Validierung syntaktischer und anderer EPK-Eigenschaften mit PROLOG

Volker Gruhn, Ralf Laue
{gruhn,laue}@ebus.informatik.uni-leipzig.de
Lehrstuhl für Angewandte Telematik und E-Business*
Universität Leipzig, Fakultät für Informatik

Abstract: Die XML-basierte Austauschsprache EPML[MN04] für Ereignisgesteuerte Prozessketten wurde entwickelt, um eine Möglichkeit des Datenaustausches zwischen Modellierungswerkzeugen zu schaffen. Wird ein EPML-Modell aus einer fremden Quelle von einem Modellierungswerkzeug importiert, sollte zunächst geprüft werden, ob die importierte XML-Datei ein syntaktisch korrektes EPK-Modell darstellt. Hierzu müssen neben den syntaktischen Forderungen, die sich aus dem XML-Schema der Sprache EPML ergeben, weitere Eigenschaften getestet werden. Diese Eigenschaften (beispielsweise die Forderung, dass sich Ereignisse und Funktionen im EPK-Kontrollfluss abwechseln) sind in [Rum99], [Kel99] und [NR02] formalisiert. Mendling und Nüttgens zeigten in [MN03b], wie der überwiegende Teil der in [NR02] formalisierten Anforderungen mit Hilfe der Sprache Schematron validiert werden kann. Neben der Tatsache, dass zwei der Anforderungen aus [NR02] auf diese Weise nicht überprüft werden können, hat der Ansatz aus [MN03b] einen weiteren Nachteil: Er setzt voraus, dass das EPML-Modell zusätzliche Attribute zum Typ der Modellelemente enthält. Dies schafft jedoch Redundanz im EPML-Modell und vergrößert unnötig Größe und Komplexität der EPML-Austauschdateien. In unserem Beitrag zeigen wir, wie alle in [Rum99], [Kel99] und [NR02] beschriebenen Eigenschaften recht einfach mit Hilfe der Sprache PROLOG überprüft werden können, ohne die erwähnten zusätzlichen Attribute zu benötigen. Damit wird eine 100%-ige Überprüfung der Eigenschaften bei gleichzeitiger Reduzierung der Komplexität des Austauschformats erreicht. Wir geben ferner Beispiele für weitere Eigenschaften an, die mit unserem Ansatz effizient überprüft werden können.

1 Einführung

Ereignisgesteuerte Prozessketten (EPK) sind eine verbreitete Sprache zur Modellierung von Geschäftsprozessen. Obwohl diese Sprache von verschiedenen Werkzeugen unterstützt wird, gestaltet sich der Austausch von Modellen zwischen den Werkzeugen oft schwierig, da die Modelle in proprietären Dateiformaten gespeichert werden.

Diese Tatsache veranlasste Mendling und Nüttgens, EPML als XML-basierte Austauschsprache für EPK-Modelle vorzuschlagen[MN04]. Dies ermöglicht den Austausch von Mo-

*Der Lehrstuhl für Angewandte Telematik und E-Business ist ein Stiftungslehrstuhl der Deutschen Telekom AG

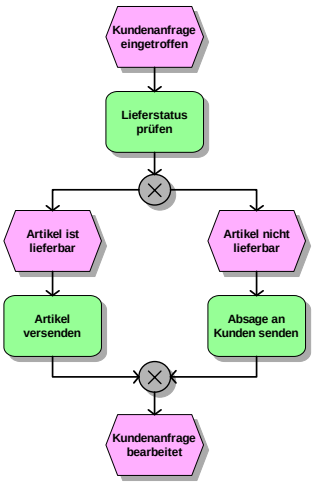

Abbildung 1: Eine Beispiel-EPK

dellen zwischen verschiedenen Modellierungs-, Simulations-, Monitoring- und anderen Werkzeugen.

Bevor ein Werkzeug eine EPML-Datei aus einer fremden Quelle importiert, sollte es sicherstellen, dass es sich bei der zu importierenden Datei tatsächlich um ein syntaktisch korrektes EPK-Modell handelt.

Unser Beitrag befasst sich mit dieser Prüfung von EPML-Modellen auf syntaktische Korrektheit.

Nachdem im Abschnitt 2 kurz die Sprache EPML skizziert wird, werden im Abschnitt 3 die Anforderungen, die an ein korrektes EPK-Modell zu stellen sind, besprochen. In Abschnitt 4 wird die in [MN03b] veröffentlichte Lösung für die Überprüfung dieser Regeln diskutiert. Im Abschnitt 5 präsentieren wir einen alternativen Ansatz und diskutieren in 6 dessen Vorteile gegenüber bisherigen Ansätzen. Schließlich zeigen wir in Abschnitt 7, dass das von uns vorgeschlagene Verfahren auch zum Testen anderer interessanter Eigenschaften genutzt werden kann.

2 Die EPC Markup Language (EPML)

Im Folgenden wird eine kurze skizzenhafte Einführung in die Sprache EPML gegeben. Diese ist bei weitem nicht vollständig, reicht aber zum Verständnis der in diesem Beitrag beschriebenen Verfahren aus. Für eine vollständige Einführung in EPML verweisen wir auf [MN04] sowie auf die Website `www.epml.de`.

Das unten gezeigte EPML-Fragment beschreibt den in Abb. 1 dargestellte EPK. Jeder Funktion der EPK entspricht ein function-Element, jedem Ereignis ein event-Element und

jedem Konnektor ein AND-, OR- oder XOR-Element. Jedes der genannten Elemente hat ein Attribut namens id, durch das es eindeutig identifiziert ist. Der Kontrollfluss wird durch arc-Elemente codiert. In unserem Beispiel bezeichnet das arc-Element mit dem Attribut id=12 den Kontrollfluss-Pfeil zwischen dem Element mit id=1 und dem Element mit id=2, also zwischen dem Ereignis „Kundenanfrage eingetroffen“ und der Funktion „Lieferstatus prüfen“.

```
<?xml version="1.0" encoding="UTF-8"?>
<epml:epml xmlns:epml="http://www.epml.de"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="epml_1_draft.xsd">
  <epc EpcId="1" Name="EPC">
    <event id="1">
    <name>Kundenanfrage eingetroffen</name>
    </event>
    <function id="2">
      <name>Lieferstatus prüfen</name>
    </function>
    <xor id="4">
      <name/>
    </xor>
    <event id="5">
      <name>Artikel ist lieferbar</name>
    </event>
    <event id="6">
      <name>Artikel nicht lieferbar</name>
    </event>
    <function id="7">
      <name>Artikel versenden</name>
    </function>
    <function id="8">
      <name>Absage an Kunden senden</name>
    </function>
    <xor id="9">
      <name/>
    </xor>
    <event id="10">
      <name>Kundenanfrage bearbeitet</name>
    </event>
    <arc id="11">
      <flow source="1" target="2"/>
    </arc>
... sowie je ein weiteres <arc>-Element für jeden Pfeil im Modell
  </epc>
</epml:epml>
```

3 Syntaktische Anforderungen an EPKs

Eine ausführliche formale Diskussion der Syntax von EPKs findet sich in [Rum99], [Kel99] und [NR02]. Im Unterschied zu diesen Arbeiten betrachten wir hier zur Vereinfachung nur EPKs ohne Prozesswegweiser und hierarchische Funktionen.

Der Grund hierfür liegt darin, dass wir beabsichtigen, die in Abschnitt 5 gezeigten Syntaxprüfungen in das Werkzeug EPCTools[Cun04] zu integrieren, das Prozesswegweiser und hierarchische Funktionen nicht darstellen kann. Grundsätzlich sollten aber mit dem vorgestellten Ansatz auch die syntaktischen Eigenschaften von EPKs mit Prozesswegweisern und hierarchischen Funktionen problemlos überprüft werden können.

Ausgehend von [Rum99], [Kel99] und [NR02] nennen wir im Folgenden die Eigenschaften, die ein Graph G erfüllen muss, damit G eine syntaktisch korrekte EPK ist. Dabei bezeichnen wir mit K die Knotenmenge und mit $E \subseteq K \times K$ die Kantenmenge von G. Die Knotenmenge V ist die Vereinigung der folgenden fünf paarweise disjunkten Mengen:

- F (Menge der Funktionen)
- E (Menge der Ereignisse)
- V_{xor} (Menge der XOR-Verknüpfungsoperatoren)
- V_{or} (Menge der OR-Verknüpfungsoperatoren)
- V_{and} (Menge der AND-Verknüpfungsoperatoren)

.

Für eine syntaktisch korrekte EPK müssen die folgenden Eigenschaften erfüllt sein:

1. Der Graph G ist gerichtet.
2. Der Graph G ist zusammenhängend.
3. Der Graph G ist endlich.
4. Enthält der Graph G eine Kante von a nach b, so gibt es in G keine Kante von b nach a und keine weitere Kante von a nach b
5. Die Mengen E und F sind nicht leer, d.h. es gibt mindestens ein Ereignis und mindestens eine Funktion.
6. Funktionen besitzen genau eine eingehende und genau eine ausgehende Kante.
7. Ereignisse besitzen genau eine eingehende oder genau eine ausgehende Kante. (Besitzt ein Ereignis genau eine eingehende und keine ausgehende Kante, nennen wir es Endereignis. Besitzt ein Ereignis keine eingehende und genau eine ausgehende Kante, nennen wir es Startereignis.)
8. Verknüpfungsoperatoren (also Knoten, die zur Menge $V_{xor} \cup V_{or} \cup V_{and}$ gehören) haben entweder genau eine eingehende und mehr als eine ausgehende Kante (dann heißen sie Split) oder mehr als eine eingehende und genau eine ausgehenden Kante (dann heißen sie Join).
9. Der Graph G enthält keinen gerichteten Zyklus, der nur aus Verknüpfungsoperatoren (also Knoten, die zur Menge $V_{xor} \cup V_{or} \cup V_{and}$ gehören) besteht.

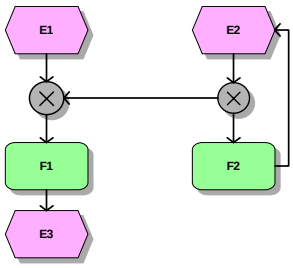

Abbildung 2: EPK, für die nur Eigenschaft 14 verletzt ist

10. Auf ein Ereignis folgen im Graphen immer n Verknüpfungsoperatoren (n=0,1,...), direkt gefolgt von einer Funktion.
11. Auf eine Funktion folgen im Graphen immer n Verknüpfungsoperatoren (n=0,1,...), direkt gefolgt von einem Ereignis.
12. Auf ein Ereignis folgen im Graphen nie n AND-Konnektoren (n=0,1,...), direkt gefolgt von einem Split aus der Menge $V_{xor} \cup V_{or}$.
13. Es gibt mindestens ein Start- und mindestens ein Endereignis.
14. Für jeden Knoten im Graphen gibt es einen gerichteten Pfad von einem Startereignis zu diesem Knoten.
15. Für jeden Knoten im Graphen gibt es einen gerichteten Pfad von diesem Knoten zu einem Endereignis.

Die Eigenschaften 1 bis 15 sind im Wesentlichen [Rum99], [Kel99] und [NR02] entnommen. Eigenschaft 3 wird in keiner der angegebenen Arbeiten ausdrücklich genannt, aber stillschweigend vorausgesetzt. Die Eigenschaften 14 und 15 werden in [Kel99], nicht aber in [Rum99] und [NR02] erwähnt. Da diese Eigenschaft keineswegs, wie man vielleicht vermuten könnte, aus den anderen Eigenschaften folgt (siehe Gegenbeispiel in Abb. 2), ist es aber sinnvoll, diese Forderung zu stellen. Auf die in [MN03b] genannte Forderung, dass der Graph keine reflexive Kante enthält (also eine Kante, die einen Knoten mit sich selbst verbindet) verzichten wir, da sich diese Forderung als Folgerung aus den Forderungen 9, 10 und 11 ergibt.

4 Bekannte Lösungsansätze

Mendling und Nüttgens untersuchten in [MN03b], wie gut sich XML-Schemasprachen dazu eignen, die im vorigen Abschnitt genannten Syntaxforderungen zu validieren. Es wurde gezeigt, dass mit XML Schema [Wor01a, Wor01b] sowie Relax NG[CM01] nur die einfachsten Eigenschaften (Eigenschaften 1, 3, 4 und 5) validieren lassen. Daher sind diese Schemasprachen nicht geeignet, um die syntaktische Korrektheit von EPML-Dateien

zu überprüfen. Wesentlich bessere Resultate wurden in [MN03b] bei Nutzung der Sprache Schematron[Jel02] erzielt. Schematron gestattet es, erwünschte Eigenschaften eines XML-Dokuments unter Nutzung von XPath[Wor99] zu beschreiben. Diese Abfragesprache erlaubt komplexe Operationen, z.B. logische Verknüpfungen. Daher kann, wie in [MN03b] gezeigt, Schematron benutzt werden, um den größten Teil der Forderungen für syntaktisch korrekte EPML-Dateien zu validieren. Allerdings können auch durch den in [MN03b] präsentierten Ansatz nicht alle Eigenschaften validiert werden: Die Eigenschaften 2, 9, 14 und 15 können nicht mit Schematron geprüft werden.

Um Schematron wie in [MN03b] vorgeschlagen zur Syntaxvalidierung nutzen zu können, hat man allerdings einen gewissen Preis zu zahlen: Zu den Elementen der EPML-Datei müssen zusätzliche Attribute notiert werden, die Aussagen über die Knoten der EPK treffen. Beispielsweise wird das Endereignis der in Abb. 1 dargestellten EPK statt

```
<event id="10">
  <name>Kundenanfrage bearbeitet</name>
</event>
<arc id="11">
  <flow source="1" target="2"/>
</arc>
```

jetzt notiert:

```
<event id="10" type="EventEnd">
  <name>Kundenanfrage bearbeitet</name>
</event>
<arc id="11" type="EventFunctionArc">
  <flow source="1" target="2"/>
</arc>
```

Damit wird angegeben, dass es sich beim Ereignis „Kundenanfrage bearbeitet“ um ein Endereignis handelt und die gerichtete Kante zwischen dem Startereignis (mit der id=1) und der Funktion „Lieferstatus prüfen“ (mit der id=2) den Typ „Ereignis-Funktion-Kante“ hat[MN03a]. Die zusätzlichen Attribute sind im XML Schema der Sprache EPML seit der Anfangsversion 1.0 vorhanden. Ihre Nutzung hat allerdings zwei Nachteile: Zum einen werden die EPML-Dokumente größer und komplexer. Zum zweiten sind die Informationen, die aus diesen Attributen gewonnen werden können, ausnahmslos *redundant*. Die kurze Version der EPML-Datei (ohne zusätzliche Attribute) enthält nämlich schon alle Informationen, um beispielsweise festzustellen, dass es sich beim Ereignis „Kundenanfrage bearbeitet“ um ein Endereignis handelt. Da eine Austauschsprache möglichst einfach und ohne redundante Informationen sein sollte, halten wir einen Verzicht auf die zusätzlichen Attribute für wünschenswert. Im kommenden Absatz werden wir ein Validierungsverfahren vorstellen, das ohne die zusätzlichen „type“-Attribute arbeitet.

Neben den bereits genannten Lösungsansätzen, die bereits in [MN03b] auf ihre Eignung zur Validierung von EPML-Dateien bewertet wurden, ist die Constraint Language in XML (CLiXML)[DJ06] betrachtenswert. Diese Sprache wurde mit dem Ziel entwickelt, komplexere Validierungen von XML-Dateien durchführen zu können. Die Validierung der

meisten Regeln aus Abschnitt 3 sollte mit CLiXML leicht möglich sein. Allerdings führt [DJ06] gerade XML-Darstellungen von gerichteten Graphen, bei denen Knoten und Kanten analog zu EPML kodiert werden, als Beispiel an, bei dem Validierungen mit CLiXML deutlich erschwert werden.

5 Eigenschaftsüberprüfung mit PROLOG

Unser Ansatz zur Validierung der Eigenschaften nutzt das Programmierprinzip der logischen Programmierung, um Eigenschaften von EPML-Dateien zu validieren. Die Grundidee besteht darin, ein EPML-Dokument in eine Menge logischer Aussagen (z.B. „Es gibt eine Kante vom Knoten 1 zum Knoten 2“) zu übersetzen. Diese logischen Aussagen werden als Fakten in der logischen Programmiersprache PROLOG dargestellt. Die PROLOG-Fakten enthalten die selbe Information wie die ursprüngliche EPML-Datei und bilden einen Teil der PROLOG-Wissensbasis. Den zweiten Teil der PROLOG-Wissensbasis bilden Regeln, mit denen wir die „Sprache der EPKs“ in PROLOG „erklären“. Eine solche Regel besagt beispielsweise, dass ein Ereignis Endereignis ist, wenn es keine ausgehenden Kanten hat.

Die PROLOG-Wissensbasis enthält somit alle Informationen der ursprünglichen EPML-Datei und Regeln, die die Eigenschaften von EPK-Elementen beschreiben. Ausgehend von dieser Wissensbasis können wir nun dem PROLOG-System Fragen stellen - etwa, ob es mindestens ein Startereignis gibt. Insbesondere können wir für jede der in Abschnitt 3 aufgeführten Regeln erfragen, ob sie erfüllt sind. Die einzelnen Schritte unseres Ansatzes sind in den folgenden Unterabschnitten. dargestellt.

5.1 Übersetzung von EPML in PROLOG-Fakten

Die in unserem durchgehenden Beispiel genutzte EPML-Datei enthält Informationen zu den Knoten und Kanten des Graphen, der die EPK darstellt. So enthält der Abschnitt

```
<event id="10">
  <name>Kundenanfrage bearbeitet</name>
</event>
<arc id="11">
  <flow source="1" target="2"/>
</arc>
```

drei logische Aussagen:

1. Es gibt im Modell ein Ereignis mit id=10
2. Das Ereignis mit id=10 hat den Namen „Kundenanfrage bearbeiten“
3. Es gibt im Modell eine Kante vom Element mit id=1 zum Element mit id=2.

Im folgenden Listing sind die logischen Aussagen zu der in Abb. 1 dargestellten EPK, formuliert als Fakten der Sprache PROLOG, dargestellt. Die Bedeutung der Prädikate sollte ohne weitere Erklärung verständlich sein.

```
event(i_1).
elementname(i_1,'Kundenanfrage eingetroffen').
event(i_5).
elementname(i_5,'Artikel ist lieferbar').
event(i_6).
elementname(i_6,'Artikel nicht lieferbar').
event(i_10).
elementname(i_10,'Kundenanfrage bearbeitet').
function(i_2).
elementname(i_2,'Lieferstatus prüfen').
function(i_7).
elementname(i_7,'Artikel versenden').
function(i_8).
elementname(i_8,'Absage an Kunden senden').
arc(i_1,i_2).
arc(i_2,i_4).
arc(i_4,i_5).
arc(i_4,i_6).
arc(i_5,i_7).
arc(i_6,i_8).
arc(i_7,i_9).
arc(i_8,i_9).
arc(i_9,i_10).
xor(i_4).
xor(i_9).
```

Diese Fakten enthalten die selben Informationen wie Abb. 1 bzw. die zugehörige EPML-Datei. Um die Fakten aus der EPML-Datei zu generieren, benutzen wir eine einfache XSLT-Transformation, die wir im Anhang A angeben.

5.2 Die Regeln der Wissensbasis

Damit das PROLOG-System Schlussfolgerungen zu den aus der EPML-Datei gewonnenen Fakten ziehen kann, wird ihm ein gewisses „Grundwissen“ über EPKs in Form von logischen Regeln mitgeteilt. Die meisten dieser Regeln sind einfache Informationen zu verwendeten Bezeichnungsweisen. Dies soll an einigen Beispielregeln gezeigt werden:

```
connector(I) :- clause(and(I),true) ; clause(or(I),true);
clause(xor(I),true).
```

besagt, dass wir unter einem Verknüpfer einen AND-, XOR- oder OR-Verknüpfer verstehen.

```
no_outgoing_arcs(X) :- not(arc(X,_)).
```

besagt, dass für ein Element das Prädikat „hat keine ausgehenden Kanten“ erfüllt ist, wenn es keine Kante (d.h. kein arc-Element in der EPML-Datei) gibt, die von diesem Element ausgeht.

```
endevent(X) :- event(X),no_outgoing_arcs(X).
```

definiert ein Ereignis als Endereignis, wenn es keine ausgehenden Kanten hat.

Die komplette Regelbasis ist in Anhang B aufgeführt. Hervorzuheben ist die Kürze der Regeln: Lediglich die Regeln, die Aussagen zu Themen wie „Erreichbarkeit“ und „Zusammenhang“ treffen, benötigen mehr als eine einzelne Programmzeile, und auch diese Regeln zur Erreichbarkeit umfassen gerade einmal 13 PROLOG-Zeilen.

5.3 Die Validierung der EPK-Korrektheitsregeln

Nachdem durch die im vorigen Abschnitt besprochenen Regeln die für EPKs definierten Begriffe dem PROLOG-System „erklärt“ wurden, können wir nun Anfragen an das System stellen. Insbesondere können wir fragen, ob die im Abschnitt 3 aufgeführten Regeln für eine vorliegende EPK erfüllt sind. Dabei gehen wir davon aus, dass bereits geprüft wurde, dass die zu Grunde liegende EPML-Datei dem XML-Schema für EPML[MN04] entspricht. Dadurch sind die Regeln 1, 3 und das Verbot mehrfacher Kanten zwischen zwei Knoten (Teil von Regel 4) bereits „automatisch“ erfüllt.

Die Anfragen, mit denen die im Abschnitt 3 genannten Syntaxregeln für eine gegebene EPK überprüft werden können, sind im Anhang C angegeben. In der Regel sind sie so aufgebaut, dass dem PROLOG-System die Aufgabe gestellt wird, ein Beispiel zu finden, das die entsprechende Regel verletzt. Somit erhält der Benutzer nicht nur die Aussage, dass eine Syntaxregel verletzt ist, sondern es wird auch gezeigt, wo dies der Fall ist.

Einige Beispiele sollen das verdeutlichen: Als Teil von Regel 4 ist zu zeigen, dass keine Kante von a nach b existieren kann, wenn es schon eine Kante von b nach a gibt. Die entsprechende Anfrage an das PROLOG-System lautet:

```
prop4(X,Y) :- arc(X,Y),arc(Y,X).
```

Das Komma steht in PROLOG für die logische und-Verknüpfung, die angegebene Klausel drückt also die Forderung „finde zwei Elemente X und Y, für die sowohl eine Kante von X nach Y als auch eine Kante von Y nach X existiert“ aus. Ist die entsprechende Eigenschaft verletzt, werden die Knoten, die zur Verletzung führen, ausgegeben. Bei einem korrekten Modell antwortet das PROLOG-System auf die Anfrage prop4(X,Y) mit „no“, was heißt, dass keine Verletzung gefunden werden kann.

Da mit der im vorigen Kapitel definierten Regelbasis grundlegende Sprechweisen wie „ein Element X ist mit einem Element Y (möglicherweise über Verknüpfungsoperatoren) verbunden“[1] definiert wurden, gestalten sich die Anfragen nach Stellen, an denen eine Regel

[1] Diese Beziehung zwischen zwei Elementen X und Y ist im Prädikat successor definiert

verletzt wurden, recht einfach. Um die in Eigenschaft 10 aufgestellte Forderung, dass ein Ereignis immer (möglicherweise über Verknüpfungsoperatoren) mit einer Funktion verbunden ist zu prüfen, stellt man dem PROLOG-System die Frage:

```
prop10(X,Y) :- event(X),successor(X,Y),event(Y).
```

Damit wird das PROLOG-System veranlasst ein Gegenbeispiel zu suchen: Ein Ereignis, dass (möglicherweise über Verknüpfungsoperatoren) nicht mit einer Funktion, sondern mit einem Ereignis verbunden ist. Wie immer, besagt auch hier die Antwort „no“ auf die Anfrage prop10(X,Y), dass kein Gegenbeispiel gefunden werden kann, also die entsprechende Eigenschaft validiert werden konnte.

6 Bewertung

Der im vorangehenden Abschnitt vorgestellte Ansatz folgt einem gänzlich anderen Prinzip als die in [MN03b] diskutierten Schemasprachen. Indem die tatsächlich in der EPML-Datei steckende Information in logische Aussagen übersetzt wird, kann eine logikbasierte Sprache wie PROLOG komplexe Fragestellungen zum Modell leicht beantworten. Der zu notierende Code ist ähnlich kurz wie die in [MN03b] vorgeschlagenen Schematron-Tests. Da die gängigen PROLOG-Systeme wie das von uns verwendete SWI-Prolog offene Schnittstellen zu zahlreichen anderen Programmiersprachen bieten, können die Tests leicht in andere Tools eingebunden werden.

Ein Vorteil unserer Lösung ist, dass erstmals *alle* im Abschnitt 3 aufgeführte Regeln validiert werden können. Im Gegensatz dazu ist es mit dem Ansatz von [MN03b] nicht möglich, die Regeln 2, 9, 14 und 15 zu validieren, da dies ein teilweises Durchlaufen des Graphen erfordert.

Wir widersprechen der Aussage aus [MN03a], dass dieses Durchlaufen des Graphen teuer und folglich unperformant ist. Da EPK-Modelle in der Praxis wohl höchstens eine dreistellige Zahl von Elementen haben, erweisen sich die Tests auch der Regeln 2, 9, 14 und 15 selbst an relativ großen praktischen Modellen als hinreichend schnell, obwohl wir zugeben müssen, dass unsere Prolog-Regel zum Zusammenhang von Graphen nicht besonders effizient programmiert ist. In unseren Tests wurden alle diese Eigenschaften in weniger als einer Sekunde geprüft. Hinzu kommt, dass bei der in [MN03a] vorgeschlagenen Nutzung zusätzlicher type-Attribute für die Elemente der EPML-Datei keineswegs auf das Durchlaufen des Graphen verzichtet werden kann. Dieser muss zwar dank der type-Attribute nicht mehr bei der Validierung der Syntaxregeln erfolgen, dafür aber beim Erzeugen der EPML-Datei, da schließlich zu diesem Zeitpunkt bestimmt werden muss, wie die type-Attribute zu belegen sind.[2]

[2] Theoretisch könnte man zwar beim Bauen eines EPK-Modells in einem Editor die type-Attribute immer sofort in dem Moment bestimmen, in dem ein neues Element gezeichnet wird. Das setzt allerdings voraus, dass beim Editieren immer nur ein Einzelelement mit dem schon modellierten Teil der EPK verbunden wird. Werden Teile der EPK zunächst unabhängig voneinander modelliert, um schließlich durch Kanten verbunden zu werden, ist im Allgemeinen ein Graph-Durchlauf zur Bestimmung der type-Attribute notwendig.

Ein Vorteil unserer Lösung ist es, dass in den Dateien des Austauschformats EPML auf die zusätzlichen type-Attribute verzichtet werden kann. Dies führt dazu, dass das Austauschformat einfacher wird und folglich seine Verwendung einfacher implementiert werden kann. Generell halten wir die Notation der type-Attribute für nicht erstrebenswert, da diese Attribute (wenn sie denn korrekt sind, was in jedem Einzelfall ohnehin zunächst geprüft werden muss) lediglich redundante Informationen liefern.

7 Weiterführende Anwendungsmöglichkeiten unseres Ansatzes

Hauptziel dieses Beitrags ist es, die Nutzung logischer Programmierung zur Validierung von Eigenschaften ereignisgesteuerter Prozessketten zu zeigen. Wir wollen aber noch darauf hinweisen, dass sich die Anwendungsmöglichkeiten unserer Ideen keinesfalls auf die in Abschnitt 3 genannten Eigenschaften beschränken. Vielmehr kann man dem Prologsystem auch weitere Fragen zu anderen (statischen) Eigenschaften einer EPK stellen.

Beispielsweise kann es beim Model Checking großer EPK-Modelle sinnvoll sein, diese Modelle in kleinere Teilmodelle zu untergliedern, die dann getrennt im Model Checker untersucht werden. Damit kann sich der zu untersuchende Zustandsraum erheblich verringern. Es entsteht die Frage, an welchen Stellen man ein EPK-Modell „zerschneiden“ kann, so dass die entstehenden Teilmodelle selbst syntaktisch korrekte EPKs sind (oder zumindest durch das Hinzufügen der obligatorischen Start- und Endereignisse zu solchen ergänzt werden können.) Offenbar ist ein solches „Zerschneiden“ an solchen Kanten möglich, die die einzige Verbindung zwischen zwei Zusammenhangskomponenten des die EPK repräsentierenden Graphen darstellen. Eine solche Kante hat die Eigenschaft, dass der Graph nicht mehr zusammenhängend ist, wenn diese Kante entfernt wird. Ob das für eine Kante von X nach Y der Fall ist, prüft man leicht mit der folgenden Prolog-Regel:

```
cut_here(X,Y) :- arc(X,Y),
                 (retract(arc(X,Y)),prop2,assertz(arc(X,Y));
                 (assertz(arc(X,Y),fail))).
```

In diesem Beispiel wird zunächst verlangt, dass zwischen X und Y tatsächlich eine Kante existiert, dargestellt durch den Fakt arc(X,Y). Dann wird mittels `retract` der Fakt, dass diese Kante existiert, aus der Wissensbasis entfernt. Nun wird die Eigenschaft 2 aus Abschnitt 3 (Zusammenhang des Graphen) für die um die Kante $X \rightarrow Y$ verminderte Wissensbasis getestet. Wie bei all unseren Tests, wird die Antwort „yes“ geliefert, wenn ein Gegenbeispiel gefunden wurde, also der Graph nicht (mehr) zusammenhängend ist. Die anschließenden `assertz`-Befehle dienen dazu, anschließend den ursprünglichen Fakt, dass es eine Kante von X nach Y gibt, wieder in die Wissensbasis aufzunehmen.

Weitere denkbare Anwendungsfälle sind das Erkennen von Modellfehlern (etwa nach dem in [vDMvdA06] vorgestellten Ansatz) oder das Aufspüren schlechten Modellierungsstils (z.B. von Startereignissen, von denen aus direkt in einen durch einen Split-Konnektor eingeleiteten Kontrollblock „hineingesprungen“ wird).

8 Zusammenfassung

In unserem Beitrag haben wir einen auf dem Prinzip der logischen Programmierung beruhenden Ansatz zur Validierung der syntaktischen Korrektheit von EPML-Dateien vorgestellt. Dieser kann im Gegensatz zu bisherigen Lösungen[MN03b] erstmals *alle* in der Literatur[Rum99, Kel99, NR02] aufgestellten Forderungen an korrekte Syntax validieren. Dabei kommt er mit einem knapperen, redundanzfreien Format der Austauschsprache EPML aus, was deren Nutzung als Austauschformat erleichtern kann.

Literatur

[CM01] James Clark und Murata Makoto. *RELAX NG Specification*. OASIS, 1. Auflage, December 2001.

[Cun04] Nicolas Cuntz. Über die effiziente Simulation von Ereignisgesteuerten Prozessketten. Diplomarbeit, Universität Paderborn, 2004.

[DJ06] Ulrich Ultes Nitsche Dominik Jungo, David Buchmann. Testing of semantic properties in XML documents. In *Proceedings of the 4th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, Paphos, Cyprus*, 2006.

[Jel02] Rick Jelliffe. *The Schematron Assertion Language 1.5*. Academia Sinica Computing Centre, October 2002.

[Kel99] Gerhard Keller. *SAP R/3 prozessorientiert anwenden*. Addison-Wesley, München, 1999.

[MN03a] J. Mendling und M. Nüttgens. EPC Modelling based on Implicit Arc Types, 2003.

[MN03b] Jan Mendling und Markus Nüttgens. EPC Syntax Validation with XML Schema Languages. In *EPK*, Seiten 19–30, 2003.

[MN04] J. Mendling und M. Nüttgens. Exchanging EPC Business Process Models with EPML. In M. Nüttgens und J. Mendling, Hrsg., *XML4BPM 2004, Proceedings of the 1st GI Workshop XML4BPM – XML Interchange Formats for Business Process Management at 7th GI Conference Modellierung 2004, Marburg Germany, March 2004*, Seiten 61–80, March 2004.

[NR02] Markus Nüttgens und Frank J. Rump. Syntax und Semantik Ereignisgesteuerter Prozessketten (EPK). In *Promise 2002 - Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen*, Seiten 64–77, 2002.

[Rum99] Frank J. Rump. *Geschäftsprozeßmanagement auf der Basis ereignisgesteuerter Prozeßketten*. B. G. Teubner Verlag Stuttgart Leipzig, 1999.

[vDMvdA06] B.F. van Dongen, J. Mendling und W.M.P. van der Aalst. Structural Patterns for Soundness of Business Process Models. *EDOC*, Seiten 116–128, 2006.

[Wor99] World Wide Web Consortium. *XML Path Language (XPath)*, November 1999.

[Wor01a] World Wide Web Consortium. *XML Schema Part 1: Structures*, May 2001.

[Wor01b] World Wide Web Consortium. *XML Schema Part 2: Datatypes*, May 2001.

A XSLT-Stylesheet zur Transformation der EPML-Datei in PROLOG-Fakten

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="epc">
    <xsl:apply-templates select="event"/>
    <xsl:apply-templates select="function"/>
    <xsl:apply-templates select="arc"/>
    <xsl:apply-templates select="and"/>
    <xsl:apply-templates select="or"/>
    <xsl:apply-templates select="xor"/>
</xsl:template>

<xsl:template match="function">
  function(i_<xsl:value-of select="@id"/>).
   elementname(i_<xsl:value-of select="@id"/>,'<xsl:value-of select="name"/>').
</xsl:template>

<xsl:template match="event">
  event(i_<xsl:value-of select="@id"/>).
   elementname(i_<xsl:value-of select="@id"/>,'<xsl:value-of select="name"/>').
</xsl:template>

<xsl:template match="and">
  and(i_<xsl:value-of select="@id"/>).
</xsl:template>

<xsl:template match="or">
  or(i_<xsl:value-of select="@id"/>).
</xsl:template>

<xsl:template match="xor">
  xor(i_<xsl:value-of select="@id"/>).
</xsl:template>

<xsl:template match="arc">
   arc(i_<xsl:value-of select="flow/@source"/>,
   i_<xsl:value-of select="flow/@target"/>).
</xsl:template>

</xsl:stylesheet>
```

B PROLOG-Regeln

```
% some useful facts about arcs
uarc(X,Y) :- arc(X,Y);arc(Y,X).
more_than_one_incoming_arcs(X) :- arc(A,X),arc(B,X),A \==B.
more_than_one_outgoing_arcs(X) :- arc(X,A),arc(X,B),A \==B.
no_incoming_arcs(X) :- not(arc(_,X)).
no_outgoing_arcs(X) :- not(arc(X,_)).
% successor(X,Y) means that X is followed by Y (possibly via some connectors)
% (Note: prop9 must be tested before calling successor(X,Y) in order to avoid
% infinite cycles.)
successor(X,Y) :- arc(X,Y).
successor(X,Y) :- arc(X,C),connector(C),successor(C,Y).
% types of elements
connector(I) :- clause(and(I),true) ; clause(or(I),true) ; clause(xor(I),true).
element(I) :- event(I);function(I);connector(I).
startevent(X) :- event(X),no_incoming_arcs(X).
endevent(X) :- event(X),no_outgoing_arcs(X).
% When using split(X) and join(X), we assume that prop8 already has been tested.
split(X) :- connector(X),more_than_one_outgoing_arcs(X).
join(X) :- connector(X),more_than_one_incoming_arcs(X).
% paths and reachability
path(A,B,Path) :- travel(A,B,[A],Q),
       reverse(Q,Path).

travel(A,B,P,[B|P]) :- arc(A,B).
travel(A,B,Visited,Path) :- arc(A,C),
       C \== B,
       \+member(C,Visited),
       travel(C,B,[C|Visited],Path).

% Neighbourhood contains all elements of List and their neighbours.
neighbourhood(List,Neighbourhood) :-
findall(N,(member(X,List),uarc(X,N)),Neighbours),
union(List,Neighbours,List_And_Neighbours),
list_to_ord_set(List_And_Neighbours,Neighbourhood).

% Find the elements which are weakly connected to at least one element in List
connected_elements(List,X) :- neighbourhood(List,List), X = List.
connected_elements(List,X) :- neighbourhood(List,Neighbourhood),
connected_elements(Neighbourhood,X).
```

C Anfragen zur Validierung der EPK-Syntaxregeln

```
% Property 1 - The EPC is a directed graph
% Property 2 - The EPC is a coherent graph
prop2 :- element(E1),connected_elements([E1],Y),!,element(X),not(member(X,Y)).
% Property 3 - The EPC is finite
% Property 4 - There are no multiple arcs between two vertices
% (no need to check this, because they are already forbidden by the EPML Schema)
% The EPC is an antisymmetric graph
prop4(X,Y) :- arc(X,Y),arc(Y,X).
% Property 5 - The set of events is not empty and the set of functions is not empty
prop5 :- not(clause(event(_),true)).
prop5 :- not(clause(function(_),true)).
% Property 6 - Functions have exactly one incoming arc and exactly one outgoing arc.
prop6(X) :- function(X),
               (more_than_one_outgoing_arcs(X);more_than_one_incoming_arcs(X);
               no_incoming_arcs(X);no_outgoing_arcs(X)).
% Property 7 - Events have at most one incoming and at most one outgoing arc.
% (note: This would allow events with no incoming and no outgoing arc, but this
% would be detected when verifying prop 2)
prop7(X) :- event(X),
               (more_than_one_incoming_arcs(X);more_than_one_outgoing_arcs(X)).
% Property 8 - Two kinds of connectors are allowed:
%     split connectors with exactly one incoming arc and at least two outgoing arcs
%     join connectors with at least two incoming arc and exactly one outgoing arc.
%     (connectors with no incoming and no outgoing arcs are already detected
%     when verifying prop 2)
prop8(X) :- connector(X),
               more_than_one_outgoing_arcs(X),more_than_one_incoming_arcs(X).
% Property 9 - Cycles made up only of connectors are forbidden
connectors_only([L|Rest]) :- connector(L),!,connectors_only(Rest).
connectors_only([]).
prop9(Path) :- path(X,X,Path),connectors_only(Path).
% Property 10 - If an event has an outgoing arc, this arc connects the event
%      (possibly via one or more connectors) to a function.
prop10(X,Y) :- event(X),successor(X,Y),event(Y).
% Property 11- The outgoing arc of a function connects this function
%      (possibly via one or more connectors) to an event
prop11(X,Y) :- function(X),successor(X,Y),function(Y).
% Property 12 - If an event is followed by one or more connectors,
%       none of these connectors is an XOR-split or OR-split
prop12(X) :- event(X),successor(X,Y) ,
                (clause(or(Y),true) ; clause(xor(Y),true)).
% Property 13a - There is at least one start event
startevents(L) :- findall(X,startevent(X),L).
prop13a :- startevents(_) == [].
% Property 13b - There is at least one end event
endevents(L) :- findall(X,endevent(X),L).
prop13b :-endevents(_) == [].
% Property 14 - For every element X, there is a path from a start event to X
reachable_from_startevent(X) :- startevent(S),path(S,X,_).
prop14(X) :- element(X),not(startevent(X)),not(reachable_from_startevent(X)).
% Property 15 - For every element X, there is a path from X to an end event
reaches_endevent(X) :- endevent(E),path(X,E,_).
prop15(X) :- element(X),not(endevent(X)),not(reaches_endevent(X)).
```

D Anmerkungen zum Programmcode

1. Der oben angeführte Code wurde mit SWI-Prolog, Version 5.2.13 getestet. Er sollte jedoch auch auf anderen PROLOG-Systemen lauffähig sein, ggf. müssen nicht vorhandene Builtin-Prädikate wie list_to_ord_set zum System hinzugefügt werden.

2. Prinzipiell ist es auch problemlos realisierbar, dass das PROLOG-System die EPML-Datei direkt einliest und verarbeitet. Das von uns verwendete System SWI-Prolog stellt hierfür die Bibliothek sgml2p zur Verfügung. Der Umweg über die XSLT-Transformation könnte dann entfallen. Wir haben in unserem Beitrag den Weg über die XSLT-Transformation gewählt, da so der Code leichter lesbar wird.

3. Gelegentlich ist die Reihenfolge, in der die Regeln aufgerufen werden, wichtig. Dies ist dann als Kommentar im Code vermerkt. Beispielsweise muss zunächst sichergestellt werden, dass es mindestens ein Ereignis gibt (Property5), bevor das entsprechende Prädikat `event` an anderer Stelle verwendet wird, um weitere Eigenschaften von Ereignissen zu prüfen.