nodeX);
8     Node preSplit = pred(splitNode);
9     Node pastJoin = succ(joinNode);
10
11    remove(nodeX);
12
13    node beforeNode = succ(splitNode);
14    Node afterNode = pred(joinNode);
15
16    Transition t1 = beforeNode.getArrivingTransition();
17    Transition t2 = afterNode.getDefaultLeavingTransition();
18
19    t1.setFrom(preSplit);
20    t2.setTo(pastJoin);
21
22    remove(splitNode, joinNode);
23   }
24   ...
25  }
26  ...
27 }
```

Listing 5.9: Auflösung einer minimalen Verzweigungen mit jBPM

5.4.2 Konsistenz und Bereich der Adaption

In Abschnitt 5.1 wurde bereits detailliert darauf eingegangen, wie die Konsistenzsicherung durch die Implementierung der Statusinformationen realisiert werden kann. Im folgenden wird daher nur noch kurz näher erläutert, wie diese Konzepte mittels der bereitgestellten Operationen umgesetzt werden. Dazu gehört auch die Vorgehensweise bei der Festlegung des „Bereichs der Adaption“ (vgl. Abschnitt 5.2.3).

Um den Status einer Aktivität zu setzen, muss lediglich das entsprechende `ElementState` Objekt mit der enthaltenen Variable `stateType` angepasst werden. Hierfür wurden die relevanten Knotentypen angepasst und um Methoden zum Setzen bzw. Abfragen des Status erweitert. In Listing 5.10 wird die Vorgehensweise mittels Pseudocode und jBPM Methoden verdeutlicht.

```
1  public String getNodeState(String selectedNode){
2   Node nodeX = ProcessDefinition.findNode(selectedNode);
3   String stateType = nodeX.getState().getStateType();
4   return stateType;
5  }
6
7  public void setNodeState(String selectedNode, String stateType){
8   Node nodeX = ProcessDefinition.findNode(selectedNode);
9   nodeX.getState().setStateType(stateType);
10 }
```

Listing 5.10: Abfrage und Setzen von Status mit der jBPM Erweiterung

```
1  public void lockAdaptionRange(String selectedNode){
2   Node nodeX = ProcessDefinition.findNode(selectedNode);
3   setNodeState(nodeX, "locked");
4   if (isPartOfAndSplit(nodeX)) {
5    Node activeNode = nodeX;
6    while (pred(activeNode).getClass() != Fork) {
7     activeNode = pred(activeNode);
8     setNodeState(activeNode, "locked");
9    }
10   Node activeNode = nodeX;
11   while (succ(activeNode).getClass() != Join) {
12    activeNode = succ(activeNode);
13    setNodeState(activeNode, "locked");
14   }
15  }
16  else if (isPartOfOrSplit(nodeX)) {
17  ... //Vorgehensweise analog
18  }
19  else {
20   setNodeState(pred(nodeX), "locked");
21   setNodeState(succ(nodeX), "locked");
22  }
23 }
```

Listing 5.11: Definition des Bereichs der Adaption

In den Adaptionsoperationen werden entsprechende Abfragen mittels `getNodeState()` eingebaut, um die in Abschnitt 5.1 beschriebenen Konsistenzkriterien zu gewährleisten. Liefert die Abfrage einen Status zurück, der nicht „not_activated“ entspricht, so kann die gewünschte

Adaptionsoperation nicht ausgeführt werden und wird abgebrochen. Ähnlich ist der Ablauf beim Setzen des Status einer Aktivität. Der Status muss dann von einer Adaptionsoperation verändert werden, wenn die Aktivität von einer Anpassung direkt oder indirekt betroffen ist. Indirekt ist sie betroffen, wenn sie im Bereich der Adaption liegt. Wie bereits beschrieben, muss im Falle einer Adaption dieser Bereich auf den Status „locked“ gesetzt werden, damit die Ausführung des Workflow nicht in den Bereich vordringt. Hier stehen sich zum Einen die Workflow-Engine — die den Status der Aktivitäten während der Ausführung anpasst und überprüft — und zum anderen den Adaptionsdienst — der den Status vor einer Anpassung überprüft und den Bereich der Adaption abschirmt — gegenüber. Die beiden Komponenten sind somit auf das korrekte Setzen des Status durch die jeweils andere Komponente angewiesen.

In Listing 5.11 wird dargestellt, wie der Bereich der Adaption vom Adaptionsdienst festgelegt und der Status auf „locked“ gesetzt wird. Nach der Adaption wird der Status des Bereichs in einem analogen Schritt wieder auf „not_activated“ zurück gesetzt. Durch die Anpassung der Workflow-Engine in der Hinsicht, dass keine Aktivitäten im Status „locked“ ausgeführt werden dürfen, wird die Zugriffskonsistenz gewährleistet.

5.5 Architektur

Die Architektur des Adaptionsdienstes im Gesamtsystem ist in Abbildung 5.24 dargestellt. Grundsätzlich wird der Adaptionsdienst als eigentständige Komponente in jBPM integriert in wird somit Teil des Gesamtsystems. Der Adaptionsdienst bildet somit die Schnittstelle zwischen Workflow-Engine und dem Administrator, der die Adaptionen am Workflow während der Laufzeit vornimmt. In Abbildung 5.24 ist ebenfalls zu erkennen, dass die Workflow-Engine bereits Schnittstellen anbietet, über die zur Designzeit auf die ProcessDefinition und zur Laufzeit auf die ProcessInstance von den jeweils relevanten Personen zugegriffen werden kann. Ein Entwickler entwirft zur Designzeit eine Prozess Definition, oder nimmt Änderungen an einer bereits vorhandenen vor. Die Bearbeiter greifen indirekt auf die Prozessinstanz zur Laufzeit zu, indem sie die gestellten Aufgaben bearbeiten. Die Fortschritte im Workflow werden so in der Prozessinstanz repräsentiert.

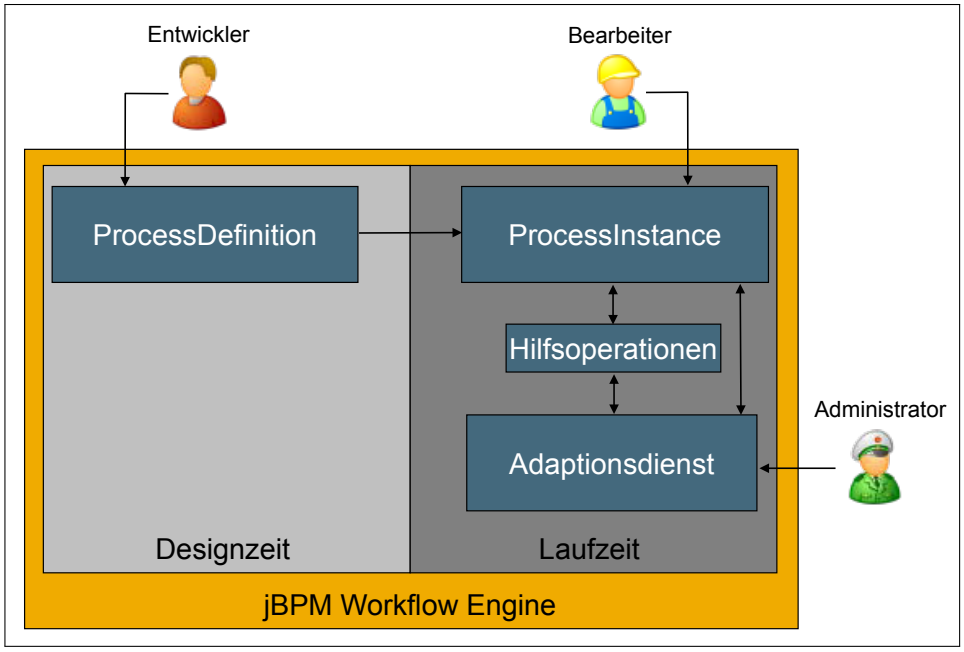

Abbildung 5.24: Architektur Adaptionsdienst

Neu ist der Administrator, der zur Laufzeit auf den Adaptionsdienst zugreift. Der Adaptionsdienst besitzt wiederum eine Schnittstelle zu den beschriebenen Hilfsoperationen, die eine Schnittstelle zur ProcessInstance besitzen und so die zugrunde liegende ProcessDefinition editieren können. Außerdem besitzt der Adaptionsdienst eine direkte Schnittstelle zur ProcessInstance, da nicht alle Anfragen über die Hilfsoperationen abgedeckt werden. So wird es ermöglicht, den Workflow während der Laufzeit anzupassen. Die Komponente des Adaptionsdienstes kapselt die in 5.3 beschriebenen Operationen wie `alternateInsert` oder `deleteNode`, während die Komponente der Hilfsoperationen die entsprechenden Methoden wie `getPredecessor` anbietet.

6 Implementierung und Validierung

In diesem Kapitel werden die Vorgehensweise bei der prototypischen Implementierung der vorgestellten Konzepte erläutert sowie die zum Einsatz gekommenen Anwendungen, Bibliotheken und Programmiersprachen vorgestellt. Im Anschluss erfolgt eine Validierung und Bewertung der Konzepte und des entstandenen Prototyps.

6.1 Implementierung

Die Implementierung des Prototyps fand komplett unter der Entwicklungsumgebung Eclipse Ganymede (Version 3.4.2) in Java statt. Die Version 3.1 der jBPM-Workflow Engine diente als Grundlage für die Erweiterung. Neben den im verwendeten jBPM Paket mitgelieferten Bibliotheken kam das Java J2SE Development Kit 5.0 (JDK 1.5) zum Einsatz. Zusätzlich wurde das JUnit Framework (Version 3.8.1) als Grundlage für die Validierung und zum Testen der Implementierung eingesetzt.

Die Eclipse Umgebung wurde durch das Plugin „JBoss jBPM Designer“ in Version 3.1.3 SP2 erweitert, womit die jPDL Prozess Definitionen erstellt und editiert werden können. Außerdem kamen unterschiedliche Anwendungen zur Versionsverwaltung zum Einsatz, zum einen das Eclipse Plugin Subclipse, zum anderen das Standalone Tool TortoiseSVN.

Die UML Diagramme (Klassendiagramme, Aktivitätsdiagramme, Zustandsdiagramme) wurden mit der Anwendung ArgoUML erstellt.

6.2 Validierung und Bewertung

Die Validierung der entworfenen Konzepte und des implementierten Prototyps wurde mit JUnit Testfällen realisiert. Hierfür wurden entsprechende Prozess Definitionen im jPDL-XML-Format erstellt, die die nötige Komplexität für die betroffenen Testfälle aufweisen. Beispielhaft ist in Listing 6.1 die Vorgehensweise im jUnit Testfall zu erkennen.

Durch dieses Konzept ist es möglich die jBPM Workflow-Engine ohne das vorherige Deployment auf einem Application Server zu testen und somit auch die neu implementierten Adaptionsoperationen zu validieren. Die Funktionsweise hierbei ist, dass in einem jUnit Testfall bestimmte Annahmen getroffen werden, die im Laufe des Tests verifiziert werden. So kann z.B. die Annahme getroffen werden, dass sich der Token nach einer Anpassung in einem bestimmten Knoten befindet. Diese Annahme wird durch die jUnit Methode `assertSame` überprüft. Falls die Annahme korrekt war, gelingt der Test und die Ausgabe ist „grün“. Falls die Annahme nicht korrekt war, weil z.B. während der Adaption ein Fehler aufgetreten ist, misslingt der Test und die Ausgabe ist „rot“. Durch die Durchführung mehrerer — auf spezielle Aspekte des Systems zugeschnittener — Tests können so die Funktionen des Adaptionsdienstes nach und nach überprüft werden.

```
ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(
 "<process-definition>" +
 "<start-state name='start-state'>" +
 "      <transition to='state1' />" +
 "</start-state>" +
 "<state name='state1'>" +
 "      <transition to='state2' />" +
 "</state>" +
 "<fork name='fork1'>" +
 "      <transition to='state3' name='to state3' />" +
 "      <transition to='state4' name='to state4' />" +
 "</fork>" +
 "<state name='state2'>" +
 "      <transition to='fork1' />" +
 "</state>" +
 "<state name='state3'>" +
 "      <transition to='join1' />" +
 "</state>" +
 "<state name='state4'>" +
 "      <transition to='join1' />" +
 "</state>" +
 "<join name='join1'>" +
 "      <transition to='state5' />" +
 "</join>" +
 "<state name='state5'>" +
 "      <transition to='end-state' />" +
 "</state>" +
 "<end-state name='end' />" +
 "</process-definition>"
);
ProcessInstance processInstance = new
    ProcessInstance(processDefinition);
```

Listing 6.1: Parsen einer XML ProcessDefinition für jUnit Testfälle

Grundsätzlich ist jedoch zu erwähnen, dass die entworfenen Operationen nur in dieser abgeschlossenen Testumgebung überprüft wurden. Der Einsatz auf einem laufenden jBPM System war durch den zu großen damit verbundenen Aufwand nicht möglich. Aus diesem Grund haben die implementierten Operationen auch nur einen rein prototypischen Charakter. Soweit die Überprüfung der Anwendungsfälle mittels jUnit möglich war, wurde dies durchgeführt. Allerdings konnten auf diesem Wege z.B. die Aspekte der Persistenz (Stichwort Hibernate) nicht überprüft werden. Es ist damit zu rechnen, dass in einer Umgebung die auf die Persistenz mittels Hibernate und entsprechenden Hibernate-Mappings setzt, die Operationen noch eine umgehenden Anpassung benötigen. Jegliche Aspekte der Persistenz wurden beim Entwurf und der prototypischen Implementiertung nicht berücksichtigt.

Außerdem ist damit zu rechnen, dass bei stark verschachtelten Prozess Definitionen unvorhergesehene Fehler auftreten können. Die prototypische Implementierung der Operationen zur Navigation in der ProcessDefinition ist nicht soweit ausgereift, dass sie in einer beliebig komplexen Defintion weiterhin mit Sicherheit das korrekte Ergebnis liefert. Hier müssten bei der Übertragung in ein lauffähiges System noch weitere Überprüfungen und gegebenenfalls Anpassungen stattfinden.

Im allgemeinen ist das Ergebnis der Arbeit die Darlegung der grundsätzlichen Realisierbarkeit von ad-hoc Adaptionen in der jBPM Workflow-Engine. Zusätzlich wurden die Operationen so entworfen, dass die Korrektheit des Workflow Schemas durch eine Adaption nicht gefährdet

werden kann. Auch die Konsistenz der Daten wurde durch die Erweiterung der Engine um Statusrepräsenationen gewährleistet.

Der Zugriff auf die Operationen findet im aktuellen Zustand nur direkt über eine Programmierschnittstelle statt. Eine entsprechende Benutzeroberfläche wurde nicht implementiert. Jedoch sollte dies keine größeren Probleme aufwerfen. Diese Schnittstelle müsste in die Benutzeroberfläche integriert werden, die die jBPM Workflow Engine visualisiert. Mitgeliefert wird von jBPM lediglich ein Web-Interface — es ist allerdings möglich, eigene Benutzeroberflächen für jBPM zu entwerfen. Bei dieser Implementierung muss eine entsprechende Interaktion mit dem Adaptionsdienst vorgesehen werden.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Workflow Management Systeme werden zunehmend in Bereichen eingesetzt, die sich vom eigentlichen Einsatzgebiet — den Geschäftsprozessen in Unternehmen — stark unterscheiden. Aus diesem Grund ist es wichtig, dass die Systeme auf neue Anforderungen in diesen Bereichen hin untersucht und angepasst werden. Eine wichtige Anforderung ist die Flexibilität eines Workflows. Flexibilität ist natürlich auch im ursprünglichen Einsatzgebiet von Vorteil. In den neuen Bereichen wie z.B. dem Katastrophenmanagement oder der Krankenversorgung ist sie jedoch unabdingbar und Grundlage für den Einsatz solcher Systeme.

Im Rahmen dieser Arbeit wurde eine spezifische Workflow-Engine um die benötigten Funktionalitäten für den Einsatz im Katastrophenmanagement erweitert. Diese Funktionalitäten werden allgemein bei der Anpassung von Workflows während der Laufzeit benötigt. Speziell für die — im relevanten Anwendungsfeld eingesetzte — Workflow-Engine jBPM wurde eine Architektur entworfen, die die erforderlichen Funktionen bereit stellt. Dabei wurde insbesondere darauf geachtet, dass die zum Einsatz kommenden Operationen die Korrektheit und Konsistenz des Workflows zu keiner Zeit gefährden.

Dies war auch die große Herausforderung des Entwurfs. Es musste eine Möglichkeit gefunden werden, wie die Korrektheit und Konsistenz sichergestellt werden kann. Der hierbei entstandene Ansatz lässt sich mit dem Begriff „Korrektheit by Design“ umschreiben — die Adaptionsoperationen wurden so entworfen, dass sie ein korrektes Workflow Schema nur in ein wiederum korrektes Workflow Schema überführen können. Durch die Beschränkung der angebotenen Operationen auf vorher definierte Anpassungs-Muster konnte diese Anforderung umgesetzt werden. Aus diesem Grund muss die Korrektheit eines Workflows nach der Adaption nicht mühsam überprüft werden, sondern die Korrektheit wird durch die Adaption an sich gewährleistet. Ähnlich verhält es sich mit der erzielten Umsetzung der Status-Informationen, welche zur Gewährleistung der Konsistenz beitragen. Die entwickelten Konzepte wurden in einer prototypische Implementierung umgesetzt und (soweit möglich) validiert.

7.2 Ausblick

Der Entwurf der Operationen und die protoypische Implementierung des Adaptionsdienstes zeigt, dass die grundsätzliche Anpassungsfähigkeit eines jBPM Workflows zur Laufzeit erreichbar ist. Um das Potential weiter ausnutzen zu können, wird natürlich eine entsprechende (graphische) Benutzerschnittstelle zum Adaptionsdienst benötigt. Eine solche Benutzerschnittstelle sollte im Rahmen der Gesamtimplementierung der Workflow-Engine eingearbeitet werden.

Der eigentliche Adaptionsdienst bietet ebenfalls noch weitere Ansatzpunkte zur weiteren Entwicklung. So wären die in Abschnitt 5.3.3 angedeuteten Komposit-Operationen umzusetzen,

indem die bereits vorhandenen Operationen angepasst und kombiniert werden. Ein zusätzlicher Ansatzpunkt für weitere Betrachtungen ist der Bereich der Persistenz. Es ist davon auszugehen, dass eingesetzte Caching-Mechanismen zu Problemen beim schreibenden Zugriff auf die ProcessInstance führen. In diesem Zusammenhang sollte auch darauf eingegangen werden, inwiefern adaptierte Prozess Definitionen möglicherweise wieder zurück in ein XML Format transformiert werden können, um eine Versionierung oder Migration von Prozess Definitionen zu ermöglichen (Stichwort: Workflow Evolution).

Literaturverzeichnis

[1] Aalst, W. M. P. van der: Verification of Workflow Nets. In: *ICATPN '97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets*. London, UK : Springer-Verlag, 1997, S. 407–426. – ISBN 3-540-63139-9 34

[2] Aalst, W. M. P. van der ; Jablonski, S.: Dealing with workflow change: identification of issues and solutions. In: *International Journal of Computer Systems Science and Engineering* 15 (2000), September, Nr. 5, S. 267–276 27, 28, 29, 99

[3] Aalst, W.M.P. van der ; Voorhoeve, M.: Ad-hoc workflow: problems and solutions. In: *DEXA '97: Proceedings of the 8th International Workshop on Database and Expert Systems Applications*. Washington, DC, USA : IEEE Computer Society, Sep 1997, S. 36–40 34, 35, 36, 99

[4] Baeyens, Tom: *jPDL: Simplified Workflow for Java*. 2006. – Presentation 20, 99

[5] Baeyens, Tom ; Faura, Miguel V.: *The Process Virtual Machine*. Online. Mai 2007. – URL `http://docs.jboss.com/jbpm/pvm/article/` 19

[6] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns*. Addison-Wesley Professional, January 1995 50

[7] Goedertier, Stijn ; Vanthienen, Jan: Designing Compliant Business Processes with Obligations and Permissions. In: *Business Process Management Workshops* Bd. 4103, Springer Berlin / Heidelberg, 2006, S. 5–14. – ISBN 978-3-540-38444-1 53

[8] Heinl, Petra ; Horn, Stefan ; Jablonski, Stefan ; Neeb, Jens ; Stein, Katrin ; Teschke, Michael: A comprehensive approach to flexibility in workflow management systems. In: *SIGSOFT Softw. Eng. Notes* 24 (1999), Nr. 2, S. 79–88 29

[9] jBPM: *JBoss jBPM jPDL 3.2 - jBPM jPDL User Guide*. Online. 2008. – URL `http://docs.jboss.org/jbpm/v3/userguide/` 19, 20, 21

[10] Kammer, Peter J. ; Bolcer, Gregory A. ; Taylor, Richard N. ; Hitomi, Arthur S. ; Bergman, Mark: Techniques for Supporting Dynamic and Adaptive Workflow. In: *Computer Supported Cooperative Work* 9 (2000), November, Nr. 3-4, S. 269–292 36, 37, 38, 99

[11] Küster, Jochen M. ; Ryndina, Ksenia ; Gall, Harald: Generation of Business Process Models for Object Life Cycle Compliance. In: *Business Process Management* Bd. 4714, Springer Berlin / Heidelberg, 2007, S. 165–181. – ISBN 978-3-540-75182-3 53

[12] Mendling, Jan ; Muehlen, Michael zur ; Price, Adrian: *Standards for Workflow Definition and Execution*. S. 281–316. In: Dumas, Marlon (Hrsg.) ; Aalst, W. M. P. van der (Hrsg.) ; Hofstede, A. H. M. ter (Hrsg.): *Process-Aware Information Systems: Bridging People and Software Through Process Technology*, Wiley Publishing, 2005 13

[13] Müller, Robert ; Greiner, Ulrike ; Rahm, Erhard: AgentWork: A Workflow-System Supporting Rule-Based Workflow Adaptation. In: *Data Knowl. Eng.* 51 (2004), Nr. 2, S. 223–256 40, 41, 42, 99

[14] Murata, Tadao: Petri nets: Properties, analysis and applications. In: *Proceedings of the IEEE* 77 (1989), Nr. 4, S. 541–580 23

[15] Narendra, N. C.: Adaptive workflow management - an integrated approach and system architecture. In: *SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*. New York, NY, USA : ACM, 2000, S. 858–865. – ISBN 1-58113-240-9 30, 45

[16] Narendra, N. C.: Flexible Support and Management of Adaptive Workflow Processes. In: *Information Systems Frontiers* 6 (2004), Nr. 3, S. 247–262 30, 31, 99

[17] Reichert, Manfred ; Dadam, Peter: ADEPTflex: Supporting Dynamic Changes of Workflow without Loosing Control. In: *Journal of Intelligent Information Systems* 10 (1998), S. 93–129 41, 42, 43, 44, 52, 53, 99

[18] Russell, Nick ; Aalst, Wil van der ; Hofstede, Arthur ter: Workflow Exception Patterns. In: *Advanced Information Systems Engineering* Bd. 4001/2006, Springer Berlin / Heidelberg, 2006, S. 288–302 32, 33, 34, 99

[19] Siebert, Reiner: Anpassungsfähige Workflows zur Unterstützung unstrukturierter Vorgänge, PoliFlow. In: *Proceedings EMISA-Fachgruppentreffen 1997: Workflow-Management-Systeme im Spannungsfeld einer Organisation* Bd. 1-1998, 1998, S. 87–90 39, 40, 99

[20] SoKNOS: *SoKNOS Projekt Homepage*. Online. – URL `http://www.soknos.de/` 9

[21] Swenson, Keith D. ; Shapiro, Robert M.: *BPM in Practice - A Primer for BPM and Workflow Standards*. März 2008 14

[22] Weber, Barbara ; Rinderle, Stefanie ; Reichert, Manfred: Change Patterns and Change Support Features in Process-Aware Information Systems. In: *Advanced Information Systems Engineering* Bd. 4495, Springer Berlin / Heidelberg, 2007, S. 574–588 58

[23] Weber, Barbara ; Rinderle-Ma, Stefanie ; Reichert, Manfred: Change Support in Process-Aware Information Systems - A Pattern-Based Analysis / University of Twente, Enschede, The Netherlands. 2007. – Forschungsbericht 58

[24] WfMC: *Workflow Management Coalition: Terminology & Glossary (Issue 3.0)*. 1999 13, 14, 18, 99

Glossar

ad-hoc von lat. „zu diesem, hierfür“ - im übertragenen Sinne bezeichnend für improvisierte Handlungen, die „aus dem Stegreif heraus“ spontan bzw. speziell für eine Situation entstanden sind. 13, 15, 17–21, 24, 34

Aktivität Einzelner Arbeitsvorgang, der einen logischen Schritt innerhalb eines Prozesses darstellt. 14–16, 18, 19, 37

Aktivitätsinstanz Die Repräsentation einer Aktivität innerhalb der einzelnen Ausführung eines Prozesses (d.h. innerhalb einer Prozessinstanz). 15, 18

Aktivitätsstatus Repräsentation der internen Bedingungen, die den Status der Aktivitätsinstanz zu einem bestimmten Zeitpunkt definieren. 15

Alternative Verzweigung Stelle im Workflow, an der die Entscheidung getroffen wird, welcher Zweig eines Workflows ausgeführt werden soll (falls mehrere alternative Wege vorhanden sind). 17, 54, 62, 69

Asynchrone Zusammenführung Stelle im Workflow, an der zwei oder mehrere alternative Verzweigungen des Workflows wieder in eine einzelne gemeinsame Aktivität zusammengeführt werden. Keine Synchronisierung notwendig, da keine Aktivitäten parallel ausgeführt wurden. 17, 62, 69

Bearbeiter Die Ressource, die den Arbeitsvorgang durchführt, der durch die Aktivitätsinstanz beschrieben wird. Dieser Arbeitsvorgang erklärt sich üblicherweise durch eine oder mehrere Work Items, die dem Bearbeiter durch seine Worklist zugewiesen werden. 14, 15, 18

Bereich der Adaption Die von einer Adaption betroffene Aktivität bzw. strukturelle Konstruktion (Sequenz, Nebenläufigkeit, Alternative). Zusätzlich Vorgänger- und Nachfolgeknoten des direkt betroffenen Konstrukts in der jBPM ProcessDefinition. 56, 57, 81

Event Das Auftreten einer bestimmten Bedingung (intern oder extern), die eine oder mehrere Aktionen des WfMS auslöst. 19

Geschäftsprozess Menge miteinander verknüpfter Arbeitsvorgänge, die gemeinsam ein definiertes Ziel erreichen sollen. 14

Geschäftsprozessmodellierung Die Zeitspanne, in der die Beschreibung eines Prozesses definiert oder modifiziert wird — also die Erstellung der Process Definition — auch Designzeit. 17, 18, 20, 39

Iteration Zyklus von Aktivitäten im Workflow, der die periodische Wiederholung einer oder mehrerer Aktivität/en erzwingt, bis eine bestimmte Bedingung erfüllt ist. 17

Nebenläufigkeit Teil eines Prozesses, in dem zwei oder mehrere Aktivitäten im Workflow parallel ausgeführt werden. Üblicherweise ermöglicht durch Parallele Verzweigungen und folgende Synchronisierung. 16, 53, 61

Parallele Verzweigung Stelle im Workflow, an der ein einzelner Kontroll-Thread in zwei oder mehrere Threads aufgeteilt wird, die dann parallel ausgeführt werden und es so ermöglichen, dass mehrere Aktivitäten gleichzeitig ausgeführt werden können. 16, 60, 66

Prozess Die formalisierte Ansicht eines Geschäftsprozesses, der durch eine aufeinander abgestimmte Menge von Prozessaktivitäten (parallel oder sequentiell vernetzt) repräsentiert wird. 15

Prozess Definition Repräsentation eines Geschäftsprozesses die es ermöglicht, dass automatisierte Manipulationen vorgenommen werden können. Die Prozess Definition besteht aus einem Netz von Aktivitäten und deren Verhältnisse zueinander, Bedingungen zum Start bzw. der Terminierung des Prozesses und Informationen über die einzelnen Aktivitäten (wie Bearbeiter, verknüpfte Anwendungen und Daten, ...) — auch Workflow Schema. 14–18, 27–31, 35–37, 41, 43, 58, 59

Prozess-Status Repräsentation der internen Bedingungen, die den Status der Prozessinstanz zu einem bestimmten Zeitpunkt definieren. 15

Prozessinstanz Repräsentation eines definierten Prozesses zur Laufzeit. Beinhaltet alle mit diesem Vorgang verknüpften Daten. Des Weiteren besitzt jede Prozessinstanz Schnittstellen, über die der entsprechende Prozess eindeutig angesprochen werden kann. 15–18, 25, 26, 31, 37, 43

Sequenz Teil eines Prozesses, in dem mehrere Aktivitäten im Workflow sequentiell ausgeführt werden. 16, 53

Sub Prozess Prozess, der innerhalb eines anderen (initiierenden) Prozesses (oder Sub Prozesses) ausgeführt oder aufgerufen wird und Teil des (initiierenden) Gesamtprozesses ist. Mehrere Ebenen von Sub Prozessen sind möglich. 15, 35

Synchronisierung Stelle im Workflow, an der zwei oder mehrere parallel laufende Aktivitäten zu einem einzelnen Kontroll-Thread zusammengeführt werden. 16, 17, 26, 60, 66

Transition Stelle während der Ausführung einer Prozessinstanz, an der eine Aktivität erledigt ist und der Kontroll-Thread zur folgenden Aktivität übergeht und diese startet. 16

Vorgangsbearbeitung Die Zeitspanne, in der der Prozess in Betrieb ist und Prozessinstanzen erstellt und/oder verwaltet werden — auch Laufzeit. 17, 18, 20, 33, 40, 58

Work Item Die Repräsentation der Aufgabe, die durch den Bearbeiter im Kontext einer Aktivität innerhalb der Prozessinstanz ausgeführt werden muss. 14, 15, 33, 64

Workflow Computergestützte Automatisierung eines Geschäftsprozesses. 14–16

Workflow Control Data Daten, die von einem WfMS und/oder einer Workflow Engine intern verwaltet werden. Die Daten repräsentieren den dynamischen Status des Workflow Systems und der Prozessinstanzen, beispielsweise Statusinformationen über jede Prozessinstanz oder den Status von Aktivitätsinstanzen. 18, 19

Workflow Engine Software-Dienst bzw. „Engine“, der die Laufzeitumgebung für eine Prozessinstanz bereitstellt. 13, 15, 16

Workflow Management System Software-System, das Workflows sowohl definiert und erstellt, als auch deren Ausführung überwacht. Die Ausführung erfolgt auf einer oder mehreren Workflow Engines. Das System interpretiert die Process Definition und interagiert mit den Bearbeitern, außerdem steuert es die Aufrufe von externen (IT) Anwendungen. 15

Workflow Relevant Data Daten die von einem WfMS genutzt werden um mögliche Zustandsübergänge einer Prozessinstanz zu bestimmen. Zum Beispiel Daten innerhalb von Übergangsbedingungen oder bei der Zuweisung von Bearbeitern. 18

Worklist Eine Liste von Work Items, die mit einem bestimmten Bearbeiter verknüpft sind (oder ggf. eine Gruppe von Bearbeitern, die eine gemeinsame Worklist besitzen). 15

Zustandsübergang Bewegung von einem internen Zustand (einer Prozess- oder Aktivitätsinstanz) zu einem anderen, die die Veränderung im Status des Workflows widerspiegelt. 15, 18

Übergangsbedingung Logischer Ausdruck der von einer Workflow Engine ausgewertet werden kann, um die Abfolge bei der Ausführung von Aktivitäten zu bestimmen. 16, 18

Akronyme

AWS Adaptives Workflow System. 39

BPM Business Process Management. 14

BPMS Business Process Management System. 14

ECA Event/Condition/Action. 41

GOP Graph Oriented Programming. 21, 24

JEMS JBoss Enterprise Middleware Suite. 19

jPDL jBPM Process Definition Language. 19–21

PVM Process Virtual Machine. 19, 20

SOA Serviceorientierte Architektur. 9

SWF Safe WorkFlow Net. 35

WfMC Workflow Management Coalition. 13, 14, 20, 24

WfMS Workflow Management System. 9, 10, 13–15, 18, 19, 25, 28, 30, 40, 64

Abbildungsverzeichnis

Tabellenverzeichnis

Listings