bflow* Toolbox

Anpassen und Erweitern von bflow*

9. März 2015

Autoren: Ralf Laue, Arian Storch,
überarbeitet von Mathias Grunert, Florian Schönfelder

Anfragen zur bflow* Toolbox sind willkommen an bflow@bflow.org. Softwareentwickler sind eingeladen, an der Entwicklung unserer freien Software mitzuwirken. Wenn Sie Erweiterungen erstellt haben, lassen Sie es uns bitte wissen. Vielleicht sind diese ja auch für andere Nutzer interessant.

Mehr Informationen zur bflow* Toolbox gibt's auf `http://www.bflow.org`. Bitte kontaktieren Sie uns auch, wenn Sie Fehler oder Verständnisschwierigkeiten im vorliegenden Handbuch finden.

Inhaltsverzeichnis

Einleitung

Dieses Handbuch richtet sich an Administratoren mit Programmiererfahrung, die den Leistungsumfang von bflow* erweitern wollen. An einigen Stellen bietet die bflow* Toolbox hierfür einfache Schnittstellen.

Es ist jedoch nicht Ziel dieses Handbuchs, eine vollständige Dokumentation für Entwickler zu liefern.

Endanwender werden mit den hier vorgestellten Erweiterungen nicht direkt in Berührung kommen; für sie genügt die Lektüre des bflow*-Benutzerhandbuches.

Dieses Handbuch is ist drei Abschnitte unterteilt.

Der erste Abschnitt befasst sich mit dem Austausch von Modellen, die in anderen Modellierungssprachen oder mit anderen Werkzeugen erstellt wurden. Dieser Austausch wird über Import- und Exportskripte realisiert. Wie eigene Skripte zur Unterstützung weiterer Formate eingebunden bzw. erstellt werden können, wird im ersten Abschnitt erläutert.

Der zweite Abschnitt widmet sich der Add-on-Schnittstelle von bflow*. Diese Schnittstelle erlaubt es, den Funktionsumfang der bflow* Toolobx zu erweitern, indem eigene Programme eingebunden werden. Diese können zusätzliche Funktionen (z.B. zur Simulation oder zur Kostenrechnung) realisieren. Es wird anhand eines ausführlichen Beispiels beschrieben, wie eigene Add-ons erstellt und genutzt werden können.

Die bisher genannten Erweiterungsmöglichkeiten kommen ohne Änderung des Programmcodes der bflow* Toolbox aus.

Im dritten Abschnitt dagegen werden Erweiterungsmöglichkeiten vorgestellt, bei denen der bflow*-Programmcode erweitert werden muss. Erläutert werden das Erstellen eigener Validierungsregeln und eigener Farbschemata.

Wenn für eine Erweiterung von bflow* der bflow*-Quelltext geändert weden muss, ist das daran erkennbar, dass in der Überschrift die Anmerkung „für Entwickler“ steht.

Übrigens: Wenn Sie eine Erweiterung programmiert haben, würden wir uns freuen, davon zu hören (E-Mail an bflow@bflow.org). Denn vielleicht sind auch andere Nutzer an Ihrem Export oder Add-on interessiert, und Ihr Code könnte in die nächste Version der bflow* Toolbox aufgenommen werden.

1 Erweitern des Formataustausches

Um eine größtmögliche Flexibilität beim Austausch von Modellen mit anderen Werkzeugen zu erreichen, besitzt die bflow* Toolbox eine Funktionen zur Umwandlung verschiedener Formate. Diese Umwandlungen werden über Import- und Exportskripte realisiert.

Dieses Feature wird unter anderem von den Add-Ons genutzt, um das zu bearbeitende bzw. zu analysierende Modell in ein Format zu übersetzen, welches ein einzubindendes externes Programm lesen und verarbeiten kann.

In den folgenden Abschnitten wird nun erläutert, wie eigene Import- und Exportskripte für das Add-On-Plugin entwickelt werden können.

1.1 Hinzufügen eines Exports

1.1.1 Erstellen einer Beschreibungsdatei für den Export

Zur Definition eines Exports ist zunächst eine Beschreibungsdatei anzulegen, die die wesentlichen Informationen zum Export enthält. Hierzu zählen unter anderem die Bezeichnung unter der das Exportformat im Menü zu finden ist, kompatible Diagrammtypen, sowie die Angabe der Arbeitsschritte, die für die Transformation auszuführen sind. Um diese Arbeitsschritte zu beschreiben, werden die auszuführenden Skripte angegeben. Die eigentliche Logik für die Transformation befindet sich in den Skripten. Es ist auch möglich, mehrere Skripte hintereinander auszuführen.

Die Beschreibungsdatei ist mit der Dateinamenswerweiterung .exd zu speichern.

Beispiele für Beschreibungsdateien (für die Exporte, die im Lieferumfang der bflow* Toolbox enthalten sind) findet man unter:

`http://sourceforge.net/p/bflowtoolbox/code/HEAD/tree/trunk/plugins/org.bflow.toolbox.contributions.addons/exportscripts/`

Das formale Grundgerüst eine Beschreibungsdatei kann wie folgt aussehen:

```
1 [public] interchange 'Titel' (Zielformat) [<> Diagrammtyp] {
2 [description: Beschreibungstext]
3 @script: 'Pfad' [, Parameter, ...]
4 [...]
5 }
```

Ausdrücke in eckigen Klammern stehen dabei für optionale Angaben. Beginnt und endet ein Ausdruck mit "%", so handelt es sich dabei um einen Parameter, der ausgefüllt werden muss. Es folgen zwei Beispiele für eine korrekte und vollständig ausgefüllte Beschreibungsdatei:

```
1 public interchange 'EPML-Export (epml) {
2 description: 'Exportiert ein Diagramm als EPML.'
3 @script: '/files/epc2epml-teil1.xslt'
4 @script: '/files/epc2epml-teil2.xslt'
5 @script: '/files/epml-und-epc-resize.xslt', Multiplikator="1",
      Streckung="1", MultiKonnektor="1.0"
6 }
```

```
1 interchange 'UML' (umlx|uml|xml) <> oepc{
2 description: 'Erzeugt ein UML-Klassendiagramm.'
3 @script: '/files/uml1.vt', factor=4, canGrow=true, defaultSize=40.5
4 @script: '/files/umlX.vt'
5 @script: '/files/uml2.vt', plain=true
6 }
```

Eine Beschreibung der Parameter ist in folgender Tabelle zu finden:

Name	Beschreibung
Titel	Dieses Feld steht für den Titel bzw. den Namen des Austauschformates. Dieser wird in der Auswahlanzeige des Exportdialoges verwendet.
Zielformat	Dieses Feld steht für eine Auflistung von Dateiendungen, die für Ziel- bzw. Quelldateien eines Transformationsvorgangs verwendet werden können. Die Auflistung muss mindestens ein Kürzel enthalten. Mehrere Angaben können durch „|“ getrennt werden. Standardmäßig wird für einen Export immer das erste Kürzel für die Zieldatei verwendet.
Diagrammtyp	Legt eine Auflistung von Diagrammtypen fest, für die das jeweils definierte Austauschformat anwendbar ist. Diese Angabe ist optional. Wird keine Angabe gemacht, so ist das Austauschformat für alle Diagrammtypen verfügbar. Die Auflistung muss mindestens eine Angabe enthalten. Als Angabe gilt dabei jeweils die Dateiendung der Modelldatei des jeweiligen Modelleditors. Mehrere Angaben können mittels Komma getrennt werden.
Text	Siehe dazu die Beschreibung des optionalen Parameters „description“ der nachfolgenden Tabelle.
Pfad	Gibt den relativen Pfad zur Skriptdatei an. Dabei wird von der Lage der Beschreibungsdatei ausgegangen.
Parameter	Siehe dazu die Beschreibung des optionalen Parameters „Parameters“ der nachfolgenden Tabelle.

Tabelle 1.1: Parameterbeschreibung

Eine Auflistung und Beschreibung der optionalen Parameter ist in folgender Tabelle zu finden:

Name	Wertebereich	Beschreibung
public		Wenn gesetzt, dann ist das Austauschformat nicht nur für Add-ons sondern auch für den gewöhnlichen Exportvorgang verfügbar. Das bedeutet, dass das Austauschformat über den Exportdialog auswählbar ist.
description	String	Die hier angegebene Beschreibung wird im Dialogfenster beim Exportieren angezeigt. Sie kann eine ausführliche Beschreibung zum Exportformat geben.
Diagrammtyp	String	epc (EPK-Modelle) oder oepc (oEPK-Modelle)
Parameter		Parameter, die an das aufgerufene Exportskript weitergegeben werden Ein vorgegebener Parameter ist: insertAttributes=true Nur wenn dies gesetzt ist, werden die benutzerdefinierten Attribute eines Modells beim Export beachtet.

Tabelle 1.2: Optionale Parameter

Anmerkung: In früheren Versionen der bflow* Toolbox gab es noch eine weitere Form von Beschreibungsdateien, die die Dateiendung exd.xml hatten. Diese werden nach wie vor unterstützt, sollten aber beim Schreiben neuer Exporte/Importe nicht mehr verwendet werden.

1.1.2 Erstellen von Exportskripten mit Velocity

Eine Skriptdatei mit der Endung .vt nutzt *Velocity*, ein Programm zum Verarbeiten von Templates. Das folgende Beispiel zeigt ein einfaches mit *Velocity* realisiertes Exportskript, das wir im Folgenden als Template bezeichnen werden:

```
1 Liste aller Knoten:
2 #foreach( $shape in $shapes )
3 Knotenname: $shape.Name, Typ: Type.split("\.")[4].toLowerCase()
4 #end
5 - Abschluss -
```

Das Ergebnis des Transformationsprozesses könnte dann (für die EPK in Abb. 1.3) beispielsweise wie folgt aussehen:

```
Liste aller Knoten:
Knotenname: Ei ist roh, Typ: Event
Knotenname: koche Ei, Typ: Function
Knotenname: Ei ist gekocht, Typ: Event
- Abschluss
```

Anhand des Beispiels ist erkennbar, dass jedes Template aus einem statischen und einem dynamischen Anteil bestehen kann. Die statischen Teile (wie im Beispiel der Text "Liste aller Knoten") werden direkt in die Ausgabedatei geschreiben. Dynamische Anweisungen beginnen mit dem Symbol „#". Das Zeichen „$" signalisiert einen Zugriff auf eine Variable.

Die Engine, die zur Umsetzung der Transformation verwendet wird, ist *Apache Velocity* .

Zum Erstellen eigener Skripte ist daher ein Studium der Dokumentation (`http://velocity.apache.org/`) zu empfehlen.

Beim Programmieren können die GenericTools von Velocity (`http://velocity.apache.org/tools/devel/generic/index.html`) viel Arbeit sparen. Insbesondere sei darauf hingewiesen, dass beim Erstellen von Ausgabedateien im XML-Format die XML-Entitäten (also z.B. `&` statt des Zeichens `&`) vom Programmierer richtig erzeugt werden müssen. Hier hilft das Escape-Tool der GenericTools.

Ebenso nützlich ist es, dass sich String-Methoden von Java verwenden lassen (wie z.B. `$var.trim()` oder `$var.replace(...)`. Oft müsen beispielsweise Zeilenumbrüche, die in der Beschriftung eines Modellelements vorkommen, entfernt werden. Dies gelingt wie folgt:

```
#set( $label = $label.replaceAll("\n", " ") )
#set( $label = $label.replaceAll("\r", " ") )
```

Zum Verstehen des Vorgehens bei der Erstellung von Velocity-Exporten hilft ein Studium der im bflow*-Lieferumfang enthaltenen Exportskripte, siehe:

`http://sourceforge.net/p/bflowtoolbox/code/HEAD/tree/trunk/plugins/org.bflow.toolbox.contributions.addons/exportscripts/files/`

In den folgenden Abschnitten sind die von bflow* verwendeten Variablen und ihre Eigenschaften dokumentiert:

Verfügbare Variable und deren Eigenschaften Die folgenden Variablen stehen bei jedem Transformationsvorgang zur Verfügung:

Variable	Eigenschaft	Bedeutung
model		stellt den Zugriff auf Eigenschaften bereit, die das Modell besitzt
	Id	Id des Modells
	Type	Typ des Modells
	Attributes	Key-Value-Map aller dem Modell zugeordneten Attribute (falls der Paramter `insertAttributes=true` gesetzt wurde)
shapes		Array, das alle im Modell vorkommenden Knoten beinhaltet. Jeder Knoten innerhalb dieses Arrays besitzt die in den folgenden Zeilen gezeigten Eigenschaften.
	Id	Id des Knotens
	Type	Typ des Knotens (vgl. Tab. 1.4und Tab.1.5)
	Attributes	Key-Value-Map aller dem Knoten zugeordneten Attribute (falls der Paramter `insertAttributes=true` gesetzt wurde)
	Name	Name des Knotens
	Width	Breite des Shapes, das den Knoten darstellt
	Height	Höhe des Shapes, das den Knoten darstellt
	X	x-Koordinate des Shapes, das den Knoten darstellt
	Y	y-Koordinatedes Shapes, das den Knoten darstellt
	Image	Bild des Shapes, das den Knoten darstellt
edges		Array, das alle im Modell vorkommenden Kanten beinhaltet. Jede Kante innerhalb des Arrays besitzt die in den folgenden Zeilen gezeigten Eigenschaften.
	Id	Id der Kante
	Type	Typ der Kante
	Attributes	Key-Value-Map aller der Kante zugeordneten Attribute (falls der Paramter `insertAttributes=true` gesetzt wurde)
	Source	Gibt den Knoten an, den die Kante als Quelle hat. Das hinterlegte Objekt besitzt dabei die gleichen Eigenschaften wie jeder Knoten.
	Target	Gibt den Knoten an, den die Kante als Ziel hat. Das hinterlegte Objekt besitzt dabei die gleichen Eigenschaften wie jeder Knoten.
params		Key-Value-Map, die die beim Aufruf des Skripts mitgegebenen Parameter enthält (vgl. Abschnitt 1.1.1)

Tabelle 1.3: Variablen, die beim Velocity-Export verwendet werden können

Attribute und Methoden von Geschäftsobjekten in einer oEPK Auf Methoden und Attribute (letztere nicht zu verwechseln mit den benutzerdefinierten Attributen, die jedem Modellelement hinzugefügt werden können) von Geschäftsobjekten kann wie im folgenden Beispiel zugegriffen werden:

```
#foreach( $shape in $shapes)
 Elementname: $shape.Name
 #foreach( $attr in $shape.BusinessAttributes )
   BusinessAttribute: $attr.Name
 #end
 #foreach( $method in $shape.BusinessMethods )
   BusinessMethod: $method.Name
 #end
#end
```

1.1.3 Erstellen von Exportskripten mit XSLT

Alternativ zum Export mittels Velocity-Templates können auch XSLT-Transformationen zum Dateiexport verwendet werden. Dabei ist es am einfachsten, nicht das interne Speicherformat von bflow* zum Ausgangspunkt zu nehmen, sondern das deutlich leichter verständliche EPML-Format (siehe `http://www.mendling.com/EPML/`). Es existiert nämlich bereits eine XSLT-Transformation in dieses EPML-Format, zu finden unter:

`http://sourceforge.net/p/bflowtoolbox/code/HEAD/tree/trunk/plugins/org.bflow.toolbox.contributions.addons/exportscripts/files/epc2epml-teil1.xsltund`

`http://sourceforge.net/p/bflowtoolbox/code/HEAD/tree/trunk/plugins/org.bflow.toolbox.contributions.addons/exportscripts/files/epc2epml-teil1.xslt`

Die Beschreibungsdatei für einen Export könnte somit so aussehen:

```
public interchange 'eigener Export (xml) <> epc {
description: 'Exportiert in ein gewünschtes XML-Format.'
@script: '/files/epc2epml-teil1.xslt'
@script: '/files/epc2epml-teil2.xslt'
@script: '/files/eigene-transformation.xslt'
}
```

Dabei wird vorausgesetzt, dass die Transformationen epc2epml-teil1.xslt und epc2epml-teil1.xslt in den Unterordner .export/files des bflow*-Arbeitsbereichs kopiert wurden.

Eine Einführung in die Transformationssprache XSLT findet sich z.B. auf `http://www.w3schools.com/xsl/`.

1.1.4 Einbinden der Beschreibungsdatei und der Skripte für den Export

Nachdem Beschreibungsdatei und Skript angelegt wurde, muss im Arbeitsbereich (Workspace) ein Unterordner ".export" angelegt werden, in dem die Exportbeschreibungsdatei

gespeichert wird. Ebenso müssen die Skripte, die die eigentliche Transformationsarbeit erledigen, unter dem Namen gespeichert werden, der in der Beschreibungsdatei angegeben wurde. Anschließend muss bflow* neu gestartet werden.

Nun steht bei Aufrufen des Export-Funktion ein neues Exportformat zur Verfügung:

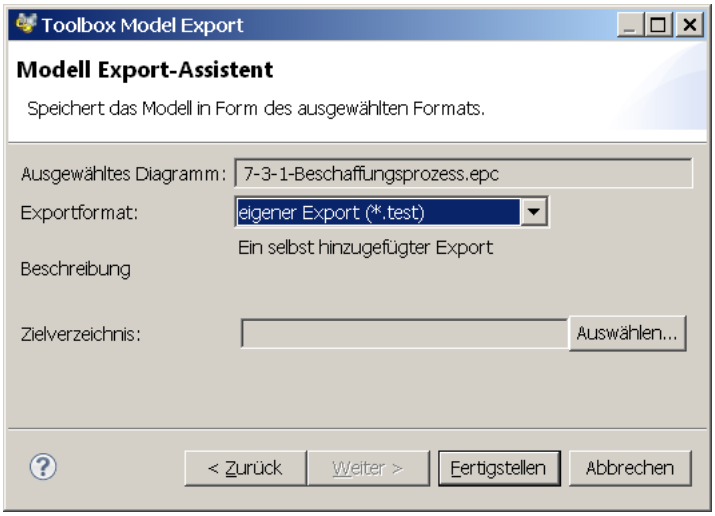

Abbildung 1.1: Neuer bflow*-Export

1.1.5 Nützliches Wissen zu benutzerdefinierten Attributen beim Dateiexport

In der bflow*-Ansicht „Attribute View" können dem Modell als Ganzen oder einzelnen Modellelementen benutzerdefinierte Attribute hinzugefügt werden.

Diese benutzerdefinierten Attribute werden direkt in der Modelldatei gespeichert. Die Notation in dem von bflow* verwendeten XMI-Format ist simpel gestaltet und sieht in etwa so aus:

```
<addon:AddonAttributes xmi:id="_kvgkEY9WEeCarrMVd1typg">
  <attributes xmi:type="addon:Attribute" xmi:id="_kvgkEo9WEeCarrMVd1typg" id="_NVGrQlanEeCVr4GvizSVWg" name="Vendor" value="John Doe"/>
  <attributes xmi:type="addon:Attribute" xmi:id="_kvgkE49WEeCarrMVd1typg" id="_NVGrQlanEeCVr4GvizSVWg" name="Version" value="1.0"/>
  <attributes xmi:type="addon:Attribute" xmi:id="_kvgkFI9WEeCarrMVd1typg" id="_c-uEclanEeCVr4GvizSVWg" name="Name" value="Toni"/>
  <attributes xmi:type="addon:Attribute" xmi:id="_kvgkFY9WEeCarrMVd1typg" id="_i_Kj8lanEeCVr4GvizSVWg" name="Description" value="http://www.mydesc.org"/>
  <attributes xmi:type="addon:Attribute" xmi:id="_kvgkFo9WEeCarrMVd1typg" id="_i_Kj8lanEeCVr4GvizSVWg" name="file" value="file://D:/test.txt"/>
  <attributes xmi:type="addon:Attribute" xmi:id="_kvgkF49WEeCarrMVd1typg" id="_i_Kj8lanEeCVr4GvizSVWg" name="Level" value="80"/>
  <attributes xmi:type="addon:Attribute" xmi:id="_kvgkGI9WEeCarrMVd1typg" id="_bgjdUlanEeCVr4GvizSVWg" name="Name" value="Toni"/>
 </addon:AddonAttributes>
</xmi:XMI>
```

Abbildung 1.2: Benutzerdefinierte Attribute in der Modelldatei

Die ID gibt die ID des Modellelements an, das das jeweilige Attribut besitzt. Diese ist eindeutig und wird vom EMF/GMF-Framework vergeben. Danach folgen Name und Wert des Attributs.

Die benutzerdefinierten Attribute stehen auch in exportierten EPML- und Prologdateien zur Verfügung.

1.2 Hinzufügen eines Imports

1.2.1 Erstellen einer Beschreibungsdatei für den Import

Das Hinzufügen eigener Importformate geschieht völlig analog zum beschriebenen Hinzufügen eigener Exportformate. Jedoch müssen die Beschreibungsdateien mit der Endung .ixd im bflow*-Arbeitsbereich in einem Ordner mit dem Namen .import abgelegt werden.

Der Inhalt der Importbeschreibung muss nach folgendem Schema entworfen werden:

```
1 public interchange 'Import vom XY-Werkzeug' (xml) <> epc {
2 description: 'Ein ganz toller eigener Import.'
3 @script: '/files/eigener_import.ads', insertAttributes=true
4 }
```

Durch die Angabe von "(xml)" in der ersten Zeile wird entschieden, welche Dateien durch den Import importiert werden können (Im vorliegenden Beispiel alle Dateien mit der Endung .xml.).

Die Importskripte, die die eigentliche Arbeit erledigen, werden in den nach @skript angegebenen Dateinamen gespeichert.

1.2.2 Erstellen von Importskripten im ADS-Format

Eigene Importskripte können in Dateien mit der Endung .ads gespeichert werden. Diese bezieht sich auf die Grammatik *Aditus*, die dem Import zugrunde liegt. Damit ist es möglich, Werten von XPath-Ausdrücken entsprechenden Eigenschaften des zu erstellenden Modells zuzuordnen. Eine Einführung zu XPath, sowie entsprechende Tutorials sind unter `http://www.w3schools.com/XPath/` zu finden.

Nehmen wir an, wir wollen eine EPK-Datei importieren, die im EPML-Format ist und so aussieht:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <epc>
3     <event id="1" name="Ei ist roh" x="50" y="50" height="30" width="
          150" />
4     <function id="2" name="koche Ei" x="50" y="150" height="30" width=
          "150" />
5     <event id="3" name="Ei ist gekocht" x="50" y="250" height="30"
          width="150" />
6     <flow from="1" to="2" />
7     <flow from="2" to="3" />
8 </epc>
```

Diese EPK beschreibt zwei Ereignisse und eine Funktion. Ein Ei ist am Anfang roh, daraufhin wird es gekocht und ist danach gekocht. „Ei ist roh“ und „Ei ist gekocht“ sind hier also die Ereignisse und „Ei kochen“ die Funktion. Mit den zwei flow-Eelementen werden die Kanten zwischen der Funktion und den Ereignissen ausgedrückt. Am Ende des Imports sollte die importierte EPK wie folgt aussehen:

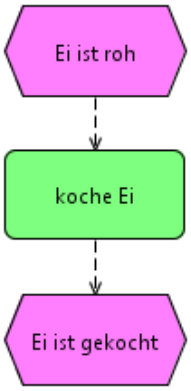

Abbildung 1.3: EPK “Ei kochen”

Nun muss noch das Importskript für das Einlesen des Beispielimports geschrieben werden. Das oben genannte Tutorial für XPath ist hierbei hilfreich. Das Metamodell von Aditus sieht zwei grundlegende Modellelemente vor: Knoten (shapes) und Kanten (edges). Das hier beschriebene Beispiel besitzt Ereignisse, Funktionen (beides sind Knoten) und Kanten. Als erstes werden die Ereignisse eingelesen. Hierzu wird beschrieben, dass der Import alle Ereignisse als Knoten erkennt.

```
1 shapes = $//epc/event$
```

Hinweis: “$...$” beschreibt hier einen XPath-Ausdruck.

Damit werden alle Ereignisse aus der EPK herausgesucht. Als nächstes werden die Attribute benötigt. Für den vorliegenden Fall wären das die ID, der Typ (beschreibt das dazugehörige Bflow*-Element), Höhe und Breite des Ereignisses und die X-Y-Koordinaten.

```
1  # @ is used to access xml tag attributes
2  id = $@id$
3  #type of element created in bflow
4  type = 'org.bflow.toolbox.epc.diagram.Event_2006'
5  name = $@name$
6  # | is an alternative, for exmaple: if attribute @width is not found,
7  120 is used instead
8  width = $@width$ | 120
9  height = $@height$ | 30
10 x = $@x$ | 0
11 y = $@y$ | 0
```

Mit „#“ wird signalisiert, dass es sich um ein Kommentar handelt. „@“ wird genutzt um Attribute zu kennzeichnen. Wenn Attribute wie beispielsweise Breite und Höhe nicht gefunden werden können, können alternative Werte angeben werden, welche durch ein „|“ gekennzeichnet werden.

Um die Typzuordnung der Elemente zu bflow*-Elementen mit XPath zu vereinfachen, kann folgende Tabelle genutzt werden:

Ereignis	"org.bflow.toolbox.epc.diagram.Event_2006"
Funktion	"org.bflow.toolbox.epc.diagram.Function_2007"
OR-Konnektor	"org.bflow.toolbox.epc.diagram.OR_2001"
XOR-Konnektor	"org.bflow.toolbox.epc.diagram.XOR_2008"
AND-Konnektor	"org.bflow.toolbox.epc.diagram.AND_2003"
Anwendung	org.bflow.toolbox.epc.Application_2004
Organisationseinheit	org.bflow.toolbox.epc.Participant_2002
Gruppe	org.bflow.toolbox.epc.Group_2009
Standort	org.bflow.toolbox.epc.Location_2014
Stelle	org.bflow.toolbox.epc.Position_2013
Cluster	org.bflow.toolbox.epc.Cluster_2010
Interne Person	org.bflow.toolbox.epc.InternalPerson_2012
Externe Person	org.bflow.toolbox.epc.ExternalPerson_2011
Personentype	org.bflow.toolbox.epc.PersonType_2015
Produkt	org.bflow.toolbox.epc.Product_2021
Ziel	org.bflow.toolbox.epc.Objective_2020
Prozessschnittstelle	org.bflow.toolbox.epc.ProcessInterface_2005
Fachbegriff	org.bflow.toolbox.epc.TechnicalTerm_2016
Datei	org.bflow.toolbox.epc.File_2019
Dokument	org.bflow.toolbox.epc.Document_2018
Kartei	org.bflow.toolbox.epc.CardFile_2017
Ablaufsteuerung (Kante)	"org.bflow.toolbox.epc.diagram.Arc_4001"
Verbindung (Kante)	org.bflow.toolbox.epc.Relation_4002
Informationssteuerung (Kante)	org.bflow.toolbox.epc.InformationArc_4003

Tabelle 1.4: Type-Attribute der Modellelemente von EPKs

Für oEPKs ist die Zuordnung wie folgt:

Ereignis	oepc.Event_2001
Geschäftsobjekt	oepc.BusinessObject_2005
OR-Konnektor	oepc.ORConnector_2007
XOR-Konnektor	oepc.XORConnector_2004
AND-Konnektor	oepc.ANDConnector_2006
IT-System	oepc.ITSystem_2002
Organisationseinheit	oepc.OrganisationUnit_2003
Dokument	oepc.Document_2008
Ablaufsteuerung (Kante)	oepc.ControlFlowEdge_4001
Informationssteuerung (Kante)	oepc.InformationEdge_4002

Tabelle 1.5: Type-Attribute der Modellelemente von oEPKs

Anschließend werden Elemente und Attribute zusammengefügt. Das Ergebnis könnte beispielsweise so aussehen:

```
# Events
# select tags to create shapes from
# here: select all event tags within the epc tag of the xml
file to create events
shapes = $//epc/event$ {
     # @ is used to access xml tag attributes
     id = $@id$
     # type of element created in bflow
     type = 'org.bflow.toolbox.epc.diagram.Event_2006'
     name = $@name$
     # | is an alternative, for exmaple: if attribute @width
is not found, 120 is used instead
     width = $@width$ | 120
     height = $@height$ | 30
     x = $@x$ | 0
     y = $@y$ | 0
}
```

Das Auslesen von Funktionen funktioniert nach dem gleichen Prinzip wie das Auslesen von Ereignissen.

```
# Functions
# same procedure as event
shapes = $//epc/function$ {
id = $@id$
type = 'org.bflow.toolbox.epc.diagram.Function_2007'
name = $@name$
width = $@width$ | 120
height = $@height$ | 30
x = $@x$ | 0
y = $@y$ | 0
}
```

Zum Schluss müssen die Kanten eingelesen werden. Abgesehen davon, dass anstelle des Schlüsselworts “shapes” nun “edges” genutzt wird, verhält sich die Erstellung analog zu den Ereignissen bzw. Funktionen.

```
# Edges
# mind that the keyword edges is used instead of
shapes now edges = $//epc/flow$ {
      type = 'org.bflow.toolbox.epc.diagram.Arc_4001'
      #source is set to the shape with the id read
out of the from-attribute in the xml flow tag
      source = shape <=> shape.id == $@from$
      target = shape <=> shape.id == $@to$
}
```

Nach dem Zusammenführen aller drei Teilabschnitte kann das Importskript die wichtigsten Elemente einer EPK einzulesen.

1.2.3 Erstellen von Importskripten mit XSLT

Analog zum Export (vgl. Abschnitt 1.1.3) kann auch für den Import von Modellen eine XSLT-Transformation verwendet werden.

Auch hier stehen wieder XSLT-Skripts zur Verfügung, die eine EPML-Datei in das interne Format der bflow* Toolbox umwandeln:

`http://sourceforge.net/p/bflowtoolbox/code/HEAD/tree/trunk/plugins/org.bflow.toolbox.contributions.addons/importscripts/files/epml2epc-teil1.xslt`
und

`http://sourceforge.net/p/bflowtoolbox/code/HEAD/tree/trunk/plugins/org.bflow.toolbox.contributions.addons/importscripts/files/epml2epc-teil2.xslt`

1.2.4 Import von Nicht-XML-Dateien

Die obige Dokumentation erklärt den Import von XML-Daten. Es ist auch möglich, nicht XML-Daten in XML-Daten vor dem Einlesen mit Aditus umzuwandeln. Genutzt wird dieser Mechanismus beispielsweise durch das BAI-Format. Dort wird zunächst das Ausgangsformat in XML umgeformt, das dann wie üblich importiert werden kann. Details findet man im Quellcode:

`http://sourceforge.net/p/bflowtoolbox/code/HEAD/tree/trunk/plugins/org.bflow.toolbox.contributions.addons/src/org/bflow/toolbox/contributions/addons/BAIInterchangeList`
`java`

1.3 Hinzufügen eigener Import- und Exportformate zu den bflow*-Quellen (für Entwickler)

Soll ein Import oder Export in die bflow*-Quellen aufgenommen werden, müssen Beschreibungsdatei und Skripte in ein Eclipse-Plug-In gepackt werden und über den entsprechenden Extension-Point registriert werden. Dies kann dann wie in Abbildung 1.4 aussehen.

Die zu verwendenden Extension-IDs lauten wie folgt:

Export org.bflow.toolbox.contributions.addons.exportscripts

Import org.bflow.toolbox.contributions.addons.importscripts

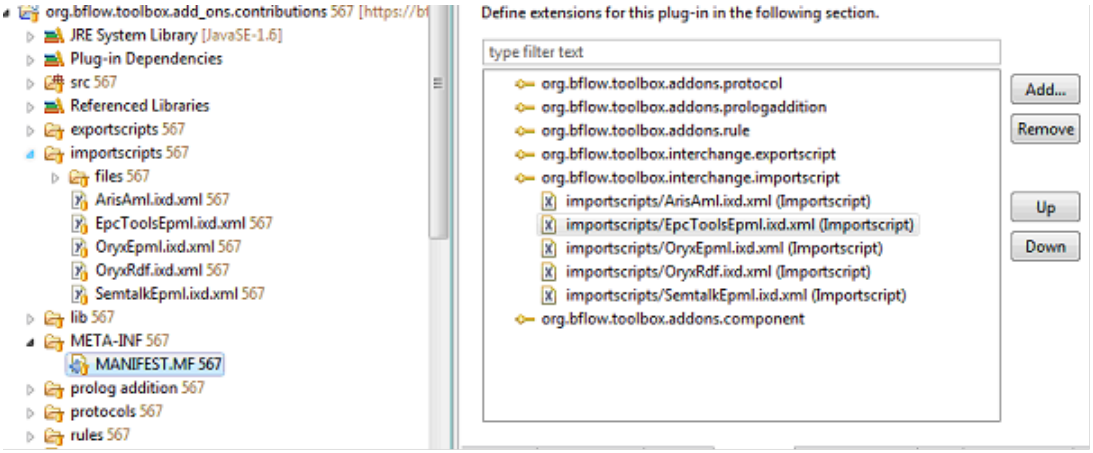

Abbildung 1.4: Hinzufügen eines Importskriptes über einen Extension-Point

Achtung! Wenn ein Export bzw. Import schon erfolgreich per Einbinden in das Unterverzeichnis .export bzw. .import des Arbeitsbereichs getestet wurde, muss trotzdem noch eine kleine Änderung an der Beschreibungsdatei vorgenommen werden:

Steht in einer Beschreibungsdatei im Verzeichnis .export:

```
@script: '/files/abc.vt'
```

so muss beim Einbinden in die bflow*-Quellen stehen:

```
@script: '/exportscripts/files/abc.vt'
```

2 Die Add-on-Schnittstelle

Add-ons bieten dem Entwickler und Anwender vielfältige Möglichkeiten, den Funktionsumfang der bflow* Toolbox auf verschiedene Weisen zu erweitern. Eines der Hauptmerkmale ist es, externe Programme in die Bearbeitung und Analyse von Modellen integrieren zu können.

Damit ist es möglich:

- den Modellinhalt in anderen Programmen zu nutzen und zu analysieren (z.B. für eine Simulation)
- die Ergebnisse dieser externen Programme in bflow* anzuzeigen
- als Ergebnis der Ausführung der externen Programme Modellelemente im bflow*-Modell zu hinzuzufügen, zu löschen oder zu ändern.

Das Prinzip dabei ist denkbar einfach: Jede Interaktion mit einem externen Programm wird durch eine Add-on-Definition geregelt. Die folgenden Abschnitte zeigen an einem Beispiel, wie externe Programme in Add-ons verwendet werden können.

2.1 Welche Programme lassen sich einbinden?

Über die Add-on-Schnittstelle lassen sich alle Programme, die auf dem verwendetem Betriebssystem selbstständig lauffähig sind, einbinden. Dabei spielt es keine Rolle, ob es sich um eine exe-, jar- oder Batch-Datei (bzw. ein Shell-Skript) handelt. Lässt sich das Programm manuell per Kommandozeile oder Doppelklick starten, so kann es auch von bflow* gestartet werden.

2.2 Verwenden von Add-ons

Die Verwendung von Add-ons gestaltet sich sehr einfach. Alle korrekt installierten und konfigurierten Add-ons werden im bflow*-Menü unter „Add-ons“ angezeigt.

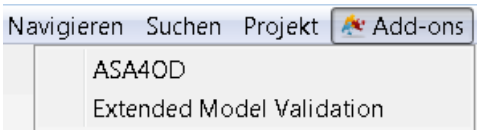

Abbildung 2.1: Add-on-Menü der Modellierungsumgebung

Um ein Add-on auszuführen, muss auf den Namen des entsprechenden Add-ons geklickt werden. Der Name bleibt grau, wenn das Add-on auf den gerade bearbeiteten Modelltyp nicht anwendbar ist oder wenn die notwendingen Dateien (die ausführbare Datei oder ein notwendiger Export) nicht gefunden werden können.

Vor dem Ausführen eines Add-ons muss das Modell, üfr das das Add-on ausgeführt werden soll, gespeichert werden: (Abbildung 2.2).

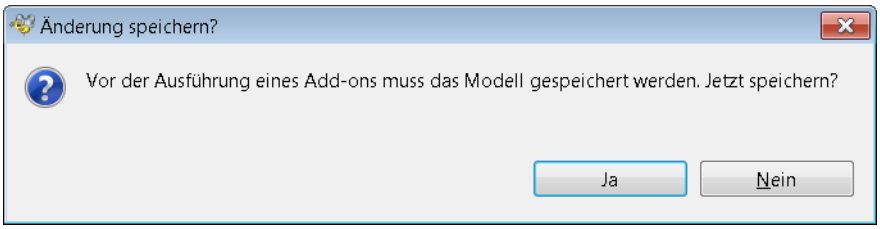

Abbildung 2.2: Speichern des Diagramms vor der Ausführung eines Add-ons

2.3 ASA4OD - Ein Beispiel-Add-on mit einem externen Java-Programm

In diesem Abschnitt wird anhand eines Beispiels erläutert, wie ein eigenes Add-on eingerichtet werden kann. Als Beispielprogramm soll das für diese Dokumentation erstellte und in Java geschriebene Programm „ASA4OD“ verwendet werden. Alle Quellen sind im Anhang zu finden und können von der Sourceforge-Projektseite von bflow* heruntergeladen werden.

2.3.1 Vorbereitung

Der Quelltext des Programms bzw. die für das Programm benötigten Exportskripte befinden sich im Anhang. Die Skripte, eine ausführbare Version des Programms, eine importierbare Add-on-Definition und eine Beispiel-EPK können von `http://sourceforge.net/projects/bflowtoolbox/files/bflow_%20Documentation/` heruntergeladen werden. Das Programm benötigt eine JRE in Version 1.7 oder höher.

2.3.2 Einrichten des Exports

Damit dem Beispiel-Add-on Daten, welche es weiterverarbeiten kann, übergeben werden können, muss zunächst ein Export eingerichtet werden (vgl. Abschnitt 1.1). Dazu muss im aktuellen Arbeitsbereich ein Ordner „.export“ angelegt werden. Dies sollte nicht in bflow*, sondern von der Kommandozeile des Betriebssystems aus erledigt werden. In diesen Ordner wird nun die Exportbeschreibung „asa4od.exd“ und in einen Unterordner

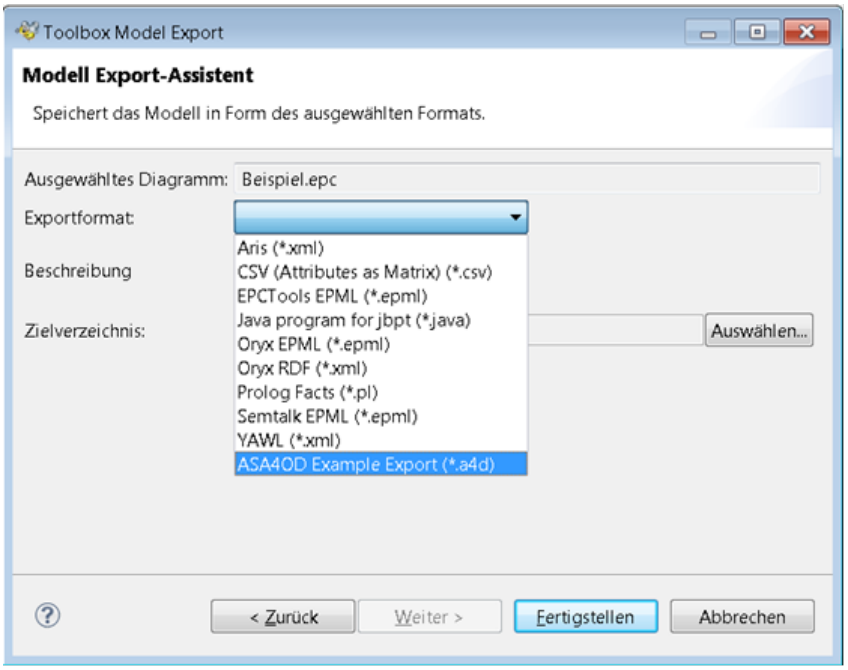

Abbildung 2.3: Beispielexport in der Liste der Modellexporte

„files“ das Skript „asa4od.vt“ kopiert. Anschließend ist ein Neustart von bflow* erforderlich.

Um zu überprüfen, ob der Export erfolgreich hinzugefügt wurde, kann versucht werden eine EPK zu exportieren. Wurde der Export richtig erkannt, sollte er in der Liste der Exportformate auftauchen (vgl. Abbildung 2.3). (Anmerkung: Oft wird ein Export nur für die Benutzung eines speziellen Add-ons nötig sein. Dann will man gar nicht, dass dieser in der Liste der unterstützten Exportformate auftaucht. Man erreicht das, indem das Wort “public” in der Exportbeschreibung weggelassen wird.)

Der Beispielexport ist für .epc-Dateien zu verwenden und durchläuft alle Elemente auf der Suche nach zur Ausführung des Skriptes markierten Elementen. Diese werden als ein Tripel, bestehend aus dem Typ des Elements, der Element-ID und dessen Name, zeilenweise aufgeschrieben. Am Ende wird noch die Anzahl der im Diagramm vorhandenen Kanten ausgegeben.

2.3.3 Das Beispiel-Programm registrieren

Im zweiten Schritt muss das Beispiel-Programm in Form der ausführbaren Datei „asa4od.jar“ in bflow* registriert werden. Unter dem Menüpunkt „Fenster“ → „Benutzervorgaben“→ „Bflow“ → „Tools“ können externe Programme (Tools), hinzugefügt werden.

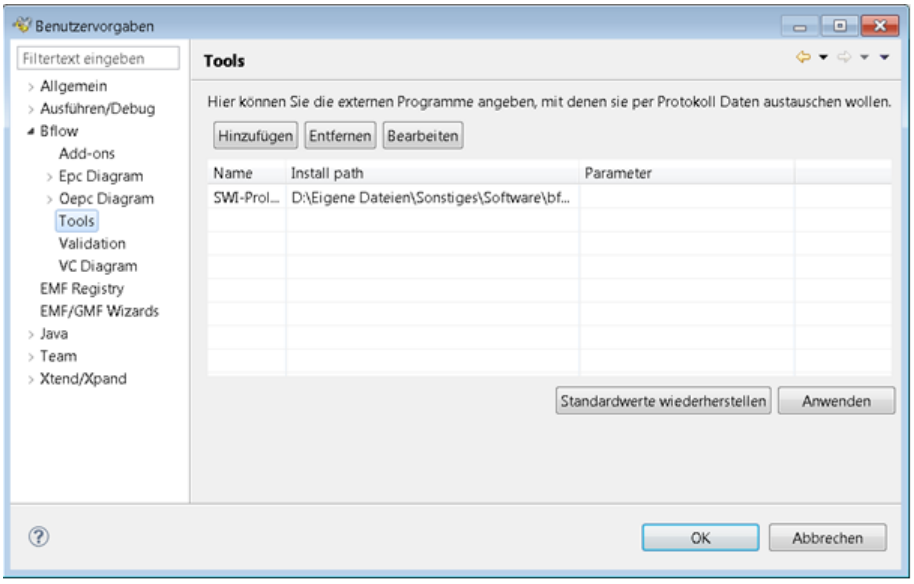

Abbildung 2.4: Übersicht der registrierten Programme

Wird nun ein neues Tool hinzugefügt, müssen ein Name, der Pfad zum ausführbaren Programm und die (falls nötig) Aufrufparameter angegeben werden. Diese können z. B. wie folgt aussehen.

Name: ASA4OD

Pfad: D:\Eigene Dateien\asa4od.jar

Aufrufparameter: $source

Das fertige Dialogfenster sollte in etwa in wie in Abbildung 2.5 aussehen.

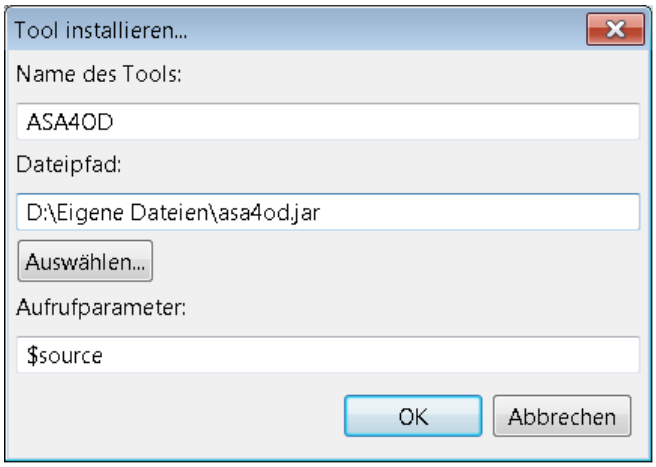

Abbildung 2.5: Ausgefüllter Dialog zum Registrieren eines externen Programms

Anmerkung:

- Unter „Dateipfad" muss immer der vollständige Pfad zur ausführbaren Datei angegeben werden. Die Angabe des Dateinamens ohne Pfadangabe reicht nicht - auch, wenn das Verzeichnis in der Umgebungsvariablen PATH des Betriebssystems enthalten ist.
- Wie man am Beispiel oben sieht, können jar-Dateien ebenso wie ausführbare Dateien direkt als Tool eingebunden werden. Das funktioniert aber nur, wenn sich ein in einer jar-Datei enthaltenes Programm auch per Doppelklick von der Oberfläche des Betriebssystems starten lässt. Unter Windows ist das allzu oft nicht der Fall. Dann muss entweder durch Einträge in der Registry dafür gesorgt werden, dass sich jar-Dateien per Doppelklick vom Explorer starten lassen. Oder aber man gibt unter „Dateipfad" den vollständigen Pfad zu java.exe an, und als ersten Aufrufparameter „-jar vollständiger_Pfad_zur_jar_Datei".
- Die Aufrufparameter sind Argumente, mit denen das Programm gestartet wird. Hier können prinzipiell alle beliebigen Werte eingetragen werden. Daneben gibt es vordefinierte Add-on-Schlüsselwerte, die stets mit „$" beginnen. Im obigen Fall wird bewirkt, dass der Ausdruck „$source" durch den Pfad der exportierten Modelldatei ersetzt wird. Beim Exportvorgang während der Ausführung eines Add-ons werden die Diagramme im temporären Verzeichnis des Betriebssystems gespeichert. Nachdem das Add-on beendet wurde, werden diese, sowie alle anderen temporären Dateien, die während des Vorgangs erzeugt wurden, gelöscht. Eine Liste mit allen reservierten Programmparametern ist im Abschnitt 2.4 zu finden. Durch Bestätigen mit „Ok" sollte sich der Dialog schließen und „asa4od" in der Liste der Tools aufgenommen worden sein (siehe Abbildung 2.6).

2.3.4 Das Add-on einrichten

Nun muss der Punkt „Add-ons" ausgewählt werden. Es ist eine Übersicht der bereits eingerichteten Add-ons zu sehen. Um Näheres zu einem Add-on zu erfahren, kann die Maus darüber gehalten werden, und ein Tooltipp mit einer Beschreibung erscheint.

Mit Drücken auf „Hinzufügen" kann ein neues Add-on definiert werden.

Der Name des Add-ons ist beliebig. Es ist der Name, der dann im bflow*-Menü "Add-ons" angezeigt wird. Er könnte z.B. „ASA4OD" lauten.

Der Beschreibungstext ist ebenfalls beliebig, sollte aber aussagekräftig sein. Er könnte z.B. wie folgt lauten:

„Das Programm ermittelt die zur Ausführung markierten Ereignisse und Funktionen und gibt diese auf der Ansicht „Fehler" aus. Weiterhin werden Schriftstil und -größe von Funktionen bzw. Ereignissen verändert."

Nun müssen die Schritte, die beim Ausführen eines Add-ons durchzuführen sind, beschrieben werden. Diese Schritte bezeichnen wir als **Komponenten**.

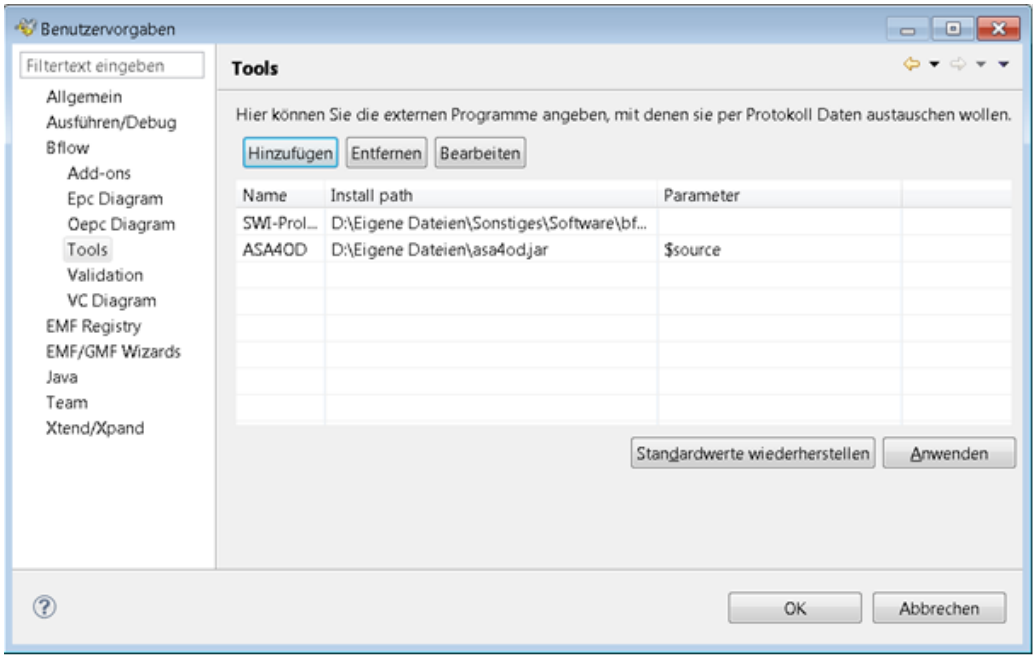

Abbildung 2.6: Übersicht der registrierten Programme jetzt mit ASA4OD

Mit Drücken auf "Hinzufügen" lässt sich eine Komponente hinzufügen. Darunter ist ein Teilschritt bei der Abarbeitung des Add-ons zu verstehen. Zunächst erscheint eine mit "Auswahlen" beschriftete Auswahlliste. Wir wählen zunächst „Diagram export". Anschließend kann über das Parameterfeld, welches in diesem Fall ebenfalls eine Auswahlliste ist, das zu exportierende Format ausgewählt werden.

Es stehen alle im bflow*-Lieferumfang enthaltenen sowie die selbst hinzugefügten Exportformate zur Verfügung. In unserem Beispiel wählen wir den gerade selbst hinzugefügten Export mit dem Namen „ASA4OD Example Export".

Um Näheres zu einer Komponente zu erfahren, kann der Mauszeiger über diese gehalten werden, und ein Tooltipp mit einer Beschreibung erscheint.

Eine weitere Komponente muss hinzugefügt und nun „Tool run" ausgewählt werden. Diese Komponente bewirkt den Start eines externen Programms. Als Parameter stehen die unter „Fenster" → „Benutzervorgaben"→ „Bflow" → „Tools" registrierten Tools zur Verfügung. Für unser Beispiel wählen wir „ASA4OD".

Nun muss die Komponente „Shell analysis" hinzugefügt werden. Diese Komponente analysiert die Konsolenausgaben, die von dem gestarteten Programm erstellt werden. Bei dieser Analyse werden aber nicht alle Informationen verarbeitet, sondern nur die, die eine festgelegte Struktur besitzen (vgl. Abschnitt 2.6). Diese Komponente wird benötigt, da ASA4OD seine Ausgabe in die Konsole (Standardausgabe) schreibt. Schreibt ein Programm seine Ergebnisse in eine Datei, so müsste die „File analysis"-Komponente ausgewählt werden. Hier ist das aber nicht der Fall.

Wird nun versucht, in das Parameterfeld zu klicken, passiert nichts. Das liegt daran, dass nicht jede Komponente Parameter benötigt. Falls eine Komponente ohne Parameter auskommt, so ist das entsprechende Feld nicht veränderbar.

Als nächstes wird die „Attribute adjust“-Komponente benötigt. Diese Komponente durchsucht die analysierten Meldungen nach Kommandos, um Attribute anzulegen oder anzupassen. Parameter werden hier ebenfalls nicht benötigt.

Abschließend muss die „Problems view display“-Komponente hinzugefügt werden. Diese Komponente erzeugt Meldungen in der Ansicht „Fehler”. Parameter werden nicht benötigt.

Die Add-on-Definition sollte nun wie in Abbildung 2.7 aussehen.

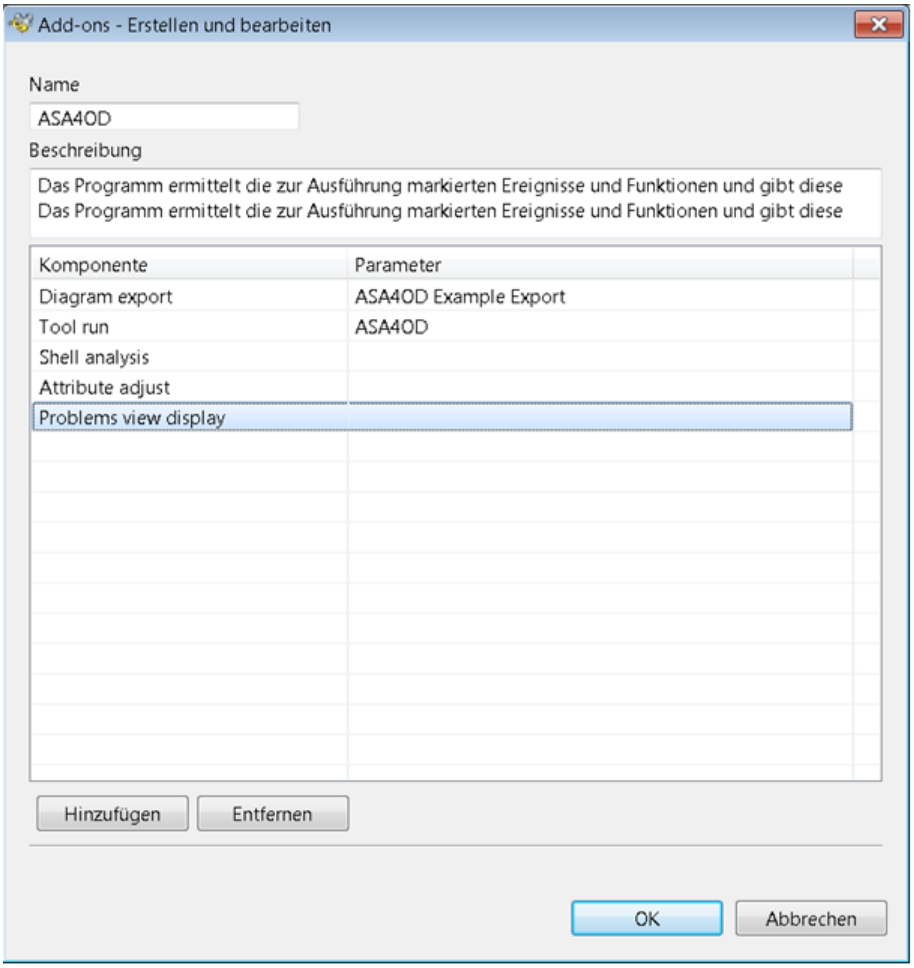

Abbildung 2.7: Vollständig ausgefüllte Add-on-Definition

Mit einem Klick auf „Ok“ wird die Definition gespeichert und erscheint nun in der Add-on-Liste.

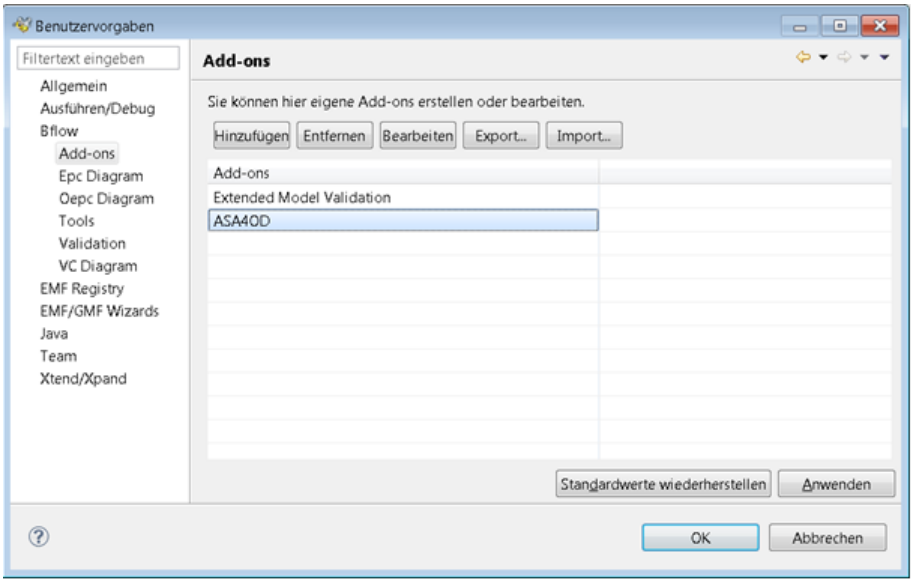

Abbildung 2.8: Liste der Add-ons jetzt mit ASA4OD

Das Add-on ist nun sofort (ohne Neustart) einsatzbereit und über das Add-on-Menü ausführbar.

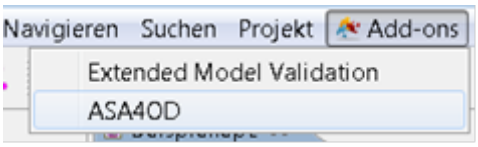

Abbildung 2.9: Add-on-Menü der Modellierungsumgebung

2.3.5 Der schnellere Weg: Die Add-on-Definitionsdatei

Existiert bereits eine zuvor exportierte Add-on-Definitionsdatei, wie z. B. ASA4OD.xml, sind weniger Schritte notwendig, um zum gleichen Ergebnis zu gelangen.

Unter dem Menüpunkt „Fenster“ → „Benutzervorgaben“ → „Bflow“ → „Add-ons“ muss der Button „Import“ gewählt und anschließend über den Datei-Dialog die Add-on-Definition „ASA4OD.xml“ ausgewählt werden. Sollte noch kein Tool namens „ASA4OD“ existieren, wird unter dem Punkt „Bflow“ → „Tools“ ein Eintrag angelegt. Dieser muss ggf. noch bearbeitet und der Pfad zum Programm angepasst werden.

Nun ist das Add-on eingerichtet und kann genutzt werden.

Beim Erstellen eines Add-ons ist zu empfehlen, dem Benutzer eine solche Add-on-Definitionsdatei zur Verfügung zu stellen („Fenster“ → „Benutzervorgaben“ → „Bflow“ → „Add-ons“ → „Export“.

Hinweis: Wenn die Add-on-Definitionsdatei mit einer älteren Version von bflow* erstellt wurde, ist es möglich, dass sie sich nicht korrekt importieren lässt.

2.3.6 Was beim Ausführen des Add-ons passiert

Wird das Add-on gestartet, werden die in der Add-on-Definition festgelegten Komponenten nacheinander ausgeführt.

Zunächst wird das zum Start geöffnete Modell exportiert. Dabei wird das Exportskript „asa4od.vt“ ausgeführt. Die von dem Skript erzeugten Ausgaben werden in ein temporäres Verzeichnis des Benutzers geschrieben.

Im zweiten Schritt führt bflow* das registrierte „ASA4OD“-Tool aus. Da in der Definition des Tools als Parameter „$source“ angegeben ist, wird dem Programm beim Start der Pfad zur temporären Datei, welche zuvor durch das Skript erzeugt wurde, als Parameter übergeben. Nun kann das Programm die in „args[0]“ befindliche Pfadangabe nutzen, die Datei einlesen und die Informationen weiterverarbeiten. Im Laufe der Verarbeitung werden Ausgaben auf die Laufzeitumgebung (Standardausgabe) geschrieben.

Die im vorherigen Schritt erzeugte Ausgabe auf der Standardausgabe wird nun durch die „Shell Analysis“-Komponente eingelesen und auf Zeichenketten, welche ein bestimmtes Muster erfüllen, hin untersucht (vgl. Abschnitt 2.6). Diese Muster fangen mit „addon:“ an und werden auch vom „ASA4OD“-Tool erzeugt (vgl. ASA4OD.java: printFormatString(...)-Methode). Die Zeichenketten, welche einer bestimmten Formkonvention entsprechen, können dann von der „Attribute adjust“- bzw. der „Problems view display“-Komponente weiterverarbeitet werden.

Die „Attribute adjust“-Komponente berücksichtig all jene Zeichenketten, welche dem Formattyp „ATTRIBUTE“ entsprechen. Auf diese Weise werden Änderungen an Attributen des Modells bzw. Modellelementen durchgeführt.

Die „Problems view display“-Komponente berücksichtigt hingegen nur all jene Zeichenketten, welche dem Formattyp „MESSAGE“ entsprechen. Je nach Inhalt der Zeichenketten werden so Informationen, Warnungen bzw. Fehler in der Ansicht „Fehler” von bflow* angezeigt

2.4 Tool-Parameter

In dem folgenden Abschnitt werden die reservierten Parameter für Programmaufrufe (Tool-Run) erläutert. Sollten Unsicherheiten bestehen, wofür diese Parameter verwendet werden können, kann dies im Abschnitt 2.3.3 noch einmal nachgelesen werden.

Parameter	Bemerkung
$source	Ersetzt beim Aufruf des Programms das Schlüsselwort durch die Datei, die bearbeitet werden bzw. als Eingabe dienen soll. Beispiel: „-f:$source“ → „-f:C:\bflow\lagerverwaltung.epml“
$wsSource	Ersetzt beim Aufruf des Programms das Schlüsselwort durch die Datei des Diagrammmodells im Arbeitsbereich des Modellierungswerkzeugs. Damit erhält man den Dateinamen der ursprünglichen Modelldatei. Beispiel: „-d:$wsSource“ →„-d:C:\bflow\workspace\seminars\semA.ood“
$project	Ersetzt beim Aufruf des Programms das Schlüsselwort durch den absoluten Pfad zum Projekt (im bflow*-Arbeitsbereich), das die zu bearbeitend Modelldatei beinhaltet. Beispiel: „-pr:$project“ → „-pr:C:\bflow\workspace\seminars“
$file1, $file2, . . . , $file10, . . .	wird benutzt, wenn die Add-on-Definition vorsieht, dass mehrere Dateien erzeugt werden, die während dessen Ausführung benötigt werden Angenommen, eine Add-on-Definition beinhaltet drei externe Programmaufrufe. Programm 1 und Programm 2 erzeugen jeweils eine Datei. Programm 3 muss mit diesen Dateien und der Quelldatei arbeiten. Dann sollte Programm 3 folgende Parameter besitzen: „$source $file1 $file2“
$addon_temp	Dieser Parameter liefert den Namen des temporären Verzeichnisses, in dem bei der Ausführung des Add-ons Zwischenergebnisse (einschl. der exportierten Modelldatei) gespeichert werden. Beispiel: „$addon_temp“ → „C:\Windows\temp\“
$install_dir	Dieser Parameter liefert einen Zeiger auf das Verzeichnis, in dem das gestartete externe Programm (Tool) installiert ist. Beispiel: „$install_dir“ → „D:\bflow\addons\epcmetrik\“

Tabelle 2.1: Toolparameter

2.5 Add-on-Komponenten

In dem folgenden Abschnitt werden die verfügbaren Add-on-Komponenten erläutert. Sollten Unsicherheiten bestehen, wofür diese Komponenten verwendet werden können, kann

dies im Abschnitt 2.3.4 noch einmal nachgelesen werden.

Komponente	Beschreibung
Attribute adjust	Diese Komponente muss verwendet werden, wenn Modellattribute verändert werden sollen. Sie sorgt dafür, dass die neuen und geänderten Attribute erkannt und das Modell entsprechend angepasst wird. Damit diese Erkennung korrekt funktioniert, muss die Information aus dem externen Programm einer Formkonvention entsprechen (siehe Abschnitt 2.6). Diese Komponente kann nur nach der „File analysis“ oder „Shell analysis“-Komponente verwendet werden. Sie benötigt keine Parameter.
Console display	Sollen die Ausgaben des externen Programms in der bflow*-Ansicht „Konsole” angezeigt werden, so kann dies mit dieser Komponente eingestellt werden. Sie nimmt dabei keine Veränderungen an diesen Informationen vor und gibt sie anschließend an die nächste Komponente weiter. Diese Komponente kann nur nach der „File analysis“, „Shell analysis“ oder „Shell record“-Komponente verwendet werden. Sie benötigt keine Parameter.
Diagram export	Diese Komponente exportiert das zu bearbeitende Diagramm in ein gewünschtes Format. Als Format stehen alle installierten Exporte zur Verfügung. Im Normalfall ist dies stets die erste Komponente einer Add-on-Definition! Daher kann diese Komponente keiner anderen Komponente folgen. Als Parameter stehen in einer Auswahlliste alle installierten Exporte zur Verfügung.
File analysis	Diese Komponente kann genutzt werden, wenn das externe Programm seine Ergebnisse in eine Datei schreibt. Die Komponente analysiert diese Datei nach Informationen, die für das Add-on-Plugin verwendbar sind. Dies können Meldungen für die bflow*-Ansicht „Fehler” oder Attributänderungen sein. Damit diese Erkennung korrekt funktioniert, muss die Information aus dem externen Programm einer Formkonvention entsprechen. (siehe Abschnitt 2.6). Diese Komponente kann nur nach der „Tool run“ oder „Prolog run“-Komponente verwendet werden. Sie benötigt keine Parameter.

File record	Sollen lediglich die Ergebnisse, die ein externes Programm in eine Datei schreibt, ohne Analyse aufgezeichnet werden, so muss diese Komponente verwendet werden. Da keine Analyse stattfindet, müssen auch keine Konventionen eingehalten werden. Dies kann aber je nach Ausgabe des externen Programms zur Darstellung von unlesbaren Zeichen führen. Sollen die Informationen in der bflow*-Ansicht „Konsole" angezeigt werden, muss die „Console display"-Komponente nach dieser eingefügt werden. Diese Komponente kann nach der „Tool run" oder „Prolog run"-Komponente verwendet werden. Sie benötigt keine Parameter.
HTML View	Anzeigen einer HTML-Datei mit dem in Eclipse definierten Programm zur Anzeige von HTML (dies kann ein interner Viewer oder ein externes Programm sein) Als Parameter muss die anzuzeigende HTML-Datei angegeben werden.
Problems view display	Soll das Ergebnis der Analyse der Daten des externen Programms in der bflow*-Ansicht „Fehler" angezeigt werden, so kann diese Komponente verwendet werden. Die Ausgaben des externen Programms müssen dazu einer Formkonvention entsprechen aus dem externen Programm einer Formkonvention entsprechen (siehe Abschnitt 2.6). Diese Komponente kann nach der „Shell analysis"-, „File analysis" oder „Attribute adjust"-Komponenten eingesetzt werden. Wenn diese Komponente verwendet werden soll, so sollte sie im Normalfall die letzte in der Add-on-Definition sein. Sie benötigt keine Parameter.
Prolog run	Diese Komponente startet den Prolog-Interpreter. Sie kann nach der „Diagram export"- oder „Tool adapter"-Komponente genutzt werden. Die Komponente benötigt zwingend zwei Parameter: „-pl:" und „-p". Wofür diese Parameter sind und welche weiteren es gibt, kann im Abschnitt 2.9.2 nachgelesen werden.

Shell analysis	Diese Komponente kann genutzt werden, wenn das externe Programm seine Ausgaben auf die Konsole (Standardausgabe) schreibt. Die Komponente analysiert diese Ausgabenebung nach Informationen, die für das Add-on-Plugin verwendbar sind. Dies können Meldungen für die bflow*-Ansicht „Fehler" oder Attributänderungen sein. Damit diese Erkennung korrekt funktioniert, muss die Information aus dem externen Programm einer Formkonvention entsprechen (siehe Abschnitt 2.6). Diese Komponente kann nur nach der „Tool run" oder „Prolog run"-Komponente verwendet werden. Sie benötigt keine Parameter.
Shell record	Sollen lediglich die Ergebnisse, die das externe Programm auf die Konsole (Standardausgabe) schreibt, ohne Analyse aufgezeichnet werden, so kann diese Komponente verwendet werden. Da keine Analyse stattfindet, müssen auch keine Konventionen eingehalten werden. Dies kann aber je nach Ausgabe des externen Programms zur Darstellung von unlesbaren Zeichen führen. Sollen die Informationen in der bflow*-Ansicht „Konsole" angezeigt werden, so muss im nächsten Schritt die „Console display"-Komponente eingefügt werden. Diese Komponente kann nach der „Tool run" oder „Prolog run"-Komponente verwendet werden. Sie benötigt keine Parameter.
Tool adapter	Sollen mehrere externe Programme hintereinander ausgeführt werden, so muss diese Komponente als Brücke zwischen beiden verwendet werden. Sie prüft, welche Eingabe das vorhergehende Programm für das nächstfolgende bereitstellt. Es prüft außerdem, ob eine Adaption überhaupt möglich ist und nimmt ggf. Anpassungen vor. Diese Komponente kann nur nach der „Tool run"- oder „Prolog run"-Komponente eingesetzt werden. Achtung: Wenn Programm A das richtige Format für Programm B bereits bereitstellt und somit keine Anpassung nötig ist, so sollte als Parameter aus der Auswahlliste „equal" ausgewählt werden. Der Adapter leitet in diesem Fall nur die Pfadangabe bzw. Prozesslaufzeitumgebungsausgabe weiter. Wird eine Konvertierung von der auf die Standardausgabe geschriebene Ausgabe in eine Datei benötigt, so muss „shell to file" als Parameter verwendet werden, für den umgekehrten Fall „file to shell".

Tool run	Mit dieser Komponente können beliebige externe Programme gestartet werden. Bedingung dafür ist, dass sich das Programm innerhalb des Betriebssystems selbständig starten lässt. Als Parameter stehen die unter „Fenster“ → „Benutzervorgaben“→ „Bflow“ → „Tools“ registrierten Tools zur Verfügung. Diese Komponente kann nur nach der „Diagram export“- oder „Tool adapter“-Komponente ausgeführt werden.

2.6 Formkonvention zum Erzeugen von Ausgaben und Modelländerungen

In diesem Abschnitt soll erklärt werden, welche Form die Meldungen der externen Programme haben muss, damit die Ergebnisse eines gestarteten Tools dazu führen, dass Anzeigen in der bflow*-Ansicht „Fehler” erfolgen oder Änderungen am Diagramm vorgenommen werden. Diese Meldungen können entweder in einer Datei stehen („File analysis“-Komponente) oder in die Prozesslaufzeitumgebung („Shell analysis“-Komponente) geschrieben werden.

Für Informationen, die ausgewertet werden sollen, gilt generell:

- Informationen werden zeilenweise erfasst. Gibt es innerhalb der Ausgabe einen Zeilenumbruch, so wird der nachfolgende Inhalt als neue Ausgabezeile und somit neue Information interpretiert.
- Jede Zeile , die vom Add-on-Plugin verarbeitet werden soll, muss mit dem Präfix „addon:“ beginnen. Alle anderen Ausgabezeilen werden ignoriert.

Soll aus einer Ausgabezeile eine Meldung für die Problems-View erzeugt werden, so muss die folgende Struktur erfüllt sein:

`[MESSAGE][TYP][ELEMENT ID][MELDUNG][ZUSATZ]#FS#`

Message	Gibt den Typ der Meldung an. Ist Programmkonstante und muss erscheinen!
Typ	Muss „INFO“, „WARNING“ oder „ERROR“ sein
Element id	Optional. Wenn vorhanden, dann wird die Meldung mit dem Modellelement mit der entsprechenden ID verknüpft.
Meldung	Nachricht, die angezeigt werden soll
Zusatz	Optional. Wird nicht ausgewertet

Beispiel:

```
1 addon:[MESSAGE][INFO][_eah882k094][Beschriftung fehlt!][]#FS#
2 addon:[MESSAGE][ERROR][_fgk19o78h][Element unnötig][]#FS#
```

$name	Legt den Namen des Modellelements fest
$fontname	Legt die Schriftart fest, die das Modellelement verwendet
$fontsize	Legt die Größe der Schrift fest, die das Modellelement verwendet
$fontstyle	Legt den Schriftstil fest, den das Modellelement verwendet (BOLD = fettdruck, ITALIC = kursiv, NORMAL = normal)
$location	Bestimmt die Position eines Modellelements im Diagramm. Dabei kann die Position entweder absolut oder relativ vorgegeben werden. Für eine relative Positionierung müssen Vorzeichen verwendet werden.
$color	(nur für Kanten): Kantenfarbe in RGB-Notation, also etwa [255,0,70]
$strokewidth	(nur für Kanten): Strichstärke

Tabelle 2.3: Grafische Elementattribute

Soll eine Ausgabezeile bewirken, dass Änderungen am aktuellen Modell vorgenommen werden, so muss die folgende Struktur erfüllt sein:

`[ATTRIBUTE][TYP][ELEMENT ID][NAME][WERT]#FS#`

Attribute Gibt den Typ der Meldung an. Ist Programmkonstante und muss erscheinen!

Typ Muss „ADD“, „SET“ oder „REMOVE“ sein

Element id ID des dazugehörigen Modellelements

Name Name des Attributs

Wert Wert des Attributs

Beispiel:

```
1 addon:[ATTRIBUTE][SET][_eah882k094][Planungskosten][1200]#FS#
2 addon:[ATTRIBUTE][ADD][_fgk19o78h][Gezählte Anfragen][21]#FS#
3 addon:[ATTRIBUTE][REMOVE][_uzkkla182f][Erwartete Durchläufe][]#FS#
```

Das Add-on-Plugin unterstützt zwei Arten von Attributen: Modell- bzw. Elementattribute und grafische Attribute. Modell- und Elementattribute sind solche, die dem Modell bzw. jedem Modellelement mittels der bflow*-Ansicht „Attribute-View“ zugeordnet werden können.

Grafische Attribute sind der Name eines Modellelements sowie die zur Darstellung verwendete Schriftart, Schriftgröße, Schriftstil (fett, kursiv, unterstrichen) sowie die Position eines Modellelements. Wenn diese Eigenschaften verändern werden sollen, so werden als Name des Attributs die folgenden Schlüsselwörter benötigt.

Eine Ausgabe könnte dann wie folgt aussehen:

```
1 addon:[ATTRIBUTE][SET][_eah882k094][$name][checked]#FS#
2 addon:[ATTRIBUTE][SET][_eah882k094][$fontname][Tahoma]#FS#
3 addon:[ATTRIBUTE][SET][_eah882k094][$fontsize][16]#FS#
4 addon:[ATTRIBUTE][SET][_eah882k094][$fontstyle][BOLD]#FS#
5 addon:[ATTRIBUTE][SET][_eah882k094][$location][540,120]#FS#
6 addon:[ATTRIBUTE][SET][_eah882k094][$location][+20,-60]#FS#
7 addon:[ATTRIBUTE][SET][_eah882k0z5][$color][255,0,70]#FS#
8 addon:[ATTRIBUTE][SET][_eah882k0z5][$strokewidth][2]#FS#
```

2.7 Arbeiten mit externen Dateien

Manchmal kann es notwendig sein, dass externe Programme ihr Ergebnis in eine Datei schreiben. Damit ein Add-on solche Tools nutzen kann, müssen in der Add-on-Definition die „File analysis“- bzw. die „File record“-Komponente genutzt werden. Weiterhin muss das Tool mitteilen, wo die externe Datei zu finden ist. Dies geschieht, in dem es eine Ausgabezeile auf die Konsole (Standardausgabe) schreibt. Diese Zeile muss mit dem Präfix „addon:“ beginnen. Nach dem Präfix folgt eine relative oder absolute Pfadangabe zur Datei, deren Inhalt aufgezeichnet bzw. analysiert werden soll.

Würde das Beispiel-Tool „asa4od.jar“ (siehe Abschnitt 2.3) seine Ausgaben nicht auf die Standardausgabe, sondern in eine externe Datei schreiben, müsste es eine Ausgabezeile mit dem Pfad auf die Konsole (Standardausgabe) schreiben. Diese Ausgabezeile könnte z. B. `addon:./result.dat` lauten, sofern die Datei im gleichen Ordner wie das Programm liegt. Schreibt das Programm z. B. in eine Datei, die im Verzeichnis /tmp/ liegt, könnte die Ausgabe `addon:/tmp/bflow/result.dat` lauten. Die dazugehörige Add-on-Definition müsste an das Verhalten angepasst und die „Shell analysis“- durch die „File analysis“-Komponente ersetzt werden. Somit würde sich das Add-on die benötigten Informationen aus der Datei holen.

Warnung: Vermeiden Sie Verzeichnisnamen mit Leer- und ähnlichen Sonderzeichen. Probleme kann es auch mit der Eigenart neuerer Windows-Versionen geben: Wenn das externe Programm in ein Verzeichnis schreiben soll, für das keine Schreibrechte existieren, wird der Schreibzugriff stillschweigend in das Verzeichnis \Users\Benutzername\AppData\Roaming\ umgeleitet. Ein Zugriff auf diese geschriebene Datei durch eine andere Komponente im bflow*-Addon kann dann fehlschlagen.

2.8 Weitergabe von Add-ons

Für einen leichteren Austausch von Add-ons zwischen Add-on-Anwendern besitzt die Add-on-Schnittstelle eine Im- und Exportfunktion für Add-ons. Damit können einfach eigene erstellte Add-ons an andere Add-on-Anwender weitergegeben oder deren Add-ons in eigenen Modellierungsumgebungen verwendet werden.

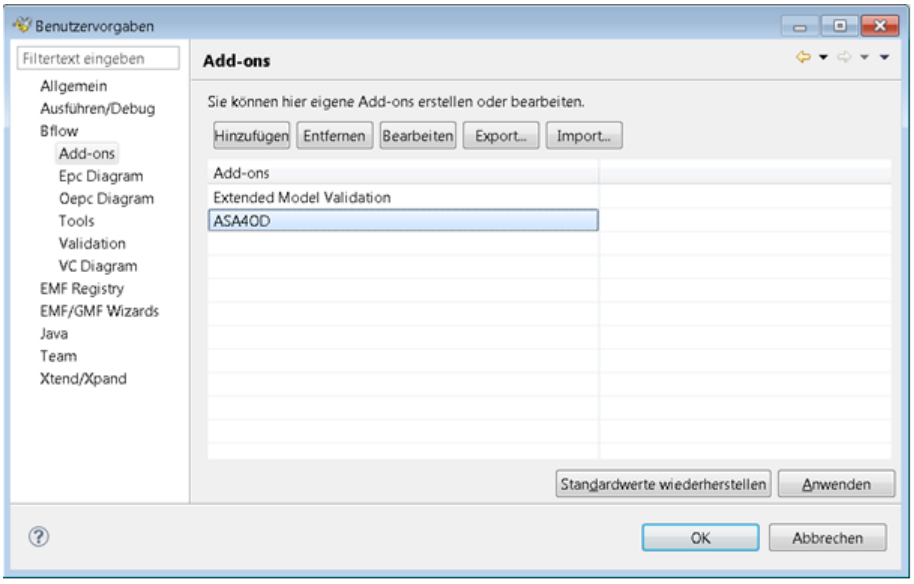

Abbildung 2.10: Übersicht installierter Add-ons

Unter dem Menü „Fenster“ → „Benutzervorgaben“ und dann „Bflow“ → „Add-ons“ befindet sich eine Übersicht über die bereits eingebundenen Add-ons (Abbildung 2.10).

Für den Export eines Add-ons müssen folgende Schritte durchgeführt werden:

Das zu exportierende Add-on muss markiert sein. Dann kann auf „Export“ geklickt werden. Es öffnet sich ein Dialog zum Speichern. Nach der Eingabe des Speicherortes und eines Namens für die Add-on-Definition ist der Vorgang mit Betätigung des Buttons „Speichern” abgeschlossen.

Sollte das Add-on Aufrufe von externen Programmen bzw. manuell hinzugefügten Exporte bzw. Importe enthalten, muss natürlich zusätzlich zur gespeicherten Add-on-Definition auch die zu startenden externen Programme bzw. die zu verwendenden Exporte weitergegeben werden.

Für den Import eines Add-ons müssen folgende Schritte durchgeführt werden:

Bei Betätigung des Buttons „Import“ öffnet sich ein Dialog. Nun kann eine zuvor exportierte Add-on-Definition ausgewählt und mit Betätigen des Buttons „öffnen“ importiert werden. Falls es sich um ein korrekt exportiertes Add-on handelt, ist es nun in der Liste der installierten Add-ons zu sehen.

Benötigt ein importiertes Add-on ein externes Programm (Tool), welches noch nicht eingetragen ist, so legt es die Rahmenbedingungen (Name und Parameter) automatisch an. Es muss dann noch der Pfad zum Programm eingetragen werden. Ebenso sind (falls nötig) Exporte zu definieren (siehe Abschnitt 2.3.2).

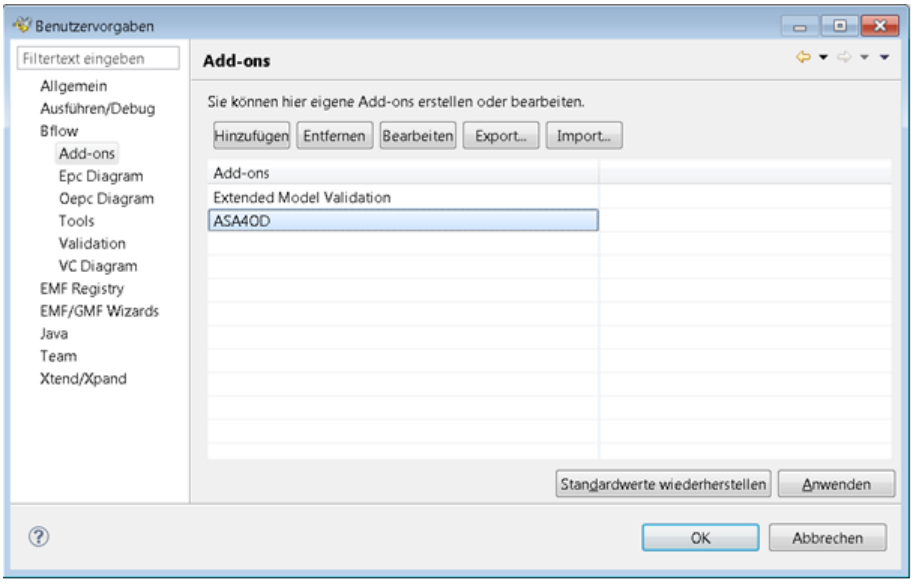

Abbildung 2.11: Übersicht installierter Add-ons

2.9 Verwendung von Prolog-Programmen in Add-ons

Eine mächtige Funktion von bflow* ist es, dass Prolog-Programme von bflow* aus gestartet werden können. Diese können insbesondere zur Analyse von Modellen eingesetzt werden.

Die Programme müssen für SWI-Prolog 5.6.64 lauffähig sein. (siehe `http://www.swi-prolog.org/`) Neuere Versionen von SWI-Prolog werden derzeit noch nicht unterstützt. SWI-Prolog muss dabei **nicht** gesondert installiert werden.

Der folgende Abschnitt erläutert, wie ein eigenes Prolog-Programm als Add-on eingebunden werden kann.

2.9.1 Add-on-Definition zum Start des Prolog-Programms

Es wird davon ausgegangen, dass ein entsprechendes Prolog-Programm vorhanden ist. Für dieses Beispiel heißt das Programm „konvertierungen.pl“. Die Ergebnisse dieses Programms werden auf die Konsole (Standardausgabe) geschrieben. Dazu kann der Prolog-Befehl `print` und zum Erzeugen eines Zeilenumbruchs der Befehl `nl` genutzt werden.

Da Prolog als Tool bereits angemeldet ist, muss nur noch eine Add-on-Definitioni erstellt werden.

Dazu muss das Menü „Fenster“ → „Benutzervorgaben“ geöffnet und anschließend „Bflow“ → „Add-ons“ ausgewählt werden. Ein Fenster wie in Abbildung 2.11 sollte zu sehen sein.

Mit einen Klick auf „Hinzufügen“ öffnet sich ein Dialog zum Erstellen eines Add-ons. Für den Namen kann z. B. „Prolog Konvertierung“ eingegeben werden. Der Name des Add-ons ist prinzipiell frei wählbar. Anschließend sollte ein kleiner Beschreibungstext eingegeben werden. Dieser könnte wie folgt lauten:

„Wandelt die Diagrammstruktur und dessen Elemente in ein anderes Format um.“

Der Beschreibungstext ist beliebig, sollte aber aussagekräftig sein. Nun kann mit der Einrichtung der Komponenten begonnen werden.

Durch Klicken auf „Hinzufügen“ erscheint in der Tabelle mit „ – Auswählen –“ beschriftete Auswahlliste. Aus dieser muss zunächst „Diagramm export“ gewählt werden. Anschließend muss das Parameterfeld angeklickt und aus der Auswahlliste der Eintrag „Prolog Facts“ ausgewählt werden. Dies bewirkt, dass die in einer EPK enthaltene Information in Prolog-Fakten transformiert wird (andere Diagrammtypen als EPK werden derzeit nicht unterstützt).

Als nächste Komponente wird „Prolog run“ ausgewählt. Diese Komponente benötigt im Wesentlichen zwei Parameter: den Pfad zum Prolog-Programm und zusätzliche Angaben zur Steuerung des Prolog-Programms. Folgender Text kann nach einem Klick auf das Parameterfeld eingefügt werden:

```
-pl:D:\\bflow\\pls\\konvertierungen.pl -p:x
```

Der Parameter „-pl“ gibt an, wo das auszuführende Prolog-Programm liegt. Mit dem Parameter „-p“ lassen sich zusätzliche Programmoptionen übergeben. Wenn das Prolog-Programm keine weiteren Parameter benötigt, reicht hier einfach ein „x“

Es existieren für den Aufruf von Prolog-Programmen weitere Parameter wie es bei den Add-on-Tools zu sehen ist. Diese beginnen ebenso typisch mit einem „$“. Eine vollständige Liste dieser Schlüsselworte und deren Verwendung ist im Abschnitt 2.9.2 zu finden.

Jetzt muss eine weitere Komponente hinzugefügt und „Shell record“ ausgewählt werden. Diese Komponente zeichnet die Ausgaben des Prolog-Programms auf der Konsole (Standardausgabe) auf.

Noch eine letzte Komponente muss hinzugefügt werden. Dabei handelt es sich um die Komponente „Console display“. Diese Komponente gibt gesammelte Informationen in der bflow*-Ansicht „Konsole” aus. Parameter werden nicht benötigt. Ebenso wäre es allerdings auch denkbar, dass das Prolog-Program wie unter Abschnitt 2.6 beschrieben, Anweisungen für Ausgaben in die bflow*-Ansicht „Fehler” oder zum Ändern von Modellelementen ausgibt. Tatsächlich ist das genau der Mechanismus, den das fest in bflow* eingebaute Add-on „Erweiterte Modellprüfung” nutzt.

Die Add-on-Definition sollte nun wie in Abbildung 2.12 dargestellt aussehen:

Mit einem Klick auf „OK“ wird die Add-on-Definition gespeichert, und in der Liste der verfügbaren Add-ons ist nun das Add-on „Prolog Konvertierung” enthalten.

Das Add-on ist nun sofort (ohne Neustart) einsatzbereit und über das bflow*-Menü „Add-ons” ausführbar.

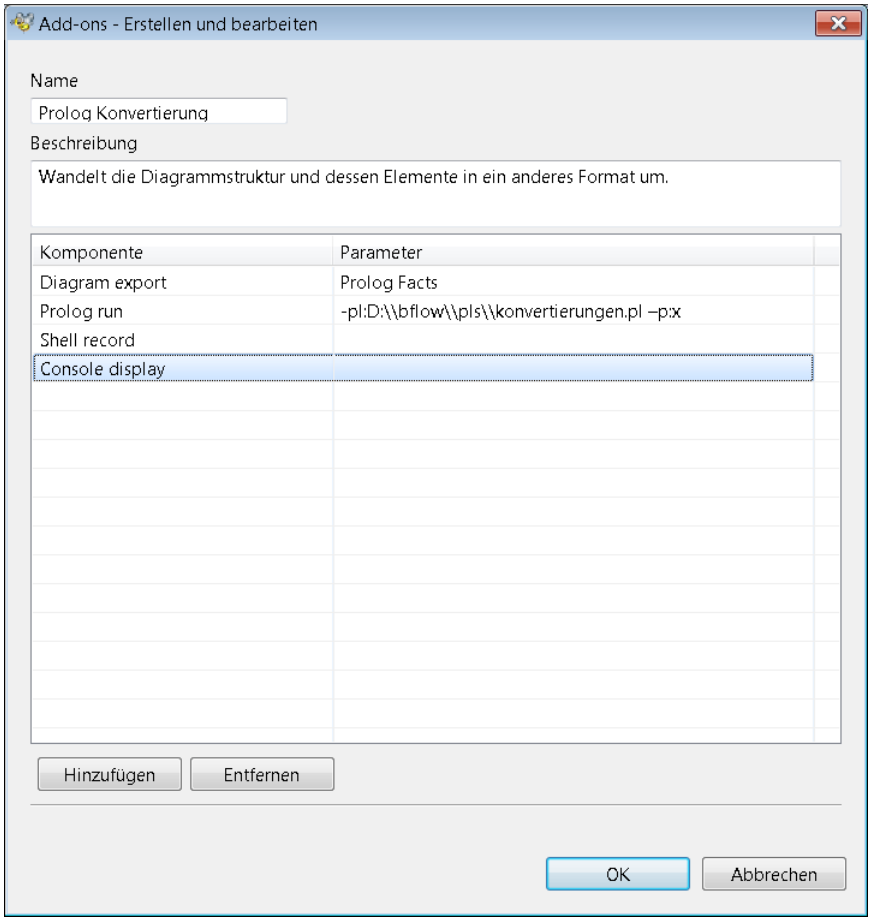

Abbildung 2.12: Fertig eingerichtetes Add-on mit Prolog

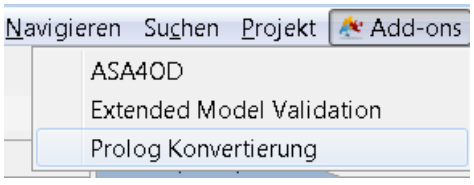

Abbildung 2.13: “Prolog Konvertierung” im Add-on-Menü

2.9.2 Aufrufparameter eines Prolog-Programms

Für die Ausführung der „Prolog run“-Komponente existieren reservierte Schlüsselwörter, die das Verhalten des Prolog-Interpreters steuern.

Parameter	Bemerkung
-pl:	Gibt an, welches Prolog-Programm verwendet werden soll. Die Angabe muss ein absoluter Pfad sein. Beispiel: -pl:C:\bflow\prolog\myEval.pl
-p:	Gibt eine Liste von Parametern an, die zum Start des Prolog-Programms übergeben werden soll. Die Angabe ist optional und kann Programmkonstanten beinhalten. Beispiel: (1) -p:simple,nonX,active (2) -p:$SETUP
-s:	Gibt die an das Prolog-Programm gestellte Anfrage an. Fehlt dieser Parameter, so wird das Prolog-Programm mit der Anfrage „addon_query(...)“ gestartet. Soll das Programm anders gestartet werden, so muss den entsprechend Aufruf ohne Klammern hier angegeben werden. Beispiel: (1) –s:a (2) –s:do_request
-e:	Gibt eine Ausgabezeile an, mit der das Prolog-Programm signalisiert, dass die Abarbeitung beendet ist. Fehlt dieser Parameter, so wird erwartet, dass das Prolog-Programm mit Schreiben der Ausgabezeile „#query_end#“ beendet wird. Wird ein Programmende durch eine andere Ausgabe angezeigt, so muss der Befehl hier ohne Klammern angegeben werden. Beispiel: (1) –e:STOP (2) –e:#request_finished#

Tabelle 2.4: Prolog-Parameter

2.9.3 Die im Prolog-Programm gestartete Anfrage

Die Anfrage, mit der das Prolog-Programm gestartet wird, muss im Prolog-Programm die folgende Form haben:

```
STARTBEFEHL ( SPRACHE, [PARAMETERLISTE]).
```

Als Startbefehl gilt, sofern kein eigener festgelegt wurde, die Zeichenkette „addon_query“. Danach folgt ein Sprachkürzel. Dies könnte „de“ oder „en“ usw. sein. Dieses wird auto-

matisch festgelegt und entspricht der in bflow* eingestellten Sprache. Abschließend wird eine Liste von Parametern angehängt. Diese Parameter entsprechen den Programmparametern, die mittels „-p:“ bei der Einrichtung des Prolog-Programms festgelegt werden können.

Angenommen, es wurde in einer Add-on-Definition der folgende Prolog-Aufruf festgelegt:

-pl:D:\bflow\pls\myEval.pl –p:short,lang,xtract –s:finde_was_tolles –e:query_end

Dies bewirkt, dass an das Prolog-Programm D:\bflow\pls\myEval.pldie Anfrage `finde_was_tolles` gestellt wird. Diese Anfrage hat zwei Parameter: Der erste Parameter ist `de` oder `en` (je nachdem, welche Sprache in bflow* aktuell eingestellt ist). Der zweite Parameter ist die Liste `[short,lang,xtract]`. Somit wird an das Prolog-Programm die Anfrage `finde_was_tolles(de,[short,lang,xtract])` gestellt.

Weiterhin wird dem Add-on-Plug-In mitgeteilt, dass dass Prolog-Programm das Ende der Abarbeitung dadurch mitteilt, dass es die Ausgabezeile „query_end“ schreibt.

2.10 Hinzufügen von Add-ons zu den bflow*-Quellen (für Entwickler)

Wenn Sie interessante Erweiterungen für die bflow* Toolbox programmieren, schreiben Sie doch bitte eine E-Mail an bflow@bflow.org. Vielleicht ist Ihre Erweiterung ja auch für andere Nutzer interessant und kann in die nächste Version von bflow* aufgenommen werden.

2.10.1 Hinzufügen eigener Komponenten zu den bflow*-Quellen

Sollte der Fall eintreten, dass die bereits vorhandenen Komponenten nicht ausreichend sind, so können mithilfe eines Eclipse-Extension-Points eigene Komponenten entwickelt und in Add-on-Definitionen eingesetzt werden.

Um eigene Komponenten verwenden zu können, muss eine Java-Klasse geschrieben werden, die das „IComponent“-Interface implementiert. Diese Schnittstelle ist Teil des „org.bflow.toolbox.addons“ Pakets. Anschließend muss die Komponente mithilfe des folgenden Extension-Points registriert werden.

„org.bflow.toolbox.hive.addons.component”

Nun kann die Komponente in Add-on-Definitionen eingesetzt werden.

Als einfaches Beispiel für eine Komponente kann der Quellcode der HTMLViewComponent studiert werden.

2.10.2 Hinzufügen eigener Add-ons zu den bflow*-Quellen

Zunächst muss eine Add-on-Definition angelegt und exportiert werden (siehe Abschnitt 2.3.5). Sehen wir uns einmal die bei diesem Export erzeugte XML-Datei an:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <protocol name="ExtendedModelValidation">
3      <display>
4          <default>extended model validation</default>
5          <de>Erweiterte Modellprüfung</de>
6      </display>
7      <description>
8          <default>Does some great analysis!</default>
9          <de>Ganz großartige Analyse!</de>
10     </description>
11     <components>
12     <component class="org.bflow.toolbox.hive.addons.components.
           DiagramExportComponent" params="CSV"/>
13     <component class="org.bflow.toolbox.hive.addons.components.
           ToolRunComponent" params="QoS Tool"/>
14     <component class="org.bflow.toolbox.hive.addons.components.
           ShellAnalysisComponent" params=""/>
15     <component class="org.bflow.toolbox.hive.addons.components.
           ProblemsViewGeneratorComponent" params=""/>       </components>
16 </protocol>
```

Das Attribut „Name“ im Abschnitt <protocol> muss eindeutig sein.

Im Bereich <display> können die Namen des Add-ons für verschiedene Sprachen festgelegt werden. Der Bereich <default> kennzeichnet dabei den Namen des Add-ons, wenn eine Sprachangabe nicht verfügbar ist. Ebenso verhält es sich mit dem Abschnitt <description>.

Von entscheidender Bedeutung ist der Abschnitt <components>. Hier werden die auszuführenden Komponenten des Add-ons sowie deren Parameter festgelegt. Die Attribute „class“ und „params“ sind immer Pflichtangaben. Der angegebene Klassenname muss immer den vollständigen Namen der Klasse der auszuführenden Komponente beinhalten. Die Bedeutung der Parameter ist analog der Konfiguration von Add-ons (vgl. Abschnitt 2.3.4) Die Reihenfolge der Komponenten in der Add-on-Definition entspricht der Reihenfolge während der Ausführung eines Add-ons.

Die ToolRun-Komponente benötigt einen weiteren Parameter „tool“. Dieser sollte den Namen des Tools angeben, so wie es im Einstelldialog im „Bflow“ → „Tools“ im Modellierungswerkzeug definiert ist. Andernfalls lässt sich das Add-on nicht starten.

Ist die Add-on-Definitionsdatei korrekt erzeugt, muss sie über den Extension-Point org.bflow.toolbox.addons.protocol registriert werden. Ebenso müssen natürlich die einzubindenden externen Programme und Exporte (vgl. dazu Abschnitt 1.3) vorhanden sein. Beachten Sie beim Einbinden externer Programme die Lizenzbestimmungen dieser Programme!

2.10.3 Hinzufügen eigener Prolog-Programme zu den bflow*-Quellen

Nachdem das einzubindende Prolog-Programm geschrieben ist, muss es in ein Plug-In gepackt werden. Anschließend muss das Prolog-Programm über den Extension-Point org.bflow.toolbox.addons.prologaddition registriert werden:

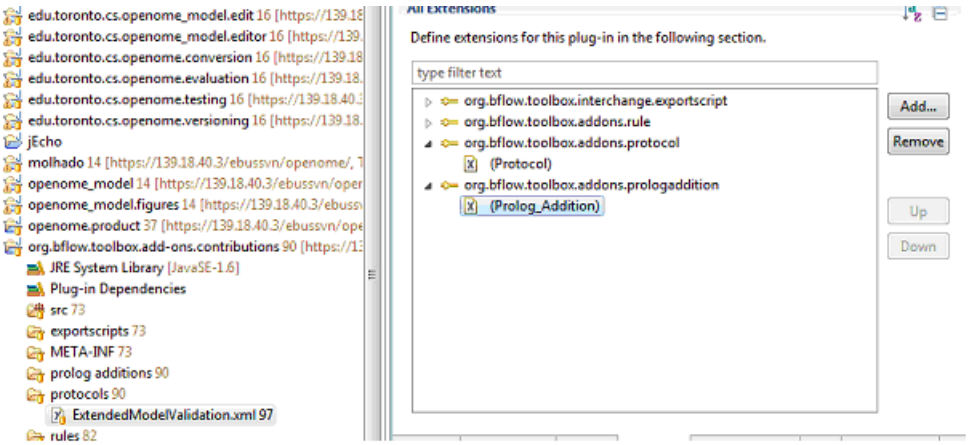

Abbildung 2.14: Registrierung eines Prolog-Programms

Hinweis: Die Angabe im Feld ID* entspricht dem Programmnamen, der nötig ist, um das Prolog-Programm innerhalb eines Add-ons (mit dem Parameter -pl) zu starten. Diese muss also ggf. Anwendern oder Entwicklern mitgeteilt werden. Ohne Kenntnis dieser ID kann das Programm nicht zugeordnet und somit nicht gestartet werden.

3 Arbeit mit Validierungsregeln

Im folgenden Kapitel wird die Arbeit mit Validierungsregeln beschrieben. bflow* bietet die Möglichkeit, EPK- und oEPK-Modelle zu validieren. Hierbei wird geprüft, ob das Modell entsprechend den Regeln der Modellierungssprache erstellt wurde, ob also z.B. eine Funktion nie mehr als eine eingehende Kante hat.

3.1 Auswahl von Validierungsregeln

Im Normalfall werden bei jeder Validierung alle verfügbaren Regeln geprüft. Es besteht jedoch die Möglichkeit, die Verwendung einzelner Regeln festzulegen. Wie dies geschieht, ist im bflow*-Handbuch im Abschnitt 5.4 beschrieben.

3.2 Anpassen von Fehlermeldungen

Sollte es notwendig sein auf eigene Fehlermeldungen zurückzugreifen, besteht die Möglichkeit, diese zu bearbeiten. Im Abschnitt 5.5 des bflow*-Handbuches ist dies ausführlich dokumentiert.

3.3 Im- und Export von Regelkonfigurationen

Wenn eine eigene Regelzusammenstellung mit anderen Modellieren ausgetauscht werden soll, ist es möglich die Im- und Exportfunktion zu nutzen. Dazu muss „Fenster" → „Benutzervorgaben" → „Bflow" → „Validation" aufgerufen werden.

Mit einem Klick auf "Export" wird der Dateiauswahldialog geöffnet. Hier muss der Pfad und der Name der Datei angegeben werden, in der die Regelkonfiguration gespeichert werden soll.

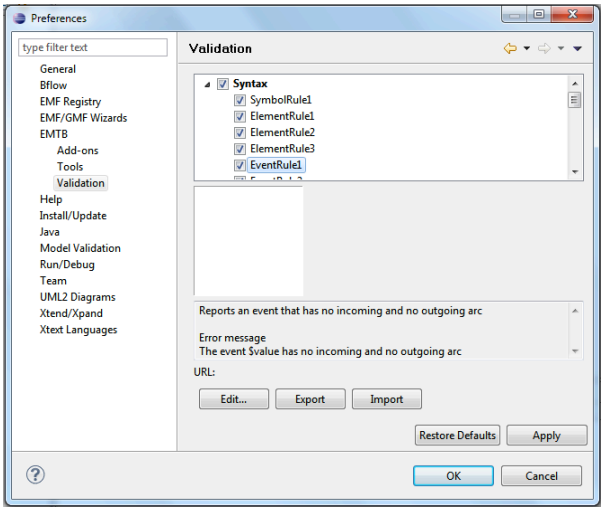

Abbildung 3.1: Dateiauswahldialog für Regelsetexport

Mit einem Klick auf “Speichern” wird der Vorgang abgeschlossen.

Um eine Regelkonfiguration zu importieren, wird nach dem Klick auf “Import” der Vorgang analog zum Exportvorgang vorgenommen.

Da zu jedem Zeitpunkt nur eine Regelkonfiguration genutzt werden kann, muss im folgenden Dialog entscheiden werden, was mit der alten Regelkonfiguration passieren soll.

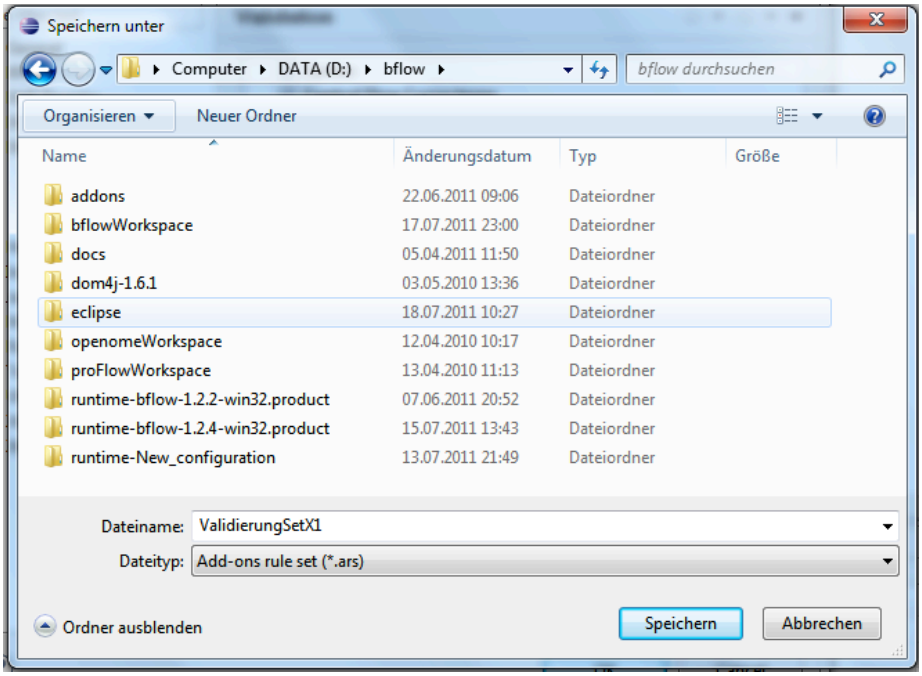

Abbildung 3.2: Verhalten gegenüber der bisherigen Regelkonfiguration

Hier bieten sich zwei Möglichkeiten. Beim Schließen des Dialogs mit “OK” werden alle alten Einstellungen überschrieben. Beim Klick auf “Cancel” jedoch werden die alte und die neue Konfiguration zusammengeführt. Es werden keine Regeln deaktiviert und es gelten sowohl die Regeln der alten als auch die der neuen Regelkonfiguration.

3.4 Erstellung eigener Regeln (für Entwickler)

Sollte es notwendig sein, Regeln zu nutzen welche nicht standardmäßig angeboten werden, so gibt es die Möglichkeit, selbst Regeln zu definieren. Um dies zu realisieren, muss der Quellcode von bflow* angepasst werden. Wie dies zu bewerkstelligen ist, wird im folgenden näher beschrieben:

3.4.1 Regel-Konfigurationsdatei

Die Regeln sind in Dateien im Ordner org.bflow.toolbox.contributions.addons/rules/ definiert. Es existiert jeweils eine Datei für EPK- und oEPK-Modelle in englischer und deutscher Sprache. Ist es nötig eine neue Regel für die Validierung hinzuzufügen, so muss die entsprechende Datei ergänzt werden.

Die Regel-Konfigurationsdatei hat die folgende Struktur:

```
<?xml version="1.0" encoding="UTF-8"?>
<language value="de">
     <rule>
          <id>SyntaxEventRule1</id>
          <name>EventRule1</name>
          <message>Ereignis $value hat mehr als einen eingehenden
              Kontrollfluss. Eingehende Kontrollflüsse ueber
              Konnektoren synchronisieren.
          </message>
          <class>Syntaxprüfung</class>
          <description>Prüft, ob das Ereignis mehr als einen
              eingehenden Kontrollfluss hat.
          </description>
          <image></image>
          <default>true</default>
     </rule>
     <rule>
          <id>SyntaxEventRule2</id>
          <name>EventRule2</name>
          <message>Ereignis $value hat mehr als einen ausgehenden
              Kontrollfluss. Ausgehende Kontrollflüsse über Konnektoren
               synchronisieren.
          </message>
          <class>Syntaxprüfung</class>
          <description>Prüft, ob das Ereignis mehr als einen
              ausgehenden Kontrollfluss hat.
          </description>
          <image></image>
          <default>true</default>
     </rule>
</language>
```

Dabei ist die Bedeutung der Einträge wie folgt:

- id: Wird zur Identifizierung der Regel benutzt und muss innerhalb eines Diagrammtyps (EPC oder oEPC) eindeutig sein.
- name: Name der Regel, der in dem unter „Fenster" → „Benutzervorgaben" → „Bflow" → „Validation" aufrufbaren Dialogfenster zur Auswahl der anzuwendenden Regeln erscheint. Dieser kann beliebig gewählt werden, sollte aber nachvollziehbar sein.
- message: Dieses Attribut legt fest, wie die Fehlermeldung aussieht, wenn sie erscheinen sollte. Fügt man in den Meldungstext den Wert „$value" ein, so wird dieser durch einen entsprechenden Wert ersetzt.
- class: Gibt die Art bzw. die Klasse der Regel an. Diese kann beliebig gewählt werden und dient nur zur besseren Zuordnung von Regeln im Dialog zur Auswahl der anzuwendenden Regeln
- description: Ein Beschreibungstext, der die Regel kurz näher beschreiben sollte.

- url: (optional) URL einer Website, die nähe Informationen zum Fehler liefert. Diese kann aus dem Dialog zur Auswahl der anzuwendenden Regeln heraus geöffnet werden.
- image: An dieser Stelle kann ein Pfad zu einem Bild eingefügt werden, welches die Regel symbolisiert. Dieses Bild wird im Dialog zur Auswahl der anzuwendenden Regeln gezeigt.
  Derzeit liegen alle Bilder im Verzeichnis
  org.bflow.toolbox.contributions.addons/images.
- default: Gibt an, ob die Regel standardmäßig ein- oder ausgeschaltet ist.
- type: Gibt den Typ der Regel an. (check, epsilon oder prolog)

Nachdem die Regelkonfiguration geschrieben und in ein Plug-in-Projekt gepackt wurde, ist es möglich, diese mit Hilfe des folgenden Extension-Points zu registrieren:

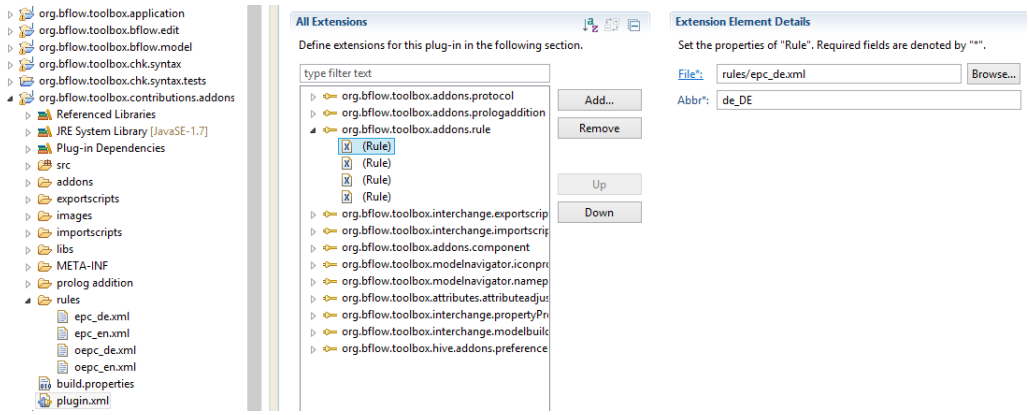

Abbildung 3.3: Registrierung einer Regelbeschreibung

Wichtig ist hierbei, ein korrektes Sprachkürzel anzugeben. Verwendet werden die standardisierten internationalen Codes nach ISO-3166.

3.4.2 Prüflogik mittels "Check"

"Check" ist ein Teil des Xtend-Plugins, welches mit dem EMF-Plugin ab Eclipse-Version 3.5 ausgeliefert wird.

Eine Dokumentation findet sich auf

`http://help.eclipse.org/indigo/topic/org.eclipse.xpand.doc/help/core_reference.html`. Hilfreich zum Verstehen der Logik beim Schreiben von Prüfregeln mit Check ist auch ein Studim der bestehenden Regeln, die unter unter org.bflow.toolbox.chk.syntax/*.chk zu finden sind.

Eine Prüfregel für EPK-Modelle kann folgendermaßen aussehen:

```
1 context epc::Event if(shallCheck("SyntaxEventRule1"))
2     ERROR getErrorMessage("SyntaxEventRule1", this.name) :
3     !(this.outgoingControlFlows().isEmpty &&
4 this.incomingControlFlows().isEmpty);
```

Zunächst erfolgt der Aufruf der Methode „shallCheck()“ um zu prüfen, ob diese Regel überhaupt angewendet werden soll. Die ID der Regel entspricht der in der Regel-Konfigurationsdatei angegebenen (vgl. Abschnitt 3.4.1). Unter „Fenster” → „Benutzervorgaben” → „Bflow” → „Validation” kann der Benutzer konfigurieren, ob die Regel geprüft werden soll.

Soll die Regel angewendet werden und tritt in dem zu prüfenden Modell der entsprechend betrachtete Fehler auf, so wird die Methode „getErrorMessage()“ aufgerufen. Diese liefert die zu der Regel gehörende Fehlermeldung für den Anwender.

Der erste Parameter ist wieder die ID der Regel. Der zweite Parameter ist der Wert, der den „$value“ Teil im „message“ Attribut ersetzen soll (falls im Eintrag „message” der Regel-Konfigurationsdatei ein solcher Teil vorkommt). Im obigen Beispiel wird die Beschriftung des Ereignisses eingesetzt.

Nach dem „:“ folgt die eigentliche Prüflogik.

3.4.3 Prüflogik mittels “Epsilon”

“Epsilon” ist Bestandteil des Eclipse GMT-Projekts (`http://eclipse.org/gmt/`). Das Werkzeug Epsilon besteht aus mehreren Sprachen für verschiedene Anwendungen. Für den Zweck der Erweiterung der Validierungsregeln wird im speziellen die Sprache „evl“4 bzw. „eol“5 verwendet.

Auch hier wird für generelle Sprachspezifikationen das Studium des “Epsilon”-Handbuches empfohlen. Dieses ist als PDF-Datei unter `http://www.eclipse.org/epsilon/doc/book/` zu finden.

Die in bflow* enthaltenen mit Epsilon realisierten Prüfregeln sind unter org.bflow.toolbox.oepc.evl zu finden. Die Namensgebung hier ist nicht besonders clever: Das Paket “epc” beinhaltet die Regeln für die EPK-Diagramme (“epcSyntax.evl”) und der Ordner “validation” die Regeln für die oEPK-Diagramme (“syntax.evl”).

Als Beispiel für eine implementierte Prüflogik sei folgendes Codebeispiel gegeben:

```
1 context BflowSymbol {
2
3     critique HasName {
4
5     guard: (self.isTypeOf(Event) or self.isTypeOf(BusinessObject))
6     check: self.hasName() or not prefRequester.shallCheck('
          EpsilonSymbolRule1')
7     message: prefRequester.getErrorMessage('EpsilonSymbolRule1', self.
          eClass().name)
```

```
8       fix {
9           title: 'Standardnamen vergeben'
10          do {
11              self.name := 'Element';
12          }
13      }
14      fix
15      {
16          title: 'Prüfregel ausschalten'
17          do {
18              prefRequester.setRuleEnabled('EpsilonSymbolRule1', false);
19          }
20      }
21  }
```

Diese Regel prüft alle Elemente vom Typ "BflowSymbol" darauf, ob sie einen Namen haben.

Der Bereich "check" beinhaltet dabei die Prüflogik und die Abfrage, ob diese Regel angewendet werden soll. Dies übernimmt die Methode "prefRequester.shallCheck()". Das Anwenden der Regeln lässt sich vom Anwender unter „Fenster" → „Benutzervorgaben" → „Bflow" → „Validation" festlegen.

Als Parameter für den Aufruf wird die ID der Regel verwendet, so wie sie in der Regel-Konfigurationsdatei (vgl. Abschnitt 3.4.1) zu finden ist.

Soll die Regel angewendet werden und tritt der entsprechend betrachtete Fehler auf, so wird die Methode „prefRequester.getErrorMessage()" aufgerufen. Diese liefert die zu der Regel gehörende Fehlermeldung.

Der erste Parameter ist wieder die ID der Regel. Der zweite ist der Wert, der den „$value" Teil im „message"-Attribut ersetzen soll.

Ein weiterer interessanter Punkt sind die „fix"-Bereiche. Diese dienen dazu, dem Anwender „Quick-Fix"-Optionen anzubieten, welche aufgetretenen Fehler schnell löst.

Die zweite Fix-Option schaltet die Prüfregel generell ab. Dies übernimmt der Methodenaufruf „prefRequester.setRuleEnabled()". Der erste Parameter ist die ID der Regel und der zweite ob die Regel ein- oder ausgeschaltet werden soll. Im Normalfall sollte hier „false" stehen, da dieser Fix die Regel abschalten soll.

3.4.4 Prüflogik mittels Prolog

Während die Check- und Epsilon-Regeln während des Modellierens im Hintergrund geprüft werden können, kann der Aufruf von mit Prolog realisierten Validierungsregeln nur über den Add-on-Mechanismus von bflow* erfolgen (siehe Abschnitt **??**.) Werden Prolog-Programme den bflow*-Quellen hinzugefügt, so lassen sich diese per Add-on über die vergebenen IDs aufrufen. So hat zum Beispiel das von bflow* verwendete ExtendedModelValidation-Prolog-Programm die ID „bflow.epc2009". Der Aufruf erfolgt somit über „-pl:bflow.epc2009" (ohne weitere Pfadangabe).

Wie im bflow*-Benutzerhandbuch beschrieben ist, lässt sich (mit „Fenster" → „Benutzervorgaben" → „Bflow" → „Validation") einstellen, welche Regeln vom Prolog-Programm überprüft werden sollen.

Auf diese Auswahl kann ein selbst erstelltes Prolog-Programm durch Nutzung der folgenden Parameter zugreifen:

Parameter	Bemerkung
$ALL	Alle vorhandenen Regeln werden geprüft. Die Einstellungen im Einstelldialog werden ignoriert.
$DEFAULT	Alle Regeln, die per Default-Parameter aktiviert sind, werden geprüft. Alle anderen Einstellungen im Einstelldialog werden ignoriert.
$SETUP	Dieser Parameter sollte als Standard gewählt werden. Hier werden nur die Regeln geprüft, die im Einstelldialog aktiviert sind.

Tabelle 3.1: Parameter für die Arbeit mit Regelsets

Gemäß dieser Parameter könnte ein Prolog-Aufruf dann wie folgt aussehen:

-pl:bflow.epc2009 -p:$SETUP

Dabei wird dann das sich hinter der ID "bflow.epc2000" befindliche Prolog-Programm der folgenden Anfrage aufgerufen:

addon_query(de,[$SETUP])

Dabei wird "$SETUP" durch die vom Benutzer tatsächlich ausgewählten Regeln ersetzt.

Das für die erweiterte Modellprüfung verwendete Prolog-Programm org.bflow.toolbox.contributions.addons/prolog\ addition/epc2009.pl zeigt, wie Regelprüfungen umgesetzt werden können.

Aufgerufen wird dort addon_query(Sprache,Testliste), was wiederum die Anfrage `a` aufruft. Diese wiederum ruft alle Tests auf, die per `analyze0` und `analyze` angesprochen werden. Ein leicht zu verstehender Test, an dem man die Arbeitsweise eines Prolog-Tests leicht nachvollziehen kann, ist property6(X):

```
1 property6(X) :- tocheck(syntax6),
2         function(X),more_than_one_outgoing_arcs(X),
3         compose_message(de,Message,['Die Funktion hat mehr als einen
              ausgehenden Kontrollflusspfeil!']),
4         compose_message(en,Message,['The function has more than one
              outgoing control-flow arc!']),
5         message('ERROR',X,Message,'syntax6').
```

Wenn Sie selber Prolog-Tests erstellen wollen, sollte jedoch der erste Schritt sein, ein EPK-Modell mittels Rechtsklich auf die Datei in der Ansicht „Projektexplorer" → „Exportieren" → „Multi-Target Export" ins Format „Prolog Facts" zu exportieren und die Schreibweise der zum Modell gehörenden Prolog-Fakten zu studieren.

3.4.5 Abfrage der vom Benutzer gewählten Validierungsregeln

Sollte es einmal notwendig sein, in selbst erstelltem Java-Code auf die Einstellungen, die der Anwender unter „Fenster" → „Benutzervorgaben" → „Bflow" → „Validation" vorgenommen hat, zuzugreifen, zugreifen, kann die statische Instanz der Klasse org.bflow.toolbox.hive.addons/vali genutzt werden. Folgende Methoden stehen dabei zur Verfügung.

Methode	Erklärung
isEnabled(ID)	Prüft anhand der ID, ob die Regel aktiviert ist.
getErrorMessage(ID,VALUE)	Erfragt die gespeicherte Fehlermeldung anhand der ID. Der übergebene Parameter VALUE ersetzt dabei das Schlüsselwort "$value" in der Meldung
setEnabled(ID,VALUE)	Setzt den Status der Regel, die anhand der ID identifiziert wird. VALUE kann entweder wahr oder falsch sein.

Ein typischer Aufruf in Java könnte wie folgt aussehen:

```
1 public String getConstraintMessage() {
2     if(ValidationService.getInstance().isEnabled("crossing_con") {
3         return ValidationService.getInstance().getErrorMessage ("
              crossing_con", "element x1");
4     } else {
5         return null;
6     }
7 }
```

4 Hinzufügen eigener Farbschemas (für Entwickler)

Eigene Farbschemen können unter org.bflow.toolbox.diagram.extensions/src/org/bflow/toolbox/extensions/colorschemes definiert werden. Klassen zur Definition eigener Farbschemas implementieren das Interface IGlobalColorSchema.

Die Klassen der einzelnen Modellelemente, auf die in den Farbschemas Bezug genommen wird, finden sich in

org.bflow.toolbox.epc.model\src\org\bflow\toolbox\epc (for EPK-Diagramme models) und

org.bflow.toolbox.oepc\src\oepc\ (für oEPK-Diagramme).

Für Vorgangskettendiagramme werden zur Zeit Farbschemas noch nicht berücksichtigt.

Anhang: Quelltexte für Beispiel-Add-on

```
1  public interchange 'ASA4OD Example Export' (a4d) <> epc {
2
3  description: 'Just a simple example for the velocity export interface.
       It can be used in combination with the ASA4OD Add-on.'
4
5  @script: '/files/asa4od.vt', insertAttributes=true
6
7  }
```

Listing 4.1: Exportbeschreibung asa4od.exd

```
1  #foreach( $shape in $shapes)
2  #foreach( $key in $shape.Attributes.keySet())
3  #if( $key == "marked" && $shape.Attributes.get($key) == "true")
4  #set( $type = $shape.Type.split("\.")[4].toLowerCase() )
5  $type:$shape.Id:$shape.Name
6  #end
7  #end
8  #end
9  edges:$edges.size()
```

Listing 4.2: Exportskript asa4od.vt

```
1  package de.fh_zwickau.bumb_l_b.asa4od;
2
3  import java.io.BufferedReader;
4  import java.io.FileReader;
5  import java.io.IOException;
6
7  /**
8   * This class is an example application for a bflow* toolbox add-on. It
        shows
9   * how an add-on can look like and demonstrates to print messages on the
        problems
10  * view and adjust element's attributes.
11  *
12  * @author Mathias Grunert, Florian Schoenfelder
13  *
14  */
15 public class ASA4OD {
16
17     /**
18      * Executes the analysis of given arguments.
19      *
```

```
20          * @param args
21          * - arguments for execution
22          */
23         public void execute(final String[] args) {
24
25             if (args.length < 1) {
26                 printError("There is no path to a file given.");
27                 System.exit(1);
28             }
29             String path = args[0];
30
31             int eventCounter, functionCounter = eventCounter = 0;
32
33             try (BufferedReader br = new BufferedReader(new FileReader(path)
                   )) {
34                 String line;
35                 while ((line = br.readLine()) != null) {
36                     // [0]:type, [1]:id,[2]:name
37                     String[] pieces = line.split(":", 3);
38                     switch (pieces[0]) {
39                     case "event":
40                         printInfo("Event \"" + pieces[2] + "\" was marked. "
                               ,
41                                 pieces[1]);
42                         printAttributeString("SET", pieces[1], "$fontstyle",
                               "BOLD");
43                         eventCounter++;
44                         break;
45                     case "function":
46                         printInfo("Function \"" + pieces[2] + "\" was marked
                               . ",
47                                 pieces[1]);
48                         printAttributeString("SET", pieces[1], "$fontsize",
                               "30");
49                         functionCounter++;
50                         break;
51                     case "edges":
52                         printInfo("The model contains " + pieces[1] + "
                               edges.");
53                         break;
54                     }
55                 }
56                 if (eventCounter == 0 && functionCounter == 0) {
57                     printWarning("0 events or functions were marked.");
58                 }
59             } catch (IOException e) {
60                 printError("An error occured while execution.");
61             }
62         }
63
64         /**
65          * Prints a message of type info.
66          *
67          * @param message
```

```
68          */
69         private void printInfo(String message) {
70             printInfo(message, "");
71         }
72 
73         /**
74          * Prints a message of type warning.
75          *
76          * @param message
77          */
78         private void printWarning(String message) {
79             printMessage("WARNING", message, "");
80         }
81 
82         /**
83          * Print a message of type info connected to the given element.
84          *
85          * @param message
86          * @param elementId
87          */
88         private void printInfo(String message, String elementId) {
89             printMessage("INFO", message, elementId);
90         }
91 
92         /**
93          * Prints a message of type error.
94          *
95          * @param message
96          */
97         private void printError(String message) {
98             printMessage("ERROR", message, "");
99         }
100 
101         /**
102          * Prints a, for the bflow* problems-view formated, string with
                given type,
103          * message and element.
104          *
105          * @param type
106          * - type of message
107          * @param message
108          * - message for the problems view
109          * @param elementId
110          * - the with the message connected element
111          */
112         private void printMessage(String type, String message, String
                elementId) {
113             printFormatString("MESSAGE", type, elementId, message, "");
114         }
115 
116         /**
117          * Prints a string for attribute modification.
118          *
119          * @param type
```

```
120         * @param elementId
121         * @param key
122         * @param value
123         */
124        private void printAttributeString(String type, String elementId,
125                String key, String value) {
126            printFormatString("ATTRIBUTE", type, elementId, key, value);
127        }
128
129        /**
130         * Prints a for bflow* analysable string.
131         *
132         * @param formatType
133         * @param type
134         * @param elementId
135         * @param key
136         * @param value
137         */
138        private void printFormatString(String formatType, String type,
139                String elementId, String key, String value) {
140            System.out.println(String.format("addon:[%s][%s][%s][%s][%s]#FS#
                "
141                    formatType, type, elementId, key, value));
142
143        }
144
145        /**
146         * Starts the application.
147         *
148         * @param args
149         */
150        public static void main(String[] args) {
151            new ASA4OD().execute(args);
152        }
153    }
```

Listing 4.3: Quelltext ASA4OD.java