PCN – Eine DB-integrierte Ablaufumgebung für High-Level Petri-Netze

Markus Bon, Jürgen Zimmermann
Universität Kaiserslautern
(bon, jnzimmer)@informatik.uni-kl.de

Keywords: Workflow, Petri-Netz

Abstract:
Viele im Workflow-Management angebotenen Techniken sind auch sinnvoll in anderen Anwendungsbereichen einsetzbar. Aufgrund ihrer tiefen Integration in ein Workflow-Management-System kann man sie jedoch selten isoliert nutzen. Deshalb ist es sinnvoll, die einzelnen Dienste außerhalb eines integrierten WfMS in Form modularer Basisdienste bereitzustellen. Ein solcher Basisdienst ist eine flexible Abhängigkeitssteuerung. Für diese stellen wir eine Realisierung in Form einer in ein ORDBMS integrierten Petri-Netz-Komponente zur Verfügung.

I. Einleitung & Motivation

Workflow-Management ist in der kleinsten Hütte! Zumindest finden Methoden des Workflow-Managements (WfM) in viel mehr Bereichen ihre Anwendung, als man erwartet. Leider werden jedoch Workflow-Management-Systeme oft nicht eingesetzt, weil sie – bedingt durch ihr sehr spezielles Workflow-Modell – für andere Anwendungsgebiete nicht optimal ausgelegt sind. Viele Anwendungsfelder nutzen lediglich ein stark eingeschränktes Spektrum der Funktionalität, beispielsweise zur Kontrollfluß- oder Datenflußsteuerung. Zwar ist eine Separierung in voneinander unabhängige Aspekte bereits in MOBILE ([6]) motiviert, jedoch integriert MOBILE diese wieder in ein WfMS, so daß sie nicht unabhängig nutzbar sind. Uns erscheint es jedoch sinnvoller, Komponenten für diese Aufgaben (Aspekte) in Form von frei nutzbaren Basisdiensten unabhängig von einem WfMS zu realisieren. Eine Anwendung muß dann nicht das gesamte WfMS zur Modellierung ihrer Anforderungen heranziehen, sie kann sich auf die Integration des jeweiligen Basisdienstes beschränken. Andererseits können die Basisdienste wiederum zur Realisierung von angepaßten WfMS genutzt werden, da sie nicht auf ein bestimmtes Modell festgelegt sind.

Welche Basisdienste sollen denn nun angeboten werden? Als einen solchen Dienst haben wir eine flexible Abhängigkeitskontrolle identifiziert, welche sich für die Verwaltung und Kontrolle der verschiedensten Arten von Abhängigkeiten nutzen läßt. Wir haben ihn in Form eines „Dependency Controller“ (DC) realisiert. Als Grundlage der Implementierung haben wir uns für Petri-Netze ([2]) entschieden, da diese sich bereits in MOBILE und anderen WfMS für die Kontrollflußmodellierung und -steuerung hinreichend bewährt haben. Unserer Ansicht nach sind sie aber auch für andere Abhängigkeiten (z. B. für die Datenflußkontrolle) gut verwendbar. Die von uns eingesetzten Prozeßkontrollnetze (PCN, [7]) weisen gegenüber den klassischen Petri-Netzen verschiedene Erweiterungen auf, die uns eine höhere Modellierungsmächtigkeit bieten. Marken sind beispielsweise strukturiert und tragen Information über die Objekte, die sie repräsentieren. Diese Information kann zur Auswertung von Prädikaten benutzt werden. Weiterhin ermöglicht uns die neue Kantenart der Testkanten, die Struktur des Netzes dem aktuellen Zustand anzupassen. Genaugenommen handelt es sich bei den PCN um eine hinsichtlich effektiver und einfacher Implementierung modifizierte Variante der Produktnetze ([1], [8]).

Da der DC (oder die „PCN-Komponente“) universell einsetzbar sein soll, ist er vollkommen unabhängig von einer konkreten Anwendung. Wir können jedoch davon ausgehen, daß nutzende Anwendungen mit hoher Wahrscheinlichkeit für ihre Datenhaltung sowohl aus Stabilitäts- als auch aus Integritätsgründen Datenbanksysteme einsetzen. Es liegt daher nahe, für den DC dieses vorhandene DBMS mitzunutzen. So können sowohl die Struktur der einzelnen PCN, als auch ihr Zustand zur Laufzeit ohne zusätzliche Anforderungen an die Ablaufumgebung persistent gehalten werden. Da der komplette DC einen Basisdienst darstellen soll, ist es auch nur konsequent, die Datenhaltung in der Datenbank um die benötigte Anwendungslogik zu erweitern. Auf diese Art ist der gesamte Dienst innerhalb des DBMS gekapselt! Eine geeignete Plattform für diese Art der Realisierung stellen ORDBMS dar, da sie eine Server-seitige Integration von Anwendungslogik in Form von Erweiterungsroutinen ermöglichen. So kann fast der komplette DC innerhalb eines ORDBMS realisiert werden, lediglich die Anbindung an externe Applikationen durch sog. Aktoren befindet sich noch außerhalb.

Unseren Artikel haben wir wie folgt aufgebaut: Im nächsten Abschnitt beschreiben wir kurz die Struktur von PCN und gehen auf ihre Dynamik ein. Abschnitt 3 beschreibt die konkrete Realisierung des DC mit Hilfe eines ORDBMS, abschließend folgen Zusammenfassung und Ausblick.

II. Prozeßkontrollnetze

Prozeßkontrollnetze sind eine Petri-Netz-Variante. Wir haben sie aus den in [11] eingeführten Produktnetzen abgeleitet und diese dabei für eine einfachere und effektivere Implementierung angepaßt bzw. erweitert. Prozeßkontrollnetze sind High-Level Petri-Netze [2], d. h. sie besitzen strukturierte, individuelle Marken und „gewichtete“ Kanten. Weiterhin wurden, zusätzlich zu Eingangs- und Ausgangskanten, Testkanten eingeführt, die abhängig vom aktuellen Netz-Status Transitionen aktivieren oder blockieren können.

High-Level Petri-Netze bieten uns eine Reihe von Vorteilen. Die Modellierung von komplexen Problemen ist, verglichen mit normalen Petri-Netzen, sehr viel einfacher, da sehr ausdrucksmächtige Konstrukte angeboten werden. In einem Vorspann kann der Benutzer frei bestimmen, welchen Definitionsbereich eine Stelle haben soll. Er legt damit gleichzeitig auch fest, welche Information von einer Marke transportiert werden kann. Soll beispielsweise ein Dokument näher beschrieben werden, so kann z. B. in der Marke nicht nur die Dokumentnummer, sondern auch der Autor, das Entstehungsdatum oder die Seitenanzahl enthalten sein. Wir haben die Möglichkeit, die Struktur des Netzes dem Problem anzupassen!

Weiterhin können vom Benutzer Funktionen eingeführt werden, die innerhalb des Netzes in Kantenanschriften oder Transitionspädikaten frei benutzt werden können. Wir können somit auch die Semantik dem Problem anpassen.

Zu guter Letzt erlauben uns Testkanten, die Struktur des Netzes während der Laufzeit abhängig vom Systemzustand zu ändern, in dem Transitionen durch sie aktiviert oder deaktiviert werden! Sie ermöglichen uns weiterhin das Formulieren von Verbotskanten, wie sie in [11] beschrieben wurden. Erst mit diesen wird (im Gegensatz zu normalen Petri-Netzen) volle Berechenbarkeit erreicht! Ein „ODER“ ist mit einem normalen Petri-Netz nicht darstellbar, da (aufgrund des indeterministischen Schaltens) der „UND“-Fall immer auch beide „ODER“-Fälle schaltbereit werden läßt. Wie in Abb. 1 zu sehen ist, läßt sich mit Testkanten leicht ein „ODER“ konstruieren! Die Kantenanschrift „~<>“ an den Testkanten sperrt die zugehörige Transition, sobald eine Marke vorhanden ist. Sind in beiden Eingangsstellen Marken vorhanden, kann nur die mittlere Transition feuern!.

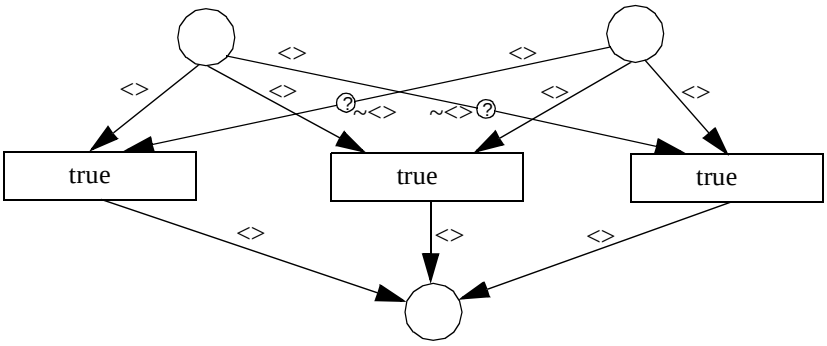

Abb. 1: Ein „Oder“-PCN

Wir geben nun zuerst einen kurzen Überblick über die statische Struktur von PCN, danach beschäftigen wir uns mit dem dynamischen Ablaufmodell.

1. Statische Elemente

In diesem Abschnitt betrachten stellen wir die statischen Elemente eines PCN vor, die wir für die Beschreibung der Struktur einsetzen können.

Wie alle Petri-Netze bestehen auch PCN aus den Grundelementen Stellen, Transitionen und Kanten (vgl. Abb. 2). Der aktuelle Zustand des Netzes ergibt sich aus Marken, die in den Stellen verteilt sind. Im Gegensatz zu den einfachen Marken des klassischen Petri-Netzes sind die Marken eines PCN individuell unterscheidbar und strukturiert, können somit auch mehr Information transportieren. Welche Information eine Marke in einer Stelle aufnehmen kann, wird durch deren Definitionsbereich („Domain“) festgelegt. Dieser Definitionsbereich setzt sich aus einer Menge von

(einfachen) Attributen zusammen, denen wiederum bestimmte Wertebereiche zugeordnet sind. Aus diesem „Produkt von Mengen“ leitete sich auch der Name „Produktnetze“ ab.

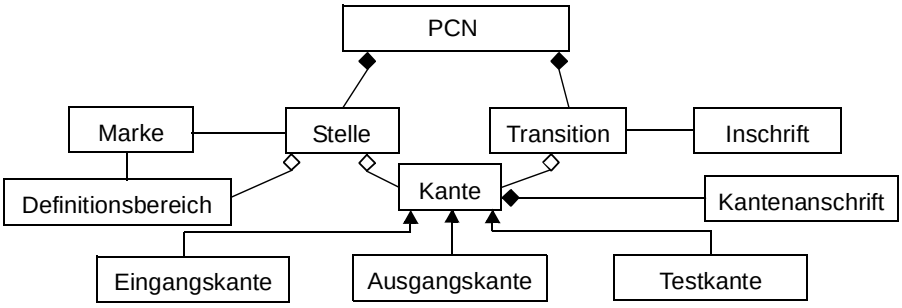

Abb. 2: Metamodell der PCN

Jedes einfache Attribut entspricht einem Basistyp. Als Basistypen sind alle in gebräuchlichen DBMS angebotenen Basistypen wie beispielsweise CHAR, STRING oder INTEGER zulässig. Wie nicht anders zu erwarten, besteht eine Attributdeklaration nun aus einem Namen und einem zugeordneten Typ. Der Definitionsbereich zu einer Stelle läßt sich beispielsweise wie folgt beschreiben:

Domain_S = {Name: String(50); Alter: Integer; Abteilung: String(3);}
DOM(S) = Domain_S;

Kanten lassen sich in Eingangs-, Ausgangs- und Testkanten unterscheiden. Dabei führen Eingangskanten von einer Stelle zu einer Transition, Ausgangskanten von einer Transition zu einer Stelle. Testkanten verbinden zwar ebenfalls Stellen mit Transitionen, jedoch fließen keine Marken über sie. Sie dienen vielmehr dazu, eine Transition entsprechend des aktuellen Systemzustands wie ein Schalter zu blockieren oder freizugeben. Jeder Kante ist eine Kantenanschrift zugeordnet, die sich aus einzelnen „Markenbeschreibungen“ mit zugehöriger Vielfachheit (Gewichtung) zusammensetzt. Bei Eingangskanten kann auf diese Weise bestimmt werden, welcher Form Marken genügen müssen, um von einer Transition verarbeitet zu werden. Bei Ausgangskanten werden die zu generierenden neuen Marken beschrieben.

Wie sieht denn nun die Struktur einer Marke aus? Wie schon erwähnt, besitzen Stellen Definitionsbereiche. Eine Marke muß dem Definitionsbereich der Stelle entstammen, auf der sie sich befindet. Die Marke *<Name:‘Maier’, Alter: 47, Abteilung: ‘A01’>* kann beispielsweise auf einer Stelle liegen, die den oben eingeführten Definitionsbereich besitzt. An dieser Beispielmarke zeigt sich auch ein weiterer Unterschied zu den Produktnetzen: Die Attribute werden über den Namen identifiziert, nicht über die Reihenfolge (*<Name: ‘Maier’, Alter: 47, Abteilung: ‘A01’>* ist äquivalent zu *<Alter: 47, Abteilung: ‘A01’, Name: ‘Maier’>*). Dies bringt bei der Implementierung einen entscheidenden Vorteil: Attribute, die bei der Auswertung nicht berücksichtigt werden, brauchen nicht über „freie Variablen“ ausgeblendet zu werden, wie dies bei reinen Positionsparametern notwendig ist. Gerade bei komplexeren Marken macht sich dies bemerkbar, d. h., sie sind deutlich angenehmer zu handhaben.

Zum Konsumieren/Erzeugen/Testen von Marken kommen Kantenanschriften zum Einsatz, die alle eine ähnliche, aber je nach zugehöriger Kantenart doch leicht verschiedene Semantik besitzen. Eine Kantenanschrift besteht immer aus Markenbeschreibungen, die gegebenenfalls mit einer Gewichtung versehen sind und untereinander verknüpft werden können. Über sogenannte „gebundene Variablen“ ist es sogar möglich, Anforderungen an Marken in verschiedenen Eingangsstellen zu verknüpfen.

Betrachten wir nun aber der Reihe nach die möglichen Kantenanschriften: Eine typische Anschrift einer Eingangskante sieht z. B. wie folgt aus: *<Alter: a, Name: n, Abteilung:‘A01’>*. ‘A01’ ist dabei eine Konstante, *a* und *n* sind gebundene Variablen, ihr Wert wird durch die Attributwerte der tatsächlich ausgewählten Marke bestimmt („gebunden“). Konstanten werden als Anforderung an die auszuwählende Marke benutzt, gebundene Variablen zur Formulierung von Transitionsinschriften. Dabei handelt es sich um Prädikate, die erfüllt sein müssen, um eine Transition feuern zu lassen. Beispielsweise sorgt die Inschrift *„a = 47“* dafür, daß nur Marken konsumiert werden, deren Alter-Attribut mit dem Wert 47 belegt ist. Gebundene Variablen spielen auch bei Ausgangs- und Testkanten eine

wichtige Rolle, die wir gleich noch näher betrachten werden. Die vollständige Syntaxbeschreibung für Kantenanschriften befindet sich in Anhang C.

Die Markenbeschreibung kann neben der einfachen Zuordnung eines Attributs zu einer gebundener Variable auch einfache arithmetische Ausdrücke oder Funktionsaufrufe enthalten. Als Parameter sind Konstanten und schon belegte gebundene Variablen zulässig. Die Anschrift „*<a:u> + <a:(INC; x=u)>*“ ist beispielsweise eine gültige Anschrift für die Eingangskante des Netzes in Abb. 3, „*<a:(INC; x=u)>*“ hingegen ist unzulässig, da „*u*“ nirgends gebunden wird.

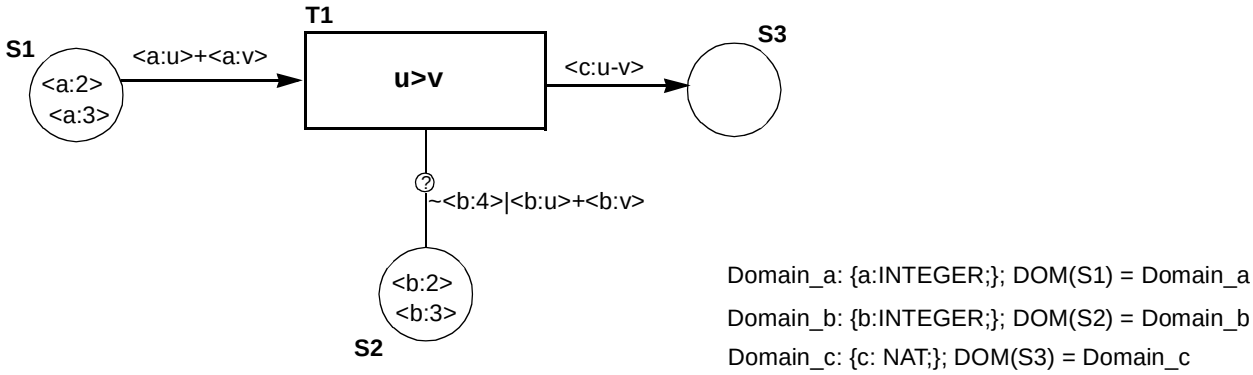

Abb. 3: Graphische Notation eines Beispiel-PCN

Eine Kantenanschrift kann mehrere Marken gleichzeitig auswählen. Sollen diese Marken in den aufgeführten Attributen wertgleich sein, so wird einfach die Markenbeschreibung mit einem Gewicht versehen. Wollen wir beispielsweise immer zwei Marken mit gleichem Namen und beliebigem Alter verarbeiten (und den Namen in einer gebundenen Variable bereitstellen), so funktioniert dies mit der Kantenanschrift „*2<name: n>*“. Sollen verschiedene Marken ausgewählt werden, können mehrere Markenbeschreibungen konjunktiv verknüpft (+) werden. „*<name:m> + <name:n>*“ wählt zwei Marken aus und ordnet die (nicht unbedingt veschiedenen) Werte der Namensattribute den gebundenen Variablen „*m*“ und „*n*“ zu. Soll sichergestellt werden, daß die Namen der beiden qualifizierten Marken wirklich verschieden sind, so läßt sich dies über die Transitionsinschrift „*n != m*“ erreichen. Die Zuordnung der gebundenen Variablen ist transitionslokal, d. h. alle gebundenen Variablen mit gleichem Namen müssen in allen Markenbeschreibungen an allen Kanten gleich belegt werden! Die Zuordnung gebundener Variablen geschieht nur in Eingangskanten. Insbesondere können daher – zur Wahrung der Integrität – in Ausgangs- und Testkanten nur Variablen benutzt werden, die auch an mindestens einer Eingangskante auftauchen. Die Abbildung aller gebundenen Variablen auf einen Wert bezeichnen wir als *Belegung*.

An den Ausgangskanten beschreibt die Kantenanschrift, welche Marken generiert werden sollen. Da diese Marken auf der zugehörigen Ausgangsstelle abgelegt werden sollen, muß die Verträglichkeit mit deren Definitionsbereich gewährleistet sein. Nach dem Schalten der Transition werden alle (konjunktiv verknüpften) Markenbeschreibungen mit Hilfe von Konstanten und gebundenen Variablen, arithmetischen Operationen und Funktionen dazu benutzt, neue Marken zu erzeugen.

Eine besondere Rolle spielen Testkanten, da sie zwar eine Kantenanschrift besitzen, über sie jedoch weder Marken konsumiert noch erzeugt werden. Sie dienen vielmehr dazu, eine Transition in Abhängigkeit vom aktuellen Zustand zu blockieren („Verbotskante“) oder freizugeben. Wie auch bei den Ausgangskanten dürfen in den Markenbeschreibungen der Kantenanschrift nur Konstanten, von den Eingangskanten belegte gebundene Variablen, arithmetische Operatoren und Funktionsaufrufe benutzt werden. Markenbeschreibungen können mit einem Gewicht versehen werden, mehrere Markenbeschreibungen können *konjunktiv (+)* verknüpft werden. Liegen in der Teststelle ausreichend Marken, um die Kantenanschrift zu erfüllen, ist die Transition freigegeben. Die Testkante verhält sich in diesem Fall wie eine Eingangskante (nur, daß eben keine neuen gebundenen Variablen eingeführt werden dürfen und keine Marken verbraucht werden). Oft existiert jedoch mehr als nur ein Systemzustand, auf den die Transition reagieren können soll. Daher ist es bei einer Testkante möglich, mehrere Kantenanschriften *disjunktiv (|)* zu verknüpfen. Weiterhin wollen wir in bestimmten Situationen die Transition explizit sperren. Daher kann eine (Teil-) Kantenanschrift mit Hilfe von *NOT (~)* negiert werden. Die zugehörige Transition wird folglich genau dann gesperrt, wenn genügend Marken vorhanden sind, um die Kantenanschrift zu erfüllen.

Um uns später eine einfache Implementierung zu ermöglichen, beschränken wir uns auf Kantenanschriften in Disjunktiver Normalform (DNF). Dies bedeutet jedoch für den Netzmodellierer keine große Einschränkung, da die DNF der „natürlichen“ Formulierung einer Kantenanschrift sehr entgegenkommt.

In Abb. 3 ist ein Beispiel-PCN mit drei Stellen, einer Transition mit Inschrift, einer Eingangs-, einer Ausgangs- und einer Testkante zu sehen. Die Kantenanschrift der Eingangskante fordert zwei Marken, und tatsächlich liegen in S1 zwei Markenkandidaten bereit. Es sind zwei Belegungen möglich: B_1={u:2, v:3} oder B_2={u:3, v:2}. Mit B_1 ist die Transitionsbedingung „$u > v$“ verletzt, sie muß verworfen werden. Für B_2 ist die Bedingung aber erfüllt. Nun muß noch die Testkante überprüft werden. Für B_2 ergibt sich aus der Kantenanschrift: „~<b:4> | <b:3>+<b:2>“. Die Marken in S2 erfüllen in diesem Fall sogar beide Teilbedingungen. T1 ist folglich feuerbereit.

Wie schon mehrfach angesprochen, können in Markenbeschreibungen und Transitionsinschriften Funktionen benutzt werden. Dazu muß die Funktion eine passende Signatur aufweisen, d. h., einen passenden Rückgabewert liefern, und eine entsprechende Anzahl formaler Parameter des richtigen Typs besitzen (denn offensichtlich macht beispielsweise „INC(‘Maier’)“ wenig Sinn). Für die tatsächliche Implementierung der Funktion stehen uns alle Möglichkeiten offen: Wir können beispielsweise für einfache Auswertungen die Möglichkeiten nutzen, die uns SQL bietet (immer bedenkend, daß wir ja ein Datenbanksystem als Basis einsetzen). Komplexere Funktionen können z. B. in Java realisiert werden. Da diese jedoch über SQL-Wrapper dem System zugänglich gemacht werden können, stehen auch diese zur Verwendung mit SQL zur Verfügung.

2. Dynamische Interpretation

Wir beschreiben nun, wie ein PCN animiert werden kann, d. h., wie Transitionen aufgefunden werden können, die zum Schalten bereit sind, was beim Schalten passiert und wie nach dem Schalten einer Transition weiter verfahren werden muß.

Die Dynamik (Animation) von PCN wird durch das Schalten (Feuern) von Transitionen bestimmt. Dabei werden Marken aus Stellen, die über Eingangs- oder Abräumkanten mit der Transition verbunden sind, entfernt und neue Marken in Ausgangsstellen eingefügt. Damit eine Transition feuern kann, muß ihre Schaltbedingung erfüllt sein. Diese setzt sich aus den Kantenanschriften der Eingangs- und Testkanten, sowie der Transitionsinschrift zusammen.

Bei Eingangskanten wird überprüft, ob sich in der zugeordneten Stelle entsprechend viele Marken befinden, um jeder Markenbeschreibung der Kantenanschrift eine eigene, passende Marke zuzuordnen. Diese Ansammlung von Marken bezeichnen wir als *Auswahl*. Die Auswahl bestimmt, wie die gebundenen Variablen der Kantenanschrift belegt werden. Die Abbildung der gebundenen Variablen in ihre Werte heißt daher auch *Belegung*. Für Testkanten muß sichergestellt sein, daß die Kantenanschrift erfüllt wird. Dazu müssen in der zugehörigen Teststelle genügend Marken vorhanden sein, um die Kantenanschrift hinsichtlich der durch die Eingangskanten erzeugten Belegung zu erfüllen. Die Transitionsinschrift ist ein Prädikat, das über den gebundenen Variablen formuliert ist, und muß natürlich ebenfalls erfüllt werden.

Wenn für eine Transition eine Auswahl gefunden wird, für die alle Bedingungen erfüllt sind, so kann die Transition feuern. Hierzu werden zunächst alle Marken der Auswahl aus den zugehörigen Stellen entfernt. Über die Ausgangskanten werden Marken in die Ausgangsstellen eingefügt.

Als Auslöser für das potentielle Schalten einer Transition, d. h. das Anstoßen der Bedingungsüberprüfung, gelten folgende Ereignisse:

- das Einfügen einer Marke in eine Stelle mit Eingangskante,
- das Einfügen einer Marke in eine Stelle mit Testkante,
- das Entfernen einer Marke aus einer Stelle mit Testkante,
- Änderung des Bearbeitungszustands an Marken in *Magischen Stellen*.

Magische Stellen sind dabei Stellen, in denen Marken Aufträge an (externe) Aktoren darstellen. Sie ermöglichen somit eine Verbindung zur Außenwelt. Eine Markenänderung erfolgt durch eine für das Netz unsichtbare Komponente auf „magische“ Art und Weise. Marken in Magische Stellen werden mit einem Bearbeitungszustand versehen und sind erst nach erfolgreicher Bearbeitung für das Schalten nachfolgender Transitionen wieder relevant.

Wurde eine Stelle auf eine der vier beschriebenen Arten verändert, prüft das System für alle über Eingangs- oder Testkanten verbundenen Transitionen, ob sie feuerbereit sind. Aus der Menge der feuerbereiten Transitionen wird nun eine ausgewählt[1] und gefeuert. Durch dieses Feuern kann zweierlei geschehen: Zum einen können ehemals feuerbereite Transitionen jetzt nicht mehr feuerbar sein, zum anderen werden durch das Konsumieren/Erzeugen von

Marken neue Transitionen feuerbar. Das Anpassen dieses „Pools“ feuerbarer Transitionen umfaßt folglich zwei Schritte:

1. Überprüfen der bisher feuerbereiten Transitionen
2. Hinzufügen der neuen Transitionen

Aus dem angepaßten Pool wird dann wiederum eine Transition zum Feuern ausgewählt. Dieser Vorgang wiederholt sich, bis der Pool leer ist.

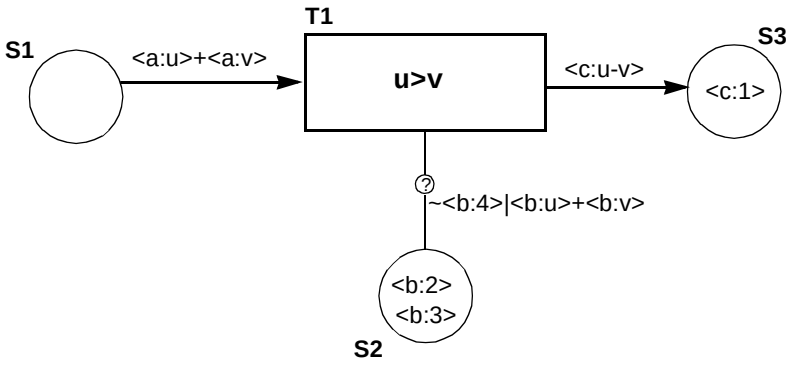

Abb. 4: Beispiel-PCN nach dem Schalten

In Abb. 4 ist noch einmal das Netz aus Abb. 3 dargestellt. Im letzten Abschnitt hatten wir die Belegung B_2 gefunden, mit der T1 feuerbar war. Nach dem Feuern ergibt sich die dargestellte Situation. Über die Ausgangskante wurde eine neue Marke in S3 gelegt. Der Pool feuerbarer Belegungen/Transitionen ist leer. Da S3 nicht mit einer Eingangskante verbunden und S2 unverändert ist, kommen auch keine neuen Transitionen hinzu. Das Netz ist terminiert.

III. Realisierung mit Hilfe eines ORDBMS

Nachdem wir die Konzepte der PCN eingeführt haben, beschäftigen wir uns in diesem Abschnitt mit der Realisierung des DC in Form der PCN-Komponente unter Zuhilfenahme eines ORDBMS. Die Ansiedlung im ORDBMS hat vor allem einen Grund: Unsere Implementierung nutzt intensivst Datenbankanfragen, um den Zustand des Netzes zu bestimmen, bzw. zu manipulieren. Unser Ansatz erspart einfach die teure Kommunikation aus dem Server heraus. Die fehlende Logik ist minimal und läßt sich leicht integrieren. Weiterhin wird uns mit dem Trigger-Mechanismus von SQL das Auffinden neuer feuerbarer Transitionen sehr leicht gemacht.

Die Komponente präsentiert PCN in drei unterschiedlichen Teilen (vgl. Abb. 5):

- die reine Verwaltung der Strukturen, wie sie durch die Modellierung gegeben sind (Struktur-DB)
- der aktuelle Zustand, d. h. die Marken in den Stellen, in dem sich ein PCN befindet (Zustand-DB)
- die eigentliche Animationslogik (Animation)

Für die Anwendungsentwicklung auf Basis der PCN-Komponente werden zwei Schnittstellen angeboten: Eine PCN-Struktur-API bietet Funktionen für die Struktur- und Zustandsmanipulation an, bspw. durch ein graphisches

1. Die Auswahlstrategie soll hier noch keine Rolle spielen; da Petri-Netze jedoch nicht deterministisch sind, kann eine beliebige feuerbare Transition mit einer beliebigen gültigen Belegung ausgewählt werden. Es ist also genauso möglich, dies zufällig zu tun, wie auch beliebig komplexe Verfahren zur Auswahl einer „optimalen“ Belegung durchzuführen.

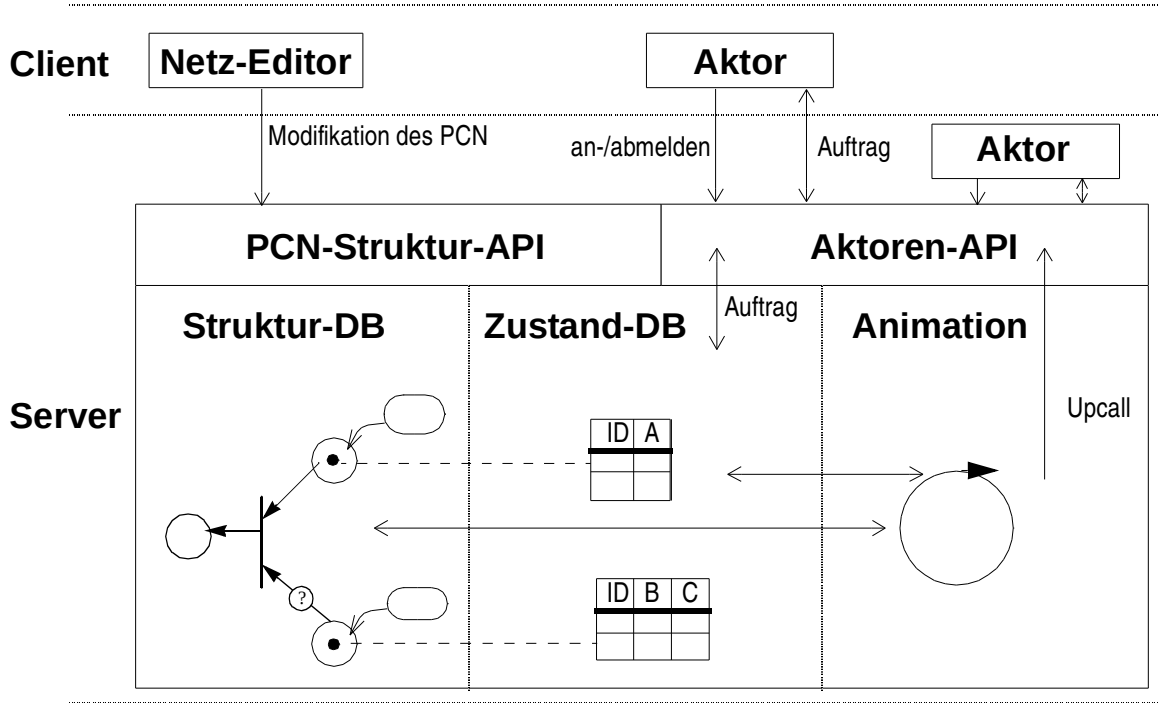

Abb. 5: Architekturmodell der PCN-Komponente

Entwicklungswerkzeug. Die Anbindung von Applikationen zur Laufzeit und deren Kommunikation mit dem DC geschieht über die Aktoren-API.

Sowohl Struktur als auch Zustand eines PCN werden im Datenbanksystem gehalten, um die Informationen persistent und unter den Transaktionseigenschaften des ORDBMS zu halten. Weiterhin wird die Animationslogik in das ORDBMS integriert, weil hierdurch die gesamte Verarbeitung des PCN (bis zu einem stabilen Zustand, in dem keine Transition mehr schalten kann) Server-seitig stattfinden kann. Dadurch entfällt die Kommunikation wie bei einer Client-seitigen Verarbeitung, und es eröffnet sich die Möglichkeit, auch DB-seitige Mechanismen (*Trigger*) für die Aktivierung einer Transition zu nutzen.

ORDBMS bieten sich als eine geeignete Zielplattform an, da sie – neben der Datenhaltung – auch die Server-seitige Integration von Anwendungslogik erlauben.

1. Statische Abbildung

Bevor wir uns mit der Realisierung der Animation beschäftigen, erwähnen wir nur kurz die Abbildung der statischen PCN-Struktur auf ein DB-Schema. Bis auf die Modellierung von Kanten und Marken erweist sich die Umsetzung der PCN-Struktur als einfach, so daß wir auf diese Modellierung, die in Anhang A dargestellt ist, an dieser Stelle nicht näher eigehen.

Zur Speicherung von Marken existiert eine globale Tabelle (`PCN_STATE_TOKEN`), in der jede Marke neben der Zuordnung zu ihrer Stelle eine eindeutige `ID` bekommt. Die Werte der Marken werden in einer eigens für jeden Definitionsbereich angelegten Tabelle gespeichert. Jede dieser Tabellen, die durch geeignete CREATE-Statements erzeugt werden, spiegelt direkt die Struktur des jeweiligen Definitionsbereichs wider und besitzt weiterhin ein Attribut `ID`, das gleichzeitig als Primärschlüssel dient und als Fremdschlüssel auf `ID` von `PCN_STATE_TOKEN` fungiert. Dadurch wird die systemweite Identifikation von Marken und ihren Werte gewahrt. Die Trennung von Marken und ihren Werten erlaubt es, die Schemaevolution für das Anlegen neuer Definitionsbereiche unkritisch zu gestalten, da lediglich neue Tabellen generiert werden und keine bestehenden Tabellen geändert werden.

Kanten werden in einer einzigen Tabelle (`PCN_STATIC_ARC`) abgelegt, die zugehörigen Kantenanschriften sind in einer eigenen Tabelle (`PCN_STATIC_ARC_ANNOTATION`) gespeichert. Hierzu wird jede Markenbeschreibung als Bestandteil der Kantenanschrift einzeln abgelegt und mit einem Fremdschlüssel auf die jeweilige Kante gespeichert.

2. Realisierung der Dynamik

Kernproblem des DC ist das dynamische Verhalten des PCN. Deshalb werden wir im folgenden die zu lösenden Probleme genauer diskutieren.

Damit eine Transition feuern kann, muß eine geeignete Auswahl und Belegung gefunden werden, für die auch die Nebenbedingungen erfüllt sind. Zur Suche ist die Formulierung einer SQL-Anfrage naheliegend (insbesondere, da die gesamte Information zur Struktur und zum Zustand des Petri-Netzes in der Datenbank abgelegt ist), die genau diejenigen Auswahlergebnisse mit zugehöriger Belegung liefert, welche sich potentiell zum Feuern eignen.

Dazu müssen alle Bedingungen, die in den Kantenanschriften und dem Transitionsprädikat enthalten sind, in eine Form überführt werden, aus der sich auf einfache Art und Weise eine solche SQL-Anfrage generieren läßt. Dies geschieht in Form eines Operatorbaums, dessen Aufbau wir im nächsten Abschnitt betrachten. Danach zeigen wir, wie aus diesem Operatorbaum eine konkrete SQL-Anfrage zur Auswahl-Findung hergeleitet werden kann.

a) Aufbau des Operatorbaums

An dem Beispiel aus Abb. 3 wollen wir den Aufbau des Operatorbaums erläutern. Dazu führen wir schrittweise die für die Umsetzung benötigten Operatoren ein und erklären die Vorgehensweise. Die einzelnen Knoten des Operatorbaums werden in unserer Darstellung durchnumeriert. Links von der Zuweisung „:=“ stehen jeweils die Ergebniswerte der Operatoren, die über an Identifikatoren gebunden werden. Die Operatoren stehen, gefolgt von ihren Operanden, rechts von dem „:=“. Tretende Identifikatoren als Operanden auf, so wird auf den Wert des jeweiligen Identifikators bezug genommen.

Die Wurzel jedes Operatorbaums bildet der „root“-Knoten.

(1) $vars, $places := root ()

Die hier angegebenen Resultate stellen zum einen den aktuellen Zustand des PCN ($places) dar, wie er sich aus der konkreten Stellen-Markierung ergibt, zum anderen eine leere Variablentabelle ($vars), in welche die Bindungen der gebundenen Variablen eingetragen werden.

Als nächstes wollen wir die Kantenanschrift („<a:u>+<a:v>“) der Eingangskante analysieren. Sie besteht aus zwei Markenbeschreibungen, die jeweils das (implizite) Gewicht „1“ haben. Für jede Markenbeschreibung (aller Eingangskanten) wird ein „AccessPlace“-Operator erzeugt, der ausgehend aus einem Zustand des PCN für eine Stelle soviel paarweise verschiedene Marken binden soll, wie es seinem Gewicht entspricht. Das Ergebnis ist eine entsprechende Markenbeschreibung.

(2) t1 := AccessPlace ($places(1), S1, 1)

(3) t2 := AccessPlace ($places(1), S1, 1)

Die in Klammern hinter $places angegebene Zahl beschreibt den Operator, der zuletzt eine Manipulation des Zustandes von $places vorgenommen hat, also in diesem Fall der „root“-Operators, der diesen Zustand „erzeugt“ hat.

Wenn für alle Eingangskanten die Markenbeschreibungen gebunden sind, werden diese zu einer Auswahl kombiniert. Dies ist die Aufgabe des „Selection“-Operators, der weiterhin darauf achtet, daß alle in der erzeugten Auswahl aufgeführten Marken verschieden sind.

(4) c1 := Selection (t1, t2)

Als nächstes werden die Werte der gebundenen Variablen festgelegt. Hierzu wird die vorhandene Variablentabelle um die Attribute ergänzt, die den Variablenwert bestimmen. Dabei geht der aktuell gültige Zustand der Variablentabelle als Operand in den „BindVariable“-Operator ein, der als Ergebnis den neuen Zustand der Variablentabelle liefert. Weiterhin muß noch festgelegt werden, auf welche Auswahl sich die Bindung bezieht, welche Variable gebunden werden soll und an welchen Attributwert von welcher Markenbeschreibung sie gebunden werden soll.

(5) $vars := BindVariable ($vars(1), c1, u, t1, a)

(6) $vars := BindVariable ($vars(5), c1, v, t2, a)

In Knoten (6) wird auf den Zustand der Variablentabelle bezug genommen, der durch die Variablenbindung in Knoten (5) hervorgeht.

Bis hierhin haben wir nun die gesamte Information, die sich in der Kantenanschrift der Eingangskante wiederfindet, in den Operatorbaum übertragen. Als nächstes widmen wir uns dem Transitionsprädikat, das – obwohl notiert durch „u > v“ – eigentlich das Ergebnis eines Funktionsaufrufs „GreaterThan (u, v)“ darstellt. Bevor wir die Funktion auswerten können, müssen wir die Werte der gebundenen Variablen „u“ und „v“ aus der Variablentabelle extrahieren. Hierzu dient der „AccessVariable“-Operator:

(7) v1 := AccessVariable ($vars(6), u)

(8) v2 := AccessVariable ($vars(6), v)

Anschließend können wir aus diesen Werten mit Hilfe eines Funktionsaufrufs den Wert der Funktion „GreaterThan“ ermitteln:

(9) b1 := FunctionCall (GreaterThan, v1, v2)

Da das Transitionsprädikat eine Bedingung an die Auswahl und Belegung stellt, die ebenso erfüllt sein muß, erweitern wir die Auswahl um eben diese Bedingung durch Anwenden des „Condition“-Operators. Das Ergebnis ist eine um die Bedingung erweiterte Auswahl.

(10) c2 := Condition (c1, b1)

Die weiteren Bedingungen, die eine Auswahl erfüllen muß, werden durch die Testkanten beschrieben. Dazu werden die Kantenanschriften, die disjunktiv kombiniert sind, isoliert voneinander betrachtet und ähnlich der bisherigen Vorgehensweise zu je einer Auswahl zusammengesetzt. Die Kantenanschrift „~<b:4>“ wird in folgende Knoten umgesetzt:

(11) t3 := AccessPlace ($places(1), S2, 1)

(12) c3 := Selection (t3)

(13) c4 := ValueRestrict (c3, t3, b, =, 4)

(14) b2 := IsEmpty (c4)

Neu hinzugekommen sind die Operatoren der Knoten (13) und (14). „ValueRestrict“ (Knoten (13)) führt eine Werterestriktion der Auswahl durch, indem gefordert wird, daß das angegebene Attribut („t3.b“) in einer bestimmten Relation („=“) zu einem Wert (Konstante „4“) steht. „IsEmpty“ in Knoten (14) realisiert die Semantik der Testkante für die Negation, nämlich daß keine solche Auswahl existieren darf (sprich: die Auswahl-Menge ist leer).

Die weiteren Knoten aus dem zweiten Teil der Testkanten („<b:u>+<b:v>“) ergeben sich wie folgt:

(15) t4 := AccessPlace ($places(1), S2, 1)

(16) t5 := AccessPlace ($places(1), S2, 1)

(17) c5 := Selection (t4, t5)

(18) v3 := AccessVariable ($vars(6), u)

(19) v4 := AccessVariable ($vars(6), v)

(20) c6 := ValueRestrict (c5, t4, b, =, a3)

(21) c7 := ValueRestrict (c6, t5, b, =, a4)

(22) b3 := NotEmpty (c7)

„NotEmpty“ in Knoten (22) realisiert in diesem Abschnitt die positive Auswirkung der Testkante, es muß nämlich eine solche Auswahl existieren! Nachdem wir nun die Ergebnisse der beiden disjunktiv verknüpften Teile der Testkantenanschrift ermittelt haben, müssen wir noch die Disjunktion vornehmen und unsere ursprünglich erhaltene Auswahl um diese Bedingung erweitern. Die Disjunktion geschieht durch einen „OR“-Operator, der eine logische Oder-Verknüpfung durchführt.

(23) b4 := OR (b2, b3)

(24) c8 := Condition (c2, b4)

Da wir nun alle Eingangs- und Testkanten sowie das Transitionsprädikat abgehandelt haben, müssen wir noch die Ausgangskanten in unseren Operatorbaum integrieren. Hierzu berechnen wir zunächst für alle Attribute der erzeugten Marken die entsprechenden Werte. In unserem Beispiel ist dies „u-v“ oder „Substract (u, v)“ in Funktionsschreibweise. Die erzeugten Knoten bedürfen keiner weiteren Erläuterung:

(25) v5 := AccessVariable ($vars(6), u)

(26) v6 := AccessVariable ($vars(6), v)

(27) v7 := FunctionCall (Substract, v5, v6)

Bevor nun neue Marken durch das Feuern generiert werden dürfen, muß eine Auswahl gewählt werden, die entsprechenden Marken müssen konsumiert werden. Dies geschieht durch den „Consume“-Operator, der einen neuen Zustand der „$places“-Variablen erzeugt.

(28) $places, v8 := Consume ($places(1), $vars(6), c8, v7)

Nun können die neuen Marken erzeugt werden, wobei jede Markenbeschreibung aus den vorhandenen Werten aufgebaut wird.

(29) n1 := NewToken (Domain_c)

(30) n2 := SetAttribute (n1, c, v8)

(31) $places := InsertToken ($places(28), S3, n2, 1)

Die Knoten (29), (30), (31) beschreiben nacheinander das Erzeugen einer neuen Markenbeschreibung (NewToken) für den angegebenen Definitionsbereich (Domain_c), das Setzen des Attribut „c“ dieser Markenbeschreibung auf den Wert „v8“ (also „u - v“) und das einmalige Einfügen einer Marke, welche dieser Markenbeschreibung genügt, in die Stelle „S3“.

Um den Zustand des PCN nach dem Feuern zu verdeutlichen, wird ein Abschlußknoten erzeugt, welcher den neuen Zustand der Stellen repräsentiert:

(32) result ($places(31), $vars(6))

Aus dem Operatorbaum ergibt sich folgende Belegung der Variablentabelle: „u = t1.a; v = t2.a;“

b) Erzeugung von SQL-Anweisungen aus dem Operatorbaum

Um nun die Verarbeitung dieses Operatorbaums auf das Datenbanksystem verlagern zu können, muß dieser in entsprechende SQL-Anweisungen, die das ORDBMS versteht, übersetzt werden. Dabei können wir zwischen folgenden Aufgaben unterscheiden:

1. Finden der Auswahl und Belegung sowie Ermitteln der Werte für die erzeugten Marken,
2. Löschen der konsumierten Marken,
3. Erzeugen der produzierten Marken.

Da die Punkte 2 und 3 vom Ergebnis aus Punkt 1 abhängig sind, betrachten wir zunächst, wie wir aus dem Operatorbaum eine SQL-Anfrage generieren können, die alle gültigen Auswahlen, Belegungen und daraus resultierende Werte enthält. Da die dazu benötigten Informationen im „Consume“-Knoten des Operatorbaums zusammengefaßt sind, werten wir von diesem ausgehend rekursiv alle referenzierten Knoten aus und attributieren diese mit Informationen, die für den Aufbau der SQL-Anfrage relevant sind. Dabei gibt es Operatoren, welche einfach nur einen textuellen Beitrag leisten (z. B. ein Funktionsaufruf) und solche, die die Struktur der SQL-Anweisung in den einzelnen Klauseln SELECT, FROM und WHERE ergänzen (z. B. der „Selection“-Operator). Als Anker der rekursiven Auswertung dient der „root“-Knoten, der die Rekursion stoppt. Wir beschreiben nun die Umsetzungsregeln der einzelnen Operatoren nach SQL.

Wie schon erwähnt, wird die rekursive Auswertung solange durchgeführt, bis der „root“-Knoten erreicht ist. Dieser liefert lediglich eine leere SELECT-Klausel, welche die Grundlage für die Variablenbindung darstellt. Darauf aufsetzend ist der „AccessPlace“-Operator, der den Zugriff auf Marken realisiert. Da alle Marken in Tabellen der Datenbank repräsentiert sind, liefert der „AccessPlace“-Operator entsprechend einen Beitrag für die SQL-Anfrage. Dabei werden in der FROM-Klausel zunächst für jede Marke zwei Einträge generiert, die diese Marke repräsentieren. Wegen der Trennung von Markenzustand und Markendaten müssen in der WHERE-Klausel die Primärschlüssel (aufgrund der zwischen ihnen bestehenden Fremdschlüsselbeziehung) in eine Wertegleichheitsbeziehung gesetzt werden. Die Identifikation der Marken ergänzt die SELECT-Klausel.

Der „Selection“-Operator wird als nächstes ausgewertet. Seine Aufgabe ist es, die Ergebnisse der „AccessPlace“-Operatoren zu einer Auswahl zu verknüpfen, was zunächst nichts anderes bedeutet, als die einzelnen SELECT-, FROM- und WHERE-Klauseln der referenzierten „AccessPlace“-Operatoren syntaktisch korrekt (und im Falle der WHERE-Klausel mit „AND“ verknüpft) zu konkatenieren. Darüber hinaus muß er die Eindeutigkeit jeder Marke garantieren, so daß die WHERE-Klausel noch um entsprechende „Ungleich“-Forderungen ergänzt wird.

Die durch den „BindVariable“ ausgedrückte Variablenbindung übersetzen wir in eine Umbenennung des entsprechenden Teilausdrucks in der SELECT-Klausel mit Hilfe des „AS“-Konstrukts von SQL, so daß die gebundenen Variablen direkt unter ihrem Namen zugreifbar sind. Genau dies macht sich der „AccessVariable“-Operator zunutze,

der lediglich die textuelle Repräsentation des Variablennamens zum Ergebnis hat. Ähnlich verhält es sich mit dem „FunctionCall“-Operator, bei dem in den Metadaten der PCN die SQL-Repräsentation bereits vorgeben ist, in der lediglich die benötigten Parameter durch ihre textuelle Repräsentation ersetzt werden müssen. Dies ist deshalb möglich, da nur Operatoren, die eine textuelle Repräsentation haben, anstelle der Parameter auftreten können.

Der im Beispiel folgende „Condition“-Knoten stellt eine Bedingung an die bisher zu ermittelnde Ergebnismenge dar, so daß er durch eine einfache Ergänzung der bisherigen SQL-Anfrage in der WHERE-Klausel um das angegebene Prädikat umgesetzt werden kann. Auch der „ValueRestrict“-Knoten stellt lediglich eine solche Ergänzung dar, die jedoch explizit die Vergleichsrelation und den Vergleichswert beinhaltet.

Die beiden Knoten „IsEmpty“ bzw. „NotEmpty“ realisieren einen Mengentest und führen dazu, daß die durch den eingehenden Operator erzeugte SQL-Anfrage textuell expandiert werden muß und komplett in eine „NOT EXISTS ()“- bzw. „EXISTS ()“-Zeichenkette eingebettet wird.

Zur kompletten SQL-Anfrage fehlt uns noch der „Consume“-Operator. Dieser konkateniert die SELECT-, FROM- und WHERE-Klauseln der eingehenden „Selection“ und der Variablenbindung. Ebenso wird die SELECT-Klausel um die für die Ausgangskanten benötigten Funktionswerte ergänzt. Das so erzeugte Ergebnis (Umsetzung des Operatorbaums in Anhang B) wird in eine Zeichenkette überführt, die eigentliche SQL-Anfrage für diese Transition. Für unser Beispiel siht diese folgendermaßen aus:

```
SELECT  ts1.id,ts2.id,tw1.a AS u,tw2.a AS v,Substract (u, v)
FROM    PCN_STATE_TOKEN ts1, Domain_a tw1, PCN_STATE_TOKEN ts2, Domain_a tw2
WHERE   ts1.id = tw1.id AND ts2.id = tw2.id AND ts1.id <> ts2.id AND
        GreaterThan (u, v) AND
        (NOT EXISTS(  SELECT ts3.id
                      FROM   PCN_STATE_TOKEN ts3, Domain_b tw3
                      WHERE  ts3.id = tw3.id AND tw3.b = 4)
        OR EXISTS(SELECT ts4.id,ts5.id
                  FROM   PCN_STATE_TOKEN ts4, Domain_b tw4,
                         PCN_STATE_TOKEN ts5, Domain_b tw5
                  WHERE  ts4.id = tw4.id AND ts5.id = tw5.id AND
                         ts4.id <> ts5.id AND tw4.b = u AND tw5.b = v)
        );
```

c) Verarbeitung des Ergebnisses aus der generierten SQL-Anweisung

Das Ergebnis dieser Anfrage liefert die gültige Auswahl-Menge und zugehörige Belegungen, für die die Transaktion feuern kann. Ebenso sind jeweils noch die Werte aufgeführt, welche für die Markenbeschreibungen in den Ausgangskanten benötigt werden. In unserem Beispiel besteht das Ergebnis aus genau einem Tupel: (ts1.id = 4711, ts2.id = 4712, u = 3, v=2, Substract(u,v) = 1).

Um nun zu feuern, muß lediglich eine Zeile des Anfrageergebnisses ausgewählt werden (in unserem Beispiel die einzige, allgemein aber eine beliebige), für die die Aktionen der Schritte 2 und 3 durchgeführt werden müssen. Dies bedeutet, daß die durch die aufgeführten IDs bestimmten Marken gelöscht werden müssen, was im Beispiel durch

```
DELETE FROM PCN_STATE_TOKEN s WHERE s.id IN (4711, 4712)
```

geschieht. Die Einträge in den Tabellen der Definitionsbereiche werden durch die spezifizierten referentiellen Aktionen „DELETE CASCADE“ automatisch gelöscht.

Weiterhin müssen Marken über die Ausgangskanten erzeugt werden. Der Aufbau der dazu benötigten INSERT-Statements läßt sich aus dem bisher nicht bearbeiteten Teil des Operatorbaums, der auf den „Consume“-Operator folgt, herleiten. Allgemein muß für jede erzeugte Marke eine neue, eindeutige ID erzeugt und ein Eintrag in der Tabelle PCN_STATE_TOKEN angelegt werden. Darüber hinaus ist in der Tabelle des jeweiligen Definitionsbereichs ein Eintrag zu erzeugen, welcher auf den Eintrag in der PCN_STATE_TOKEN-Tabelle verweist und die Werte der Marke trägt.

In unserem Beispiel bedeutet dies, daß folgende INSERT-Statements generiert werden, wobei $NEWID eine neu erzeugte, eindeutige TupelID enthält und $VAL_S den Wert der „Substract“-Funktion darstellt.

```
INSERT INTO PCN_STATE_TOKEN(TID, PLACE) VALUES ($NEWID, ‘S3’)
INSERT INTO Domain_c (TID, c) VALUES ($NEWID, $VAL_S)
```

d) Integration in den Dependency-Controller

Alle erläuterten Verarbeitungsschritte sind in einer Funktion gekapselt, der als Parameter eine bestimmte Transition übergeben wird. Von dieser Transition ausgehend ermittelt sie aus den Metadaten des jeweiligen PCN alle zugehörigen Elemente, wie Kanten mit zugehörigen Kantenanschriften und Stellen mit ihren Definitionsbereichen. Mit

diesen Informationen kann sie dann wie besprochen verfahren, um das Feuern der Transition zu realisieren. Die Integration in das ORDBMS erfolgt in Form einer Java-basierten Implementierung auf Basis von JDBC ([9]) bzw. SQL/J ([10]), die auf einer im DB-Kern integrierten Java-Virtual-Machine ([4]) abläuft.

Um die Konsistenz des PCN-Zustandes zu erhalten, müssen alle beschriebenen Schritte, angefangen von der Überprüfung der Schaltbedingung und den daraus resultierenden Belegungen über das Entfernen der Marken und dem Erzeugen neuer Marken, in einer einzigen Transaktion erfolgen. Nur so kann die Atomarität des Zustandsübergangs gewahrt werden. Mehrere dieser Verarbeitungszykeln können auch in einer Transaktion zusammengefaßt werden, um so die (teuere) Commit-Behandlung des DBMS für jeden Einzelschritt zu umgehen.

IV. Zusammenfassung & Ausblick

In diesem Beitrag haben wir vorgestellt, wie wir einen Basisdienst für Abhängigkeitskontrolle, wie sie in WfMS häufig z. B. zur Kontrollflußsteuerung benutzt wird, in ein ORDBMS integrieren können. Hierfür haben wir eine Variante der Produktnetze – PCN – auf ein DB-Schema abgebildet und eine benutzerdefinierte Routine in das ORDBMS integriert, welche auf Basis dieses DB-Schemas die Animation der PCN unter Ausnutzung von vorhandener DB-Funktionalität (SQL-DML) durchführt. Gerade dies hat die Implementierung erheblich vereinfacht, da sich das Dynamik-Modell auf eine natürliche Weise abbilden läßt.

Die bestehende Realisierung wird sicherlich im Laufe der Zeit Anpassungen erfahren. Das bestehende Schema könnte unter Nutzung der in SQL:1999 ([5]) eingeführten Möglichkeiten zur Strukturierung modifiziert werden, aber auch z. Zt. existierende Einschränkungen bzgl. der Basistypen von Definitionsbereichen könnten durch Hinzunahme von benutzerdefinierten Typen aufgehoben werden.

V. Literatur

[1] Andreas Oberweis; *Geschäftsprozeßmodellierung*; Tutorium anläßlich der BTW´97 in Ulm

[2] K. Jensen, G. Rozenberg (Eds.); *High-level Petri Nets*; Springer-Verlag, 1991

[3] OMG; *UML Notation Guide, Version 1.1*; OMG Document ad/97-08-05, September 1998

[4] Tim Lindholm, Frank Yellin; *The JavaTM Virtual Machine Specification; http://www.javasoft.com/docs/books/vmspec/html/VMSpecTOC.doc.html;* September 1996

[5] A. Eisenberg, J. Melton; *SQL:1999, formerly known as SQL3;* ACM SIGMOD Records, 28(1); März 1999

[6] S. Jablonski, C. Bussler; *Workflow-Management-Systeme – Modellierung und Architektur;* Thomson Publishing, Bonn, 1995

[7] Marcus Sträter; *Integration einer Ablaufsteuerungskomponente in ein objekt-relationales DBMS – Konzeption und Implementierungsaspekte;* Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, Februar 1999

[8] Stefanie Poßner; *Entwicklung eines Prozeßmodells für Workflow-Management-Systeme am Beispiel der Warenwirtschaft in Handelsunternehmen;* Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, Juli 1996

[9] Seth White, Mark Hapner; *JDBC-2.0 API, Version 1.0; http://www.javasoft.com/products/jdbc/jdbc20.ps;* Sun Microsystems, Inc.; Mai 1998

[10] Sybase, Inc.; *SQLJ: SQL Routines using the Java Programming Language: Working Draft; FTP://ftp.calweb.com/business/sqlj/SQLJ-P1.pdf;* Oktober 1998

[11] Heinz Jürgen Burkhardt, Peter Ochsenschläger, Rainer Prinoth; *Product Nets - A Formal Description Technique for Cooperating Systems;* GMD-Studien Nr.165, September 1989

Anhang A. Schema der PCN

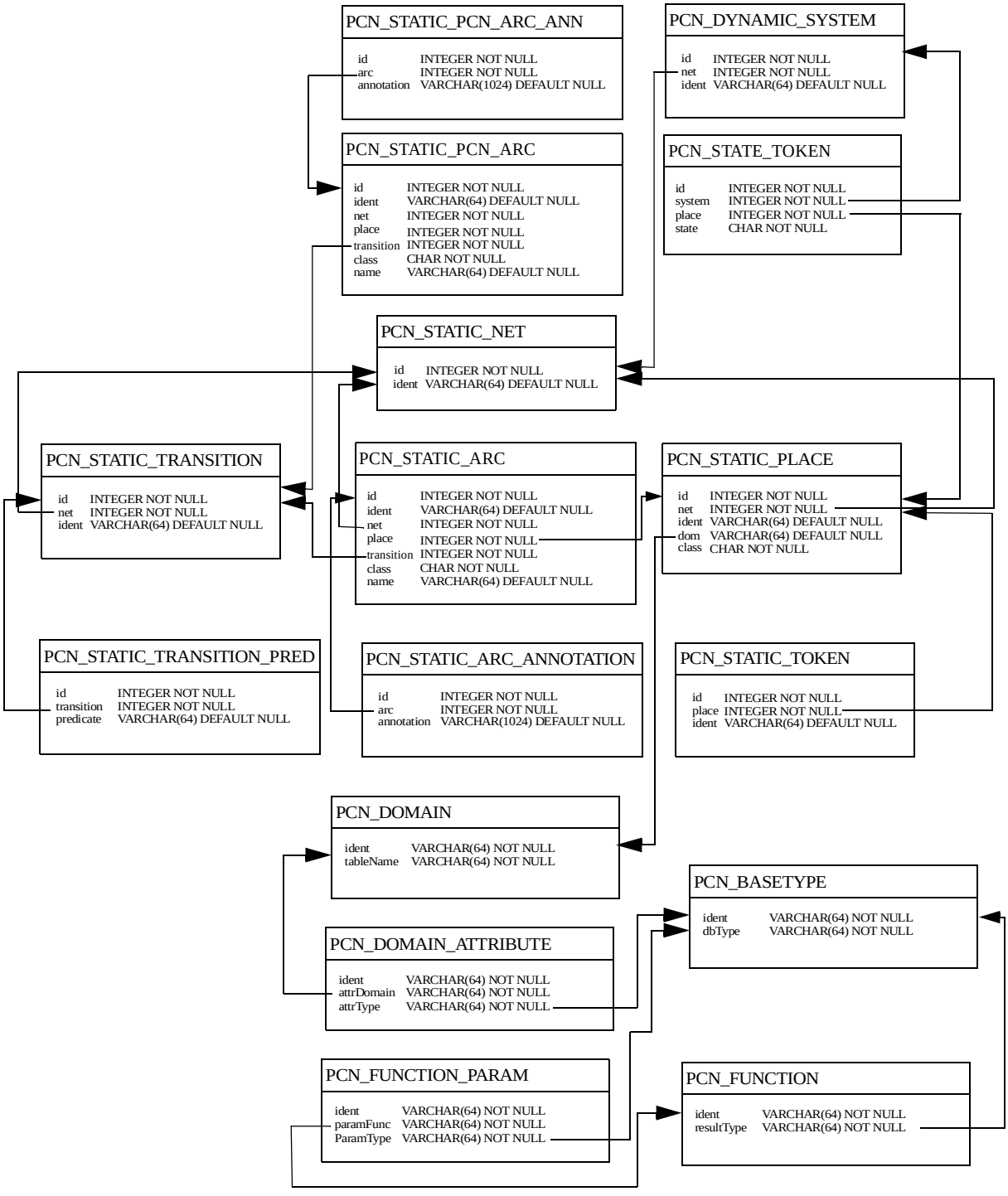

Abb. 6: Schema der PCN

Anhang B. Attributierung des Operatorbaums für die Übersetzung nach SQL

(1) VARTAB = {SELECT: „“}

(2) t1 -> { SELECT: „ts1.id“;
FROM: „PCN_STATE_TOKEN ts1, Domain_a tw1“;
WHERE: „ts1.id = tw1.id AND ts1.place = ‘S1’“}

(3) t2 -> { SELECT: „ts2.id“;
FROM: „PCN_STATE_TOKEN ts2, Domain_a tw2“;
WHERE: „ts2.id = tw2.id AND ts2.place = ‘S1’“}

(4) c1 -> { SELECT: „$ts1.SELECT, $ts2.SELECT“;
FROM: „$t1.FROM, $t2.FROM“,
WHERE: „$t1.WHERE AND $t2.WHERE AND ts1.id <> ts2.id“}

(5) -> VARTAB(1) append {SELECT: „tw1.a AS u“}

(6) -> VARTAB(5) append {SELECT: „tw2.a AS v“}

(7) v1 -> „u“

(8) v2 -> „v“

(9) b1 -> „GreaterThan ($v1, $v2)“

(10) c2 -> c1 append {WHERE: „AND $b1“)}

(11) t3 -> { SELECT: „ts3.id“;
FROM: „PCN_STATE_TOKEN ts3, Domain_b tw3“;
WHERE: „ts3.id = tw3.id AND ts3.place = ‘S2’“}

(12) c3 -> { SELECT: „$ts3.SELECT“;
FROM: „$t3.FROM“;
WHERE: „$t3.WHERE“}

(13) c4 -> c3 append {WHERE: „AND tw3.b = 4“}

(14) b2 -> { WHERE: „NOT EXISTS ($c4)“

(15) t4 -> { SELECT: „ts4.id“;
FROM: „PCN_STATE_TOKEN ts4, Domain_b tw4“;
WHERE: „ts4.id = tw4.id AND ts4.place = ‘S2’“}

(16) t5 -> { SELECT: „ts5.id“;
FROM: „PCN_STATE_TOKEN ts5, Domain_b tw5“;
WHERE: „ts5.id = tw5.id AND ts5.place = ‘S2’“}

(17) c5 -> { SELECT: „$ts4.SELECT, $ts5.SELECT“;
FROM: „$t4.FROM, $t5.FROM“;
WHERE: „$t4.WHERE AND $t5.WHERE AND ts4.id <> ts5.id“

(18) v3 -> „u“

(19) v4 -> „v“

(20) c6 -> c5 append {WHERE: „AND tw4.b = $v3“}

(21) c7 -> c6 append {WHERE: „AND tw5.b = $v4“}

(22) b3 -> { WHERE: „EXISTS ($c7)“}

(23) b4 -> { WHERE: „($b2 OR $b3)“}

(24) c8 -> c2 append {WHERE: „AND $b4“}

(25) v5 -> „u“

(26) v6 -> „v“

(27) v7 -> „Substract ($v5, $v6)“

(28) Consume = c8 append $VARTAB(6) append {SELECT: „$v7“}

Anhang C.