

Studienarbeit

Statische Analyse von BPEL4WS-Prozeßmodellen

Thomas Heidinger

Betreuer: Dr. Karsten Schmidt

8. Dezember 2003

Inhaltsverzeichnis

1	Einleitung	4
2	Statische Analyse	5
2.1	Verbände	6
2.2	Fixpunkte	7
2.3	Datenflußgleichungen	7
2.4	Allgemeines Schema	8
3	Web Services	9
3.1	Scopes	9
3.2	Variablen	10
3.3	Basisaktivitäten	10
3.3.1	invoke	11
3.3.2	receive	11
3.3.3	reply	12
3.3.4	assign	13
3.3.5	wait	14
3.3.6	throw	14
3.3.7	terminate	15
3.3.8	empty	15
3.4	Strukturierte Aktivitäten	15
3.4.1	sequence	16
3.4.2	switch	16
3.4.3	while	17
3.4.4	pick	18
3.4.5	flow	18
4	Nichtinitialisierte Variablen	21
4.1	Struktur, Repräsentation der Daten	21
4.2	Datenflußgleichungen	22
4.3	Lösung	28
5	Beispiel	29
6	Zusammenfassung	37

Abbildungsverzeichnis

2.1	KFG	5
3.1	invoke	11
3.2	receive	12
3.3	reply	12
3.4	assign	13
3.5	wait	14
3.6	throw	14
3.7	terminate	15
3.8	empty	15
3.9	sequence	16
3.10	switch	17
3.11	while	17
3.12	pick	18
3.13	flow	19
3.14	link	20
4.1	process	23
4.2	reply, wait, empty	23
4.3	receive	23
4.4	invoke	24
4.5	assign	24
4.6	sequence	24
4.7	while	25
4.8	switch	25
4.9	pick	26
4.10	flow	27
4.11	link, source	27
4.12	link, target	27
4.13	terminate	27
5.1	Purchase Order Process	29
5.2	Kontrollflußgraph Purchase Order Process	34
5.3	Lesender Zugriff auf Variablen	36

1 Einleitung

Web Services haben in den vergangenen Jahren eine immer größere Bedeutung in Business Prozessen gewonnen. Die Sprache BPEL ermöglicht es, das Verhalten eines Web Service zu beschreiben. Am Institut für Informatik der Humboldt Universität wurde eine Petrinetz-Semantik für diese Sprache entwickelt. Wird nun ein Web Service mit Hilfe von BPEL beschrieben, dann stellt sich die Frage, ob dieser Prozeß nichts Verbotenes macht, d.h. sich an alle Vorgaben der Spezifikation hält. Die Syntax läßt sich einfach durch einen BPEL-Parser [bpe03] überprüfen. Um die Korrektheit eines Modells zu beweisen, müssen jedoch auch semantische Eigenschaften überprüft werden. Dazu existieren verschiedene Herangehensweisen, so könnte man z.B. die erstellten Petrinetze untersuchen. Dafür können unter anderem Invarianten aufgestellt werden oder durch Model Checking der gesamte Zustandsraum analysiert werden. In dieser Arbeit soll jedoch ein ganz anderer Ansatz verfolgt werden: die statische Programmanalyse.

Im folgenden Kapitel werden die theoretischen Grundlagen der statischen Analyse gelegt und die vier verschiedenen Datenflußschemata (*sicher, möglich, vorwärts, rückwärts*) vorgestellt.

Im Kapitel 3 werden die für unsere Betrachtung wichtigen BPEL-Konstrukte anhand ihres Quelltextes vorgestellt und kurz beschrieben. Außerdem wird ein Pattern für jede BPEL-Aktivität angegeben, mit deren Hilfe sich der Kontrollflußgraph eines BPEL-Prozeßmodells aufbauen läßt. Auf diesem Kontrollflußgraphen können wir dann unsere Datenflußanalyse durchführen.

Das Kapitel 4 beschäftigt sich mit dem Problem der nicht initialisierten Variablen. Es wird gezeigt, wie diese Eigenschaft unter Zuhilfenahme der Pattern aus dem Kapitel 3 überprüft werden kann, wodurch beispielhaft demonstriert wird, daß sich die statische Analyse eignet, um eine interessante Eigenschaft eines BPEL-Prozeßmodells nachzuweisen.

Im abschließenden Kapitel 5 führen wir unsere Analysemethode anhand eines Beispiels vor.

2 Statische Analyse

Statische Analyse beschäftigt sich mit der Analyse von Programmen zur Compilezeit, und grenzt sich damit von der dynamischen Analyse ab, bei der Informationen über ein Programm zur Ausführungszeit ermittelt werden. Ein Teilgebiet der statischen Analyse ist die Abstrakte Interpretation [Cou96, CC77], bei der man allgemeingültige, von konkreten Eingabewerten entkoppelte Aussagen über ein Programm treffen möchte. Zu den wichtigsten Ergebnissen auf diesem Gebiet gehören die Arbeiten von Cousot [CC79, CC92, CC76]. Bedeutende Anwendungen der statischen Analyse sind nicht nur die Programmverifikation, sondern auch der Compilerbau [ASU99a] und damit verbunden die Programmoptimierung. So wird sie zum Beispiel zum Finden toten Codes, zur Feldgrenzenüberprüfung von Arrays, Konstantenpropagierung (welche Variablen einen konstanten Wert haben) und der Überprüfung, ob Variable vor dem ersten Lesen initialisiert wurden, eingesetzt. Grundlegende Informationen zur Datenflußanalyse im Compilerbau und zur Programmoptimierung finden sich in [ASU99b].

Betrachten wir zunächst ein kleines Beispiel:

```
read(x);
  while (x > 1)
x = x-2;
print(x);
```

Dieses Programm hat folgenden Kontrollflußgraphen:

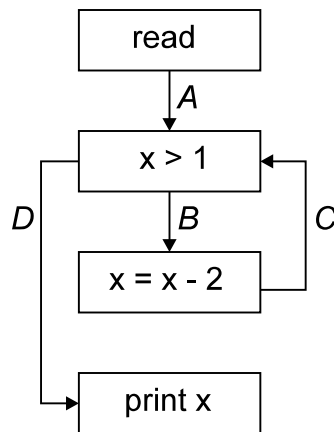


Abbildung 2.1: KFG

Nun ergeben sich die folgenden Datenflußgleichungen:

$$A = \{input\}$$

$$\begin{aligned} B &= A \cup C \cap \{x \mid x > 1, x \in \mathbf{N}\} \\ C &= \{y \mid y = x - 2, x \in B\} \\ D &= A \cup C \cap \{x \mid x \leq 1, x \in \mathbf{N}\} \end{aligned}$$

Bei einer Eingabe mit $\{6\}$ ergibt sich iterativ dieser Datenfluß:

$$\begin{aligned} A &= \{6\} \\ B &= \{6, 4, 2\} \\ C &= \{4, 2, 0\} \\ D &= \{0\} \end{aligned}$$

Es handelt sich um den kleinsten Fixpunkt des Gleichungssystems und entspricht der kleinsten Lösung.

Zur statischen Analyse von Programmeigenschaften möchten wir im folgenden ein allgemeines Schema angeben. Dazu werden wir nun die theoretischen Grundlagen legen. Eine genauere Betrachtung findet sich in [Sch] und [NNH99].

2.1 Verbände

Halbordnung

Sei U eine Menge und \sqsubseteq eine binäre Relation über U , die folgende drei Eigenschaften erfüllt:

- Reflexivität: $\forall x \in U : x \sqsubseteq x$
- Transitivität: $\forall x, y, z \in U : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
- Antisymmetrie: $\forall x, y \in U : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$

Dann ist $H = (U, \sqsubseteq)$ eine *Halbordnung*.

Obere Schranke

Sei $X \subseteq U$. $s \in U$ ist eine *obere Schranke* für X ($X \sqsubseteq s$), wenn $\forall x \in X : x \sqsubseteq s$.

Untere Schranke

Sei $X \subseteq U$. $s \in U$ ist eine *untere Schranke* für X ($s \sqsubseteq X$), wenn $\forall x \in X : s \sqsubseteq x$.

Supremum

Das *Supremum* oder auch die *kleinste obere Schranke* $\sqcup X$ ist definiert durch:

$$X \sqsubseteq \sqcup X \wedge \forall y \in U : X \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y$$

Infimum

Das *Infimum* oder auch die *größte untere Schranke* $\sqcap X$ ist definiert durch:

$$\sqcap X \sqsubseteq X \wedge \forall y \in U : y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$$

Verband

Ein *Verband* ist eine Halbordnung, in der eine kleinste obere und größte untere Schranke für alle $X \subseteq U$ existieren.

Daraus ergibt sich, daß ein Verband ein größtes Element $\top = \sqcup U$ und ein kleinstes Element $\perp = \sqcap U$ besitzt. Weiterhin existiert ein binärer Infimum-Operator \sqcap und ein binärer Supremum-Operator \sqcup . Offensichtlich gilt: $x \sqcup \top = \top$ und $x \sqcap \perp = \perp$.

2.2 Fixpunkte

Eine Funktion $f : U \rightarrow U$ heißt *monoton*, wenn $\forall x, y \in U : x \leq y \Rightarrow f(x) \leq f(y)$.

Die Funktion f ist *stetig*, wenn $f(\sqcup X) = \sqcup f(X)$ mit $f(X) = f([x_1, x_2, \dots]) = [f(x_1), f(x_2), \dots]$.

Es gilt: f monoton $\wedge U$ endlich $\Rightarrow f$ stetig

Fixpunktsatz

Für einen Verband mit kleinstem Element \perp und stetige monotone Funktion $f : U \rightarrow U$ gilt: kleinster Fixpunkt $X = \bigsqcup_{i \geq 0} f^i(\perp)$. X iterativ berechenbar.

2.3 Datenflußgleichungen

Generell kann man eine Analysemethode in vier verschiedene Kategorien einteilen; sie kann *vorwärts* oder *rückwärts*, *sicher* oder *möglich* sein.

Eine *vorwärts* Analyse berechnet für jeden Programmpunkt Informationen über die Vergangenheit, wohingegen eine *rückwärts* Analyse Informationen über die Zukunft berechnet.

Eine *sicher* Analyse berechnet Informationen, die garantiert richtig sind und daher eine untere Approximation darstellen. Eine *möglich* Analyse berechnet dagegen Informationen, die richtig sein können und somit eine obere Approximation darstellen.

Im folgenden bezeichne

- M Programmstelle
- A Anweisung
- $E(M)$ Eigenschaft an Programmstelle M
- $kill(A)$ Eigenschaften E , die durch A beseitigt werden
- $gen(A)$ Eigenschaften E , die durch A neu definiert werden.

Damit lassen sich folgende vier DFA-Schemata angeben:

2 Statische Analyse

- vorwärts und sicher:

$$\begin{aligned}E_{in}(A) &= \bigcap_{X \in pre(A)} E_{out}(X) \\E_{out}(A) &= E_{in}(A) - kill(A) \cup gen(A)\end{aligned}$$

- rückwärts und sicher:

$$\begin{aligned}E_{out}(A) &= \bigcap_{X \in post(A)} E_{in}(X) \\E_{in}(A) &= E_{out}(A) - kill(A) \cup gen(A)\end{aligned}$$

- vorwärts und möglich:

$$\begin{aligned}E_{in}(A) &= \bigcup_{X \in pre(A)} E_{out}(X) \\E_{out}(A) &= E_{in}(A) - kill(A) \cup gen(A)\end{aligned}$$

- rückwärts und möglich:

$$\begin{aligned}E_{out}(A) &= \bigcup_{X \in post(A)} E_{in}(X) \\E_{in}(A) &= E_{out}(A) - kill(A) \cup gen(A)\end{aligned}$$

2.4 Allgemeines Schema

Um ein Datenflußproblem zu lösen, kann man in vielen Fällen nach folgendem Schema Vorgehen:

- Zu analysierende Eigenschaft bestimmen.
- Einen geeigneten Verband festlegen.
- Analysemethode wählen (*vorwärts*, *rückwärts*, *sicher*, *möglich*).
- Stetige, monotone Datenflußgleichungen aufstellen.
- Fixpunkt des Gleichungssystems berechnen.

3 Web Services

Der Begriff Web Service scheint ein neues Schlagwort der Industrie geworden zu sein. Viele Unternehmen verwenden ihn in unterschiedlichen Kontexten, so daß es schwierig ist, eine einheitliche Definition zu finden. Eine sinnvolle allgemeine Definition ist aber sicherlich die Folgende: Ein Web Service ist eine abgeschlossene, selbsterklärende Software-Komponente, welche über das Internet veröffentlicht, gesucht und benutzt wird und mit anderen Web Services zu neuen Services verbunden werden kann.

Um einen Web Service bereitzustellen, müssen nicht komplett neue Technologien verwendet werden, vielmehr baut der sogenannte Web Service Technologiestack auf bereits bekannten Technologien auf. Die Kommunikation mit Web Services kann z.B. mit HTTP, TCP/IP und SOAP erfolgen. Die Schnittstellenbeschreibung eines Web Service erfolgt mit WSDL, einem XML Vokabular und letztendlich kann das Verhalten eines Web Service mit BPEL4WS, ebenfalls einem XML Vokabular, beschrieben werden.

Im Folgenden sollen die wichtigsten BPEL Konstrukte kurz vorgestellt werden, die zum Beschreiben eines Web Service nötig sind. Dabei soll auf erweiterbare Konzepte wie z.B. Fehlerbehandlung nicht genauer eingegangen werden. Weiterführende Informationen finden sich in [ACD⁺03]. Außerdem wird in diesem Kapitel gezeigt, wie ein Kontrollflußgraph für ein BPEL-Prozeßmodell erzeugt werden kann. Dazu wird für jedes BPEL-Konstrukt ein Pattern definiert. Der Kontrollflußgraph ergibt sich dann durch einfaches zusammenstecken solcher Pattern. Jedem ein- und ausgehenden Zweig eines Knoten im Kontrollflußgraphen wird ein Bezeichner zugeordnet, der je nach zu analysierender Eigenschaft mit einer Menge assoziiert wird. Auf dem entstandenen Kontrollflußgraphen kann man dann eine statische Analyse durchführen und z.B. toten Code finden. Im anschließenden Kapitel wird beispielhaft das Problem der nichtinitialisierten Variablen behandelt.

3.1 Scopes

Ein `<scope>` stellt den Verhaltenskontext einer Aktivität zur Verfügung. Innerhalb eines `<scope>` können sogenannte `correlationSets`, `faultHandlers`, `compensationHandlers`, `eventHandlers` und `variables` definiert werden, auf die wir im nächsten Abschnitt noch genauer eingehen werden.

```
<scope variableAccessSerializable="yes|no"
  name="ncname"?
  joinCondition="bool-expr"?
  suppressJoinFailure="yes|no"?>
<target linkName="ncname"/>*
<source linkName="ncname" transitionCondition="bool-expr"?/>*
```

```

<variables>?
...
</variables>

<correlationSets>?
...
</correlationSets>

<faultHandlers>?
...
</faultHandlers>

<compensationHandler>?
...
</compensationHandler>

<eventHandlers>?
...
</eventHandlers>

activity

</scope>

```

3.2 Variablen

Variablen dienen dem Nachrichtenaustausch mit Partnern und dem Speichern von Daten. Jede Variable hat einen Typ. Dazu kann man unter `messageType` ein WSDL message type oder unter `type` ein XML Schema simple type oder unter `element` ein XML Schema element angeben. Der Sichtbarkeitsbereich einer Variable ist der Scope, in der die Variable definiert wurde. Eine Variable ist auch in jedem inneren Scope sichtbar, sofern sie nicht durch eine Variable mit gleichem Namen überdeckt wird. Variablen müssen initialisiert werden, bevor sie gelesen werden.

```

<variables>
  <variable name="ncname" messageType="qname"?
            type="qname"? element="qname"?/>+
</variables>

```

3.3 Basisaktivitäten

Die Basisaktivitäten sind die elementaren Aktivitäten eines Prozesses, die nicht weiter verschachtelt werden können, d.h. keine anderen Aktivitäten enthalten.

3.3.1 invoke

Ein `<invoke>` dient dazu, einen anderen Web Service aufzurufen. Es gibt ein synchrones und ein asynchrones `<invoke>`. Beim asynchronen wird nur eine `inputVariable` definiert, beim synchronen zusätzlich eine `outputVariable`. Es wird dem `partnerLink` die `inputVariable` übergeben und am `portType` die `operation` ausgeführt. Im synchronen Fall wird auf eine Antwort in der `outputVariable` gewartet.

```

<invoke partnerLink="ncname" portType="qname" operation="ncname"
  inputVariable="ncname" outputVariable="ncname"?
  name="ncname"? joinCondition="bool-expr"?
  suppressJoinFailure="yes|no"?>
  <target linkName="ncname"/>*
  <source linkName="ncname" transitionCondition="bool-expr"?/>*
  <correlations>?
    <correlation set="ncname" initiate="yes|no"?
      pattern = "in|out|out-in" />+
  </correlations>
  <catch faultName="qname" faultVariable="ncname"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
  <compensationHandler>?
    activity
  </compensationHandler>
</invoke>

```

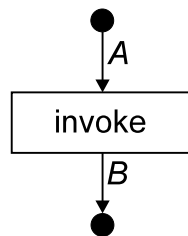


Abbildung 3.1: invoke

3.3.2 receive

Ein `<receive>` empfängt eine Nachricht eines Partners. Es wird beim `partnerLink` am `portType` die `operation` ausgeführt und die erwartete Nachricht an die `variable` gebunden.

```

<receive partnerLink="ncname" portType="qname" operation="ncname"
  variable="ncname"? createInstance="yes|no"?

```

```

    name="ncname"? joinCondition="bool-expr"?
    suppressJoinFailure="yes|no"?>
<target linkName="ncname"/>*
<source linkName="ncname" transitionCondition="bool-expr"?/>*
<correlations>?
    <correlation set="ncname" initiate="yes|no"?>+
</correlations>
</receive>

```

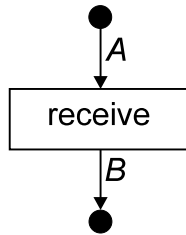


Abbildung 3.2: receive

3.3.3 reply

Ein `<reply>` sendet die Nachricht in der `variable` an die operation des `portType` des `partnerLink`.

```

<reply partnerLink="ncname" portType="qname" operation="ncname"
    variable="ncname"? faultName="qname"?
    name="ncname"? joinCondition="bool-expr"?
    suppressJoinFailure="yes|no"?>
<target linkName="ncname"/>*
<source linkName="ncname" transitionCondition="bool-expr"?/>*
<correlations>?
    <correlation set="ncname" initiate="yes|no"?>+
</correlations>
</reply>

```

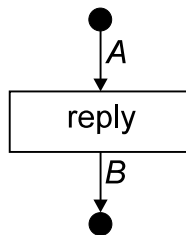


Abbildung 3.3: reply

3.3.4 assign

Ein `<assign>` kopiert einen typkompatiblen Wert von der Quelle `from-spec` zum Ziel `to-spec`. So läßt sich unter anderem einer Variablen ein Wert zuweisen.

```
<assign name="ncname"? joinCondition="bool-expr"?
  suppressJoinFailure="yes|no"?>
  <target linkName="ncname"/>*
  <source linkName="ncname" transitionCondition="bool-expr"?/>*
  <copy>+
    from-spec
    to-spec
  </copy>
</assign>
```

Die `from-spec` muß eines der sechs folgenden Formen haben:

```
<from variable="ncname"/>
<from variable="ncname" part="ncname"/>
<from partnerLink="ncname" endpointReference="myRole|partnerRole"/>
<from variable="ncname" property="qname"/>
<from expression="general-expr"/>
<from> ... literal value ... </from>
```

Und für die `to-spec` kann man unter folgenden wählen:

```
<to variable="ncname"/>
<to variable="ncname" part="ncname"/>
<to variable="ncname" property="qname"/>
<to partnerLink="ncname"/>
```

So ist es möglich, eine Variable nur teilweise zu initialisieren, indem nicht allen parts und properties ein Wert zugewiesen wird.

Um den Kontrollflußgraph aufzubauen, wird für ein `<assign>` für jedes `<copy>` Element das Pattern aus Abbildung 3.4 in Sequenz geschaltet.

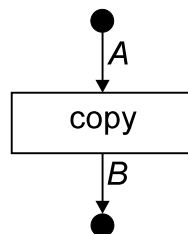


Abbildung 3.4: assign

3.3.5 wait

Ein `<wait>` wartet eine bestimmte Zeit (`duration-expr`) oder erwartet einen Zeitpunkt (`deadline-expr`).

```
<wait (for="duration-expr" | until="deadline-expr")
  name="ncname"? joinCondition="bool-expr"?
  suppressJoinFailure="yes|no"?>
  <target linkName="ncname"/>*
  <source linkName="ncname" transitionCondition="bool-expr"?/>*
</wait>
```

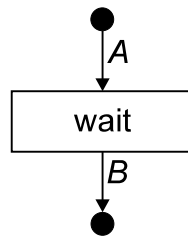


Abbildung 3.5: wait

3.3.6 throw

Ein `<throw>` wirft eine Ausnahme, die durch einen Faulthandler des zugehörigen Scopes gefangen werden kann.

```
<throw faultName="qname" faultVariable="ncname"?
  name="ncname"? joinCondition="bool-expr"?
  suppressJoinFailure="yes|no"?>
  <target linkName="ncname"/>*
  <source linkName="ncname" transitionCondition="bool-expr"?/>*
</throw>
```

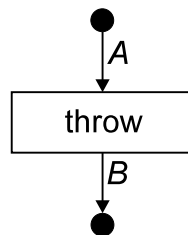


Abbildung 3.6: throw

3.3.7 terminate

Ein `<terminate>` beendet den Business Prozeß sofort. Daher endet in einem `<terminate>` der Kontrollfluß.

```
<terminate name="ncname"? joinCondition="bool-expr"?
  suppressJoinFailure="yes|no"?>
  <target linkName="ncname"/>*
  <source linkName="ncname" transitionCondition="bool-expr"?/>*
</terminate>
```

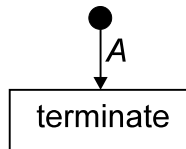


Abbildung 3.7: terminate

3.3.8 empty

Ein `<empty>` macht nichts. Es kann zum Beispiel zum Synchronisieren von Aktivitäten oder als Defaultimplementaion eines Handlers verwendet werden.

```
<empty name="ncname"? joinCondition="bool-expr"?
  suppressJoinFailure="yes|no"?>
  <target linkName="ncname"/>*
  <source linkName="ncname" transitionCondition="bool-expr"?/>*
</empty>
```

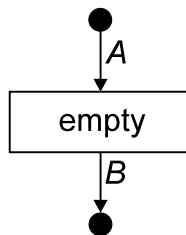


Abbildung 3.8: empty

3.4 Strukturierte Aktivitäten

Die strukturierten Aktivitäten können rekursiv geschachtelt werden und bestehen aus den Basisaktivitäten, deren Abarbeitungsreihenfolge sie beeinflussen.

3.4.1 sequence

Die Aktivitäten innerhalb einer `<sequence>` werden gemäß ihrer Anordnung ausgeführt. Der Kontrollfluß verläuft daher linear von Aktivität zu Aktivität.

```
<sequence name="ncname"? joinCondition="bool-expr"?
  suppressJoinFailure="yes|no"?>
  <target linkName="ncname"/>*
  <source linkName="ncname" transitionCondition="bool-expr"?/>*
  activity+
</sequence>
```

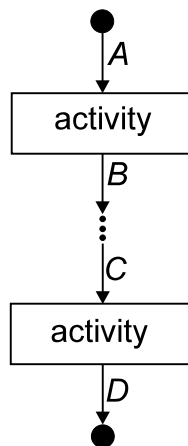


Abbildung 3.9: sequence

3.4.2 switch

Ein `<switch>` ermöglicht eine datenabhängige Verzweigung des Kontrollflusses. Es wertet einen Ausdruck aus und wählt dementsprechend den passenden Zweig aus. Gibt es keinen passenden `<case>` Zweig, dann wird der `<otherwise>` Zweig ausgeführt.

```
<switch name="ncname"? joinCondition="bool-expr"?
  suppressJoinFailure="yes|no"?>
  <target linkName="ncname"/>*
  <source linkName="ncname" transitionCondition="bool-expr"?/>*
  <case condition="bool-expr">+
    activity
  </case>
  <otherwise>?
    activity
  </otherwise>
</switch>
```

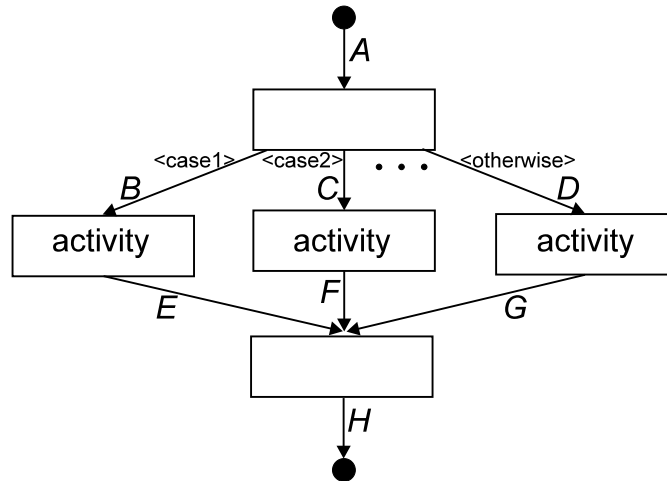


Abbildung 3.10: switch

3.4.3 while

Ein `<while>` ermöglicht die Einführung von Zyklen in den Kontrollfluß. Die eingeschlossene Aktivität wird bis zum Erreichen der Abbruchbedingung ausgeführt.

```

<while condition="bool-expr"
  name="ncname"? joinCondition="bool-expr"?
  suppressJoinFailure="yes|no"?>
  <target linkName="ncname"/>*
  <source linkName="ncname" transitionCondition="bool-expr"?/>*
  activity
</while>

```

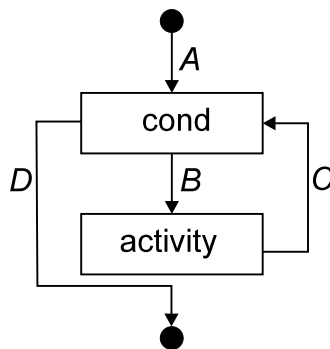


Abbildung 3.11: while

3.4.4 pick

Ein `<pick>` ermöglicht die nichtdeterministische Verzweigung abhängig von externen Nachrichten. Es wählt abhängig von einer einkommenden Nachricht (`<onMessage>`) oder eines Alarms (`<onAlarm>`) einen Bearbeitungszweig aus.

```

<pick createInstance="yes|no"?
  name="ncname"? joinCondition="bool-expr"?
  suppressJoinFailure="yes|no"?>
  <target linkName="ncname"/>*
  <source linkName="ncname" transitionCondition="bool-expr"?/>*
  <onMessage partnerLink="ncname" portType="qname"
    operation="ncname" variable="ncname"?>+
    <correlations?>?
      <correlation set="ncname" initiate="yes|no"?>+
    </correlations>
    activity
  </onMessage>
  <onAlarm (for="duration-expr" | until="deadline-expr")>*
    activity
  </onAlarm>
</pick>

```

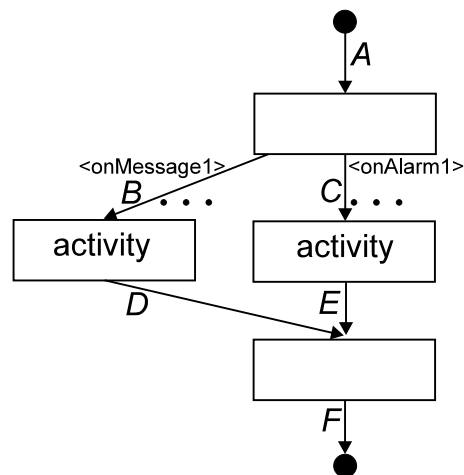


Abbildung 3.12: pick

3.4.5 flow

Aktivitäten innerhalb eines `<flow>` werden parallel ausgeführt. Innerhalb eines `<flow>` kann jedoch zusätzlich mit Hilfe von Links eine Ordnung definiert werden.

```

<flow name="ncname"? joinCondition="bool-expr"?

```

```

    suppressJoinFailure="yes|no"?>
<target linkName="ncname"/>*
<source linkName="ncname" transitionCondition="bool-expr"?/>*
<links>?
    <link name="ncname">+
</links>
    activity+
</flow>

```

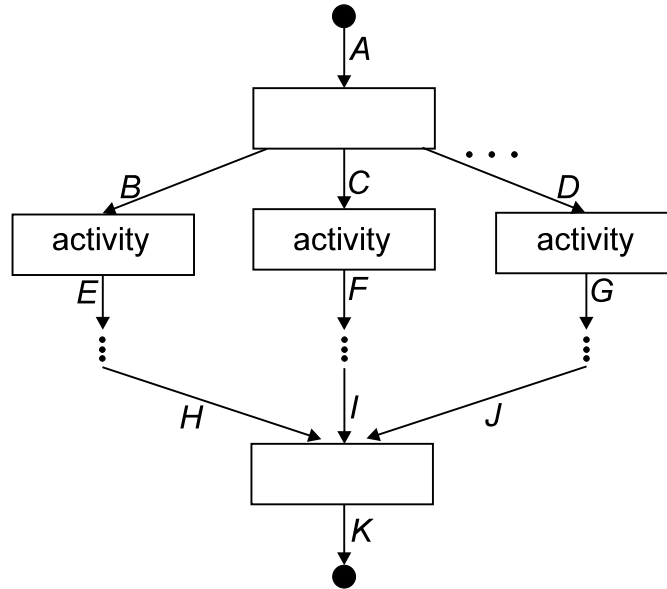


Abbildung 3.13: flow

Links

Um Aktivitäten innerhalb eines Flows zu synchronisieren, können Links definiert werden. Ein Link beginnt und endet an einer Aktivität. Dazu werden die `<source>` und `<target>` Elemente angegeben. Jedes `<source>` Element kann zusätzlich ein `transitionCondition` Attribut definieren, das als Wächter des Links fungiert. Nur wenn die `transitionCondition` den Wert `true` hat, wird der Link verfolgt. Standardmäßig hat die `transitionCondition` den Wert `true`.

Da eine Aktivität das Ziel mehrerer Links sein kann, ist es sinnvoll, eine `joinCondition` angeben zu können. Eine Aktivität wird nur ausgeführt, wenn die Auswertung der `joinCondition` der eingehenden Links den Wert `true` liefert. Standardmäßig ist die `joinCondition` das OR aller eingehenden Links. Ergibt die Auswertung der `joinCondition` den Wert `false`, dann wird, falls `suppressJoinFailure='no'`, die normale Abarbeitung unterbrochen und der Standardfehler `bpws:joinFailure` geworfen. Ist jedoch `suppressJoinFailure='yes'` festgelegt, dann wird kein Fehler geworfen, die Aktivität wird übersprun-

gen und alle ausgehenden Links dieser Aktivität bekommen einen negativen Wert. Standardmäßig hat das Attribut `suppressJoinFailure` den Wert "no".

Um den Kontrollflußgraphen aufzubauen, werden die entsprechenden Quell- und Zielaktivitäten wie in Abbildung 3.14 dargestellt mit einem zusätzlichen Knoten versehen, an dem sich der Kontrollfluß aufspaltet oder vereinigt.



Abbildung 3.14: link

4 Nichtinitialisierte Variablen

In diesem Kapitel soll das Problem der nicht initialisierten Variablen in BPEL Prozeßmodellen durch statische Analyse gelöst werden, d.h. es sollen Prozeßmodelle dahingehend überprüft werden, ob es möglich ist eine Variable zu lesen, obwohl sie noch nicht initialisiert wurde. Dafür müssen wir wie folgt vorgehen:

Es muß eine geeignete Struktur für die Berechnung festgelegt und für den BPEL Code der dazugehörige Kontrollflußgraph aufgebaut werden, wobei für jeden Knoten eine Datenflußgleichung aufgestellt werden muß. Die Lösung des so entstandenen Gleichungssystems liefert für jeden Knoten die Menge der Variablen, parts und properties, die garantiert initialisiert wurden, so daß man an jedem Programmpunkt weiß, welche Variablen einen definierten Wert haben.

4.1 Struktur, Repräsentation der Daten

In BPEL muß jede Variable einem Typ genügen. Für eine Variable können mehrere parts und properties spezifiziert werden, wobei es möglich ist, nur einem part oder property einen Wert zuzuweisen, so daß nur teilweise initialisierte Variable entstehen können. Eine Variable ist vollständig initialisiert, wenn alle parts und properties initialisiert sind. Es muß also möglich sein, nicht nur die Variablen, sondern auch die parts und properties darstellen zu können. Ein weiteres Problem ergibt sich durch Scopes, die den Sichtbarkeitsbereich der Variable darstellen. Variable, die in einem Scope definiert wurden, sind auch in jedem inneren Scope sichtbar, umgekehrt jedoch sind Variablen eines inneren Scopes nicht in einem umschließenden Scope sichtbar. Eine Variable eines inneren Scope überdeckt eine Variable mit gleichem Namen eines umschließenden Scopes, ist also de facto eine neue Variable. Um Variablen eindeutig zu identifizieren, werden wir jede Variable in der Form *scope.variable* schreiben. Ein part einer Variablen identifizieren wir dementsprechend mit *scope.variable.part* und analog ein property mit *scope.variable.property*. Bezeichne:

$$\begin{aligned}\mathbf{Vars} &= \{\text{alle Variablen in der Form } \textit{scope.variable}\} \\ \mathbf{Parts} &= \{\text{alle parts in der Form } \textit{scope.variable.part}\} \\ \mathbf{Props} &= \{\text{alle properties in der Form } \textit{scope.variable.property}\} \\ \mathbf{All} &= \mathbf{Vars} \cup \mathbf{Parts} \cup \mathbf{Props}\end{aligned}$$

Dann eignet sich folgender Verband für die Berechnung:

$$\mathbf{V} = \{2^{\mathbf{All}}, \subseteq\}$$

4.2 Datenflußgleichungen

Im vorherigen Kapitel haben wir die acht Basisaktivitäten

`<invoke>` dient dazu, einen anderen Web Service aufzurufen.

`<receive>` empfängt eine Nachricht eines Partners.

`<reply>` sendet eine Nachricht an einen Partner.

`<assign>` weist Variablen einen Wert zu.

`<wait>` wartet eine bestimmte Zeit.

`<throw>` wirft eine Ausnahme.

`<terminate>` beendet den Prozeß sofort.

`<empty>` macht nichts.

und die fünf strukturierten Aktivitäten

`<sequence>` die Aktivitäten werden gemäß ihrer Anordnung ausgeführt.

`<switch>` ermöglicht eine datenabhängige Verzweigung des Kontrollflusses. Es wertet einen Ausdruck aus und wählt dementsprechend einen Zweig aus.

`<while>` ermöglicht die Einführung von Zyklen in den Kontrollfluß. Die eingeschlossene Aktivität wird bis zum Erreichen der Abbruchbedingung ausgeführt.

`<pick>` ermöglicht die nichtdeterministische Verzweigung abhängig von externen Nachrichten.

`<flow>` führt Aktivitäten parallel aus. Es kann jedoch zusätzlich mit Hilfe von Links eine Ordnung definiert werden.

kennengelernt.

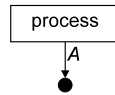
Das Problem der nichtinitialisierten Variablen unterteilt die Aktivitäten in zwei Gruppen. Die Aktivitäten `<assign>`, `<receive>`, `<pick>` und `<invoke>` können Variable manipulieren, die restlichen Aktivitäten lassen die Variablen unverändert. Von besonderem Interesse sind aber auch die Aktivitäten, die den Kontrollfluß aufspalten.

Initialisierung einer Variable ist ein Vorgang, der in der Vergangenheit stattgefunden haben muß, also ist eine *vorwärts* Analyse nötig. Da wir gesicherte Informationen brauchen, ist zudem eine *sicher* Analyse angebracht.

Im folgenden werden nun für jedes BPEL Konstrukt die Datenflußgleichungen für eine *vorwärts-sicher* Analyse angegeben.

process

Das Wurzelement eines jeden BPEL Prozeßmodells ist `<process>`. Zu Beginn ist keine Variable initialisiert, also gilt:

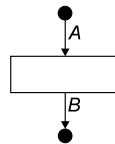


$$A = \{ \}$$

Abbildung 4.1: process

reply, wait, empty

Die Aktivitäten `<reply>`, `<wait>` und `<empty>` verändern die Menge der initialisierten Variable nicht und spalten den Kontrollfluß nicht auf. Daher ergibt sich:

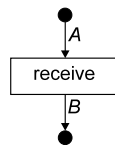


$$B = A$$

Abbildung 4.2: reply, wait, empty

receive

Bei einem `<receive>` wird eine Variable mit der empfangenen Nachricht belegt. Diese Variable wird in die Menge der initialisierten Variable aufgenommen.



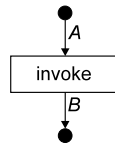
$$B = A \cup \{scope.variable\}$$

Abbildung 4.3: receive

invoke

Beim `<invoke>` muß man den synchronen und asynchronen Fall unterscheiden. Beim synchronen `<invoke>` wurde nach Ende der Aktivität eine Variable geschrieben, im asynchronen Fall jedoch nicht. Daher gilt:

4 Nichtinitialisierte Variablen

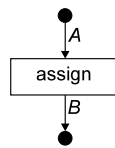


- asynchron: $B = A$
- synchron: $B = A \cup \{scope.variable\}$

Abbildung 4.4: `invoke`

assign

In dem `<copy>` Element eines `<assign>` können wir in der `<to>` Klausel angeben, ob wir einer Variable, einem part oder property einer Variable einen Wert zuweisen möchten. Abhängig von der `<to>` Angabe ergibt sich daher:

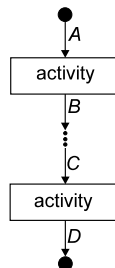


- $B = A \cup \{scope.variable\}$
- $B = A \cup \{scope.variable.part\}$
- $B = A \cup \{scope.variable.property\}$

Abbildung 4.5: `assign`

sequence

In einer `<sequence>` werden die Aktivitäten nacheinander ausgeführt, also ergibt sich:



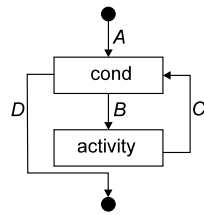
$$D = A \cup B \cup C$$

Abbildung 4.6: `sequence`

while

Zur Compilezeit läßt sich nicht entscheiden, ob die `<while>`-Schleife einmal durchlaufen wird. Da wir eine *sicher* Analyse durchführen, müssen wir davon ausgehen, daß die activity keinmal ausgeführt wird. Demnach gilt:

4 Nichtinitialisierte Variablen



$$B = A$$

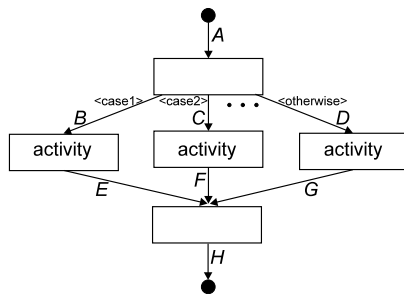
$$C = A$$

$$D = A$$

Abbildung 4.7: while

switch

Bei `<switch>` wird ein Zweig abgearbeitet, zur Compilezeit wissen wir jedoch nicht welcher. Da wir sichere Informationen brauchen, müssen wir den Durchschnitt aller Zweige nehmen.



$$B = A$$

$$C = A$$

$$D = A$$

$$H = E \cap F \cap G$$

Abbildung 4.8: switch

pick

Wie beim `<switch>` wird auch beim `<pick>` nur ein Zweig abgearbeitet. Jedoch erfolgt die Auswahl durch eine einkommende Nachricht, die einer Variablen zugewiesen wird oder durch einen ausgelösten Alarm. Wird ein `<onAlarm>` Ereignis abgearbeitet, dann ändert das `<pick>` selbst keine Variable, bei Auswahl eines `<onMessage>` Ereignisses wird jedoch eine Variable initialisiert. $variable_i$ bezeichne die Variable der `<onMessage>` Elemente. Zur Übersetzungszeit kann man sich nur dann sicher sein, daß die Variable var_i geschrieben wird, wenn $var_i = var_1 = var_2 = \dots$ und kein `<onAlarm>` Zweig definiert ist.

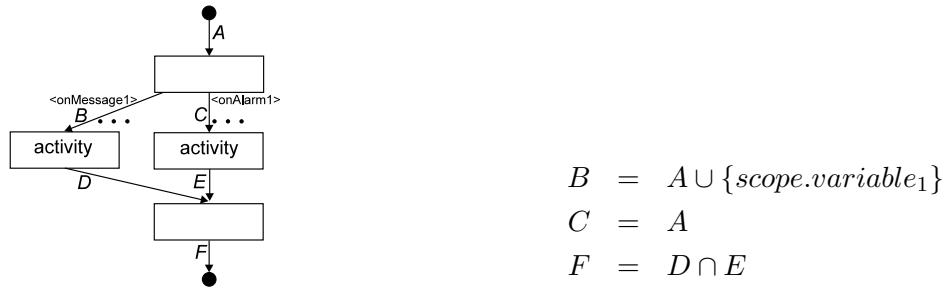


Abbildung 4.9: pick

flow

In einem `<flow>` werden die einzelnen Zweige parallel ausgeführt, daher werden wir im allgemeinen nicht von einem Zweig auf den anderen schließen können. Es ist möglich, daß ein Zweig schon abgearbeitet ist, während ein anderer noch nicht einmal begonnen hat. Abhilfe schaffen hier die Links. Ist eine Aktivität ein `<target>` eines Links, ist gesichert, daß bei Ausführung dieser Aktivität auch schon alle Aktivitäten des anderen Zweiges bis einschließlich zur `<source>`-Aktivität ausgeführt wurden. Nun müssen wir noch die zwei Fälle `suppressJoinFailure='no'|'yes'` unterscheiden.

1. `suppressJoinFailure='no'`: Wenn die `joinCondition` den Wert `'false'` annimmt, dann wird ein `bpws:joinFailure` geworfen und die Fehlerbehandlung beginnt. Da wir vom positiven Fall ausgehen und jegliche Fehlerbehandlung nicht betrachten, nehmen wir für eine sinnvolle weitere Abarbeitung den Fall `joinCondition='true'` an.
2. `suppressJoinFailure='yes'`: Im Falle `joinCondition='true'` wird die Aktivität, die das Ziel der Links ist, ausgeführt. Andernfalls wird sie übersprungen und damit unterbleibt eine eventuelle Initialisierung einer Variable. Da wir eine *sicher* Analyse durchführen, muß dieser Fall angenommen werden.

Dieser Sachverhalt ist den Abbildungen 4.10, 4.11 und 4.12 zu entnehmen. In Abbildung 4.12 bedeute f diejenige Funktion, die durch die konkrete Aktivität gegeben ist.

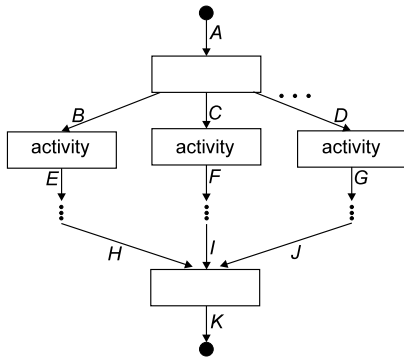
terminate

Da `<terminate>` den Prozeß lediglich beendet, werden die Variablen nicht verändert.

throw

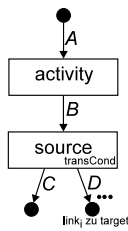
In dieser Arbeit gehen wir von einem fehlerfreien Ablauf des Business Prozesses aus und klammern daher Fehlerbehandlung aus. `<throw>` spielt also in dieser Betrachtung keine Rolle.

4 Nichtinitialisierte Variablen



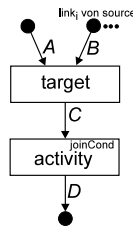
$$\begin{aligned}
 B &= A \\
 C &= A \\
 D &= A \\
 K &= H \cup I \cup J
 \end{aligned}$$

Abbildung 4.10: flow



$$\begin{aligned}
 C &= B \\
 D &= B
 \end{aligned}$$

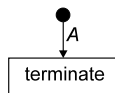
Abbildung 4.11: link, source



$$C = A \cup B$$

- suppressJoinFailure="no": $D = f(C)$
- suppressJoinFailure="yes": $D = C$

Abbildung 4.12: link, target



es entstehen keine neuen Gleichungen

Abbildung 4.13: terminate

4.3 Lösung

Nun lassen sich für jedes BPEL-Prozeßmodell, das in einen Kontrollflußgraphen mit Hilfe der Pattern aus Kapitel 3 überführt wurde, die Datenflußgleichungen aufstellen. Da ein Programm nur endlich viele Variable definiert und unsere Datenflußgleichungen auf Mengen erster Stufe elementare Operationen wie Vereinigung und Durchschnitt durchführen, sind alle Funktionen monoton und stetig. Nach dem Fixpunktsatz aus Kapitel 2 gibt es daher einen Fixpunkt des Gleichungssystems, der sich iterativ berechnen läßt. Man kann dann an jeder Stelle, an der eine Variable gelesen wird, überprüfen, ob sie in der Menge der bereits initialisierten Variable vorkommt. Es existiert somit ein konstruktives Verfahren, mit dem es möglich ist, zur Compilezeit sicherzustellen, daß auf keine nicht initialisierte Variable lesend zugegriffen wird.

5 Beispiel

In diesem Kapitel möchten wir anhand eines einfachen Geschäftsprozesses die Tauglichkeit der in dieser Arbeit beschriebenen Analysemethode demonstrieren. Betrachten wir hierzu das einleitende Beispiel aus [ACD⁺03].

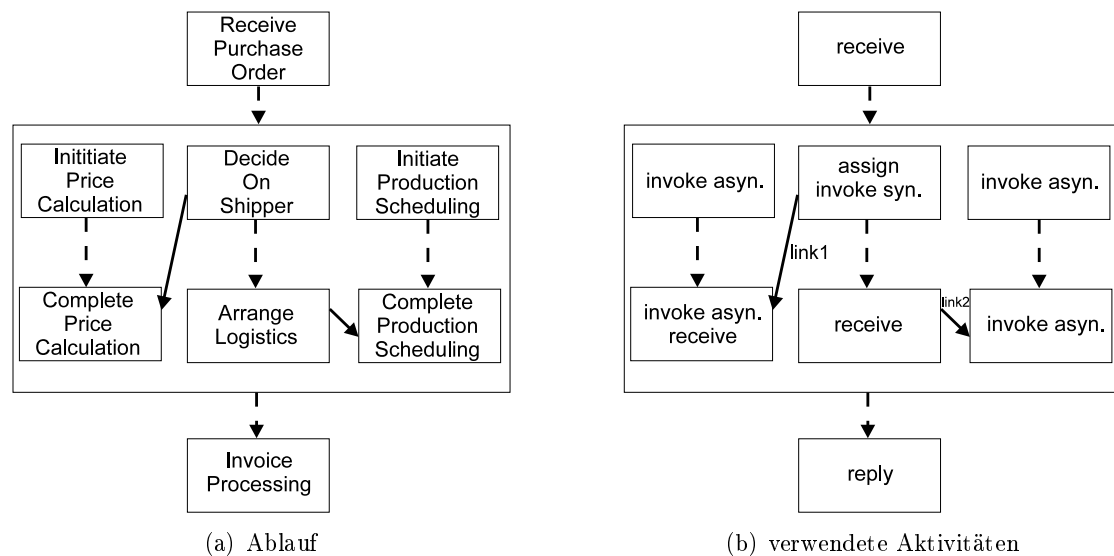


Abbildung 5.1: Purchase Order Process

Dieser Geschäftsprozeß verarbeitet die Bestellung eines Kunden. Nachdem der BPEL-Prozeß die Bestellung des Kunden erhalten hat, werden drei Aufgaben parallel initiiert: die Berechnung des Endpreises für die Bestellung, die Auswahl eines Spediteurs und die zeitliche Planung der Produktion und des Versandes. Zwischen diesen drei Aufgaben existieren Abhängigkeiten, so dass sie nicht vollständig parallel ausgeführt werden können. Um den endgültigen Preis zu berechnen, muß der Spediteur bekannt sein und für die Fertigstellung des Ausführungsplanes wird der Versandtermin benötigt. Nachdem diese Aufgaben erledigt sind, kann die Rechnung an den Kunden gesendet werden.

Im folgenden wird auszugsweise das WSDL-File angegeben, dem man die nötigen Typdefinitionen entnehmen kann.

```
<definitions
targetNamespace="http://manufacturing.org/wsd1/purchase"
  xmlns:sns="http://manufacturing.org/xsd/purchase"
  xmlns:pos="http://manufacturing.org/wsd1/purchase"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

5 Beispiel

```
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/">

<import namespace="http://manufacturing.org/xsd/purchase"
  location="http://manufacturing.org/xsd/purchase.xsd"/>

<message name="POMessage">
  <part name="customerInfo" type="sns:customerInfo"/>
  <part name="purchaseOrder" type="sns:purchaseOrder"/>
</message> <message name="InvMessage">
  <part name="IVC" type="sns:Invoice"/>
</message> <message name="orderFaultType">
  <part name="problemInfo" type="xsd:string"/>
</message> <message name="shippingRequestMessage">
  <part name="customerInfo" type="sns:customerInfo"/>
</message> <message name="shippingInfoMessage">
  <part name="shippingInfo" type="sns:shippingInfo"/>
</message> <message name="scheduleMessage">
  <part name="schedule" type="sns:scheduleInfo"/>
</message>

<!-- portTypes supported by the purchase order process -->
...

<!-- portType supported by the invoice services -->
...

<!-- portType supported by the shipping service -->
...

<!-- portType supported by the production scheduling process -->
...

<!-- partnerLinkTypes -->
...

</definitions>
```

Hier nun der BPEL-Code des Purchase Order Processes:

```
<process name="purchaseOrderProcess"
  targetNamespace="http://acme.com/ws-bp/purchase"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:lns="http://manufacturing.org/wsdl/purchase">

  <partnerLinks>
    <partnerLink name="purchasing"
      partnerLinkType="lns:purchasingLT"
```

5 Beispiel

```
        myRole="purchaseService"/>
    <partnerLink name="invoicing"
        partnerLinkType="lns:invoicingLT"
        myRole="invoiceRequester"
        partnerRole="invoiceService"/>
    <partnerLink name="shipping"
        partnerLinkType="lns:shippingLT"
        myRole="shippingRequester"
        partnerRole="shippingService"/>
    <partnerLink name="scheduling"
        partnerLinkType="lns:schedulingLT"
        partnerRole="schedulingService"/>
</partnerLinks>

<variables>
    <variable name="PO" messageType="lns:POMessage"/>
    <variable name="Invoice"
        messageType="lns:InvMessage"/>
    <variable name="POFault"
        messageType="lns:orderFaultType"/>
    <variable name="shippingRequest"
        messageType="lns:shippingRequestMessage"/>
    <variable name="shippingInfo"
        messageType="lns:shippingInfoMessage"/>
    <variable name="shippingSchedule"
        messageType="lns:scheduleMessage"/>
</variables>

<faultHandlers>
    <catch faultName="lns:cannotCompleteOrder"
        faultVariable="POFault">
        <reply partnerLink="purchasing"
            portType="lns:purchaseOrderPT"
            operation="sendPurchaseOrder"
            variable="POFault"
            faultName="cannotCompleteOrder"/>
    </catch>
</faultHandlers>

<sequence>

    <receive partnerLink="purchasing"
        portType="lns:purchaseOrderPT"
        operation="sendPurchaseOrder"
        variable="PO">

    </receive>

</flow>
```

5 Beispiel

```
<links>
  <link name="ship-to-invoice"/>
  <link name="ship-to-scheduling"/>
</links>

<sequence>
  <assign>
    <copy>
      <from variable="P0" part="customerInfo"/>
      <to variable="shippingRequest"
        part="customerInfo"/>
    </copy>
  </assign>

  <invoke partnerLink="shipping"
    portType="lns:shippingPT"
    operation="requestShipping"
    inputVariable="shippingRequest"
    outputVariable="shippingInfo">
    <source linkName="ship-to-invoice"/>
  </invoke>

  <receive partnerLink="shipping"
    portType="lns:shippingCallbackPT"
    operation="sendSchedule"
    variable="shippingSchedule">
    <source linkName="ship-to-scheduling"/>
  </receive>

</sequence>

<sequence>

  <invoke partnerLink="invoicing"
    portType="lns:computePricePT"
    operation="initiatePriceCalculation"
    inputVariable="P0">
  </invoke>

  <invoke partnerLink="invoicing"
    portType="lns:computePricePT"
    operation="sendShippingPrice"
    inputVariable="shippingInfo">
    <target linkName="ship-to-invoice"/>
  </invoke>

  <receive partnerLink="invoicing"
    portType="lns:invoiceCallbackPT"
    operation="sendInvoice"
    variable="Invoice"/>


```

5 Beispiel

```
</sequence>

<sequence>
  <invoke partnerLink="scheduling"
    portType="lns:schedulingPT"
    operation="requestProductionScheduling"
    inputVariable="PO">
  </invoke>
  <invoke partnerLink="scheduling"
    portType="lns:schedulingPT"
    operation="sendShippingSchedule"
    inputVariable="shippingSchedule">
    <target linkName="ship-to-scheduling"/>
  </invoke>
</sequence>
</flow>

<reply partnerLink="purchasing"
  portType="lns:purchaseOrderPT"
  operation="sendPurchaseOrder"
  variable="Invoice"/>
</sequence>

</process>
```

Dem BPEL- und WSDL-Code entnimmt man:

Vars = {*PO*, *Invoice*, *POFault*, *shippingRequest*, *shippingInfo*, *shippingSchedule*}
Parts = {*PO.customerInfo*, *PO.purchaseOrder*, *Invoice.IVC*,
POFault.problemInfo, *shippingRequest.customerInfo*,
shippingInfo.shippingInfo, *shippingSchedule.schedule*}
Props = {}
All = **Vars** \cup **Parts** \cup **Props**

Womit sich folgender Verband ergibt:

$$\mathbf{V} = \{2^{\mathbf{All}}, \subseteq\}$$

Aus dem BPEL-Code läßt sich mit Hilfe der Pattern aus Kapitel 3 der Kontrollflußgraph aus Abbildung 5.2 aufbauen und die zugehörigen Datenflußgleichungen angeben:

5 Beispiel

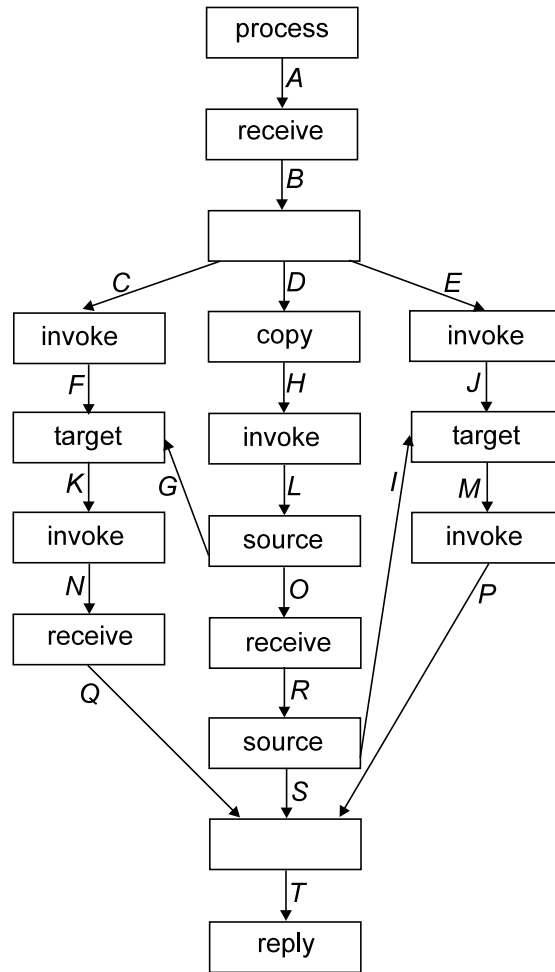


Abbildung 5.2: Kontrollflußgraph Purchase Order Process

- $A = \{ \}$
- $B = A \cup \{PO\}$
- $C = B$
- $D = B$
- $E = B$
- $F = C$
- $G = L$
- $H = D \cup \{shippingRequest.customerInfo\}$
- $I = R$
- $J = E$

5 Beispiel

$$\begin{aligned}K &= F \cup G \\L &= H \cup \{shippingInfo\} \\M &= I \cup J \\N &= K \\O &= L \\P &= M \\Q &= N \cup \{Invoice\} \\R &= O \cup \{shippingSchedule\} \\S &= R \\T &= P \cup Q \cup S\end{aligned}$$

Die Lösung des Gleichungssystems ist:

$$\begin{aligned}A &= \{\} \\B &= \{PO\} \\C &= \{PO\} \\D &= \{PO\} \\E &= \{PO\} \\F &= \{PO\} \\G &= \{PO, shippingRequest.customerInfo, shippingInfo\} \\H &= \{PO, shippingRequest.customerInfo\} \\I &= \{PO, shippingRequest.customerInfo, shippingInfo, shippingSchedule\} \\J &= \{PO\} \\K &= \{PO, shippingRequest.customerInfo, shippingInfo\} \\L &= \{PO, shippingRequest.customerInfo, shippingInfo\} \\M &= \{PO, shippingRequest.customerInfo, shippingInfo, shippingSchedule\} \\N &= \{PO, shippingRequest.customerInfo, shippingInfo\} \\O &= \{PO, shippingRequest.customerInfo, shippingInfo\} \\P &= \{PO, shippingRequest.customerInfo, shippingInfo, shippingSchedule\} \\Q &= \{PO, shippingRequest.customerInfo, shippingInfo, Invoice\} \\R &= \{PO, shippingRequest.customerInfo, shippingInfo, shippingSchedule\} \\S &= \{PO, shippingRequest.customerInfo, shippingInfo, shippingSchedule\} \\T &= \{PO, shippingRequest.customerInfo, shippingInfo, shippingSchedule, Invoice\}\end{aligned}$$

In Abbildung 5.3 sind die Variablen an den Stellen angegeben, an denen auf sie lesend zugegriffen wird. Es ist nun leicht ersichtlich, daß nur initialisierte Variable gelesen werden.

5 Beispiel

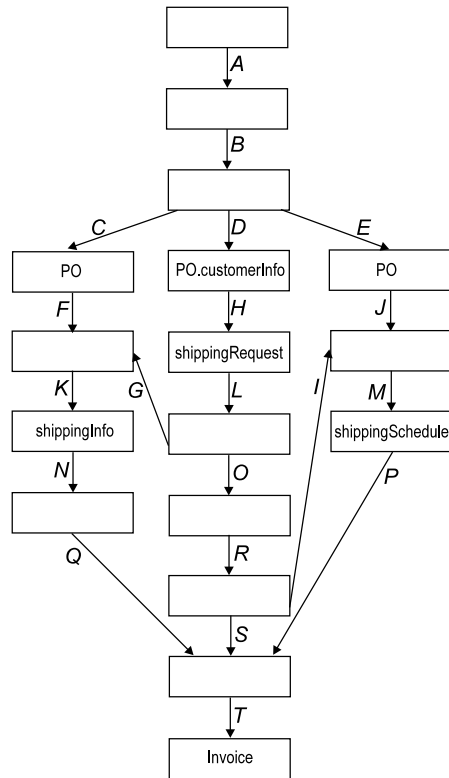


Abbildung 5.3: Lesender Zugriff auf Variablen

6 Zusammenfassung

In dieser Arbeit haben wir eine Methode angegeben, wie sich die statische Programm-analyse auf BPEL4WS Prozeßmodelle anwenden läßt. Dazu haben wir im Kapitel 2 einen Überblick über die theoretischen Grundlagen gegeben. Hier sei noch einmal auf den Begriff des Verbandes und den Fixpunktsatz als die Basis unserer Berechnung hingewiesen.

In Kapitel 3 wurde eine für unsere Zwecke ausreichende Einführung in die wesentlichen Merkmale der Sprache BPEL4WS gegeben und für jedes BPEL-Konstrukt ein Pattern angegeben, mit deren Hilfe sich ein Kontrollflußgraph eines BPEL-Codes aufbauen läßt.

Im darauf folgenden Kapitel 4 haben wir uns schließlich mit dem Problem der nicht initialisierten Variablen in BPEL-Prozeßmodellen beschäftigt. Dazu haben wir einen Verband festgelegt und Datenflußgleichungen für jede BPEL-Aktivität angegeben, mit deren Hilfe sich das Problem der nicht initialisierten Variable zur Compilezeit lösen läßt.

Die Anwendbarkeit der entwickelten Methode wurde im Kapitel 5 anhand eines Beispiels demonstriert.

Abschließend läßt sich sagen, daß sich die statische Programmanalyse erfolgreich auf BPEL4WS-Prozeßmodelle anwenden läßt. Die in Kapitel 3 angegebenen Pattern sind universell. Es lassen sich andere Anforderungen finden, die sich ebenfalls durch eine statische Analyse sicherstellen lassen. Als Beispiel sei hier nur die Forderung genannt, daß ein `<reply>` ein vorangehendes zugehöriges `<receive>` haben muß. Dazu müssen natürlich ein anderer Verband und andere Datenflußgleichungen aufgestellt werden. Gegenstand weiterer Arbeit könnte die Einbindung erweiterter Konzepte wie Fehler- und Kompensationsbehandlung und die Implementation der Analysemethode sein.

Literaturverzeichnis

- [ACD⁺03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. *Business Process Execution Language for Web Services, Version 1.1*. IBM, BEA, Microsoft, SAP, Siebel Systems, 5 May 2003.
- [ASU99a] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau Teil1*. Oldenbourg, 1999.
- [ASU99b] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau Teil2*. Oldenbourg, 1999.
- [bpe03] Bpel for web services java run time (bpws4j). <http://www.alphaworks.ibm.com/tech/bpws4j>, April 2003.
- [cB93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the International Conference on Formal Methods in Programming and their Applications*, Lecture Notes in Computer Science 735, pages 128–141. Springer-Verlag, June 1993.
- [CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.
- [CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
- [CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.
- [CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

Literaturverzeichnis

- [Cou96] P. Cousot. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM Computing Surveys*, 28(2):324–328, June 1996.
- [NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.
- [Sch] Michael I. Schwartzbach. *Lecture Notes on Static Analysis*. BRICS, Department of Computer Science, University of Aarhus, Denmark.
- [WC02] Sanjiva Weerawarana and Francisco Curbera. Understanding bpm4ws, part1. Technical report, IBM, August 2002.