# Benutzer Handbuch

**Backtracking für intelligente Softwareagenten**

**Haitzer Thomas 0325781**
**Parapatics Peter 0225859**
**Sverak Sascha 0325870**

# Inhaltsverzeichnis

## Installationsanweisungen

Die Datei *release.zip* einfach in einen Ordner extrahieren. Man sieht nun die Datei *binary.zip* und die Datei *src.zip*.
*src.zip* enthält den gesamten Quellcode der Anwendung.
Für uns interessant ist die Datei *binary.zip*. Diese einfach in einen Ordner ihrer Wahl extrahieren und fertig. Gegebenenfalls müssen die Berechtigungen der Scripts welche im Unterordner *scripts* zu finden sind, so gesetzt werden, dass sie ausführbar sind.

Sämtliche benötigte Aufrufe um die einzelnen Programmteile zu starten sind im Testdurchlauf zu finden.

Nähere Informationen zu den einzelnen Programmteilen sind im Abschnitt „Die Programme“ zu finden.

Zum Ausführen der Programme ist ein Java Runtime Environment(/Java Developement Kit) in Version 1.5 notwendig. Zum kompilieren der Source-Dateien ist ein Java Developement Kit notwendig (ebenfalls Version 1.5). Zum kompilieren unter Linux/Unix einfach das src.zip entpacken und mittels **make** kompilieren. Dazu muss im makefile die Variable JAVADIR auf das Verzeichnis des Sun-JDKs gesetzt werden.

Unter Windows können die Dateien mit dem Befehl
*javac -target 1.5  -cp .\src -d .\bin .\src\planlibrary*.java*
kompiliert werden. Dazu muss der Ordner **bin** existieren.

# Anforderungen an den Benutzer

### *DTD*

Der Benutzer ist selbst dafür verantwortlich, dass alle selbst erstellten Dateien, darunter fallen

- Condition-Files
- Planlibrary-Files
- Reverseplan-Files
- Search-Files

die passenden DTDs inkludiert haben. Sollten diese DTDs nicht vorhanden sein, kann dies zu Fehlermeldungen führen.

Die DTDs sind im Unterverzeichnis „DTD“ zu finden.

1. **Doctype-Definition für DNF-Condition-Files:**

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT conditions (phi,psi)?>
<!ELEMENT phi (or|and|not|term)>
<!ELEMENT psi (or|and|not|term)>


<!ELEMENT or (and|not|term)+>
<!ELEMENT and (not|term)+>
<!ELEMENT not (term)>
<!ELEMENT term (#PCDATA)>
```

2. **Doctype-Definition für Condition-Files:**

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT conditions (phi,psi)>
<!ELEMENT phi (all|exists|or|and|not|term)>
<!ELEMENT psi (all|exists|or|and|not|term)>

<!ELEMENT all (all|exists|or|and|not|term)>
<!ATTLIST all quantifier CDATA #REQUIRED>

<!ELEMENT exists (all|exists|or|and|not|term)>
<!ATTLIST exists quantifier CDATA #REQUIRED>

<!ELEMENT or (all|exists|or|and|not|term)+>
<!ELEMENT and (all|exists|or|and|not|term)+>
<!ELEMENT not (all|exists|or|and|not|term)>
<!ELEMENT term (#PCDATA)>
```

3. **Doctype-Definition für Planlibrary-Files:**

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT plan-library (plan*)>
<!ELEMENT plan (action-sequence, conditions+)>
<!ATTLIST plan id CDATA #REQUIRED>

<!ELEMENT action-sequence (action*,concurrent-action*)*>
<!ELEMENT action (param*)>
<!ATTLIST action name CDATA #REQUIRED>

<!ELEMENT concurrent-action (action+)>

<!ELEMENT param (#PCDATA)>
<!ELEMENT conditions (phi,psi,revplans)>
<!ELEMENT phi (all|exists|or|and|not|term)>
<!ELEMENT psi (all|exists|or|and|not|term)>

<!ELEMENT all (all|exists|or|and|not|term)>
<!ATTLIST all quantifier CDATA #REQUIRED>

<!ELEMENT exists (all|exists|or|and|not|term)>
<!ATTLIST exists quantifier CDATA #REQUIRED>

<!ELEMENT or (all|exists|or|and|not|term)+>
<!ELEMENT and (all|exists|or|and|not|term)+>
<!ELEMENT not (all|exists|or|and|not|term)>
<!ELEMENT term (#PCDATA)>

<!ELEMENT revplans (action-sequence+)>
```

## 4. Doctype-Definition für Reverseplan-Files:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT reverse-plans (plan*)>
<!ELEMENT plan (action-sequence)>
<!ATTLIST plan
            id CDATA #REQUIRED
>

<!ELEMENT action-sequence (action+)>
<!ELEMENT action (param*)>
<!ATTLIST action
            name CDATA #REQUIRED
>
<!ELEMENT param (#PCDATA)>
```

## 5. Search-Files

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT search-plan (action-sequence, conditions)>
<!ATTLIST search-plan id CDATA #IMPLIED>

<!ELEMENT action-sequence (action*, concurrent-action*)*>

<!ELEMENT concurrent-action (action+)>

<!ELEMENT action (param*)>
<!ATTLIST action name CDATA #REQUIRED
>
<!ELEMENT param (#PCDATA)>

<!ELEMENT conditions (pi+)>
<!ELEMENT pi (all|exists|or|and|not|term)>

<!ELEMENT all (all|exists|or|and|not|term)>
<!ATTLIST all quantifier CDATA #REQUIRED>

<!ELEMENT exists (all|exists|or|and|not|term)>
<!ATTLIST exists quantifier CDATA #REQUIRED>

<!ELEMENT or (all|exists|or|and|not|term)+>
<!ELEMENT and (all|exists|or|and|not|term)+>
<!ELEMENT not (all|exists|or|and|not|term)>
<!ELEMENT term (#PCDATA)>
```

### *DNF (Conditions)*

Die einzelnen Conditions die im Programm benötigt werden, sei es für die DLV-Transformation, für das Eintragen in die Planlibrary oder für das Suchen von Reverseplans sind immer in DNF zu verfassen.

Andere Formate werden nicht unterstützt und müssen, falls gewollt, selbst implementiert werden.

### *Conditions-Files*

Es gibt meistens zwei Arten die Conditions-Files anzugeben. In „Textform“ und in „XML-Form“.

**Bsp Textform**

```
PHI: AND(X,Y)
PSI: OR(x,y)
```

**Bsp XML-Form**

```
-- Siehe Testdurchlauf –
```

## Syntax für Formeln in Textfiles

Formeln in Textfiles müssen in Prefix Notation spezifiziert werden.
Die Syntax dafür sieht folgendermaßen aus:

Ein unärer Operator wird folgendermaßen verwendet:
*UNÄRER_OPERATOR(SUBFORMEL)*
Mögliche unäre Operatoren: NOT

Ein binärer Opertor wird folgendermaßen verwendet:
*BINÄRER_OPERATOR(Subformel1, Subformel2,..., SubformelN)*
Mögliche binäre Operatoren sind: AND, OR

Ein Quantor kann folgendermaßen verwendet werden:
*QUANTOR VARIABLE (Subformel)*
Mögliche Quantoren: EXISTS, ALL

Eine Subformel kann folgende Struktur haben:
Entweder sie geht wieder in ein Konstrukt aus Operator und Subformel über oder sie geht in einen Term über.

Der Term hat folgende Syntax:
*termname* --- wenn der Term kein Prädikat ist (z.B. a,b,….)
*[termname(param1,param2)]* --- wenn der Term ein Prädikat ist (z.B. throw(a,b,c)); wichtig dabei ist, dass der Term in [ ] steht

# Syntax für Formeln in XML-Files

Die Formeln für XML-Files unterteilen sich in

- Terme
- Unäre Operatoren
- Binäre Operatoren
- Quantoren

In den folgenden Beispielen ist der Begriff „SUBFORMEL“ als Platzhalter zu sehen, der entweder wieder in ein Konstrukt aus Operator und Subformel oder in einen Term übergeht.

**Terme**
Um einen Term innerhalb eines Operators zu verwenden benötigt er folgende Syntax:

```
<term> TERM </term>
```

Als „TERM“ versteht man hier Ausdrücke wie z.B. „on(a,b)“ oder „supported(x)“.

**Unäre Operatoren („NOT“)**

```
<not>
     SUBFORMEL
</not>
```

**Binäre Operatoren („AND“, „OR“)**
Die binären Operatoren „and“ und „or“ müssen mindestens zwei, können aber beliebig viele Subformeln enthalten.

```
<and>
     SUBFORMEL
     SUBFORMEL
</and>

<or>
     SUBFORMEL
     SUBFORMEL
</or>
```

**Quantoren („ALL“, „EXISTS“)**

```
<all quantifier="VARIABLE">
     SUBFORMEL
</all>

<exists quantifier="VARIABLE">
     SUBFORMEL
</exists>
```

# Syntax für Pläne in XML-Files

Die Pläne für XML-Files lassen sich in folgende Teilbereiche unterteilen:

- Action Sequence
- Concurrent Actions
- Actions

**Action Sequence**
Der Tag action-sequence kennzeichnet den Anfang und das Ende eines Plans.

„SUBACTIONS“ steht hier als Platzhalter für eine beliebige Anzahl an concurrent-actions oder actions.

```
<action-sequence>
     SUBACTIONS
</action-sequence>
```

**Concurrent Actions**
Der Tag concurrent-action kennzeichnet den Anfang und das Ende einer concurrent-action.

„ACTIONS“ steht hier als Platzhalter für eine beliebige Anzahl an actions.

```
<concurrent-action>
     ACTIONS
</concurrent-action>
```

**Actions**
Eine action besteht aus dem Namen der Action und einer beliebigen Anzahl an Parametern.

```
<action name="NAME">
     <param> PARAMETER1 </param>
     <param> PARAMETER2 </param>
</action>
```

# Die Programme

Hier folgt eine kurze Beschreibung der einzelnen Programme, was sie tun, welche Inputs sie benötigen und welche Optionen sie gegebenenfalls haben.

## *ReverseDomain*

Dieser Programmteil dient zur Transformation der DLV-K Domäne in die Reverseplan-Domäne.

**Aufruf**

```
planlibrary.ReverseDomain (-t|-x) .plan-File .dl-File cond-File
      path-to-dlv planlength
```

`(-t|-x)` dieser Parameter steht für das Format des Conditions-File
`-t` besagt, dass das Conditions-File in „Text-Form“ vorliegt und
`-x` besagt, dass das Conditions-File in „XML-Form“ vorliegt

`.plan-File` das File enthält die Domainbeschreibung

`.dl-File` das File enthält das Hintergrundwissen

`cond-File` das File enthält die Conditions, entweder in Text oder in XML format

`path-to-dlv` hier muss man den Pfad zu DLV angeben

`planlength` dieser Parameter steht für die gewünschte Planlänge
eine Planlänge von 3 wäre für einen Vorwärtsschritt mit einem Rückwärtsschritt

Syntax der Input-Files:
Das cond-file muss der conditions-dnf.dtd aus dem DTD Verzeichnis genügen, falls es als .xml File angegeben wird.

Falls es als Textfile angegeben sieht sie Syntax folgenermaßen aus:

PHI: gültige Formel (siehe dazu den Abschnitt **Syntax für Formeln in Textfiles**)
PSI: gültige Formel (siehe dazu den Abschnitt **Syntax für Formeln in Textfiles**)

## *Library*

Dieser Programmteil fügt bereits generierte Pläne zur Planlibrary hinzu. Sollte das Planlibrary-XML-File anfangs nicht vorhanden sein, wird es vom Programm angelegt.

**Aufruf**

```
planlibrary.Library dlv-plans cond-file planlibrary min,max min,max
        (add|delete)
```

`dlv-plans` plan-file das die generierten Pläne enthält

`cond-file` conditions-file in xml-form

`planlibrary` .xml File welches die Planlibrary enthält

`min,max` (erstes Vorkommen) spezifiziert die maximale und die minimale Anzahl an Vorwärtsschritten für die Pläne, die in die Library aufgenommen werden sollen. (max auf -1 setzen für beliebig viele)

***Beispiel:***
1,2 um folgende Beispielpläne in die Library aufzunehmen:
throw(x,y,z);sep;[…]
move(a,b); throw(x,y,z); sep; […]

`min,max` (zweites Vorkommen) spezifiziert die maximale und die minimale Anzahl an Rückwärtsschritten für die Pläne, die in die Library aufgenommen werden sollen. (max auf -1 setzen für beliebig viele)

***Beispiel:***
1,2 um folgende Beispielpläne in die Library aufzunehmen:
[…]; sep; move(x,y);
[…]; sep; move(a,b); move(x,y,z);

`(add|delete)` `add` fügt die gegebenen Pläne in die Library hinzu
`delete` löscht die bereits vorhandene Library und ersetzt sie durch die neuen Pläne

Syntax der Input-Files:
Das cond-file muss der conditions.dtd aus dem DTD Verzeichnis genügen.
Das planlibrary File muss entweder leer oder der planlibrary.dtd genügen.

## *Search*

Dieser Programmteil befasst sich mit dem Suchen von Reverseplans für einen gegebenen Plan.

**Aufruf**

```
planlibrary.Search (-t|-x) (all|k) planlibrary proofer pathToProofer
      inputFile (-xml-output|-text-output) [outputFile]
```

`(-t|-x)` `-t` falls das Inputfile in Textform vorliegt
`-x` falls das Inputfile in XML-Form vorliegt

`(all|k)` `all` falls man alle Reverseplans haben möchte
`k` falls man nur k-viele Reverseplans haben möchte

`planlibrary` xml-File welches die Planlibrary enthält

`proofer` Zahlenwert welcher den Proofer symbolisiert. In unserem Fall 1 für Otter

`pathtoProofer` Pfad zum gewählten Proofer

`inputFile` Inputfile welches den zu suchenden Plan mit Conditions enthält

`(-xml-output|-text-output)` -xml-output um die Pläne in xml-form auszugeben
-text-output um die Pläne in text-form auszugeben

`outputFile` angeben falls die Reversepläne in eine Dateigespeichert werden sollen. Falls nicht angegeben wird alles am Bildschirm ausgegeben

Syntax der Input-Files:
Falls der zu suchende Plan als .xml File angegeben wird muss dieser der „search.dtd“ Datei aus dem DTD Verzeichnis genügen.

Falls der Input als Text File angegeben wird muss das File die folgende Form haben:

PLAN: action(param1,…paramN);action2(param1,…paramN);action3
PI: gültige Formel (siehe dazu den Abschnitt **Syntax für Formeln in Textfiles**)
PI: gültige Formel (siehe dazu den Abschnitt **Syntax für Formeln in Textfiles**)
PI: gültige Formel (siehe dazu den Abschnitt **Syntax für Formeln in Textfiles**)

*Ad PI)*
Jedes PI steht für einen Zustand, wobei PI0 den Zustand vor der Ausführung des Planes angibt. PI1 gibt dann den Zustand nach der Ausführung der 1. Action an. PIn gibt also den Zustand nach Ausführung der n. Action an.
Aus diesem Grund muss immer ein PI mehr geben als Actions im Plan.

Sollten die Zustände zwischen den einzelnen Actions nicht bekannt sein, so sollten die unbekannten PIs auf „true“ gesetzt werden.

# Testdurchlauf 1

In diesem Abschnitt beschäftigen wir uns mit einem Testdurchlauf durch das Programm. Wir widmen uns den drei ausführbaren Programmteilen, welche die ReverseDomain erstellen, die Library ins Programm laden und ReversePläne suchen.
Bei sämtlichen angegebenen Dateien ist immer der Pfad dorthin anzugeben, da in den Testfällen davon ausgegangen wird, dass die Dateien im selben Verzeichnis wie das Programm liegen.
Sämtliche Befehle gehen davon aus, dass man sich bereits im passenden Verzeichnis befindet.

Es wird Schritt für Schritt eine Planlibrary aufgebaut. Um einzelne Schritte zu überspringen stehen die Zwischenergebnisse der einzelnen Schritte in Unterordnern des Ordners **complexToilet** zur Verfügung.

Im Ordner demo/complexToilet/**ReverseDomain** befinden sich die in den ReverseDomain Schritten generierten .out Files. Das File bsp.out_cond wird im Zuge des Schritts ***ReverseDomain(1)*** das bsp.out_conditions wird im Zuge des Schritts ***ReverseDomain(2)*** erstellt.

Im Ordner demo/complexToilet/**Library** befinden sich die Files planlibrary_bsp.out_conditions.xml, planlibrary_bsp.out_cond.xml und planlibrary.xml.

planlibrary_bsp.out_conditions.xml … enthält NUR die Pläne aus bsp.out_conditions.xml
planlibrary_bsp.out_cond.xml ... enthält NUR die Pläne aus bsp.out_cond
planlibrary.xml ... enthält die Pläne aus bsp.out_cond UND bsp.out_conditions

Im Ordner demo/complexToilet/**Search** befinden sich die Dateien planlibrary.xml und search.txt. Die Datei planlibrary.xml ist dieselbe wie die aus dem Ordner Library. Sie enthält die Pläne aus bsp.out_cond UND bsp.out_conditions. Es kann in diesem Ordner mittels search.txt gleich der Schritt ***Search*** durchgeführt werden, ohne vorher den gesamten Durchlauf machen zu müssen

## *ReverseDomain(1)*

Dieser Programmteil beschäftigt sich mit der DLV-K Domäne.
Die DLV-K Domäne wird hier in die ReversePlan-Domäne transformiert.

Bevor man diesen Programmteil benutzen kann benötigt man…

- Ein .plan File, welches die Domänbeschreibung beinhaltet
- Ein .dl File, welches das Hintergrundwissen beinhaltet
- Ein Conditions-File, welches die Conditions Phi und Psi enthält
- DLV

***Beispiel***

**demo/complexToilet/bsp.plan**

```
%     This example is based on the "bomb in the toilet"-setting,
%     where there exist a number of packages and a number of toilets.
%     Each of the packages can hold a bomb (be armed) but does not have to.
%     When a package is dunked into a toilet, the toilet is clogged and has
%     to be flushed before another package can be put into it.
%
%     The goal is a safe state, where no package is armed. In this special
```

```
%     version there are 4 actions allowed to reach this goal.
%
%     The actions dunk, flush,take, and schupf:
%           - dunk:     places a package in a toilet, this only works if
%                       the toilet is not clogged
%           - flush:    flushs a toilet; it is -clogged anymore and a
%                       package in the toilet is disarmed
%           - take:     remove a package from a toilet; it is –clogged
%                       anymore and the package is returned to the packpile
%           - schupf:   the package is thrown into a toilet; This action is
%                       indeterministic, it can result in different states.
%                       The package can be on every possible location after
%                       this action(one of the toilets, the packpile).
%                       this action ignores if a toilet is clogged or not!


fluents :
      clogged(T) requires toilet(T).
      armed(P) requires package(P).
      in(P,T) requires package(P), toilet(T).
      packpile(P) requires package(P).
      unsafe.

actions :
      dunk(P,T) requires package(P), toilet(T).
      schupf(P,T) requires package(P), toilet(T).
      flush(T) requires toilet(T).
      take(P) requires package(P).

always :
      executable flush(T).
      executable dunk(P,T) if not clogged(T), packpile(P).
      executable schupf(P,T) if packpile(P).
      executable take(P).

      caused in(P,T) after dunk(P,T).
      caused -armed(P) after flush(T), in(P,T).
      caused -in(P,T) after flush(T), in(P,T).
      caused -clogged(T) after flush(T).
      caused packpile(P) after take(P).
      caused -in(P,T) after take(P).
      caused -clogged(T) after take(P), in(P,T).
      caused clogged(T) if in(P,T).

      caused -packpile(P) if in(P,T).
      caused unsafe if armed(P).

      total in(P,T) after schupf(P,T).
      total packpile(P) after schupf(P,T).

      forbidden in(P,T1), in(P,T2), T1<>T2.
      forbidden in(P,T), packpile(P).

      noConcurrency.

      inertial armed(P).
      inertial -armed(P).
      inertial clogged(T).
      inertial -clogged(T).
      inertial packpile(T).
      inertial -packpile(T).
      inertial in(P,T).
      inertial -in(P,T).
```

```
initially :
      -clogged(t2).
      armed(p1).
      packpile(p1).
      -in(p1,t2).
goal:
      not unsafe?
```

**demo/complexToilet/bsp.dl File**

```
      toilet(t1).
      toilet(t2).
      package(p1).
      package(p2).
```

**demo/complexToilet/cond.xml**

```
      <?xml version="1.0" encoding="UTF-8"?>
      <!DOCTYPE conditions SYSTEM "../DTD/conditions.dtd">
      <conditions>
            <phi>
                  <term> true </term>
            </phi>
            <psi>
                  <and>
                        <term> -clogged(t1) </term>
                        <term> -clogged(t2) </term>
                        <term> packpile(p1) </term>
                        <term> packpile(p2) </term>
                        <term> armed(p1) </term>
                        <term> armed(p2) </term>
                  </and>
            </psi>
      </conditions>
```

**Aufruf**

```
      java –cp ./plan-library-binary.jar planlibrary.ReverseDomain
            –x demo/complexToilet/bsp.plan demo/complexToilet/bsp.dl
            demo/complexToilet/cond.xml /dlv 3
```

`-x` steht für das Format des Conditions-Files. In diesem Fall ist das Conditions-File im XML-Format gegeben

`/dlv` steht für den Pfad zum DLV-Binary

`3` steht für die Planlänge z.B.: 3 wäre *eine* Vorwärtsaktion und *eine* Reverse-Aktion (eine Aktion geht immer durch das „sep“ – den Umkehrschritt „verloren“)

**Output (wird in unserem Fall in die Datei „bsp.out“ geschrieben)**

```
      PLAN: dunk(p2,t2); sep; take(p2)
      PLAN: dunk(p1,t2); sep; take(p1)
      PLAN: flush(t2); sep; flush(t2)
      PLAN: flush(t2); sep; flush(t1)
      PLAN: flush(t2); sep; take(p2)
      PLAN: flush(t2); sep; take(p1)
      PLAN: flush(t2); sep; (no action)
      PLAN: flush(t1); sep; flush(t2)
      PLAN: flush(t1); sep; flush(t1)
      PLAN: flush(t1); sep; take(p2)
      PLAN: flush(t1); sep; take(p1)
      PLAN: flush(t1); sep; (no action)
      PLAN: take(p2); sep; flush(t2)
      PLAN: take(p2); sep; flush(t1)
      PLAN: take(p2); sep; take(p2)
      PLAN: take(p2); sep; take(p1)
      PLAN: take(p2); sep; (no action)
```

```
PLAN: take(p1); sep; flush(t2)
PLAN: take(p1); sep; flush(t1)
PLAN: take(p1); sep; take(p2)
PLAN: take(p1); sep; take(p1)
PLAN: take(p1); sep; (no action)
PLAN: schupf(p2,t2); sep; take(p2)
PLAN: schupf(p1,t2); sep; take(p1)
PLAN: (no action); sep; flush(t1)
PLAN: schupf(p2,t1); sep; take(p2)
PLAN: schupf(p1,t1); sep; take(p1)
PLAN: (no action); sep; take(p2)
PLAN: dunk(p2,t1); sep; take(p2)
PLAN: dunk(p1,t1); sep; take(p1)
PLAN: (no action); sep; take(p1)
PLAN: (no action); sep; flush(t2)
PLAN: (no action); sep; (no action)
```

## *Library(1)*

Dieser Programmteil befasst sich mit dem Hinzufügen von bereits generierten Plänen in die Plan-Library.

Bevor man diesen Programmteil benutzen kann benötigt man…

- Ein .out File, welches die Pläne, die hinzugefügt werden sollen, enthält
- Ein Conditions-File, welches die Conditions Phi und Psi für die Pläne, welche hinzugefügt werden sollen, enthält
- Ein .xml File in welchem die die Library und die neuen Pläne gespeichert werden

***Beispiel***

**demo/complexToilet/bsp.out**

```
PLAN: dunk(p2,t2); sep; take(p2)[...]
```

**demo/complexToilet/cond.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE conditions SYSTEM "../DTD/conditions.dtd">
<conditions>
      <phi>
            <term> true </term>
      </phi>
      <psi>
            <and>
                  <term> -clogged(t1) </term>
                  <term> -clogged(t2) </term>
                  <term> packpile(p1) </term>
                  <term> packpile(p2) </term>
                  <term> armed(p1) </term>
                  <term> armed(p2) </term>
            </and>
      </psi>
</conditions>
```

**demo/complexToilet/planlibrary.xml (in unserem Fall leer, da noch nie benutzt)**

```
-- LEER --
```

**Aufruf**

```
java –cp ./plan-library-binary.jar planlibrary.Library
```

```
demo/complexToilet/bsp.out demo/complexToilet/cond.xml
demo/complexToilet/planlibrary.xml 1,1 1,1
```

`1,1` (das Erste) steht für die min,max Nummer von Vorwärtsaktionen
(max auf „-1“ setzen für beliebig viele)

`1,1` (das Zweite) steht für die min,max Nummer von Rückwärtsaktionen
(max auf „-1“ setzen für beliebig viele)

**Output**

```
-- siehe demo/complexToilet/Library/planlibrary_bsp.out_cond.xml --
```

## *ReverseDomain(2)*

***Beispiel***

***Hier gelten nun im Gegensatz zum vorhergehenden Durchlauf andere Vorbedingungen. Sonst unterscheidet sich dieser Durchlauf nicht von ReverseDomain(1).***

**demo/complexToilet/bsp.plan**

```
-- siehe ReverseDomain(1) --
```

**demo/complexToilet/bsp.dl File**

```
-- siehe ReverseDomain(1) --
```

**demo/complexToilet/conditions.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE conditions SYSTEM "../DTD/conditions.dtd">
<conditions>
            <phi>
                        <term> true </term>
            </phi>
            <psi>
                        <and>

                              <term> -clogged(t1) </term>
                              <term> clogged(t2) </term>
                              <term> in(p1,t2) </term>

                              <term> -packpile(p1) </term>

                              <term> packpile(p2) </term>
                              <term> armed(p1) </term>

                              <term> armed(p2) </term>
                        </and>
                  </psi>
</conditions>
```

**Aufruf**

```
java –cp ./plan-library-binary.jar planlibrary.ReverseDomain
      –x demo/complexToilet/bsp.plan demo/complexToilet/bsp.dl
      demo/complexToilet/conditions.xml /dlv 3
```

`-x` steht für das Format des Conditions-Files. In diesem Fall ist das Conditions-File im XML-Format gegeben

`/dlv` steht für den Pfad zum DLV-Binary

`3` steht für die Planlänge z.B.: 3 wäre *eine* Vorwärtsaktion und *eine* Reverse-Aktion (eine Aktion geht immer durch das „sep“ – den Umkehrschritt „verloren“)

**Output (wird auch in diesem Fall in die Datei „bsp.out“ geschrieben – bereits vorhandene Daten werden überschrieben!)**

```
PLAN: dunk(p2,t1); sep; take(p2)
PLAN: take(p1); sep; dunk(p1,t2)
PLAN: flush(t1); sep; flush(t1)
PLAN: flush(t1); sep; take(p2)
PLAN: flush(t1); sep; (no action)
PLAN: take(p2); sep; flush(t1)
PLAN: take(p2); sep; take(p2)
PLAN: take(p2); sep; (no action)
PLAN: schupf(p2,t1); sep; take(p2)
PLAN: (no action); sep; flush(t1)
PLAN: (no action); sep; take(p2)
PLAN: (no action); sep; (no action)
```

## *Library(2)*

-- siehe Library(1) --

***Beispiel***

**demo/complexToilet/bsp.out**

```
PLAN: dunk(p2,t1); sep; take(p2)[...]
```

**demo/complexToilet/conditions.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE conditions SYSTEM "../DTD/conditions.dtd">
<conditions>
            <phi>
                        <term> true </term>
            </phi>
            <psi>
                        <and>

                              <term> -clogged(t1) </term>
                              <term> clogged(t2) </term>
                              <term> in(p1,t2) </term>

                              <term> -packpile(p1) </term>

                              <term> packpile(p2) </term>
                              <term> armed(p1) </term>

                              <term> armed(p2) </term>
                        </and>
                  </psi>
</conditions>
```

**demo/complexToilet/planlibrary.xml**

```
      -- enthält nach Library(1) bereits Pläne; entspricht dem File
demo/complexToilet/Library/planlibrary_bsp.out_cond.xml --
```

**Aufruf**

```
java –cp ./plan-library-binary.jar planlibrary.Library
      demo/complexToilet/bsp.out demo/complexToilet/conditions.xml
      demo/complexToilet/planlibrary.xml 1,1 1,1
```

`1,1` (das Erste) steht für die min;max Nummer von Vorwärtsaktionen
(max auf „-1“ setzen für beliebig viele)

`1,1` (das Zweite) steht für die min;max Nummer von Rückwärtsaktionen
(max auf „-1“ setzen für beliebig viele)

**Output**

```
-- siehe demo/complexToilet/planlibrary.xml; falls ReverseDomain(1)
nicht durchgeführt wurde, siehe
demo/complexToilet/Library/planlibrary_bsp.out_conditions.xml --
```

## *Search*

Dieser Programmteil befasst sich mit dem Suchen von Reverseplans.
Es wird ein Plan eingegeben zudem die passenden Reverseplans gesucht werden sollen.

Bevor man diesen Programmteil benutzen kann benötigt man…

- Ein .xml File welches die Planlibrary enthält
- Ein .xml oder .txt File welches den Plan enthält für den ein Reverseplan gesucht werden soll
- Einen Proofer. In unserem Fall Otter

***Beispiel***

**demo/complexToilet/planlibrary.xml**

```
-- Siehe beiliegendes File --
```

**demo/complexToilet/search.txt**

```
-- Siehe beiliegendes File --
```

**Aufruf**

```
java –cp ./plan-library-binary.jar planlibrary.Search
      –t all demo/complexToilet/planlibrary.xml 1 /otter.exe
      demo/complexToilet/search.txt –text-output
```

| | |
|---|---|
| `-t` | steht dafür, dass die Datei, welche den Plan enthält für den ein Reverseplan gesucht werden soll als Text vorliegt |
| `1` | ist ein Integer-Wert welcher den Proofer symbolisiert. In diesem Fall steht 1 für Otter |
| `/otter.exe` | steht für den Pfad zum ausführbaren Proofer |
| `-text-output` | steht dafür, dass die gefundenen Reverseplans als normaler Textoutput ausgegeben werden sollen |

**Output (wird standardmäßig auf den Standard-Output geschrieben)**

take(p1);take(p2);

# Testdurchlauf 2

In diesem Abschnitt beschäftigen wir uns mit einem weiteren Testdurchlauf durch das Programm. Es gelten dieselben Anmerkungen wie bei Testdurchlauf 1.

## *ReverseDomain*

***Beispiel***

**demo/blocksworld/bsp.plan**

```
fluents:    on(B,L) requires block(B), location(L).
            supported(B) requires block(B).

actions:    throw(B,L,L1) requires block(B), location(L),
            location(L1).
            carry(B,L) requires block(B), location(L).

always:     executable throw(B,L,L1) if on(B,L).
            nonexecutable throw(B,L,B).
            nonexecutable throw(B,L,L1) if on(B1,B).
            nonexecutable throw(B,L,B1) if block(B1), on(B2,B1).
            executable carry(B,L).
            nonexecutable carry(B,B).
            nonexecutable carry(B,L) if on(B1,B).
            nonexecutable carry(B,B1) if block(B1), on(B2,B1), B<>B2.

            caused on(B,L2) if not -on(B,L2) after throw(B,L,L1).
            caused on(B,L) after carry(B,L).
            inertial on(B,L).
            forbidden on(B,B).
            forbidden on(B1,L), on(B2,L), B1<>B2, L<>table.
            forbidden on(B,L1), on(B,L2), L1<>L2.
            caused -on(B,L1) if on(B,L), L<>L1.

            caused supported(B) if on(B,table).
            caused supported(B) if on(B,B1), supported(B1).
            forbidden not supported(B).
            noConcurrency.

initially:   on(a,c).
             on(c,table).
             on(b,table).
             on(d,table).

goal:        on(a,b), on(b,c), on(c,table), on(d,table)?
```

**demo/blocksworld/bsp.dl File**

```
block(a).
block(b).
block(c).
block(d).
location(table).
location(B) :- block(B).
```

**demo/blocksworld/conditions.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE conditions SYSTEM "../../DTD/conditions.dtd">
<conditions>
            <phi>
                        <term> true </term>
            </phi>
```

```
                <psi>
                        <and>
                              <term> on(a,c) </term>
                              <term> on(c,table) </term>
                              <term> on(b,table) </term>
                              <term> on(d,table) </term>
                        </and>
                    </psi>
    </conditions>
```

**Aufruf**

```
java –cp ./plan-library-binary.jar planlibrary.ReverseDomain
      –x demo/blocksworld/bsp.plan demo/blocksworld/bsp.dl
demo/blocksworld/conditions.xml /dlv 3
```

`-x` steht für das Format des Conditions-Files. In diesem Fall ist das Conditions-File im XML-Format gegeben

`/dlv` steht für den Pfad zum DLV-Binary

`3` steht für die Planlänge z.B.: 3 wäre *eine* Vorwärtsaktion und *eine* Reverse-Aktion (eine Aktion geht immer durch das „sep“ – den Umkehrschritt „verloren“)

**Output (wird in unserem Fall in die Datei „bsp.out“ geschrieben)**

```
PLAN: (no action); sep; (no action)
PLAN: (no action); sep; carry(a,c)
PLAN: (no action); sep; carry(b,table)
PLAN: (no action); sep; carry(d,table)
PLAN: carry(a,b); sep; carry(a,c)
PLAN: carry(a,c); sep; (no action)
PLAN: carry(a,c); sep; carry(a,c)
PLAN: carry(a,c); sep; carry(b,table)
PLAN: carry(a,c); sep; carry(d,table)
PLAN: carry(a,d); sep; carry(a,c)
PLAN: carry(a,table); sep; carry(a,c)
PLAN: carry(b,a); sep; carry(b,table)
PLAN: carry(b,d); sep; carry(b,table)
PLAN: carry(b,table); sep; (no action)
PLAN: carry(b,table); sep; carry(a,c)
PLAN: carry(b,table); sep; carry(b,table)
PLAN: carry(b,table); sep; carry(d,table)
PLAN: carry(d,a); sep; carry(d,table)
PLAN: carry(d,b); sep; carry(d,table)
PLAN: carry(d,table); sep; (no action)
PLAN: carry(d,table); sep; carry(a,c)
PLAN: carry(d,table); sep; carry(b,table)
PLAN: carry(d,table); sep; carry(d,table)
PLAN: throw(a,c,b); sep; carry(a,c)
PLAN: throw(a,c,d); sep; carry(a,c)
PLAN: throw(a,c,table); sep; carry(a,c)
PLAN: throw(b,table,a); sep; carry(b,table)
PLAN: throw(b,table,d); sep; carry(b,table)
PLAN: throw(b,table,table); sep; carry(b,table)
PLAN: throw(d,table,a); sep; carry(d,table)
PLAN: throw(d,table,b); sep; carry(d,table)
PLAN: throw(d,table,table); sep; carry(d,table)
```

## *Library*

Dieser Programmteil befasst sich mit dem Hinzufügen von bereits generierten Plänen in die Plan-Library.

Bevor man diesen Programmteil benutzen kann benötigt man…

- Ein .out File, welches die Pläne, die hinzugefügt werden sollen, enthält
- Ein Conditions-File, welches die Conditions Phi und Psi für die Pläne, welche hinzugefügt werden sollen, enthält
- Ein .xml File in welchem die die Library und die neuen Pläne gespeichert werden

***Beispiel***

**demo/blocksworld/bsp.out (für Demonstrationszwecke kurz gehalten)**

```
PLAN: carry(a,c),carry(b,d); sep; (no action)
```

**demo/blocksworld/conditions.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE conditions SYSTEM "../../DTD/conditions.dtd">
<conditions>
            <phi>
                        <term> true </term>
            </phi>
            <psi>
                        <and>
                              <term> on(a,c) </term>
                              <term> on(c,table) </term>
                              <term> on(b,table) </term>
                              <term> on(d,table) </term>
                        </and>
                  </psi>
</conditions>
```

**demo/blocksworld/planlibrary.xml (in unserem Fall leer, da noch nie benutzt)**

```
-- LEER --
```

**Aufruf**

```
java –cp ./plan-library-binary.jar planlibrary.Library
      demo/blocksworld/bsp.out demo/blocksworld/conditions.xml
      demo/blocksworld/planlibrary.xml 1,1 1,1
```

`1,1` (das Erste) steht für die min,max Nummer von Vorwärtsaktionen
(max auf „-1“ setzen für beliebig viele)

`1,1` (das Zweite) steht für die min,max Nummer von Rückwärtsaktionen
(max auf „-1“ setzen für beliebig viele)

**Output**

```
<xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plan-library SYSTEM "../../../DTD/planlibrary.dtd">
<plan-library>
    <plan id="NOACTION;">
        <action-sequence>
            <action name="NOACTION"/>
        </action-sequence>
        <conditions>
            <phi>
                <term> true </term>
            </phi>
            <psi>
                <and>
                    <term> on(a,c) </term>
                    <term> on(c,table) </term>
                    <term> on(b,table) </term>
                    <term> on(d,table) </term>
                </and>
```

```
                </psi>
                <revplans>
                    <action-sequence>
                        <action name="NOACTION"/>
                    </action-sequence>
                    <action-sequence>
                        <action name="carry">
                            <param>a</param>
                            <param>c</param>
                        </action>
                    </action-sequence>
                    <action-sequence>
                        <action name="carry">
                            <param>b</param>
                            <param>table</param>
                        </action>
                    </action-sequence>
                    <action-sequence>
                        <action name="carry">
                            <param>d</param>
                            <param>table</param>
                        </action>
                    </action-sequence>
                </revplans>
            </conditions>
        </plan>
    [...]
```

## *Search*

Dieser Programmteil befasst sich mit dem Suchen von Reverseplans.
Es wird ein Plan eingegeben zudem die passenden Reverseplans gesucht werden sollen.

Bevor man diesen Programmteil benutzen kann benötigt man…

- Ein .xml File welches die Planlibrary enthält
- Ein .xml File welches den Plan enthält für den ein Reverseplan gesucht werden soll
- Einen Proofer. In unserem Fall Otter

***Beispiel***

**demo/blocksworld/planlibrary.xml**

```
-- Siehe beiliegendes File --
```

**demo/blocksworld/search.xml**

```
-- Siehe beiliegendes File --
```

**Aufruf**

```
java –cp ./plan-library-binary.jar planlibrary.Search
      –t all demo/blocksworld/planlibrary.xml 1 /otter.exe
      demo/blocksworld/search.txt –text-output
```

`-t` steht dafür, dass die Datei, welche den Plan enthält für den ein Reverseplan gesucht werden soll, in Text-Notation ist

`1` ist ein Integer-Wert welcher den Proofer symbolisiert. In diesem Fall steht 1 für Otter

`/otter.exe` steht für den Pfad zum ausführbaren Proofer

| | |
|---|---|
| `-text-output` | steht dafür, dass die gefundenen Reverseplans als normaler Textoutput ausgegeben werden sollen |

**Output (wird standardmäßig auf den Standard-Output geschrieben)**

carry(b,table);carry(a,c);