ISA Dialog Manager

DEBUGGER

A.06.03.c

In diesem Handbuch ist der Debugger des ISA Dialog Managers beschrieben. Der Debugger ist ein Werkzeug für das Auffinden von Fehlern in Dialogen.

ISA Informationssysteme GmbH

Meisenweg 33

70771 Leinfelden-Echterdingen

Deutschland

Darstellungskonventionen

DM wird in diesem Handbuch synonym zu "Dialog Manager" verwendet.

Die Bezeichnung UNIX schließt generell alle unterstützten UNIX-Derivate ein - außer in den explizit angegebenen Fällen.

Dort wo für geläufige englische Fachbegriffe keine gängigen deutschen Übersetzungen existieren, wird zur Vermeidung von Unklarheiten der englische Begriff verwendet.

< >	muss durch einen entsprechenden Wert ersetzt werden
color	Schlüsselwort ("keyword")
.bgc	Attribut
{ }	optional (0 oder einmal)
[ ]	optional (0 oder n-mal)
<A> | <B>	entweder <A> oder <B>

Beschreibungsmodus

Alle Schlüsselwörter sind fett und unterstrichen, z.B.

variable **integer** **function**

Indizierung von Attributen

Syntax für indizierte Attribute:

[I]

[I,J] bzw. [row,column]

Identifikatoren

Identifikatoren müssen mit einem Großbuchstaben oder einem "Unterstrich" ('_') beginnen. Die weiteren Zeichen können Groß-, Kleinbuchstaben, Zahlen oder Unterstriche sein.

Der Bindestrich ('-') ist für die Benennung von Identifikatoren als Zeichen ***nicht*** zugelassen!

Die maximale Länge eines Identifikators beträgt *31* Zeichen.

Beschreibung der zugelassenen Identifikatoren in Backus-Naur-Form

<Identifikator> ::= <erstes Zeichen>{<Zeichen>}

<erstes Zeichen> ::= _ | <Großbuchstabe>

<Zeichen> ::= _ | <Kleinbuchstabe> | <Großbuchstabe> | <Ziffer>

<Ziffer> ::= 1 | 2 | 3 | … 9 | 0

<Kleinbuchstabe> ::= a | b | c | … x | y | z

<Großbuchstabe> ::= A | B | C | … X | Y | Z

Inhalt

1 Einleitung

Der DM-Debugger erlaubt die kontrollierte Ausführung von DM-Dialogskripten. Ein Dialogskript, das unter der Kontrolle des DM-Debuggers gestartet wird, hält bei Erreichen der ersten ausführbaren Anweisung an und erwartet Befehle des Benutzers. Dieser kann nun über eine grafische oder zeilenorientierte Benutzerschnittstelle eine Anzahl von verschiedenen Operationen vornehmen. In diesem Handbuch wird die zeilenorientierte Schnittstelle als "STDIO-Schnittstelle" und die grafische als "Fensterschnittstelle" bezeichnet.

1.1 Starten des Debuggers

Anders als üblich wird der DM-Debugger über das Programm gestartet, das er debuggen soll. Dazu wird dem ausführbaren, mit der DM-Bibliothek gebundenen Programm oder dem Simulations-Programm zusätzlich zu den anderen Argumenten die Option **-IDMDBG** mit einer Spezifikation übergeben wird, die die gewünschte Art der Schnittstelle identifiziert. Das Programm bleibt dann vor der Ausführung der ersten Regel-Anweisung stehen, gibt einen Prompt aus und wartet auf Benutzereingaben.

Bei der Art der Schnittstelle kann sowohl auf eine grafische Oberfläche als auf eine befehlsorientierte Schnittstelle zurückgegriffen werden.

1.2 Verfügbarkeit der Schnittstellen

Die Fenstersystemschnittstelle ist auf allen vom Dialog Manager unterstützten Systemen verfügbar. Die STDIO-Schnittstelle ist nur auf den Unix oder Unix-ähnlichen Systemen verfügbar.

Unix-Besonderheit

Wurde das Programm mit der -IDMDBG-Option gestartet, kann man auch in den Debugger gelangen, indem man während des Programmlaufs ein INT-Signal an den Prozess schickt (dies kann üblicherweise mit der Tastenkombination Control-C geschehen). Vor der Bearbeitung der nächsten Anweisung wird dann der Debugger aktiviert.

1.3 Weitere Hilfen zur Fehleranalyse

Neben dem Debugger stehen weitere Hilfen zur Fehleranalyse von Dialog Manager Anwendungen zur Verfügung:

- Ablaufverfolgung (Tracing).
- Herausschreiben von DM Zustandsinformationen (Dumpstate). Dieses Hilfsmittel ist für die DM Versionen A.05.01.g3 und A.05.01.h sowie ab A.05.02.e verfügbar.

Diese Hilfen zur Fehleranalyse sind im Handbuch „Entwicklungsumgebung“ beschrieben.

2 Konzept des Debuggers

Bevor der Befehlsvorrat des DM-Debuggers erläutert wird, führen wir kurz die grundlegenden Konzepte ein:

Der Regel-Stack

Der Regel-Stack enthält Informationen über die gerade in Abarbeitung befindlichen DM-Regeln und ihre Zeilennummern. Wenn etwa die Regel R1 durch ein Ereignis aktiviert wurde, diese in Zeile 7 die Regel R2 aufgerufen hat und diese in Zeile 9 die Regel R3 und der nächste abzuarbeitende Befehl sich schließlich in Zeile 5 in der Regel R3 befindet, befinden sich genau die Regeln R1/Zeile 7, R2/Zeile 9 und R3/Zeile 5 auf dem Regel-Stack. R1/Zeile 7 ist dabei das "obere" Ende des Stacks und R3/Zeile 5 das "untere" Ende.

Die betrachtete Abarbeitungsstelle

Wenn die Abarbeitung des Programms unterbrochen ist und der Debugger aktiviert wird, dann kann auf eine Stelle im Programm besonders leicht zugegriffen werden. Dies ist die "betrachtete Regel" in der "betrachteten Zeile" auf der "betrachteten Stackebene". Dies ist häufig die als nächstes auszuführende Zeile des Programms. Stets bildet aber die Stelle des momentanen Regel-Stacks die "betrachtete Regel und Zeile".

Haltepunkte

Wenn bei der Abarbeitung des Programms gewisse Programmstellen erreicht werden, wird die Abarbeitung unterbrochen und dem Benutzer wird Gelegenheit gegeben, Befehle einzugeben. Diese Programmstellen heißen "Haltepunkte". Es können "feste" und "flüchtige" Haltepunkte unterschieden werden.

Ein "fester" Haltepunkt wird durch folgendes festgelegt:

1. eine Regel,
2. eine Zeile,
3. ein Kennzeichen, ob der Haltepunkt aktiviert ist (enabled/disabled) und
4. möglicherweise eine Bedingung.

Vor der Ausführung der angegebenen Zeile in der angegebenen Regel wird die Abarbeitung unterbrochen, wenn der Haltepunkt aktiviert ist und entweder keine Bedingung ausgegeben ist oder die angegebene Bedingung erfüllt ist.

"Flüchtige" Haltepunkte sind solche, bei denen die Regel oder die Zeilennummer, in der die Abarbeitung des Programms unterbrochen wird, nicht explizit angegeben wird. Solche Haltepunkte werden benutzt, wenn das Programm bei der nächsten Anweisung überhaupt, bei der nächsten Anweisung auf derselben Stackebene oder bei der nächsten Anweisung auf einer höheren

Stackebene anhalten soll. Bei Erreichen irgendeines Haltepunkts, sei dies ein flüchtiger oder ein fester, löst sich ein vorhandener flüchtiger Haltepunkt auf.

Überwachungsausdrücke

An eine Regel können Überwachungsausdrücke angebunden werden, die angezeigt werden wenn die Regel betrachtet wird.

Regelnamen

Bei verschiedenen Befehlen werden Regelnamen als Argumente angegeben. Neben ihrem eigentlichen Namen lässt sich jede Regel auch mit einem Kurznamen ansprechen, der mit einem kleinen "r" beginnt, auf welches eine Nummer folgt. Der zu einer Regel gehörige Kurzname wird mit ausgegeben, wenn die Regeln mit dem Kommando "rules" aufgelistet werden.

Neustart eines Programms

Will man ein bereits laufendes Programm noch einmal von vorn beginnen, muss man den aktuellen Lauf abbrechen (hierzu dient der Befehl "quit") und das Programm von Anfang an neu starten. Mit dem Abbruch des Programms gehen jedoch gesetzte Haltepunkte und Überwachungsausdrücke verloren. Um deren erneutes Setzen zu erleichtern, wurde ein Mechanismus eingeführt, der es erlaubt, die Liste der gesetzten Haltepunkte und der überwachten Ausdrücke in einer Datei zu sichern und den in einer solchen Datei gesicherten Zustand wiederherzustellen.

Ausdrücke

Bei manchen Befehlen können als Argumente DM-Ausdrücke angegeben werden. Diese werden mit Hilfe des normalen Regelparsers gelesen und übersetzt. Wenn man hier Syntaxfehler macht, kann es zu Fehlermeldungen des Dialog Managers einschließlich Zeilennummer des Fehlers kommen. Diese Zeilennummer bezieht sich auf maschinengenerierten Code.

3 Die grafische Schnittstelle

3.1 Konzept

Das grundlegende Konzept der Fensterschnittstelle besteht darin, dass diese als eigenständiger Prozess abläuft, der mit dem zu debuggenden Dialog(-prozess) kommuniziert. Die Kommunikation erfolgt dabei auf Basis des vom jeweiligen Betriebssystem unterstützten Kommunikationsprotokolls (TCPIP oder DDE).

Der zu debuggende Dialogprozess heißt "Client".

Um einen Client debuggen zu können, ist es erforderlich, dass die zugehörige Programmdatei mit der Dialog Manager-Entwicklungsbibliothek zusammen gelinkt ist.

Die Fensterschnittstelle dient in diesem Zusammenhang als "Server" und besteht aus zwei Dateien:

1. der ausführbaren Datei "idmdbg" (Microsoft Windows) und
2. der dazu gehörenden Binär-Dialogdatei "idmdbg.dlg".

Diese Dateien werden automatisch bei der Installation des Dialog Managers mitinstalliert.

3.2 Manueller Start

Der Server kann zum einen von Hand gestartet werden, zum anderen gibt es die Möglichkeit des automatischen Starts.

Für einen manuellen Start genügt es, die ausführbare Datei des Debuggers zu starten. Protokoll und Port bzw. Servicename werden in diesem Fall automatisch ermittelt.

```
idmdbg
```

Zum manuellen Start des Debuggers gibt es noch folgende Optionen:

-port <portNumber>

Dieser Parameter ist die Portnummer, die unter UNIX zum Aufbau einer **TCP/IP**-Verbindung verwendet werden soll. Diese Option ist nur auf Unix oder Unix-ähnlichem System verfügbar. Lässt man *<portNumber>* weg oder gibt hier 0 an, so wird automatisch ein freier Port ermittelt.

-service <serviceName>

Dieser Parameter ist der eindeutige Name des **DDE-Service**, der unter MICROSOFT WINDOWS für die Kommunikation verwendet werden soll. Hier kann zum Beispiel "IDMDBG" angegeben werden. Diese Option ist nur auf Microsoft Windows verfügbar.

Nach dem Start wartet der Server auf den Verbindungsaufbau zu einem Client.

3.3 Automatischer Start

Der Server kann automatisch von einem Client gestartet werden.

Dies ist die normale Art, einen Server zu starten. Dazu müssen dem Client über die Kommandozeile entsprechend die Optionen mitgegeben werden, dass der Debugger gestartet werden soll. Dieses funktioniert nur, wenn das Debugger-Programm im Pfad enthalten ist.

Der Client wird mit seiner Standard-Kommandozeile und zusätzlichen Optionen gestartet. Diese Optionen definieren die Kommunikationsmethode, die für die Verbindung mit der Debugger-Fensterschnittstelle verwendet wird, sowie zusätzliche Debugger-Argumente.

-IDMDBGautostart <true|false>

Über diesen optionalen Parameter wird gesteuert, ob der Debugger vom Client aus gestartet werden soll oder ob der Client eine Verbindung zu einem bereits gestarteten Debugger aufbauen soll. Wird für diesen Parameter der Wert *true* angegeben, wird der Server automatisch gestartet. Diese ist auch der interne Default, falls diese Option nicht angegeben wird. Wird hier *false* angegeben, muss der Server von Hand gestartet werden.

Für einen automatischen Start ohne weitere Angaben von Parametern muss die Option -IDMdbg mit angegeben werden.

-IDMdbg

Der Parameter definiert, dass die Kommunikation automatisch via DDE (Microsoft Windows) bzw. TCP/IP (Unix) erfolgen soll. <portNumber> bzw. <serviveName> werden automatisch vom IDM gewählt.

Beim automatischen Start können auch die Argumente (<dialogFile> und <portNumber> oder <serviceName>) durch die Argumente der Optionen "-IDMDBG" und "-IDMDBGscript" bestimmt werden.

-IDMDBG <communicationMethod>

Dieser Parameter beschreibt die Kommunikationsmethode, die für die Client-Server-Verbindung verwendet wird. Es gibt die folgenden Alternativen:

Kommunikationsart	Bedeutung
STDIO	Die Kommunikation erfolgt über STDIO-Funktionen. Diese Option ist nur auf Unix-basierten Systemen verwendbar.

Kommunikationsart	Bedeutung
TCPIP{<hostName>}<:<portNumber>>	Die Kommunikation erfolgt über TCP/IP. <portNumber> ist eine Portnummer, zu der sich der Client zu verbinden versucht. Das Argument <hostName> ist nur zulässig, wenn für "-IDMDBGautostart" *false* gesetzt ist. In diesem Fall, versucht sich der Client mit dem Port auf dem Host <hostName> zu verbinden. Andernfalls wird die angegebene Portnummer <portNumber> auf der lokalen Maschine verwendet. Die Angabe von <portNumber> ist optional, wird diese nicht angegeben, so wird vom IDM automatisch ein freier Port gewählt. Diese Kommunikationsmethode existiert unter Unix-Derivaten.
DDE<:<serviceName>>	Die Kommunikation erfolgt über DDE. <serviceName> ist der Name des Services zu dem sich der Client zu verbinden versucht. Die Angabe von <serviceName> ist optional, ist wird er nicht angegeben, so wird vom IDM automatisch ein Servicename gewählt. Diese Kommunikationsart ist auf den PC-basierten Systemen verfügbar.

3.4 Beispiele für das Starten des Debuggers

» Starten des Debuggers unabhängig vom Betriebssystem durch die Simulation:

```
idm -IDMdbg list.dlg
```

» Starten des Debuggers auf Microsoft Windows durch die Simulation:

```
idm -IDMDBG DDE:IDMDBG list.dlg
```

(Servicename *IDMDBG* ist vom Anwender vorgegeben)
oder

```
idm -IDMDBG DDE list.dlg
```

(Servicename wird automatisch gewählt)

» Starten des Debuggers auf einer HP Workstation mit Motif als Oberfläche:

```
idm -IDMDBG TCPIP:4711 list.dlg
```

(Port *4711* ist vom Anwender vorgegeben)
oder

```
idm -IDMDBG TCPIP list.dlg
```

(zu verwendender Port wird automatisch ermittelt)

» Separater Start des Debuggers auf Microsoft Windows:

```
idmdbg -service IDMDBG
```

Danach Start der Simulation über

```
idm -IDMDBGautostart false -IDMDBG DDE:IDMDBG list.dlg
```

3.5 Bedienung

3.5.1 Sprachumschaltung

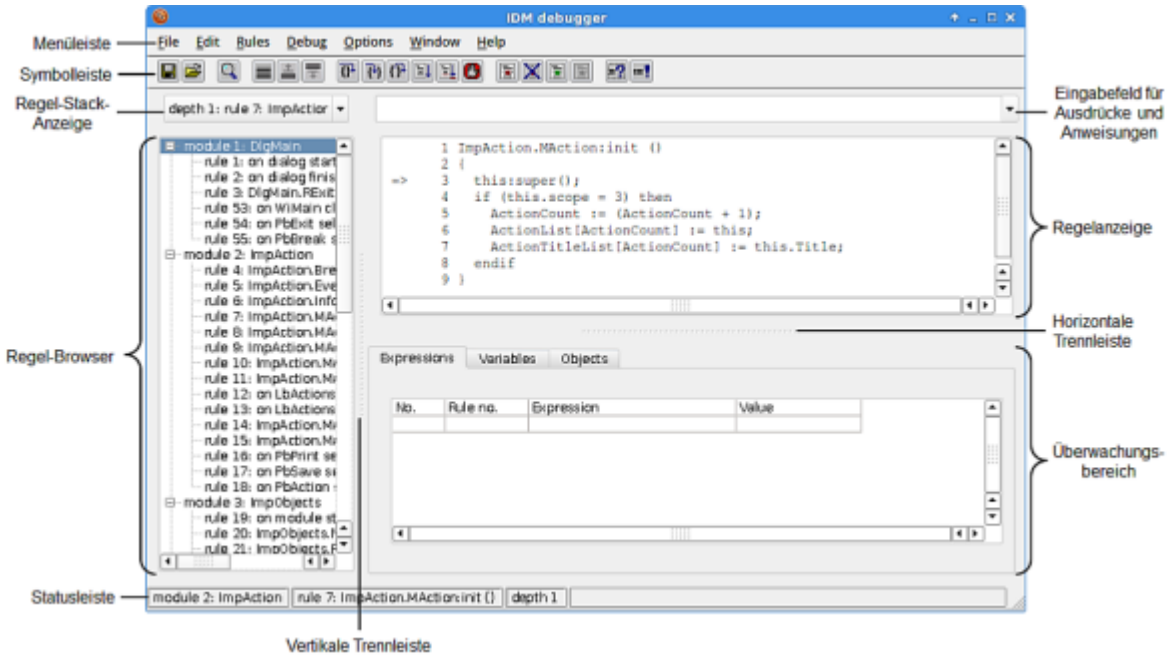
Die Defaultversion ist Englisch. Die deutsche Version kann man beim manuellen Start mit der Option `-IDMlanguage=2` einstellen.

3.5.2 Layout

Die Benutzeroberfläche besteht aus einem Hauptfenster und einem Ausgabefenster.

Das Hauptfenster (siehe folgende Abbildung) besteht aus:

- Menüleiste
- Symbolleiste
- Regel-Stack-Anzeige (en. „Stack Display“)
- Eingabefeld für Ausdrücke und Anweisungen (en. „Expression/Statement“)
- Regel-Browser (en. „Rule Browser“)
- Regelanzeige (en. „Rule Display“)
- Überwachungsbereich für Ausdrücke (en. „Expressions“), Variablen (en. „Variables“) und Objekte (en. „Objects“)
- Statusleiste



Mit den beiden Trennleisten kann die horizontale und vertikale Aufteilung des Fensters verändert werden. Durch Verschieben der vertikalen Trennleiste mit der Maus kann die Breite von Regel-Stack-

Anzeige und Regel-Browser links sowie Eingabefeld für Ausdrücke und Anweisungen, Regelanzeige und Überwachungsbereich rechts angepasst werden. Das Verschieben der horizontalen Trennleiste ändert die Höhe der Regelanzeige oben und des Überwachungsbereichs unten.

3.5.2.1 Menüleiste

Die Menüleiste setzt sich aus den Menüs *„File“* (de. *„Datei“*), *„Edit“* (de. *„Bearbeiten“*), *„Rules“*(de. *„Regeln“*), *„Debug“* (de. *„Debuggen“*), *„Options“* (de. *„Optionen“*), *„Window“* (de. *„Fenster“*) und *„Help“* (de. *„Hilfe“*) zusammen.

File (Datei)

Das Menü *„File“* hat die Einträge:

- *„Zustand speichern…“*: Speichern des aktuellen Zustands.
- *„Zustand wiederherstellen…“*: Ausführen von Regeln.
- *„Sitzung beenden“*: Schließt die Verbindung zu dem zu debuggenden Dialog (Client); falls der Debugger mit der Option `-stayalive=true` gestartet wurde, wartet der Debugger anschließend darauf, dass sich ein neuer Client mit ihm verbindet; bei `-stayalive=false` wird der Debugger beendet.
- *„Beenden“*: Beenden des Debuggers.

Edit (Bearbeiten)

Das Menü "Edit" hat die Einträge:

- *„Suchen…“*: Zeigt eine Dialogbox für die Suche nach Regeln an; der Eintrag kann über den Accelerator `F3` angesprochen werden.
- *„Breakpoints…“*: Zeigt eine Liste aller definierten Haltepunkte an.

Rules (Regeln)

Das Menü *„Rules“* enthält Einträge für die momentan abgearbeitete Regel und maximal die 5 zuletzt angezeigten Regeln.

Debug (Debuggen)

Das Menü *„Debug“* enthält die Steuerbefehle des Debuggers, die in vier Befehlsgruppen eingeteilt sind.

- Befehle zur Bearbeitung von Haltepunkten:
  - *„Set Break“* (de. *„Breakpoint setzen“*); Accelerator `F9`
  - *„Remove Break“* (de. *„Breakpoint entfernen“*)
  - *„Enable Break “*(de. *„Breakpoint einschalten“*)
  - *„Disable Break“* (de. *„Breakpoint ausschalten“*)
- Befehle zur kontrollierten Ausführung des Debuggers:

- *„Next“* (de. *„Nächste Anweisung“*); Accelerator `F10`
- *„Step“* (de. *„Regel betreten“*); Accelerator `F11`
- *„Finish“* (de. *„Regel beenden“*); Accelerator `Shift` + `F11`
- *„Continue“* (de. *„Weiter“*); Accelerator `F5`
- *„Runto“* (de. *„Weiter bis Cursor“*); Accelerator `Strg` + `F10`
- *„Stop“* (de. *„Stopp“*)

- Befehle für Operationen auf dem Regel-Stack:
  - *„Stack“* (de. *„Regel-Stack ausgeben“*)
  - *„Up“* (de. *„Höher im Regel-Stack“*)
  - *„Down“* (de. *„Tiefer im Regel-Stack“*)
- Befehle zum Ausdrucken und Ändern von Werten:
  - *„Print“* (de. *„Wert ausgeben“*)
  - *„Eval“* (de. *„Wert zuweisen“*)

Options (Optionen)

Das Menü *„Options“* enthält folgende Einträge

- *„Toolbar“* (de. *„Symbolleiste“*), mit dem die Symbolleiste ein- und ausgeschaltet werden kann.
- *„Autostop“*: Ein- und Ausschalten des Anhaltens des Debuggers in der ersten, ausführbaren Regel eines Moduls.
- *„Errorstop“*: Ein- und Ausschalten des Anhaltens des Debuggers bei einem Fehler in der Regelausführung.

Window (Fenster)

Das Menü *„Window“* enthält zwei Einträge:

- *„Output“* (de. *„Ausgabefenster“*): Blendet das Ausgabefenster ein und aus (Accelerator `Alt` + `1`).
- *„Profiler“*: Startet und beendet den Profiler (Accelerator `Alt` + `2`).

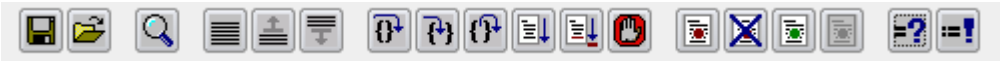
Help (Hilfe)

Das Menü *„Help“* besteht aus den folgenden Einträgen:

- *„Index“*: Startseite der Dokumentation.
- *„Product Information“* (de. *„Produktinformation“*); u.a. Versionsnummer.

3.5.2.2 Symbolleiste

Die Symbolleiste enthält Schaltflächen für häufig benötigte Befehle. Sie wird unterhalb der Menüleiste angezeigt und kann mit dem Menübefehl *„Options“* → *„Toolbar“* ein- und ausgeblendet werden.



Die Schaltflächen entsprechen folgenden Menübefehlen:

	„File“ → „Save State…“	*„Datei“ → „Zustand speichern…“*
	„File“ → „Restore State…“	*„Datei“ → „Zustand wiederherstellen…“*
	„Edit“ → „Find…“	*„Bearbeiten“ → „Suchen…“*
	„Debug“ → „Stack“	*„Debuggen“ → „Regel-Stack ausgeben“*
	„Debug“ → „Up“	*„Debuggen“ → „Höher im Regel-Stack“*
	„Debug“ → „Down“	*„Debuggen“ → „Tiefer im Regel-Stack“*
	„Debug“ → „Next“	*„Debuggen“ → „Nächste Anweisung“*
	„Debug“ → „Step“	*„Debuggen“ → „Regel betreten“*
	„Debug“ → „Finish“	*„Debuggen“ → „Regel beenden“*
	„Debug“ → „Continue“	*„Debuggen“ → „Weiter“*
	„Debug“ → „Runto“	*„Debuggen“ → „Weiter bis Cursor“*
	„Debug“ → „Stop“	*„Debuggen“ → „Stopp“*
	„Debug“ → „Set Break“	*„Debuggen“ → „Breakpoint setzen“*
	„Debug“ → „Remove Break“	*„Debuggen“ → „Breakpoint entfernen“*
	„Debug“ → „Enable Break“	*„Debuggen“ → „Breakpoint einschalten“*
	„Debug“ → „Disable Break“	*„Debuggen“ → „Breakpoint ausschalten“*
	„Debug“ → „Print“	*„Debuggen“ → „Wert ausgeben“*
	„Debug“ → „Eval“	*„Debuggen“ → „Wert zuweisen“*

Für jede Schaltfläche wird ein Tooltip eingeblendet, wenn der Mauszeiger über sie bewegt wird.

Insensitive Schaltflächen werden ausgegraut dargestellt.

3.5.2.3 Regel-Stack-Anzeige

Diese Dropdown-Liste enthält die Regeln, die gerade auf dem Regel-Stack liegen. Durch die Selektion eines Eintrags kann im Regel-Stack im Gegensatz zu den Schaltflächen der Symbolleiste bzw.

den Menüeinträgen für *„Up“* und *„Down“* auch mehr als eine Stufe abwärts oder aufwärts gesprungen werden. Die selektierte Regel wird in der Regelanzeige ausgegeben.

3.5.2.4 Eingabefeld für Ausdrücke und Anweisungen

Die Befehle *„Print“* (de. *„Wert ausgeben“*) und *„Eval“* (de. *„Wert zuweisen“*) werten den Inhalt dieses Eingabefelds aus. Diese Befehle sind auch über das Kontextmenü (rechte Maustaste) erreichbar.

Da die Eingabe als Combobox ausgeführt ist, kann auch eine der maximal 5 zuletzt eingegebenen Ausdrücke oder Anweisungen durch Selektion in das Feld übernommen werden.

3.5.2.5 Regel-Browser

Der Regel-Browser dient zur Navigation durch die Regeln.

Ein Doppelklick auf eine Regel gibt diese Regel in der Regelanzeige aus, ein Doppelklick auf ein Modul blendet die zugehörigen Regeln im Browser ein oder aus.

Der Regel-Browser hat ein Kontextmenü mit folgenden Einträgen:

- *„Find…“* (de. *„Suchen…“*): Suchen nach Regeln.
- *„Open all“* (de. *„Alles anzeigen“*): Zeigt alle Regeln aller Module an.
- *„Close all“* (de. *„Oberste Ebene anzeigen“*): Zeigt nur die Module an.

3.5.2.6 Regelanzeige

In der Regelanzeige wird der Inhalt einer Regel ausgegeben. Dabei markiert „=>“ in Regeln, die auf dem Regel-Stack liegen, die momentan betrachtete Zeile. Zeilen, in denen ein Haltepunkt gesetzt ist, sind mit „+“ (eingeschalteter Haltepunkt) bzw. „-“ (ausgeschalteter Haltepunkt) gekennzeichnet.

Die Regelanzeige hat ein Kontextmenü mit folgenden Einträgen:

- *„Print“* (de. *„Wert ausgeben“*): Gibt den Wert eines im Regeltext selektierten Ausdrucks im Ausgabefenster aus.
- *„Eval“* (de. *„Wert zuweisen“*): Führt eine im Regeltext selektierte Anweisung aus.
- *„New breakpoint“* (de. *„Neuer Breakpoint“*): Setzt einen neuen Haltepunkt in der Zeile, in der der Cursor steht.
- *„Remove breakpoint“* (de. *„Breakpoint entfernen“*): Löscht den Haltepunkt in der Zeile, in der der Cursor steht.
- *„Enable breakpoint“* (de. *„Breakpoint einschalten“*): Schaltet den Haltepunkt in der Zeile, in der der Cursor steht ein.
- *„Disable breakpoint“* (de. *„Breakpoint ausschalten“*): Schaltet den Haltepunkt in der Zeile, in der der Cursor steht aus.

» *„Edit breakpoint…“* (de. *„Breakpoint bearbeiten…“*): Zeigt die Dialogbox mit der Liste aller Haltepunkte an.

» *„Runto“* (de. *„Weiter bis Cursor“*): Ausführen bis zu der Zeile, in der der Cursor in der Regelanzeige steht.

3.5.2.7 Überwachungsbereich

Im Überwachungsbereich gibt es Registerkarten für die Liste der Überwachungsausdrücke, die Liste der lokalen Variablen und die Objektansicht.

3.5.2.7.1 Liste der Überwachungsausdrücke

Die Liste der Überwachungsausdrücke befindet sich auf der Registerkarte *„Expressions“* des Überwachungsbereichs. Hier können Ausdrücke eingetragen werden, deren Werte immer dann angezeigt werden, wenn die Regel ausgeführt wird, in der die Ausdrücke definiert wurden. Geänderte Werte werden durch eine andere Hintergrundfarbe hervorgehoben.

Die Tabelle hat ein Kontextmenü mit den folgenden Einträgen:

» *„New expression“* (de. *„Neuer Ausdruck“*): Fügt eine neue Zeile vor dem markierten Ausdruck ein.

» *„Remove expression(s)“* (de. *„Ausdrücke entfernen“*): Löscht alle markierten Ausdrücke.

Der Wert eines Überwachungsausdrucks kann geändert werden, indem man bei dem Ausdruck in der Spalte *„Value“* doppelklickt und einen neuen Wert eingibt.

3.5.2.7.2 Variablenliste

Die Variablenliste befindet sich auf der Registerkarte *„Variables“* des Überwachungsbereichs. In ihr werden die lokalen Variablen (inklusive statischer Variablen) und Parameter (Argumente) der aktuell ausgeführten Regel sowie alle globalen Variablen angezeigt.

Der Wert einer Variablen oder eines Parameters kann geändert werden, indem man bei der Variablen bzw. dem Parameter in der Spalte *„Value“* doppelklickt und einen neuen Wert eingibt.

3.5.2.7.3 Objektansicht (Object View)

In der Objektansicht (Object View) des Debuggers können Objekte mit ihren vordefinierten und benutzerdefinierten Attributen angezeigt und die Attributwerte während des Debuggens überwacht und manuell geändert werden. Die Objektansicht befindet sich im unteren Bereich des Debuggers auf einer eigenen Registerkarte *„Objects“*, neben den Registern der überwachten Ausdrücke (*„Expressions“*) und Variablen (*„Variables“*).

Die Objektansicht unterteilt sich in die Objektliste auf der linken Seite und die Objektanzeige auf der rechten Seite. Die Objektliste besteht aus einem Suchfeld oben und der Ergebnisliste darunter. Die Objektanzeige enthält für jedes überwachte Objekt eine Registerkarte mit jeweils einer Attributtabelle

und einem Eingabefeld für einen Filterausdruck über der Tabelle. In der Objektanzeige können bis zu 3 Objekte gleichzeitig überwacht werden.

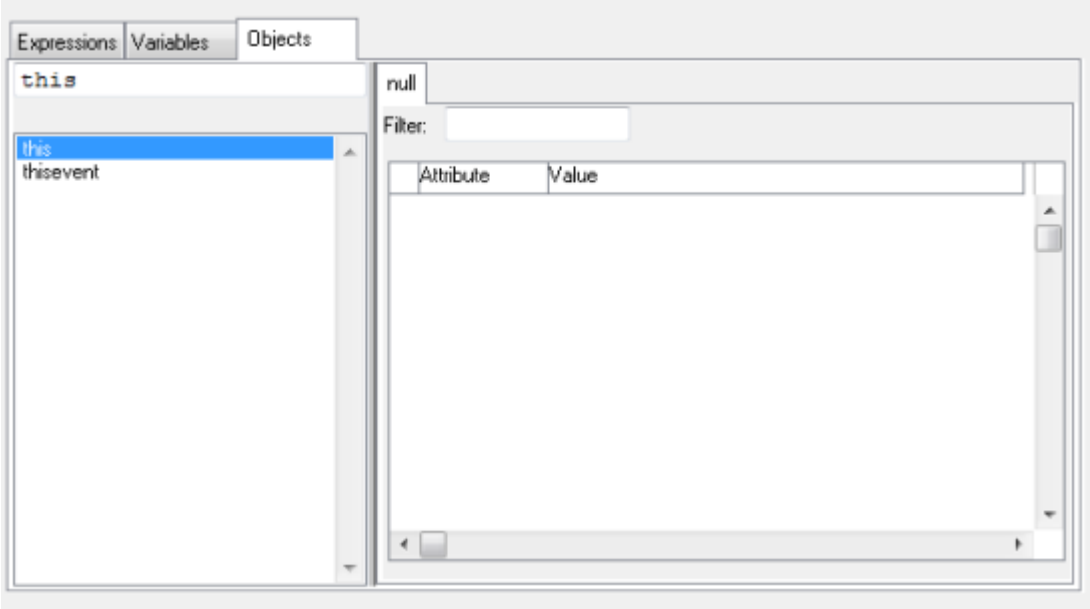


Abbildung 1: *Objektansicht im Initialzustand*

Beim Start des Debuggers ist das Suchfeld der Objektliste mit „this“ vorbelegt. In der Ergebnisliste werden das ***this***-Objekt, das ***thisevent***-Objekt und eventuell weitere Objekte mit „this“ im Namen angezeigt. Die Objektanzeige ist zunächst leer, da noch kein Objekt zur Überwachung ausgewählt wurde.

3.5.2.7.3.1 Objekte suchen

Im Suchfeld der Objektliste können Objektnamen (Label), Objektpfade oder Objektklassen (***window***, ***edittext***, ***record***…) vollständig oder teilweise eingegeben werden. Objekte, die dem Suchausdruck entsprechen, werden dann in der Ergebnisliste unter dem Suchfeld angezeigt, wobei der beste Treffer markiert wird.

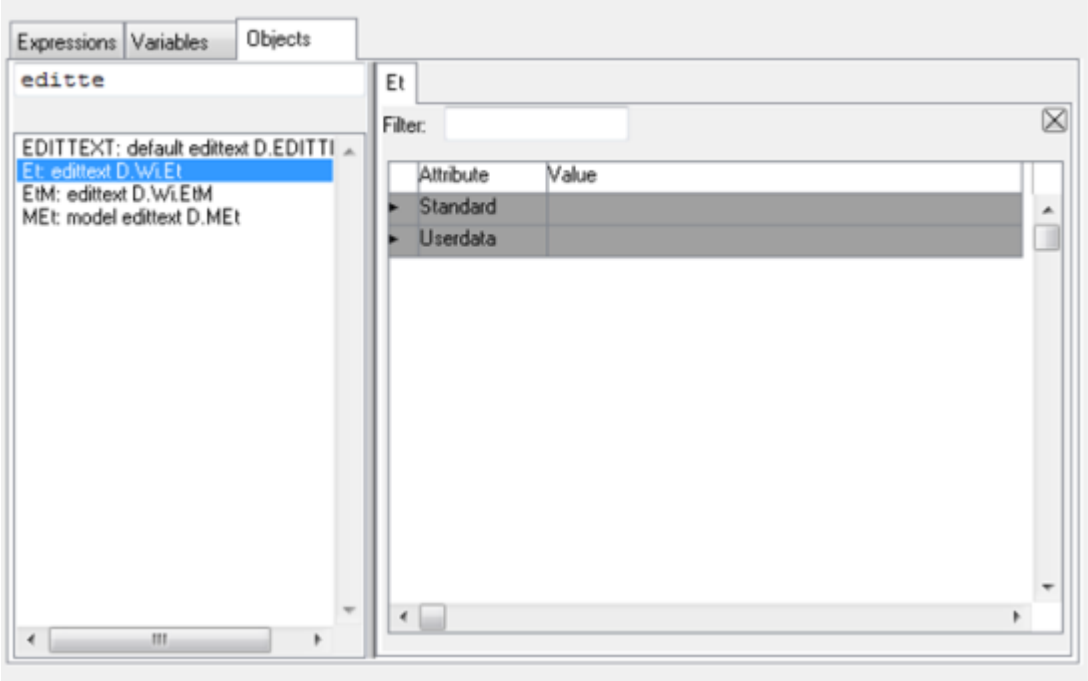


Abbildung 2: *Suche mit passenden Ergebnisobjekten*

Die Suche umfasst folgende Objektarten:

- Visuelle Objekte (Fenster, Edittext, Tablefield usw.),
- Application, Record,
- Document, Doccursor,
- Ressourcen (Accelerator, Color, Font usw.),
- This-Objekt,
- Thisevent-Objekt,
- Setup-Objekt,
- Clipboard-Objekt.

3.5.2.7.3.2 Objekte und ihre Attribute überwachen

Ein Objekt aus der Ergebnisliste wird mit einem Doppelklick in die Objektanzeige übernommen. Nach der Übernahme in die Objektanzeige sieht man dort zunächst die zugeklappten Attributkategorien *„Standard“* mit den vordefinierten Attributen und *„Userdata“* mit den benutzerdefinierten Attributen (siehe „Abbildung 2“, rechter Bereich). Klappt man diese Kategorien auf, wird die Liste der jeweiligen Attribute angezeigt.

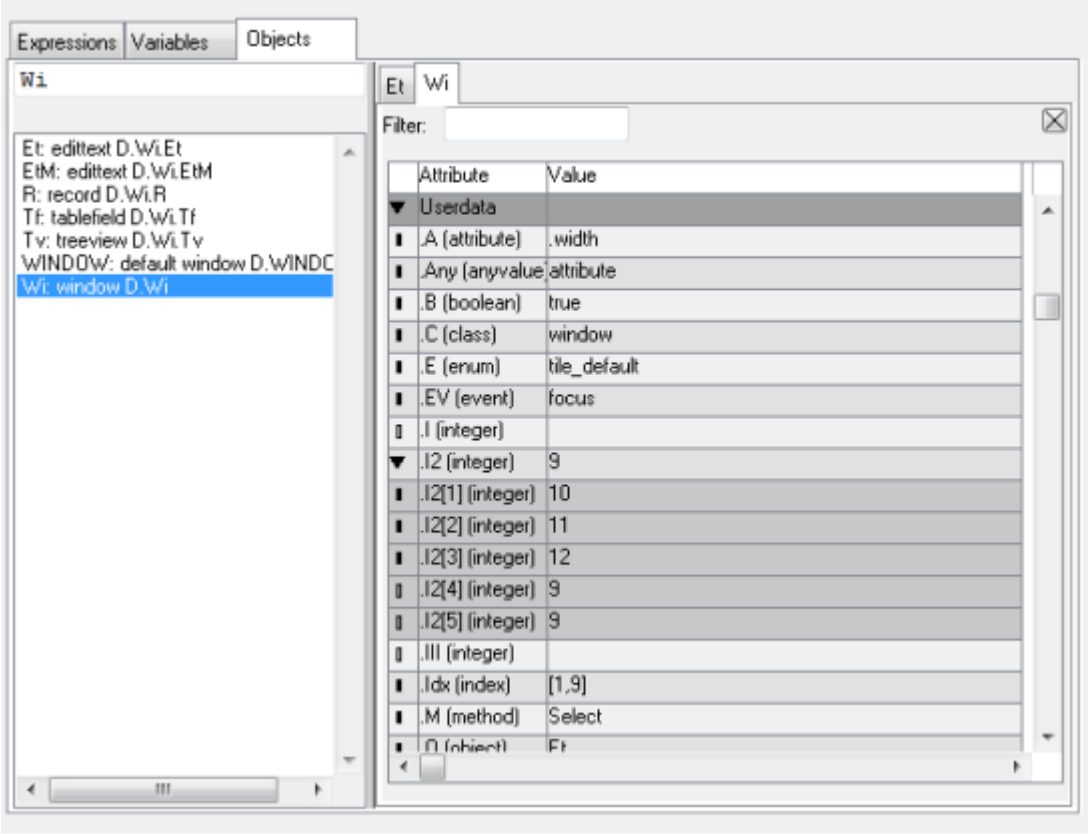


Abbildung 3: *Attribute des überwachten Objektes Wi*

Zu jedem Attribut werden in der zweiten Listenspalte Name und Datentyp und in der dritten Spalte der Wert angezeigt. In der ersten Spalte zeigt ein Viereck (Qt, Windows) bzw. Stern (Motif), dass der Wert des Attributs am überwachten Objekt gesetzt wurde. Bei geerbten Werten ist das Viereck nicht ausgefüllt (Qt, Windows) bzw. die erste Spalte leer (Motif).

Indizierte Attribute erkennt man an Pfeilsymbolen in der ersten Spalte. Sie zeigen nach rechts, wenn die Einzelwerte des Attributs ausgeblendet sind, und nach unten, wenn die Einzelwerte eingeblendet sind. Die Liste der Einzelwerte kann per Mausklick auf- und zugeklappt werden. Einzelwerte indizierter Attribute sind durch ein dunkleres Grau als Hintergrund und die Indexangabe beim Namen gekennzeichnet.

Wenn ein überwachtes Objekt im Verlauf des Debuggens zerstört wird, dann wird nach einer Mitteilung an den Anwender die Registerkarte des Objekts aus der Objektanzeige entfernt und die Objektliste aktualisiert.

Mit der Schaltfläche *„X“* kann ein Objekt aus der Objektanzeige entfernt werden. Wenn bereits 3 Objekte überwacht werden, muss zunächst ein Objekt aus der Objektanzeige entfernt werden, bevor man ein weiteres Objekt zur Überwachung auswählen kann.

Wird das ***this***-Objekt zur Überwachung ausgewählt, dann wird das zum Zeitpunkt der Auswahl enthaltene Objekt in die Objektanzeige übernommen. Das überwachte Objekt hat anschließend keinen Bezug zum ***this***-Objekt mehr.

Hinweise

- Es sollten keine Objekte mit besonders vielen Attributen oder großen indizierten Attributen überwacht werden. Bei ihnen kann es unter Umständen zu großen Zeitverzögerungen bei der Aktualisierung und Darstellung kommen.
- Ändert sich die Größe eines indizierten Attributs aus dem Dialog heraus, wird das gesamte Attribut beim nächsten Halt bzw. Schritt rot markiert.
- Der Vererbungsstatus eines Attributs lässt sich nicht immer verlässlich nachverfolgen. Deshalb kann es vorkommen, dass nicht korrekt angezeigt wird, ob ein Attributwert geerbt oder am Objekt gesetzt ist.

3.5.2.7.3.3 Attribute filtern

Wenn man nicht alle Attribute eines überwachten Objekts angezeigt bekommen möchte, kann man im Eingabefeld *„Filter“* einen Attributnamen oder einen Teil davon als Filterausdruck eingeben. Dann werden in der Tabelle nur die vordefinierten und benutzerdefinierten Attribute angezeigt, deren Name den Filterausdruck enthält. Um wieder alle Attribute anzuzeigen, löscht man den Filterausdruck und bestätigt mit `Return`.

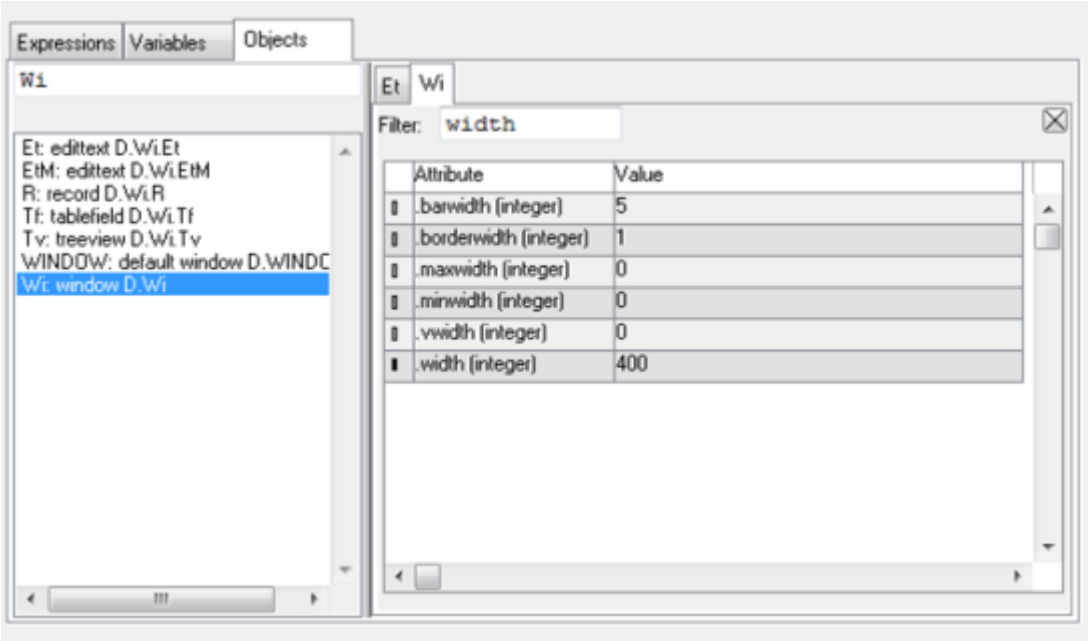


Abbildung 4: *Gefilterte Attributansicht*

3.5.2.7.3.4 Attributänderungen

Attribute, die sich im Verlauf des Debuggens ändern, werden mit ihren neuen Werten rot dargestellt (siehe „Abbildung 5“).

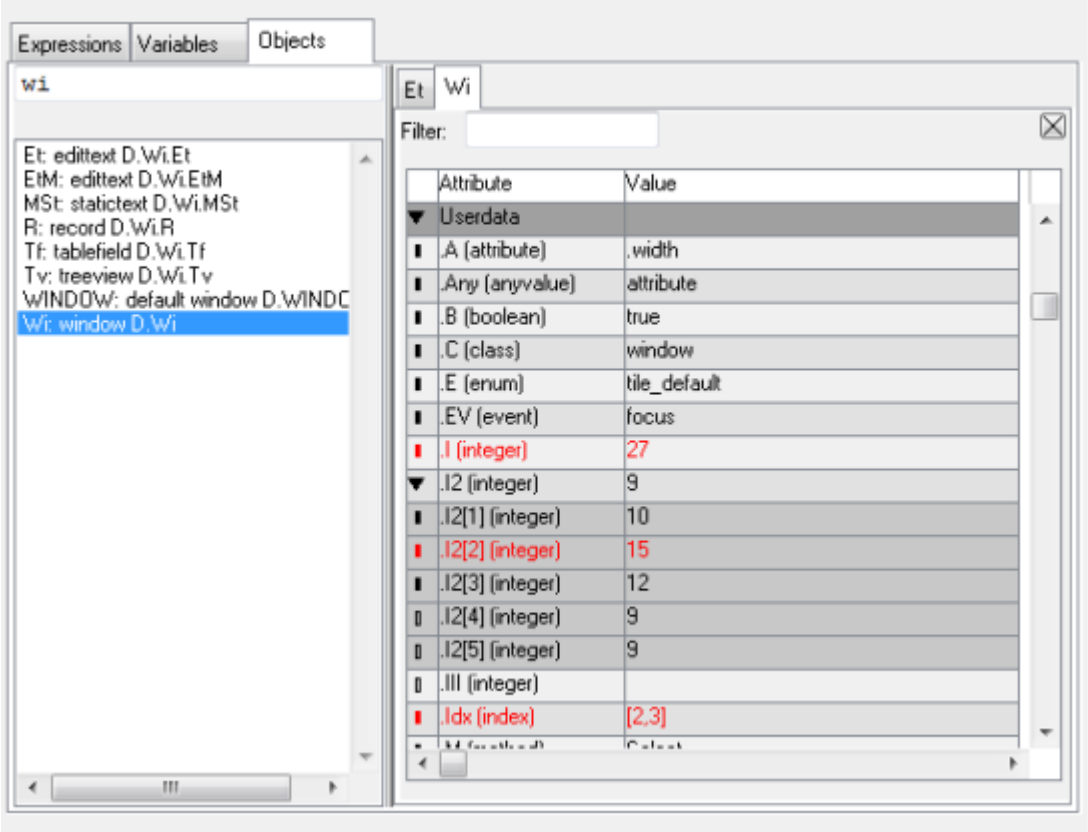


Abbildung 5: *Geänderte Attribute*

Attributwerte können manuell geändert werden, wenn die Programmausführung an einem Breakpoint angehalten wurde. Ein Attributwert lässt sich nach einem Doppelklick in das Feld mit dem Wert bearbeiten.

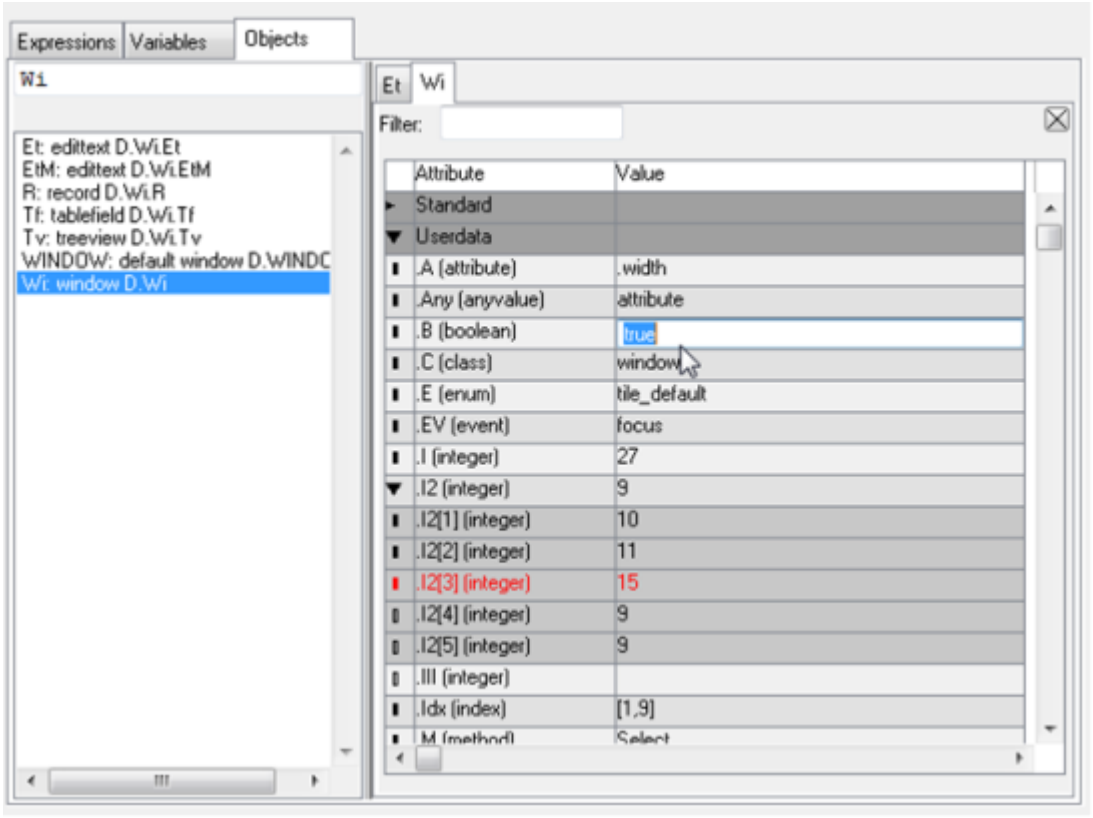


Abbildung 6: *Manuelles Ändern von Attributwerten*

Beim Ändern von Attributen mit dem Datentyp *string* muss der Wert in Anführungszeichen eingeschlossen werden (z. B. "New Value").

Die Größe von indizierten Attributen und assoziativen Arrays kann manuell nicht verändert werden, das heißt es können keine Werte hinzugefügt und entfernt werden.

Hinweise

- Wird für ein Attribut dessen Datentyp sowohl Objekte als auch Strings akzeptiert (*anyvalue* oder *string*) ein Wert ohne Anführungszeichen eingegeben, der nicht als Objektbezeichner identifiziert werden kann, dann wird der Wert in einen String gewandelt.
- Wird bei einem String-Wert nur das öffnende oder schließende Anführungszeichen eingegeben, dann wird das fehlende Anführungszeichen automatisch ergänzt.
- Objekte aus noch nicht geladenen Modulen können weder von der Objektsuche gefunden noch einem Attribut als Wert zugewiesen werden.

3.5.2.8 Statuszeile

Die Statuszeile zeigt für Menüeinträge einen einzeiligen Infotext an. Falls kein Menüeintrag ausgewählt ist, dann enthält sie den Namen der Regel, deren Text momentan in der Regelanzeige ausgegeben wird, den Namen des Moduls, zu der die Regel gehört. Außerdem wird für Regeln, die sich momentan auf dem Regel-Stack befinden, deren Stackposition angegeben.

3.5.3 Ausgabefenster

Im Ausgabefenster werden Werte von Ausdrücken und der Regel-Stack angezeigt.

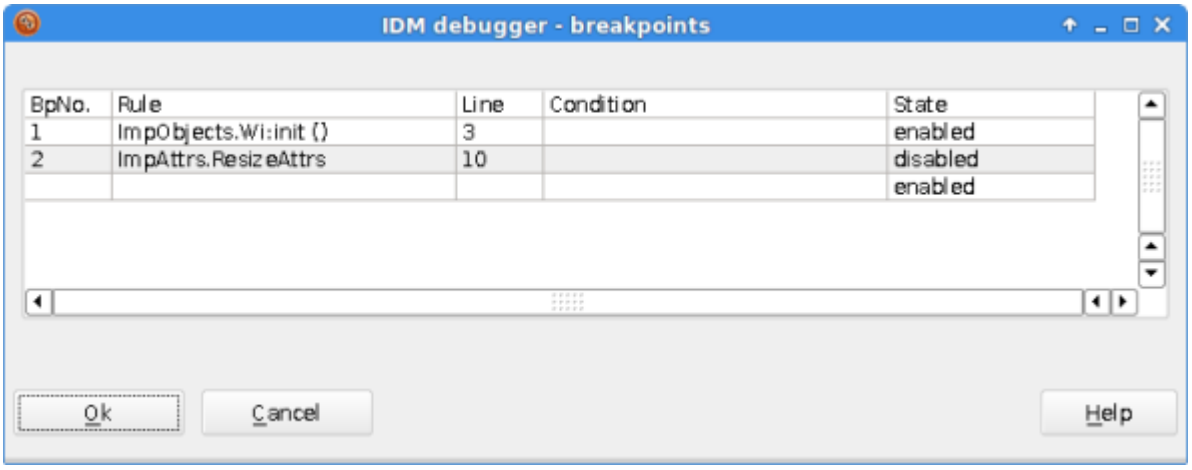
Abbildung 7: *Ausgabefenster*

3.5.4 Bedienung

3.5.4.1 Haltepunkte setzen, entfernen, einschalten und ausschalten

Es gibt grundsätzlich zwei verschiedene Möglichkeiten, Haltepunkte zu bearbeiten: in der Haltepunktliste oder direkt in der Regelanzeige.

Die Haltepunktliste wird über den Menüeintrag *„Breakpoints…“* im Menü *„Edit“* (de. *„Bearbeiten“*) oder über den Eintrag *„Edit breakpoint…“* (de. *„Breakpoint bearbeiten…“*) im Kontextmenü der Regelanzeige eingeblendet.



Die Einträge in den Spalten *„Rule“* (de. *„Regel“*), *„Line“* (de. *„Zeile“*) und *„Condition“* (de. *„Bedingung“*) können direkt bearbeitet werden. Die Änderung des Zustands in der Spalte *„State“* (de. *„Zustand“*) erfolgt über das Kontextmenü der Tabelle.

Das Kontextmenü besitzt folgende Einträge:

» *„New breakpoint“* (de. *„Neuer Breakpoint)“*: Fügt eine neue Zeile vor dem markierten Haltepunkt ein.

- *„Remove breakpoint(s)“* (de. *„Breakpoint(s) entfernen“*): Löscht alle markierten Haltepunkte.
- *„Enable breakpoint(s)“* (de. *„Breakpoint(s) einschalten“*): Schaltet alle markierten Haltepunkte ein.
- *„Disable breakpoint(s)“* (de. *„Breakpoint(s) ausschalten“*): Schaltet alle markierten Haltepunkte aus.

Ein einzelner Haltepunkt kann auch direkt in der Regelanzeige bearbeitet werden. Dazu muss die Regelanzeige aktiv sein, und der Cursor muss sich in der Zeile des zu bearbeitenden Haltepunkts befinden. Das Setzen, Löschen, Einschalten oder Ausschalten eines Haltepunkts erfolgt über die entsprechenden Einträge im Menü *„Debug“* (de. *„Debuggen“*) oder im Kontextmenü der Regelanzeige. Alternativ dazu können die Schaltflächen in der Symbolleiste verwendet werden.

Eingeschaltete Haltepunkte sind durch ein „+“-Symbol gekennzeichnet, ausgeschaltete Haltepunkte durch ein „-“-Symbol.

3.5.4.2 Ausdrücke anzeigen

Der Wert von Ausdrücken kann einmalig oder permanent angezeigt werden.

Um einen Ausdruck einmalig anzuzeigen, muss er im Eingabefeld für Ausdrücke und Anweisungen eingetragen oder in der Regelanzeige selektiert sein. Danach kann über den Menüeintrag *„Print“* (de. *„Wert ausgeben“*) im Menü *„Debug“* (de. *„Debuggen“*) bzw. im Kontextmenü oder über die zugehörige Schaltfläche in der Symbolleiste der Wert des Ausdrucks im Ausgabefenster angezeigt werden.

Ein Ausdruck wird ständig angezeigt, wenn er in der Liste der Überwachungsausdrücke eingetragen ist und die Regel, in der er definiert wurde, momentan abgearbeitet wird. Ändert sich der Wert eines Ausdrucks, dann wird der zugehörige Tabelleneintrag farblich hervorgehoben. Tabelleneinträge, die in momentan nicht abgearbeiteten Regeln definiert wurden, sind insensitiv.

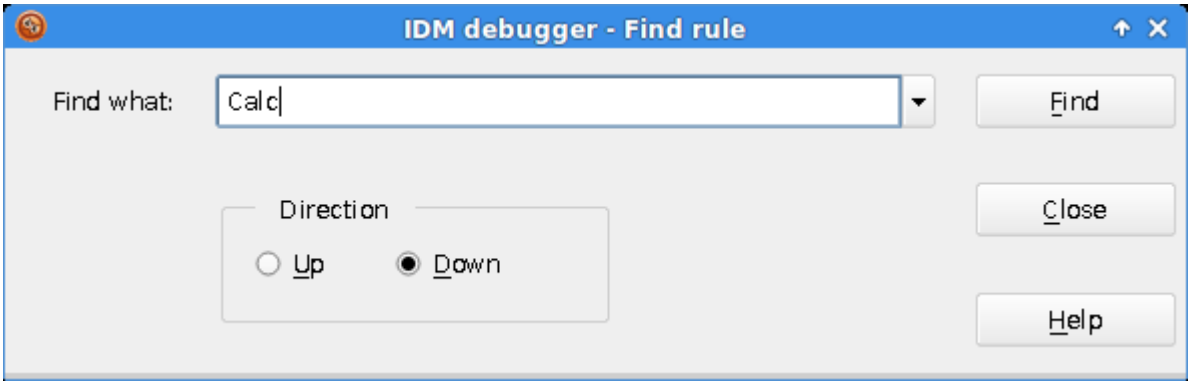
3.5.4.3 Variable oder Attribute ändern

Es gibt zwei Möglichkeiten, Werte von Variablen oder Attributen des zu debuggenden Dialogs zu ändern:

- Anweisung im Eingabefeld für Ausdrücke und Anweisungen eintragen und Zuweisung über den Menüeintrag *„Eval“* (de. *„Wert zuweisen“*) im Menü *„Debug“* (de. *„Debuggen“*) bzw. im Kontextmenü des Eingabefelds oder über die zugehörige Schaltfläche in der Symbolleiste ausführen.
- Wert direkt in der Liste der Überwachungsausdrücke ändern (Doppelklick auf Wertspalte).
- Variablenwert direkt in der Variablenliste im Überwachungsbereich ändern (Doppelklick auf Wertspalte).
- Attributwert über das jeweilige Objekt in der Objektansicht ändern (Objekt in die Überwachung aufnehmen und in der Attributliste Doppelklick auf entsprechenden Attributwert)

3.5.4.4 Regeln suchen

Es besteht die Möglichkeit, Regeln im Regel-Browser zu suchen. Die Dialogbox zur Regelsuche kann über den Menüeintrag *„Find…“* (de. *„Suchen…“*) im Menü *„Edit“* (de. *„Bearbeiten“*) bzw. im Kontextmenü des Regel-Browsers oder über den Accelerator `F3` aktiviert werden.



In der Combobox *„Find what“* (de. *„Suche nach“*) kann eine Zeichenfolge, die im Namen der gesuchten Regel enthalten ist, eingetragen werden. Die Suche beachtet Groß- und Kleinschreibung. Das Drücken der Schaltfläche *„Find“* (de. *„Suchen“*) startet die Suche. Die nächste Regel, welche die eingegebene Zeichenfolge enthält, wird danach im Regel-Browser markiert. Ein erneutes Drücken der Schaltfläche sucht die nächste passende Regel. Die Suchrichtung kann über die Radiobuttons bei *„Direction“* (de. *„Richtung“*) eingestellt werden. Die letzten 5 Suchmuster werden gespeichert und sind direkt in der Liste von *„Find what“* selektierbar.

3.5.4.5 Ausschneiden, Kopieren und Einfügen

Es besteht die Möglichkeit, Ausdrücke oder Anweisungen zwischen Eingabefeld für Ausdrücke und Anweisungen, Regelanzeige und Liste der Überwachungsausdrücke über den Ausschneiden/Kopieren/Einfügen-Mechanismus auszutauschen. Dazu können die üblichen Acceleratoren `Strg` + `X` (Ausschneiden), `Strg` + `C` (Kopieren) und `Strg` + `V` (Einfügen) oder das Kontextmenü des Systems benutzt werden. Das Systemmenü wird angezeigt, wenn man gleichzeitig die `Strg`-Taste und die rechte Maustaste drückt.

3.5.4.6 Debuggerzustand sichern, Debugger verlassen und wieder aufsetzen

Diese Funktionen werden über das Menü *„File“* aktiviert, das folgende Einträge enthält:

Save State…

Dieser Eintrag entspricht dem Kommando `state` der STDIO-Schnittstelle. Sobald er ausgewählt wird, erscheint eine Dialogbox, wo der Name der Datei einzugeben ist, in der der Debuggerzustand gesichert werden soll. Wird kein Dateiname eingegeben, erfolgt die Ausgabe des Zustandes im Sekundärfenster *„Output“*.

Execute…

Dieser Eintrag entspricht dem Kommando `execute` der STDIO-Schnittstelle. Sobald er ausgewählt wird, erscheint eine Dialogbox, wo der Name der Datei einzugeben ist, aus der die auszuführenden Debugger-Kommandos gelesen werden sollen.

Exit

Durch Auswahl dieses Eintrags kann der Server verlassen werden. Geschieht dies während der laufenden Regelabarbeitung, bleibt der Client davon unbeeinflusst, d.h. auch die Haltepunkte bleiben vorhanden. Sobald nun die Regelabarbeitung unterbrochen wird und der Client versucht, mit dem nicht mehr vorhandenen Server zu kommunizieren, führt dies zu einem Fehler.

Wird der Server dagegen verlassen, während die Regelabarbeitung unterbrochen ist, wird auch der Client regulär beendet.

4 Die STDIO-Schnittstelle

Zur Illustration der Benutzung der STDIO-Schnittstelle des DM-Debuggers seien hier die Ein- und Ausgaben einer **Beispielsitzung** angegeben, in deren Verlauf die Funktionalität des Debuggers illustriert wird.

Aufruf

Die STDIO-Schnittstelle des DM-Debuggers wird aufgerufen, indem dem Programm die Option **-IDMDBG=STDIO** übergeben wird. Sollen etwa in einem DM-Skript **prime.dlg** mit Hilfe des Simulators **idm** Fehler gesucht werden, lautet der Aufruf:

```
idm -IDMDBG=STDIO prime.dlg
```

Hier wird ein Debugginglauf mit dem folgenden Dialogskript beschrieben, das in der Datei **prime.dlg** gespeichert ist. Das Dialogskript berechnet die Primzahlen im Bereich Zwei bis Hundert.

```
dialog TestDialog

!! Determine the greatest integer with a square not greater than X
rule integer Root(integer X)
{
  variable integer I := 0;
  variable integer J := X;
  variable integer K;

  if X <= 0 then
    return 0;
  endif

  while J - I > 1 do
    K := (I + J)/2;
    if K*K > X then
      J := K;
    else
      I := K;
    endif
  endwhile

  return I;
}

!! Determine if M is prime
rule boolean IsPrime(integer M)
{
  variable boolean P:=true;
  variable integer W;
```

```
  variable integer O;

  W := Root(M);

  O := 2;
  while (O <= W) and P do
    P := 0 <> (M%O);
    O := O + 1;
  endwhile

  return P;
}

!! Dialog start rule
on dialog start
{
  variable integer I;

  for I:=2 to 100 do
    if IsPrime(I) then
      print I;
    endif
  endfor
}
```

4.1 Starten des Debuggers

Der Debugginglauf beginnt mit dem Start des DM-Skriptes mit Hilfe des Simulators. Nach dem Start bleibt der Debugger bei der ersten Anweisung der ersten ausgeführten Regel stehen und wartet auf Eingaben des Benutzers.

Immer wenn sich die betrachtete Programmstelle geändert haben könnte, weil DM-Anweisungen ausgeführt wurden oder weil vom Debugger verlangt wurde, eine andere Stelle im Programm anzuzeigen, wird die als nächstes auszuführende Zeile ausgegeben. Die Zeile erscheint dabei nicht unbedingt so, wie sie eingegeben wurde, sondern so, wie sie der DM-Editor ausgeben würde, ergänzt um eine führende Zeilennummer.

Nach der als nächstes auszuführenden Zeile wird immer ausgegeben, welche Regel, welche Stackebene und welche Zeile gerade betrachtet wird. Die Regeln sind durchnummeriert, und statt eines Regelnamens kann man stets auch die Regelnummer mit vorgeschaltetem kleinem *„r“* schreiben. Die Regelnummer der betrachteten Regel ist ebenfalls angegeben.

Damit ist die folgende Ausgabe beim Programmstart erklärt:

```
idm -IDMDBG=STDIO prime.dlg
7    for I := 2 to 100 do
(Rule r1:"on dialog start" at depth 1 line 7)>
```

4.2 Regeln betrachten

Die Liste der vorhandenen Regeln kann mit dem Kommando `rules` ausgegeben werden. Wenn als Argument ein Muster angegeben wird, das neben Buchstaben Sternchen und Fragezeichen enthalten darf, dann werden nur die Regeln ausgegeben, deren Name zu diesem Muster passt. Sternchen stehen dabei für beliebige Zeichenfolgen, jedes Fragezeichen für genau ein beliebiges Zeichen.

Der Code einer einzigen Regel wird mit dem Befehl `list` ausgegeben. Je nach Angabe einer Regel oder einer Zeilennummer wird die ganze Regel oder nur ein Teil derselben ausgegeben. In Kapitel „Der Befehlssatz des Debuggers“ steht, unter welchen Bedingungen was ausgegeben wird.

Wenn ein Argument für einen Befehl Leerzeichen enthält, wie im nun folgenden Beispiel der Name einer Regel, die mit dem letzten `list`-Befehl ausgegeben werden soll, dann müssen diese Leerzeichen mit Rückstrichen markiert werden, damit die STDIO-Schnittstelle das Argument an diesen Stellen nicht auftrennt.

```
rules
r1: "on dialog start" in module   1: TestDialog
r2: "TestDialog.Root" in module   1: TestDialog
r3: "TestDialog.IsPrime" in module   1: TestDialog
(Rule r1:"on dialog start" at depth 1 line 7)>rules T*
r2: "TestDialog.Root" in module   1: TestDialog
r3: "TestDialog.IsPrime" in module   1: TestDialog
(Rule r1:"on dialog start" at depth 1 line 7)>list
5    variable integer I;
6
7    for I := 2 to 100 do
8        if IsPrime(I) then
9            print I;
10        endif
11    endfor
12  }
(Rule r1:"on dialog start" at depth 1 line 7)>list 9
7    for I := 2 to 100 do
8        if IsPrime(I) then
9            print I;
10        endif
11    endfor
12  }
(Rule r1:"on dialog start" at depth 1 line 7)>list 5
3 on dialog start
4 {
5    variable integer I;
6
7    for I := 2 to 100 do
8        if IsPrime(I) then
9            print I;
10        endif
```

```
(Rule r1:"on dialog start" at depth 1 line 7)>list r2
1 !! Determine greatest integer with a square not greater than X
3 rule integer Root (integer X input)
4 {
5    variable integer I := 0;
6    variable integer J := X;
7    variable integer K;
8
9    if (X <= 0) then
10        return 0;
11    endif
12    while ((J - I) > 1) do
13       K := ((I + J) / 2);
14       if ((K * K) > X) then
15           J := K;
16       else
17           I := K;
18       endif
19    endwhile
20    return I;
21 }
(Rule r1:"on dialog start" at depth 1 line 7)>list\ on dialog\ start
1 !! Dialog start rule
2 on dialog start
3 {
4    variable integer I;
5    for I := 2 to 100 do
6        if IsPrime(I) then
7            print I;
8        endif
9    endfor
10 }
```

4.3 Kontrollierte Ausführung des Programms

Die Befehle `next`, `step` und `finish` dienen dazu, einzelne Anweisungen auszuführen. `next` schreitet dabei über Unterprogramme hinweg, `step` geht hinein und `finish` führt so lange Anweisungen aus, bis eine höhere Stackebene erreicht ist, erlaubt es also, die momentan betrachtete Regel bis zum Ende laufen zu lassen.

Weiter wird hier demonstriert, dass die Eingabe einer Leerzeile zur wiederholten Ausführung des zuletzt eingegebenen Befehls führt. Die Eingabe von Leerzeichen wird hier durch das Zeichen `" "` angedeutet.

Zudem wird gezeigt, dass Befehle abgekürzt werden können. Statt `step`, `finish` und `list` werden hier die Formen `s`, `fin` und `l` benutzt.

```
(Rule r1:"on dialog start" at depth 1 line 7)>next
8        if IsPrime(I) then
(Rule r1:"on dialog start" at depth 1 line 8)>" "
9            print I;
(Rule r1:"on dialog start" at depth 1 line 9)>" "
2
8        if IsPrime(I) then
(Rule r1:"on dialog start" at depth 1 line 8)>" "
9            print I;
(Rule r1:"on dialog start" at depth 1 line 9)>" "
3
8        if IsPrime(I) then
(Rule r1:"on dialog start" at depth 1 line 8)>" "
8        if IsPrime(I) then
(Rule r1:"on dialog start" at depth 1 line 8)>" "
9            print I;
(Rule r1:"on dialog start" at depth 1 line 9)>" "
5
8        if IsPrime(I) then
(Rule r1:"on dialog start" at depth 1 line 8)>s
5     variable boolean P := true;
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)>l r3
1 !! Determine if M is prime
2
3 rule boolean IsPrime (integer M)
4 {
5    variable boolean P:=true;
6    variable integer W;
7    variable integer O;
8
9    W := Root(M);
10    O := 2;
11    while (O <= W) and P do
12       P := 0 <> (M % O);
13       O := O + 1;
14    endwhile
15    return P;
16 }
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)>s
9     W := Root(M);
(Rule r3:"TestDialog.IsPrime" at depth 2 line 9)>s
5     variable integer I := 0;
(Rule r2:"TestDialog.Root" at depth 3 line 5)>l r2
1 !! Determine greatest integer with a square not greater than X
2
3 rule integer Root (integer X input)
4 {
```

```
5    variable integer I := 0;
6    variable integer J := X;
7    variable integer K;
8
9    if (X <= 0) then
10        return 0;
11    endif
12    while ((J - I) > 1) do
13       K := ((I + J) / 2);
14       if ((K * K) > X) then
15           J := K;
16       else
17           I := K;
18       endif
19    endwhile
20    return I;
21 }
(Rule r2:"TestDialog.Root" at depth 3 line 5)>fin
10     O := 2;
(Rule r3:"TestDialog.IsPrime" at depth 2 line 10)>fin
8        if IsPrime(I) then
```

4.4 Ausdrücke anzeigen

Mit einer Anzahl von Befehlen ist es möglich, sich die Werte von Ausdrücken einmal oder immer wieder anzeigen zu lassen. Für eine einmalige Ausgabe ist der Befehl `print` gedacht. Der Befehl `display` meldet einen Ausdruck bei einer Regel als Überwachungsausdruck an. Wann immer sich die betrachtete Stelle in einer Regel befindet, werden die zu dieser Regel gehörigen Ausdrücke neu ausgewertet und ihre Werte ausgegeben. Die Liste aller überwachten Ausdrücke kann mit `dplist` ausgegeben werden, und `undisplay` entfernt einen Ausdruck aus dieser Liste.

Wenn die Auswertung eines Ausdrucks zu einem Fehler führt, dann wird auch das ausgegeben. Wie beim Dialog Manager üblich, wird diese Fehlermeldung durch die Ausgabe der Regel, in der der Fehler aufgetreten ist, ergänzt, und der problematische Ausdruck wird hervorgehoben dargestellt. Die in einem solchen Falle ausgegebene Regel ist allerdings maschinengeneriert.

```
(Rule r1:"on dialog start" at depth 1 line 8)>p I
7
(Rule r1:"on dialog start" at depth 1 line 8)>disp I
1: I = 7
8        if IsPrime(I) then
(Rule r1:"on dialog start" at depth 1 line 8)>s
5     variable boolean P := true;
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)>disp M
2: M = 7
5     variable boolean P := true;
```

```
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)>disp P
2: M = 7
3: P = nothing
5     variable boolean P := true;
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)>disp W
2: M = 7
3: P = nothing
4: W = nothing
5     variable boolean P := true;
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)>disp O
2: M = 7
3: P = nothing
4: W = nothing
5: O = nothing
5     variable boolean P := true;
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)>n
2: M = 7
3: P = true
4: W = nothing
5: O = nothing
9     W := Root(M);
(Rule r3:"TestDialog.IsPrime" at depth 2 line 9)>s
5     variable integer I := 0;
(Rule r2:"TestDialog.Root" at depth 3 line 5)>disp X
6: X = 7
5     variable integer I := 0;
(Rule r2:"TestDialog.Root" at depth 3 line 5)>disp I
6: X = 7
7: I = nothing
5     variable integer I := 0;
(Rule r2:"TestDialog.Root" at depth 3 line 5)>disp J
6: X = 7
7: I = nothing
8: J = nothing
5     variable integer I := 0;
(Rule r2:"TestDialog.Root" at depth 3 line 5)>disp K
6: X = 7
7: I = nothing
8: J = nothing
9: K = nothing
5     variable integer I := 0;
(Rule r2:"TestDialog.Root" at depth 3 line 5)>disp J-I
disp J-I
***ERROR IN EVAL: invalid types
rule anyvalue A9 (integer X input)
variable integer I;
variable integer J;
```

```
variable integer K;
return (J - I);
6: X = 7
7: I = nothing
8: J = nothing
9: K = nothing
10: J-I = nothing
5     variable integer I := 0;
(Rule r2:"TestDialog.Root" at depth 3 line 5)>n
***ERROR IN EVAL: bad type combination
rule anyvalue A9 (integer X input)
variable integer I;
variable integer J;
variable integer K;
return (J - I);
6: X = 7
7: I = 0
8: J = nothing
9: K = nothing
10: J-I = nothing
6     variable integer J := X;
(Rule r2:"TestDialog.Root" at depth 3 line 6)>n
6: X = 7
7: I = 0
8: J = 7
9: K = nothing
10: J-I = 7
9     if (X <= 0) then
(Rule r2:"TestDialog.Root" at depth 3 line 9)>n
6: X = 7
7: I = 0
8: J = 7
9: K = nothing
10: J-I = 7
12     while ((J - I) > 1) do
(Rule r2:"TestDialog.Root" at depth 3 line 12)>n
6: X = 7
7: I = 0
8: J = 7
9: K = nothing
10: J-I = 7
13        K := ((I + J) / 2);
(Rule r2:"TestDialog.Root" at depth 3 line 13)>n
6: X = 7
7: I = 0
8: J = 7
9: K = 3
```

```
10: J-I = 7
14        if ((K * K) > X) then
(Rule r2:"TestDialog.Root" at depth 3 line 14)>n
6: X = 7
7: I = 0
8: J = 7
9: K = 3
10: J-I = 7
15            J := K;
(Rule r2:"TestDialog.Root" at depth 3 line 15)>n
6: X = 7
7: I = 0
8: J = 3
9: K = 3
10: J-I = 3
13        K := ((I + J) / 2);
(Rule r2:"TestDialog.Root" at depth 3 line 13)>n
6: X = 7
7: I = 0
8: J = 3
9: K = 1
10: J-I = 3
14        if ((K * K) > X) then
(Rule r2:"TestDialog.Root" at depth 3 line 14)>n
6: X = 7
7: I = 0
8: J = 3
9: K = 1
10: J-I = 3
17            I := K;
(Rule r2:"TestDialog.Root" at depth 3 line 17)>n
6: X = 7
7: I = 1
8: J = 3
9: K = 1
10: J-I = 2
13        K := ((I + J) / 2);
(Rule r2:"TestDialog.Root" at depth 3 line 13)>n
6: X = 7
7: I = 1
8: J = 3
9: K = 2
10: J-I = 2
14        if ((K * K) > X) then
(Rule r2:"TestDialog.Root" at depth 3 line 14)>n
6: X = 7
7: I = 1
```

```
8: J = 3
9: K = 2
10: J-I = 2
17            I := K;
(Rule r2:"TestDialog.Root" at depth 3 line 17)>n
6: X = 7
7: I = 2
8: J = 3
9: K = 2
10: J-I = 1
20     return I;
(Rule r2:"TestDialog.Root" at depth 3 line 20)>n
2: M = 7
3: P = true
4: W = 2
5: O = nothing
10     O := 2;
(Rule r3:"TestDialog.IsPrime" at depth 2 line 10)>n
2: M = 7
3: P = true
4: W = 2
5: O = 2
11     while ((O <= W) and P) do
(Rule r3:"TestDialog.IsPrime" at depth 2 line 11)>n
2: M = 7
3: P = true
4: W = 2
5: O = 2
12        P := (0 <> (M % O));
(Rule r3:"TestDialog.IsPrime" at depth 2 line 12)>n
2: M = 7
3: P = true
4: W = 2
5: O = 2
13        O := O + 1;
(Rule r3:"TestDialog.IsPrime" at depth 2 line 13)>n
2: M = 7
3: P = true
4: W = 2
5: O = 3
15        return P;
(Rule r3:"TestDialog.IsPrime" at depth 2 line 15)>n
1: I = 7
9            print I;
(Rule r1:"on dialog start" at depth 1 line 9)>dplist
Displayed expr 1: I in rule "on dialog start"
Displayed expr 2: M in rule "TestDialog.IsPrime"
```

```
Displayed expr 3: P in rule "TestDialog.IsPrime"
Displayed expr 4: W in rule "TestDialog.IsPrime"
Displayed expr 5: O in rule "TestDialog.IsPrime"
Displayed expr 6: X in rule "TestDialog.Root"
Displayed expr 7: I in rule "TestDialog.Root"
Displayed expr 8: J in rule "TestDialog.Root"
Displayed expr 9: K in rule "TestDialog.Root"
Displayed expr 10: J-I in rule "TestDialog.Root"
(Rule r1:"on dialog start" at depth 1 line 9)>undisplay 1
Deleted displayed expression 1
(Rule r1:"on dialog start" at depth 1 line 9)>n
7
8        if IsPrime(I) then
(Rule r1:"on dialog start" at depth 1 line 8)>n
8        if IsPrime(I) then
```

4.5 Regel-Stack anzeigen

Der Befehl `stack` zeigt den aktuellen Regel-Stack an, die Befehle `up` und `down` erlauben die Navigation im Regel-Stack. Am Anfang wird die Tiefe angegeben, dann der Name der Regel und schließlich die Zeile.

Das Ergebnis des Befehls `finish` in diesem Abschnitt illustriert, dass dieser Befehl bei der Bestimmung der Stackebene, auf der wieder angehalten werden wird, nicht von der augenblicklichen Ausführungsstelle, sondern von der mit `up` oder `down` angesteuerten Stelle ausgeht.

```
(Rule r1:"on dialog start" at depth 1 line 8)>stack
1:               on dialog start    8
(Rule r1:"on dialog start" at depth 1 line 8)>n
8        if IsPrime(I) then
(Rule r1:"on dialog start" at depth 1 line 8)>s
2: M = 10
3: P = nothing
4: W = nothing
5: O = nothing
5     variable boolean P := true;
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)>stack
1:               on dialog start    8
2:            TestDialog.IsPrime    5
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)>n
2: M = 10
3: P = true
4: W = nothing
5: O = nothing
9     W := Root(M);
(Rule r3:"TestDialog.IsPrime" at depth 2 line 9)>s
***ERROR IN EVAL: invalid types
```

```
rule anyvalue A9 (integer X input)
variable integer I;
variable integer J;
variable integer K;
return (J - I);
6: X = 10
7: I = nothing
8: J = nothing
9: K = nothing
10: J-I = nothing
5     variable integer I := 0;
(Rule r2:"TestDialog.Root" at depth 3 line 5)>stack
1:               on dialog start    8
2:            TestDialog.IsPrime    9
3:               TestDialog.Root    5
(Rule r2:"TestDialog.Root" at depth 3 line 5)>up
2: M = 10
3: P = true
4: W = nothing
5: O = nothing
9     W := Root(M);
(Rule r3:"TestDialog.IsPrime" at depth 2 line 9)>down
***ERROR IN EVAL: invalid types
rule anyvalue A9 (integer X input)
variable integer I;
variable integer J;
variable integer K;
return (J - I);
6: X = 10
7: I = nothing
8: J = nothing
9: K = nothing
10: J-I = nothing
5     variable integer I := 0;
(Rule r2:"TestDialog.Root" at depth 3 line 5)>undisplay 10
Deleted displayed expression 10
(Rule r2:"TestDialog.Root" at depth 3 line 5)up
2: M = 10
3: P = true
4: W = nothing
5: O = nothing
9     W := Root(M);
(Rule r3:"TestDialog.IsPrime" at depth 2 line 9)>fin
8        if IsPrime(I) then
```

4.6 Haltepunkte setzen

Ein sehr mächtiges Hilfsmittel beim Debuggen besteht darin, Haltepunkte zu setzen. Wenn die Abarbeitung an einer solchen Stelle ankommt, wird der Debugger aktiviert.

Das Kommando `break` erlaubt es, solche Haltepunkte zu setzen. Regel und Zeilennummer können dabei implizit bestimmt oder explizit angegeben werden. Wie das genau geschieht, kann dem Kapitel „Der Befehlssatz des Debuggers“ entnommen werden.

Ein Haltepunkt kann mit einer Bedingung ausgestattet sein. Der hier angegebene Ausdruck wird bei Erreichen des Haltepunkts ausgewertet. Wenn das Ergebnis *false* ist, wird der Haltepunkt als inaktiv behandelt, ansonsten ist er gültig.

Weiter kann ein Haltepunkt mit den Befehlen `enable` und `disable` aktiviert und deaktiviert werden.

Die Liste aller Haltepunkte kann man mit `bplist` ausgeben, `delete` löscht einen Haltepunkt endgültig aus der Liste.

Der Befehl `continue` schließlich setzt die Ausführung des Programms fort, bis der nächste Haltepunkt erreicht ist.

```
(Rule r1:"on dialog start" at depth 1 line 8)>break
Breakpoint 1 set at rule "on dialog start", line 8
(Rule r1:"on dialog start" at depth 1 line 8)>continue
11
8        if IsPrime(I) then
(Rule r1:"on dialog start" at depth 1 line 8)>disp I
11: I = 12
8        if IsPrime(I) then
(Rule r1:"on dialog start" at depth 1 line 8)>c
11: I = 13
8        if IsPrime(I) then
(Rule r1:"on dialog start" at depth 1 line 8)>c
13
11: I = 14
8        if IsPrime(I) then
(Rule r1:"on dialog start" at depth 1 line 8)>b r3
Breakpoint 2 set at rule "TestDialog.IsPrime", line 5
(Rule r1:"on dialog start" at depth 1 line 8)>c
2: M = 14
3: P = nothing
4: W = nothing
5: O = nothing
5     variable boolean P := true;
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)>br r2 15
Breakpoint 3 set at rule "TestDialog.Root", line 15
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)>c
6: X = 14
7: I = 0;
8: J = 14
```

```
9: K = 7
15            J := K;
(Rule r2:"TestDialog.Root" at depth 3 line 15)>bplist
Bp   1: on dialog start               , line    8, enabled
Bp   2: TestDialog.IsPrime            , line    5, enabled
Bp   3: TestDialog.Root               , line   15, enabled+
(Rule r2:"TestDialog.Root" at depth 3 line 15)>disable 1
Disabled breakpoint 1
(Rule r2:"TestDialog.Root" at depth 3 line 15)>c
6: X = 14
7: I = 3;
8: J = 7
9: K = 5
15            J := K;
(Rule r2:"TestDialog.Root" at depth 3 line 15)>disable 3
Disabled breakpoint 3
(Rule r2:"TestDialog.Root" at depth 3 line 15)>c
2: M = 15
3: P = nothing
4: W = nothing
5: O = nothing
5     variable boolean P := true;
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)>bplist
Bp   1: on dialog start               , line    8, disabled
Bp   2: TestDialog.IsPrime            , line    5, enabled
Bp   3: TestDialog.Root               , line   15, disabled
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)>delete 3
Deleted breakpoint 3
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)>bplist
Bp   1: on dialog start               , line    8, disabled
Bp   2: TestDialog.IsPrime            , line    5, enabled
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)>br r1 8 if I=17
Breakpoint 4 set at rule "on dialog start", line 8 (if I=17)
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)>c
2: M = 16
3: P = nothing
4: W = nothing
5: O = nothing
5     variable boolean P := true;
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)del 2
Deleted breakpoint 2
(Rule r3:"TestDialog.IsPrime" at depth 2 line 5)c
11: I = 17
8        if IsPrime(I) then
```

4.7 Debuggerzustand sichern, Debugger verlassen und wieder aufsetzen

Ein Programmlauf kann nur durch Abbruch und völligen Neustart des Programms neu gestartet werden. Damit in einem solchen Fall die bestehenden Überwachungsausdrücke und die Haltepunkte leichter wieder gesetzt werden können, gibt es die Befehle `state` und `execute`. `state` sichert den Zustand in einer Datei (oder zeigt ihn auf dem Bildschirm an), `execute` stellt diesen Zustand wieder her.

```
(Rule r1:"on dialog start" at depth line 8)>state
break r1 8 disabled
break r1 8 if I=17
display M r3
display P r3
display W r3
display O r3
display X r2
display I r2
display J r2
display K r2
display I r1
(Rule r1:"on dialog start" at depth line 8)>state /tmp/stat.dbg
(Rule r1:"on dialog start" at depth line 8)>quit

$ idm -IDMDBG=STDIO prime.dlg
7       for I := 2 to 100 do
(Rule r1:"on dialog start" at depth 1 line 7)>exec /tmp/stat.dbg
Breakpoint 1 set at rule òn dialog start", line 8
Breakpoint 1 set at rule òn dialog start", line 8 (if I=17)
9: I = nothing
7     for I := 2 to 100 do
(Rule r1:"on dialog start" at depth 1 line 7)>bplist
Bp   1: on dialog start               , line     8, disabled
Bp   2: on dialog start               , line     8, enabled if I=17
(Rule r1:"on dialog start" at depth 1 line 7)>dplist
Displayed expr 1: M in rule "TestDialog.IsPrime"
Displayed expr 2: P in rule "TestDialog.IsPrime"
Displayed expr 3: W in rule "TestDialog.IsPrime"
Displayed expr 4: O in rule "TestDialog.IsPrime"
Displayed expr 5: X in rule "TestDialog.Root"
Displayed expr 6: I in rule "TestDialog.Root"
Displayed expr 7: J in rule "TestDialog.Root"
Displayed expr 8: K in rule "TestDialog.Root"
Displayed expr 9: I in rule "on dialog start"
```

4.8 Hilfe

Die verfügbaren Befehle lassen sich mit dem Kommando `help` ausgeben.

```
(Rule r1:"on dialog start" at depth 1 line 7)>help
break [<rule>] [<line>] [disabled] [if <condition>]: break at current or
given rule and/or line (if wished: disable this breakpoint, or only break if
condition is fulfilled)
bplist: print list of breakpoints
continue: go on
disable <breakpoint>: disable given breakpoints
display <expression> [<rule>]: display given expression when in current
or given rule
delete <breakpoint>: delete the given breakpoints
down: go down a stack frame
dplist: show list of displayed expressions
enable <breakpoint>: enable given breakpoints
execute <file>: execute commands from file
finish: go on and break at exit from current rule
help: give help
list [<rule>] [<line>]: print current or given rule (at line, if given)
modules: print list of modules
next: go on and break at next statement in this rule
print <expression>: print given expression
quit: quit program
rules [<pattern> | rebuild]: print rules matching <pattern> with
wildcards ? and *, or rebuild rules list
step: go on and break at next statement
stack: print execution stack
state [<file>]: show state, or save to file
up: go up a stack frame
undisplay <id>: delete a displayed expression
(Rule r1:"on dialog start" at depth 1 line 7)>quit
```

4.9 Die Eingabe von Befehlszeilen

Bei der Benutzung der STDIO-Schnittstelle werden dem DM-Debugger die Befehle ähnlich wie bei Benutzung einer Unix-Shell eingegeben. Das bedeutet, dass als erstes ein Kommandoname angegeben werden muss, der Argumente erhalten kann. Argumente werden gewöhnlich durch Leerzeichen voneinander getrennt. Wenn ein Argument Leerzeichen enthält, müssen diese mit Rückstrichen (\) markiert werden, damit der DM-Debugger die Eingabe an einem solchen Leerzeichen nicht in einzelne Argumente auseinanderbricht. Als zusätzliche Vereinfachung wird innerhalb von Strings, also Zeichenketten, die durch doppelte Anführungszeichen (") begrenzt werden, die Eingabezeile nicht an den Leerzeichen der Argumente aufgetrennt. Auch in Strings dient der Rückstrich als Markierung. Hier ist er dazu da, doppelte Anführungsstriche, die im String auftreten, zu markieren, so dass sie nicht als Stringende fungieren.

Es ist auch eine Syntax für Kommentarzeichen vorgesehen. Zeilen, die mit einem Doppelkreuz (#) beginnen, werden ignoriert. Alle anderen Zeilen sind Befehlszeilen.

Der DM-Debugger erleichtert die Eingabe von Befehlen auf zweierlei Weise:

- Wenn die gleiche Befehlszeile mehrfach hintereinander eingegeben werden soll (etwa beim Durchlaufen einer Regel mit wiederholten `step`-Befehlen), dann genügt es, wenn ab dem zweiten Male eine Leerzeile eingegeben wird. Der DM-Debugger führt bei Eingabe einer Leerzeile den zuvor eingegebenen Befehl nochmals aus.
- Es genügt, statt eines gesamten Befehls nur dessen Anfangsbuchstaben einzugeben. Der erste Befehl, der in einem Anfang mit den eingegebenen Zeichen übereinstimmt, wird ausgeführt. Dabei werden die Befehle in der Reihenfolge mit der Eingabe verglichen, in der der Befehl `help` sie ausgibt. Ein Beispiel: Die Eingabe *„sta“*, die sowohl zum Befehl `stack` als auch zum Befehl `state` passt, wird als `stack` verstanden, weil bei Eingabe von `help stack` vor `state` ausgegeben wird.

5 Der Befehlssatz des Debuggers

Der Befehlsvorrat des DM-Debuggers lehnt sich an den des verbreiteten GNU-Debuggers "GDB" an. Im folgenden werden die Befehle und ihre Argumente beschrieben.

Für die Angabe der Argumente wird eine spezielle Notation benutzt:

1. Ein Argument, das in geschweiften Klammern angegeben wird ({...}), kann entfallen.
2. Ein Begriff, der in spitzen Klammern angegeben wird (<...>), steht für einen variablen Wert. Dieser Klammertyp kann mit geschweiften Klammern kombiniert werden ({<...>}).
3. Ein Begriff, der in eckigen Klammern angegeben wird ([...]), steht für eine mehrfache Wiederholung des angegebenen Ausdrucks. Die Anzahl der Wiederholungen darf auch null betragen.
4. Ein Begriff, der ohne Klammern angegeben ist, muss wörtlich angegeben werden.
5. Begriffe, die innerhalb von Klammern durch einen senkrechten Strich getrennt sind, sind als Alternativen zu verstehen.

Ein Beispiel zur Illustration: Die Argumente des Befehls "break" sind folgendermaßen angegeben:

```
break {<Regel>} {<Zeile>} {disabled} {if <Bedingung>}
```

Dies bedeutet:

1. Als erstes Argument kann ein Ausdruck angegeben werden, der als Regelidentifikation gedeutet wird. Dieses Argument darf auch fehlen.
2. Als nächstes Argument kann ein Ausdruck kommen, der als Zeilennummer gedeutet wird. Auch dieses Argument darf fehlen.
3. Das nächste Argument kann wörtlich "disabled" lauten, darf aber auch fehlen.
4. Endlich darf ein wörtliches "if" folgen. Wenn es nicht da ist, muss die Folge der Argumente beendet sein, ansonsten muss noch ein weiteres Argument folgen, das eine Bedingung darstellt.

5.1 autostop - Anhalten in der ersten, ausgeführten Regel eines Moduls

Bedeutung

Der Dialog wird beim Debuggen in der ersten, ausgeführten Regel eines Moduls angehalten

Eingabe im grafischen Debugger

Im Hauptfenster des Debuggers ist das Kommando über das Menü ‚Options' erreichbar

Syntax

```
autostop (on|off)
```

Erklärung der Argumente

on

Anhalten in der ersten, ausgeführten Regel eines Moduls (Defaultwert)

off

Debugger wird nicht in der ersten, ausgeführten Regel eines Moduls angehalten

5.2 bplist - Liste der Haltepunkte

Bedeutung

Die Liste der Haltepunkte wird ausgegeben.

Eingabe im grafischen Debugger

Die Eingabe erfolgt implizit im Hauptfenster des Debuggers über das Menü "Edit" "Breakpoints…"

Syntax

```
bplist
```

Erklärung der Argumente

Dieser Befehl hat keine Argumente.

5.3 break - Setzen eines Haltepunkts

Bedeutung

Ein Haltepunkt wird auf eine angegebene oder die betrachtete Regel gesetzt, und zwar auf eine gewünschte Zeile. Der Haltepunkt kann deaktiviert erstellt werden. Wenn eine Bedingung angegeben wird, wird die Ausführung nur dann unterbrochen, wenn die optional angegebene Bedingung wahr ist.

Eingabe im grafischen Debugger

Selektion der Schaltfläche "Set Break" in der Symbolleiste.

Syntax

```
break {<rule>} {<line>} {disabled} {if <condition>}
```

Erklärung der Argumente

rule

Die Regel, in der ein Haltepunkt gesetzt werden soll. Wenn dieser Parameter fehlt, wird die betrachtete Regel dafür eingesetzt.

line

Die Zeile der Regel, in der der Haltepunkt gesetzt werden soll. Wenn sie fehlt und eine Regel explizit angegeben ist, wird der Haltepunkt auf den ersten Ausdruck der Regel gesetzt; wenn die Zeilennummer fehlt und keine Regel explizit angegeben ist, wird der Haltepunkt auf die betrachtete Zeile in der betrachteten Regel gesetzt.

disabled

Ist dieser Parameter angegeben, wird der Haltepunkt zunächst deaktiviert, sonst ist er aktiviert.

condition

Ist hier eine Bedingung angegeben, wird der Haltepunkt nur dann beachtet, wenn die zugehörige Bedingung erfüllt ist.

Achtung

Wenn der Ausdruck, der die Bedingung angibt, Leerzeichen außerhalb von Strings enthält, müssen diese mit Rückstrichen "\" markiert sein.

5.4 continue - Fortführung der Programmausführung

Bedeutung

Die Programmausführung wird fortgesetzt, bis ein flüchtiger oder fester Haltepunkt erreicht wird. Dann hält der Debugger und wartet auf Benutzereingaben.

Eingabe im grafischen Debugger

Selektion der Schaltfläche "Continue" in der Symbolleiste.

Syntax

```
continue
```

Erklärung der Argumente

Dieser Befehl hat keine Argumente.

5.5 delete - Löschen von Haltepunkten

Bedeutung

Die angegebenen Haltepunkte werden gelöscht.

Eingabe im grafischen Debugger

Das Löschen von Haltepunkten passiert im grafischen Debugger im Unterfenster "breakpoints".

Syntax

```
delete [<breakpoints>]
```

Erklärung der Argumente

breakpoints

In diesem Parameter kann eine Liste von Haltepunkten angegeben werden, die gelöscht werden sollen.

5.6 disable - Deaktivierung von Haltepunkten

Bedeutung

Die angegebenen Haltepunkte werden deaktiviert.

Eingabe im grafischen Debugger

Das Deaktivieren von Haltepunkten passiert im grafischen Debugger im Unterfenster "breakpoints".

Syntax

```
disable [<breakpoints>]
```

Erklärung der Argumente

breakpoints

In diesem Parameter kann eine Liste von Haltepunkten angegeben werden, die deaktiviert werden sollen.

5.7 display - Einrichtung eines Überwachungsausdrucks

Bedeutung

Es wird ein Überwachungsausdruck für eine Regel eingerichtet.

Eingabe im grafischen Debugger

Die Einrichtung im grafischen Editor wird durch die Eingabe des Ausdrucks in das Hauptfenster und anschließender Selektion der Schaltfläche *„Display“* vorgenommen.

Syntax

```
display <expression> {<rule>}
```

Erklärung der Argumente

expression

In diesem Parameter wird der Überwachungsausdruck angegeben. Dieser wird immer angezeigt, wenn die Regel betrachtet wird.

Achtung

Wenn der Überwachungsausdruck Leerzeichen außerhalb von Strings enthält, müssen diese mit Rückstrichen markiert werden.

rule

Die Regel, für die der Überwachungsausdruck einzurichten ist. Wenn dieses Argument fehlt, wird die betrachtete Regel benutzt.

5.8 down - Verringern der betrachteten Stackebene

Bedeutung

Die betrachtete Stackebene wird um eine Einheit verringert. Die kleinste Stackebene entspricht dabei dem Ort der tiefsten Aufrufschachtelung.

Eingabe im grafischen Debugger

Die Eingabe erfolgt durch Selektion der Schaltfläche "Down" in der Symbolleiste.

Syntax

```
down
```

Erklärung der Argumente

Dieser Befehl hat keine Argumente.

5.9 dplist - Liste der Überwachungsausdrücke

Bedeutung

Die Überwachungsausdrücke werden aufgelistet.

Eingabe im grafischen Debugger

Die Ausgabe im grafischen Editor erfolgt im Fenster "displayed expressions".

Syntax

```
dplist
```

Erklärung der Argumente

Dieser Befehl hat keine Argumente.

5.10 enable - Aktivierung von Haltepunkten

Bedeutung

Die angegebenen Haltepunkte werden wieder aktiviert.

Eingabe im grafischen Debugger

Das Aktivieren von Haltepunkten passiert im grafischen Debugger im Unterfenster "breakpoints".

Syntax

```
enable [<breakpoints>]
```

Erklärung der Argumente

breakpoints

In diesem Parameter kann eine Liste von Haltepunkten angegeben werden, die aktiviert werden sollen.

5.11 eval - Ausführung einer DM-Anweisung

Bedeutung

Eine Anweisung wird ausgeführt. Damit können beliebige Variablen und Attribute im Dialogablauf verändert werden.

Eingabe im grafischen Debugger

Die auszuführende Anweisung wird im Hauptfenster eingegeben. Danach muss die Schaltfläche "Eval" selektiert werden.

Syntax

```
eval <Statement>
```

Erklärung der Argumente

Dieser Befehl hat keine Argumente.

5.12 execute - Ausführung einer Datei

Bedeutung

Über diesen Befehl können abzuarbeitende Debugger-Befehle aus einer Datei eingelesen werden. Diese Befehle und Haltepunkte wurden in einer früheren Sitzung über den Befehl "state" gesichert.

Eingabe im grafischen Debugger

Die Eingabe im grafischen Debugger erfolgt über das Menü "File", Eintrag "execute". In dem dann erscheinenden Dialogfenster kann der Name der zu ladenden Datei eingegeben werden.

Syntax

```
execute <Datei-Name>
```

Erklärung der Argumente

Dieser Befehl hat keine Argumente.

5.13 finish - Beendigung des betrachteten Regelaufrufs

Bedeutung

Es wird ein flüchtiger Haltepunkt auf den Aufrufer der betrachteten Regel gesetzt und mit der Abarbeitung fortgefahren.

Eingabe im grafischen Debugger

Die Eingabe erfolgt durch Selektion der Schaltfläche "Finish" in der Symbolleiste.

Syntax

```
finish
```

Erklärung der Argumente

Dieser Befehl hat keine Argumente.

5.14 help - Ausgabe der vorhandenen Befehle

Bedeutung

Eine Liste der Debugger-Befehle wird ausgegeben.

Eingabe im grafischen Debugger

Die Hilfe wird über die `F1`-Taste (Windows und Motif) bzw. `Shift`+`F1` (Qt) aufgerufen.

Syntax

```
help
```

Erklärung der Argumente

Dieser Befehl hat keine Argumente.

5.15 list - Ausgabe einer Regel

Bedeutung

Es wird eine Regel ausgegeben.

Eingabe im grafischen Debugger

Dieser Befehl wird im grafischen Editor automatisch im Hauptfenster ausgeführt.

Syntax

```
list {<Regel>} {<Zeile>}
```

Erklärung der Argumente

Regel

Hier kann die auszugebende Regel angegeben werden. Wenn dieses Argument fehlt, wird die betrachtete Regel genommen.

Zeile

Hier kann eine Zeilennummer angegeben werden, deren Umgebung ausgegeben wird. Wenn dieses Argument fehlt, wird die betrachtete Zeilennummer genommen.

5.16 modules - Ausgabe der Liste der geladenen Module

Bedeutung

Die Liste der Module wird ausgegeben.

Eingabe im grafischen Debugger

Die Ausgabe der geladenen Module erfolgt durch Selektion der Schaltfläche "Modules".

Syntax

```
modules
```

Erklärung der Argumente

Dieser Befehl hat keine Argumente.

5.17 next - Abarbeitung bis zur nächsten Anweisung im betrachteten Regelaufruf

Bedeutung

Es wird ein flüchtiger Haltepunkt auf den betrachteten Regelaufruf gesetzt und die Abarbeitung wird dann fortgesetzt.

Eingabe im grafischen Debugger

Die Eingabe erfolgt durch Selektion der Schaltfläche "Next" in der Symbolleiste.

Syntax

```
next
```

Erklärung der Argumente

Dieser Befehl hat keine Argumente.

5.18 print - Auswertung eines Ausdrucks

Bedeutung

Ein Ausdruck wird ausgewertet und ausgegeben.

Eingabe im grafischen Debugger

Der Ausdruck wird im Hauptfenster eingegeben und anschließend die Schaltfläche *„Print“* selektiert.

Syntax

```
print <Ausdruck>
```

Erklärung der Argumente

Ausdruck

In diesem Parameter wird der auszuwertende Ausdruck angegeben.

Achtung

Wenn der Ausdruck, dessen Wert ausgegeben werden soll, außerhalb von Strings Leerzeichen enthält, müssen diese mit Rückstrichen markiert werden.

5.19 quit - Abbruch des Programmlaufs

Bedeutung

Der Debugger wird verlassen.

Eingabe im grafischen Debugger

Die Eingabe erfolgt durch die Selektion des "Exit" Eintrags im Menü "File".

Syntax

```
quit
```

Erklärung der Argumente

Dieser Befehl hat keine Argumente.

5.20 rules - Ausgabe einer Liste von Regeln

Bedeutung

Eine Liste von Regeln wird ausgegeben oder die Liste von Regeln wird neu aufgebaut.

Eingabe im grafischen Debugger

Die Ausgabe einer Regelliste erfolgt durch die Selektion des Menüs "Rules" Eintrag "Select". Danach erscheint ein Unterfenster, in dem die Regeln spezifiziert werden können, die aufgelistet werden sollen.

Syntax

```
rules {rebuild | <Pattern>}
```

Erklärung der Argumente

rebuild

Ist dieses Argument angegeben, wird die Liste der Regeln neu aufgebaut. Dies ist etwa nötig, wenn auf Regeln in einem Modul oder Dialog zugegriffen werden soll, die erst nach dem Start des ersten Dialogs geladen wurden. Nach dem Zuladen ist also ein "rules rebuild" nötig.

pattern

Ist dieses Argument angegeben, werden die Regeln ausgegeben, deren Namen dem angegebenen Muster entsprechen. Dabei werden im Muster die Zeichen Stern (*) und Fragezeichen (?) besonders interpretiert: Ein Stern steht für eine (gegebenenfalls leere) Folge beliebiger Zeichen und ein Fragezeichen für ein einzelnes beliebiges Zeichen.

Ist überhaupt kein Argument gegeben, hat dies dieselbe Wirkung wie die Angabe eines Musters *, es werden also alle Regeln ausgegeben.

5.21 stack - Ausgabe des Regel-Stacks

Bedeutung

Der Regel-Stack wird ausgegeben.

Eingabe im grafischen Debugger

Die Eingabe erfolgt durch Selektion der Schaltfläche "Stack" in der Symbolleiste. Danach wird der aktuelle Stack ausgegeben.

Syntax

```
stack
```

Erklärung der Argumente

Dieser Befehl hat keine Argumente.

5.22 state - Ausgabe des Systemzustands

Bedeutung

Haltepunkte und Überwachungsausdrücke werden in einer Form abgespeichert, die per `execute` wieder eingelesen werden kann. Damit können diese Zustände über mehrere Debugger-Sitzungen hinweg erhalten werden.

Eingabe im grafischen Debugger

Die Eingabe erfolgt über das Menü *„Files“* Eintrag *„State“*. In dem erscheinenden Unterfenster kann dann der Name der Datei eingegeben werden.

Syntax

```
state {<Datei-Name>}
```

Erklärung der Argumente

Datei-Name

Dieser Parameter ist der Name der Datei, in den die Befehle geschrieben werden. Wenn das Argument fehlt, dann werden die Befehle auf den Bildschirm geschrieben.

5.23 step - Führe einen Berechnungsschritt aus

Bedeutung

Ein Schritt wird ausgeführt, dann wird wieder angehalten.

Eingabe im grafischen Debugger

Die Eingabe erfolgt durch Selektion der Schaltfläche "Step" in der Symbolleiste.

Syntax

```
step
```

Erklärung der Argumente

Dieser Befehl hat keine Argumente.

5.24 stop - Unterbrechung an der nächsten ausgeführten Regel

Bedeutung

Ein Haltepunkt wird auf die nächste ausgeführte Regel des Dialogs gesetzt. Dieser Haltepunkt ist nur einmalig und dient dazu, einen laufenden Dialog zu unterbrechen.

Eingabe im grafischen Debugger

Die Eingabe erfolgt durch die Selektion der Schaltfläche "Stop" in der Symbolleiste.

Syntax

```
stop
```

Erklärung der Argumente

Dieser Befehl hat keine Argumente.

5.25 undisplay - Löschen eines Überwachungsausdrucks

Bedeutung

Die durch Nummern angegebenen Überwachungsausdrücke werden gelöscht.

Eingabe im grafischen Debugger

Die Eingabe erfolgt durch Selektion der Schaltfläche *„Undisplay“* im Unterfenster *„displayed expressions“*.

Syntax

```
undisplay [<Liste von Ausdrücken>]
```

Erklärung der Argumente

Liste von Ausdrücken

In diesem Parameter werden die Nummern der zu löschenden Überwachungsausdrücke angegeben.

5.26 up - Erhöhen der betrachteten Stackebene

Bedeutung

Die betrachtete Stackebene wird um eine Einheit erhöht. Die kleinste Stackebene entspricht dabei dem Ort der tiefsten Aufrufschachtelung.

Eingabe im grafischen Debugger

Die Eingabe erfolgt durch Selektion der Schaltfläche "Up" in der Symbolleiste.

Syntax

```
up
```

Erklärung der Argumente

Dieser Befehl hat keine Argumente.

Index

E

F

G

H

I

K

L

T

U

V

Z