Karsten Schmidt Christian Stahl (Hrsg.)

12. Workshop „Algorithmen und Werkzeuge für Petrinetze“ (AWPN 2005)

29. - 30. September 2005,
Humboldt-Universität zu Berlin,
Proceedings

Humboldt-Universität zu Berlin
Institut für Informatik
Unter den Linden 6
10099 Berlin

Vorwort

Die Workshop-Reihe „Algorithmen und Werkzeuge für Petrinetze“ wurde 1994 mit dem Ziel initiiert, Entwickler und Anwender von petrinetzbasierten Algorithmen und Werkzeugen zusammenzubringen und damit die Abstimmung zwischen Angebot und Nachfrage nach petrinetzbasierter Technologie zu verbessern. Der Workshop bietet Vorträge über Algorithmen, Werkzeuge, Fallstudien sowie Werkzeugdemonstrationen mit Bezug zu Petrinetzen oder verwandten Systemmodellen.

Der Workshop „Algorithmen und Werkzeuge für Petrinetze“ findet 2005 zum zwölften Mal statt. Veranstalter ist wie immer die Fachgruppe 0.0.1 „Petrinetze und verwandte Systemmodelle“ der Gesellschaft für Informatik. Veranstaltungsort in diesem Jahr ist nach 1994 zum zweiten Mal Berlin.

Um auch die Vorstellung noch unfertiger Ideen oder noch in Entwicklung befindlicher Werkzeuge zu ermöglichen, fand wie in den vergangenen Jahren kein formaler Begutachtungsprozess statt. Die eingereichten Beiträge des Workshops wurden aber durch Mitglieder des Steering Commitees der Fachgruppe 0.0.1 auf Relevanz für das Workshop-Thema hin geprüft. Wir denken, dass wir ein interessantes Programm anbieten, welches Anknüpfungspunkte für viele Diskussionen bietet.

August 2005

Karsten Schmidt
Christian Stahl

Steering Committee

Die Mitglieder des Steering Committees der Fachgruppe 0.0.1 „Petrinetze und verwandte Systemmodelle“ der Gesellschaft für Informatik sind:

Jörg Desel, Eichstätt (Leiter)

Karsten Schmidt, Berlin (Stellvertreter)

Gabriel Juhas, Eichstätt

Peter Kemper, Dortmund

Ekkart Kindler, Paderborn

Kurt Lautenbach, Koblenz

Rüdiger Valk, Hamburg

Inhaltsverzeichnis

Eine Charakterisierung einfacher Petrinetz-Schemata 1
A. Glausch

STG Decomposition: Optimised Backtracking and Component Generation 7
M. Schaefer

A Multi Purpose 3D Simulation Tool Based on Discrete Event Systems .. 13
C. Stehno

Petri Nets and the Real World 19
E. Kindler, F. Nillies

A First View on a Generalised Modelling Toolkit for Graph-based Languages 25
S. Phillippi, A. Pinl, G. Rausch

Ein Petrinetzsystem zur Modellierung selbstmodifizierender Petrinetze ... 31
V. Tell, D. Moldt

Model Checking of Bounded Petri Nets Using Interval Diagrams 37
A. Tovchigrechko

Recycling Model Checking Tools for New Application Domains 43
C. Stehno

Unentscheidbarkeit des Beschränktheitsproblems für allgemeine Referenznetze 48
R. Dietze

Schwach beschränkte Petrinetze 54
J. Desel

Reduction Rules for Interaction Graphs 60
D. Weinberg, K. Schmidt

Semantische Annotation von Petri-Netzen 66
A. Koschmider, D. Ried

Modellierung und Analyse transaktionaler Geschäftsprozesse 72
C. Frenkler, K. Schmidt

Operating Guidelines for Services 78
P. Massuthe, K. Schmidt

Service Modeling Based on High-Level Petri Nets 84
M. Köhler, J. Ortmann

Are Visual Methods Mandatory for the Modeling of Business Processes? . 90
C. Simon

Eine Charakterisierung einfacher Petrinetz-Schemata

Andreas Glausch
Humboldt-Universität zu Berlin, Institut für Informatik

Wir untersuchen die Ausdrucksmächtigkeit von Petrinetz-Schemata. Dazu führen wir die Teilklasse der *einfachen Petrinetz-Schemata* ein und zeigen, welche Systeme sich durch ein einfaches Petrinetz-Schema darstellen lassen und welche nicht. Die hinreichenden und notwendigen Eigenschaften dazu formulieren wir in einem Theorem.

1 Einleitung

Petrinetz-Schemata (kurz PN-Schemata) sind eine häufig verwendete Notationsform für verteilte Algorithmen. Sie zeichnen sich dadurch aus, dass sie einerseits eine einfache und verständliche grafische Darstellung besitzen und andererseits auf einer formalen Grundlage mit einer gut ausgebauten Theorie basieren.

Ein PN-Schema besteht aus einem Petrinetz, dessen Transitionen durch logische Ausdrücke und dessen Pfeile durch Terme beschriftet werden. Abbildung 1 stellt drei PN-Schemata grafisch dar. Die Terme werden dabei aus *Funktionssymbolen* und *Variablensymbolen* gebildet. Beispielsweise sind in Abb. 1 f, g und r Funktionssymbole, während x und y Variablensymbole sind. *true* dagegen ist kein Funktions- oder Variablensymbol, sondern bezeichnet den logischen Ausdruck, der immer erfüllt ist. Um in einem PN-Schema die Terme durch konkrete Werte interpretieren zu können, benötigen wir für jedes Funktionssymbol eine Interpretation durch eine konkrete Funktion und für jedes Variablensymbol eine Interpretation durch einen konkreten Wert. Die Interpretation der Variablensymbole kann sich in einem Ablauf eines Schemas ändern, die Interpretation der Funktionssymbole dagegen bleibt konstant. Für eine ausführliche Einführung in PN-Schemata inklusive weiterer Beispiele sei [4] empfohlen.

Wir möchten uns in dieser Arbeit mit der Frage auseinandersetzen, welche Ausdruckskraft PN-Schemata besitzen. [3] beantwortet diese Frage für elementare Petrinetze, d.h. Petrinetze auf deren Plätzen jeweils höchstens eine Marke liegen kann. Dazu wurde die Klasse der *elementaren Transitionssysteme* definiert und anschließend nachgewiesen, dass diese Klasse genau die elementaren Petrinetze charakterisiert. In [5] wurde die-

se Idee auf PN-Schemata ohne Variablen erweitert, indem diese durch die Klasse der *algorithmischen Transitionssysteme* charakterisiert wurden.

In dieser Arbeit stellen wir nun die Charakterisierung einer Variante von PN-Schemata mit Variablen vor, den *einfachen PN-Schemata*. Im Gegensatz zu [3] und [5] betrachten wir dabei auch Markierungen mit Multimengen. Wir werden außerdem für die Charakterisierung nicht Transitionssysteme verwenden, sondern führen den Begriff des *abstrakten Schrittsystems* ein. Ein abstraktes Schrittsystem beschreibt die Zustandübergänge der Markierungen lokal, anders als Transitionssysteme. Damit lassen sich aus einem abstrakten Schrittsystem auch direkt verteilte Abläufe konstruieren.

2 Einfache PN-Schemata

In diesem Abschnitt führen wir die Klasse der einfachen PN-Schemata ein. Dazu werden zunächst einige grundlegende Begriffe definiert. Diese Begriffe sind in der Literatur wohlbekannt und werden aus diesem Grund nicht näher erläutert.

Ein *Netz* $N = (\mathbf{P}, \mathbf{T}, \mathbf{F})$ besteht aus zwei nichtleeren Mengen $\mathbf{P}$ und $\mathbf{T}$ und einer Relation $\mathbf{F} \subseteq (\mathbf{P} \times \mathbf{T}) \cup (\mathbf{T} \times \mathbf{P})$. $\mathbf{P}$ heißt die *Menge der Plätze*, $\mathbf{T}$ die *Menge der Transitionen* und $\mathbf{F}$ die *Flussrelation* von N. Für jedes $x \in \mathbf{P} \cup \mathbf{T}$ bezeichnet $\bullet x := \{y | (y, x) \in \mathbf{F}\}$ den *Vorbereich* und $x\bullet := \{y | (x, y) \in \mathbf{F}\}$ den *Nachbereich* von x in N. Ein Netz $N = (\mathbf{P}, \mathbf{T}, \mathbf{F})$ heißt *endlich*, wenn $\mathbf{P}$ und $\mathbf{T}$ jeweils endlich sind.

Zu einer gegebenen Menge V ist eine Funktion $M : V \to \mathbb{N}$ eine *Multimenge über* V. Die Menge aller Multimengen über V wird mit $\text{Bag}(V)$ bezeichnet.

Eine Signatur $\Sigma = (f_1, \ldots, f_k, n_1, \ldots, n_k)$ besteht aus Funktionssymbolen $f_1, \ldots, f_k$ mit jeweils zugeordneter Stelligkeit $n_1, \ldots, n_k \in \mathbb{N}$. Zusammen mit einer Menge von Variablensymbolen X lassen sich dann induktiv Σ-X-Terme bilden: Jedes 0-stellige Funktionssymbol aus Σ und jedes Variablensymbol aus X ist ein Σ-X-Term; für jedes n-stellige Funktionssymbol f und Σ-X-Terme $t_1, \ldots, t_n$ ist $f(t_1, \ldots, t_n)$ ein Σ-X-Term. Zu einem Σ-X-Term t bezeichnet $\text{var}(t)$ die Menge aller Variablensymbole in t und $|t|$ die Anzahl der Symbole in t. Die Menge aller Σ-X-Terme wird mit $T_\Sigma(X)$ bezeichnet.

Eine Σ-*Algebra* A enthält eine Menge $U(A)$, genannt das *Universum* von A, und für jedes n-stellige Funktionssymbol f aus Σ eine Funktion $f_A : U^n \to U$. Zwei Σ-Algebren A_1, A_2 heißen *isomorph*, wenn es eine Bijektion $\phi : U(A_1) \to U(A_2)$ gibt mit $\phi(f_{A_1}(u_1, \ldots, u_n)) = f_{A_2}(\phi(u_1), \ldots, \phi(u_n))$ für jedes n-stellige Funktionssymbol f aus Σ und alle $u_1, ..., u_n \in U(A_1)$.

Eine Funktion $\alpha : X \to V$ zu einer gegebenen Menge von Variablensymbolen X und einer Menge V heißt *Belegung von X über V*. Sind eine Σ-Algebra A und eine Belegung α von X über $U(A)$ gegeben, so lässt sich jeder Σ-X-Term t zu einem Element $t_{A,\alpha}$ aus $U(A)$ interpretieren: Falls $t \in X$, dann ist $t_{A,\alpha} := \alpha(t)$ und falls t ein 0-stelliges Funktionssymbol aus Σ ist, dann ist $t_{A,\alpha} := t_A$. Ist dagegen $t = f(t_1, \ldots, t_n)$, dann ist $t_{A,\alpha} := f_A(t_{1A,\alpha}, \ldots, t_{nA,\alpha})$.

Aus Σ-X-Termen lassen sich boolesche Σ-X-Ausdrücke bilden: *true* ist ein boolescher Ausdruck und sind t_1, t_2 Σ-X-Terme, so ist $t_1 = t_2$ ein boolescher Σ-X-Ausdruck. Sind e_1, e_2 boolesche Σ-X-Ausdrücke, dann sind $\neg e_1$ und $e_1 \wedge e_2$ ebenfalls boolesche Σ-X-

Ausdrücke. Die *Erfülltheit* eines booleschen Σ-X-Ausdruckes in einer Σ-Algebra A und einer Belegung α von X über $U(A)$ ergibt sich dann folgendermaßen: *true* ist immer erfüllt, $t_1 = t_2$ ist erfüllt gdw $t_{1A,\alpha} = t_{2A,\alpha}$, $\neg e_1$ ist erfüllt gdw e_1 nicht erfüllt ist und $e_1 \wedge e_2$ ist erfüllt gdw e_1 und e_2 erfüllt sind. $\text{var}(e)$ bezeichnet die Menge alle Variablensymbole in e und $E_\Sigma(X)$ bezeichnet die Menge aller booleschen Σ-X-Ausdrücke.

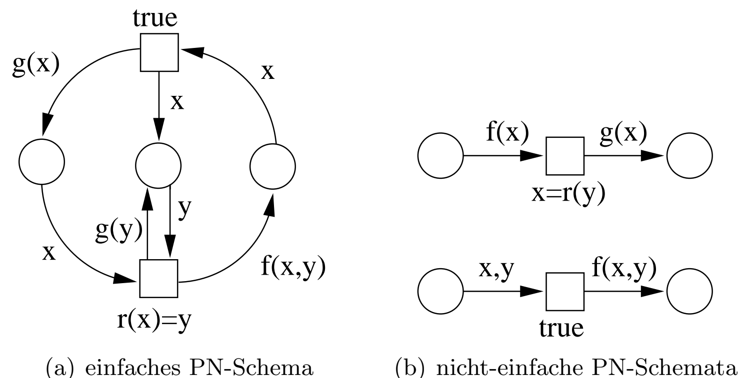
Wir können nun die Klasse der *einfachen PN-Schemata* definieren:

Definition 2.1 (Einfaches PN-Schema)**.** Seien Σ eine Signatur, X eine Menge von Variablensymbolen und $(\mathbf{P}, \mathbf{T}, \mathbf{F})$ ein endliches Netz. Weiter seien $\psi : \mathbf{T} \to E_\Sigma(X)$ und $\omega : \mathbf{F} \to T_\Sigma(X)$ Funktionen, so dass für jede Transition $t \in \mathbf{T}$ gilt: Bezeichne

$$\text{var}(t) := \text{var}(\psi(t)) \cup \bigcup_{p \in \bullet t} \text{var}(\omega(p,t)) \cup \bigcup_{p \in t \bullet} \text{var}(\omega(t,p))$$

die Menge aller Variablensymbole bei t, dann gibt es zu jedem $x \in \text{var}(t)$ ein $p \in \bullet t$ mit $\omega(p,t) = x$ oder ein $p \in t\bullet$ mit $\omega(t,p) = x$. Dann ist $N = (\mathbf{P}, \mathbf{T}, \mathbf{F}, \Sigma, X, \psi, \omega)$ ein *einfaches PN-Schema.*

Ein einfaches PN-Schema ist also ein endliches Netz, dessen Pfeile durch jeweils einen Term und dessen Transitionen durch jeweils einen booleschen Ausdruck beschriftet sind. Zu jeder Variable x, die bei einer Transition t verwendet wird, muss außerdem eine Kante an t mit x beschriftet sein. Abbildung 1(a) enthält ein Beispiel und Abb. 1(b) zwei Gegenbeispiele von einfachen PN-Schemata.



(a) einfaches PN-Schema (b) nicht-einfache PN-Schemata

Abbildung 1: Beispiele von PN-Schemata

3 Semantik einfacher PN-Schemata

Nachdem wir im vorherigen Abschnitt die Syntax einfacher PN-Schemata festgelegt haben, stellen wir nun die zugehörige Semantik vor. Diese entspricht im wesentlichen der klassischen Entfaltungssemantik von PN-Schemata und wurde nur auf die einfachen PN-Schemata angepasst.

Der Zustand eines PN-Schemas wird durch eine *Markierung* beschrieben:

Definition 3.1 (Markierung)**.** Sei $\mathbf{P}$ eine Menge von Plätzen und sei V eine Menge. Dann ist eine Funktion $\mu : \mathbf{P} \to \mathrm{Bag}(V)$ eine *Markierung von* $\mathbf{P}$ *über* V.

Eine Markierung ordnet also jedem Platz eine Multimenge von Elementen einer Menge V zu. Eine Markierung kann durch einen *Schritt* verändert werden:

Definition 3.2 (Schritt)**.** Seien $\mathbf{P}$ eine Menge von Plätzen und V eine Menge. Dann bezeichnet $\mathrm{Var}(\mathbf{P}) = \{p^-, p^+ | p \in \mathbf{P}\}$ die *Menge der Platzvariablen von* $\mathbf{P}$. Eine partielle Funktion $s : \mathrm{Var}(\mathbf{P}) \to V$ ist ein *Schritt von* $\mathbf{P}$ *über* V.

Definition 3.3 (Ausführen eines Schrittes)**.** Seien $\mathbf{P}$ eine Menge von Plätzen und V eine Menge. Seien weiter μ eine Markierung von $\mathbf{P}$ über V und s ein Schritt von $\mathbf{P}$ über V. Falls $\mu(p)(s(p^-)) > 1$ für alle $p^- \in \mathrm{dom}(s)$, dann ist $\mu \oplus s$ eine Markierung von $\mathbf{P}$ über V und ist definiert als:

$$(\mu \oplus s)(p)(u) := \begin{cases} \mu(p)(u) - 1 & \text{, falls } s(p^-) = u \text{ und } s(p^+) \neq u \\ \mu(p)(u) + 1 & \text{, falls } s(p^+) = u \text{ und } s(p^-) \neq u \\ \mu(p)(u) & \text{, sonst.} \end{cases}$$

Ein Schritt s ordnet also einigen Platzvariablen aus $\mathrm{Var}(\mathbf{P})$ einen Wert aus V zu. Ist $s(p^-)$ für einen Platz p definiert, dann wird bei Ausführung von s vom Platz p das Element $s(p^-)$ entfernt. Analog wird, sofern $s(p^+)$ für einen Platz p definiert ist, zum Platz p das Element $s(p^+)$ hinzugefügt. Ein Schritt s fügt zu einem Platz p also immer höchstens ein Element hinzu bzw. entfernt höchstens ein Element.

Wir können nun definieren, welche Schritte ein einfaches PN-Schema erzeugt:

Definition 3.4 (Schritte eines einfachen PN-Schema)**.** Seien $N = (\mathbf{P}, \mathbf{T}, \mathbf{F}, \Sigma, X, \psi, \omega)$ ein einfaches PN-Schema und $t \in \mathbf{T}$. Sei weiterhin A eine Σ-Algebra und α eine Belegung von $\mathbf{P}$ über $U(A)$, so dass $\psi(t)$ von A und α erfüllt wird. Sei dann s der Schritt von $\mathbf{P}$ über $U(A)$ mit:

$$s(p^-) := \begin{cases} \omega(p,t)_{A,\alpha} & \text{, falls } p \in \bullet t \\ \text{undefiniert} & \text{, sonst,} \end{cases}$$

$$s(p^+) := \begin{cases} \omega(t,p)_{A,\alpha} & \text{, falls } p \in t\bullet \\ \text{undefiniert} & \text{, sonst.} \end{cases}$$

Dann ist s der *Schritt von* N *zu* A *und* α *bei* t und wird mit $s_N(A, \alpha, t)$ bezeichnet. Weiterhin ist

$$S_N(A) := \{s_N(A, \alpha, t) | t \in \mathbf{T} \text{ und } A \text{ und } \alpha \text{ erfüllen } \psi(t)\}$$

die *Menge aller Schritte von* N *zur Algebra* A.

Zu einer gegebenen Algebra A erzeugt ein einfaches PN-Schema N also eine Menge von Schritten $S_N(A)$. $S_N(A)$ enthält dabei *alle* Schritte, die N potentiell ausführen kann. Mit Hilfe dieser Schrittmenge können wir sehr einfach *sequentielle Abläufe* konstruieren:

Definition 3.5 (sequentieller Ablauf einer Schrittmenge)**.** Seien $\mathbf{P}$ eine Menge von Plätzen, U eine Menge und S eine Menge von Schritten von $\mathbf{P}$ über U. Eine (endliche oder unendliche) Sequenz (μ_n) von Markierungen von $\mathbf{P}$ über U ist ein *sequentieller Ablauf von* S, wenn es für jeden Index i ein $s \in S$ gibt mit $\mu_{i+1} = \mu_i \oplus s$.

Definition 3.6 (sequentieller Ablauf eines einfachen PN-Schema)**.** Sei $N = (\mathbf{P}, \mathbf{T}, \mathbf{F}, \Sigma, \psi)$ ein PN-Schema, sei A eine Σ-Algebra und sei (μ_n) ein sequentieller Ablauf von $S_N(A)$. Dann ist (μ_n) ein *sequentieller Ablauf von* N.

Aus einer gegebenen Schrittmenge $S_N(A)$ lassen sich auch sehr leicht *verteilte Abläufe* konstruieren [1].

4 Eine Charakterisierung einfacher PN-Schemata

In diesem Abschnitt stellen wir nun eine Charakterisierung einfacher PN-Schemata vor. Einige wichtige Ideen dazu stammen aus Gurevichs Charakterisierung der sequentiellen Algorithmen [2].

Betrachten wir ein PN-Schema N ganz abstrakt, so erzeugt N zu einer gegebenen Algebra A eine Menge von Schritten $S_N(A)$. Systeme dieser Art bezeichnen wir als *Schrittsysteme*:

Definition 4.1 (abstraktes Schrittsystem)**.** Sei Σ eine Signatur und sei $\mathbf{P}$ eine endliche Menge von Plätzen. Sei S eine Funktion, die jeder Σ-Algebra A eine Menge von $U(A)$-Schritten über $\mathbf{P}$ zuweist. Außerdem gelte für zwei isomorphe Σ-Algebren A_1, A_2 mit einem Isomorphismus ϕ, dass $S(A_2) = \{\phi \circ s | s \in S(A_1)\}$. Dann ist $\mathcal{A} = (\mathbf{P}, \Sigma, S)$ ein *abstraktes Schrittsystem.*

Ein abstraktes Schrittsystem $\mathcal{A}$ erzeugt also zu jeder Σ-Algebra eine Menge von Schritten. Sind außerdem zwei Σ-Algebren isomorph, so erzeugt $\mathcal{A}$ entsprechend isomorphe Schrittmengen. Völlig analog zu den PN-Schemata lässt sich dann für abstrakte Schrittsysteme ein Ablaufbegriff definieren:

Definition 4.2 (sequentieller Ablauf eines abstrakten Schrittsystems)**.** Sei $\mathcal{A} = (\mathbf{P}, \Sigma, S)$ ein abstraktes Schrittsystem, sei A eine Σ-Algebra und sei (μ_n) ein sequentieller Ablauf von $S(A)$. Dann ist (μ_n) ein *sequentieller Ablauf von* $\mathcal{A}$.

Nicht jedes abstrakte Schrittsystem kann durch ein einfaches PN-Schema dargestellt werden. Lässt man z.B. anstelle der booleschen Ausdrücke auch prädikatenlogische Ausdrücke als Beschriftung der Transitionen zu, so entsteht eine ausdrucksstärkere Klasse von abstrakten Schrittsystemen. Wir definieren aus diesem Grund folgende eingeschränkte Klasse abstrakter Schrittsysteme:

Definition 4.3 (beschränkt-abstraktes Schrittsystem)**.** Sei $\mathcal{A} = (\mathbf{P}, \Sigma, S)$ ein abstraktes Schrittsystem und sei $c \in \mathbb{N}$, so dass für alle Σ-Algebren A_1, A_2 gilt: Falls $s \in S(A_1)$ und $t_{A_1,s} = t_{A_2,s}$ für alle $t \in T_\Sigma(\mathrm{dom}(s))$ mit $|t| \leq c$, dann ist $s \in S(A_2)$. Dann ist $\mathcal{A}$ ein *beschränkt-abstraktes Schrittsystem.*

Um zu entscheiden, ob ein Schritt s in der Schrittmenge $S(A)$ einer Algebra A liegt, darf ein beschränkt-abstrakten Schrittsystem $\mathcal{A}$ nur Terme mit einer maximalen Länge von c interpretieren. Bei der Interpretation dieser Terme kann $\mathcal{A}$ den Schritt s als Variablenbelegung verwenden. Eine ausführlichere Erläuterung dazu befindet sich in [1].

Das folgende Theorem stellt nun die eigentliche Charakterisierung dar:

Theorem 4.1. *Sei $\mathcal{A} = (\mathbf{P}, \Sigma, S_{\mathcal{A}})$ ein beschränkt-abstraktes Schrittsystem. Dann gibt es ein einfaches PN-Schema N, so dass $S_N(A) = S_{\mathcal{A}}(A)$ für alle Σ-Algebren A.*

Jedes beschränkt-abstrakte Schrittsystem lässt sich also vollständig durch ein einfaches PN-Schema darstellen, d.h. sie erzeugen zu jeder Algebra gleiche Schrittmengen und damit auch gleiche sequentielle (und verteilte) Abläufe. Umgekehrt ist jedes einfaches PN-Schema N mit der zugehörigen Funktion S_N auch ein beschränkt-abstraktes Schrittsystem. Der Beweis dazu und zu Theorem 4.1 ist in [1] zu finden.

5 Zusammenfassung

Wir haben mit der Klasse der *einfachen PN-Schemata* eine syntaktisch einfache Teilmenge der klassischen PN-Schemata identifiziert. Die Semantik eines einfachen PN-Schemas haben wir dann als eine Abbildung realisiert, die jeder Σ-Algebra eine Menge von *Schritten* zuordnet. Anschließend haben wir ganz allgemein Systeme mit einer solchen Abbildung, von uns *abstrakte Schrittsysteme* genannt, untersucht, um die Ausdrucksmächtigkeit der einfachen PN-Schemata zu bestimmen. Als Resultat haben wir in einem Theorem eine Eigenschaft formuliert, die genau diejenigen abstrakten Schrittsysteme beschreibt, die sich durch ein einfaches PN-Schema darstellen lassen.

Verallgemeinerungen dieses Theorems auf PN-Schemata mit Multitermen oder prädikatenlogischen Ausdrücken scheint nur eine Frage des technischen Aufwands zu sein und ist Bestandteil zukünftiger Forschung.

Literatur

[1] A. Glausch. A Characterization of Simple Petri Net Schemata. Informatik-Bericht, Humboldt-Universität zu Berlin, 2005. erscheint im September.

[2] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, Juli 2000.

[3] M. Nielsen, G. Rozenberg, and P. S. Thiagarajan. Elementary transition systems. *Theor. Comput. Sci.*, 96(1):3–33, 1992.

[4] W. Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer-Verlag, 1998.

[5] W. Reisig. On the Expressive Power of Petri Net Schemata. In *ICATPN*, pages 349–364, 2005.

STG Decomposition: Optimised Backtracking and Component Generation*

Mark Schaefer

University of Augsburg, Germany
mark.schaefer@informatik.uni-augsburg.de

1 Introduction

Asynchronous circuits are a promising type of digital circuits. They perform better, use less energy and emit less radiation than conventional synchronous circuits. A widely used formalism for their modelling are *signal transition graphs* or *STGs*, which are interpreted Petri nets.

The main drawback of this model is the inefficient and complex synthesis into real-life circuits; for this, the reachability graph of an STG is needed, which could lead to state explosion and - even worse - the synthesis needs an effort which is at least quadratic in size of this reachability graph.

One way to avoid this, is to decompose an STG into several smaller ones which perform together in the same way as the original one. The advantages are a faster synthesis and a reduced peak memory usage. This paper deals with the decomposition method of [VW02,VK05]. In particular, four methods to improve the efficiency and quality of the components are introduced and discussed.

The next section gives a condensed overview of the field of asynchronous circuits and STGs. The third section deals with the original decomposition algorithm and introduces the new methods. The paper ends with the results of some benchmark examples and their discussion.

2 Circuits and STGs

A *signal* is a model of a physical wire, it has a boolean value of *0* or *1* depending on the interpretation of the voltage of the wire. In the following we will abstract from the physical reality and only talk about signals and boolean values.

A *signal edge* is a change of the value of a signal, either from 0 to 1 called *rising edge* and denoted by a '+' after the signal name, or from 1 to 0 called *falling edge* and denoted by a '−'.

An *asynchronous circuit* or just *circuit* is an electrical device with *input* signals which are controlled by the *environment* of the circuit and *output* signals which are controlled by the circuit itself; *internal* signals are output signals, which are not observed by the environment, e.g. signals for internal communication. A circuit calculates a boolean function depending on its input *and* (usually) its output signals. This function is a sufficient description of the circuit. Normally, a circuit is built up of some elementary circuits – called *gates* – which calculate basic boolean functions like *and*, *not* or *xor*. Every output of a gate is an output of the circuit, either a real one or an internal one.

An STG may contain transitions labelled with λ called *dummy* transitions. They are a design simplification and describe no physical reality, but they play an important part in our decomposition algorithm. To *lambdarise* a transition means to change its label to λ, to *delambdarise* means to change the label back to the initial one.

To keep the notation short, input/output/internal signal edges are just called input/output/internal edge. The set of transitions labelled with a certain signal is sometimes identified with the signal itself, e.g. lambdarising signal a means to change the label of all transitions labelled with a to λ.

Synthesis is the calculation of a function describing a circuit from a formal behavioural description under observance of some *(timing) constraints*. We use the *speed-independent* model with the following properties:

* This work was partially supported by the DFG-project 'STG-Dekomposition' Vo615/7-1.

– Input and outputs edges can occur arbitrarily mixed in time.
– Signals (wires) are considered to have no delay, i.e. a signal edge is received immediately by all listeners.
– The circuit must work properly according to its formal description under arbitrarily delays of each gate.

An *STG* is an interpreted Petri net, which describes the behaviour of an asynchronous circuit *and* assumptions on the environment (Figure 1). The transitions are labelled with signals edges, the interpretation is as follows:

– The firing rule is as usual.
– If, and only if a transition labelled with an input signal is activated, the circuit described by the net must be ready to receive this signal edge from the environment.
– If, and only if a transition labelled with an output/internal edge is activated, the circuit described by the net must produce this signal edge.

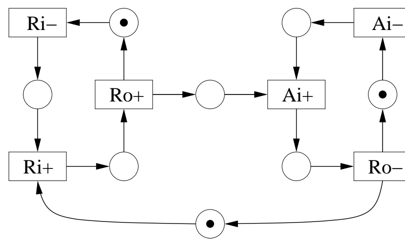


Fig. 1. Example of an STG: `pastor`. Inputs: `Ai, Ri` - Output: `Ro`

An STG can model more behaviour than a real-life circuit can show. The most important problem are *dynamic conflicts*, i.e. two transitions of an STG are enabled under some marking, and firing one would disable the other. This is a form of non-determinism, which in most cases cannot be handled by a digital circuit. There are three problematic cases. 1. One transition is labelled with an input edge, the other with an output edge, 2. both transitions are labelled with an output edge and 3. an *auto-conflict*, i.e. both transitions are labelled with the same signal.

1. This conflict is very hard to implement, since both signal edges are independently generated and may occur at the same time. 2. A circuit which can handle such conflicts is called *arbiter*, it cannot be implemented purely digital. STGs with such conflicts can be handled by our method and new conflicts are not introduced. For a detailed discussion see [VW02,VK05]. 3. This is a non-deterministic choice, which can hardly be handled by circuits. During our decomposition algorithm we consider a newly generated auto-conflict as an indication that to many signals were lambdarised in an STG.

However, to detect dynamic conflicts one has to generate the reachability graph, which we want to avoid. Instead, we look for *structural auto-conflicts*, i.e. two transitions with a common place in their presets. This is a necessary precondition for dynamic conflicts, which can be checked structurally.

On the other hand the decomposition algorithm of [VW02,VK05] makes it possible to ignore structural auto-conflicts, i.e. to consider them not as indications for a dynamic auto-conflicts. The first approach is called *conservative*, the latter one is called *risky*. Despite of its name the risky approach is quite sensible: most structural conflicts are very often only this, and not a dynamic conflict, which leads to unnecessary backtracking when using the conservative method; furthermore, the decomposition algorithm preserves dynamic auto-conflicts, thus accidently generated ones will be detected by the synthesis tool and no erroneous circuit will be generated.

For a detailed description of the decomposition process see [VW02,VK05]. For this paper it is only important to know that we start with a collection of STGs called (*initial components*), each of them a copy of the original STG N with some lambdarised signals and some former output signals being considered as input signals. The following operations are applied to each component:

– *Secure contractions* of dummy transitions

– Deletion of redundant places and redundant transitions
– *Backtracking*

The contraction of a transition t generates a set of new places $\{(p,q)|p \in {}^\bullet t, q \in t^\bullet\}$ (each one of them inherits the tokens and arcs of its 'inner' places) and removes t from the net. Contractions are only performed if they are secure (implying language preservation) and no new structural auto-conflict is generated.

Backtracking means to delambdarise a signal of the initial component, to consider it as an input signal and to start decomposition anew. This is applied if there are still dummy transitions left but none of the other operations can be performed. This situation is considered as an indication that too many signals of a component were lambdarised to produce their output signals appropriately; this can be changed by adding another input signal and – informally speaking – providing more information to the circuit.

The decomposition algorithm itself is non-deterministic. Although, for some examples the order of operations is crucial for the final result in terms of the size or number of added signals. The question is how to find a good order of operations to get the best possible result. Furthermore, backtracking means to undo all operations performed so far, which is very inefficient and the question is whether this is really needed. Viewing the decomposition of all components together, a lot of work is done several times and the question is whether it is possible to reuse intermediate results for the decomposition of other components. Answers to this questions are given in the next section.

3 Optimised Decomposition Algorithms

3.1 Version 2 - Order Transition Contractions

A very simple way to improve the results of the original decomposition arises from performing decompositions by hand. To keep this simple, one usually contracts those transitions first, which generate the smallest number of new places. It turned out to be a good heuristic also for the automatic decomposition to contract transitions in this order; sorting is only performed once at the beginning, the order is not updated during the algorithm.

3.2 Version 3 - Lazy Backtracking

In the original implementation backtracking was performed by restarting the calculation of a component at their initial component. Of course this method is quite natural and plays an important part in the proof of correctness in [VW02,VK05]. On the other hand, it can obviously be rather inefficient: if for example the transitions are contracted by grouping them by their former signals and backtracking has to be performed during the last group, it is unnecessary to start from the very beginning. Instead, it is possible to use the last suitable intermediate result. The algorithm works as in Figure 2.

$$N \overset{\lambda}{\Longrightarrow} N_0 \overset{a_0}{\longrightarrow} N_1 \overset{a_1}{\longrightarrow} \cdots \underbrace{N_k}_{\text{no conflict}} \overset{a_k}{\longrightarrow} \underbrace{N_{k+1} \overset{a_{k+1}}{\longrightarrow} \cdots N_{i-1} \overset{a_{i-1}}{\longrightarrow} N_i}_{\text{conflict}} \underset{\text{for } a_i}{\overset{a_i}{\longrightarrow}}$$

Fig. 2. Backtracking of Version 3

Starting from N, all initially useless signals are lambdarised yielding STG N_0, but instead of contracting them in an arbitrary order choose a signal a_0 and try to contract it completely. If this is possible, save the resulting STG N_1 and proceed with the next signal a_1 and so on. If for some STG N_i the contraction of signal a_i is not possible, delambdarise it in N_i and look for an auto-conflict of a_i. If there is no such conflict, proceed with a new signal a'_i.

If there is a conflict, one has to find the signal whose contraction caused it. To do this, consider N_{i-1} with a_i delambdarised: if the same conflicts exits here, go back to N_{i-2} and so on. If finally, in

STG N_k the conflict disappears, it is obvious that the contraction of signal a_k caused the conflict for signal a_i and has also to be delambdarised.

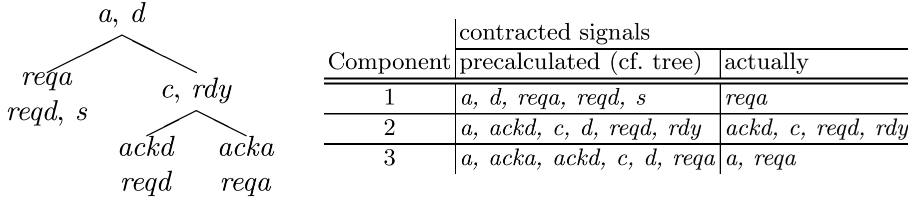
Now look for an auto-conflict for a_k in N_k. If there is none start from N_k with a new dummy signal a'_k. Otherwise, find the signal whose contraction caused this new conflict and so on. Backtracking stops, if an STG N_l is reached without an auto-conflict of the delambdarised signals is reached or N_0, which means that N initially contains an auto-conflict for the last delambdarised signal.

3.3 Version 4 - Tree Decomposition

The methods described so far are improvements for the decomposition of a single component. This section deals with a method for improving the *overall* efficiency of the decomposition of all components.

If we take a look at examples of decomposition, it becomes clear that in most cases two components do not have many signals in common. Therefore the existence of an intermediate STG N' should be possible, from which two or more components could be derived. Instead of performing the decomposition until N' for every component, it is sufficient to do this only once and to proceed separately with each component after the generation of N', thus saving a lot of work.

This can be generalised: if we have found a common ancestor N' of two components, maybe there is a common ancestor N'' of N' and a another component and so on. In general, decomposition is performed according to a *decomposition tree*, i.e. a tree in which each node u contains a set of signals $s(u)$ which should be contracted there (Figure 3, left).



Component	contracted signals	
	precalculated (cf. tree)	actually
1	*a, d, reqa, reqd, s*	*reqa*
2	*a, ackd, c, d, reqd, rdy*	*ackd, c, reqd, rdy*
3	*a, acka, ackd, c, d, reqa*	*a, reqa*

Fig. 3. Left: the precalculated decomposition tree for `locked2`, right: actually contracted signals, components are numbered from left to right

Tree decomposition works as follows: start with the initial STG N without lambdarised signals at the root node. Whenever entering a node u with an STG N' lambdarise the signals $s(u)$ perform decomposition as usual. If there are no conflicts enter each child node with its own copy of the resulting STG. If u is a leaf a component is finished.

If some signal $a \in s(u)$ could not be contracted in N', it is postponed, i.e. the signal a is added to every child node of u (if there are any). This is reasonable, because the contraction of a may have caused an auto-conflict for a signal a', which is lambdarised deeper in the tree. After a' is eventually contracted the contraction of a could be possible.

Using this method, the final components contain less signals. On the other hand, the precalculated tree might not reflect any longer the signals contained in the final components: in the table in Figure 3, right, one can see which signals were actually contracted during tree decomposition, e.g. the signal d – although scheduled for contraction in the root node – is contained in every final component.

Observe that – in contrast to lazy backtracking – once the decomposition of a node is finished, it is not necessary to come back to this node and to delambdarise additional signals. Since signals are lambdarised just in time when entering a node, there are no dummy transitions left and every potential auto-conflict has become visible.

A decomposition tree is a special case of a *preset tree* [KK01]. Finding an optimal preset tree is NP-complete, but in [KK01] a heuristic algorithm is described which performs reasonably well. We use this algorithm for the calculation of decomposition trees.

4 Results

In this section the results of some benchmark examples circulating in the STG community can be found. They were made with the tool DesiJ, which can work in a commandline mode and also provides a graphical user interface for interactive decomposition and STG editing. The main purpose for its development was to provide an easy-to-use decomposition tool and an easy-to-extend STG/decomposition framework, the latter guaranteed by a strictly object-oriented design. DesiJ and a collection of benchmark examples can be downloaded from http://www.informatik.uni-augsburg.de/lehrstuehle/swt/ti/mitarbeiter/mark/projekte/desij.

Each decomposition algorithm was tested with the conservative and the risky auto-conflict detection, the results are listed in Table 1. In columns labelled with '$\sum$' the overall number of signals of all final components is printed, in the 'D' columns (only for 'risky') the overall number of signals for which an undetected dynamic auto-conflict exists in the final components can be found.

For most STGs the risky conflict detection is not very successful, i.e. there are dynamic auto-conflicts in the final components. Since the times of this approach and the conservative method do not vary much, the risky method seems to be inappropriate. For the conservative method the best time and the smallest number of signals are printed bold.

As one can see, in the very most cases version 4 performs best, and if not, the differences are very small. Only in case 58 version 4 uses twice the time of the best method version 3. Furthermore, in 50 of 67 cases version 3 returns components which were minimal in the number of signals. If this is not the case, usually version 3 returns the smallest components.

Only for STG 48 the basic decomposition algorithm is significantly better than version 2. In cases 59-67 (which are very small STGs) it is some 1/1000 sec faster, probably because of the overhead of sorting the transitions first, but normally version 2 is several times faster. (Up to 4400% for case 22.)

Summing it up, the clear winner is version 4 – tree decomposition. followed by version 3 – lazy backtracking, furthermore in three examples only version 4 was able to finish decomposition, the other algorithms terminated abnormally due to lack of memory. The risky conflict detection turned out be useless in most cases while not saving much time.

	Version 1					Version 2					Version 3					Version 4				
	Conservative		Risky			Conservative		Risky			Conservative		Risky			Conservative		Risky		
Nr.	time	$\sum$	time	$\sum$	D	time	$\sum$	time	$\sum$	D	time	$\sum$	time	$\sum$	D	time	$\sum$	time	$\sum$	D
1	1.071	44	0.948	42	2	0.572	**34**	0.501	32	2	0.755	42	0.723	40	2	0.28	**34**	0.275	32	2
2	1.966	54	1.601	52	2	0.549	**32**	0.488	30	2	0.903	54	0.865	52	2	**0.292**	**32**	0.287	30	2
3	3.579	**52**	3.148	50	2	3.067	**52**	2.644	50	2	4.996	80	4.703	78	2	**0.696**	**52**	0.665	50	2
4	13.39	96	12.649	94	2	2.838	**50**	2.335	48	2	5.545	96	5.377	94	2	**0.849**	64	0.973	62	2
5	15.364	**70**	13.958	68	2	13.902	**70**	12.778	68	2	18.484	128	18.42	126	2	**2.424**	90	2.408	88	2
6	71.515	138	66.113	136	2	13.249	**68**	11.933	66	2	21.815	138	22.642	136	2	**2.833**	94	2.754	92	2
7	168.292	133	163.978	131	2	47.709	**88**	43.209	86	2	81.215	156	79.781	154	2	**5.754**	94	5.511	92	2
8	302.317	180	304.549	178	2	44.678	**86**	40.763	84	2	114.545	180	111.145	178	2	**9.32**	104	9.229	102	2
9	0.285	**19**	0.276	19	0	0.266	**19**	0.262	19	0	0.324	**19**	0.32	19	0	**0.204**	**19**	0.2	19	0
10	0.264	**19**	0.263	19	0	0.246	**19**	0.241	19	0	0.292	**19**	0.285	19	0	**0.2**	**19**	0.193	19	0
11	1.997	**37**	1.932	37	0	1.923	**37**	1.788	37	0	2.532	**37**	2.46	37	0	**0.553**	**37**	0.531	37	0
12	1.694	**37**	1.629	37	0	1.43	**37**	1.378	37	0	2.135	**37**	2.149	37	0	**0.481**	**37**	0.475	37	0
13	13.709	**55**	13.791	55	0	9.171	**55**	9.246	55	0	14.86	**55**	14.828	55	0	**1.968**	**55**	1.87	55	0
14	11.412	**55**	10.904	55	0	6.316	**55**	6.198	55	0	12.591	**55**	12.613	55	0	**1.427**	**55**	1.441	55	0
15	4.743	79	3.96	73	3	1.821	**53**	1.369	47	3	2.11	70	2.194	64	3	**0.474**	**53**	0.459	47	3
16	21.843	101	20.627	95	3	1.657	**50**	1.185	44	3	2.919	101	2.783	95	3	**0.48**	**50**	0.458	44	3
17	27.947	109	25.369	103	3	11.029	**80**	9.142	74	3	20.713	160	19.689	154	3	**1.672**	**80**	1.416	74	3
18	260.215	182	257.227	176	3	10.218	**77**	8.656	71	3	18.222	182	18.399	176	3	**1.489**	**77**	1.399	71	3
19	55.508	**107**	46.604	101	3	48.645	**107**	39.758	101	3	65.914	172	66.776	166	3	**4.275**	**107**	4.224	101	3
20	781.511	263	779.179	257	3	44.359	**104**	36.89	98	3	89.668	263	90.173	257	3	**4.169**	**104**	4.119	98	3
21	205.74	**134**	180.301	128	3	175.305	**134**	153.0	128	3	363.08	305	362.77	299	3	**27.648**	172	26.989	166	3
22	7066.425	344	7016.173	338	3	155.739	**131**	134.763	125	3	309.4	344	309.839	338	3	**27.389**	167	27.837	161	3

	Version 1					Version 2					Version 3					Version 4				
	Conservative		Risky			Conservative		Risky			Conservative		Risky			Conservative		Risky		
Nr.	time	$\sum$	time	$\sum$	D	time	$\sum$	time	$\sum$	D	time	$\sum$	time	$\sum$	D	time	$\sum$	time	$\sum$	D
23	0.785	**28**	0.763	28	0	0.586	**28**	0.563	28	0	1.129	**28**	1.096	28	0	**0.301**	**28**	0.3	28	0
24	0.489	**28**	0.478	28	0	0.442	**28**	0.43	28	0	0.71	**28**	0.695	28	0	**0.269**	**28**	0.269	28	0
25	9.437	**55**	9.551	55	0	7.026	**55**	6.905	55	0	18.996	**55**	18.755	55	0	**1.157**	**55**	1.157	55	0
26	5.856	**55**	5.778	55	0	4.192	**55**	4.126	55	0	9.414	**55**	9.145	55	0	**0.882**	**55**	0.884	55	0
27	82.646	**82**	82.86	82	0	50.081	**82**	50.114	82	0	152.07	**82**	147.176	82	0	**6.779**	**82**	6.591	82	0
28	32.764	**82**	32.979	82	0	25.421	**82**	25.427	82	0	70.359	**82**	69.205	82	0	**10.232**	**82**	10.063	82	0
29	56.114	164	35.746	132	44	48.71	158	26.019	119	34	13.985	149	12.619	113	20	**12.159**	**141**	8.258	122	22
30	57.036	164	37.767	135	41	48.286	157	26.98	121	34	14.623	**154**	12.934	126	27	**11.842**	155	7.514	135	26
31	58.771	165	43.174	136	41	50.288	159	27.745	121	34	**14.707**	**155**	13.094	126	27	24.401	**155**	16.302	135	30
32	30.815	153	24.173	135	19	28.016	145	19.213	122	19	9.286	138	9.256	125	16	**3.259**	**133**	2.869	127	16
33	18.819	121	14.506	108	11	16.536	104	11.347	98	9	5.865	108	5.322	101	15	**2.845**	**100**	2.191	92	9
34	29.211	143	21.848	123	15	24.751	132	19.256	117	12	8.271	143	7.458	128	11	**5.196**	**129**	3.929	117	12
35	39.032	166	30.946	147	16	37.268	160	26.034	135	20	11.456	**142**	10.914	133	19	**5.24**	145	4.084	124	22
36	56.383	164	37.337	135	41	48.535	157	27.208	121	34	14.632	**154**	12.897	126	27	**11.872**	155	7.365	135	26
37	55.51	164	35.433	132	44	48.677	158	26.047	119	34	14.091	149	12.625	113	20	**12.103**	**141**	8.109	122	22
38	31.198	153	23.851	135	19	27.8	145	19.296	122	19	9.373	138	9.667	125	16	**3.153**	**133**	2.839	127	16
39	18.185	121	14.277	108	11	16.845	104	11.466	98	9	5.933	108	5.355	101	15	**3.219**	**100**	2.17	92	9
40	**0.102**	**5**	0.099	5	0	0.105	**5**	0.101	5	0	0.108	**5**	0.106	5	0	0.111	**5**	0.107	5	0
41	33.461	113	29.387	104	9	27.772	106	23.371	96	5	16.89	**101**	7.859	97	9	**6.689**	**101**	6.013	96	5
42	49.231	**93**	39.884	89	67	41.266	**93**	36.217	89	66	**13.841**	**93**	8.004	89	66	27.301	**93**	21.751	89	66
43	20.351	104	18.363	98	6	16.527	**92**	15.175	90	2	11.92	100	5.297	91	8	**5.127**	**92**	4.913	90	2
44	63.778	143	67.477	129	7	42.442	**134**	41.086	131	2	26.692	141	14.849	133	0	**15.546**	138	14.514	132	1
45																**26.046**	**108**	21.393	103	60
46	53.593	136	53.325	123	8	37.982	**129**	35.929	125	4	21.116	**129**	11.458	125	5	**13.881**	130	12.715	123	3
47																**19.603**	**110**	15.434	102	60
48	99.457	**171**	70.541	148	18	77.255	176	61.817	159	5	43.174	178	19.685	148	19	**18.461**	172	17.48	160	2
49	134.895	210	103.237	164	15	101.007	203	78.867	171	6	58.86	**193**	25.456	171	17	**22.597**	195	20.737	182	6
50	149.996	210	122.754	164	15	102.479	203	79.521	171	6	56.514	**186**	22.461	165	17	**22.995**	195	21.129	182	6
51	147.185	210	123.186	164	15	101.057	203	78.974	171	6	56.18	**187**	22.422	166	17	**22.618**	195	20.613	182	6
52	169.8	229	144.339	178	25	136.941	214	111.125	182	6	66.721	209	26.492	177	20	**29.962**	**204**	26.574	192	5
53	178.68	229	144.877	178	25	132.072	214	104.849	182	6	66.748	209	26.602	177	20	**33.974**	**206**	32.649	196	3
54	50.773	136	51.223	124	8	36.776	129	34.856	125	4	21.022	129	11.127	125	5	**10.835**	**128**	10.158	124	4
55																**19.462**	**112**	14.604	99	59
56	134.065	210	103.097	164	15	101.257	203	79.139	171	6	58.783	**193**	25.757	171	17	**22.507**	195	20.805	182	6
57	33.674	113	29.791	104	9	27.528	106	23.16	96	5	16.856	**101**	7.816	97	9	**6.8**	**101**	6.03	96	5
58	48.199	**93**	39.234	89	67	41.095	**93**	36.062	89	66	**13.763**	**93**	7.904	89	66	27.549	**93**	21.748	89	66
59	**0.134**	**13**	0.122	12	1	0.137	**13**	0.119	12	1	0.136	**13**	0.133	12	1	0.14	**13**	0.126	12	1
60	0.435	**26**	0.476	26	0	0.421	**26**	0.411	26	0	0.589	**26**	0.571	26	0	**0.28**	**26**	0.279	26	0
61	0.19	**17**	0.186	17	0	0.347	**17**	0.185	17	0	0.226	**17**	0.221	17	0	**0.18**	**17**	0.175	17	0
62	**0.131**	**8**	0.126	8	0	0.133	**8**	0.127	8	0	**0.131**	**8**	0.129	8	0	0.167	**8**	0.135	8	0
63	0.225	**18**	0.222	18	0	0.218	**18**	0.214	18	0	0.275	**18**	0.268	18	0	**0.195**	**18**	0.191	18	0
64	0.23	**22**	0.226	22	0	0.231	**22**	0.223	22	0	0.318	**22**	0.296	22	0	**0.214**	**22**	0.207	22	0
65	**0.131**	**13**	0.13	13	0	0.132	**13**	0.132	13	0	0.142	**13**	0.139	13	0	0.136	**13**	0.133	13	0
66	0.492	**20**	0.366	20	4	0.384	**20**	0.375	20	4	0.315	**20**	0.308	20	4	**0.3**	**20**	0.301	20	4
67	0.296	19	0.231	17	1	0.299	19	0.233	17	1	0.268	**18**	0.263	17	1	**0.226**	19	0.208	17	1

Table 1: Results for some benchmark examples.

References

[KK01] V. Khomenko, , and M. Koutny. Towards an efficient algorithm for unfolding petri nets. In K.G. Larsen and M. Nielsen, editors, *CONCUR 2001*, Lect. Notes Comp. Sci. 2154, 2001.

[VK05] W. Vogler and B. Kangsah. Improved decomposition of signal transition graphs. In *ACSD 2005*, 2005.

[VW02] W. Vogler and R. Wollowski. Decomposition in asynchronous circuit design. In J. Cortadella et al., editors, *Concurrency and Hardware Design*, Lect. Notes Comp. Sci. 2549, 152 – 190. Springer, 2002.

A Multi Purpose 3D Simulation Tool Based on Discrete Event Systems

Christian Stehno

Parallel Systems Group, Department for Computing Science,
University of Oldenburg, Germany
stehno@informatik.uni-oldenburg.de

Abstract. We present the P-UMLaut tool, a versatile tool for animating and testing system behaviour. It provides a 3D visualization environment used to interactively explore possible system behaviour, and observe automatically simulated system evolution for testing purposes. The underlying simulation framework connects simulation engines with visualization frontends using translation maps in order to map event names, and network APIs to support a distributed application setup.
As an example for the application of the tool the simulation is using Petri nets created from UML 2.0 sequences diagrams, but the tool is not limited to such kind of descriptions.

1 Introduction

The use of formal software engineering methods for the development of large software systems increases. An important step towards a widely accepted technique is the Unified Modelling Language 2.0 (UML, [10]) defined by the OMG. Although the current UML standard still lacks a strict formal semantics it allows to specify complete system behaviour using abstract diagrams.

Since systems engineers usually need not know about the underlying formal models of the applied techniques a flexible framework to support interaction with the formal model from the more abstract level of the specification is needed. A typical scenario for such a framework is the simulation of a formally specified system based on its semantics. The framework has to take care for correct translation between system description and formal model. During simulation events have to be translated to their equivalents on each level, i.e. from abstract events to those of the formal model, and vice versa.

Although the use of abstract specification techniques facilitates the understanding of complex systems it is far from properly representing the final system. Thus, tools provide means to simulate the system using, e.g., a prototyped GUI. An underlying framework therefore has to cope with various and independent representations of the system which are all related by a common semantics.

In this paper we present a flexible simulation framework, which is able to connect arbitrary discrete event simulation engines with a number of different visualization targets. In combination with a compiler frontend that transforms

UML 2.0 Sequence Diagrams to Petri nets it is possible to create a realistic simulation of a system specified on a rather abstract level together with its expected environment. The use of formal methods to specify systems further enables application of verification for in-depth examination of the models. The P-UMLaut tool may support this process, e.g., by visualizing counter examples.

2 The Tool Chain

In this section we will shortly introduce the basics of the P-UMLaut tool within its originally intended application domain UML, and describe in more detail the main elements in three subsections. The P-UMLaut tool is available free of charge with full source code at `http://www.p-umlaut.de`.

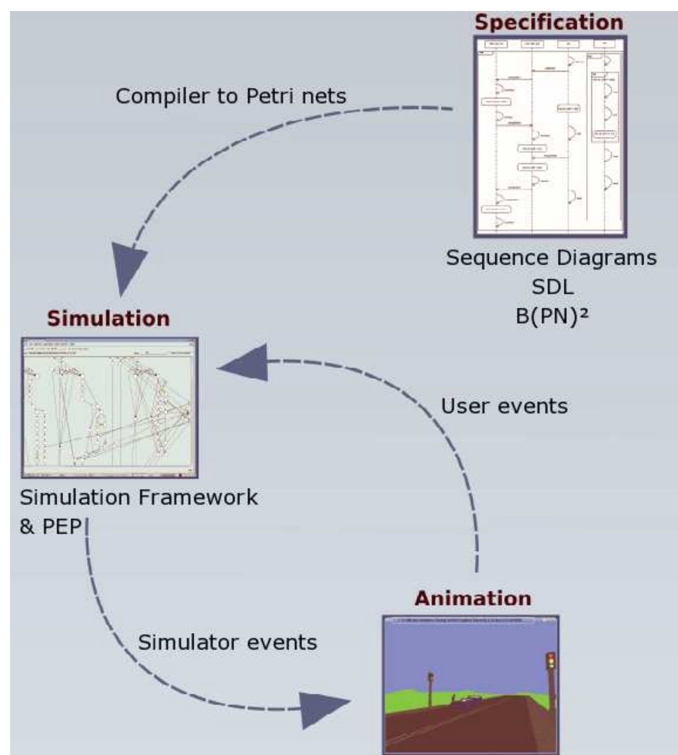


Fig. 1. P-UMLaut tool architecture

The tool has been developed to support software engineers in prototyping visualizations of system behaviour. Its overall architecture is shown in Fig. 1. In order to preserve the usual workflow the visualization components are complemented with a UML to Petri net compiler. Thus, the tool chain starts with

a previously created set of sequence diagrams describing the possible evolutions of the system. Diagrams are compiled into Petri nets which serve as the formal model. The simulation framework drives the different visualizations based on simulation steps executed within the Petri net. The 3D visualization is a standalone application which offers an interactively explorable user defined world. Using its XML-RPC [11] interface the 3D world may act as a target of the simulation framework. By adding animations to the static world system behaviour can be visualized. In conjunction with the Petri net simulator this exemplary use of the simulation framework is comparable to approaches described in [7, 5].

2.1 The UML to Petri Net Compiler

The compiler transforms the diagrams into a semantically equivalent Petri net representation as described in [3]. Due to a lack of tool support for the standardized interchange formats of UML 2.0 (XMI, [12]) the compiler currently requires a proprietary XML description of the diagrams as input. Using its open API the compiler can be easily extended with proper XMI support upon availability of such tools, similar to its already existing XMI support for UML 1.5.

The Petri nets produced are high-level Petri nets as defined in [1]. In order to allow exchange with various tools from the community the compiler provides a flexible backend API to easily add additional file formats to the already existing formats PEP [8] and PNML [2]. The latter is using a simplified approximation of the proposed high-level formats from [9] as long as the ISO standard for high-level PNML [6] is not finalized.

In addition to the Petri net a reference file is created during compilation which contains a table of related events. Using this file equivalent elements of the two different representations may be mapped onto each other.

The compiler features different types of optimizations to support various areas of application for the generated Petri nets. A general optimization for all purposes simplifies transition guards and eliminates dead elements of the net. More specific optimizations apply to the net if it is going to be unfolded to a low-level net to use verification tools. In such a case transitions with a high number of connected arcs would create an exponential growth of the unfolded net. These transition can often be split into several steps while preserving the semantics. For high-level simulations it is more convenient to skip such optimizations since each state of the simulation automatically limits the exponential number of possible bindings by the available tokens to only a few.

2.2 The Simulation Framework

The simulation framework is basically a translation filter which queries its simulation engine for the next event to be executed. This event is translated into the equivalent event of each of the target visualizations and sent out to each instance thereof. In order to support interactive simulations a reception mechanism for events from targets is provided which requires an additional mapping from target events back to simulator events. Such interactive events are usually

prohibited from automatic execution and may be thus only simulated upon user request.

The plugin mechanism of the simulator framework allows to use different simulation engines, mappings, and visualization targets. Plugins are built as dynamically loadable Java archive files using the APIs defined in three interfaces and use Java 5.

For the moment writing the PEP simulation server is the only supported simulation engine. The server offers fast high-level Petri net simulation accessible through an XML-RPC interface. The server is implemented as a standalone application in C++. Integration of a different simulation engine, even one using a completely different formal model, should be easily possible as long as the simulation engine provides an externally accessible interface.

The mapping plugins are capable of creating mapping functions from different file representations of previously generated event relations. Such mappings are used to translate between the different domains of simulation engine and visualization targets. Usually, these mappings are automatically generated while transforming, e.g., UML diagrams into Petri nets. Thus, most mappings are application specific as well. Two generally applicable mappings provided allow to use identity as a mapping for tools directly supporting the simulator events and a mapping defined on the command line upon execution of the simulation framework. The two other currently provided mappings support the mapping file format of the UML to Petri net compiler described in the last subsection, and the reference file syntax used by various compilers of the PEP tool.

Due to the lack of UML 2.0 tools with appropriate interfaces for the purpose of simulation of UML diagrams the main visualization target is the 3D world described in the next subsection. Besides that a logging target has been implemented which allows for easy file capturing of simulation traces. Multiple visualization plugins can be instantiated multiple times in order to use the logging facility in parallel to an interactive animation, or to animate 3D worlds on different computers at once. The interactive features may be restricted to distinct targets in such cases. Similarly, a non-interactive interface is available which allows simpler implementation of plugins.

2.3 3D Visualization

The 3D world exploration and animation tool developed for this project is as well as all previously described tools a standalone application. It allows to visualize user defined 3D worlds and interactively explore those worlds by flying through the world using a moveable camera.

The 3D world and its objects as well as available animations are described in a simple XML format. The supported objects are restricted by the underlying 3D engine used, the freely available Irrlicht engine [4]. These objects define complex objects such as cars, persons, or other rather complete objects of the world.

The objects may have animations already defined. Animations may be also defined by the user based on translation, rotation, scale, and skew operators to transform an object. Some specialized additional operators are defined to change

transparency of an object, and to bind and unbind objects to/from an object in order to dynamically group objects. Each animation has a duration during which it is uniformly executed. Animation groups allow to sequentially compose animations, and an arbitrary number of animations may be executed in parallel.

Animations are executed upon reception of events, which are usually issued by an external source, e.g. the simulation framework. The user may in turn create events by clicking on an object of the 3D world, which are sent out to the external control application. Mappings from events to animations and from objects to events are defined in a second XML file. Both event queues are externally accessible using XML-RPC. If the 3D world is used without an external controller the world remains static due to a missing event source and event handling mechanism.

3 Conclusion and Future Work

The presented tool chain offers support for several aspects of formal software engineering. The UML to Petri net compiler creates the formal semantics crucial for application of formal techniques including verification with model checking techniques. The simulation framework enables different ways of simulating, testing, and visualizing software models, thus making abstract models more comprehensible even for non computer scientists.

Fig. 2. Screenshot of pedestrian crossing example

The P-UMLaut framework has been successfully applied to two larger case studies modelling: a pedestrian crossing shown in Fig.2 and a multi level elevator.

Both examples were modelled as sequence diagrams, compiled to Petri nets, and equipped with interactive 3D models. Test runs of the models showed some flaws in early versions of the sequence diagrams, e.g. deadlocks due to lack of synchronization, or due to race conditions. Formal verification using the model checking facilities of the PEP tool allowed for further analysis of the models, and prove presence of some desired properties such as mutual exclusion of the critical road crossing section.

The flexibility of the simulation framework enables its use for a great range of application domains. Thus, the next steps in the tool's development will be to add new plugins to support additional simulation engines and different visualization targets. The 3D world will be enhanced by further animation types and overlapping animations. The integration of the P-UMLaut tool into the PEP GUI will be a long term goal as well as a more tightly integrated verification facility.

The author would like to thank all members of the P-UMLaut group for the good work performed in the last month. This project has been partially supported by the DAAD project Comete.

References

1. Eike Best, Wojciech Frączak, Richard P. Hopkins, Hanna Klaudel, and Elisabeth Pelz. M-nets: an Algebra of High-level Petri Nets, with an Application to the Semantics of Concurrent Programming Languages. *Acta Informatica*, 35(10):813–857, 1998.
2. Jonathan Billington, Søren Christensen, Kees van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In Eike Best and Wil van der Aalst, editors, *ATPN 03*, volume 2679 of *Lecture Notes in Computer Science*, pages 483–506, Eindhoven, The Netherlands, 2003. Springer-Verlag.
3. Christoph Eichner, Hans Fleischhack, Roland Meyer, Ulrik Schrimpf, and Christian Stehno. Compositional Semantics for UML 2.0 Sequence Diagrams Using Petri Nets. In Andreas Prinz, Rick Reed, and Jeanne Reed, editors, *SDL 2005: Model Driven*, volume 3530 of *Lecture Notes in Computer Science*, pages 133–148. Springer-Verlag, 2005.
4. Irrlicht 3D engine. http://irrlicht.sourceforge.net.
5. Michael Kater. 3D-Visualisierung und Animation paralleler Prozesse. Master's thesis, Universität Hildesheim, 1996.
6. Ekkart Kindler. High-level Petri Nets – Transfer Format. Working draft 0.9.0, International Standard Organisation, 2005. ISO/IEC 15909 Part 2.
7. Ekkart Kindler and Csaba Páles. 3D-Visualization of Petri Net Models: A Concept. In Jordi Cortadella and Wolfgang Reisig, editors, *ATPN 2004*, volume 3099 of *Lecture Notes in Computer Science*, pages 464–473. Springer-Verlag, 2004.
8. The PEP tool. http://peptool.sourceforge.net.
9. Christian Stehno. Interchangeable High-Level Time Petri Nets. PNML Workshop Helsinki, 2005.
10. UML homepage. http://www.omg.org/uml.
11. Dave Winer. XML-RPC Specification, 2003. http://xmlrpc.org/spec.
12. XMI homepage. http://www.omg.org/technology/documents/formal/xml.htm.

Petri Nets and the Real World

Ekkart Kindler and Frank Nillies
Department of Computer Science
University of Paderborn
D-33098 Paderborn, Germany
[kindler|frank]@upb.de

***Abstract*— Two years ago, we extended Petri nets by a simple but powerful concept that transforms a Petri net, originally designed for analyzing a system, into an interactive 3D-visualization of that system. We call this extension of Petri nets and the tool implementing the 3D-visualization *PNVis*. The basic idea of PNVis is to equip the Petri net with information on how the tokens on different places correspond to physical objects and how these objects behave. Up to now, PNVis associates the simulation of tokens of the Petri net with objects in a virtual 3D-world.**

In this paper, we take the next step and use the concepts of PNVis to associate the tokens of a Petri net with the real world. This way, a Petri net can be used as a controller of some plant, which might be useful for rapid prototyping a controller. Conceptually, this works for any kind of hardware – for simplicity, we used a Petri net for controlling a simple toy train.

What is more, we showed that the control and the 3D-visualization could even be synchronized so that the visualization, basically, showed the behaviour of the real world. Altogether, this demonstrates that the concepts of PNVis are a powerful means for designing, prototyping, and validating controllers.

I. Introduction

Petri nets are a well-accepted formalism for modelling concurrent and distributed systems. The main advantages of Petri nets are their graphical notation, their simple semantics, and the rich theory for analyzing and verifying their behaviour.

In spite of their graphical nature, getting an understanding of a complex system just from studying the Petri net model itself is quite hard – if not impossible. In particular, this applies to experts from some application area who, typically, are not experts in Petri nets. 'Playing the token-game' is not enough for understanding the behaviour of a complex system. The concepts of *PNVis* improve this situation by providing a simple mechanism for animating the behaviour of the system modelled as a Petri net in a 3D-visualization. The extensions of Petri nets that are necessary for a 3D-visualization are remarkably simple [2], [3]: the tokens of the Petri net are associated with objects of a virtual world and are assigned a behaviour. A simple feedback mechanism allows the 3D-visualization to have an effect on the behaviour of the Petri net.

The interaction between the actual *Petri net simulator* (*PNSim*) and the *visualization* (*Vis*) could be realized by a simple protocol. Therefore, it was easy to use the very same 3D-visualization for animating the behaviour of systems that were modelled by other formalisms than Petri nets. Moreover, it occurred to us that, likewise, we could easily replace the 3D-visualization by a control panel that allows users to interact with the Petri net simulation, which would result in a tool similar to *ExSpect* with its dash boards [5]. Even more interestingly, the virtual world of the visualization could be replaced by the real world, e. g. a plant, which allows us to control the plant directly by a Petri net (simulator).

In this paper, we will present the basic concepts that allow us to control a plant by a Petri net. The concepts are, basically, the same as for visualizing a Petri net, and the controller and visualization can even be used synchronously, which allows us to visualize the real behaviour of a plant while running. Since, in addition, these concepts are quite simple and compatible with the standard Petri net semantics, these concepts seem to be quite universal for relating the behaviour and the analysis results of Petri nets to the real world.

II. PNVis

In this section, we give a brief account on the extension needed for visualizing Petri nets by the help of PNVis. To this end, a Petri is equipped with some information on the shape and the dynamic behaviour of the objects corresponding to tokens on some places.

Shapes and animation functions: In a first step, we distinguish those places of a Petri net that correspond to virtual objects. We call these *animation* places. The idea is that each token on such a place corresponds to an object with its individual appearance and behaviour. In order to visualize and to animate a physical object, we need two pieces of information: its shape and its behaviour.

Defining the *shape* of the object is easy: Each animation place is associated with a *3D-model* (e. g. a VRML file) that defines the shape of all tokens on this place. Defining the *behaviour* of an object is similar: Each animation place is associated with an *animation function*, where the animation function is constructed from some predefined animation functions. When a token is produced on an animation place, an object with the corresponding shape appears and behaves according to the animation function. For example, the object could *move* along a predefined line, the object could *rotate*, or the object could simply *appear* at some position.

In order to illustrate these concepts, let us consider a simple example: a toy-train. Figure 1 shows the layout of a toy-train, which consists of two semicircle tracks *sc1* and *sc2*, which are composed to a full circle. We call this layout the underlying *geometry*. For defining such a geometry, there is

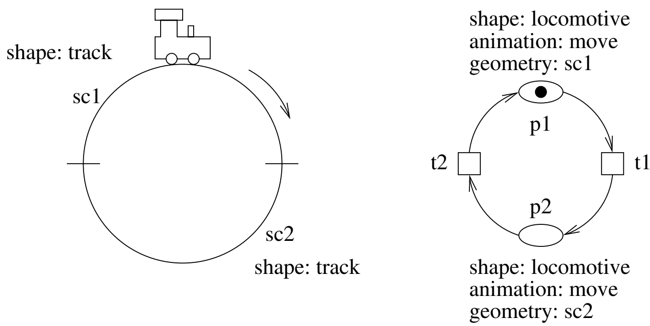


Fig. 1. A toy-train

a set of predefined geometrical objects such as lines, circle segments, and Beziér curves. In our example, there is one toy-locomotive moving clockwise on this circle. The right-hand side of Fig. 1 shows the corresponding Petri net model, where both places *p1* and *p2* are animation places. In this example, the correspondence between the Petri net model and the physical model is clear from the similar layout. Formally, this correspondence is defined by annotating each place with a reference to the corresponding element in the geometry. The annotation *shape* defines the shape of the objects. In our example, it is a locomotive for both places, where the details of the definition of the shape are discussed in [2], [3]. Here, we can think of it as the reference to some VRML model of a toy locomotive. The annotation *animation* defines the behaviour of the object, which is started when a token is added to the place. In our example, it is a *move* animation. Note that, without additional parameters, each animation function refers to the geometry object corresponding to that place. So, a locomotive corresponding to a token on place *p1* moves on track *sc1*, and a locomotive corresponding to a token on place *p2* moves on track *sc2*.

In order to make our example complete, we must also give some information on how to visualize the geometry objects themselves. To this end, each geometry object can have an annotation *shape*, too. In our example, the semicircles should be visualized as tracks (see [2], [3] for details). Once we have provided this information, we can start PNVis for visualizing this system. Figure 2 shows a screen-shot of the animation of our example, where the locomotive on place *p1* has almost reached the end of its move animation on *sc1*.

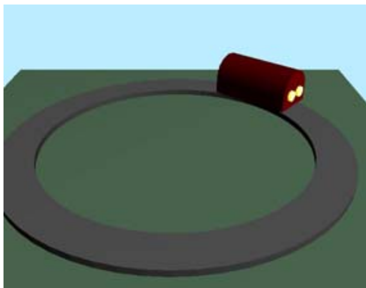


Fig. 2. Screen-shot of the visualization

Object identities: Up to now, the objects and shapes corresponding to the tokens on the two places *p1* and *p2* are completely independent of each other. When transition *t1* fires, an object corresponding to the token on place *p1* is deleted and a new object corresponding to the new token on place *p2* is created and the move animation is started. Clearly, this is not what happens in reality. In reality, the same object, the locomotive, moves from track *sc1* to track *sc2*. In order to keep the identity of an object when a token is moved from one place to another, we equip the arcs of the Petri net with annotations of the form *id:n*, where *n* is some identifier. We call *n* the *identity* of that arc. By assigning the same identity to an in-coming arc and an out-going arc of a transition, we express that the corresponding object is moved between those two places. In order not to clone an object, we require that there is a one-to-one *correspondence* between the identities of the in-coming and out-going arcs of a transition; i. e. each identity occurs exactly once in all in-coming arcs and exactly once in all out-going arcs. Figure 3 shows the toy-train example equipped with such identities[1].

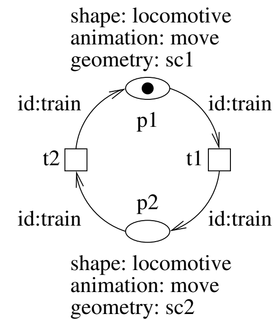


Fig. 3. The model with identities

Animation results: Next, we consider the relation of the behaviour of the Petri net and the animations of the objects corresponding to the tokens in more detail. When a token is added to an animation place by firing a transition, the animation for the corresponding object is started. But, what will happen, if a token is removed before the animation is terminated? In our example, this does not make much sense – the locomotive would jump from some intermediate position of the track to the start of the next track. Assuming that transition firing does not take any time, this behaviour is physically impossible. But, there are other examples in which a transitions could fire while an animation is running. Therefore, the Petri net model must explicitly define whether a transition may remove a token with an animation still running on the corresponding object or not. When the transition should wait until the animation of a token has terminated before removing it, we add the annotation *result:* {..} to the corresponding arc. Actually, an animation function has a return value, and the annotation *result* says for which return value of the animation

[1]Both transitions have only one in-coming and out-going arc. Therefore, the examples is not very interesting. We will see more interesting examples, soon.

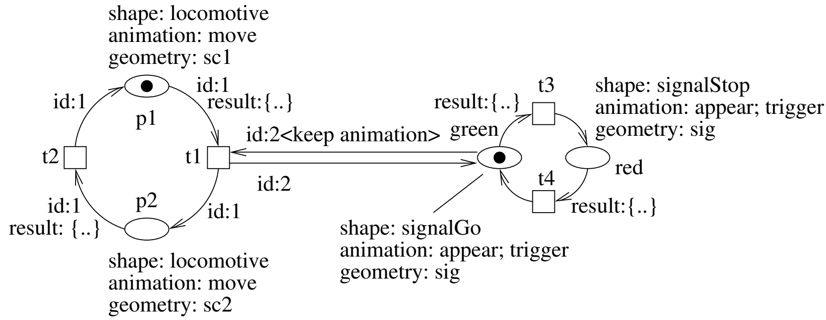


Fig. 4. A toy-train with a signal

the corresponding transition may fire. The set {..} stands for all possible return values. So, *return:* {..} means that the animation must be terminated – with any return value. If there is no such annotation at the arc, the transition need not wait until the animation of the corresponding object is terminated – when fired, the transition simply stops the animation[2].

In order to illustrate these new concepts, we extend our example. We assume that there is a signal at the end of track *sc1*. When the signal is in state *stop*, the locomotive stops at the end of track *sc1*; when the signal is in state *go*, the locomotive may enter track *sc2*. In order to have a position for the signal in the layout, the geometry is extended by a point *sig* at the end of semicircle *sc1*. Figure 4 shows the Petri net model of this extended system. The two places *p1* and *p2* as well as the transitions *t1* and *t2* are the same as before. The arcs are equipped with identities in order to keep the same object, the locomotive, on the tracks. The annotation *result:*{..} guarantees that the transitions wait until the move animation of the locomotive has come to an end (i. e. the locomotive has reached the end of the track). Next we consider the signal: The two states of the signal are represented by the places *stop* and *go*. The object corresponding to a token on place *stop* is a signal with its red light on: *SignalStop*. The object corresponding to a token on place *go* is a signal with its green light on: *SignalGo*. These objects will appear at the point sig of the geometry (somewhere at the end of *sc1*). Due to the loop between place *go* and transition *t1*, transition *t1* can only fire, when the signal is in state *go*. The interesting parts of this model are the identities of transition *t1*; when transition *t1* is fired, the object of the signal from place *go* stays on this place. Moreover, the animation is not restarted, because the identity is equipped with the *keep animation tag*.

Another interesting issue is the animation of the signal. The animation function is composed from two predefined animation functions: *appear; trigger*. The meaning is that these animations are started sequentially. When the first animation function has finished, the second is started. So, in both cases the signal appears at position sig; then, it behaves as a trigger. A *trigger* is an animation function that simply waits for a user to click on that object. When this happens, the animation terminates (where the result depends on where the user clicked). In combination with the annotations *result:*{..} at the in-coming arcs of transitions *t3* and *t4*, the user can toggle the state of the signal by clicking on the signal in the 3D-visualization.

[2]If the corresponding arc has an identity, there is an option *keep animation* that does not stop the current animation on the object, but continues the animation while the token is on another place.

III. Controlling Plants

In this section, we show how the concepts of PNVis can be used for controlling a plant via the same interface as the visualization; i. e. the Petri net simulator does not interact with the 3D-visualization, but with the hardware. In our implementation, we used a Märklin toy train, which has an interface to some Linux workstation in order to interact with it. A picture of the toy train system is shown in Fig. 5.

Actions and events: Analogously to starting an animation of an object in the visualization when a token is added to a place, the interaction with the hardware issues some *action*, when the simulator adds a token to some place. Such an action could be switching some actuators in the hardware. Likewise, the interaction with the hardware can issue an action, when the simulator removes a token from some place. The details on how to define actions, and how to associated them with some place of a Petri net will be discussed later.

In order to give the simulator some feedback on the behaviour of the hardware, we will use *events*. An example for an event could be a rising edge at some position sensor, indicating that the toy-train has reached the end of some track, so that the token on a place representing the train on that track could be removed and added to a place representing the next track. When the event occurs, a token on the corresponding place will be assigned a result value (which could depend on the event). This way, the Petri net simulator knows that the token can be removed. In our example, we have one sensor at the end of each track. The details on how to define events and how to relate them to a place will be discussed later.

In a nutshell, the actions on the hardware correspond to starting an animation in the visualization, and the events correspond to the termination of an animation function in the visualization. This allows the Petri net simulator to use the very same interface for interacting with the visualization and with the hardware; we call this interface the *interaction*

Fig. 5. The hardware: A Märklin toy train

handler interface. But, we do not go into the technical details of this interface here.

Panels and buttons: Of course, the user would also like to interact with the hardware. For example, the user might want to toggle some signal to red or to green, or the user might want to toggle some switch from left to right or vice versa.

To this end, the hardware handler supports the definition of *buttons* on some control *panel*, which can be pressed by the user in order to interact with the hardware. An example panel for our toy-train example is shown in Fig. 6. The buttons can be activated and deactivated by corresponding actions, i.e. when tokens are removed and added to some places. The activated buttons will be highlighted and can be pressed by the user. Pressing an activated button, triggers an event, which in turn can be used to return a result value to some token, which enables the transition to fire.

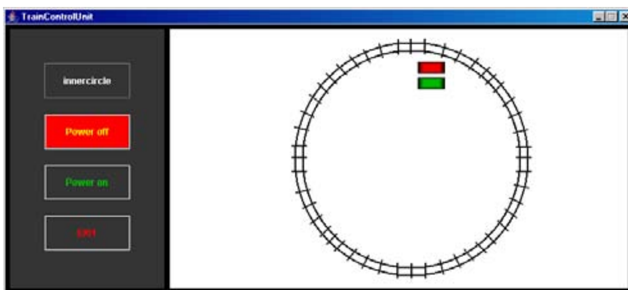


Fig. 6. Screen-shot of a simple control panel

Example: The actions and events as well as the panel with its button are defined in an XML file. A simplified version of the XML file for our toy-train example of Fig. 4 is shown in Fig. 7 at the end of this article.

The first part of this file, defines the initialization of the hardware, i. e. the initial setting of all actuators. In our example, the signal (withe hardware address 101) is set to green. In a second part, two buttons are defined, which allow the user to interact with the system. Each button has a position, a size (dimension), and an image that appears on that button. Moreover, the two colours `acolor` and `hcolor` define the colour in the activated and the deactivated state of the button. The definition of each button implicitly defines an events, which occurs when a user presses the button. The name of this event is given in the corresponding attribute in the button definition.

Moreover, the file defines some actions and events. Each action is assigned a name, and it refers to some component in the hardware by some id; actually, this id refers to some software object representing the hardware component in the Hardware Abstraction Layer (HAL). The attribute type defines the class of this object and the attribute perform refers to the method to be called on this object when the action is initiated.

Likewise, the definition of an event defines the name, and it refers to some hardware id (resp. a software object representing it). The attribute type defines its class, and the attribute trigger defines the value which triggers this event – actually, it is a change to this value triggering the event. In our example, the event endSC1 is triggered once the sensor S1 changes its value to 1, which indicates that the toy train

has reached the end of track sc1.

A second XML file, defines how the different actions and events are associated with the places of the Petri net. The XML file for our example is shown in Fig. 8 at the end of this article. In this file, place p1 is assigned the end-event endSC1. When this event occurs a token on place p1 will receive 0 as its result value[3]. For each of the places green and red representing the two states of the signal, we define two actions that are invoked, when a token is added to them: the first action enables the button for switching the signal to the other state, the second action actually switches the hardware signal to the state corresponding to the place (red or green). The action associated with the removal of a token is the deactivation of the other button. Moreover, the token will be assigned the result value 0 when the button is pressed, which will allow the Petri net simulator to fire the transition from red to green or vice versa.

When the Petri net simulator and the hardware handler are started with the Petri net from Fig. 4 and the two XML files from Fig. 7 and 8, they actually control the real hardware, where the user can interact with it via the panel shown in Fig. 6.

Synchronizing interaction handlers: Actually, we have another implementation of an interaction handler, which is capable of synchronizing one *master interaction handler* with many other *slave interaction handlers*. This way, it is possible to start the Petri net simulation with the hardware handler as the master and a visualization handler as the slave. Then the visualization, basically, shows the behaviour of the real hardware. Due to variations in speed, there might be minor mismatches: For example, the toy-train in the visualization might stop at the end of a track because the real train did not arrive there yet; or the toy-train of the visualization might jump to the next track from the middle of the current track when the real toy-train reaches the end of its track first. But, the deviations will be synchronized when the transitions fire and could be minimized by dynamically adjusting the speeds.

This synchronization shows that the interface has been well-designed and implemented. But, we cannot go into the details of this interfaces here. You will find some more details in [1], [4].

IV. Conclusion

In this paper, we have shown that the simple concepts of PNVis can be used not only for visualizing the behaviour of a real system, but also for controlling it. The basic concept is the result value of a token, which is set either by the visualization or by the real hardware. Dependent on the additional labels of the net a transition can only fire, when tokens have particular result values. Note that this is compatible with the traditional firing rule, where transitions fire non-deterministically and are not even required to fire at all. The result values just make the behaviour more deterministic. Therefore, the analysis results for the original Petri net are still valid.

[3] Actually, the result values are irrelevant in this example.

This way, a single Petri model can be used throughout the design process including analysis and verification as well as validation and communication with a customer.

References

[1] Dennis Beck. Steuerung von Anlagen durch Petrinetzmodelle. Bachelor thesis, Department of Computer Science, University of Paderborn (in German), June 2004.

[2] Ekkart Kindler and Csaba Páles. 3D-visualization of Petri net models: A concept. In G. Juhas and R. Lorenz, editors, *Workshop Algorithmen und Werkzeuge für Petrinetze*, pages 69–78, September 2003.

[3] Ekkart Kindler and Csaba Páles. 3D-visualization of Petri net models: Concept and realization. In J. Cortadella and W. Reisig, editors, *Application and Theory of Petri Nets 2004, 25th International Conference*, *LNCS* 3099, pages 464–473. Springer, June 2004.

[4] Frank Nillies. Synchronisation einer 3D-Visualisierung mit einer realen Anlage auf der Basis von Petrinetzmodellen. Bachelor thesis, Department of Computer Science, University of Paderborn (in German), May 2005.

[5] Eric Verbeek. ExSpect 6.4x product infromation. In K. H. Mortensen, editor, *Petri Nets 2000: Tool Demonstrations*, pages 39–41, June 2000.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE occurrences SYSTEM "occurrence.dtd">

<hardwaredefinition>
  <hardwareinitialisation>
    <initial type="signal" adressid="101"
             state="green"/>
  </hardwareinitialisation>

  <buttons>
    <pushbutton id="red" event="redPressed"
      position="320,295" dimension="40,12"
      acolor="#FF0000" hcolor="#4C4C4C" />
    <pushbutton id="green" event="greenPressed"
      position="320,310" dimension="40,12"
      acolor="#00C800" hcolor="#4C4C4C" />
  </buttons>

  <events>
    <event name="endSC1">
      <attributes type="sensor" id="s1"
                  trigger="1"/>
    </event>
    <event name="endSC2">
      <attributes type="sensor" id="s2"
                  trigger="1"/>
    </event>
  </events>

  <actions>
    <action name="enableGreen">
      <attributes type="button" id="green"
                  perform="settoenable"/>
    </action>
    <action name="disableGreen">
      <attributes type="button" id="green"
                  perform="settodisable"/>
    </action>
    <action name="enableRed">
      <attributes type="button" id="red"
                  perform="settoenable"/>
    </action>
    <action name="disableRed">
      <attributes type="button" id="red"
                  perform="settodisable"/>
    </action>

    <action name="fire101green">
      <attributes type="signal" id="101"
                  perform="switchToGreen"/>
    </action>
    <action name="fire101red">
      <attributes type="signal" id="101"
                  perform="switchToRed"/>
    </action>
  </actions>
</hardwaredefinition>
```

Fig. 7. Actions, events and panel definition

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE relations SYSTEM "relation.dtd">

<eventactiondefinition>

  <place name="p1">
    <endEvent>
      <event name="endSC1" result="0"/>
    </endEvent>
  </place>

  <place name="p2">
    <endEvent>
      <event name="endSC2" result="0"/>
    </endEvent>
  </place>

  <place name="green">
    <onAdd>
      <action name="enableRed"/>
      <action name="fire101green"/>
    </onAdd>
    <onRemove>
      <action name="disableRed"/>
    </onRemove>
      <endEvent>
    <event name="redPressed" result="0">
      </endEvent>
    </place>

  <place name="red">
    <onAdd>
      <action name="enableGreen"/>
      <action name="fire101red"/>
    </onAdd>
    <onRemove>
       <action name="disableGreen"/>
    </onRemove>
    <endEvent>
      <event name="greenPressed" result="0">
    </endEvent>
  </place>

</eventactiondefinition>
```

Fig. 8. Associating actions and events with the places

A First View on a Generalised Modelling Toolkit for Graph-based Languages

Stephan Philippi*, Alexander Pinl**, Gerrit Rausch

University of Koblenz, Department of Computer Science,
P.O.Box 201 602, 56016 Koblenz, Germany
stephan.philippi@uni-koblenz.de

Abstract. This article gives a first view on a generalised toolkit for the modelling of complex systems which in principle allows for the integration of arbitrary graphical notations. The early prototype of the tool to be presented provides support for low-level Petri-Nets, UML class diagrams and a process modelling language. Several aspects of the design and implementation of this tool are discussed throughout the article. The coverage includes the modelling with different graph-based languages within a single tool, the formal analysis of Petri-Nets as well as the dynamic import from and export into XML based storage formats.

1 Motivation

With the advent of the UML [OMG04b] and its predecessors a plethora of tools to work with complex graphical models has been developed in the last decade, like ArgoUML [Argo05] and others. Since the UML does not provide a sound formal basis of its languages, these tools usually do not support the simulation and analysis of systems. Another family of modelling tools is centered around the notion of Petri-Nets. Due to the formal underpinning of Petri-Nets these tools not only allow for the modelling of systems but also for their simulation and, depending on the supported class of Petri-Nets and its characteristics, the formal analysis of specific system properties. There are plenty of tools available in this area, like Design/CPN [ChJoKr97], TimeNet [ZiFrHo01], Renew [KuWiDu04] and others. As Petri-Nets are not part of the UML, both families of tools are usually either focused on the UML *or* Petri-Nets. Since Petri-Nets on the one hand are not well suited to represent more static aspects like the architecture of a system or a data model, and the UML on the other hand does mostly not allow for the simulation and analysis of models, a combination of both is a promising approach. Despite the obvious advantages only very few tools integrate support for Petri-Nets *and* other modelling languages. The `NEPTUN` [Phil02] and `POSEIDON`

* The presented work is kindly supported by the European Science Foundation under Exchange Grant 2005/603.

** A. Pinl is supported by the DFG under grant LA 1042/7-1

[LaMüPh02] tools developed in the last years at the University of Koblenz prototypically integrated cross-language approaches supporting Petri-Nets and OMT notations [Rumb*91] in earlier and UML class diagrams in more recent versions. Since NEPTUN was focused on applications in software engineering, class diagrams served for the modelling of object-oriented system architectures, while different classes of Petri-Nets were utilised to specify dynamic and functional system aspects. While one of the main features of NEPTUN was automatic generation of concurrently executable source code from formally based object-oriented models, POSEIDON, in contrast, was more focused on the simulation and analysis of different classes of Petri-Nets. In this context OMT and UML class diagrams were provided as data modelling languages for the specification of data structures used in high-level Petri-Nets, as opposed to the more functionally oriented approach of Design/CPN.

Although the different combinations of modelling languages turned out to be beneficial for many applications, from a technical point of view both tools suffered from their development history. Having their origins already in the late 80ies, their common predecessor GRANIT was developed in the C programming language for a UNIX system ([Krause90][Piet90][Rogge90][Sabel90]). After the migration to C++ and the Solaris platform in the beginning of the 90ies the existing program core served as a basis for the further development of NEPTUN and POSEIDON. With the switch to LINUX in the middle of the 90ies and the advent of the Windows operating system, more and more migration problems had to be solved which finally lead to the development of a Java based graphical user interface on top of the existing C/C++ data structures and algorithms. Despite the fact that the original concepts supported by both tools proved to be useful in practical and theoretical applications, the tools grew more and more difficult to extend and maintain due to their heritage. In order to further develop the concepts underlying NEPTUN and POSEIDON a new tool was envisaged to support a generalised cross-language approach to graph-based modelling. A first view on the early prototype of this tool is given in this article.

Starting from this perspective section two gives insights into some of the requirements for the development of the generalised modelling toolkit. Afterwards the high-level architecture is described and also some light is shed on specific aspects of the implementation. Section four concludes the article with an overview of current and future work on the project.

2 Requirements

Given the multitude of problems we encountered over the years while migrating between different hardware platforms and operating systems, platform independent software development is a direct consequence of our past experiences. In this context the Java programming language seems to be a sensible choice, since almost any platform is supported and with newer versions of this language there is no longer any performance penalty in comparison to other languages like C++.

The intention to be able to support in principle arbitrary modelling languages within a single tool leads to the definition of a *project* as a set of instances from the types of graphs supported by the tool. Important requirements in this context are the administration of an arbitrary number of projects within the tool by means of a workspace concept as well as the graph-based organization of the contents of single projects.

In order to be able to include arbitrary graph-based languages into a modelling toolkit, the architecture of such a software has to be as modular as possible. A generic framework for these purposes ideally offers a plug-in interface for specific editors which transparently provides basic services in order to minimize the efforts needed for plug-in development, like undo/redo, grouping, zoom and grid functionality.

For the first prototype of the toolkit different kinds of editor plug-ins were required, namely for Petri-Nets, UML class diagrams [OMG04b] and the APRIL process modelling language [Hill05]. With respect to Petri-Nets the question arises which of the very many classes of Petri-Nets to support? Since a single net class can hardly satisfy the needs in different application areas an extendable approach which supports in principle arbitrary classes of Petri-Nets is needed. Another important requirement for the tool is the ability to handle complex models, i.e. the Petri-Net plug-in should support hierarchical and non-hierarchical modularization capabilities. In order to simulate and analyse models the tool has to include specific components for these purposes. Due to the generic approach to support Petri-Nets also the simulation and analysis components have to be built in a modular way which allows for easy extensions.

Import and export of models into standardized XML formats for the exchange of models between different tools is another important requirement. A more direct coupling of tools is required to (optionally) benefit e.g. from the capabilities of Mathematica [Wolf99] and similar software.

3 High-Level Architecture and Implementation Aspects

The high-level architecture of the toolkit is given in figure 1. Based on the principles of the model-view-controller design pattern [Gamma*95], structures for data storage, visual presentation and user interaction are separated from each other. The toolkit is designed around core data structures which are capable of storing arbitrary kinds of graphs. More specialized structures are derived from the generic graph model for the handling of e.g. Petri-Nets. All other modules in the toolkit (will) access this core in order to provide specific functionality like analysis and simulation of models, import/export from and into various file formats, different kinds of visualizations and more.

The underlying ideas for the realization of the analysis and import/export components will be described in more detail in the following.

One of the main requirements for the *analysis component* is that it should be easily extendable with new algorithms. In order to comply to this requirement

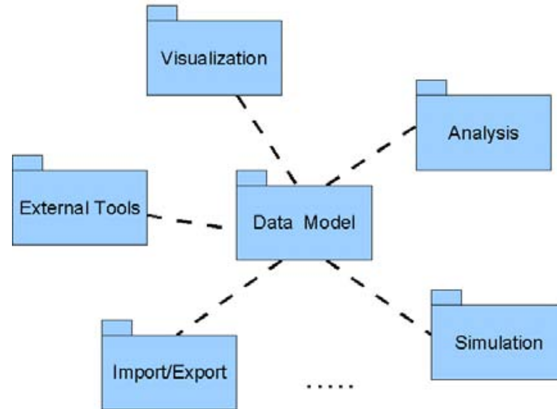


Fig. 1. High-level architecture of the modelling toolkit.

the toolkit provides an infrastructure for different kinds of analysis algorithms which consists of abstract representations for:

- graph-based algorithms, like a test for the free-choice property
- algorithms based on linear algebra, like P/T-invariants
- reachability graph based analyses

Algorithms in these classes can be easily integrated into the toolkit. The data structures and services needed for analysis algorithms in the given categories, like matrice represenions of given Petri-Net models for the calculation of invariants, are transparently provided by the surrounding infrastructure.

The XML based exchange of data between tools has become a topic of major interest in recent years. Several standards exist in this area for the exchange of different kinds of graph-based models. While the GXL [Winter02] is a standard for the exchange of graphs in general, XMI [OMG04a] defines a standard for the exchange of UML models and the PNML [Bill*03] for the exchange of Petri-Nets. Since our modelling toolkit supports in principle arbitrary kinds of visual languages, specialised file formats like XMI and PNML are generally not well suited as a storage format. In order to be able to store models from arbitrary modelling languages, our *import/export component* instead makes use of GXL as a more generic approach to model exchange. However, the choice for a GXL based storage format does not imply that the tool does not support other formats for import/export purposes. Quite the contrary, a dynamic approach to import/export which makes use of XSLT stylesheets is pursued. Since XSLT can transform any XML stored model into other representations, exchange formats like PNML and XMI as well as proprietary file formats from other tools can be easily supported.

Figure 2 gives a first impression of the early prototype of the modelling toolkit. The tree view is used throughout the tool for the administration of and

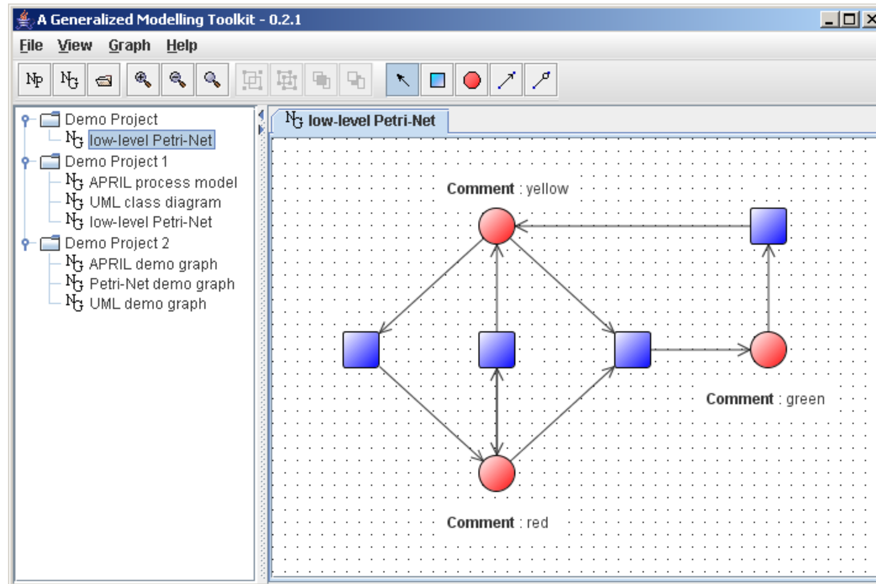


Fig. 2. Project view and Petri-Net editor.

the navigation in projects. The main window contains an editor for the selected modelling language. A dynamic toolbar provides the interaction elements needed to actually create and edit graph-based models. Figure 2 depicts a low-level Petri-Net which represents a simple traffic light.

4 An Outlook on Current and Future Work

Since the presented toolkit is still in its infancy there are many areas for improvements. Currently we are working on the completion of the ideas presented above for a standard conforming approach to support storage formats like GXL, PNML and XMI. Furthermore we are working on the completion of the components for the editing and analysis of Petri-Nets as well as the editor for UML class diagrams.

Future work includes the set up of the simulation component as well as the development of a module for automatic code generation from Petri-Nets.

Ackknowledgements: The authors would like to thank Mario Demuth, Frank Jensen, Andreas Kind, Markus Knopf, Steffi Lueck, Markus Pinl, Alexander Probst, Christoph Surges and Kerstin Susewind for their contributions to the project as well as Kurt Lautenbach for valuable discussions.

References

[Argo05] **ArgoUML Development Team**. *'ArgoUML Documenation'*. 'argouml.tigris.org', 2005.

[Bill*03] **J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno und M. Weber**. *'The Petri Net Markup Language: Concepts, Technology, and Tools'*. *'Proceedings of the 24th Int. Conference on Theory and Applications of Petri-Nets'*, Eindhoven, Netherlands, 2003.

[ChJoKr97] **S. Christensen, J. B. Jorgensen und L. M. Kristensen**. *'Design/CPN - A Computer Tool for Coloured Petri Nets'*. *'TACAS'97 - Tools and Algorithms for the Construction and Analysis of Systems'*. LNCS 1217, Springer, Berlin, 1997.

[Gamma*95] **Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides**. *'Design Patterns'*. Addison Wesley, Reading, MA, 1995.

[Hill05] **H. J. Hill**. *'Visualisierung von Prozessabläufen: Benutzergestützte Generierung von Animationen für APRIL-Diagramme'*. Diploma Thesis (in german), University of Koblenz, 2005.

[Krause90] **I. Krause**. *'Berechnung von Invarianten in unären Prädikat/Transitionsnetzen'*. Diploma Thesis (in german), Rheinische Friedrich-Wilhelms-University Bonn, 1990.

[KuWiDu04] **O. Kummer, F. Wienberg und M. Duvigneau**. *'Renew - User Guide, Release 2.0'*. 'www.renew.de', 2004.

[LaMüPh02] **K. Lautenbach, J. Müller und S. Philippi**. *'Modellierung, Simulation und Analyse mit dem Petri-Netz-Tool POSEIDON'*. *'PROMISE 2002 - Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen'*. Lecture Notes in Informatics, GI-Edition, 2002.

[OMG04a] **OMG (Object Management Group)**. *'UML 2 Diagram Interchange'*. 'www.omg.org/uml', 2004.

[OMG04b] **OMG (Object Management Group)**. *'UML Specification 2.0'*. 'www.omg.org/uml', 2004.

[Phil02] **S. Philippi**. *'A CASE-Tool for the Development of Concurrent Object-Oriented Systems based on Petri-Nets'*. Petri-Net Newsletter 62, 2002.

[Piet90] **R. Pietschmann**. *'Berechnung von Invarianten in regulären Netzen'*. Diploma Thesis (in german), Rheinische Friedrich-Wilhelms-University Bonn, 1990.

[Rogge90] **W. Rogge**. *'Berechnung von Invarianten in Stellen/Transitionsnetzen und Free Choice Netzen'*. Diploma Thesis (in german), Rheinische Friedrich-Wilhelms-University Bonn, 1990.

[Rumb*91] **J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy und W. Lorensen**. *'Object-oriented modeling and design'*. Prentice Hall International, 1991.

[Sabel90] **N. Sabel**. *'Berechnung von Invarianten in assoziativen Netzen'*. Diploma Thesis (in german), Rheinische Friedrich-Wilhelms-University Bonn, 1990.

[Winter02] **A. Winter**. *'GXL - Overview and Current Status'*. *'Proceedings of the International Workshop on Graph-Based Tools (GraBaTs)'*, Barcelona, 2002.

[Wolf99] **S. Wolfram**. *The Mathematica Book*. Cambridge University Press, 1999.

[ZiFrHo01] **A. Zimmermann, J. Freiheit und G. Hommel**. *'Discrete Time Stochastic Petri Nets for Modeling and Evaluation of Real-Time Systems'*. *'Proceedings of the Int. Workshop on Parallel and Distributed Real-Time Systems'*, San Francisco, 2001.

Ein Petrinetzsystem zur Modellierung selbstmodifizierender Petrinetze

Volker Tell, Daniel Moldt
University of Hamburg, Computer Science Department
Arbeitsbereich TGI (Theoretische Grundlagen der Informatik),
Vogt-Kölln-Str. 30, D-22527 Hamburg {9tell,moldt}@informatik.uni-hamburg.de

Zusammenfassung— **Zur Fundierung eines petrinetz- und agentenorientierten Softwareentwicklungsansatzes wird auf der Basis von Referenznetzen ein einfaches und universelles Petrinetzsystem präsentiert, das es erlaubt, zur Laufzeit die Struktur eines Petrinetzes zu verändern. Anhand des Producer/Consumer-Beispiels wird das Modellierungskonzept illustriert.**

Die Erweiterungen des Werkzeugs Renew zur prototypischen Implementierung werden skizziert.

Keywords: Höhere Petrinetze, Einheitentheorie, Renew, Modellierung, Strukturdynamik, Netzstrukturänderung, Selbstmodifikation, selbstmodifizierende Petrinetze

I. Einleitung

Flexibilität innerhalb der Modelle der Softwaretechnik ist ein erklärtes Ziel. Konzeptionell werden dazu zunehmend agentenorientierte statt objektorientierter Konzepte eingesetzt. Modellierungstechniken wie UML (Unified Modeling Language) (siehe [OMG05]) werden zunehmend um Konstrukte zur Erhöhung der Ausdrucksmächtgikeit erweitert. So sind z.B. einiger der bisher in AUML (Agent UML) (siehe [Ode05]) vorhandenen Konstrukte übernommen worden.

Im Bereich der Petrinetze wird die Ausdrucksmächtigkeit ebenfalls kontinuierlich erhöht. So entstanden aus B/E-Netzen die S/T-Netze, die gefärbten Netze und die Netze-in-Netzen. Charakteristisch ist, dass damit eine zunehmende Flexibilität der Struktur des Systems einhergeht, obwohl es sich lediglich um Faltungen der (unendlichen) B/E-Netze handelt. Referenznetze (siehe [Kum02]) bieten durch ihre Konzepte der synchronen Kanäle, der Netzklassen und der Erzeugung von Netzinstanzen zur Laufzeit eine hohe Flexibilität. Durch ihre Implementierung im Werkzeug Renew werden die Konzepte unmittelbar unterstützt (siehe [KWD05]). Während neue Netzklassen durch einen speziellen „Net Loader“ zur Laufzeit eingeführt werden können, ist es trotz der hohen Flexibilität nicht möglich, Netzinstanzen zur Laufzeit zu verändern. Dies liegt daran, dass Netzinstanzen konzeptionell mit Objekten gleichgesetzt werden.

In diesem Beitrag werden die grundlegenden konzeptionellen Überlegungen zur Modellierung eines Satzes von Netzfragmenten präsentiert. Insgesamt ergeben die Fragmente ein Petrinetzsystem, das alle wesentlichen Aspekte der Einheitentheorie (Et) (siehe Abschnitt II.) modelliert. Die Gesamtmenge der Fragmente lässt alle Operationen innerhalb eines Petrinetzsystems zu, die für die beliebige Adaption von Petrinetzstrukturen auf der Basis von Referenznetzen benötigt werden. Daher wird zudem die technische Umsetzung und die notwendige Änderung von Renew diskutiert. Der am Arbeitsbereich TGI von Moldt und anderen entwickelte Paose-Ansatz sowie die Et, Mulan oder Renew können hier nur kurz skizziert werden. Ziel dieses Beitrages ist es vielmehr eine Konzentration auf die Kernideen der Modellierung eines Satzes von Netzfragmenten. Die Grundideen der Erweiterung von Renew werden in Form einer möglichen Implementierung dieser Konzepte präsentiert. Die damit ermöglichte dynamische Petrinetzstruktur wird anhand eines einfachen Beispiels illustriert. Die sich für die Entwicklung von verteilten Systemen ergebenden Möglichkeiten werden zum Abschluss diskutiert.

II. Vorarbeiten und Grundlagen

Der Paose*-Ansatz:* Deren Gesamtmenge lässt alle Operationen innerhalb eines Petrinetzsystems zu, die für die beliebige Adaption von Petrinetzstrukturen auf der Basis von Referenznetzen benötigt werden. In den letzten Jahren wurde am Arbeitsbereich der Autoren an den verschiedenen Facetten eines petrinetz- und agentenorientierten Ansatzes zur agentenorientieren Softwareentwicklung (Paose-Ansatz) gearbeitet. So wurden die Techniken (Referenznetze und Java), die Methoden (ablauf- und agentenorientierte Strukturierung der Tätigkeiten und Modellierung) und die Werkzeuge (Renew, CVS, Eclipse etc.) aufeinander abgestimmt. Weiterhin wurde an dem zugrundeliegenden Paradigma und den Prinzipien gearbeitet. Dazu wurde die Einheitentheorie (Et) als konzeptionelle Grundlage des Paose-Ansatzes von den Autoren gemeinsam festgehalten, während insbesondere die technische Umsetzung auf der Basis von Renew durch Volker Tell entwickelt wurde. In diesem Zusammenhang wurde von den Autoren an einer Integrationsbasis gearbeitet, die konzeptionelle Aspekte des Ansatzes durch eine

prototypische Implementierung auf der Basis von Renew exemplarisch evaluiert (siehe dazu insbesondere [Tel05]). Für die Entwicklung agentenbasierter Systeme ist eine hohe Flexibilität der Systeme notwendig. Da auf der Basis der petrinetzbasierten Agentenarchitektur Mulan (siehe [Röl04]) gearbeitet werden soll, lag eine Verwendung von Renew nahe.

Begriffe: Im Rahmen der Systemspezifikation werden zahlreiche Begriffe verwendet, deren Bedeutungen vom jeweiligen Kontext abhängen. Hier sollen kurz einige Begriffe in Anlehnung an [Mol96] umrissen werden.

Im Rahmen von Systemspezifikationen wird zwischen Konzepten und Konstrukten unterschieden. Die Konzepte repräsentieren die prinzipiellen, grundlegenden Aspekte eines Begriffes, seine Essenz. Konstrukte stellen spezifische Ausprägungen dieser Konzepte dar. So kann z.B. Nebenläufigkeit durch Transitionen oder durch Text beschrieben werden.

Petrinetze: Im Rahmen der „Concurrency-Theory“ von Petri (siehe z.B. [Pet87], [Pet96] und [Kum96]) werden konzeptionelle Aussagen zur Nebenläufigkeit geliefert. Bedingungs/Ereignisnetze (B/E-Netze) liefern eine spezielle Repräsentation verschiedener Konzepte. Mit Hilfe von B/E-Netzen lassen sich Nebenläufigkeit, Sequentialität, Konflikt und Konfusion unmittelbar modellieren. Petrinetze liefern in den jeweiligen Ausprägungen spezielle Konstrukte zur Modellierung, unter Beibehaltung der Semantik. Für dieselben Eigenschaften finden sich z.B. in B/E-Netzen und gefärbten Petrinetzen (siehe [Jen92]), wobei sich Konstrukte aus letzteren auf die B/E-Netze zurückführen lassen. Aufgrund der Faltung von bestimmten B/E-Netzausschnitten erreicht man jedoch auf der abstrakteren Ebene neue Konzepte, die auf den einfacheren Konzepten aufbauen und führt dafür auf der höheren Ebene neue Konstrukte ein. Prinzipiell sind diese also auch auf der Ebene der B/E-Netze vorhanden, sie erfahren dort aber keine direkte / explizite Repräsentation.

Die Einheitentheorie*:* In diesem Beitrag wird von einer grundlegenden Modellierungsebene ausgegangen: (unendliche) B/E-Netze. Zwar können diese beliebige Sachverhalte modellieren, aber aufgrund der fehlenden Abstraktionen sind solche Modelle nur in speziellen Fällen für die Softwareentwicklung verwendbar, da die unmittelbare Ausdrucksmächtigkeit fehlt. Die im Folgenden angeführte Et basiert auf den (unendlichen) B/E-Netzen und bedarf ebenfalls einer Einbettung, damit sie praktisch verwendbar wird. Daher wird die Einheitentheorie in den Paose-Ansatz eingebettet, der von der prozessorientierten Sicht der Et profitiert, und diese um agentenorientierte Konzepte zur Strukturierung erweitert.

Eine für die Et wichtige Modellierungstechnik sind die Referenznetze (siehe [Kum02]). Diese erlauben eine ausdrucksstarke Modellierung der grundlegenden Konzepte. Zudem werden sie unmittelbar in Form der Java-Referenznetze von Renew durch intuitive Konstrukte unterstützt. Zusätzlich zu den Konzepten der B/E-Netze stehen somit höhere Modellierungskonzepte und -konstrukte, wie z.B. die synchronen Kanäle und Referenzen zur Verfügung. Die Abbildung dieser Erweiterungen auf die Petrinetztheorie ist, wie in [Kum02] gezeigt, weiterhin möglich.

Der zentrale Gedanke der Einheitentheorie ist es, die Welt als ein großes System zu betrachten. Für dieses System oder einen beliebigen Ausschnitt gibt es ein Petrinetz, z.B. in Form eines (unendlichen) B/E-Netzes oder einer entsprechenden Faltung. Die Abwicklung dieses Petrinetzes erzeugt den bisherigen Prozess der Welt, also alle bisher stattgefundenen Ereignisse. Es gibt eine potenzielle Zukunft in Form von Branching-Prozessen. Ein beliebiges Modell, das in der Systementwicklung erstellt werden soll, ist also immer ein Weltausschnitt. Dieser lässt sich über eine entsprechende Teilmenge aller möglichen Prozesse des Weltnetzes definieren. Das sich ergebende Petrinetz als Modell des Weltausschnittes wird als *das grundlegende Petrinetz* bezeichnet. In [Mol96] wurde in Form der Szenariennetze, in Anlehnung an den Ansatz der Strukturierten Analyse, ein petrinetzbasierter Ansatz präsentiert, der Systemmodelle über die Menge aller möglichen Kausalnetze definiert. Die Et greift dies auf und definiert kleinste Bausteine (Einheit) und ihre Komposition (zusammengesetzte Einheit). Durch die Verwendung der Referenznetze lassen sich diese Modelle entsprechend kompakter darstellen als in [Mol96], wo die Faltung nicht näher beschrieben wurde. Jeder beliebige Ausschnitt des „grundlegenden Petrinetzes“ ist eine Einheit, entweder eine grundlegende Einheit (Stelle oder Transition) oder eine zusammengesetzte Einheit (Netz). Das Konzept der Einheit ist also eine am Betrachter- oder Modellierer orientierte Gruppierung von System- oder Modellteilen zu *einem* abstrakten Ganzen. Dieses Ganze (diese Einheit) ist das umfassende Modell des gewählten Ausschnittes. Einheiten, als abstrakte Modellausschnitte müssen auf dem grundlegenden Netz nicht disjunkt definiert sein. Damit ergibt sich ein einfaches, universelles Denkzeug, das zentrale Eigenschaften verteilter Systeme unmittelbar konzeptionell abdecken kann (siehe dazu den Begriff der „Petrinetze als Denkzeug“ [Mol05]).

Prototypische Implementierung: Zur Validierung der grundlegenden Ideen der Et wurde im Rahmen von [Tel05] unter anderem ein Prototyp in Renew implementiert. Renew zeichnet sich durch eine ausdrucksstarke Plug-In Architektur aus (für die praktische Implementierung siehe [KWD04] und für die konzeptionelle Basis siehe [CDMR05]). Bisher wurde Dynamik in Renew zum einen durch diese neue Plug-In Architektur erreicht, zum anderen durch die Art der Modellierung (siehe insbesondere das Mulan-Rahmenwerk dazu). Obwohl mittels der agentenorientierten Modellierung alles ausgedrückt werden konnte, war für die Flexibilität von Strukturen trotzdem immer noch eine weitere Indirektionsstufe notwendig. So gibt es beispielsweise eine Protokollfabrik, die zur Laufzeit die

Pläne von Agenten, d.h., deren interne Ablaufpläne, verändern kann. Dies wird durch eine zur Laufzeit durchgeführte Interpretation einer symbolischen Repräsentation erreicht. Es fehlt jedoch die direkte Manipulationsmöglichkeit von instanziierten Ablaufmodellen. Da die instanziierten Abläufe direkt in Petrinetzen dargestellt werden, müsste deren Struktur verändert werden. Diese Struktur ist aber, aufgrund der engen Interpretation der Netzinstanzen als (Java)Objekte, nicht variabel. Daher wurde nach einem allgemeinen Konzept für die Flexibilität der Netzstruktur gesucht. Diese wird im folgenden Abschnitt gezeigt. Damit wird die Frage nach einem (minimalen) Satz an Modellierungskonzepten und -konstrukten formuliert, die die Grundlage eines Ansatzes zur Modellierung nebenläufiger Systeme bilden.

III. Ein grundlegendes Petrinetzsystem der Et

Im Folgenden wird eine Menge von Modellierungskonstrukten für die Et vorgestellt. Dabei handelt es sich um die Operationen, die auf den Petrinetzen notwendig sind, um neue Strukturen zu schaffen.

- **Komposition:** Die Komposition ermöglicht die Verbindung von zwei Einheiten im Sinne einer „besteht aus"–Beziehung. Eine Möglichkeit der Realisierung dieser Beziehung besteht in der Erstellung einer neuen Einheit. Diese neue Einheit enthält dann beide zu komponierenden Einheiten. Eine andere Möglichkeit besteht in der Erweiterung einer der zu komponierenden Einheiten durch die andere zu komponierende Einheit. Beide Möglichkeiten werden durch die Abbildungen III.1 (Generierung) und III.2 (Modifikation) illustriert. Dabei sind hier und im Folgenden Einheiten durch eine freihängig gezeichnete, geschlossene Linie hervorgehoben, ohne dass dies eine direkte semantische Bedeutung hat. Die Abbildungen zeigen jeweils auf der linken Seite die Markierung des Netzes vor dem Schalten und auf der rechten Seite die Markierung des Netzes nach dem Schalten. In Abbildung III.1 wird eine neue Einheit aus zwei bestehenden Einheiten zusätzlich erzeugt. In Abbildung III.2 wird eine neue Einheit um eine weitere Einheit ergänzt.
  Ein wichtiger Aspekt ist die Erhaltung der an der Komposition beteiligten Einheiten. Durch diese können bereits komponierte Einheiten mit verschiedenen anderen Einheiten verbunden werden. Als Folge entsteht die Möglichkeit der Modellierung von untereinander nicht disjunkten Einheiten.
- **Verbindung:** Die Verbindung ermöglicht es, zwei (grundlegende) Einheiten durch eine Relation (Kante) miteinander zu verbinden. Die verschiedenen Arten der Relation (Testkante, flexible Kante, etc.) werden im Folgenden nicht berücksichtigt. Vielmehr wird nur das Konzept zwei grundlegende Einheiten miteinander zu verbinden als Netzfragment modelliert (siehe Abbildung III.3).

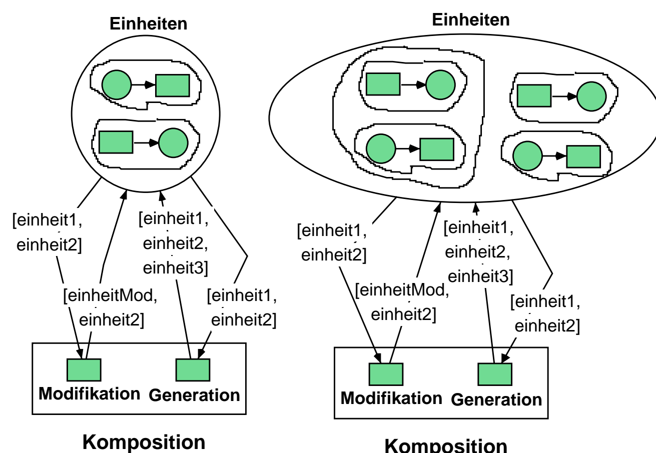


Fig. III.1. Die generierende Komposition

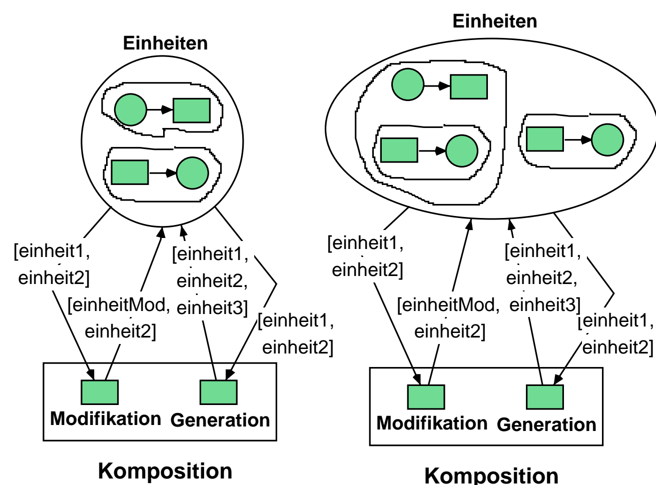


Fig. III.2. Die modifizierende Komposition

- **Belegung:** Die Belegung von Stellen mit Marken ist im Bereich der Petrinetze eine wichtige Operation. Im Rahmen der Et kann auf eine Stelle eine beliebige Einheit als Marke gelegt werden. Bei der Belegung ist ebenso wie bei der Komposition zwischen einer generierenden und modifizierenden Belegung zu unterscheiden (siehe Abbildungen III.4 und III.5). Dabei bezieht sich die Generierung auf die Erstellung einer neuen Einheit, die die Marke und das umgebende Netz in einer neuen Einheit repräsentiert.[1]
- **Erzeugung:** Im Bereich der Erzeugung können beliebige grundlegende Einheiten (Stelle, Transition)

[1]Diese generierende Operation könnte aus einer generierenden Komposition und einer modifizierenden Belegung zusammengesetzt werden.

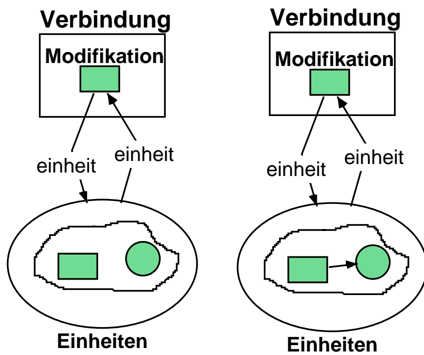


Fig. III.3. Die Verbindung als Petrinetz

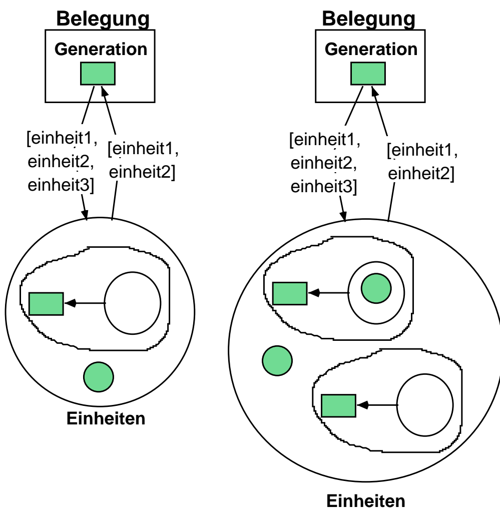


Fig. III.4. Die generierende Belegung als Petrinetz

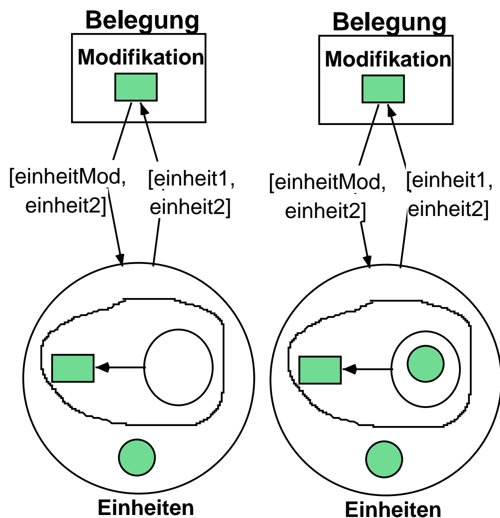


Fig. III.5. Die modifizierende Belegung als Petrinetz

erzeugt werden.

- **Löschung:** Die in [Tel05] beschriebene Löschung von Einheiten (Stellen, Transitionen oder zusammengesetzte Einheiten), das Dekomponieren von zusammengesetzten Einheiten (Netzsichten aufheben / entfernen von Zusammensetzungen)[2], die Entnahme von Einheiten (Marken) und Löschen von Kanten werden hier aus Platzgründen nicht einzeln dargestellt. Es handelt sich lediglich um die entgegengesetzten Operationen auf den jeweiligen Einheiten. Die Netzdarstellungen für das „Löschen“ sind aber in Abbildung III.6 enthalten, auch wenn die einzelnen Operationen vorher nicht gezeigt wurden.

Die vorgestellten Netzfragmente können zu einem neuen Gesamtnetz zusammengefügt werden. Dieses Gesamtnetz (siehe Abbildung III.6) kann als Grundlage für die Implementierung eines die Einheitentheorie umsetzenden Werkzeuges dienen (siehe Abschnitt IV).

Das Gesamtnetz bietet die Möglichkeit, zur Laufzeit Einheiten zu erzeugen bzw. durch die betrachteten Operationen zu verändern. Wenn das Gesamtnetz als Einheit gesehen wird, welche als Referenz sich selbst enthält, kann das Gesamtnetz sich selbst verändern. Aus dieser Überlegung der Selbstreferenz des Gesamtnetzes resultieren verschiedene Fragestellungen. Als erstes steht die Frage nach der Einordnung des Konzeptes der Referenz in die Et. Eine weitere Frage ist, wie das Gesamtnetz eine Referenz auf sich selbst erzeugen kann. Im Folgenden steht jedoch nicht die Beantwortung der aufgeworfenen Fragen im Vordergrund, sondern die Möglichkeiten eines selbstreferenzierenden Gesamtnetzes[3]. Dieses Netz kann zur Laufzeit die Menge der verfügbaren Operationen der Et verändern und die Funktionsweise bestehender Operationen verändern. Die Strukturen eines Werkzeugs auf Basis des Gesamtnetzes können durch den Einsatz des Werkzeugs selbst zur Laufzeit verändert werden. Dies schließt die „Selbstzerstörung“ des Werkzeuges mit ein.

Im folgenden Abschnitt werden verschiedene Möglichkeiten zur Umsetzung vorgestellten Netzfragmente mit Hilfe von Renew betrachtet.

IV. Renew-Erweiterung

Die Umsetzung der in Abschnitt III formulierten Netzfragmente wurde unter Ausnutzung der Plug-In-Architektur von Renew realisiert. Es wurde ein „Einheiten-Plug-In“ implementiert. Dieses Plug-In bietet,

[2]Hier sind je nach Wert oder Referenzsemantik unterschiedliche Semantiken implementierbar.

[3]Da Referenznetze zur Modellierung verwendet werden, ist das Konzept der Referenz in Form einer einfachen Kantenanschrift realisierbar. In [Kum02] wurde gezeigt, wie und mit welchen Folgen sich Referenznetze auf traditionelle Petrinetze abbilden lassen. Damit entsteht zwar durch die Referenzen ein höheres Modellierungskonzept, die Steigerung der Mächtigkeit ist jedoch an die zugrundeliegende Netzklasse und die exakte Semantik gekoppelt (siehe auch [Köh04] für die Diskussion verschiedener Referenznetzsemantiken.

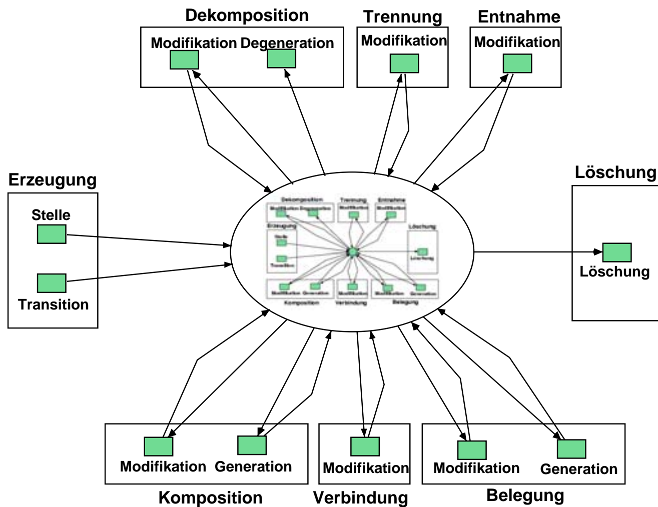


Fig. III.6. Das Gesamtnetz mit Selbstreferenz

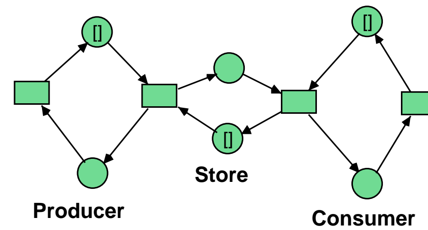


Fig. V.7. Producer/Consumer Darstellung mittels eines Netzes

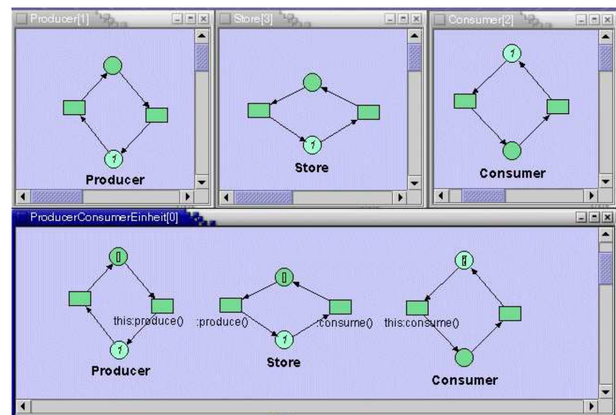


Fig. V.8. Producer/Consumer auf Basis der ET

als eine Erweiterung der bisherigen Netzklassen, das Konzept der Einheit an. Sein Funktionsumfang wird durch die im vorigen Abschnitt vorgestellten Netzfragmenten beschrieben. Die Funktionalität selbst wurde, mit Hilfe von Java-Klassen realisiert.

Zur Realisierung der dynamischen Änderung der Struktur eines Netzes war es notwendig, das bisherige Simulator-Plug-In von Renew zu modifizieren. Auf dem Weg zur Umsetzung der ET wurden verschiedene Prototypen mit unterschiedlichen Ansätzen erstellt. Die Grundidee dieser Ansätze war es die Funktionalität des Einheiten-Plug-Ins nicht durch reinen Java-Code auszudrücken, sondern mit Hilfe von Verfeinerungen, der im vorigen vorgestellten Netzfragmente. Der Nachteil einer Java-basierten Implementierung ist die statische Struktur des Java-Codes zur Laufzeit. Eine Umsetzung mit Netzen würde es ermöglichen, die Funktionen des Werkzeugs auf seine eigene Funktionalität anzuwenden, die durch die Netze gegeben ist. Der Nachteil dieser Umsetzung liegt in der Komplexität der Verfeinerung der einzelnen Operationen und dem damit resultierenden ungünstigeren Laufzeitverhalten.

Ein weiterer Ansatz ist die Nutzung von Java-Referenznetzen. Bei diesem Ansatz werden sowohl Java-Klassen als auch Netze zur Implementierung der Funktionalität verwendet. Ein Nachteil dieses Ansatzes ist ebenso, wie bei dem reinen Java-Ansatz ein Verlust der Änderbarkeit der Implementierung zur Laufzeit des Systems. Ein Vorteil ist, dass Java als eine einfache Möglichkeit für die Implementierung weiterhin zur Verfügung steht und Teile des Systems auf Basis von Netzen mit den daraus resultierenden Vorteilen implementiert werden können.

Insbesondere der dritte Ansatz wurde durch die Erstellung verschiedener Prototypen erfolgreich getestet. Ein Prototyp, basierend auf der Idee des sich selbst referenzierenden Gesamtnetzes, zeigt darüberhinaus die Möglichkeit der erfolgreichen praktischen Einsetzbarkeit der Ideen aus dem vorigen Abschnitt. Bei diesem Prototyp kann sowohl die Funktionsmenge als auch die Art der Funktionen zur Laufzeit verändert werden.

Die Möglichkeiten der Modellierung auf Basis der ET werden im folgenden Abschnitt anhand des „Producer/Consumer“ Beispiel illustriert.

V. Beispiel: Producer/Consumer

Die Abbildung V.7 zeigt eine traditionelle Modellierung des „Producer/Consumers“. Die Modellierung des Beispiels auf Basis der ET wird durch Abbildung V.8 visualisiert. Es wurde eine Producer-Einheit, eine Consumer-Einheit sowie eine Store-Einheit modelliert. Diese drei Einheiten sind durch Komposition in eine Producer/Consumer-Einheit zusammengefügt worden. Die Kommunikation der Einheiten untereinander erfolgt durch synchrone Kanäle. Die Darstellung der einzelnen Einheiten erfolgt jeweils durch eine eigene Netzinstanz.

Eine Besonderheit der Modellierung des „Producer/Consumers“ mit Einheiten liegt in der Änderbarkeit der Struktur der Einheiten zur Laufzeit. Der Producer kann z.B. um einen Zähler für die produzierten Elemente erweitert werden. Diese Erweiterung kann im Gegensatz zu Modellierungen unter dem bisherigen Renew zur Laufzeit stattfinden (siehe Abbildung V.9).

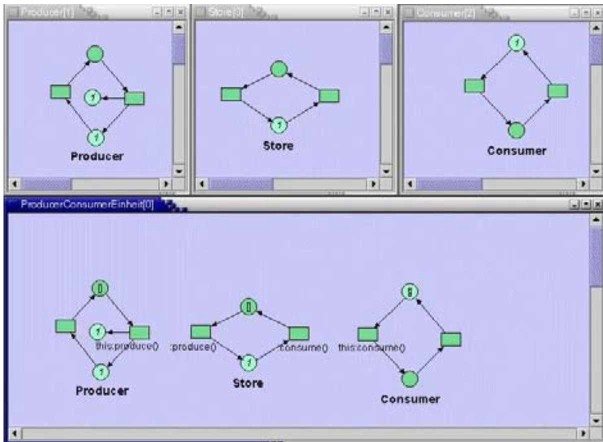


Fig. V.9. Producer/Consumer auf Basis der ET mit einem Zähler

VI. Conclusio

Die Unterstützung des PAOSE-Ansatzes durch die Bereitstellung einer einfachen und universellen Modellierungsphilosophie auf Basis von Petrinetzen erfolgt mittels der EINHEITENTHEORIE. Die in der ET enthaltenen Konzepte zur Modellierung der Veränderung von Petrinetzstrukturen wurden in diesem Beitrag gezeigt. Notwendige Änderungen des Werkzeugs RENEW wurden skizziert. Anhand des traditionellen Producer/Consumer-Beispiels konnte die sich ergebende Ausdrucksmächtigkeit illustriert werden.

Die hier präsentierten Ergebnisse untermauern den PAOSE-Ansatz und erlauben es Teilnehmern eines Projektes, die diesen Ansatz einsetzen, einen einfachen Einblick in die prinzipiellen Möglichkeiten zur Modellierung der Strukturen eines Systems zu gewinnen, um diese Konzepte dann im konkreten Projektkontext mit den bereitstehenden Konstrukten auch gezielt anzuwenden. Die in [CDMR05] präsentierte Modellierung des Plug-In-Konzepts spiegelt die Möglichkeiten im Hinblick auf die Gestaltung von Systemarchitekturen wider. Ebenso liefert das MULAN-Rahmenwerk die konzeptionelle Möglichkeit der unmittelbaren Veränderung von Systemstrukturen. Für die Petrinetzstrukturänderung ist bei den Ergebnissen von Rölke und anderen (siehe insbesondere [Röl04]) jedoch immer eine weitere Indirektion der Modellierung notwendig, die hier entfällt. Dafür ist die unmittelbare Eignung der ET für den konkreten Vorgang der Softwareentwicklung geringer.

Der Anwendung der EINHEITENTHEORIE im Hinblick auf ihr Erklärungspotenzial sind aufgrund der einfachen, fundamentalen und dadurch universellen Konzepte, die sich unmittelbar auf die Theorien von Petri beziehen, kaum Grenzen gesetzt. Mit der Abbildung III.6 wurde ein Petrinetzsystem präsentiert, das als ausführbares Modell eine mögliche Implementierung für die mindestens notwendigen Operationen der ET liefert. Die Beschränkungen liegen in der praktischen Anwendung, die aber durch den PAOSE-Ansatz abgedeckt werden. Das Potenzial der ET-Konzepte wird in [Tel05] anhand eines Leitmodells diskutiert. Dabei werden Ideen aus [Zül05] mit denen der Agentenorientierung, insbesondere dem MULAN-Rahmenwerk, kombiniert.

Literatur

[CDMR05] Lawrence Cabac, Michael Duvigneau, Daniel Moldt, and Heiko Rölke. Modeling Dynamic Architectures Using Nets-Within-Nets. In Philippe Darondeau Gianfranco Ciardo, editor, *26th International Conference on Application and Theory of Petri Nets, ICTAPN 2005, Miami, USA, June 20-25, 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 148–167, Berlin Heidelberg New York, June 2005. Springer-Verlag.

[Jen92] Kurt Jensen. *Coloured Petri Nets: Volume 1; Basic Concepts, Analysis Methods and Practical Use.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin Heidelberg New York, 1992.

[Köh04] Michael Köhler. *Objektnetze: Definition und Eigenschaften*, volume 1 of *Agent Technology – Theory and Applications.* Logos Verlag, Berlin, 2004.

[Kum96] Olaf Kummer. *Axiomensysteme für die Theorie der Nebenläufigkeit.* Berlin: Logos Verlag, 1996.

[Kum02] Olaf Kummer. *Referenznetze.* Logos Verlag, Berlin, 2002.

[KWD04] Olaf Kummer, Frank Wienberg, and Michael Duvigneau. *Renew – User Guide.* University of Hamburg, Faculty of Informatics, Theoretical Foundations Group, Hamburg, release 2.0 edition, June 2004. Available at: http://www.renew.de/.

[KWD05] Olaf Kummer, Frank Wienberg, and Michael Duvigneau. Renew – The Reference Net Workshop. Available at: http://www.renew.de/, 2005. Release 2.02.

[Mol96] Daniel Moldt. *Höhere Petrinetze als Grundlage für Systemspezifikationen.* Dissertation, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, August 1996.

[Mol05] Daniel Moldt. Petrinetze als Denkzeug. In Berndt Farwer and Daniel Moldt, editors, *Object Petri Nets, Processes, and Object Calculi; Report FBI-HH-B-265/05*, pages 51–66. University of Hamburg, Department for Computer Science, August 2005.

[Ode05] Odell, James. The FIPA Agent UML Web Site. http://www.auml.org/, 2005.

[OMG05] OMG (Object Management Group, Inc). UML (Unified Modeling Language Resource Page). http://www.uml.org/, 2005.

[Pet87] Carl Adam Petri. Concurrency Theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Lecture Notes in Computer Science: Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, September 1986*, volume 254, pages 4–24, Berlin Heidelberg New York, 1987. Springer-Verlag.

[Pet96] Carl Adam Petri. Nets, Time and Space. *Theoretical Computer Science*, 153(1–2):3–48, 1996.

[Röl04] Heiko Rölke. *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen*, volume 2 of *Agent Technology – Theory and Applications.* Logos Verlag, Berlin, 2004.

[Tel05] Volker Tell. Prototypische Umsetzung eines Multiagentensystem-basierten Leitmodells. Diplomarbeit, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, 2005.

[Zül05] Heinz Züllighoven. *Object-oriented construction handbook: developing application-oriented software with the tools and materials approach.* dpunkt, Amsterdam u.a., 2005.

Model Checking of Bounded Petri Nets Using Interval Diagrams
Extended Abstract

Alexey A. Tovchigrechko

Brandenburg University of Technology at Cottbus,
Chair Data Structures and Software Dependability,
Cottbus, Germany
alextov@informatik.tu-cottbus.de

Abstract. Model checking is a fully automated approach to formal verification. The main problem of model checking is the state explosion. A number of techniques has been introduced to deal with the problem. Considering Petri Nets, the most efforts have been done on analysis of safe (1-bounded) place/transition nets. Many tools successfully implementing different techniques are available, but there are too few tools supporting efficient analysis of bounded, but not 1-bounded P/T Nets. This paper is a report on the implementation of a symbolic CTL model checker for bounded P/T nets that is based on Interval Decision Diagrams. The implementation supports inhibitor arcs as well as state space construction for a set of initial markings.

1 Introduction

Model checking [BBF$^+$01, CGP01] is an exhaustive, fully automated approach to formal verification. The main problem of model checking is the state explosion. A number of techniques has been developed to deal with the problem. For a recent overview and details see [CGP01]. The most successful approaches are *implicit symbolic techniques* based on variations of binary decision diagrams (BDDs) and *partial-order methods*.

In this paper we will deal with Petri Nets and implicit symbolic model checking. For the running projects of our chair [HK04, HKW04] we need a stable CTL model checker for analysis of bounded Petri Nets. Unfortunately, there are too few available tools implementing symbolic techniques (see the next section). For a more complete version of this paper with examples, tables, algorithms, etc please refer to [Tov04].

2 Extensions of BDDs

BDDs have been applied first for the Petri Nets analysis in [PRCB94]. *Zero Suppressed Decision Diagrams* (ZBDDs) are perfectly suited for analysis of safe (1-bounded) Petri Nets [Spr01]. To analyze bounded, but not safe Petri Nets using BDDs, the number of tokens in a place has to be coded binary. The following problems arise then:

- To save memory and computing power, the coding should be selected such that it covers no more than a necessary integer range - which in general can be not known in advance or can actually be the goal of the analysis!
- The number of variables in a BDD grows fast. Clever variables ordering techniques become even more an issue.
- Integer operations needed for analysis of bounded Petri Nets can not be implemented as efficient as binary ones needed for safe nets.

Different extensions of BDDs have been proposed, we mention here several used for Petri Nets analysis.

Multi-valued Decision Diagrams (*MDDs*) were introduced in [Kam95], they were used for analysis of (stochastic) Petri Nets [MC99, CJMS01]. MDDs can represent functions of the form

$$S_1 \times S_2 \ldots \times S_n \rightarrow \{0, \ldots, m-1\}$$

where $S_i = \{0, \ldots, N_i - 1\}$. In the tool SMART a Petri Net has to be partitioned for analysis, Kronecker operators on sparse boolean matrices are used to encode the transition relation and a new saturation algorithm for the calculation of the state space is used. The approach is very promising for nets with a good partitioning. But it is quite diffi cult to fi nd a good partitioning for the nets met in our projects.

Interval Decision Diagrams (*IDDs*) were introduced in [ST98, ST99]. IDDs can be understood as a generalization of MDDs. Arcs are labeled by (possibly) real intervals (instead of numbers), the number of outgoing arcs of a node can vary, values of IDD variables are not bounded. To analyze Petri Nets, *Boolean IDDs* (IDDs with only two terminal nodes: 0 and 1) *over integer intervals* were used. Some experimental results are provided in [ST98], but there is no tool available.

Natural Decision Diagrams (*NDDs*) were proposed earlier in [LR95, Rid97] for Petri Nets analysis. According to the terminology introduced above, they are Boolean IDDs over integer intervals. Unfortunately, there is no stable tool available.

We have chosen the *Boolean IDDs* over integer intervals for our implementation as they promise a compact representation of state spaces of bounded Petri Nets and allow straightforward implementation of operations needed for analysis of the nets. To improve efficency, firing of transitions was implemented as a direct operation on IDDs, but with a new algorithm differing from [Rid97].

3 Definitions

Definition 1 (Interval Logic Expressions) ILE are defined recursively:

1. TRUE and FALSE are ILE
2. for variables $x_1, \ldots, x_n$, constant $c \in \mathbb{N}_0$, operation $\triangleleft \in \{=, >, <, \geq, \leq, \neq\}$ $\quad x_i \triangleleft c$ is an ILE
3. if F and G are ILE, then $F \wedge G,\ F \vee G,\ \neg F$ are ILE

Definition 2 (Cofactor) $f|_{x_i=b}$ is a cofactor of function f if x_i is replaced by a constant b:

$$f|_{x_i=b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots x_n)$$

Definition 3 (Independence Interval) I is called an independence interval of f with respect to x_i if $f|_{x_i=b} = f|_{x_i=c}\ \ \forall b, c \in I$. We define then $f|_{x_i \in I} = f|_{x_i=b}$ for some $b \in I$.

Definition 4 (Independence Interval Partition) Set $P = \{I_1, \ldots, I_k\}$ is an independence interval partition of $\mathbb{N}_0$ if $I_1, \ldots, I_k$ are independence intervals, $\bigcup_{1 \leq j \leq k} I_j = \mathbb{N}_0$ and $\forall j, m\ \ I_j \cap I_m = \emptyset$.

Definition 5 (Reduced Interval Partition) Independence interval partition is called reduced if

1. it contains no neighbored intervals that can be joined into an independence interval
2. higher bounds of all intervals build an increasing sequence with respect to their indices

It is easy to prove that for some function f a reduced interval partition wrt some variable x is unique.

Definition 6 (Boolean IDD) Boolean IDD is a directed acyclic graph with two kind of nodes $v \in V$. Non-terminal nodes v are labeled by some variable and have v_k outgoing edges labeled with intervals I_j of an independence interval partition $P = \{I_1, \ldots, I_{v_k}\}$ leading to v_k children. Let us define the following labeling functions:

- $\mathrm{var}(v)$ returns a variable
- $\mathrm{part}(v) = \{I_1, \ldots, I_{v_k}\}$ returns labels of the outgoing edges
- $\mathrm{child}_j(v)\ \in V,\ 1 \leq j \leq v_k$ returns children of a node

Terminal nodes are two special nodes labeled only with 0 and 1 and without outgoing edges. On every path from the root to terminal nodes a variable may appear as label of a node only once.

Every decision function $f : \mathbb{N}_0^n \rightarrow \mathbb{B}$ induced by an ILE can be represented by a Boolean IDD with help of Bool-Shannon expansion.

$$f = \bigvee_{1 \leq j \leq k} x_i \in I_k \wedge f|_{x_i \in I_k}$$

Every Boolean IDD over n variables represents a function f that can be written as an interval logic formula over n variables. To find the result of a function, when the values of variables are known $x_1 = a_1, \ldots, x_n = a_n$ one has to follow a path through the graph from the root to a terminal node. In a non-terminal node v an edge labeled with I_j must be chosen if $\mathrm{var}(v) = x_m$ and $a_m \in I_j$. The result of the function is defined by the label of the terminal node reached.

Definition 7 (Ordered Boolean IDD) A Boolean IDD is called ordered with respect to some variable ordering π if on every path from the root to terminal nodes all nodes are ordered with respect to their labels. If there is an edge from node v to a non-terminal node v', then $\mathrm{var}(v) <_\pi \mathrm{var}(v')$.

Definition 8 (Reduced Boolean IDD) A Boolean IDD is called reduced if,

1. the independence interval partitions $\mathrm{part}(v)$ of each non-terminal node v are reduced,
2. each non-terminal node v has at least two different children,
3. there exist no different nodes v and v' such that the subgraphs rooted by v and v' are isomorphic.

If some variable ordering π is defined then for every interval logic function f there is a unique reduced ordered wrt π Boolean IDD, representing this function f.

The proof of the statement is similar to those for ROBDDs [Bry86]. So like ROBDDs, ROBIDDs enjoy the *canonicity* property. From now on we will simply write IDDs meaning reduced ordered Boolean IDDs.

4 Symbolic Analysis of Petri Nets with IDDs

Given a Petri Net with n places we can store any set of its markings, using a characteristic function with n variables induced by an ILE. Set operations can be replaced then by logical operations on characteristic functions. As characteristic functions are induced from ILE, they can be represented by IDDs. We get then a compact and efficient representation for sets of markings.
The set of all reachable markings of a Petri Net N (S, T, F, V, M_0) can be calculated symbolically using Algorithm 1 [Spr01]. For CTL model checking we use the standard symbolic CTL algorithm.

Algorithm 1 (Symbolic state space calculation)

```
1   func ReachableSet (S, T, F, V, M0)
2     func FwdReach (M)
3       New := M
4       repeat
5         Old := New
6         forall t ∈ T do
7             New := New ∪ fire(t, New)
8         od
9       until New = Old
10      return New
11    end
12
13    begin
14      return FwdReach({M0})
15    end
```

To implement a symbolic CTL model checker for Petri Nets, the following main functions have to be provided.

$\text{empty}(F)$	tests, if $F = \emptyset$	$\text{fire}(M, t)$	returns set of markings M' reached, when transition t fires in the set of markings M
$\text{equal}(F, G)$	tests, if $F = G$		
$\text{union}(F, G)$	returns $F \cup G$	$\text{revFire}(M, t)$	returns set of markings M' from which M is reached, when transition t fires
$\text{intsec}(F, G)$	returns $F \cap G$		

We use *Shared* IDDs: several functions over the same set of variables are saved in one directed acyclic graph with multiply roots. This minimizes calculation time and storage space. To access an IDD, we use an index of its root. Implementation of $\text{equal}(F, G)$ becomes trivial, we just have to test, if IDDs F and G have the same root. Implementation of $\text{empty}(F)$ is also trivial.

For a complete discussion on other algorithms and examples please refer to [Tov04]. Here we just provide an implementation of $\text{intsec}(F, G)$, see Algorithm 2. This function is like $\text{union}(F, G)$ and $\text{diff}(F, G)$ a variation of the traditional $\text{apply}(F, G)$ function [Bry86]. Function MakeNode creates a new IDD node. It gets a label for the node, a list of intervals - the labels for the edges, and a list of children. The function takes care that the IDDs remain reduced.

The IDD library and CTL model checker have been implemented in C++ using results of Jochen Spranger and Andread Noack [Noa99, Spr01]. The IDD library was not implemented from the scratch, the library [Noa99] was rewritten to support IDDs instead of ZBDDs. The implementation was tested under Linux and SUN Solaris.

Variables ordering is always an issue for decision diagrams techniques. Static ordering is applied in our implementation. The following heuristic is used: the number of nodes in lower layers of IDD is potentially higher than in upper layers, variables that strongly depend on each other should lay possibly close in the ordering.

An initial marking of a Petri Net can be specified by an ILE. This was used to support the verification of a net for a set of initial markings. Petri Nets with inhibitor arcs are also supported by the implementation.

5 Experimental Results

Before doing CTL model checking, the state space of a Petri Net must be calculated. So it was of main interest, how compact it can be represented and how fast it can be calculated when using IDDs.
Results [1] for three Petri Nets models are provided in Table 1. As it can be seen, IDDs allow quite efficient representation for bounded nets. **RW** is a model for the readers and writers protocol. **FMS** is a model for the flexible manufacturing system [CT93] and **MUL** a Petri Net that weakly computes $x * y$ [PW03]. The nets and more complete description can be found in [Tov04].

6 Future Research

Here are some points of current and future research:

1. We are working on application of interval diagram techniques to the analysis of bounded Timed P/T nets and Timed CTL model checking [RK97]
2. A symbolic LTL model checker for bounded Petri Nets is being implemented using the IDD library.
3. It is interesting to study other heuristics and algorithms for ordering of IDDs variables for bounded Petri Nets.
4. At the moment, generation of counter examples and witnesses is still missing in the CTL model checker.

[1] The benchmark was done on a PC with Intel Pentium 4, 2.8GHz, 512MB RAM, running SUSE Linux 9.0

Algorithm 2 (Binary Operation on IDDs)

```
1   func intsec (F, G)
2     func intsecR (r1, r2)
3       if r1 = 0 ∨ r2 = 0 then return 0 fi
4       if r1 = 1 then return r2 fi
5       if r2 = 1 then return r1 fi
6       if r1 = r2 then return r1 fi
7
8       if r2 < r1 then swap(r1,r2) fi                /* intsec is commutative */
9       if ResultTable[r1, r2] ≠ ∅ then return ResultTable[r1, r2] fi
10      if var(r1) = var(r2) then
11        NewPart := MixIntervals(part(r1),part(r2))
12        forall I_j ∈ NewPart, I_k ∈ part(r1), I_l ∈ part(r2) do
13            if I_j ∩ I_k ∩ I_l ≠ ∅ then
14              NewChild_j := intsecR(child_k(r1), child_l(r2))
15            fi
16        od
17        res := MakeNode(var(r1), NewPart, NewChild)
18      elseif var(r1) < var(r2) then
19        NewPart := part(r1)
20        forall I_j ∈ NewPart do
21            NewChild_j := intsecR(child_j(r1), r2)
22        od
23        res := MakeNode(var(r1), NewPart, NewChild)
24      else
25        NewPart := part(r2)
26        forall I_j ∈ NewPart do
27            NewChild_j := intsecR(child_j(r2), r1)
28        od
29        res := MakeNode(var(r2), NewPart, NewChild)
30      fi
31      ResultTable[r1, r2] = res
32      return res
33    end
34
35    begin
36      B.root := intsecR(F.root, G.root)
37      return B
38    end
```

Example	Time (sec)	States in RG	IDD Nodes	IDD Edges	Iterations
RW 100	0.2	$6 * 10^{2}$	1,853	4,820	102
RW 500	2	$3 * 10^{3}$	6,533	18,574	502
RW 1000	10	$6 * 10^{3}$	18,053	47,120	1002
RW ≤500	13	$3 * 10^{8}$	6,533	17,074	502
FMS 20	1	$6 * 10^{12}$	1,739	8,407	25
FMS 50	18	$4 * 10^{17}$	8,819	59,462	55
FMS ≤50	50	$5 * 10^{20}$	8,819	80,187	55
MUL 6,6	3	$3 * 10^{6}$	2,418	12,022	63
MUL 6,7	5	$6 * 10^{6}$	3,087	16,154	69
MUL 6,10	30	$3 * 10^{7}$	5,562	33,046	87

Table 1. State space generation

References

[BBF+01] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[CGP01] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2001.

[CJMS01] Gianfranco Ciardo, Rob L. Jones, Andrew S. Miner, and Radu I. Siminiceanu. SMART: Stochastic Model Analyzer for Reliability and Timing. In *Tools of Aachen 2001, International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, pages 29–34, 2001.

[CT93] Gianfranco Ciardo and Kishor S. Trivedi. A Decomposition Approach for Stochastic Reward Net Models. *Performance Evaluation*, 18(1):37–59, 1993.

[HK04] Monika Heiner and Ina Koch. Petri Net Based System Validation in Systems Biology. In *Proceedings of the 25th International Conference on Application and Theory of Petri Nets*, LNCS 3099, pages 216–237, 2004.

[HKW04] Monika Heiner, Ina Koch, and Jürgen Will. Validation of Biological Pathways Using Petri Nets - Demonstrated for Apoptosis. *BioSystems*, 75/1-3:15–28, 2004.

[Kam95] Timothy Kam. *State Minimization of Finite State Machines Using Implicit Techniques*. PhD thesis, University of California at Berkeley, 1995.

[LR95] Kurt Lautenbach and Hanno Ridder. A Completion of the S-invariance Technique by Means of Fixed Point Algorithms. Fachberichte Informatik 10–95, Universität Koblenz-Landau, 1995.

[MC99] Andrew S. Miner and Gianfranco Ciardo. Efficient Reachability Set Generation and Storage Using Decision Diagrams. In *Proceedings of the 20th International Conference on Application and Theory of Petri Nets*, LNCS 1639, pages 6–25. Springer, 1999.

[Noa99] Andreas Noack. A ZBDD Package for Efficient Model Checking of Petri Nets (in German). Forschungsbericht, Branderburgische Technische Uinversität Cottbus, 1999.

[PRCB94] Enric Pastor, Oriol Roig, Jordi Cortadella, and Rosa M. Badia. Petri Net Analysis Using Boolean Manipulation. In *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*, LNCS 815, pages 416–435. Springer, 1994.

[PW03] Lutz Priese and Harro Wimmel. *Petri Netze*. Theoretische Informatik. Springer, 2003.

[Rid97] Hanno Ridder. *Analyse von Petri-Netz Modellen mit Entscheidungsdiagrammen*. PhD thesis, Universität Koblenz-Landau, 1997.

[RK97] Jürgen Ruf and Thomas Kropf. Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals. In *Proceedings of the IFIP WG 10.5 International Conference on Correct Hardware Design and Verification Methods*, pages 146–163. Chapman & Hall, Ltd., 1997.

[Spr01] Jochen Spranger. *Symbolic LTL Verification of Petri Nets (in German)*. PhD thesis, Branderburgische Technische Uinversität Cottbus, 2001.

[ST98] Karsten Strehl and Lothar Thiele. Symbolic Model Checking Using Interval Diagram Techniques. Technical report, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, 1998.

[ST99] Karsten Strehl and Lothar Thiele. Interval Diagram Techniques for Symbolic Model Checking of Petri Nets. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 756–757, Munich, Germany, 1999.

[Tov04] Alexey Tovchigrechko. Model Checking of Bounded Petri Nets Using Interval Diagrams. Techn. Report I–05, Branderburgische Technische Uinversität Cottbus, 2004.

Recycling Model Checking Tools for New Application Domains

Christian Stehno

Parallel Systems Group,
Department for Computing Science,
University of Oldenburg, Germany
stehno@informatik.uni-oldenburg.de

Abstract. We present an approach to reuse arbitrary verification tools with Petri net input interfaces for the analysis of arbitrary Kripke structures. The approach reinterprets the Kripke structure as a rather simple Petri net which can be further analysed. This procedure has been successfully applied to the verification of state spaces of Time Petri nets using model checking tools for plain P/T nets.

1 Introduction

Model Checking [2] is a technique to formally check whether or not a property of a system holds for every possible evolution of that system. Thus, model checking algorithms typically explore in a more or less optimized way all possibly reachable states. Due to the problem of state space explosion generation of the complete state space is infeasible in general. Several techniques for generation of reduced but to some extent property preserving state spaces have been developed (e.g. [3, 8]).

Although the combination of state space generation and model checking yields some additional optimization possibilities there is no reason to prohibit splitting the procedure into two separate parts. There are already some tools which use a multi stage process [9] or allow to export the state space into some file format (e.g. [1, 7]). This file interface is usually used for internal purposes only, though, e.g. for suspendable state space generation, or to analyze models that do not fit as a whole into main memory (e.g. [4]). Visualization of the state space is another reason to add access to generated graphs, e.g. in PEP [6], which uses the finite automaton GUI to present automatically layouted state graphs.

In this paper we propose a technique which reuses pre-generated state spaces in conjunction with model checking tools offering a Petri net input interface. This allows for easy implementation of new combinations of model checking and state space generation techniques. Even more, it facilitates implementation of analysis tools for new formal models. By simply implementing a state space enumeration algorithm a large number of already existing model checking tools become accessible. The model checking of Time Petri nets [5] is an example for such newly created model checking tools. Only very few of the Petri net

model checking tools support Time Petri nets, thus a large number of analysis techniques will become applicable to Time Petri nets for the first time.

Some aspects of the state space generation, the translation of the properties, and the conversion of state graphs into Petri nets is described in the next sections. Conclusions and future work are presented in the last section.

2 State Space Generation

The state space graph is a Labelled Transition System (LTS)/Kripke structure, mathematically described as a tuple (V, E, v_0, π) with a set V of vertices (states), a set $E \subseteq V \times V$ of edges (state transitions), the initial state $v_0 \in V$ and a labelling function $\pi(V) \rightarrow \mathcal{P}(\Sigma)$ over some alphabet Σ. The labels in π describe the properties which hold in a given state, all other properties of the alphabet do not hold in that state.

Since state space generation is the main performance bottleneck in model checking a large number of optimizations have been proposed. With on-the-fly techniques the property checking is done during generation. This optimization only helps if the property can be proven with a small subset of states, and thus before the whole state space is generated. Otherwise the additional overhead of repeated checking in each state would become a drawback. Additionally, only simple properties which can be evaluated with only the knowledge of the state currently visited can be checked. Reduction techniques such as Stubborn sets [8] and partial order reduction [3] try to generate only a part of the whole state space. If that chosen part can be shown to be complete in the sense that the skipped part does not possess any different behaviour model checking can be done on the usually much smaller part. Thus, reduction techniques usually create equivalent classes of states and use only one representative of each class as a vertex of the state graph. To prove equivalence of behaviour of the reduced state graph bisimulation between the two graphs with respect to a chosen temporal logic is shown. Therefore, reduction techniques preserve only a part of the properties which could have been proven on the full state graph.

The optimizations that will apply to the proposed method are mainly reduction techniques. Due to the separation into state space generation and analysis with an intermediate transformation the integration of on-the-fly analysis does not seem to be appropriate.

The state graph creation considered in this paper is starting with Petri nets. The LTS for Petri nets is their rechability graph, i.e. each vertex of the graph represents a reachable marking of the net, and each edge represents a transition that may fire at the given marking, thereby yielding the marking represented by the destination state. For Time Petri nets the markings are enhanced by additional time constraints, thus states with the same marking may be still different due to different clock values.

For Petri nets the most commonly used atomic propositions used for the alphabet Σ are the places of the net. A proposition p_1 holds iff in a given state a token is on place p_1. The proposed technique will handle any finite set of atomic

propositions, though, as long as any property can be evaluated using only the current state of the net.

The tools considered for state space generation, yet, are LoLA [7], PROD [9], Maria [4], and Tina [1]. The first three operate on plain P/T nets and high-level Petri nets, while the latter uses Time Petri nets. Using non-timed Petri nets as a source allows to apply different optimization techniques which are not supported by the model checker. Integration of Time Petri nets with commonly used model checking tools creates merely completely new model checking tools. The lack of Time Petri net model checking tools will be substantially reduced by the proposed techniques, which is why we considered that combination as a primary testbed.

The Tina tool provides generation of State classes in combination with various reduction techniques. Tina provides different output formats for the generated state space which simplifies the parsing in some cases. Most formats only support place names as atomic propositions, whereas the verbose Tina text output also includes additional clock constraints.

3 Property Translation

Due to the proposed intermediate model change care has to be taken in order to preserve the meaning of properties which describe desired or undesired behaviour using temporal logics. Simple properties such as deadlocks and reachability are better checked using special purpose analysis tools or on-the-fly model checking in order to benefit from the fact that these properties can always be checked during state space generation. General purpose model checking needs a more generic approach and is thus better suitable for the proposed procedure.

In each state a subset of all possible properties (from Σ) is present, i.e. holds. Each property describes a dedicated, discrete property which may hold in a given state, and whose truth value is independently evaluatable at each state in order to determine the truth value upon state space generation.

Although marked places are the most commonly used atomic properties for Petri net analyis a lot of other properties come into mind. For non safe Petri nets more complex comparison operators can be introduced, e.g. $p_1 > 4$ for p_1 having more than 4 tokens. For Time Petri nets the additional clock constraints can be considered to add some more propositions expressing the additional properties.

4 State Graph to Petri Net Translation

The overall idea of the translation from state graphs to Petri nets is to interpret the graph as a finite automaton. A state graph (V, E, v_0, l) consists of only one connected component. Thus, the translation can be easily achieved by using the automaton (V, E, v_0). Labels will be considered later on.

The finite automaton is in turn interpreted as a Petri net using the automaton states as places of the net, as shown in Fig. 1. Each edge is replaced by a triple

arc, transition, arc connecting two places in the same direction as the edge of the automaton. This standard translation yields a state machine which represents the state changes of the LTS as Petri net structures.

Fig. 1. Transformation from automaton to Petri net

In order to effectively represent the atomic propositions in a way such that arbitrary combinations of single properties can be used in formulae a special translation is used. Each atomic proposition $p \in \Sigma$ is represented by its own place which will be marked by a token while the property holds. Thus, all proposition places are initially marked iff the proposition holds in the initial state. Transitions are connected to proposition places such that properties which do not hold anymore after a state change have their tokens removed and those which newly hold get a token. The result of adding properties to the net from Fig. 1 is shown in Fig. 2. Since transition t_2 does not change the truth value of p_1 the token is left untouched. The truth value of p_4 does not change either, such that t_2 need not care for any other proposition, whereas t_1 changes p_1 and p_4 and leaves p_2 untouched. If s_1 was the initial state for the original LTS tokens would be added to places s_1, p_1, and p_2 as state marker and initially true propositions.

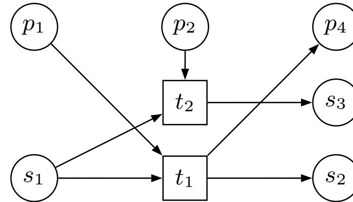


Fig. 2. Adding state properties

Using the proposed Petri net transformation the translation of the properties reduces to a simple renaming to the newly created proposition places. Due to the separated representation of each single proposition all properties are distinguishable by the model checker.

5 Conclusion and Future Work

The proposed approach of combining state space generation tools and model checking tools to build new analysis tools has been successfully applied to the

analysis of Time Petri nets. Using the Tina state space generation algorithms and an automatic translation to P/T nets various Time Petri nets from the literature have been verified. The model checking techniques applied comprise BDD-based and enumerative verification of CTL and LTL formulae.

Although verification of the examplary nets was feasible general applicability of the approach and competetivity still have to be shown in a larger benchmark. Nevertheless, the proposed techniques are useful for prototyping analysis tools for new formal methods. Implementing the state space enumeration is usually much simpler than implementing additional model checking algorithms.

In order to reflect the flexibility of the presented approach in the implementation as well a general framework will have to be defined. The current ad-hoc implementation of the transformation will have to be reengineered according to the new APIs using a plugin mechanism to support arbitrary LTS descriptions and atomic propositions. Additionally integration into the finite automata GUI of the PEP tool [6] will be considered.

References

1. B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, 42(14), July 2004.
2. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
3. P. Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In E.M. Clarke and R.P. Kurshan, editors, *2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer-Verlag, 1990.
4. Marko Mäkelä. Maria: Modular Reachability Analyser for Algebraic System Nets (Tool Presentation). In Javier Esparza and Charles Lakos, editors, *Application and Theory of Petri Nets*, volume 2360 of *Lecture Notes in Computer Science*, Adelaide, Australia, 2002. Springer-Verlag.
5. P. Merlin and D. Farber. Recoverability of Communication Protocols – Implication of a Theoretical Study. *IEEE Transactions on Software Communications*, 24:1036–1043, 1976.
6. The PEP tool. http://peptool.sourceforge.net.
7. Karsten Schmidt. LoLA: A low level analyser. In Nielsen, M. and Simpson, D., editors, *21st International Conference on Application and Theory of Petri Nets (ICATPN 2000)*, volume 1825 of *Lecture Notes in Computer Science*, pages 465–474, Aarhus, Denmark, 2000. Springer-Verlag.
8. Antti Valmari. Alleviating State Explosion during Verification of Behavorial Equivalence. Technical Report A-1991-4, University of Helsinki, 1992.
9. Kimmo Varpaaniemi, Keijo Heljanko, and Johan Lilius. PROD 3.2 - An Advanced Tool for Efficient Reachability Analysis. In Orna Grumberg, editor, *Computer Aided Verification: 9th International Conference, CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 472–475, Haifa, Israel, 1997. Springer-Verlag.

Unentscheidbarkeit des Beschränktheitsproblems für allgemeine Referenznetze

Roxana Dietze
Universität Hamburg, Fachbereich Informatik
Vogt-Kölln-Straße 30, D-22527 Hamburg
dietze@informatik.uni-hamburg.de

Zusammenfassung

In diesem Artikel wird gezeigt, dass das Beschränktheitsproblem unentscheidbar für allgemeine Referenznetze (Kummer, 2002) ist. Der Beweis benutzt die Unentscheidbarkeit dieses Problems für Netze mit Löschkanten [*engl.* reset nets, (Araki u. Kasami, 1977)].

1 Einleitung

Die Referenznetze wurden in (Kummer, 2002) eingeführt. Es wurde gezeigt, dass das Erreichbarkeitsproblem unentscheidbar für diese Klasse von Netzen ist (siehe (Kummer, 2000) für Details).

Wir werden im folgenden die nächste Definition der Referenznetze benutzen:

Definition 1 *Die Struktur* $N_i = (P_i, T_i, C_i, c_i, R_i, r_i, F_i, W_i, V_i)$ *ist ein Referenznetzmuster, falls gilt:*

- P_i *ist eine endliche Menge von Plätzen;*
- T_i *ist eine endliche Menge von Transitionen,* $T_i \cap P_i = \emptyset$*;*
- C_i *ist eine endliche Menge von Kanälen, die den Transitionen zugewiesen sind.*

  Die Kanäle sind gerichtet (downlinks und uplinks).
- c_i *ist die Kanalzuweisungsfunktion* ($c_i : T_i \to MS(C_i)$)*.*
- R_i *ist eine endliche Menge, die die Art der Neuerzeugung von Referenzen festlegt.*
- r_i *ist die Referenzerzeugungszuweisungsfunktion* ($r_i : T_i \to MS(R_i)$)*.*
- $F_i \subseteq (P_i \times T_i) \cup (T_i \times P_i)$ *ist die Flußrelation;*
- $W_i : F_i \to (\{*\} \cup V_i \to \mathbb{N})$ *ist die Gewichtsfunktion;*

  $\{*\}$ *stellt eine einfache Marke dar.*
- V_i *ist eine endliche Menge von Variablen, die von der Schaltrelation der Transitionen benutzt werden.*

Das Referenznetzmuster (*engl.* net template) ist als eine statische Struktur zu betrachten, der zwar kein Verhalten zugewiesen werden kann, die aber wichtige Informationen über die Struktur und Anfangsmarkierung eines Referenznetzes besitzt. Es ist daher ähnlich der Definition einer Klasse in einer objekt-orientierte Programmiersprache (zum Beispiel Java).

Die dynamischen Eigenschaften eines Referenznetzes werden mittels Referenznetzinstanzen (*engl.* net instance) beobachtet. Diese Instanzen können als Objekte einer bestimmten Klasse (Referenznetzmuster) betrachtet werden. Jede Referenznetzinstanz besitzt eine eigene Identität. Es wird angenommen, wie auch in (Kummer, 2002), dass der Formalismus fähig ist, unendlich viele Identitäten zu erzeugen, und verschiedene Identitäten auf Gleichheit zu prüfen.

Eine Markierung eines Netzes N wird jedem Platz des Netzes eine Multimenge von schwarzen Marken und Referenzen an verschiedene Referenznetzinstanzen zuweisen.

Beispiel 1 *In der Abbildung 1 wird ein kleines Muster angezeigt, das Transitionen mit Kanäle und mit Netzerzeugung besitzt. Die Anschriften der Kanten definieren die Gewichtsfunktion. Das Netz wird mit einfachen Marken oder Referenzen (oder beide) markiert.*

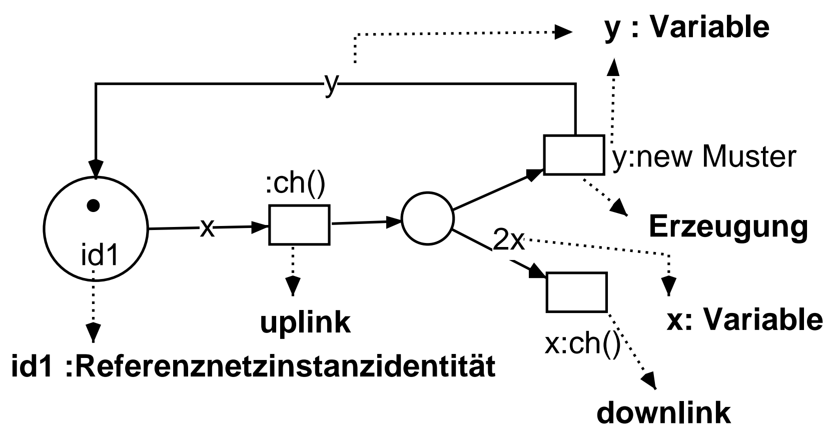


Abbildung 1: Ein Referenznetzmuster mit Markierung $(1'\{*\} + 1'id1, 0)$

Da eine Markierung nicht nur aus Marken, sondern auch aus verschiedenen Referenzen besteht, wird, wie auch für gefärbte Netze, der Begriff der Bindung benötigt, um über das Schalten einer Transition sprechen zu können. Durch eine Bindung wird es festgelegt, welche Referenzen die Bedingungen der Transition erfüllen.

Die Schaltregeln werden intuitiv erläutert:

1. eine Transition, die keine Kanäle besitzt, ist genau wie bei gefärbten Petrinetzen zu betrachten (die Bindung ist wichtig),

2. die Transitionen, die Kanäle besitzen, müssen sich synchronisieren (immer ein uplink mit ein downlink derselben Art, die Bindung der Variablen spielt auch hier eine wichtige Rolle)

3. die Transitionen, die neue Instanzen erzeugen können, werden immer **neue** Identitäten erzeugen.

Wir definieren im folgenden die Eigenschaft der Beschränktheit. Eine frühere Version der Definitionen ist in (Dietze, 2004) zu finden.

Definition 2 *a) Ein Netz $N = (P, T, C, c, R, r, F, W)$ heißt unbeschränkt falls $\nexists n \in \mathbb{N}$, mit $|M(p)| \leq n, \forall p \in P$ und $\forall M$ erreichbare Markierung von N, wo $|M(p)|$ durch die Summe der Anzahl der Marken und Referenzen, die im Platz p liegen, gegeben ist.*

b) Ein Netz N heißt 1-pseudo-beschränkt, falls N nicht unbeschränkt ist, und die Anzahl von Netzexemplaren, die referenziert werden können, unbeschränkt ist.

c) Ein Netz N heißt 2-pseudo-beschränkt, falls N nicht unbeschränkt ist, und eines der referenzierten Exemplare unbeschränkt ist.

d) Ein Netz, das nicht unbeschränkt, 1-pseudo-beschränkt oder 2-pseudo-beschränkt ist, ist ein beschränktes Netz.

Die Pseudo-beschränktheit kann als eine versteckte Unbeschränktheit betrachtet werden. Diese Eigenschaft ist weiter verfeinert worden, da es zwei Möglichkeiten gibt: entweder kann man unendlich viele Netzexemplare, die referenziert werden können, erzeugen oder einige referenzierte Exemplare unbeschränkt sein können.

Für die Analyse eines Netzes ist es sehr wichtig, die pseudo-beschränkten Netze zu erkennen, um zum Beispiel Speicher-Probleme bei der Simulation des Netzverhaltens zu vermeiden.

Definition 3 *(Beschränktheitsproblem) Gegeben ein Netz $N = (P, T, C, c, R, r, F, W)$: ist N beschränkt, 1-pseudo-beschränkt, 2-pseudo-beschränkt oder unbeschränkt?*

Wir werden im nächsten Abschnitt die Unentscheidbarkeit des Beschränktheitsproblems für Referenznetze beweisen. Zuerst wird aber an einige Definitionen und Ergebnisse für Resetnetze erinnert.

2 Reset-Netze vs. Referenznetze

Definition 4 *Ein Reset-Netz ist ein P/T Netz mit speziellen Reset-Kanten: (P, T, F, F_R). F_R steht für die Menge der Reset-Kanten, $F_R \subseteq T \times P$.*

Das Schalten einer Transition t, die auch Komponente einer Reset-Kante ist ($\exists p \in P, (t, p) \in F_R$), wird den Platz p von allen Marken entleeren.

Formell, wenn t in m aktiviert ist ($m[t >$) und $(t, p) \in F_R$, dann ist $m[t > m'$ ein gültiger Schritt des Reset-Netzes, und es gilt $m'(p) = 0$ falls $(t, p) \notin F$ und $m'(p) = 1$ falls $(t, p) \in F$.

Wir werden weiter eine Konstruktion wie die in (Lomazova u. Schnoebelen, 2000) benutzen, und werden beweisen, dass ein Reset-Netz durch ein Referenznetz simuliert werden kann.

Nehmen wir an, dass Abbildung 2 ein Auschnitt aus einem Reset-Netz ist.

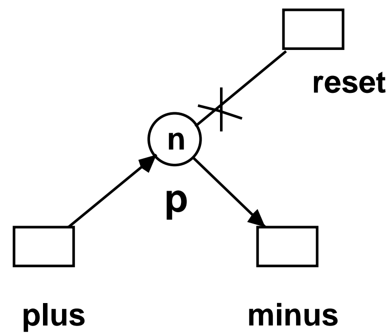


Abbildung 2: Ein Ausschnitt aus einem Reset-Netz

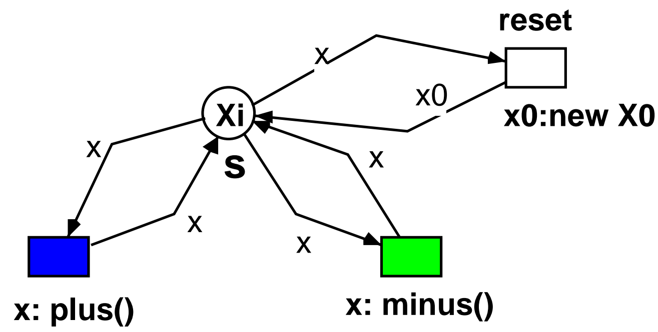


Abbildung 3: Ein Ausschnitt aus einem Referenznetz

Wir können für diesen Ausschnitt ein äquivalentes Referenznetz konstruieren, wie die Abbildung 3 zeigt.

Der Ausschnitt 3 simuliert genau das Verhalten des Ausschnittes aus Abbildung 2. Die Idee ist ähnlich derjenigen in (Lomazova u. Schnoebelen, 2000). Wir werden für jedes Reset-Netz ein Referenznetz konstruieren. Es kann gezeigt werden, dass das Netz sich wie das Reset-Netz verhält (das Referenznetz simuliert das Reset-Netz).

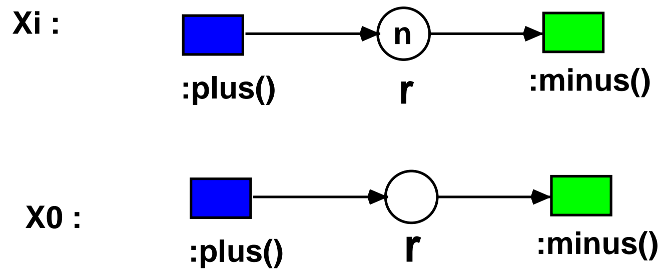


Abbildung 4: Referenznetzmuster $X0$ und X_i

Theorem 1 *Für jedes Reset-Netz gibt es ein Referenznetz, welches das Reset-Netz simuliert.*

Theorem 2 *Das Problem der Beschränktheit ist für Reset-Netze unentscheidbar.*

Beweis : Siehe (Dufourd u. a., 1998, 1999) für Details. □

Die Klasse von Referenznetze, die Reset-Netze simulieren, kann entweder beschränkt, oder pseudo-beschränkt sein.

Um das Beschränktheitsproblem lösen zu können, sollte man beantworten können, ob das Referenznetz pseudo-beschränkt 2 ist oder nicht. Diese Frage ist aber äquivalent zur Frage, ob das Reset-Netz beschränkt oder nicht beschränkt ist.

Da für ein Reset-Netz nicht entschieden werden kann, ob das Netz beschränkt oder unbeschränkt ist (Theorem 2), bedeutet dies, das für diese Klasse von Referenznetzen nicht entscheiden werden kann, ob eine von den erzeugten Referenznetzinstanzen unbeschränkt ist oder nicht.

Diese Ergebnisse und die Konstruktion von oben, haben zur Folge, dass die Eigenschaft der Beschränktheit unentscheidbar für die Klasse der Referenznetze ist.

Theorem 3 *Das Beschränktheitsproblem ist für Referenznetze unentscheidbar.*

Beweis : Die Aussage basiert auf der Tatsache, dass man ein Reset-Netz mit Hilfe eines Referenznetzes simulieren kann (Theorem 1), und für Reset-Netze ist dieses Problem unentscheidbar (Theorem 2). □

3 Zusammenfassung

In diesem Artikel wurde gezeigt, dass das Beschränktheitsproblem unentscheidbar für allgemeine Referenznetze ist, da dieses Problem unentscheidbar für die Klasse der Referenznetze, die Reset-Netze simulieren, ist.

Wir erwarten aber, dass für kleinere Klassen von Referenznetzen, wie z.B. die Klasse von Referenznetzen, die Reset-Netze mit einer Reset-Kante simulieren, sowohl das Beschränktheitsproblem, als auch das Erreichbarkeitsproblem entscheidbar werden.

Literatur

[Araki u. Kasami 1977]
ARAKI, Toshiro ; KASAMI, Tadao: Some Decision Problems Related to the Reachability Problem for Petri Nets. In: *Theoretical Computer Science* (1977), Nr. 3, S. 85–104

[Dietze 2004]
DIETZE, Roxana M.: Konzepte von Referenznetzen. In: *Proceedings of the 11thWorkshop on Algorithms and Tools for Petri Nets (AWPN 04), Ekkart Kindler (ed.), Bericht tr-ri-04-251*, 2004, S. 43–48

[Dufourd u. a. 1998]
DUFOURD, Catherine ; FINKEL, Alain ; SCHNOEBELEN, Philippe: Reset Nets between Decidability and Undecidability. In: LARSEN, Kim G. (Hrsg.) ; SKYUM, Sven (Hrsg.) ; WINSKEL, Glynn (Hrsg.): *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP'98)* Bd. 1443. Springer, 103-115

[Dufourd u. a. 1999]
DUFOURD, Catherine ; JANCAR, Petr ; SCHNOEBELEN, Philippe: Boundedness of reset P/Tnets. In: *Lecture Notes in Computer Science: Automata, Languages and Programming* 1644 (1999), S. 301–310

[Kummer 2000]
KUMMER, Olaf: Undecidability in Object-Oriented Petri Nets. In: *Petri Net Newsletter* (2000), Nr. 59, S. 18–23

[Kummer 2002]
KUMMER, Olaf: *Referenznetze*. Logos Verlag, 2002

[Lomazova u. Schnoebelen 2000]
LOMAZOVA, Irina A. ; SCHNOEBELEN, Philippe: Some Decidability Results for Nested Petri Nets. In: BJØRNER, Dines (Hrsg.) ; BROY, Manfred (Hrsg.) ; ZAMULIN, Alexandre V. (Hrsg.): *Proceedings of the 3rd International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI'99)* Bd. 1755. Springer, 208-220

Schwach beschränkte Petrinetze

Jörg Desel

Katholische Universität Eichstätt-Ingolstadt
Lehrstuhl für Angewandte Informatik
85071 Eichstätt, Germany
joerg.desel@ku-eichstaett.de

Zusammenfassung Ein Petrinetz ist schwach beschränkt, wenn die unbegrenzte Zunahme von Marken auf Stellen durch die Auswahl nebenläufig aktivierter Transitionen verhindert werden kann. Dieser Beitrag motiviert und definiert schwache Beschränktheit von Petrinetzen. Für kommunizierende Zustandsmaschinen wird eine effiziente Charakterisierung schwacher Beschränktheit angegeben.

1 Einleitung

Das in Abbildung 1 dargestellte Erzeuger/Verbraucher-System ist unbeschränkt, denn die Pufferstelle s kann beliebig viele Marken erhalten. Falls der Erzeuger schneller Marken erzeugt als der Verbraucher verbraucht, wird es auch tatsächlich eine stets steigende Markenzahl auf s geben. Falls umgekehrt der Verbraucher schneller als der Erzeuger ist (bzw. wäre, wenn er nicht stets auf die Marke auf der Stelle s warten müsste), wird die Markenzahl auf s nie größer als eins, denn jede erzeugte Marke wird konsumiert, bevor eine weitere Marke produziert wird. Falls aber der Verbraucher nur durchschnittlich schneller ist als der Erzeuger, dann kann die Markenzahl auf s zwar beliebig wachsen, wird aber langfristig nicht gegen unendlich streben.

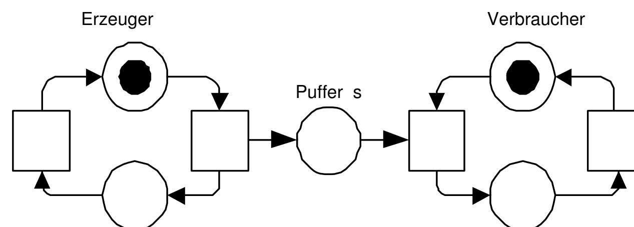


Abbildung 1. Ein schwach beschränktes Erzeuger-Verbraucher System

Abbildung 2 zeigt ebenfalls ein System mit einem Erzeuger und einem Verbraucher. Hier erzeugt der Erzeuger pro Runde zwei Marken auf s_1 und eine Marke auf s_2. Unabhängig von der Geschwindigkeit des Verbrauchers wird die Markenzahl auf s_1 beliebig wachsen, denn der Verbraucher kann wegen des Puffers s_2 niemals schneller sein als der Erzeuger. Mit jeder Runde des Erzeugers nimmt die Markenzahl auf s_1 aber um wenigstens eins zu. Die Unbeschränktheit dieses Netzes wird sich in jedem sequentiellen Ablauf zeigen, sofern der Erzeuger immer weiter Marken erzeugt. Dies ist bei dem ersten Beispiel nicht der Fall.

Anstatt Annahmen über die jeweiligen Geschwindigkeiten von Erzeuger und Verbraucher zu treffen, wollen wir nun annehmen, dass eine übergeordnete Instanz – ein Scheduler – bei nebenläufig aktivierten Transitionen auswählt, welche als nächstes schalten soll. Man sieht leicht, dass auf diese Weise eine Markenzunahme der Pufferstelle s des ersten Beispiels

verhindert werden kann, während die Stelle s_1 des zweiten Beispiels unabhängig von einem derartigen Steuerungsmechanismus überlaufen wird.

Wir nennen die Stelle s und auch das gesamte Netz aus Abbildung 1 schwach beschränkt. Die Stelle s_1 des Netzes aus Abbilung 2 ist dagegen nicht schwach beschränkt, und damit ist dies auch für das Netz nicht der Fall.

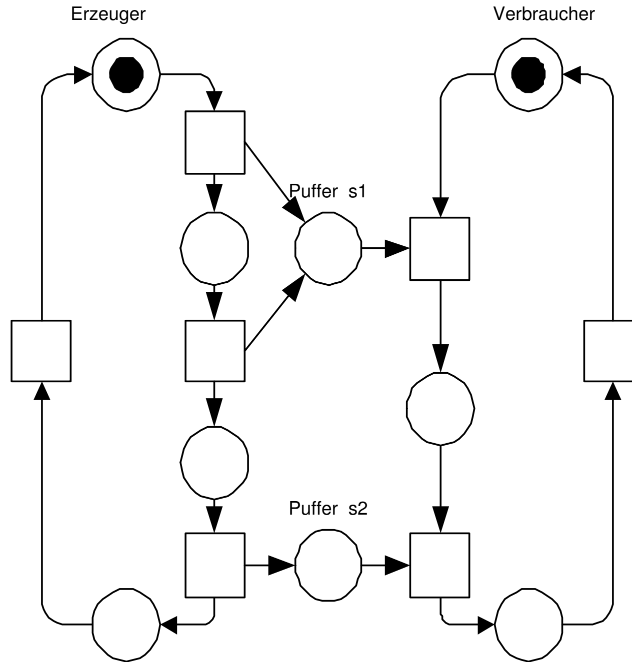


Abbildung 2. Ein nicht schwach beschränktes Erzeuger-Verbraucher System

Die Motivation für die Beschäftigung mit schwacher Beschränktheit kommt aus zwei Quellen: In [1] geht es um die Synthese reaktiver Systeme aus nebenläufigen Programmen, die über Puffer miteinander kommunizieren. Mehrere nebenläufige Programme sollen auf einem sequentiellen Rechner (bzw. auf einem Schaltkreis) ausgeführt werden, ohne dass die Puffer überlaufen. Ein dafür verantwortlicher Scheduler bestimmt stets, welcher Programmschritt als nächstes auszuführen ist. Um den Aufwand für diesen Scheduler zu reduzieren, sollen seine Entscheidungen nicht von den jeweiligen Daten, d.h. Variablenbelegungen abhängen, sondern nur auf Grundlage des Kontrollflusses der Programme erfolgen. Der Kontrollfluss eines jeden Programms wird durch ein 1-sicheres Petrinetz dargestellt, die Puffer durch weitere potenziell unbeschränkte Stellen, die diese Netze verbinden. Da datenabhängige Verzweigungen möglich sind, darf der Scheduler nicht die Entscheidung über die Auswahl zwischen alternativen Transitionen treffen, sondern nur eine Reihenfolge nebenläufig aktivierter Transitionen bestimmen. Zudem soll er keine aktivierte Transition beliebig lange verzögern. Wichtig ist zunächst die Fragestellung, ob ein derartiger Scheduler existieren kann. Dies ist offensichtlich im Beispiel aus Abbildung 1 der Fall, im Beispiel aus Abbildung 2 ist es nicht der Fall (wenn man den Erzeuger und den Verbraucher jeweils als einfaches sequentielles zyklisches Programm versteht).

Die zweite Quelle für die Überlegungen in diesem Beitrag sind Message Sequence Charts (MSCs). Diese werden für die Spezifikation von Kommunikationsprotokollen zwischen asynchronen Prozessen verwendet, die ebenfalls über Puffer miteinander kommunizieren [3]. In [4]

werden verschiedene Beschränktheitsbegriffe für diese Puffer untersucht. Wenn man MSCs als Abläufe von Petrinetzen betrachtet, so entspricht eine dieser Beschränktheitsbegriffe der hier definierten schwachen Beschränktheit von Petrinetzen. Tatsächlich nennt Anca Muscholl z.B. auf ihrem eingeladenen Vortrag auf der ACSD 2005 entsprechende MSCs *weakly bounded* – den Begriff schwache Beschränktheit habe ich von ihr entlehnt, aber in keiner Arbeit von ihr schriftlich gefunden.

Im folgenden Abschnitt wird schwache Beschränktheit zunächst formal definiert. Im dritten Abschnitt wird eine hinreichende und notwendige Bedingung für schwache Beschränktheit einer speziellen Klasse von Petrinetzen – gekoppelte Zustandsmaschinen – angegeben. Diese Bedingung ähnelt der Charakterisierung lebendiger und beschränkter Free-Choice Netze durch den Rang ihrer Inzidenzmatrix (Rank Theorem [2]).

2 Definition schwacher Beschränktheit

Sei N im folgenden ein zusammenhängendes Petrinetz mit endlicher und nichtleerer Stellenmenge S, endlicher und nichtleerer Transitionenmenge T, Flussrelation F und Anfangsmarkierung m_0. A sei eine Teilmenge von S (Alternativenmenge).

Wenn im Beispiel aus Abbildung 1 ausschließlich Transitionen des Erzeugers schalten, dann wächst die Markenzahl auf s unweigerlich. Eine Transition des Verbrauchers ist ebenfalls persistent aktiviert, schaltet aber nicht. Derartige Abläufe wollen wir ausschließen, d.h., Fortschritt (Progress) in allen Systemteilen annehmen.

Definition 1. *Eine (endliche oder unendliche) Schaltfolge $m_0\, t_1\, m_1\, t_2\, m_2 \ldots$ von N mit folgender Eigenschaft heißt fortschreitend: Falls eine Transition t von einer in der Schaltfolge vorkommenden Markierung m_i aktiviert wird, so gilt*

$$(\{t\} \cup ({}^\bullet t)^\bullet) \cap \{t_{i+1}, t_{i+2}, \ldots\} \neq \emptyset$$

(die Transition t oder eine mit ihr in strukturellem Konflikt stehende Transition schaltet).

Wir wollen die Reihenfolge nebenläufiger Transitionen in fortschreitenden Schaltfolgen so verändern, dass die dann entstehende Schaltfolge im traditionellen Sinn beschränkt ist, die Menge der erreichten Markierungen also endlich ist. Da in unendlichen Schaltfolgen unendlich viele Vertauschungen dafür notwendig sein können, gehen wir formal anders vor und verlangen nur, dass Paare alternativer Transitionen, also Transitionen mit gemeinsamer Vorbereichsstelle, in unveränderter Reihenfolge schalten.

Definition 2. *Zwei Schaltfolgen $m_0 t_1 m_1 t_2 m_2 \ldots$ und $m_0 t'_1 m'_1 t'_2 m'_2 \ldots$ heißen alternativenverträglich bezüglich A (A-verträglich), wenn für jede Stelle s aus A die Projektion von $t_1\, t_2\, t_3 \ldots$ auf $s^\bullet$ (ensteht durch Streichung aller Transitionen $t \notin s^\bullet$) mit der Projektion von $t'_1\, t'_2\, t'_3 \ldots$ auf $s^\bullet$ übereinstimmt.*

Definition 3. *Eine Stelle s von N heißt schwach k-beschränkt, wenn zu jeder fortschreitenden Schaltfolge $m_0\, t_1\, m_1\, t_2\, m_2 \ldots$ von N eine Permutation $t'_1\, t'_2\, t'_3 \ldots$ von $t_1\, t_2\, t_3 \ldots$ existiert, so dass $m_0\, t'_1\, m'_1\, t'_2\, m'_2 \ldots$ A-verträgliche Schaltfolge zu $m_0\, t_1\, m_1\, t_2\, m_2 \ldots$ ist und $m_0(s), m_1(s), m_2(s), \ldots \leq k$ gilt.*

Definition 4. *Eine Stelle s von N heißt schwach beschränkt, wenn zu jeder fortschreitenden Schaltfolge $m_0\, t_1\, m_1\, t_2\, m_2 \ldots$ von N eine Permutation $t'_1\, t'_2\, t'_3 \ldots$ von $t_1\, t_2\, t_3 \ldots$ existiert, so dass $m_0\, t'_1\, m'_1\, t'_2\, m'_2 \ldots$ A-verträgliche Schaltfolge zu $m_0\, t_1\, m_1\, t_2\, m_2 \ldots$ ist und $\{m_0(s), m_1(s), m_2(s), \ldots\}$ endlich ist. N heißt schwach beschränkt, wenn alle seine Stellen schwach beschränkt sind.*

Falls eine Stelle schwach k-beschränkt ist, so ist sie auch schwach beschränkt. Umgekehrt gilt ohne weitere Annahmen, dass für jede schwach beschränkte Stelle ein k existiert, so dass sie auch k-beschränkt ist.

3 Gekoppelte Zustandsmaschinen

Wir betrachten im folgenden Petrinetze aus folgenden Bausteinen:

– Endlich viele disjunkte stark zusammenhängende Zustandsmaschinen. Das sind stark zusammenhängende Netze, deren Transitionen jeweils eine eingehende und eine ausgehende Kante haben und die insgesamt eine Marke tragen. Diese Zustandsmaschinen repräsentieren sequentielle Programme.
– Endlich viele Pufferstellen. Pufferstellen sind zusätzliche Stellen, die potenziell unbeschränkt sind. Der Vorbereich einer Pufferstelle besteht aus beliebigen Transitionen. Der Nachbereich besteht aus einer Transition einer anderen Zustandsmaschine, mit folgender Einschränkung: Falls in dieser Zustandsmaschine zwei Transitionen im Nachbereich einer (internen) Stelle liegen, so haben entweder beide Transitionen keine Pufferstelle in ihrem Vorbereich, oder beide Transitionen haben jeweils genau eine Pufferstelle in ihrem Vorbereich.
  Der erste Fall der Verzweigung entspricht einem *if-then-else* Konstrukt bzw. einem Schleifenausgang (datenabhängige Verzweigung). Der zweite Fall entspricht einem *select* Konstrukt (signalabhängige Verzweigung).
– Das so entstehende Petrinetz soll zusammenhängend sein, aber nicht notwendigerweise stark zusammenhängend.
– Die Menge A besteht aus den vorwärts verzweigenden Stellen der Zustandsmaschinen, deren Nachbereichstransitionen keine Pufferstellen im Vorbereich besitzen.

Petrinetze dieser Art nennen wir im folgenden gekoppelte Zustandsmaschinen.

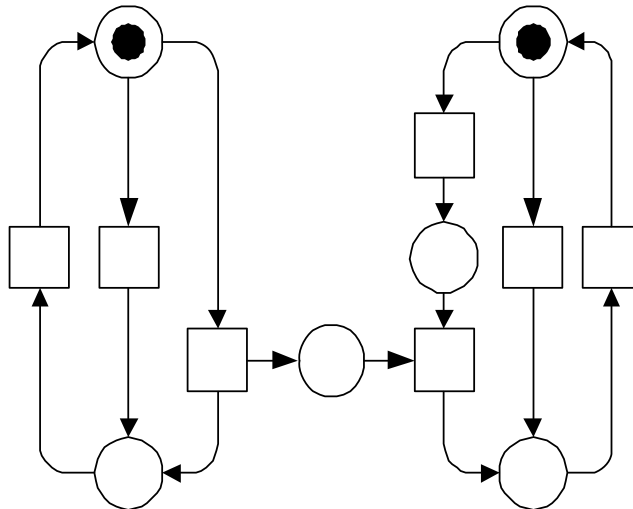


Abbildung 3. Ohne Fairnessannahme ist dieses Netz nicht schwach beschränkt.

Das Petrinetz aus Abbildung 3 ist nicht schwach beschränkt. Wenn Erzeuger und Verbraucher stets den rechten Zweig wählen, so wird der Verbraucher niemals die erzeugten Marken konsumieren, die Markenzahl auf der Pufferstelle wächst beliebig. Unter der Annahme fairen Verhaltens wählt der Verbraucher aber auch unendlich häufig den linken Zweig, und der Erzeuger unendlich oft den rechten Zweig. Durch entsprechende Vertauschungen der Schaltvorgänge lässt sich eine Schaltfolge konstruieren, bei der die Pufferstelle nie mehr als eine Marke trägt.

Definition 5. *Eine Schaltfolge heißt fair, wenn für jede Transition t gilt: Falls t unendlich häufig von einer in der Schaltfolge erreichten Markierung aktiviert wird, dann kommt t auch unendlich häufig in der Schaltfolge vor.*

Fairness ist stärker als die in Definition 1 formulierte Fortschrittsannahme.

Die folgenden Propositionen stellen eine effiziente Charakterisierung von schwacher Beschränktheit dar, wenn bereits bekannt ist, dass das betrachtete Netz eine gekoppelte Zustandsmaschine ist und als Petrinetz lebendig ist.

Proposition 1. *Sei N eine lebendige gekoppelte Zustandsmaschine. N ist, unter der Annahme von Fairness, schwach k-beschränkt für ein k gdw. der Rang der Inzidenzmatrix von N übereinstimmt mit*

$$\mid T \mid - \mid A^{\bullet} \mid + \mid A \mid -1.$$

Anstelle einer Beweisidee soll der vielleicht überraschende Zusammenhang zwischen dem Rang der Inzidenzmatrix und schwacher Beschränktheit verdeutlicht werden. Der Rang der Inzidenzmatrix entspricht der Anzahl Transitionen abzüglich der Anzahl linear unabhängiger Transitionsinvarianten. Wenn das Netz lebendig und schwach beschränkt ist, gibt es eine Schaltfolge, die von einer Markierung zu derselben Markierung führt. Ihr Schaltvektor ist eine Transitionsinvariante. Da die Alternativen der Zustandsmaschinen beliebig entschieden werden, und für jeden so entstehenden Ablauf eine beschränkte Permutation existiert, nimmt die Anzahl der Transitionsinvarianten mit jeder Alternative zu. Dies gilt aber nicht für verzweigende Stellen, deren Ausgangstransitionen auch Marken von Puffern konsumieren, denn diese Alternativen werden im Endeffekt durch die Marken auf den Puffern gesteuert.

Die Aussage von Proposition 1 gilt nicht mehr, wenn die Pufferstellen auch vorwärts verzweigen können. Im Petrinetz aus Abbildung 4 können Erzeuger und auch Verbraucher zunächst den rechten Zweig wählen. Der Verbraucher muss nun so lange warten, bis der Erzeuger einmal den linken Zweig wählt. Dies geschieht wegen Fairness schliesslich auch, aber bei jeder der endlich vielen Runden bis dahin wird eine Marke im oberen Puffer erzeugt, die zunächst nicht konsumiert werden kann. Das Netz ist daher nicht schwach k-beschränkt. Es ist aber schwach beschränkt, denn in keinem Ablauf steigt die Markenzahl auf den Pufferstellen zwingend unendlich.

Proposition 2. *Sei N eine lebendige gekoppelte Zustandsmaschine mit verzweigenden Pufferstellen. N ist, unter der Annahme von Fairness, schwach beschränkt gdw. der Rang der Inzidenzmatrix von N übereinstimmt mit*

$$\mid T \mid - \mid A^{\bullet} \mid + \mid A \mid -1.$$

4 Schluss

In diesem kurzen Beitrag konnten die Ideen zu schwacher Beschränktheit nur skizziert werden. Insbesondere fehlen die Beweise zu den Propositionen, die auch anderswo noch nicht zu finden sind. Insofern sollten diese Propositionen so lange als Vermutungen gelten, bis der Beweis publiziert ist.

Die Ideen entstanden bei einem Gastaufenthalt bei Cadence in Berkeley im August 2004. Alex Kontratyev hat mich auf die Fragestellung aufmerksam gemacht. Mit ihm habe ich auch viele Beispiele diskutiert. Der in [1] beschriebene Anwendungskontext führt zu den gekoppelten Zustandsmaschinen. Allerdings gibt es dort u.a. mit Eingabetransitionen (für eingehende Signale) weitere Modellbestandteile und Annahmen.

Schwache Beschränktheit hat interessante Anwendungen und existiert als Konzept bereits bei verwandten Ansätzen wie Message Sequence Charts. Die meisten relevanten Fragen sind aber noch offen. Dazu gehört die Entscheidbarkeit schwacher Beschränktheit und die Charakterisierung mit Hilfe von Halbordnungssemantik. Letztere liegt deswegen nahe, weil bei halbgeordneten Abläufen eine Reihenfolge nebenläufiger Transitionen nicht festgelegt wird, also auch nicht getauscht werden muss.

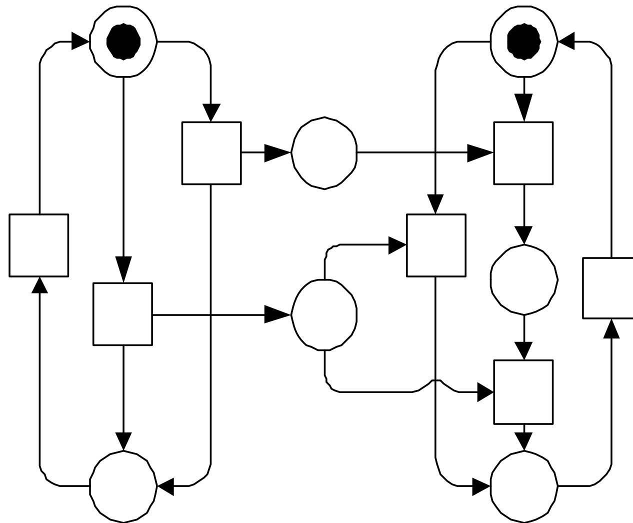


Abbildung 4. Auch unter Annahme von Fairness ist dieses Netz nicht schwach k-beschränkt

Literatur

1. Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Claudio Passerone und Yosinori Watanabe: Quasi-static schuduling of independent tasks for reactive systems. Javier Espaza und Charles Lakos (Hrsg.): Application and Theory of Petri Nets 2002. LNCS 2360, Springer (2002) 80–100
2. Jörg Desel und Javier Esparza: Free Choice Petri Nets. Cambridge University Press (1995)
3. Blaise Genest, Anca Muscholl und Doron Peled: Message Sequence Charts. Jörg Desel, Wolfgang Reisig und Grzegorz Rozenberg (Hrsg.): Lectures on Concurrency and Petri Nets. LNCS 3098, Springer (2004) 537–558
4. Markus Lohrey und Anca Muscholl: Bounded MSC communication. Information and Computation 198 (2004) 160–181

Reduction Rules for Interaction Graphs

Daniela Weinberg and Karsten Schmidt

Humboldt–Universität zu Berlin
Institut für Informatik
D–10099 Berlin
{weinberg,kschmidt}@informatik.hu-berlin.de

Abstract. The internet today has grown to be more than just being a basis for exchanging information. It steadily becomes a platform for processing business processes. Many companies distribute their service with the help of web services or integrate other web services into their own workflow.
However, before a web service gets published it should be examined well. We will introduce a way of examining the *controllability* of a web service. We propose the *interaction graph* of a web service, that is modelled by an open workflow net. To verify whether such a net is controllable or not it is sufficient to construct a reduced interaction graph. We will define reduction rules that minimize the size of the graph greatly. The analysis using the interaction graph as well as the reduction rules are implemented and have been integrated into an analysis tool kit for web services.

1 Introduction

In these days enterprises tend to source out functionalities and cooperation across borders has become increasingly important. For specific tasks so-called virtual enterprises are being formed. In this setting, *services* play an important role. Such a service basically encapsulates self-contained functions that interact through a well-defined interface. We assume the essentials of a service to include an *identifier* (id), its *interface*, and its *operational behavior*. With the help of the interface the service can communicate with its environment during the execution. The operational behavior of a service is a set of operations to be executed according to some internal control structure. The well-known class of *web services* is an implementation of services with an interface specified in WSDL [AIM01] and an id given by an URI. Throughout this paper we focus on services with operational behavior that is described as a *workflow*. We call such a service *workflow service*. With the raise of the language BPEL [BIMS02] the class of workflow services has become more and more important. BPEL provides a means to describe workflow behavior using certain control structures. It is a notation for web services whose control structure is modelled as a workflow.

A common example of a workflow service is a travel agency, that usually combines several web services. Surely a travel agency might also be a web service. A Java program, for instance, is definitely no workflow service, but it might implement a web service.

Before deploying a workflow service it is of great importance to analyse it thoroughly. A workflow service provides certain functionality. Therefore it is advisable to analyse whether this functionality can be used by another service. That means whether it is *controllable* or not. In this paper we consider workflow services as nets having an interface – the *open workflow nets*.

For those nets a technique for analysing the controllability has been introduced in [Mar03a] – the *communication graph*. The edges of those graphs represent communication steps that stand for consuming and producing messages by the net. The

nodes of the graph are divided into visible nodes indicating the start and the end of each step and not visible nodes depicting the state of the net between an input and an output of the controller. The graph therefore is compressed into communication steps. The calculation of each communication step has some implicitly defined reduction rules. In order to analyse whether the net is controllable the complete graph has to be calculated.

In this paper we will introduce a different technique for examining the controllability of open workflow nets. That is, we will define the *interaction graph* for an open workflow net which let's us decide whether the net is controllable or not. The complete interaction graph, however, shows all the communication that is possible between the net and its controller. It represents all reachable states of the net being analysed while communicating with its controller. Therefore the complete graph is huge in size (comparable to the reachability graph of Petri nets). However, this way it is possible to apply specific reduction rules while building up a reduced graph. As our case studies show, the resulting graph is extremely smaller then the complete graph. With the help of our case studies we could even show that the reduced interaction graph usually is smaller than the communication graph for the corresponding net.

The rest of this paper is divided as follows. Section 2 provides the basics of open workflow nets. The next section 3 briefly introduces the interaction graphs and shows the idea that is behind the control strategy, which is a subgraph of the interaction graph. Section 4 describes two reduction rules. Section 5 shows the results of our case studies. The last section 6 gives a summary of our results and speaks about our future research. For a more deeper insight into our work the interested reader should refer to [Wei05].

2 Open Workflow Nets

Our work is based on *open workflow nets* (oWFNs) that were introduced in [MRS05]. An oWFN is essentially a liberal version of van der Aalst workflow nets [vdA98], enriched with communication places. Each communication place of an oWFN models a channel to send (receive) messages to (from) another oWFN. This way we abstract from data and model the occurance of messages just as undistinguishable tokens.

We assume the usual representation and definition of Petri nets $N = (P, T, F)$. An open workflow net is a Petri net $N = (P, T, F)$ together with (1) two sets $in, out \subseteq P$, such that for all transitions $t \in T$ holds: if $p \in in$ ($p \in out$) then $(t, p) \notin F$ ($(p, t) \notin F$), (2) a distinguished marking m_0, called the *initial marking*, and (3) a set Ω of distinguished markings, called the *final markings* of N. The places in in (out) are called *input* (*output*) places. The set $in \cup out$ is called the *interface* of N. The *inner* of N can be obtained from N by removing all interface places, together with their adjacent arcs. We label a transition t connected to an input (output) place x with $?x$ and name it *receiving transition* ($!x$, *sending transition*). A transition that is not connected to an interface place is called τ transition.

A marking m of an oWFN is a *deadlock* if m enables no transition at all. An oWFN in which all deadlocks are final markings is called *weakly terminating*. Given an oWFN N, we call an oWFN M a *strategy* for N iff the oWFN $N \oplus M$ (composition of N and M) is weakly terminating. N and M then are *partners*. So, the net N is controllable, if there exists an oWFN M, such that the composed oWFN $N \oplus M$ weakly terminates.

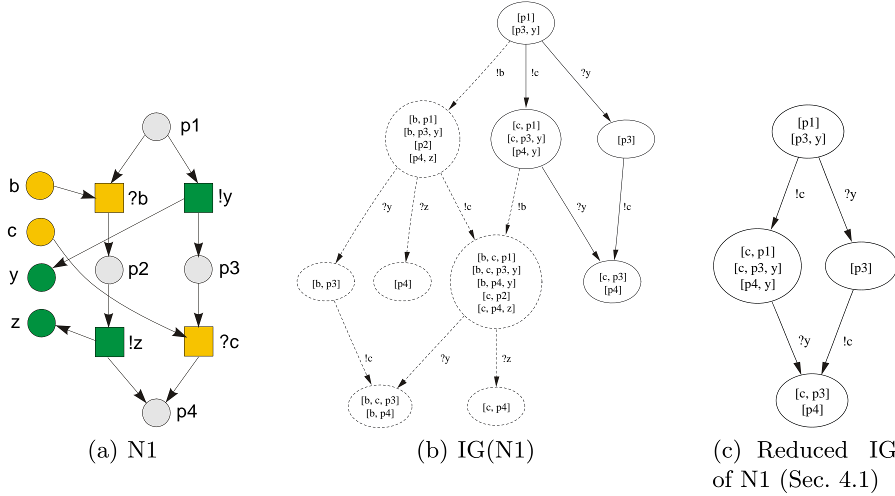
Throughout this paper we refer to M as a *controller* of N and we call a marking a *state* of the net. Further we only consider acyclic open workflow nets and we just permit those final markings that have empty interface places.

3 Interaction Graphs

The *interaction graph* (IG) of an oWFN has been developed with the reachability graphs of Petri nets [Sta90] in mind. In contrast to those graphs it represents the controller's point of view. The nodes of the graph are a set of states, which the net can reach by consuming and producing messages. The edges of the graph represent the actions of the controller – sending and receiving messages. Basically, the nodes of the graph are a *hypothesis* of the controller with respect to the state of the net. The controller only knows in which set of states the net is in. It does, however, not know the exact state of the net.

There is communication between the controller and the net. The controller can control the net in a limited way by sending messages. Whereas by receiving messages from the net, the controller gets some knowledge about the state the net might be in. We distinguish two kind of events: (1) *sending event* means that the controller sends a message to the net (labelled by !) and (2) *receiving event* represents the receiving of a message (labelled by ?) by the controller.

Example 1. Figure 1 shows the oWFN N1 (Fig. 1(a)), its complete interaction graph IG(N1) (Fig. 1(b)) and a reduced IG (see Sec. 4.1). N1 possesses the interface places



(a) N1 (b) IG(N1) (c) Reduced IG of N1 (Sec. 4.1)

Fig. 1. oWFN N1, its complete IG and its reduced IG

$\{b, c\} = in(N1)$ and $\{y, z\} = out(N1)$. The initial marking of N1 is $[p1]$ and the final marking is $[p4]$.

The root node of the graph is the state set $\{[p1], [p3, y]\}$. At first N1 is in the transient state $[p1]$. This state enables the net to switch to the maximal state $[p3, y]$ on its own since transition $!y$ is enabled. There is no other state reachable for the net at this time without the help of the controller. The state $[p1]$ activates the sending event $!b$. If there would be a b on the respective interface place, the net could switch to another state. The state $[p3, y]$ activates the sending event $!c$ and the receiving event $?y$. Since there is a token on the interface place y, the controller can consume that token – it receives the message y. Therefore, the successor node in case of the receiving event $?y$ is $[p3]$. This node activates the sending event $!c$, which results in the successor node $\{[c, p3], [p4]\}$. The controller puts a token on place c. However, the net does not necessarily have to consume that token right away. So, this results in the transient state $[c, p3]$. This state then can change to state $[p4]$

since the receiving transition $?c$ now is activated. The IG has three different end nodes (nodes, that do not activate any kind of event). The end node $\{[c, p3], [p4]\}$ is a good end node. Its state set consists of two states – $[c, p3]$ is transient leading to state $[p4]$ and $[p4]$ is maximal and the final marking of the net. The end nodes $\{[b, c, p3], [b, p4]\}$ and $\{[c, p4]\}$ are bad end nodes. Both nodes do not activate any events. Further, there is no maximal state that is a final marking of the net. There is a node $\{[p4]\}$. $[p4]$ is a final marking of the net. Hence, we could assume this node to be a good end node. However, during the analysis of controllability we declare this node to be a bad end node. This node has only one incoming edge ?z. This edge comes from node $\{[b, p1], [b, p3, y], [p2], [p4, z]\}$. The maximal state $[b, p3, y]$ activates two events – ?y and !c. Since both events lead to a bad end node, the node is declared to be a bad node. Therefore the node $\{[p4]\}$ is a bad node as well.

The control strategy[1] is a subgraph of the IG. Its leafs are good end nodes in the IG and the root node is just the root node of the IG. For every maximal state of each node within the control strategy we can find an event which leads to a node of the control strategy. The oWFN N1 is controllable. By analysing its IG we can find a control strategy. The controller has two options: sending a c and then receiving a y or first receiving y and then sending a c. Both ways lead the oWFN N1 to terminate weakly. ⋆

4 Reduction of Interaction Graphs

In this section we will introduce two reduction rules that will reduce the number of events being considered for building up the IG. We will only present the idea of the rule with the help of an example. Please refer to [Wei05] for the complete and formal description of the reduction rule. Furthermore, in [Wei05] we proof, that each reduced IG can be used for the analysis of controllability of an oWFN.

By combining all reduction rules presented in [Wei05] we get a a more compact graph, which we then can use for the analysis of controllability.

4.1 Transient States

It is possible that there are transient states within a node of the IG. Being in a transient state the net can change to another state without letting its controller know. So, just the maximal states supply a surety in a certain way. Suppose a transient state activates a sending event. If the controller now sends a message to the net, there is no way of knowing that the net will ever consume this message. The net might just have switched to another state, which does not activate this event anymore. So, we will only consider the maximal states of a node to compute the activated events.

Example 2. Let us take a look at Fig. 1 again. It shows the oWFN N1 (Fig. 1(a)) and its complete interaction graph IG(N1) (Fig. 1(b)).

There are two different states in the root node of the IG – [p1] (transient) and [p3, y] (maximal). During the calculation of the complete IG (Fig. 1(b)) the sending event !b was considered. This sending event is only activated in the transient state [p1]. The maximal state [p3, y] activates the sending event !c and the receiving event ?y. For the calculation of the reduced IG (see Fig. 1(c)) we now use the sending event !c and the receiving event ?y only. ⋆

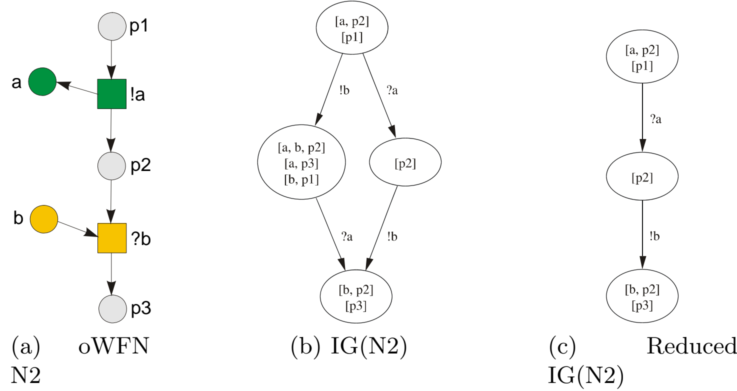
This rule will work best if the oWFN being analysed consists of sending as well as of τ transitions. That means, the nodes of the IG contain transient states.

[1] In this paper the control strategy of an interaction graph is depicted as a solid line. Those nodes and edges that do not belong to the control strategy of the graph are drawn with dashed lines.

4.2 Both kind of Events activated in one State

We turn our attention to those maximal states, that activate receiving events. If such a state activates a sending event as well, we let the receiving event occur first. The sending event will still be activated afterwards. Note, sending events with the same name can be activated by different receiving transitions. Thus, for the calculation of successor nodes we only use those sending events, that are activated by states that do not activate receiving events.

Example 3. In Fig. 2 we can see the oWFN N2 with its complete interaction graph (Fig. 2(b)) and its reduced IG (Fig. 2(c)). The root node of the two graphs contains



(a) oWFN N2 (b) IG(N2) (c) Reduced IG(N2)

Fig. 2. Receiving before sending.

the maximal state [a, p2]. This state activates the receiving event ?a as well as the sending event !b. The controller now shall receive the message a before sending b. The sending event !b is still activated in the successor node. ⋆

If the oWFN being analysed contains sequences of sending and receiving transitions (in that order) this rule will work out well. We are then able to find states that activate both kind of events. Therefore, by letting the controller receive the message first, we make sure that the successor node will not increase in size (see [Wei05] – complexity of interaction graphs) and we only calculate one successor node even though both events are activated.

5 Case Studies

Our reduction rules have been implemented in Java. They were integrated into the tool *Workflow Modelling and Business Analysis Toolkit for Web Services* (Wombat4ws, [Mar03b,Mar03a]). Wombat4ws is a prototypical application of a tool kit for the analysis of web services.

With the help of the implementation of the algorithms we could test the reduction rules in practice. Table 1 shows some of the results we obtained. As input models we took processes from the BPEL specification [BIMS02] and from [Mar03a]. As the case studies presented in table 1 show, the reduced IG, that is obtained by combining all reduction rules ([Wei05]), usually is smaller than the communication graph for the corresponding net. Besides that, we have tested our techniques using several other case studies. All showing the same result being compared to the corresponding communication graph. The reduced IG is significantly smaller than the communication graph for most oWFNs.

	oWFN		compl. IG		red. IG		reduction [%]		CG	
Process Name	#P	#T	#V	#E	#V	#E	#V	#E	#V	#E
COP	33	20	83	192	9	9	89,2	95,3	31	40
PO	30	18	183	620	7	7	96,2	98,9	34	68
eCommerce I	29	16	95	230	13	15	86,3	93,5	25	30
eCommerce II	29	16	83	192	9	9	89,2	95,3	31	40
auction service	23	14	13	19	7	7	46,2	63,2	5	4

Table 1. Sizes of the communcation graph (CG), and the complete as well as the reduced interaction graph (IG) by their number of nodes (#V) and edges (#E). Furthermore, the number of places (#P) and the number of transitions (#T) of the models is shown. COP stands for complex order process. The purchase order process (PO) and the auction service were taken from the BPEL specification.

6 Conclusions and Future Work

In this paper we introduced the interaction graphs to analyse the controllability of an oWFN. By analysing the graph we have focused on *one* property of oWFNs – controllability. We have described how this property can be verified using interaction graphs. In order to make the analysis more efficient, we have defined reduction rules for the interaction graphs. Due to the lack of space in this paper we have just briefly introduced two reduction rules. [Wei05] formalizes three more reduction rules and shows a proof that the reduced IG can still be used for the controllability analysis. Our case studies show that the theoretical assumptions we have made while developing the reduction rules indeed work very well in practice.

We currently adapt our results to a more liberal version of oWFNs. That is we want to permit final markings that do not necessarily leave the interface places empty. We also want to modify our techniques to fit to cyclic oWFNs as well. Furthermore, we work on finding more reduction rules in order to have $IG(N) \leq CG(N), \forall$ oWFN N measured by the number of nodes and edges of the graphs. One of our goals hereby is to adapt the *partial order reduction* [Val88] to fit our needs.

References

[AIM01] Ariba, IBM, and Microsoft. *WSDL – Web Services Description Language.* W3C, Standard, version 1.1 edition, March 2001. `http://www.w3.org/TR/wsdl`.

[BIMS02] BEA, IBM, Microsoft, and SAP. *BPEL4WS– Business Process Execution Language for Web Services.* Vorschlag zur Standardisierung, Version 1.1, July 2002. `http://ibm.com/developerworks/webservices/library/ws-bpel/`.

[Mar03a] Axel Martens. *Verteilte Geschäftsprozesse – Modellierung und Verifikation mit Hilfe von Web Services.* WiKu-Verlag für Wissenschaft und Kultur KG, 2003. Dissertation.

[Mar03b] Axel Martens. Wombat4ws – *Workflow Modelling and Business Analysis Toolkit for Web Services.* Institut für Informatik: Humboldt-Universität zu Berlin, 2003. `http://www.informatik.hu-berlin.de/top/wombat`.

[MRS05] Peter Massuthe, Wolfgang Reisig, and Karsten Schmidt. An operating guideline approach to the soa. Informatik-Berichte 191, Humboldt-Universität zu Berlin, 2005. Submitted to a conference.

[Sta90] Peter H. Starke. *Analyse von Petri-Netz-Modellen.* B.G. Teubner Verlag, Stuttgart, Germany, leitfäden und monographien der informatik edition, 1990.

[Val88] Antti Valmari. *State Space Generation – Efficiency and Practicality.* Dissertation, Tampere University of Technology, 1988.

[vdA98] W. M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[Wei05] Daniela Weinberg. Reduction rules for interaction graphs. Informatik-Berichte xx, Humboldt-Universität zu Berlin, 2005. to appear.

Semantische Annotation von Petri-Netzen

Agnes Koschmider und Daniel Ried

Institut für Angewandte Informatik und Formale Beschreibungsverfahren
Universität Karlsruhe (TH), 76187 Karlsruhe, Germany
{*koschmider, ried*}*@aifb.uni-karlsruhe.de*

1 Einführung

Durch unternehmensübergreifende Kooperationen können Abläufe rationalisiert, Synergien zwischen Geschäftspartnern gebildet und Unternehmensrisiken gemindert werden. Dies erfordert flexibel anpassbarer Geschäftsprozesse, die schnell mit Prozessen von Unternehmenspartnern zusammengeführt werden können. Voraussetzung für eine effiziente Prozesszusammenführung sind maschinenlesbare und maschineninterpretierbare Prozesskomponenten. Bei der Modellierung von Geschäftsprozessen in unterschiedlichen Unternehmen kann ein einheitliches Vokabular nicht vorausgesetzt werden, weshalb Prozessdaten von Systemen unterschiedlich verstanden und interpretiert werden. Durch die Verwendung von Ontologien kann ein einheitliches Verständnis über Begriffe und Beziehungen von Geschäftsprozessen definiert und die Wiederverwendung von Geschäftsprozess-Komponenten und Interoperabilität in ungleichen Systemlandschaften unterstützt werden. Zur Modellierung, Simulation und Validierung von Geschäftsprozessen eignen sich Petri-Netze, die sowohl eine formale als auch eine graphische Darstellung von Petri-Netz-Elementen ermöglichen.

In diesem Beitrag werden wir eine Ontologie zur semantischen Annotation von Petri-Netzen vorstellen. Die Ontologie wird mit der Web Ontology Language (OWL) [W3C04a] definiert, die seit Anfang 2004 als Empfehlung vom W3C veröffentlicht wurde. OWL gibt es in den Ausprägungen OWL Lite, OWL DL und OWL Full, wobei gilt: *OWL Lite* $\subset$ *OWL DL* $\subset$ *OWL Full* [W3C04b]. OWL DL (Description Logic) ist eine syntaktische Variante der $\mathcal{SHOIN}(D)$ Beschreibungslogik und ist trotz ihres hohen Grads an Ausdrucksmächtigkeit entscheidbar [Hor05]. $\mathcal{SHOIN}(D)$ bietet Verneinung, Disjunktion und eine eingeschränkte Form von All- und Existenzquantoren für Variablen. In Kombination mit bestimmten Aussagen erlauben Quantoren das automatische Schlussfolgern in Bezug auf andere Aussagen. Durch die Verwendung von Schlussfolgerungsmechanismen in semantisch annotierten Geschäftsprozessen könnte beispielsweise die Prozesszusammenführung von unternehmensübergreifenden Kooperationen durch Software implementierte Komponenten unterstützt werden. Ziel unserer Forschungsarbeit sind Flexibilisierung, leichtere Integration und automatische Zusammenführung von Geschäftsprozessen, auch wenn Geschäftspartner ein unterschiedliches Vokabular verwenden.

Unser Beitrag ist wie folgt gegliedert: zunächst werden wir eine Ontologie für elementare Petri-Netze definieren und diese Ontologie in Abschnitt 3 um ontologische Konzepte für Prädikate/Transitionen-Netze erweitern. Abschnitt 4 stellt SemPeT vor, ein Werkzeug zur Modellierung von Prädikate/Transitionen-Netzen und semantischen Annotation der Modelle durch OWL. Unser Beitrag endet mit einer Zusammenfassung und einem Ausblick auf zukünftige Forschungsarbeiten.

2 Ontologie für Petri-Netze

Ein Petri-Netz [RR98] ist ein gerichteter bipartiter Graph mit Knoten und Kanten und wird durch das Tripel $N = (S, T, F)$ beschrieben. Knoten sind Stellen (S-Elemente) und Transitionen (T-Elemente), die durch Kanten (F-Elemente) verbunden sind, wobei folgenden Eigenschaften gelten: (i) $S \cap T = \emptyset$ und (ii) $F \subseteq (S \times T) \cup (T \times S)$. In Petri-Netzen unterscheiden wir zwei Arten von Kanten: $F_r \subseteq (S \times T)$

und $F_w \subseteq (T \times S)$. Zur Abbildung von Petri-Netz-Elementen auf ontologische Konzepte, verwenden wir die OWL-Grundelemente Klassen (Class), Objekteigenschaften (ObjectProperty), Instanzen (Individuum) und Datentypeigenschaften (DatatypeProperty)[1]. Petri-Netz-Elemente definieren wir in OWL wie folgt:

- **PetriNet** für alle möglichen Petri-Netze $N = (S, T, F)$
- **Place** für alle Stellen aus S,
- **Transition** für alle Transitionen aus T,
- **FromPlace** für alle Kanten aus F_r,
- **ToPlace** für alle Kanten aus F_w.

Um auszudrücken, dass Petri-Netze Kanten und Knoten haben, ordnen wir der Klasse **PetriNet** die Objekteigenschaften

- **hasNode** für die Domäne **PetriNet** und den Wertebereich **Place** und **Transtion**,
- **hasArc** für die Domäne **PetriNet** und den Wertebereich **FromPlace** und **ToPlace**

zu. Abhängig vom Petri-Netz-Typ beinhalten Stellen Marken. Diese Eigenschaft drücken wir durch die OWL-Objekteigenschaft

- **hasMarking** für die Domäne **Place**

aus. Petri-Netze folgen einer operationalen Semantik, die die Zusammenführung und Verifikation von Geschäftsprozessen ermöglicht. Um dies in OWL auszudrücken, verwenden wir die folgenden Objekteigenschaften:

- **placeRef** für die Domäne **Transition** und den Wertebereich **Place**,
- **transRef** für die Domäne **Place** und den Wertebereich **Transition**.

Im Folgenden werden wir die Petri-Netz-Ontologie um Elemente für strikte Prädikate/Transitionen-Netze erweitern.

3 Ontologie für Pr/T-Netze

Prädikate/Transitionen-Netze (Pr/T-Netze) sind eine Variante von höheren Petri-Netzen [GL81]. Im Gegensatz zu elementaren Petri-Netzen (S/T-Netzen), in denen Marken anonyme Objekte darstellen, repräsentieren Marken in höheren Petri-Netzen Objekte mit individuellen Eigenschaften. Ein striktes Pr/T-Netz ist ein Tupel $PrT = (S, T, F, \Psi, KB, TI, M^0)$, das die folgenden Bedingungen erfüllt (i) (S, T, F) ist ein Netz und S wird als Menge von Prädikaten mit veränderlichen Ausprägungen und T als Menge von Transitionsschemata interpretiert. (ii) $\Psi = (D, FT, PR)$ ist eine Struktur bestehend aus einer Individuenmenge D, einer auf D definierten Menge von Funktionen FT und einer Menge PR von auf D definierten Prädikaten mit veränderlichen Ausprägungen. (iii) Die Kantenbeschriftung KB weist jeder Kante aus F eine Menge von Variablentupeln mit der Stelligkeit des adjazenten Prädikats zu. (iv) TI weist jeder Transition aus T eine Transitionsinschrift in Form eines über Ψ und der Menge der an den adjazenten Kanten vorkommenden Variablen gebildeten prädikatenlogischen Ausdrucks zu. (v) M^0 ist eine Markierung der Prädikate mit Mengen von konstanten Individuentupeln, deren Stelligkeit der Stelligkeit des Prädikats entspricht.

Um ein Pr/T-Netz in OWL auszudrücken erweitern wir die oben definierten OWL-Klassen und Objekteigenschaften um:

[1] In der OWL-Spezifikation werden noch Eigenschaftsmerkmale, Eigenschaftsrestriktionen oder komplexe Klassen definiert.

- **LogicalConcept** für alle Transitionsinschriften aus TI,
- **IndividualDataItem** für alle Prädikate PR,
- **Attribute** für alle Funktionen aus FT,
- **Value** für alle Teilmengen aus $D_1 \times \ldots \times D_n$,
- **Delete, Insert** für die Kantenbeschriftung mit Mengen von Variablentupeln.

Die Annotation einer Transition eines Pr/T-Netzes setzt sich aus Schaltbedingungen und Schaltoperationen zusammen, die in einer prädikatenlogischen Formel vereinigt werden. Bedingungen und Operationen drücken wir wie folgt aus:

- **Condition** für die Schaltbedingungen,
- **Operation** für die Schaltoperationen.

Den Klassen weisen wir Objekteigenschaften wie folgt zu:

- **hasLogicalConcept** mit der Domäne **Transition** und dem Wertebereich **LogicalConcept**
- **hasOperation** mit der Domäne **LogicalConcept** und dem Wertebereich **Operation**,
- **hasCondition** mit der Domäne **LogicalConcept** und dem Wertebereich **Condition**,
- **hasInscription** mit der Domäne **FromPlace**, **ToPlace** und dem Wertebereich **Delete**, **Insert**.

Für die Klassen LogicalConcept, IndividualDataItem, Delete und Insert haben wir die Objekteigenschaften:

- **hasAttribute** mit Wertebereich Attribute,

und für Attribute die Objekteigenschaft

- **hasValue** mit dem Wertebereich **Value**

identifiziert. Für die Klasse Value gibt es die folgende Objekteigenschaft:

- **hasRef** mit der Domäne und dem Wertebereich **Value**.

Um eine prädikatenlogische Formel in der Pr/T-Ontologie als Schaltbedingung einer Transition darstellen zu können, muss diese in Pränex-Normalform vorliegen. Besitzt die Formel desweiteren eine Matrix in KNF, so kann diese durch die folgenden drei charakteristischen Datentypeigenschaften dargestellt werden:

- **forall**, **exists** und **and** mit der Domäne **Condition** und dem Wertebereich **xsd:string**

Eine Schaltoperation einer Transition wird durch die Datentypeigenschaft

- **function** mit der Domäne **Operation** und dem Wertebereich **xsd:string** wiedergegeben.

Die definierte Pr/T-Ontologie kann formal mit einer Beschreibungslogik abgebildet werden. In der oben definierten Ontologie haben wir nur Klassen, Objekt- und Datentypeigenschaften definiert. Um allerdings ein Pr/T-Netz vollständig in OWL auszudrücken, benützen wir noch die Konstrukte *owl:disjointWith* um Disjunktion zwischen **Transition** und **Place** auszudrücken und Quantoren um beispielsweise zu beschreiben, dass für alle Kanten **FromPlace** der Knoten **Place** angegeben werden muss und für **ToPlace** der Knoten **Transition**[2]. OWL DL ist eine syntaktische Variante der $\mathcal{SHOIN}(D)$[3] Beschreibungslogik und ist trotz ihres hohen Grads an Ausdrucksmächtigkeit entscheidbar. In Schlussfolgerungssystemen wie Racer und FaCT wird die Beschreibungslogik $\mathcal{SHIQ}(D)$ [HST99] verwendet und unterscheidet sich von $\mathcal{SHOIN}(D)$ vor allem dadurch, dass keine Nominale unterstützt werden[4]. Um Entscheidbarkeit in $\mathcal{SHIQ}(D)$ zu erreichen, wird OWL DL um Regeln, die DL-sicher sind, erweitert [MSS04]. Abfragen für diese Erweiterung von $\mathcal{SHIQ}(D)$ mit DL-sicheren Regeln werden durch einen Algorithmus erreicht, der $\mathcal{SHIQ}(D)$ zu disjunktiven Datalog-Programmen reduziert [HMS04].

In Abbildung 1 wird die Ontolgie für Pr/T-Netze in einem Klassendiagramm gezeigt:

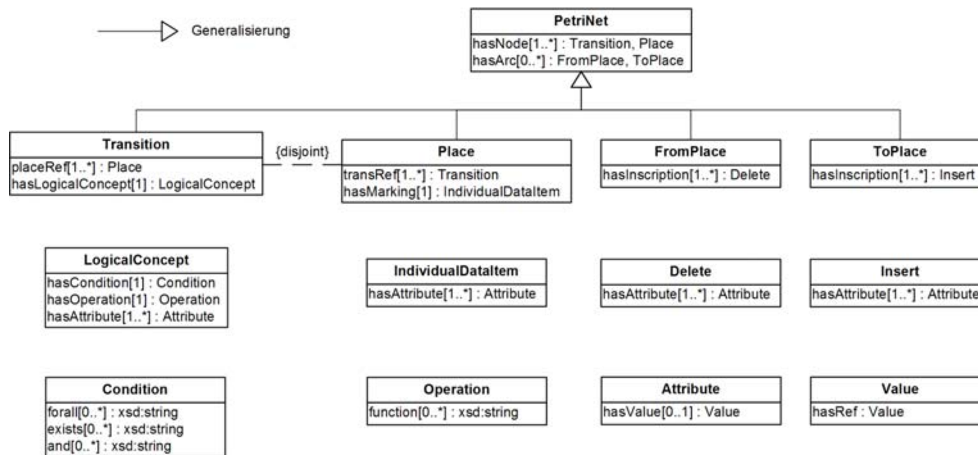


Figure 1: Klassendiagramm der Ontologie für Pr/T-Netze

4 Implementierung der Pr/T-Netz-Ontologie

Abbildung 1 zeigt das in den Abschnitten 2 und 3 definierte Vokabular der Ontologie. Eine Ontologieklasse ist darin als Klasse dargestellt. Eine Objekt- bzw. Datentypeigenschaft der Ontologie ist als Attribut der Klasse dargestellt, die als Domäne der Eigenschaft definiert ist. Der Datentyp und die Multiplizität eines Attributs sind als Wertebereich bzw. Kardinalität der Eigenschaft zu interpretieren. Liegt eine Ontologieklasse in der Domäne bzw. im Wertebereich einer Eigenschaft, so gilt dieses ebenfalls für alle Unterklassen. Die Implementierung des Vokabulars erfolgt anhand der von Jena [5] – einer Java API für RDF/OWL – zur Verfügung gestellten Klassen Class bzw. ObjectProperty und DatatypeProperty innerhalb einer sogenannten Vokabularklasse. Die taxonomischen Beziehungen der Ontologieklassen untereinander sowie diesbezügliche Einschränkungen, wie z.B. die Disjunktkeit von Klassen, werden anhand eines Ontologiemodells (OntModel) implementiert. Durch diese Vorgehensweise können Java-Anwendungen auf die Ontologie sowie deren Vokabularelemente zugreifen und Instanzmodelle erzeugt werden.

Das im Rahmen dieses Beitrags vorgestellte Werkzeug SemPeT unterstützt die grafische Modellierung und semantische Annotation von Pr/T-Netz-Instanzen durch die in Abbildung 2 dargestellte GUI.

Als grundlegende interne Datenstruktur einer modellierten Instanz wird ein von Jena bereitgestelltes OWL-Modell verwendet. Beim Hinzufügen von grafischen Elementen bzw. bei der Definition und Initialisierung

[2] Der Vorgänger und Nachfolgeknoten können durch die Objekteigenschaften **placeRef** und **transitionRef** ermittelt werden

[3] O steht für *nominals*, N für *unqualified number restrictions* und (D) für *datatypes*

[4] Nominale in der OWL DL-Syntax werden durch *owl:oneOf* oder *owl:hasValue* ausgedrückt.

[5] http://www.hpl.hp.com/semweb/jena.htm

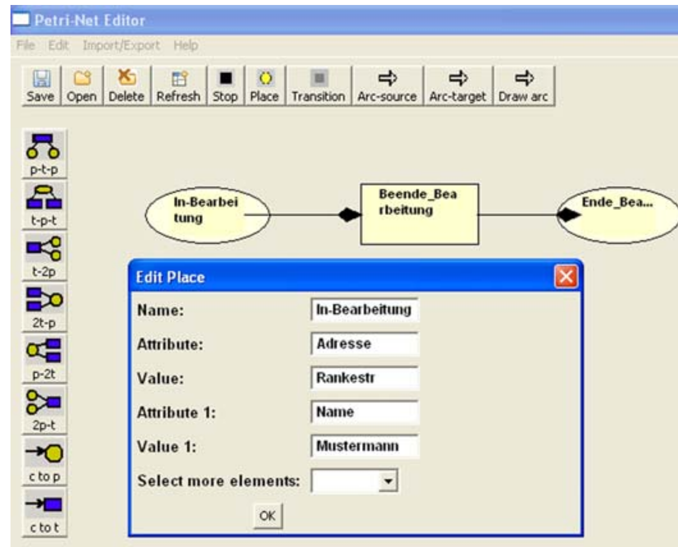


Figure 2: Grafische Modellierung von Pr/T-Netzen mit SemPeT

von Annotationen werden dem Ontologiemodell entsprechende, bzgl. der Ontologie gültige Aussagen aus dem Vokabular hinzugefügt. Eine Aussage ist ein Tripel (Subjekt, Prädikat, Objekt) mit einer Eigenschaft als Prädikat und einer Ontologieklasse aus der Domäne der Eigenschaft als Subjekt sowie mit einer Klasse bzw. einem literalen Wert (z.B. vom Typ xsd:string oder xsd:nonNegativeInteger) als Objekt. Dabei wird gewährleistet, dass eine Pr/T-Netz-Instanz korrekt auf ein Instanzmodell der vorgestellten Ontologie innerhalb des definierten Vokabulars abgebildet wird.

Die Ausgabe des Modells erfolgt serialisiert in OWL-Syntax und hat die folgende Form (Ausschnitt):

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns="http://www.aifb.uni-karlsruhe.de/apparat#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:petri="http://www.aifb.uni-karlsruhe.de/petri/PNOntologie.owl#"
  xml:base="http://www.aifb.uni-karlsruhe.de/apparat">
  <petri:PetriNet rdf:ID="Arbeitsbearbeitung">
    <petri:hasNode>
      <petri:Place rdf:ID="In-Bearbeitung">
        <petri:hasMarking>
          <petri:IndividualDataItem rdf:ID="R_In-Bearbeitung">
            <petri:hasAttribute rdf:resource="#Adresse"/>
            <petri:hasAttribute rdf:resource="#Name"/>
          </petri:IndividualDataItem>
        </petri:hasMarking>
        <petri:transRef>
          <petri:Transition rdf:ID="Beende_Bearbeitung">
            <petri:hasNode rdf:resource="#Beende_Bearbeitung"/>
          </petri:Transition>
        </petri:transRef>
      </petri:Place>
    </petri:hasNode>
  </petri:PetriNet>
</rdf:RDF>
```

Auf diese Weise wird die maschinelle Weiterverarbeitung, d.h. der unternehmensübergreifende Austausch von Prozessmodellen, die Interpretation und Validierung von Prozessmodellen anhand eines einheitlichen

Vokabulars sowie das Ableiten von Schlussfolgerungen aus den Modellen anhand eines Regelsatzes unterstützt.

5 Zusammenfassung und Ausblick

In unserem Beitrag haben wir eine semantische Annotation für Pr/T-Netze vorgestellt, womit Daten von Pr/T-Netzen in einem maschinenverständlichen und maschineninterpretierbaren Format abgebildet werden können. Derzeit implementieren wir auch ein Werkzeug, das die semantische Annotation von Pr/T-Netzen unterstützt. Aufbauend auf der Definition einer Ontologie für Pr/T-Netze untersuchen wir Techniken des Ontology Alignment, die Grundlage eines Mappings einer Pr/T-Ontologie auf eine andere Pr/T-Ontologie bilden. Durch die semantische Zusammenführung von mehreren Pr/T-Ontologien ergibt sich auch die Grundlage für weitere interessante Fragestellungen. Es wird beispielsweise zu untersuchen sein, in wieweit Ontologien um Zeitaspekte erweitert werden können, damit die zusammengeführten Pr/T-Ontologien ihre Ausgangssemantik behalten.

Literaturverzeichnis

[GL81] H. J. Genrich und K. Lautenbach. System modelling with high level Petri nets. *Theoretical Computer Science*, (13):109–136, 1981.

[HMS04] U. Hustadt, B. Motik und U. Sattler. Reducing $\mathcal{SHIQ}^-$ Description Logic to Disjunctive Datalog Programs. In *Proceedings of the 9th International Conference on Knowledge Representation and Reasoning*, Seiten 152–162. 2004.

[Hor05] I. Horrocks. Applications of description logics: State of the art and research challenges. In *Proceeding of the 13th International Conference on Conceptual Structures*, Seiten 87–90. Springer, 2005.

[HST99] I. Horrocks, U. Sattler und S. Tobies. Practical reasoning for expressive description logics. In *Proceeding of the 6th International Conference on Logic for Programming and Automated Reasoning*, Seiten 161–180. 1999.

[MSS04] B. Motik, U. Sattler und R. Studer. Query Answering for OWL-DL with Rules. In *Proceeding of the 3rd International Semantic Web Conference*. 2004.

[RR98] W. Reisig und G. Rozenberg. *Lectures on Petri Nets: Basic Models*, Band 1491 der *Lecture Notes in Computer Science*. Springer, 1. Auflage, 1998.

[W3C04a] World Wide Web Consortium. *OWL Web Ontology Language Guide*, 10. Februar 2004. Recommendation. http://www.w3.org/TR/2004/REC-owl-guide-20040210/.

[W3C04b] World Wide Web Consortium. *OWL Web Ontology Language Semantics and Abstract Syntax*, 10. Februar 2004. Recommendation. http://www.w3.org/TR/2004/REC-owl-semantics-20040210/.

Modellierung und Analyse transaktionaler Geschäftsprozesse

Carsten Frenkler Karsten Schmidt
{frenkler, kschmidt}@informatik.hu-berlin.de

Zusammenfassung

Wie erweitern in dieser Arbeit Workflow-Module um ein Konzept, mit dem der Einfluß eines Datenbanksystems auf die Bedienbarkeit eines Geschäftsprozesses untersucht werden kann. Wir integrieren transaktionale Eigenschaften als internes Verhalten in Workflow-Module und können damit Bedienbarkeit und Einhaltung transaktionaler Eigenschaften durch Analyse entscheiden.

1 Einführung

Bedienbarkeit ist eine wichtige Eigenschaft eines Geschäftsprozesses. Ein Geschäftsprozeß ist bedienbar, wenn eine Umgebung existiert, die den Geschäftsprozeß so steuern kann, daß der Prozeß immer einen wohldefinierten Endzustand erreicht. Um Bedienbarkeit zu entscheiden, wird der zu analysierende Geschäftprozeß als Workflow-Modul (WFM), einer speziellen Klasse von Petrinetzen, modelliert [3, 5]. Bedienbarkeit wird über die Analyse des internen Verhaltens des WFM, in Abhängigkeit von der Kommunikation mit einer Umgebung entschieden. Das WFM eines Geschäftsprozesses abstrahiert vollständig von Daten und deren Verarbeitung durch ein Datenbanksystem.

Wir erweitern WFM um ein Konzept, mit dem der Einfluß eines unterhalb des Geschäftsprozesses angesiedelten Datenbanksystems auf die Bedienbarkeit des Geschäftsprozesses untersucht werden kann. Unser Ansatz sieht vor, Datenbankzugriffe Aktivitäten des Geschäftsprozesses und damit Transitionen des zugehörigen WFM zuzuordnen. Der Einfluß einer Datenbank auf den Kontrollfluß wird unabhängig von einem konkreten Datenbanksystem modelliert, indem transaktionale Eigenschaften in ein Petrinetz übersetzt werden. Durch Komposition dieses Petrinetzes mit dem entsprechenden WFM werden transaktionale Eigenschaften als internes Verhalten in das WFM integriert. Die Komposition nennen wir transaktionales Workflow-Modul (taWFM). Dieser Ansatz hat den Vorteil, Bedienbarkeit und die Frage, ob ein Datenbanksystem die Transaktionen eines Geschäftsprozesses ordnungsgemäß verarbeiten kann, mit einem Analyseverfahren entscheiden zu können.

2 Grundlegende Konzepte

Wir modellieren Geschäftsprozesse als Workflow-Module. Ein WFM ist ein Workflow-Netz [1], das um Kommunikationsplätze erweitert wurde.

Definition 1 (Workflow-Modul) *Ein Stellen-/Transitions Petrinetz M ist ein* Workflow-Modul, *wenn P enthält einen ausgezeichneten* Anfangsplatz α *und einen ausgezeichneten* Endplatz ω; *P ist die disjunkte Vereinigung von Mengen P_M* (interne Plätze), P_I (Input-Plätze), *und* P_O (Output-Plätze); $F \cap (P_O \times T) = F \cap (T \times P_I) = \emptyset$; $m_{0_M}(\alpha) = 1$, $m_{0_M}(p) = 0$ *für alle $p \neq \alpha$; F enthält keine Zyklen, d.h. die transitive Hülle von F ist irreflexiv.*

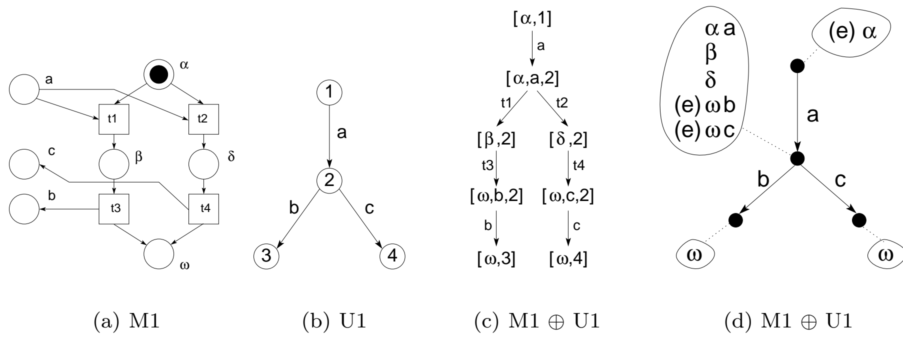
Die Markierung m_{0_M} ist die Anfangsmarkierung und m_{f_M} die Endmarkierung eines WFM. Für die Endmarkierung gilt: $m_{f_M}(\omega) = 1$ und $m_{f_M}(p) = 0$ für alle $p \neq \omega$. Die Abb.1(a) zeigt ein WFM mit dem Inputplatz a und den Ouputplätzen b und c.

Die folgenden Definitionen gehen auf [5] und [4] zurück. In [5] wird die Umgebung eines WFM als Automat eingeführt und für azyklische Workflow-Module auf Bäume mit einer maximalen berechenbaren Tiefe l_M eingeschränkt. Für ein Alphabet A bezeichnet $A*$ die Menge der endlichen Wörter über A. $\underline{a}$ bezeichne die Multimenge, die a genau einmal enthält und sonst kein Element.

Definition 2 (Umgebung) *Sei M ein Workflow-Modul und $A \subseteq (P_I \cup P_O)$. Eine mit A verbundene* Umgebung *ist ein Automat mit dem Alphabet A, einer Menge von Zuständen $Q \subseteq \{w \mid w \in A*, länge(w) \leq l_M\}$, so daß Q enthält mit q alle Präfixe von q, einer Übergangsfunktion δ mit $\delta(w,x) = \{wa\}$ wenn $wa \in Q$ und es ein a gibt mit $x = \underline{a}$, und $\delta(w,x) = \emptyset$ sonst, und λ (das leere Wort) als den Anfangszustand.*

Definition 3 (Komponiertes System) *Sei M ein Workflow-Modul und sei U eine Umgebung für M. Das* komponierte System *ist ein Transitionssystem mit $R_N(m_0) \times Q$ als die Menge von Zuständen und Kanten i)von $[m,q]$ nach $[m',q]$ wenn es eine Transition t in M gibt mit $m \xrightarrow{t} m'$; ii) von $[m,q]$ nach $[m',q']$ wenn es eine Multimenge B gibt, so daß in Q $q' \in \delta(q,B)$, für alle $p \in P_O$, $m(p) \geq B(p)$ und $m'(p) = m(p) - B(p)$, für alle $p \in P_I$ $m'(p) = m(p) + B(p)$, und für alle $p \in P_M$ gilt $m'(p) = m(p)$. Der Initialzustand ist $[m_0, \lambda]$.*

In der Abb. 1(b) ist eine Umgebung für das Modul M1 dargestellt und die Abb. 1(c) zeigt das komponierte System aus dem Modul M1 und der Umgebung U1.



(a) M1 (b) U1 (c) M1 ⊕ U1 (d) M1 ⊕ U1

Abbildung 1: Ein Workflow-Modul, eine Umgebung und das komponierte System

Definition 4 (Strategie) *Sei M ein Workflow-Modul. Eine Umgebung ist eine* Strategie *für M, wenn im komponierten System von jedem aus dem Anfangszustand erreichbaren Zustand, ein Zustand erreichbar ist, dessen erste Komponente m_{f_M} ist.*

Ein WFM ist bedienbar wenn es eine Strategie für das WFM gibt.

Definition 5 (Wissen der Umgebung) *Sei U eine mit A verbundene Umgebung und q ein Zustand von U. Sei S eine Menge von Zuständen des komponierten Systems aus U und M. Dann heißt $K(q) = \{m \mid \exists q : [m,q] \in S\}$ das Wissen der Umgebung U im Zustand q.*

Definition 6 (Deadlocks) *Sei M ein Workflow-Modul. Ein Deadlock von M ist eine Markierung, in der keine Transition aktiviert ist. Ein Deadlock m ist* intern, *wenn $m \neq m_f$, für jede Transition existiert ein $p \in P_M$ mit $[p,t] \in F$ und $m(p) = 0$ und alle Outputplätze unmarkiert sind. Ein Deadlock m ist extern, wenn er weder intern noch $m = m_f$ ist.*

Das Theorem [5] besagt, daß eine $(P_I \cup P_O)$ verbundene Umgebung U eine Strategie für ein Modul M ist, gdw. U eine nichtleere Menge von Zuständen besitzt und $K(q)$ für keinem Zustand einen internen Deadlock enthält und für jeden externen Deadlock $m \in K(q)$ das komponierte System aus M und U eine ausgehende Transition besitzt.

In der Abb. 1(d) ist das Wissen der Umgebung U1 für das Modul M1 abgebildet. In keinem Zustand enthält $K(q)$ einen internen Deadlock, und im Zustand 2 existiert für die beiden externen Deadlocks [ωb] bzw. [ωc] eine Transition im komponierten System (Abb. 1(c)). Damit ist U eine Strategie für das Modul M.

3 Transaktionale Workflow-Module

Wir erweitern WFM um ein Konzept, mit dem der Einfluß eines Datenbanksystems auf die Bedienbarkeit eines Geschäftsprozesses untersucht werden kann. Wir abstrahieren dabei von einem konkreten Datenbanksystem und beschreiben Zugriffe auf eine Datenbank als Lese-Schreibaktionen auf Variablen.

Sei *Var* eine Menge von Variablen. Eine Aktion auf einer Variablen ist $x \in Var$ ist entweder *Lesen* oder *Schreiben* von x (geschrieben r(x) bzw. w(x)). Eine Transaktion über $V \subseteq Var$ ist eine Menge von Aktionen auf Variablen $x \in V$ (geschrieben ta). Ein Schedule S einer Menge von Transaktionen TA ist eine Sequenz $S = a_1 ... a_m$ von Aktionen über dem Alphabet $\bigcup TA$. WFM mit Transaktionen werden mit Hilfe *gefärbter Workflow-Module (gWFM)* modelliert.

Definition 7 (gefärbtes Workflow-Modul) *Ein Petrinetz N ist ein* gefärbtes Workflow-Modul (gWFM)*, wenn i) N ist ein Workflow-Modul, ii)T ist die disjunkte Vereinigung von Mengen T_K (Kommunikationstransitionen), T_A (Aktionstransitionen), und T_M (interne Transitionen), iii)$F \cap (P_O \times T_A) = \emptyset$, $F \cap (T_A \times P_O) = \emptyset$, $F \cap (P_I \times T_A) = \emptyset$, $F \cap (T_A \times P_I) = \emptyset$, iv) T_A ist die disjunkte Vereinigung von Mengen T_{ta} (Transaktionen).*

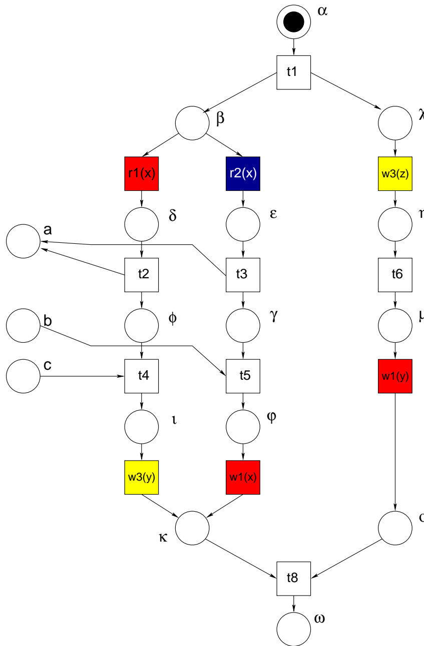


Abbildung 2: gWFM

Die Abb.2 zeigt ein gWFM über der Menge $TA = \{ta_1, ta_2, ta_3\}$ mit $ta_1 = \{r_1(x), w_1(x), w_1(y)\}$, $ta_2 = \{r_2(x)\}$, und $ta_3 = \{w_3(y), w_3(z)\}$. Zu jeder Aktionstransition wurde die zugehörige Aktion annotiert. Das komponierte System aus einem gWFM und einer Umgebung definiert eine Menge von sequentiellen Abläufen. Ein jeder dieser Abläufe entspricht einer Reihenfolge von Annotationen des Moduls. Eine solche Reihenfolge interpretieren wir als einen Schedule über TA und nennen ihn *Schedule des Moduls*. Aus der Menge der Schedules eines Moduls interessieren uns genau die Schedules, die die geforderten Eigenschaften besitzen. Geforderte Eigenschaften übersetzen wir in ein Petrinetz.

Definition 8 (Eigenschaftsnetz) *Ein Petrinetz E ist ein Eigenschaftsnetz wenn, i) P enthält eine Menge ausgezeichneter* Initialplätze *I und eine Menge ausgezeichneter* Endplätze *O; ii) $m_{0_E}(i) = 1$ für alle $i \in I$; iii) T ist die disjunkte Vereinigung von Mengen T_A (Aktionstransitionen), und T_E (interne Transitionen); iv) T_A ist die disjunkte Vereinigung von Mengen T_{ta} (Transaktionen); v) für alle $t \in T$ gilt: t kommt in jedem Ablauf höchsten einmal vor.*

In einem Eigenschaftsnetz ist M_{f_E} eine Menge von Markierungen und für alle $m_{f_E} \in M_{f_E}$ (Endmarkierung) und für alle $p \in O$ gilt $m_{f_E}(p) = 1$. Ein sequentieller Ablauf

$m_{0_E} \xrightarrow{*} m_{f_E}$ mit $m_{f_E} \in M_{f_E}$ heißt *gültiger Ablauf*. Ein Eigenschaftsnetz E heißt *gültig*, falls es einen gültigen Ablauf besitzt. Ein Eigenschaftsnetz besitzt eine Eigenschaft, wenn jeder gültige Ablauf des Eigenschaftsnetzes die Eigenschaft besitzt. Es ist darüber hinaus *vollständig* bezüglich einer Eigenschaft, wenn es eine Eigenschaft besitzt und alle möglichen Schedules über TA erzeugt werden, die diese Eigenschaft besitzen. Wenn ein Eigenschaftsnetz eine Eigenschaft besitzt, dann besitzt das Netz eine ausgezeichnete Menge von Zuständen, die nur über Abläufe erreicht werden können, die Schedules erzeugen, die die geforderte Eigenschaft besitzen. Die Gültigkeit einer Eigenschaft kann somit an einem Zustand abgelesen werden.

Wir komponieren gWFM und Eigenschaftsnetze und synchronisieren dadurch die Aktionstransitionen eines gWFM. Ein gWFM $M = (P, T, F)$ und ein Eigenschaftsnetz $E = (P', T', F')$ sind zueinander kompatibel, wenn $P \cap P' = F \cap F' = \emptyset$ und $T_A = T'_A$.

Definition 9 (transaktionales Workflow-Modul (taWFM)) *Seien $M = (P, T, F)$ ein Workflow-Modul und $E = (P', T', F')$ ein Eigenschaftsnetz und seien M und E zueinander kompatibel. Das komponierte System von M und E, geschrieben $\Pi = M \oplus E$, heißt* transaktionales Workflowmodul*, wenn gilt: i) $P_\Pi = P \cup P'$, ii) $T_\Pi = T_K \cup T_M \cup T_A \cup T_E$, iii) $F_\Pi = F \cup F'$.*

Betrachten wir jetzt die Eigenschaften, dessen Einfluß auf die Bedienbarkeit untersucht wurden. Der Untersuchung haben wir transaktionale Eigenschaften der klassischen Transaktionsverarbeitung zugrunde gelegt [6]. Wir fordern folgende Eigenschaften eines Schedules: 1.) Eine Transaktion kommt in einem Schedule über TA entweder vollständig oder gar nicht vor. 2.) Eine Aktion tritt in einem Schedule höchstens einmal auf. 3.) Kommt eine Transaktion in einem Schedule vor, dann steht die Leseaktion auf eine Variable x vor der Schreibaktion auf x in einem Schedule. Außerdem fordern wir Konfliktserialisierbarkeit (4.) und Striktheit (5.) bzw. Rigorosität (6.) [6].

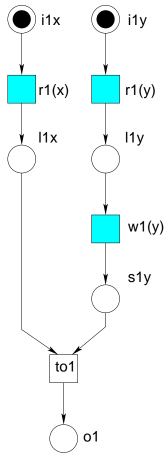


Abbildung 3: Transaktion

Die Eigenschaften 1-3 werden modelliert, indem jede Transaktion $ta \in TA$ in sein Eigenschaftsnetz übersetzt wird. In Abb. 3 ist die modellierte Transaktion $ta_1 = r_1(x), r_1(y), w_1(y)$ abgebildet. Jeder gültige Ablauf des Netzes erzeugt einen Schedule der die Eigenschaften 1-3 besitzt. Die geeignete Synchronisation der Aktionstransitionen stellt die folgende Konstruktion sicher:

Definition 10 (Konfliktrelationen von Transaktionen) *Sei TA eine Menge von Transaktionen über $V \subseteq Var$, die jeweils in ein Eigenschaftnetz E_{ta_i} modelliert wurden. Dann ist $E_0 = (P, T, F)$ mit $P = \bigcup_i P_i$, $T = \bigcup_i T_i$ $F = \bigcup_i F_i$ das aus den Transaktionen komponierte Eigenschaftsnetz. i) Zwei Transitionen $t \in T_{ta_i}$ und $t' \in T_{ta_j}$ mit $i \neq j$ stehen in E_0 zueinander in* Schreibrelation $t\mathcal{R}^1_x t'$ *bezüglich der Variablen x mit $x \in V$, gdw. für t gilt $w_i(x)$ und für t' gilt $\{r_j(x), w_j(x)\}$. ii) Zwei Transitionen $t \in T_{ta_i}$ und $t' \in T_{ta_j}$ mit $i \neq j$ stehen in E_0 zueinander in* Schreib-/Leserelation $t\mathcal{R}^2_x t'$ *bezüglich der Variablen x mit $x \in V$, gdw. für t gilt $r_i(x)$ und für t' gilt $w_j(x)$.*

Definition 11 (Konstruktion eines synchronisierten Eigenschaftsnetzes) *Sei $E_0 = (P, T, F)$ mit $P = \bigcup_i P_i$, $T = \bigcup_i T_i$, $F = \bigcup_i F_i$ ein gemäß Def. 10 konstruiertes Eigenschaftsnetz. Das synchronisierte Eigenschaftsnetz über TA wird aus E_0 wie folgt konstruiert: i)Für alle Variablen $x \in V$ gilt, wenn die Relation $\mathcal{R}^1_x$ nicht leer ist, dann entsteht das Netz E_1 aus E_0 indem E_0 der Platz mit dem Label x und die folgenden Kanten hinzugefügt werden: $[x, t]$ und $[t_{o_i}, x]$. Für alle Variablen x gilt in der Anfangsmarkierung $m_{0_E}(x) = 1$. ii) Für alle Variablen $x \in V$ gilt, wenn die Relation $\mathcal{R}^2_x$ nicht leer ist, dann entsteht das Netz E_2 aus E_1 indem für jedes Element aus $\mathcal{R}^2_x$, dem Netz E_1 ein Platz mit dem Label K_{i,j_x} hinzugefügt wird. Die Relation $\mathcal{R}^2$ ist offensichtlich asymmetrisch, so daß $K_{i,j_x} \neq K_{j,i_x}$ ist. Für alle Variablen $x \in V$ gilt, wenn $[t, t'] \in \mathcal{R}^2_x$ dann werden dem Netz folgende Kanten hinzugefügt: $[x, t]$, $[t, x]$, $[K_{i,j_x}, t]$, $[t_{o_i}, K_{i,j_x}]$, $[K_{i,j_x}, t_{o_j}]$, $[t_{o_j}, K_{i,j_x}]$. Für alle Variablen x gilt in*

der Anfangsmarkierung $m_{0_E}(K_{i,j_x}) = 1$. *Die Menge* $\mathcal{K}$ *ist die Menge aller per Konstruktion eingefügten Plätze.*

Die Abb. 4 zeigt das taWFM $\Pi = \mathsf{gM}_2 \oplus \mathsf{E}_2$. Das Netz E_2 zeigt exemplarisch die Konstruktion des Eigenschaftsnetzes über der durch gM_2 gegebenen Menge der Transaktionen $TA = \{ta_1, ta_2, ta_3\}$. In [2] konnte gezeigt werden, daß gemäß Definition 11 konstruierte Eigenschaftsnetze die Eigenschaften 1-6 besitzen und darüber hinaus vollständig bezüglich der Eigenschaften 1-6 sind.

4 Analyse transaktionaler WFM

Abbildung 4: taWFM Π

In taWFM werden transaktionale Eigenschaften als internes Verhalten eines WFM repräsentiert. Aufgrund der Beschaffenheit der Eigenschaftsnetze kann die Einhaltung einer Eigenschaft anhand eines Zustandes abgelesen werden. Da die Bedienbarkeitsanalyse eines Moduls auf der Auswertung von erreichbaren Zuständen eines komponierten Systems aus Modul und Umgebung basiert, kann die Analyse auf taWFM angewandt werden.

Für die Klassifikation von Dealocks eines taWFM muß der Begriff der Endmarkierung angepaßt werden (siehe [2]).

Die Abb. 5(a) zeigt das Wissen der Umgebung $\mathsf{U1}$ für das komponierte System $\mathsf{gM}_2 \oplus \mathsf{U1}$. Es ist offensichtlich, daß $\mathsf{U1}$ keine Strategie für das Modul gM_2 ist, da es die externen Deadlocks im Zustand $\mathsf{2}$ nicht auflösen kann. Darüber hinaus existiert für das Modul gM_2 keine Umgebung, die eine Strategie für das Modul ist. Es ist nicht bedienbar. Die Abb. 5(b) zeigt das Wissen der Umgebung $\mathsf{U1}$ für das komponierte System $\Pi \oplus \mathsf{U1}$. Das Wissen enthält im Zustand $\mathsf{3}$ einen internen Deadlock, so daß $\mathsf{U1}$ auch hier keine Strategie für das Modul Π ist. Allerdings kann jetzt eine Umgebung berechnet werden, die eine Strategie für Π ist [5]. Der Zustand $\mathsf{3}$ kann gestrichen werden, und die entstandene Umgebung ist eine Strategie für Π. Das taWFM Π ist damit bedienbar. In [2] haben wir weiterhin gezeigt, daß Bedienbarkeit unter Berücksichtigung transaktionaler Eigenschaften eingeschränkt oder unmöglich gemacht werden kann.

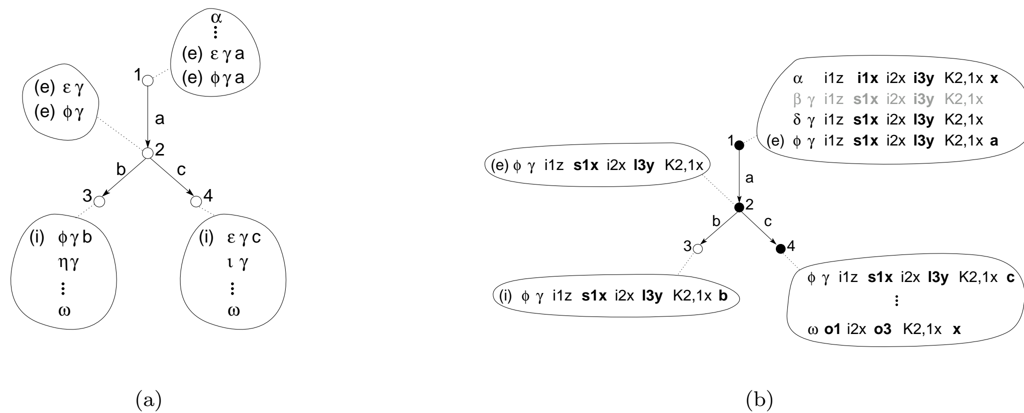


Abbildung 5: Wissen der Umgebung U1 für das System $\mathsf{gM}_2 \oplus \mathsf{U1}$ (li.) bzw. das System $\Pi \oplus \mathsf{U1}$ (re.).

5 Zusammenfassung

Wir haben in dieser Arbeit eine Möglichkeit geschaffen, Workflow-Module um transaktionales Verhalten zu erweitern. Den Einfluß einer Datenbank auf den Kontrollfluß eines Prozesses haben wir unabhängig von einem konkreten Datenbanksystem in ein Petrinetz übersetzt und in das Modul als internes Verhalten integriert. Der Nachweis der Bedienbarkeit eines taWFM beantwortet neben der Frage, ob die Funktionalität eines Moduls durch eine Umgebung genutzt werden kann auch die Frage, ob die Transaktionen des Moduls korrekt abgearbeitet werden können.

Für die weiteren Arbeiten sehen wir mehrere Ansatzpunkte. Die hier vorgestellten Ergebnisse beruhen auf azyklischen WFM. Für die Analyse zyklischer WFM existieren bereits eine Reihe von Ergebnissen. Es gilt, taWFM auf den zyklische Fall zu erweitern. Die klassischen Transaktionskonzepte sind für Geschäftsprozesse oft zu rigide. Es gilt in weiteren Arbeiten den Einfluß weiterer Transaktionskonzepte, insbesondere Konzepte mit Fehlerbehandlung und Kompensation und Konzepte für verteilte Datenbanken, auf die Bedienbarkeit zu untersuchen.

Literatur

[1] W. M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[2] Carsten Frenkler. Modellierung und Analyse transaktionaler Geschäftsprozesse. Diplomarbeit, Humboldt-Universitt zu Berlin, Juli 2005.

[3] A. Martens. *Verteilte Geschäftsprozesse - Modellierung und Verifikation mit Hilfe von Web Services*. PhD thesis, Institut für Informatik, Humboldt-Universität zu Berlin, 2004.

[4] P. Massuthe and K. Schmidt. Operating guidelines - an alternative to public view. Techn. Report 189, Humboldt-Universität zu Berlin, 2005.

[5] K. Schmidt. Controllability of business processes. Techn. Report 180, Humboldt-Universität zu Berlin, 2004.

[6] G. Vossen and M. Gross-Hardt. *Grundlagen der Transaktionsverarbeitung*. Addison-Wesley, Bonn, 1993.

Operating Guidelines for Services

Peter Massuthe and Karsten Schmidt
Humboldt-Universität zu Berlin
Institut für Informatik
Unter den Linden 6
D-10099 Berlin
{massuthe, kschmidt}@informatik.hu-berlin.de

Abstract

In the service-oriented architecture (SOA), we distinguish three roles of service owners: service providers, service requesters, and service brokers, and the three standard operations publish, find, and bind.

We provide a formal method based on Petri nets to model services. We suggest *operating guidelines* as a convenient and intuitive artifact to realize *publish*. Then, the *find* operation reduces to a matching problem between the requester's service and the operating guideline.

1 Introduction

A *service* is an artifact that consists of an interface and internal control. The *service-oriented architecture* (SOA) [2] provides a framework for the interaction of services. It distinguishes three roles of service owners: *service provider*, *service broker*, and *service requester* (see Fig. 1). It postulates a general protocol for interaction: A service provider registers at the service broker by submitting information about how to interact with its service. The service broker manages such information about all registered service providers and allows a service requester to *find* an adequate service provider. Then, the service of the provider and the service of the requester may *bind* and start interaction.

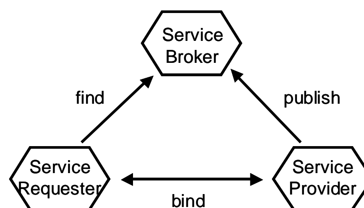


Figure 1: The service-oriented architecture.

The interaction of services may cause nontrivial communication between a requester and a provider. Consider, as a running example for a provided service, a vending machine as depicted in Fig. 2. If this service is bound to a requester's service, the requester must send a coin (€), press a button (T or C), and finally receive a beverage (B).

It is obviously desirable that a service requester gets assigned only such a service provider that their services do not ill-communicate with each other (such as running into a deadlock or sending unanticipated messages). In our example, the broker must not deliver our vending machine service to a requester that wants to pay in other currencies than € or a requester who expects the beverage before paying.

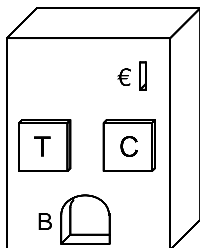


Figure 2: A vending machine that sells, for 1 Euro, either a cup of tea (press T), or a cup of coffee (press C).

For this purpose, the service broker needs information about the internal control structure of the provider's service – the provider's interface only (e.g. a WSDL specification) is not sufficient. Publishing the whole internal control to the service broker would solve the problem. This is, however, not feasible, as the service provider may want to keep its internal structure secret.

We propose the published information to be an *operating guideline* for the provider's service P. The operating guideline for P essentially represents, in a condensed form, the set of *all* well-communicating services R of requesters of the service P. Thus, the operating guideline for the vending machine in our example would cover requesters that pay in €, but not a requester who pays in £, for instance.

In our operating guidelines approach, the broker's task to select a fitting provider for a querying requester reduces to a matching problem. This is basically a check whether the requester's service "follows" the published operating guideline.

2 Models of Workflow Services

We suggest *open workflow nets* (oWFNs) to model services. An oWFN is essentially a liberal version of a van der Aalst workflow net, enriched with communication places. Each communication place of an oWFN models a channel to send (receive) messages to (from) another oWFN. Thereby, we abstract from data and just model the occurrence of messages as undistinguishable tokens.

An *open workflow net* is a place transition Petri net $N = (P, T, F)$ together with

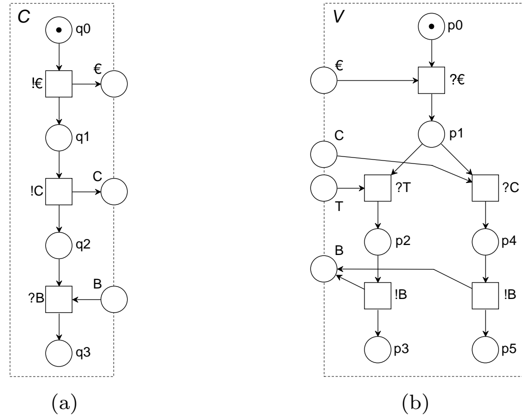
- two sets $in, out \subseteq P$, such that for all transitions $t \in T$ holds: if $p \in in$ ($p \in out$) then $(t, p) \notin F$ ($(p, t) \notin F$),
- a distinguished marking m_0, called the *initial marking*, and
- a set Ω of distinguished markings, called the *final markings* of N.

The places in *in* (*out*) are called *input* (*output*) places. The set $in \cup out$ is called the *interface* of N. As a convention, we label a transition t connected to an input (output) place x with $?x$ ($!x$). The *inner of* N can be obtained from N by removing all interface places, together with their adjacent arcs.

As an example, Fig. 3(b) shows the oWFN V, modeling the vending machine of Fig. 2. The places €, T, C, and B denote an inserted coin, the button T or C pressed, and a beverage delivered, respectively. There are two final markings of V: a single token on p3 or a single token on p5.

A corresponding customer would insert a coin, press one of the buttons and later on receive the beverage. The oWFN C, depicted in Fig. 3(a), models a customer of the vending machine, pressing the *coffee* button.

The interaction of two oWFNs is reflected by their *composition*. The composition of two oWFNs M and N is an oWFN again, denoted $M \oplus N$, and constructed essentially as the component-wise union of M and N.



(a) (b)

Figure 3: An oWFN V modeling the vending machine (right), together with an oWFN C modeling a customer who wants coffee (left).

As an example, Fig. 4 shows the oWFN $C \oplus V$. This oWFN has two terminal markings, m_1 and m_2, with $m_1(\mathsf{q3}) = m_1(\mathsf{p3}) = 1$, $m_2(\mathsf{q3}) = m_2(\mathsf{p5}) = 1$, and no tokens on all other places. Notice that $in_{C \oplus V} = \{\mathsf{T}\}$ and $out_{C \oplus V} = \emptyset$.

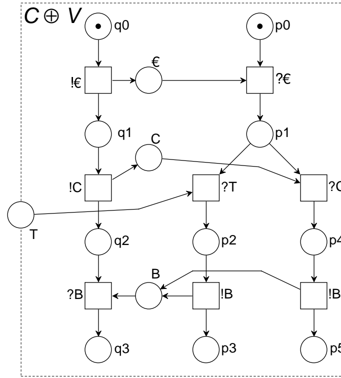


Figure 4: The composed oWFN $C \oplus V$ of Fig. 3(b) and Fig. 3(a).

Now assume another customer's oWFN E where just !€ and ?B are executed in sequence. E models an erroneous customer service of the vending machine, as the customer apparently "forgets" to press one of the machine buttons, and both services deadlock. The oWFN C represents an "adequate" partner for V, whereas E is not "adequate" for V.

In technical terms, a marking m of an oWFN is a *deadlock* if m enables no transition at all. It is easy to see that in the oWFN $C \oplus V$, the only reachable deadlock is a final marking. In contrast, in $E \oplus V$ (not shown here), the marking m with $m(\mathsf{r1}) = m(\mathsf{p1}) = 1$ and $m(p) = 0$ for all other places $p \in P_{E \oplus V}$ is a reachable deadlock which is no final marking.

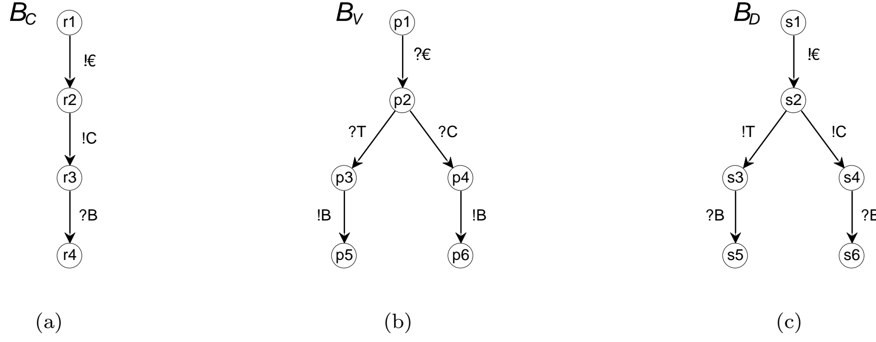
An oWFN in which all deadlocks are final markings is called *weakly terminating*. Given an oWFN P, we call an oWFN R a *strategy* for P iff the oWFN $R \oplus P$ is weakly terminating. For example, C is a strategy for V, whereas E is not.

3 Publish

As mentioned earlier in this paper, information published by a service provider on his service P must enable the service broker to decide whether or not a requester's service R is a strategy

for P. Thus, we suggest to publish directly a description of the set of all *strategies* (i.e. all properly interacting oWFNs) R for P. In fact, we provide a description of the *behaviors* of all strategies R. We call this description operating guideline for P and write OG_P.

The behavior of an oWFN N is the reachability tree of the inner of N, where transitions are annotated with $!x$ ($?x$) if they put a token on (take a token from) an interface place.



(a) (b) (c)

Figure 5: The behaviors B_C and B_V of the oWFNs C and V and a more permissive customer's behavior B_D.

For example, the behaviors B_C and B_V of the oWFNs C and V of Fig. 3(a) and Fig. 3(b) are depicted in Fig. 5(a) and Fig. 5(b), respectively.

For the purpose of simplicity only, we constrain the considerations in this paper to *deterministic* behaviors of strategies. A behavior is deterministic iff every edge of the behavior has exactly one expression $!x$ or $?y$ attached (i.e. there is no silent move τ), and there is no node in the behavior where two leaving edges have the same expression attached. All behaviors shown in this paper are deterministic.

Let the set of the (deterministic) behaviors of *all* strategies for P be denoted by $\mathcal{B}_\mathcal{P}$. We can establish a (partial order) relation, *more permissive*, to behaviors of $\mathcal{B}_\mathcal{P}$: A behavior B is more permissive than a behavior B' if B' is isomorphic to a subtree of B containing the root.

As an example, Fig. 5(c) shows the behavior B_D of some customer D of the vending machine, who first inserts a coin and then *decides* for coffee or tea. B_D is more permissive than B_C, whereas neither B_C is more permissive than B_V, nor B_V is more permissive than B_C.

In [6, 5], we show that, for every oWFN P, the set $\mathcal{B}_\mathcal{P}$ has a unique most permissive element, the *most permissive behavior*, B^*. Consequently, we call every oWFN R with the most permissive behavior as its behavior (i.e. $B_R = B^*$), a *most permissive strategy* for P.

The main property of the most permissive behavior B^* is that it comprises all behaviors of strategies for P: Every behavior B_R of a strategy R for P is (isomorphic to) a subtree of B^*. Thus, the most permissive behavior serves as the first ingredient to the operating guideline for P.

Unfortunately, the converse is not true. Not every subtree of the most permissive behavior is itself a behavior of a strategy. Thus, the remaining problem is to distinguish those subtrees of the most permissive behavior which are behaviors of strategies from those subtrees which are no behaviors of strategies. Our solution to this task is again based on a result proven in [6, 5]:

Given a provided oWFN P and a behavior B_R of some requester's service R, we can decide for each node q_R of B_R whether or not it can cause a deadlock in $R \oplus P$. This is basically determined by the edges that leave q_R: Whether or not R is a strategy for P depends on present or missing edges in B_R. Thus, we code the constraints for edges leaving q_R as a boolean formula over edge labels and annotate it to q_R. B_R satisfies these constraints if and only if R is a strategy.

Since the most permissive behavior B^* is a behavior of some strategy, we can annotate B^*, too. A subtree of B^* is thus a behavior of a strategy if and only if it still satisfies the attached annotations. The annotations to the nodes of B^* constitute the second ingredient and complete the operating guideline for P.

As an example, the operating guideline OG_V for the vending machine V (of Fig. 3(b)) is depicted in Fig. 6. The possibility to first press a button and then inserting a coin comes from the proposed asynchronous communication.

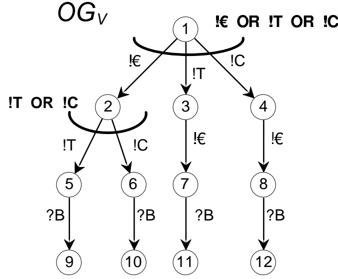


Figure 6: The operating guideline OG_V for the vending machine V. Annotations of nodes with only one outgoing edge are skipped.

The operating guideline for an oWFN P can be constructed automatically. Construction algorithms can be found in [6]. The construction bases on the provider's oWFN P, only. As we assume the construction is done by the provider himself, this is acceptable. Furthermore, the annotations just reflect needed actions of the environment such that the composed system does not deadlock. The annotations do not reflect *why* a deadlock can be reached, nor how the deadlock looks like. When published, operating guidelines therefore hide most details about the internal control structure of the provided service, that the service provider might want to keep secret.

In current research, we develop efficient representations of operating guidelines as binary decision diagrams (BDDs) [1]. BDDs are a data structure that proved to be capable of representing large transition systems in the area of model checking. Preliminary results in the application of BDDs representing operating guidelines are promising.

4 Find

Matching a requester's service with an operating guideline OG_P is rather simple. Given an oWFN R of the requester, we first compute its behavior, B_R. This is simple and well-understood state space generation. Then, we need to check whether B_R (a) is isomorphic to a subtree of OG_P and (b) satisfies the annotations. Thereby, a literal $?x$ ($!x$) at some node of OG_P is evaluated to *true* if there is an outgoing edge from the corresponding node in B_R labeled $?x$ ($!x$) and evaluated to *false*, otherwise.

It is easy to see that B_C and B_D match OG_V, whereas the behavior B_E of the erroneous partner oWFN E would not match OG_V.

Checking the subtree property can be solved by a coordinated depth-first search through both behaviors. Its run-time is linear in the size of R's behavior. Checking the annotations amounts to computing a value of a boolean formula and can thus be implemented efficiently. Thus, the *find* procedure based on operating guidelines turns out to be very efficient.

As an alternative to operating guidelines, the concept of *public views* [3, 4] has been proposed. It suggests to publish an abstract version P' of a process P to the service broker. A *find* based on public views is more complex than a find based on operating guidelines: Given a requesting service R and a public view P' of a provided service P, a service broker

must decide whether R is a strategy for P. Currently, the only available approach to this problem is to build the system composed of P' and R and to check its state space for deadlocks and end states with unconsumed messages. The size of the state space is typically in the same order of magnitude as the number of states of P' times the number of states of R. Checking the state space for deadlocks is linear in that number and thus more complex than matching R with OG_P.

5 Conclusion

We propose oWFNs as a formal model for services that use workflows as their internal control structure. We showed that it is possible to automatically compute, for an oWFN P, an operating guideline OG_P which characterizes the set of all (deterministic) behaviors of strategies for P. We propose to use OG_P as information published in service repositories. This way, it is easy for the service broker to assign well-behaving pairs of provider's and requester's services: the requester's service must match the operating guideline published for the provided service. Generating an operating guideline may be complex, but we expect that this complexity can be managed through the use of advanced technology developed in the area of model checking. In turn, matching a service with an operating guideline is considerably simpler than checking compliance between a requester's service and a public view of a provided service.

We see several directions for future research. First, we need to extend the approach to services containing cycles. We have a number of preliminary results on this matter. Second, we study specialized operating guidelines, characterizing, e.g., the set of all those strategies of the considered vending machine that inevitable lead to the delivery of coffee. Third, we investigate further important aspects which are relevant for selecting a service such as real-time constraints or cost models. We want to extend the concept of operating guidelines to those aspects.

We are convinced that our approach is well suited to implement the service discovery outlined in the SOA triangle. Our concept is quite close to those guidelines that are attached to real vending machines. The concept of operating guidelines has thus been already successful in every-day life for a long time.

References

[1] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[2] Karl Gottschalk. Web Services Architecture Overview. Ibm whitepaper, IBM developerWorks, 01 September 2000. http://ibm.com/developerWorks/web/library/w-ovr/.

[3] F. Leymann, D. Roller, and M. Schmidt. Web services and business process management. *IBM Systems Journal*, 41(2), 2002.

[4] A. Martens. *Verteilte Geschäftsprozesse - Modellierung und Verifikation mit Hilfe von Web Services*. PhD thesis, Institut für Informatik, Humboldt-Universität zu Berlin, 2004.

[5] W. Reisig, K. Schmidt, and Ch. Stahl. Kommunizierende Geschäftsprozesse modellieren und analysieren. Accepted for *Informatik – Forschung und Entwicklung*, 2005.

[6] K. Schmidt. Controllability of business processes. Techn. Report 180, Humboldt-Universität zu Berlin, 2004.

Service Modeling Based on High-Level Petri Nets

Michael Köhler, Jan Ortmann
University of Hamburg, Computer Science Department
Vogt-Kölln-Str. 30, D-22527 Hamburg
{koehler,ortmann}@informatik.uni-hamburg.de

Abstract— **Dynamic service composition and coordination requires a semantics of processes. An ontology for processes can then supply agents with a common understanding of process behavior. The process semantics should reflect behavioral aspects as well as data manipulation aspects of the process.**

In this paper we introduce an approach to capture the semantics of a process based on high-level Petri nets. Additionally ideas from traditional ontologies have been adopted providing a conceptualization of the data.

Keywords: Agents, Web services, High-level Petri Nets, Service Composition, Service Orchestration, Process Ontologies

I. Introduction

The dynamic composition of services has become an area of major interest promising great benefits for the entire e-commerce world. To enable dynamic service composition a formal semantic description of what a service and of what a composition of services does is required. Through this description, an agent can find services needed and can compose them resulting in the desired behavior. This enables the agent to dynamically adapt to changes in its environment, since process parts can be exchanged with similar ones allowing for dynamic adaption and optimization.

An agent has some knowledge about the state of the world it is located in. To change this state the (software) agent can communicate to the world via message passing (i.e. through the invocation of other services). To provide the agent with a conception of what changes take place upon calling another service, the agent needs a semantic description of the service.

In our understanding, this semantic description consists of three parts:

- A common vocabulary shared by the participants, which is provided by a common ontology,
- a description of the functional behavior of single Web services or parts thereof, and
- a description of the coordination of multiple services.

This paper provides a theoretical framework for semantically describing services and their compositions based on algebraic Petri nets. As a common vocabulary, we will use an ontology, which is "an explicit specification of a conceptualization"[7], and which is formally described by description logics [1]. Good ontology design is a major research area and we will not discuss this issue but refer to [11], [13] instead.

The description of the functional behavior will be provided by an order-sorted algebra [6]. Here, the types of the ontology are reflected by the ordering of the sorts. Roles of an ontology, i.e. binary relations, are handled by an extension of the original algebraic concepts consisting of binary predicates. These predicates can be interpreted differently according to the current state a system is in.

Workflow nets [12] formalize the composition and coordination of Web services. By the execution of the nets, we can inspect, what the result of executing a set of services is. We see an algebraic Petri net [8] based approach as promising since it can solve some major problems for service composition, e.g. the behavioral semantics as well as the operational semantics of services can be covered.

Having a net with a workflow like structure allows an agent to execute the possible processes of the net. Thus an agent can pick a sequence of services that is best suited for a given objective. Planning can be considered as reaching a particular marking from a given one. This is closely related to the planning as model checking approach as e.g. in [4].

The paper starts with a brief and rather informal introduction to Description Logics and Algebras in Section II and Section III. We will then introduce Service Description Nets in Section IV and their reduction to algebraic nets in Section V. Section VI will draw a conclusion and give an outlook.

II. Ontologies and Description Logics

Ontologies specify the concepts that are used in a given domain. A concept is determined by its attributes and by the roles existing between it and the other concepts. Concept descriptions are built inductively from a set of primitive concepts, roles and attributes by using concept and role constructors. The most widespread approach is to model the concepts of an ontology by a *Description Logic* (DL) [1]. Different Description Logics vary in their expressiveness, i.e. in the number and kind of constructors they provide. We distinguish between attributes as functional roles and roles (non-functional). An example of a concept description would be the definition of a concept

$$\text{LibraryBook} \doteq \text{Book} \sqcap \forall \text{signature.Signature} \sqcap \forall \text{subject.String}$$

Description logics knowledge bases are divided into TBoxes and ABoxes. Let N_C, N_A and N_R denote sets of concept symbols, attribute symbols and role symbols and let N_I denote the set of individuals. Additionally let N_C, N_A, N_R and N_I be pairwise disjoint. In this section we will subsume attributes under the more general concept of roles assuming $N_A \subseteq N_R$. The distinction will become more important in the next sections, however.

TBoxes (terminology boxes) hold intensional knowledge in form of a terminology. They are built through declarations of general properties of concepts and thus restrict the set of possible worlds. More formally, given a DL $\mathcal{L}$, a $\mathcal{L}$-TBox

Construct Name	Syntax	Semantics
Atomic Concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
Disjunction	$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Limited Exists Restriction	$\exists R.\top$	$(\exists R.\top)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y.\langle x,y\rangle \in R^{\mathcal{I}}\}$
Value Restriction	$\forall R.C$	$(\forall R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \forall y.\langle x,y\rangle \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
Atomic Role	P	$P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
Universal Concept	$\top$	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
Empty Concept	$\bot$	$\bot^{\mathcal{I}} = \emptyset$
Atomic Negation	$\neg A$	$(\neg A)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus A^{\mathcal{I}}$
Exists Restriction	$\exists R.C$	$(\exists R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y.\langle x,y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
Conjunction	$C \sqcup D$	$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Negation	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$

TABLE I
SYNTAX AND SEMANTICS OF THE DESCRIPTION LOGIC $\mathcal{ALC}$

In the table, A denotes an atomic concept, C and D denote arbitrary concepts and R and S represent roles.

$\mathcal{T}$ is a finite set of concept definition axioms of the form $A \doteq C$, with $A \in N_C$ being a concept symbol and C being a $\mathcal{L}$-concept description (the defining concept). The concept description is a term over the set of concept, attribute and role symbols. The syntax for terms in the DL $\mathcal{ALC}$ is given in Table I. A concept symbol is called defined (in $\mathcal{T}$) if it occurs on the left-hand side of a concept definition, otherwise it is called primitive. Defined symbols are required to occur exactly once on the left-hand side of concept definitions.

ABoxes (assertion boxes) on the other hand allow to describe a specific state of the world. They hold extensional knowledge that is specific to the individuals of the domain of discourse. A *concept assertion* is denoted $C(i)$ where C is a concept description and $i \in N_I$ is an individual. A role assertion is denoted $R(i_1, i_2)$ where $R \in N_R$ is a role symbol and $i_1, i_2 \in N_I$ are individuals. An ABox $\mathcal{A}$ is a finite set of concept and role assertions.

Let C and D be concept descriptions. D *subsumes* C, denoted as $C \sqsubseteq D$, iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for all interpretations $\mathcal{I}$. $C \equiv D \Leftrightarrow C \sqsubseteq D \wedge C \sqsupseteq D$ denotes the *equivalence* of two concept descriptions. ABoxes are defined with respect to some TBox. The LibraryBook example above would be part of the TBox $\mathcal{T}$ whereas the instance LibraryBook(book43) would be part of an ABox respecting $\mathcal{T}$.

III. FUNCTIONALITY AND ALGEBRAS

The functionality of a service is modeled by algebras. These algebraic representations are split into a static and a dynamic part. While the former represents operations that do not change their behavior (such as arithmetic operations), the latter represents operations that might be interpreted differently in different states. These ideas are related to abstract state machines [2], [9] on the one hand and hidden algebras [5] on the other hand. As opposed to standard algebras, we introduce predicates representing the dynamic part and corresponding to the roles of an ontology. The operations of the algebra determine the static behavior.

To reflect the concept hierarchy of an ontology, we use order-sorted algebra as described in [6]. This allows us to map the concepts of an ontology to the sorts of a signature. An order-sorted signature consists of a set of sort symbols together with a partial order, such that the sort symbols are partially ordered. Additionally we have a set of operation symbols on the sort symbols. For a signature of integers we might define the sorts INT and NAT, such that NAT is a subsort of INT and we might define the operation symbols $+$ and $-$ which are both defined on INT.

An order-sorted algebra interprets the symbols and sorts defined in a signature in a way that respects the order of the sorts. To each sort it assigns a corresponding set of carriers and to each operator it assigns an operation on the corresponding carrier sets. A more formal definition can be found in the Appendix.

For a signature we have a set of variables that will be assigned values by an assignment function. Terms are built from variables and operation symbols of the signature. The evaluation of a term is the extension of a given assignment to terms in the usual way.

Equations consist of a set of variables together with two terms that are stated to be equal. A specification is a signature together with a set of equations. An algebra respecting a specification respects the signature and all equations of the specification.

We augment the usual definition of order-sorted algebras by predicates, which are binary relations. While operators represent the fixed operations on different types, predicates represent the dynamic aspects and the interpretation of a predicate might be different in different states.

A *predicate signature* is a specification together with a set of predicate symbols. A *structure* of a predicate signature is an algebra specifying the carrier sets and the operations and a set of pairs assigned to each predicate symbol. A structure is functional if for each value, there is a only one pair for which this value is in the first place of the tuple.

A predicate signature is compatible with an ontology if each named concept has a corresponding sort in the signature and these sorts respect the hierarchy of the ontology and we furthermore have a domain sort for each role, i.e. if (x, y) is

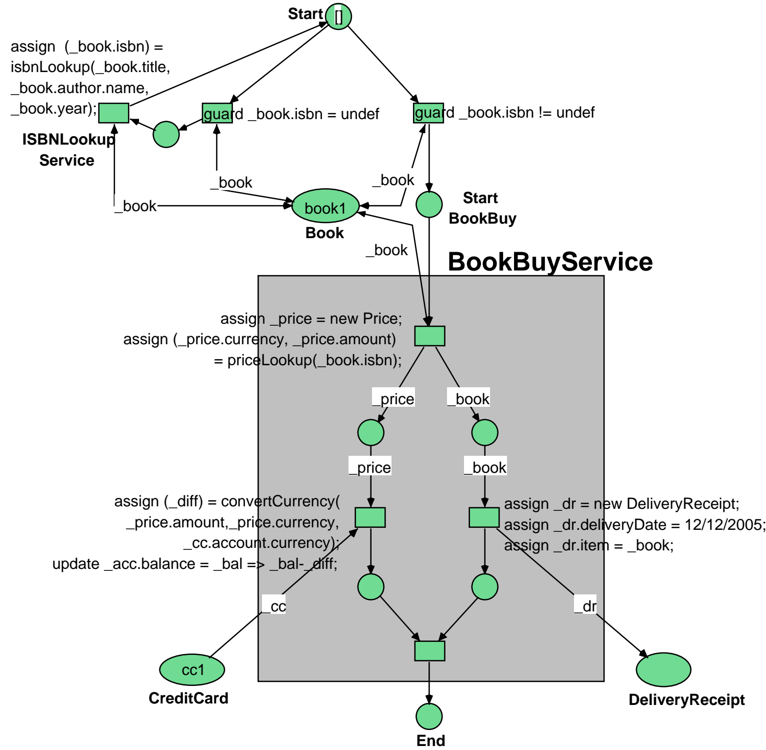


Fig. III.1. The Book Service Description Net

a role, there must be a least sort where all x are contained in. Additionally the predicate symbols need to respect the sorts as well. Again, a more formal definition is given in the Appendix.

If a predicate representation is compatible with an ontology it can represent this ontology, i.e. there is an interpretation of the predicate representation that represents the ontology. Through this, we can now introduce a net class changing the current attributes of concepts of an ontology by changing the interpretation of its predicate representation, which we will call Service Description Nets (SDN). These nets have special inscriptions to change the interpretation of the predicates, which we will call updates. Additionally we have variable assignments that assign the result of a term evaluation to a variable, so that we can create new items or use local variables.

IV. Modeling of Service Behavior using Nets

We introduce Service Description Nets (SDN) to model the potential results of executing a sequence of services. Each service affects the current state of the agents world by changing the value of data items. A service can either create new data or it can modify existing data. This can be modeled in a way that an agent can reason about the effects of a service execution by using the concepts of the last section. An algebra represents the operations defined on the data whereas a set of predicates represents the roles given by an ontology.

As inscription on a SDN we have variable assignments and (predicate) update statements. A *variable assignment* is one of the following

1. assign v = new s,
2. assign $v = \underline{t}'$, or
3. assign $v = \underline{t}'.\pi$, and

a *predicate update* is of the form

4. set $\underline{t}.\pi = \underline{t}'$ with $\underline{t}' \in \mathbb{T}_{\Sigma_O,s}(X)$.

A set of variable assignments Ass (*Assignment set*) is *consistent* if no left hand side occurs more than once in Ass, i.e. (assign $v = rexp) \in Ass$ and (assign $v = rexp') \in Ass$ implies $rexp = rexp'$. A set of predicate updates is *consistent* similarly if no left hand side occurs more than once in it. If a variable occurs on the left hand side of a variable assignment it is called bound, otherwise it is called free.

When a set of assignments and update statements is executed in a state σ first all variable assignments are executed yielding a state σ^* and then all update statements are executed yielding the successor state σ'. Through this, the execution order of the updates is irrelevant for consistent update sets. The **new** assignment creates a new item of sort s, the $v = \underline{t}'$ assignment assigns the value of $\underline{t}$ under the given assignment to v and the $v = \underline{t}'.\pi$ assignment assigns the value of x to v such that $(\bar{\alpha}(\underline{t}', x) \in \pi_A)$ under the given assignment and the given structure.

SDNs are closely related to algebraic nets [8]. The places are typed, the arcs are inscribed by variables and transitions may have guards. They are, however, more specific than algebraic nets. We will first introduce SDNs and will then show, how they can be equally represented as algebraic nets. For the control flow, we will use a dedicated sort called $s_{\mathbf{token}}$ which has the constant $\bullet$ as its only operation symbol. If an arc or place has no inscription it is implied that it is of sort

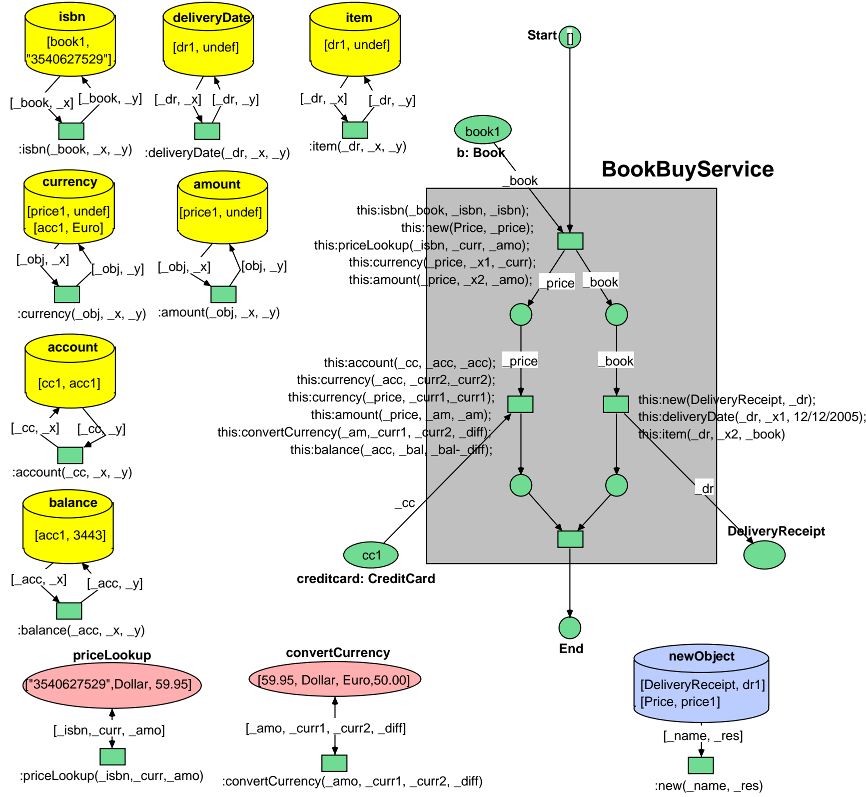


Fig. IV.2. The Book Service as Simplified Net

$s_{\mathbf{token}}$. Figure III.1 shows a SDN.

For convenience reasons we introduce two abbreviating notions for SDN inscriptions:

5. update $\underline{t}.\pi = v' \Rightarrow \underline{t}'$, which is equivalent to the assignment assign $v' = \underline{t}'.\pi$ together with the predicate update set $\underline{t}.\pi = \underline{t}'$, and
6. $\underline{t}.\pi$ within a term expression, which is equivalent to assign $v' = \underline{t}'.\pi$ and the usage of v' instead.

The second abbreviation allows us to write e.g.

assign v = _book.author.name instead of
assign v1 = _book.author and assign v = v1.name

Additionally we require a pool of items where the new assignment can obtain a new item from.

Besides it is often convenient in practice to have not only binary relations but relations of arbitrary arity. Here, we will only consider functional relations and call them predicates as well.

For an n-ary predicate π we additionally introduce

assign $(v_1, \dots, v_j) = \pi(t_1, \dots, t_i)$

which replaces the value of $v_1, \dots, v_i$ by appropriate values such that $(t_1, \dots, t_i, v_1, v_j) \in \pi_A$.

Through this, we can express database queries or external queries that assign values to more than one variable. In Figure III.1 the action operations are represented by bigger places such as `priceLookup`. For action operations the definition of the predicate representation has to be modified, to allow n-ary predicates.

In Figure III.1 we see a fictional service for buying a book. Here, variables are denoted with a leading underscore and guards are denoted by the keyword guard. In the example, first we need to lookup the isbn of the book if it is not known and then we can invoke the service. The service creates a new price object with the properties currency and amount. Then it invokes a convertCurrency service assigning the converted currency to _diff and then the balance of the account belonging to the credit card is reduced by _diff. Concurrently, a delivery receipt is created and returned.

Although this example is very simple, it shows the basic idea of describing a combination of services by their effects on some attributes. The nets could additionally have modeled the behavioral interface reflecting the actual message passing between the agent and the services invoked. This was left out to keep the example simple.

V. Service Description Nets as Algebraic Nets

Service Description Nets can easily be mapped to algebraic nets as defined in [8] and can thus benefit from the analysis techniques available for them. To show the basic idea, we repeated the BookBuyService in Figure III.1 again in Figure IV.2 only this time as algebraic nets with synchronous channels. The channels were only introduced to improve readability and can simply be replaced by arcs.

The basic idea is to represent each predicate by a corresponding place holding the values of the predicate. These database places hold key-value pairs for each role.

The $\perp_{undef}$ concept in the signature allows us to handle partial functions. If an attribute is not defined for a concept, we can simply declare it to be *undef* $\in \perp_{undef}$. Since $\perp_{undef}$ is a subsort of all other sorts (except $\perp$) every operator is defined on it. We will require, that all operators will yield the result *undef* if one of their parameters is *undef*.

It is often convenient to require of the signature to contain the sort of all named concepts. This offers the possibility to create a `new object` database place and to define an operator *isa* which can be useful in the guards, returning true, if a term is an instance of a a concept sort.

Figure IV.2 shows the algebraic Service Description Net for the BookBuyService from the previous section. Here, we have the database places depicted as containers. Each role or attribute has its own database place. We denoted the tuples on each database place in brackets ($[x, y]$) and we used infix notation for the arithmetic operators as it is usual. Besides the role and attribute database places there is one database place called `new Object - Repository` which is accessed if a service needs to create an object that has not been used before. It is simply a database place p_{new} of type $d(p_{new}) = (s_C, \top)$, containing the sorts and the creatable items, where s_C is the sort of all named concepts. This, however, requires that the number of concepts is known in advance. In most practical situations it should be possible to calculate an upper bound for the number of new invocations.

VI. Conclusion and Outlook

In this paper we have presented a formal framework to model processes with resources based on description logics, algebra and Petri nets. We see this as a step toward a semantic description of processes as it is needed in the Semantic Web services and agent context.

The ideas are related to other concepts such as abstract state machines [2], predicate transition nets [3], algebraic nets [8] and workflow nets [12].

Since Service Description Nets are essentially the a suptype of algebraic nets, analysing techniques for algebraic nets as e.g. discussed in [8] can be applied to Service Description Nets as well. Further investigation has to show if the special structure of Service Description Nets results in other interesting properties and analysis techniques. Furthermore it should be discussed to what extend it is sensible to forget about the algebraic structure and reduce Service Description Nets to S/T-Nets or workflow nets to study their behavioral properties.

Further investigation needs to show to what extend we can adopt planning techniques for a Petri net based planner as discussed in [10].

References

[1] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, Cambridge (Mass.), 2002.

[2] Egon Börger and Robert Stärk. *Abstract State Machines – A Method for High-Level System Design and Analysis*. Springer-Verlag, Berlin, New York, 2003.

[3] Hartmann J. Genrich. Predicate/transition nets. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247. Springer, 1986.

[4] Fausto Giunchiglia and Paolo Traverso. Planning as model checking. In Susanne Biundo and Maria Fox, editors, *ECP*, volume 1809 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 1999.

[5] Joseph A. Goguen and Grant Malcolm. A hidden agenda. Technical Report CS97-538, UCSD, 1997. WWW-Document: http://www-cse.ucsd.edu/users/goguen/ps/ha.ps.gz.

[6] Joseph A. Goguen and José Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theor. Comput. Sci.*, 105(2):217–273, 1992.

[7] Tom R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2), 1993.

[8] Wolfgang Reisig. Petri nets and algebraic specifications. In K. Jensen and G. Rozenberg, editors, *High-level Petri Nets – Theory and Application*, pages 137–170, Berlin Heidelberg, 1991. Springer-Verlag.

[9] Wolfgang Reisig. On gurevich's theorem on sequential algorithms. *Acta Informatica*, 39(5):273–305, 2003.

[10] Fabiano Silva, Marcos Alexandre Castilho, and Luis Allan Künzle. Petriplan: A new algorithm for plan generation (preliminary report). In Maria Carolina Monard and Jaime Simão Sichman, editors, *IBERAMIA-SBIA '00: Proceedings of the International Joint Conference, 7th Ibero-American Conference on AI*, pages 86–95, London, UK, 2000. Springer-Verlag.

[11] Steffen Staab and Rudi Studer, editors. *Handbook on Ontologies*. International Handbooks on Information Systems. Springer, 2004.

[12] Wil M. P. van der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques. In Jörg Desel and Andreas Oberweis, editors, *Business Process Management, Models, Techniques, and Empirical Studies*, pages 161–183. Springer-Verlag, 2000.

[13] Pepijn R. S. Visser, Dean M. Jones, Trevor J.M. Bench-Capon, and Michael J.R. Shave. Assessing heterogeneity by classifying ontology mismatches. In Nicola Guarino, editor, *Proceedings of 1st International Conference on Formal Ontologies in Information Systems, FOIS'98*, pages 148–162. IOS Press, 1998.

Appendix

a) Preliminaries on Petri Nets:

Definition 1 (*Petri Net*). A Petri net $\mathcal{N}$ is a triple $\mathcal{N} = (P, T, F)$, where P is a finite set of places, T is a finite set of transitions with $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$ is a flow relation (a set of arcs). ♢

For convenience reasons we introduce the following notions: ${}^\bullet x = \{y \mid (y, x) \in F\}$ and $x^\bullet = \{y \mid (x, y) \in F\}$ denote the set of all input and output nodes, respectively, of node $x \in P \cup T$. A place p is an *input place* for a transition t iff $p \in {}^\bullet t$ and an *output place* iff $p \in t^\bullet$. *Input* and *output transitions* are defined accordingly.

A marking of $\mathcal{N}$ is a mapping $\mathbf{m} : P \to \mathbb{N}$, that assigns a number of tokens to each place. A Petri net $\mathcal{N}$ together with an initial marking $\mathbf{m}_0$ is called a net system, denoted as $(\mathcal{N}, \mathbf{m}_0)$. A transition t is enabled in a marking $\mathbf{m}$ if $\forall p \in {}^\bullet t : \mathbf{m}(p) \geq 1$. An enabled transition t may *fire* in $\mathbf{m}$ resulting in the successor marking $\mathbf{m}'$, denoted as $\mathbf{m} \xrightarrow{t} \mathbf{m}'$, where $\mathbf{m}'(p) =_{def} \mathbf{m}(p) - |F \cap \{(p, t)\}| + |F \cap \{(t, p)\}|$.

b) Signatures, Algebras and Structures:

Definition 2 (*Sorted Algebras*). A *many-sorted signature* $\Sigma = (S, \Omega)$ is a pair consisting of a set of sort symbols S and a family $\Omega = (\Omega_{w,s})_{(w,s) \in S^* \times S}$ of operation symbols. An *order-sorted signature* is a triple $\Sigma_O = (S, \leq, \Omega)$ such that (S, Ω) is a many-sorted signature and $(S, \leq)$ is a poset such that $\omega \in \Omega_{w_1,s_1} \cap \Omega_{w_2,s_2}$ and $w_1 \leq w_2$ imply $s_1 \leq s_2$.

For a signature $\Sigma = (S, \Omega)$ a (many-sorted) Σ-algebra $\mathcal{A} = (S_A, \Omega_A)$ consists of a family of sets $S_A = (A_s)_{s \in S}$ and of operations $\Omega_A = (\omega_A)_{\omega \in \Omega}$ with $\omega_A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$ for all $\omega \in \Omega_{s_1 \ldots s_n, s}$. An (order-sorted) Σ_O-algebra $\mathcal{A} = (S_A, \Omega_A)$ for a signature $\Sigma_O = (S, \leq, \Omega)$ is a many-sorted algebra for (S, Ω) such that

- for $s_1, s_2 \in S$, $s_1 \leq s_2$ implies $A_{s_1} \subseteq A_{s_2}$, and
- $\omega \in \Omega_{w_1,s_1} \cap \Omega_{w_2,s_2}$ and $w_1 \leq w_2$ imply $\omega_A : A_{w_1} \to A_{s_1}$ equals $\omega_A : A_{w_2} \to A_{s_2}$ on A_{w_1} (a function must behave the same on the same input).

♢

For a signature Σ_O a pairwise disjoint family $X = (X_s)_{s \in S}$ with $X \cap \Omega = \emptyset$ is called Σ_O-*variables*. *Terms* are built from variables and operation symbols of the signature. The set of Σ_O-terms of sort s with variables X is denoted as $\mathbb{T}_{\Sigma,s}(X)$. The set of all terms is $\mathbb{T}_\Sigma(X) =_{def} \bigcup_{s \in S} \mathbb{T}_{\Sigma,s}(X)$. It contains the set of ground terms (terms with no variables) $\mathbb{T}_\Sigma =_{def} \mathbb{T}_\Sigma(\emptyset)$.

An *assignment* is a mapping $\alpha : X \to A$ with $\alpha(x) \in A_s$ for each $x \in X_s$, i.e. an assignment associates each variable with a value of the corresponding domain. A given assignment α extends inductively to an *evaluation* $\bar{\alpha} : \mathbb{T}_\Sigma(X) \to A$ of Σ_O terms into a Σ_O-algebra $\mathcal{A}$ in the usual way.

Given a signature Σ_O and a set of Σ_O-variables X. A triple $e = (X, l, r)$ with $l, r \in \mathbb{T}_{\Sigma,s}(X)$ for some $s \in S$ is called an *equation* of sort s w.r.t. Σ_O. The equation $e = (X, l, r)$ is called *valid* in a Σ_O-algebra $\mathcal{A}$ if for all assignments $\alpha : X \to A$ we have $\bar{\alpha}(l) = \bar{\alpha}(r)$. If e is valid in $\mathcal{A}$ we say $\mathcal{A}$ satisfies e. *Ground equations* are equations $e = (X, l, r)$ with $X = \emptyset$. In this case l and r are ground terms. If the set of variables X is obvious from the context, we will write an equation as (l, r) or $l \doteq r$ respectively.

A specification $\Gamma_O = (S, \leq, \Omega, E)$ consists of a signature $(S, \leq, \Omega)$ and a set E of Σ_O-equations. A Γ_O-Algebra is an algebra $\mathcal{A}$ of the signature Σ_O satisfying all equations in E.

Definition 3 (*Predicate Signature, Structure*). A *predicate signature* is a pair $\mathbf{PS} = (S, \leq, \Omega, E, \Pi)$ where $\Gamma_O = (S, \leq, \Omega, E)$ is an order-sorted specification and $\Pi = (\Pi_{s,s'})_{(s,s') \in (S \times S)}$ is a family of pairs called predicates, which is disjoint from S and Ω.

A *structure* (or interpretation) of a predicate signature $\mathbf{PS}$ is a triple $\mathcal{A}_{\mathbf{PS}} = (S_A, \Omega_A, \Pi_A)$ consisting of a Γ_O-Algebra $\mathcal{A} = (S_A, \Omega_A)$ and a family of relations $\Pi_A = (\pi_A)_{\pi \in \Pi}$ with $\pi_A \subseteq A_s \times A_{s'}$ for $\pi \in \Pi_{s,s'}$. An interpretation is *total* if $\mathcal{A}$ is total and $\forall \pi \in \Pi. \pi \in \Pi_{s,s'} \implies \forall a \in A_s. \exists a' \in A_{s'}. (a, a') \in \pi_A$. An interpretation is *functional* if $\forall \pi \in \Pi. (a, a') \in \pi_A \wedge (a, a'') \in \pi_A \implies a' = a''$.

The set of all interpretations for a given predicate representation $\mathbf{PS}$ is denoted as $Inter(\mathbf{PS})$. ♢

c) Service Description Net:

Definition 4 (*Service Description Net, Service Description Net system*). Let (Γ_O, Π) be a predicate representation compatible to an ontology $\mathcal{O}$. Furthermore, we require $\mathcal{O}$ to contain the concrete domain $\mathbb{B}$ of booleans as well as the concept $\perp_{undef}$, with $\perp \sqsubseteq \perp_{undef} \sqsubseteq C$ for all $C \neq \perp$.

A *Service Description Net* (respecting the ontology $\mathcal{O}$) is a tuple $\mathcal{TN} =_{def} (\mathbf{PS}, X, \mathcal{N}, d, W, G, upd, \mathbf{m}_0)$ where

- $\mathbf{PS} = (\Gamma_O, \Pi)$ is a predicate representation compatible to $\mathcal{O}$ and consisting of an order-sorted specification $\Gamma_O = (S, \leq, \Omega, E)$ and a set of predicates Π,
- X is a set of Γ_O-variables,
- $\mathcal{N} = (P, T, F)$ is a Petri net,
- $d : P \to S$ is a place typing,
- $W : F \to \mathbb{T}_{\Sigma_O}(X)$ is an arc inscription such that for $p \in P$ and $f \in F$ we have $W(f) \in \mathbb{T}_{\Sigma_O,s}(X \setminus Bound(upd(t)))$ with $d(p) \leq s$ if $f = (p, t)$ and $W(f) \in \mathbb{T}_{\Sigma_O,s}(X)$ with $s \leq d(p)$ if $f = (t, p)$,
- $G : T \to \mathbb{T}_{\Sigma_O,\mathrm{Bool}}(X)$ assigns an enabling condition to each transition, and
- $upd : T \to Update(X)$ assigns a regular and consistent set of assignments and updates to each transition.

A Service Description Net together with an initial state $Q_0 = (\mathbf{m}_0, \mathcal{I}^0_{\mathbf{PS}})$ consisting of an initial marking $\mathbf{m}_0 : P \to \mathbb{T}_{\Sigma_O}$ such that $\mathbf{m}_0(p) \in \mathbb{T}_{\Sigma_O,s}, s \leq d(p)$ for $p \in P$ and an initial interpretation $\mathcal{I}^0_{\mathbf{PS}} = (\mathcal{A}, \Pi_{\mathbf{PS}})$ of $\mathbf{PS} = (\Gamma_O, \Pi)$ is called a *Service Description Net system.* ♢

To guarantee the executability of Service Description Nets, we require that a variable used on an output arc of a transition is either also used on an input arc or defined through an assignment, thus reducing the search space for possible bindings. Formally, let $\textsc{Var}^-(t) =_{def} \bigcup_{p \in {}^\bullet t} \textsc{Var}(W(p, t))$ and $\textsc{Var}^+(t) =_{def} \bigcup_{p \in t^\bullet} \textsc{Var}(W(t, p))$ be the set of variables on the input and output arcs respectively. We then require that $W(t, p) \in \mathbb{T}^{\Sigma_O}_{d(p)}(\textsc{Var}^-(t) \cup Bound(upd(t)))$, where $Bound$ is extended to the set of update actions assigned to t.

Are Visual Methods Mandatory for the Modeling of Business Processes?

Carlo Simon
Institute for Management, University Koblenz-Landau
PF 201 602, 56016 Koblenz, Germany
simon@uni-koblenz.de

I. Abstract

Perhaps unusual for a Petri net workshop, this paper addresses problems of graphical methods for business process modeling that could better be solved with a formal textual notation. It therefore introduces a formal process language for which, nonetheless, the semantic is defined in terms of Petri nets. It is proposed to bring a new dimension into the discussion between those using Petri nets and related methods and those using process algebraic methods for business process modeling by presenting a solution that lies between these existing approaches.

II. Introduction

In the past years one could observe that with the development of web services some authors of the process algebraic community (especially those proposing the pi-calculus) claim their approach as a formal base for business process modeling [2]. In fact, the formalization of business processes has been conducted recently with Petri nets [1] or methods that at least base on Petri net concepts [16, pp. 12-13]. It is therefore not surprising that a discussion starts comparing established Petri net techniques with those approaches that now claim to be a better choice for formalization. In his paper [2], van der Aalst turns the discussion from exchanging abstract arguments into formulating concrete challenges (perhaps they could also be called exercises) which from his point can easily be solved with Petri nets (especially Workflow nets) but which are purely solved or probably cannot be solved with the pi-calculus approach. One of these challenges is the non-sequential execution of two sequential processes b before c before d and e before f before g where the concurrency is started by an action a and is synchronized by an action h. In order to increase the complexity of his example, he demands that for the in principle independent processes the occurrence of c before f must be guaranteed. A Workflow net model of this structure is shown in Figure 1.

Although Aalst's arguments against the pi-calculus as a formal means for business process modeling (he explicitly emphasizes the value of process algebras in general) are right, one has to state that

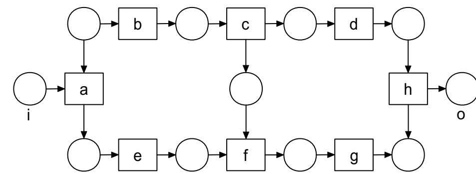


Fig. 1. Workflow net model taken from [2, p. 5]

- the Workflow net model of the described simple situation is quite space consuming, and,
- even worse, the Workflow net model does not contain any justification for the specific process structure. Especially the reason why c must occur before f, independently from the question whether it is really necessary or an artificial constrain, is not documented.

This paper aims to find a compromise between the text-oriented process algebraic methods and the Workflow net approach by introducing a language that allows both the textual specification of processes and their visualization as Petri nets. It goes back onto prior work on a Logic of Actions [6], [7], [5], [15], [18]. In opposite to this prior work, the semantic of the process language is not defined immediately as a half-order on the actions the processes are built upon. Instead of this, implementation rules are defined which canonically allow translating the words of the language into Module nets - a variation of Workflow nets - and then the Module nets' processes define the semantics of the process language.

The paper is organized as follows: in its second section, it introduces the so called Semantic Process Language (SPL) to specify process sets and how to define the semantics of its words via Module nets. Its application is shortly demonstrated at the example of Figure 1. Afterwards, a business process is chosen from the literature to show the complexity of graphical methods when the intended behavior is described precisely. Alternatively to this, the same example is modeled with SPL which allows a stepwise approach and the documentation of justifications for the specific process structure. The paper closes with a conclusion.

III. SEMANTIC PROCESS LANGUAGE

As mentioned in the introduction already, the Semantic Process Language (SPL) goes back onto earlier work on a Logic of Actions (LoA). The semantic of LoA modules - the formulas of this logic - is defined by sets of sequential processes over elementary actions which occur or are forbidden. In [5], [18], these sets are defined as a half-order over the actions, however, beyond the definition these explicitly given process sets are neither used for verification purposes nor can be presented in real-world applications to users because of the sheer complexity and amount of processes defined even by simple examples. They are only used to legitimate an alternative representation of modules as Module nets - a variation of Workflow nets.

Modules are either elementary specifying the occurrence of prohibition of an action ($[a]$ or $[\overline{a}]$). With respect to two modules M_1 and M_2, non-elementary modules are built for sequences ($[M_1 \olessthan M_2]$ and $[M_1 \ogreaterthan M_2]$), exclusive alternative and alternative ($[M_1 \oplus M_2]$ and $[M_1 \ovee M_2]$), concurrent behavior which is, however, joined over shared actions ($[M_1 \owedge M_2]$), iterations ($[M_1^*]$, $[M_1^+]$ and $[M_1^n]$), and complement building ($\overline{[M_1]}$). Moreover, the coincident occurrence of actions or their prohibition in a single synchronized step is defined with a further operand ($[M_1 \circledast M_2]$).

Modules can be implemented with Module nets. A Module net has an explicit *start* transition with empty preset and an explicit *goal* transition with empty postset. Processes of Module nets are all firing sequences which reproduce the empty initial marking by firing *start* and *goal* transition exactly once.

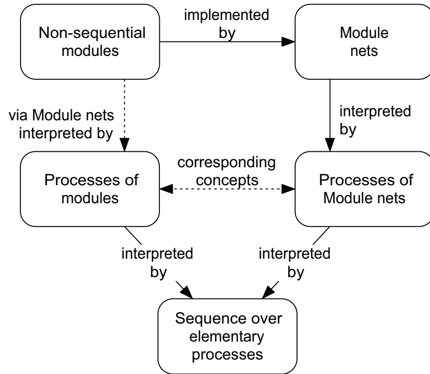


Fig. 2. Relationship between the concepts

For SPL, in opposite to the LoA approach, the process set of a module (i.e. in SPL non-sequential processes are defined with the same operands as in LoA) is not explicitly defined but rather for elementary modules also elementary Module net implementations are predefined. For non-elementary modules, canonical implementation rules take the operands' Module net implementations as input and compose complex Module nets upon them. The semantic of SPL modules is then defined by the sequential processes of their implementations. Figure 2 shows in a diagram how these concepts are related to each other.

In this paper the canonic building rules are not discussed but also introduced by examples. The first one is that of Figure 1.

In terms of modules, the behavior of the net in Figure 1 is described by

$$
\begin{aligned}
M \quad := \quad & (a \olessthan (\\
& \quad (b \olessthan c \olessthan d) \owedge (e \olessthan f \olessthan g) \\
&) \olessthan h) \owedge \\
& (c \olessthan f)
\end{aligned}
$$

The Module net implementation of M shown in Figure 3 is nearly the Workflow net of Figure 1 except the additional *start* and *goal* transitions. The advantage of the representation as Module nets is that here processes are covered by T-invariants [14]. Dehnert has discussed the close relationship between Module nets and Workflow nets in [4].

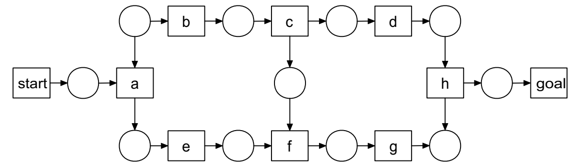


Fig. 3. Module net implementation of M

Alternatively to the definition of M in a single step, one could also break down the definition into smaller pieces which allows to add reasons for the specific process structure. A possible alternative definition could be the following:

M_1	$:=$	$b \olessthan c \olessthan d$	*first subprocess*
M_2	$:=$	$e \olessthan f \olessthan g$	*second subprocess*
M_3	$:=$	$a \olessthan (M_1 \owedge M_2) \olessthan h$	*integrate processes*
M_4	$:=$	$c \olessthan f$	*c before f*
M	$:=$	$M_3 \owedge M_4$	

More examples on how to build Module nets from modules will be given in the next section. In this section, two more differences between SPL and LoA will be pointed out:

The second significant difference is the more important role of actions for the SPL approach. While in LoA actions are considered solely as bricks the processes are built upon, the possibility to model complex (business) processes with the SPL approach requires a new comprehension of this term. Now distributed business processes and eventually distributively developed models can be merged in shared actions, i.e. actions that occur in more than one business process. Then, however, the semantic of these actions must be defined and documented

in a central glossary (or possibly ontology) to enable the merging operation. Moreover, in terms of Petri nets one is able to specify the semantics of actions concerning their manipulating effect on information objects with the aid of high-level Petri nets [9], [8], [11]. This semantic enrichment of actions in two senses justifies that the language is called *semantic* process language.

The third difference between SPL and LoA is on a basic level. While in LoA the prohibition of an action avoids its occurrence within a process as whole, in SPL the prohibition only restricts possible concurrent occurrences. This less restrictive definition has turned out to be closer to real-world requirements.

IV. An Exemplary Business Process

In order to demonstrate the complexity of precise business process models, a partial business process modeled in [19, pp. 67-68] as an event-driven process chain (EPC) and shown in Figure 4 is discussed. EPCs [10], [12], [17] are diagrams for the visualization of business processes. Their concepts are not explained here but it is referred to the standard literature on this topic.

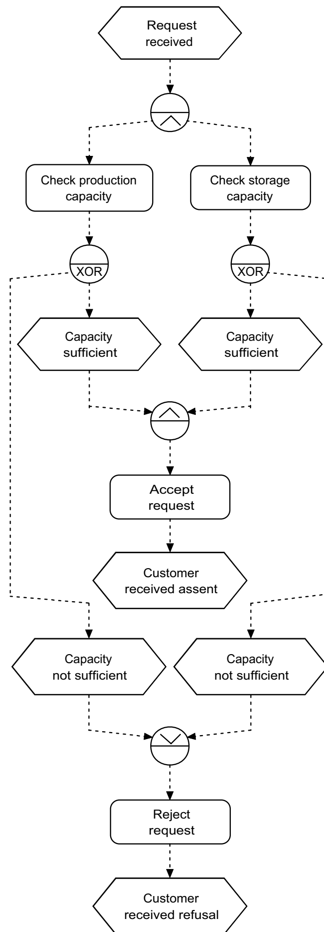


Fig. 4. EPC model of the exemplary business process

EPCs typically begin and end in events indicating the beginning and ending of the respective processes. The processes described in the example of Figure 4 is that of a company which manufactures special machines on customers' demands. For each received request, the company has to decide whether its production and storage capacity is sufficient to fulfil the request. If this is true, the request is accepted and the customer receives an acknowledgement, otherwise, the request must be rejected and the customer receives a refusal.

In the model, two different forms of synchronization can be found: synchronization via an and-connector in case that production and storage capacity are sufficient and synchronization via an or-connector in the rejection case. In the second, three different failure situations are described - two single failures and a double failure. Hereby, the single failure situations are of special interest: assuming that the storage capacity is recognized as sufficient but the production capacity not. A successful check for storage capacity, however, will cause a reservation of the capacity in the company's information system. If now the request is rejected due to the insufficient production capacity, there must be a rollback concerning this reservation which is not explicitly modeled in the EPC diagram. The second single failure case bears the same type of problem.

Beside this, there is a second imprecision in the single failure case: if storage capacity is recognized to be insufficient, is there still the need to check for the production capacity or not? The EPC model does not give the answer.

Unfortunately, this situation does not become clearer if a state semantic is added as proposed by Kindler [13], [3], because one can only conclude on aspects like rollback actions if they are explicitly enclosed in the EPC model.

V. Making the Business Process more Precise

In this section, the EPC model of the previous section is used as a requirements model form which a precise model in terms of modules and Module nets is derived. For this, in addition to the module definition of Section III, a module extension mechanism is introduced which allows to incrementally adding exceptional behavior to an initially defined norm behavior.

As the norm behavior of the business process for checking customer requests the case is chosen that both resources - storage and production - are sufficiently given. In terms of modules this is described by

$$\begin{aligned} \textit{Check} \quad := \quad & \textit{Request received} \olessthan (\\ & \quad \textit{Check prod. cap.} \owedge \textit{Check stor. cap.} \\ &) \olessthan \textit{Accept request} \end{aligned}$$

Figure 5 shows the respective Module net implemen-

tation. Within this module and its implementation, the *check* actions (or *check* transitions, respectively) do not only conduct the test for the storage or production resources but also write the respective reservations into the information system.

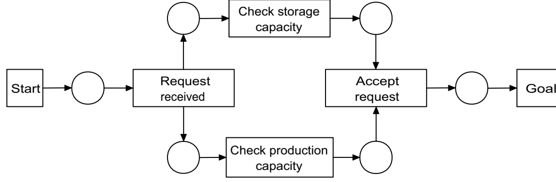


Fig. 5. Module net of the norm behavior

To both check functions alternative actions are needed to recognize that the respective resource - storage or production - is not sufficiently given. For indicating an extension of an already given module in contrast to the definition of a new one, $\cdot\!=$ is used as symbol. The first type of extension is one which starts from a specific process state and also ends in a specific state. For this, the process states are specified with respect to the module's action: for some action a, $a\bullet$ indicates the state reached immediately after a has been conducted and $\bullet a$ indicates the state immediately before a can be conducted. Moreover *start*• indicates the state immediately after process start and •*goal* the state before process end. The recognition that storage and production capacity are not sufficiently given is then specified by

$$\begin{aligned} Check \;\; \cdot\!= \;\; & \bullet Check\ prod.\ cap. \oslash (\\ & \quad Prod.\ insufficient \oslash Reject\ prod. \\ &) \oslash \bullet goal \end{aligned}$$

and

$$\begin{aligned} Check \;\; \cdot\!= \;\; & \bullet Check\ stor.\ cap. \oslash (\\ & \quad Stor.\ insufficient \oslash Reject\ stor. \\ &) \oslash \bullet goal \end{aligned}$$

Figure 6 shows the implementation of these extensions. Unfortunately, none of the added actions belongs to a legal process, because in the single failure case that part of the net where the check could have been conducted successfully remains marked. And in the double failure case, the empty initial marking can only be reproduced if the *goal* transition fires twice.

What is needed, moreover, is a second extension which synchronizes the concurrent executions in the single and the double failure cases. For the single failure case this is achieved by

$$Check \;\; \cdot\!= \;\; \bullet Check\ prod.\ cap. \oslash Reject\ stor.$$

which disables a check for the production capacity if the storage capacity is insufficiently given and

$$Check \;\; \cdot\!= \;\; \bullet Check\ stor.\ cap. \oslash Reject\ prod.$$

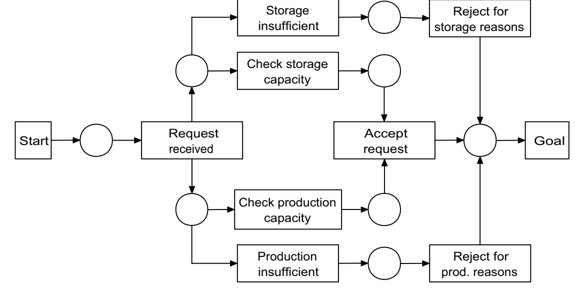


Fig. 6. Module net with failure recognition

which disables a check for the storage capacity if the production capacity is insufficiently given. The double failure situation is also described by two extensions, namely

$$\begin{aligned} Check \;\; \cdot\!= \;\; & \bullet Reject\ prod. \oslash (\\ & Reject\ double\ failure \\ &) \oslash \bullet goal \end{aligned}$$

$$\begin{aligned} Check \;\; \cdot\!= \;\; & \bullet Reject\ stor. \oslash \\ & Reject\ double\ failure \end{aligned}$$

Figure 7 shows the precise behavior in all three possible failure situations.

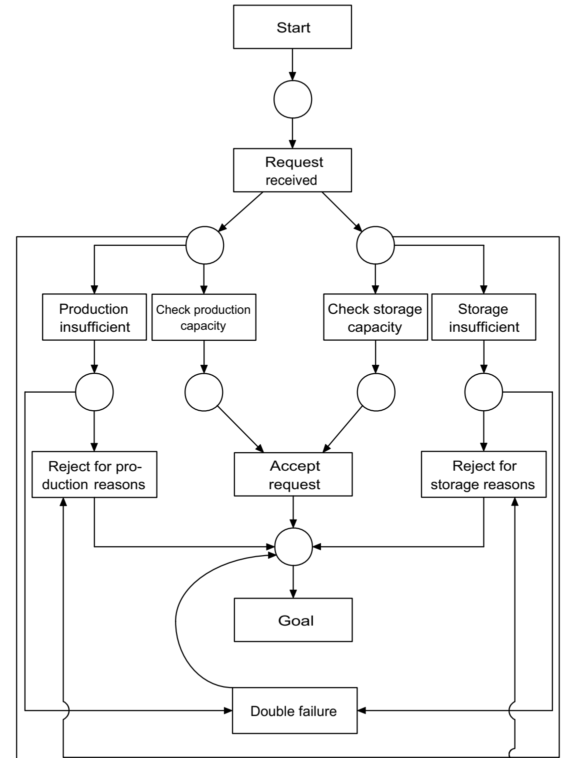


Fig. 7. Module net with process termination in case of insufficient capacities

Still the model has not reached the required degree of precision. If a failure concerning one resource type is recognized after the other resource has been reserved already, then in the current model the process would get stuck. So, the final extension that has to be added are rollbacks of successful checks for storage and production capacities. They can simply be added with the aid of the following extensions:

$$\begin{aligned} \mathit{Check} \quad &\cdot= \quad \mathit{Check\ prod.\ cap.} \bullet \varolessthan (\\ & \qquad \mathit{Rollback\ prod.\ cap.} \\ & \qquad) \varolessthan \bullet \mathit{Check\ prod.\ cap.} \end{aligned}$$

$$\begin{aligned} \mathit{Check} \quad &\cdot= \quad \mathit{Check\ stor.\ cap.} \bullet \varolessthan (\\ & \qquad \mathit{Rollback\ stor.\ cap.} \\ & \qquad) \varolessthan \bullet \mathit{Check\ stor.\ cap.} \end{aligned}$$

The final Module net of the overall behavior is shown in Figure 8. It has reached a significant degree of complexity which is the price for the high degree of precision. If confronted with the final Module net, a contemplator would clearly have difficulties in understanding all details at once - and this although the chosen situation is very simple.

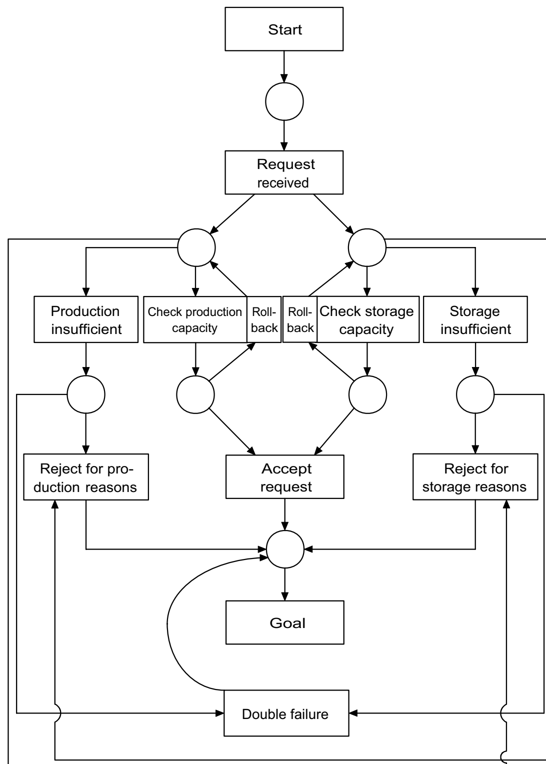


Fig. 8. Module net with process termination in case of insufficient capacities

What precisely makes the comprehension so difficult? To answer this question, two reasons must be mentioned:

1. A contemplator confronted with the entire graphical model must understand all decisions the modelers have incrementally made. In opposite to this, the approach here allows incrementally showing all evolutionary steps from the initial module of the norm behavior to the finalized model.
2. The second reason is related to the first one. For each element (i.e. transition, place, and arc) of the final model there exists justification for its existence. If all these justifications would be added in the graphical model with the aid of memos (which is quite popular in several modeling tools), the readability of the graph would be decreased further. Adding the justification to the textual specification instead is much simpler and easier to reproduce.

VI. Conclusion

In this paper a formal process specification language has been introduced where the semantics of its words is defined via their Module net implementation. Due to this, a graphical visualization of the specified processes is immediately given - a significant advantage over process algebras.

As an application, the specification of business processes has been chosen. Here the focus was on the development of precise models which, if modeled solely as graphs, are hardly readable due to the amount of exceptional behavior. As a reason for this, besides the complexity and space intensive representation of graphs, also a missing concept for the representation of justifications has been identified. They are hardly integrated and documented within graphical workflow models. Moreover, the presented approach allows a better reproduction of the modeling process for the contemplator ex post.

The reasons why the language is called a *semantic* process language has been explained in Section III and is related to the emphasized role of actions. In order to use the language not only for the definition of isolated business processes as discussed in this paper but also for mergers and for identifying possible cooperation partners, this semantic enrichment must be achieved through the discussed categorization of the actions and their specification in terms of high-level Petri nets. This, obviously, requires a tool support which is currently under development.

REFERENCES

[1] W. M. P. Aalst, van der. Structural Characterizations of Sound Workflow Nets. Computing Science Reports 96/23, Eindhoven University of Technology, 1996.

[2] W. M. P. Aalst, van der. Pi calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "Pi hype". *BPTrends*, 3(5):1–11, 2005.

[3] N. Cuntz and E. Kindler. On the semantics of EPCs: Efficient calculation and simulation. In M. Nüttgens and F. J. Rump, editors, *EPK 2004: Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, Proceedings*, pages 7–26, Luxembourgh, 2004.

[4] J. Dehnert. *A Methodology for Workflow Modeling - From business process modeling towards sound workflow specification*. PhD thesis, TU Berlin, 2003.

[5] M. Fidelak. *Integritätsbedingungen in Petri-Netzen*. PhD thesis, Universität Koblenz-Landau, 1993.

[6] H. J. Genrich. Formale Eigenschaften des Entscheidens und Handelns. Interner Bericht 09/73-11-29, GMD, St. Augustin, 1973.

[7] H. J. Genrich. Ein Kalkül des Planes und Handelns. In *Ansätze zur Organsiationstheorie rechnergestützter Informationssysteme*, GMD Bericht 111, pages 77–92. Oldenbourg Verlag, 1978.

[8] H. J. Genrich. Predicate/Transition Nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and their Properties, Advances in Petri Nets 1986, Part I*, Lecture Notes in Computer Science 254. Springer, 1987.

[9] H. J. Genrich and K. Lautenbach. System Modelling with High-Level Petri Nets. *Theoretical Computer Science*, 13, 1981.

[10] W. Hoffmann, J. Kirsch, and A.-W. Scheer. Modellierung mit Ereignisgesteuerten Prozeßketten, Methodenhandbuch. Technical report, Universität des Saarlandes, Institut für Wirtschaftsinformatik, Saarbrücken, 1993.

[11] K. Jensen. *Coloured Petri-Nets*. Band 1. Springer Verlag, Berlin, 1992.

[12] G. Keller, M. Nüttgens, and A.-W. Scheer. Semantische Prozeßmodellierung auf der Basis Ereignisgesteuerter Prozeßketten (EPK). Technical Report 89, Universität des Saarlandes, Institut für Wirtschaftsinformatik, Saarbrücken, 1992.

[13] E. Kindler. On the Semantics of EPCs: A Framework for Resolving the Vicious Circle. In J. Desel, B. Pernici, and M. Weske, editors, *Business Process Management (BPM 2004)*, volume 3080 of *Lecture Notes in Computer Science (LNCS)*, pages 82–97, Potsdam, Germany, 2004. Springer.

[14] K. Lautenbach. Linear algebraic techniques for place/transition nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and their Properties, Advances in Petri Nets 1986, Part I*, Lecture Notes in Computer Science 254, pages 142–167. Springer, 1987.

[15] K. Lautenbach and C. Simon. Verification in a Logic of Actions. In *7. Workshop Algorithmen und Werkzeuge für Petrinetze*, Koblenz, 2000.

[16] A.-W. Scheer. Modellunterstützung für das kosten-orientierte Geschüfsprozessmanagement. In C. Berkau and P. Hirschmann, editors, *Kostenorientiertes Geschäftsprozessmanagement*, pages 3–25. Vahlen, München, 1996.

[17] A.-W. Scheer. *ARIS - Business Process Frameworks*. Springer-Verlag, Berlin, 3rd edition, 1999.

[18] C. Simon. *A Logic of Actions and Its Application to the Development of Programmable Controllers*. PhD thesis, Universität Koblenz-Landau, 2001.

[19] J. L. Staud. *Geschäftsprozessanalyse*. Springer, Berlin, 2001.