Leitfaden für die SMILE-Programmierung

Jörg Rädler, dezentral gbr
und
Christoph Nytsch, FIRST

30. August 2001

1 Grundlagen

Dieses Dokument richtet sich an den intensiveren SMILE-Anwender, der auch selber Modelle erstellt oder abwandelt. Die Grundlagen von SMILE, der Programmiersprache **C** und deren Derivaten sollten vertraut sein!

1.1 Motivation dieses Dokumentes

1.1.1 Stabilität als Hauptziel

Die Erfahrung zeigt, daß bei komplexen Simulationsmodellen meist viel mehr Zeit für Fehlversuche, Fehlerbeseitigung und numerische Stabilisierungsmaßnahmen benötigt wird als zur Simulation selber. Die Erstellung von Grundfunktionen und Grundkomponenten, bei denen Stabilität Vorrang hat vor Rechenzeitüberlegungen, ergibt sich daraus zwingend. Diese Grundbausteine sollten nur aus wichtigen Gründen und gut dokumentiert geändert werden. Alle Änderungen müssen nachvollziehbar sein (Stichwort Versionsverwaltung) und sollten global erfolgen. Das heißt, daß Fehlermeldungen und Vorschläge zur Behebung an die Autoren geschickt und keine von Dritten geänderten konkurrierenden Versionen in Umlauf gebracht werden.

1.1.2 Austauschbarkeit durch Standards

Damit Modelle wiederverwendbar und flexibel sind, müssen sie sauber programmiert und dokumentiert sein. Konventionen zu Schreibweisen, Programmierstil, Namensgebung etc. sind unvermeidlich. Die vorgestellten Konventionen wurden von langjährigen SMILE-Anwendern erstellt und sollten als bindend für alle angesehen werden, die SMILEnicht nur für sich alleine benutzen!

1.1.3 Durchblick mit Versionsverwaltung

[...noch in Bearbeitung...sorry!...]

1.1.4 Einfache Handhabung durch globale Konfiguration

Das eben beschriebene Ziel, die Komponenten aus Gründen der Übersichtlichkeit und Vergleichbarkeit weitgehend als *Read-Only* zu behandeln, führt zu weiteren Maßnahmen, die gleichzeitig

die Benutzung vereinfachen. Viele Optionen, die mehrere Modelle betreffen, sind global konfigurierbar. Dazu gehören oft benötigte Initialisierungen, der Stream-Mechanismus für Fluidströme, die Aktivierung spezieller Sicherheitsabfragen und vieles andere mehr. Auch modellspezifische Einstellungen, wie etwa symbolische Konstanten zur Festlegung maximaler Feldgrößen können an zentraler Stelle festgelegt werden, ohne sich in die Tiefen der Dateistruktur der Bibliothek zu begeben.

1.2 Konventionen in diesem Handbuch

Klassennamen werden in *Italic* hervorgehoben. Namen von Modellgrößen, Symbolische Konstanten und alle anderen Codeschnipsel sind in `Typewriter` gesetzt.

2 Konventionen - Coding Standards

Manche Texteditoren nehmen sinnvolle Formatierungen des Codes selbst vor. Hier einige Hinweise:

- Der **EMACS** hat zwar noch keinen SMILE-Modus, bietet im C++- oder ObjC-Modus einige Unterstützung, was farbliche Hervorhebungen, Einrückungen etc. angeht.
- Für einen anderen weit verbreiteten Editor, den **NEDIT** gibt es einen SMILE-Modus in der Bibliothek.
- Die Entwicklungsumgebung **SourveNavigator** bietet in einer speziellen Version (siehe Bibliothek) Unterstützung für SMILE, leider sind hier noch kleinere Fehler vorhanden.

2.1 Sprache

Die Basissprache für alle Quelldateien (Modelldateien, C-Funktionen, Headerdateien etc.) ist **Englisch**! Das heißt, daß sowohl die Klassen- und Variablenbezeichnungen auf Englisch zu erfolgen haben, als auch erläuternde Kommentare in dieser Sprache zu verfassen sind. Die Dokumentation sollte wenn möglich in Englisch und Deutsch formuliert werden!

2.2 Zeilenumbrüche, Einrückungen und Klammerung

Zeilen werden nach spätestens 80 Zeichen umgebrochen! Sonst nimmt der jeweilige Betrachter oder Editor die Umbrüche selbst vor und das sieht ziemlich unübersichtlich aus! Die weitergeführte Zeile sollte soweit eingerückt werden, daß der logische Block erhalten bleibt (wie z.B. bei der fortgeführten Parameterliste der ersten Gleichung im Beispiel).

Logische Blöcke werden hierarchisch eingerückt! Die Einrückungstiefe sollte 2 (bis max. 4) Leerzeichen betragen und im ganzen Dokument beibehalten werden.

☞ **Beispiel:**

```
@eq discrete foo[0] (foo[1], foo[3], foo[5], bar[2],
                     bar[4], bla, fasel)
{
  if (foo == 1.0)
    return bar[4] * fasel;
  else
```

```
    return (foo[3] * foo[5] * bar[2] * bla * fasel);
}

@eq cont prod (fak1, fak2) { return fak1 * fak2; }
```

Auf Tabulatorzeichen im Code sollte verzichtet werden, da auch hier die Anzeige (Einrückungstiefe) vom jeweiligen Editor abhängt[1].

Weitere Formatierungsmerkmale bleiben dem Programmierer überlassen. So setzen manche Programmierer die öffnenden geschweiften Klammern nicht in eine neue Zeile oder setzen mehr oder weniger Lehrzeichen.

Mit runden Klammern, die dem Gruppieren von Ausdrücken dienen, sollte im Zweifelsfall nicht gespart werden. Sie können die Lesbarkeit des Codes durch genaues Anzeigen der Hierarchie von Operationen erhöhen, selbst wenn sie nicht zwingend notwendig sind. In komplexen Ausdrücken können ein paar zusätzliche Klammern Fehler bei späteren Erweiterungen vermeiden helfen.

2.3 Klassenbenennungen

Klassennamen haben normale Schreibweise (großer Anfangsbuchstabe, kleiner Rest). Worte (oder Wortteile) können ohne Space oder Unterstrich aneinander gehängt, Anhängsel (genaue Typbezeichnungen etc.) als letzter Teil durch Unterstrich abgeteilt werden. Eine Verfeinerung der Bezeichnung wird dem allgemeineren Namensteil nachgestellt. Wenn Namensteile zu lang werden oder keine Bedeutung mehr haben, kann verkürzt werden. Abstrakte Basisklassen, die nicht selbst funktionfähig sind und nur als Vorlage für andere Klassen dienen, erhalten als den Namensbestandteil `Base`. Klassen, die Eigenentwicklungen oder Bestandteil eines nicht allgemein verfügbaren oder sehr speziellen Modellpaketes (also einer Bibliothek) sind, sollten ein eindeutiges Kürzel dem Klassennamen mit Unterstrich voranstellen.

☞ **Beispiel:**
`HeatExchangerBase`
`HeatExchangerParallelFlow`
`HeatExchangerCounterFlow`
`HeatExchanger_AX500`
`HT_HeatExchanger_T20`

Die Klasse HeatExchangerBase stellt eine abstrakte Basisklasse dar, von der funktionsfähige Modelle für Gleich- und Gegenstrom-Wärmetauscher abgeleitet werden. Der `HeatExchanger_AX500` beschreibt ein konkretes Modell, hier sind die Eigenschaften durch die Typbezeichnung festgelegt und müssen nicht mehr als langer Klassenname beschrieben werden. Der `HT_HeatExchanger_T20` stammt aus dem HT-Paket (steht z.B. für High-Temperature), ist wahrscheinlich nicht kompatibel mit den anderen Modellen und grenzt sich durch das Kürzel von den Standardkomponenten ab.

[1] **EMACS**-Benutzer können beruhigt sein, hier wird die Eingabe eines Tab vom Editor in korrekte Formatierung umgesetzt, falls ein passender Modus aktiv ist.

2.4 Variablennamen, Einheiten und Attribute

2.4.1 Variablennamen

Diese Namenskonvention gilt für die Benennung von Variablen von SMILE-Modellen.

Es ist sinnvoll Modellvariablen innerhalb der SMILE-Modellbeschreibung nach einer verbindlichen Konvention zu benennen. Die konsequente Benutzung einer Namenskonvention trägt erheblich zum intiutiven Verständnis der Simulationsmodelle bei, gerade wenn der Autor und der Nutzer des Simulationsmodells nicht identisch sind.

Die Tabellen 2.1 bis 2.7 enthalten für die wichtigsten physikalischen Größen jeweils den Variablennamen und die dazugehörige SI-Einheit. Die in den Tabellen angeführten SI-Einheiten gelten als Empfehlung für die Stringbelegung des `unit`-Attributs bei der Variablendeklaration, es gibt im Einzelfall jedoch Gründe, von dieser Empfehlung abzuweichen.

Grundsätzlich sollte für den Variablennamen die angeführte Kurzform verwendet werden, manchmal ist es jedoch zur besseren Anschauung sinnvoll den Variablennamen in Langform zu verwenden, der bei vielen Variablennamen in Klammern hintern der Kurzform angeführt ist (z.B. `length` statt `l`).

Ein Variablename setzt sich einer Bezeichnung für eine Größe (z.B. `T` für die thermodynamische Temperatur) und einem oder mehrere angehängten Indizes (z.B. `outs` für außen) zusammen, welche die Größe genauer spezifizieren. Größe und Indizes werden mit einem Unterstrich verbunden und bilden zusammen den Variablennamen (z.B. `T_outs`). Mehrere Indizes werden mit mehreren Unterstrichen angehängt (z.B. die Lufttemperatur auf der Außenseite `T_air_outs`). Eine Ausnahme bilden zeitbezogene Größen, welche über der Variablen einen Punkt tragen, wie z.B. der Wärmestrom $\dot{Q}$. Um anzudeuten, daß es sich bei dem Punkt (dot) nicht um einen an den Variablennamen angehängten Index handelt, sondern daß er quasi über dem Variablennnamen steht, wird der Unterstrich weggelassen. Ein konvektiver Wärmestrom würde demnach mit `Qdot_cv` benannt werden.

Alle weiteren Variablennamen, die nicht in den Tabellen enthalten sind, und in einem SMILE-Modell deklariert werden, sollten klein geschrieben werden (z.B. `relation` oder `perimeter`), um sie besser von den großgeschriebenen Klassennamen unterscheiden zu können.

Dimensionslose Kennzahlen aus der Ähnlichkeitstheorie sollten wie üblich benannt werden (z.B. Re, Gr, Pr, Ar, ...).

Um auszudrücken, daß es sich um eine differentielle Größe handelt, läßt man den Variablennamen mit einem `d` beginnen. Ein differentielles Flächenelement wird z.B. mit `dA` bezeichnet.

Die Differenz einer Größe wird mit einem vorangestellten `delta` ausgedrückt, so z.B. eine Temperaturdifferenz als `deltaT`.

2.4.2 Einheiten

Manchmal kann eine Abweichung von den in den nachfolgenden Tabellen empfohlenen Einheiten notwendig werden. Gründe dafür können sein:

- Andere Konventionen: In bestimmten Branchen oder Themengebieten ist es noch üblich, andere als die oben genannten SI-Basiseinheiten zu wählen. Um die Umrechnung vor der Dateneingabe oder der Ergebnisauswertung zu vermeiden, kann hier natürlich eine gängige Einheit gewählt werden. Niemand wird die über die Lebensdauer umgesetzte Energiemenge eines Großkraftwerkes in J (Joule) angeben.

- Überlauf: Wenn die Zahlenwerte von Größen so groß oder so klein werden, daß die Grenzen der Darstellbarkeit[2] erreicht werden können, sollte natürlich eine andere Einheit gewählt werden.

- Stark unterschiedliche Größenordnungen: Es hat sich herausgestellt, daß Größen mit stark unterschiedlichen Größenordnungen (mehr als ca. 5 Zehnerpotenzen) innerhalb eines Gleichungssystems zu überhöhten Rechenzeiten und höherer Instabilität führen **können**. Gleichzeitig stellt sich hier die Frage nach einer sinnvollen globalen Toleranzangabe für den numerischen Löser. Eine genaue Erörterung des Problems würde an dieser Stelle zu weit führen. Es lohnt sich aber gerade bei komplexen Modellen, Versuche mit unterschiedlichen Einheiten durchzuführen. Diese sollten so gewählt werden, daß sich alle Größen in einer ähnlichen Größenordnung bewegen.

Es sollte aber beachtet werden, daß SMILE-Modelle so allgemeingültig und damit wiederverwendbar wie möglich gehalen werden. Gewisse Grundkomponenten können oft in Bereichen mit sehr unterschiedlichen Größenordnungen eingesetzt werden. Eine Anpassung auf eine spezielle Anwendung schränkt natürlich die Wiederverwendbarkeit stark ein.

Die Überprüfung der Einheitenkonsistenz bei im `connect`-Block verknüpften Variablen ist derzeit in SMILE über einen Stringvergleich realisiert. Daher sollten die mit dem `unit`-Attribut festgelegten Einheiten, nach einer einheitlichen Definition festgelegt werden:

Allgemein kann eine Einheit aus mehreren (Unter-)Einheiten zusammengesetzt sein. Stehen mehrere Einheiten im Zähler, so sind diese mit einem Multiplikationszeichen (`*`) voneinander zu trennen. Mehrere Einheiten im Nenner werden dagegen durch mehrere Divisionszeichen (`/`) dargestellt. Die Potenz einer Einheit wird mit dem Dachzeichen (`^`) beschrieben.

☞ **Beispiel:**

```
double lambda [doc:"heat conductivity", unit:"W/m^2/K"];
double F      [doc:"force", unit:"kg*m/s^2"];
```

[2] Alle Größen in SMILEwerden im datentyp `double` abgebildet.

2.4.3 Attribute von Größen

Die Attribute `doc` und `unit` sollten für jede Größe gesetzt werden. `min` und/oder `max` werden bei allen Größen gesetzt, wo eine Einschränkung der Modellgültigkeit vorliegt.

Die anderen Attribute beeinflussen die Behandlung einer Größe durch die Numerik, sie sollten natürlich nur bei bedarf gesetzt werden.

2.4.4 Empfehlungen für Größen

Die Tabellen 2.1 bis 2.8 stellen die die aus Erfahrung sinnvollen Konventionen für die Benennung und die Einheiten von Größen in SMILEdar.

Tabelle 2.1: SI-Basisgrößen

Größe	Variablenbezeichnung	Einheit
Länge	l (length)	m
Masse	m (mass)	kg
Simulationszeit	time[2]	s
elektrische Stromstärke	I	A
Temperatur (thermodynamische)	T (temperature)	K
Stoffmenge	n	mol
Lichtstärke	I	cd

[2]Die Variable `time` ist als Systemvariable bereits global vordefiniert.

Tabelle 2.2: Raum- und zeitbezogene Größen

Größe	Variablenbezeichnung	Einheit
Breite	w (width)	m
Beschleunigung	a (accelaration)	m/s^2
Durchmesser	d (diameter)	m
Dicke	t (thickness)	m
Fläche	A (area)	m^2
Frequenz	f (frequency)	1/s
Geschwindigkeit	v (velocity)	m/s
Höhe	h (height)	m
Ortskoordinaten	x, y, z	m
Periodendauer	period	s
Radius	r (radius)	m
Volumen	V (volume)	m^3
Wellenlänge	lambda	m
Winkel (Ebene)	alpha, beta, gamma, theta, phi	rad bzw. grad
Winkel (Raum)	omega	sr
Winkelgeschwindigkeit	omega	1/s

Tabelle 2.3: Thermodynamische Größen

Größe	Variablenbezeichnung	Einheit
Diffusionskoeffizient	D	m^2/s
Enthalpie	H	J
Enthalpiestrom	Hdot	J/s bzw. W
spezifische Enthalpie	h	J/kg
Entropie	S	J/K
Entropiestrom	Sdot	J/K/s
spezifische Entropie	s	J/kg/K
Exergie	E	J
Exergiestrom	Edot	J/s bzw. W
freie Energie	F	J
freie Enthalpie	G	J
Innere Energie	U	J
Massenstrom	mdot	kg/s
Stoffübergangskoeffizient	beta	kg/m^2/s
Stoffmengenkonzentration	c	mol/m^3
Temperatur (Celsius)	t	C
Temperaturleitfähigkeit	a	m^2/s
Volumenausdehnungskoeffizient	beta	1/K
Wärmedurchgangskoeffizient	k	W/m^2/K
Wärmekapazität	C	J/K
spezifische Wärmekapazität	c	J/K/kg
Wärmeleitfähigkeit	lambda	W/m^2/K
Wärmemenge	Q	W/m^2/K
Wärmestrom	Qdot	J/s bzw. W
spezifischer Wärmestrom	qdot	W/m^2
Wärmeübergangskoeffizient	alpha	W/m^2/K
thermischer Wirkungsgrad	eta_th	-

Tabelle 2.4: Mechanische Größen

Größe	Variablenbezeichnung	Einheit
Arbeit	W (work)	J
Dichte	rho (density)	kg/m^3
Drehimpuls	L	kg*m^2/s
Drehmoment	M	Nm
Drehzahl	n	1/s
Druck	p (pressure)	Pa
Elastizitätsmodul	E	N/m^2
Energie	E (energy)	J
Kraft	F (force)	kg*m/s^2
Kompressibilität	kappa	1/Pa
Impuls	p (momentum)	kg*m/s
mechanische Leistung	P_mech	J/s bzw. W
Masse	m (mass)	kg
Oberflächenspannung	sigma	N/m
Trägheitsmoment	J	kg*m^2
Volumenstrom	Vdot	m^3/s
Viskosität (dynamische)	eta	kg/m/s
Viskosität (kinematische)	ny	m^2/s
mechanischer Wirkungsgrad	eta_mech	-
Widerstandsbeiwert	zeta	-

Tabelle 2.5: Elektrotechnische Größen

Größe	Variablenbezeichnung	Einheit
elektrisches Dipolmoment	p	C m
elektrische Feldstärke	E	V/m
elektrische Kapazität	C	F bzw. C/V
elektrische Leistung	P_el	W
elektrische Spannung	U	V
elektrischer Widerstand	R (resistance)	Ohm bzw. V/A
elektrischer Wirkungsgrad	eta_el	-
Ladung	Q	C
magnetische Feldstärke	H	A/m
magnetische Induktion	B	T

Tabelle 2.6: Optische, strahlungs- und lichttechnische Größen

Größe	Variablenbezeichnung	Einheit
Absorptionsgrad	alpha	-
Brechzahl	n	-
Emissionsgrad	epsilon	-
Reflektionsgrad	rho	-
Solarstrahlung	Gdot	W/m^2
Transmissionsgrad	tau	-

Tabelle 2.7: sonstige Größen

Größe	Variablenbezeichnung	Einheit
Anzahl	n	-
Luftfeuchte (absolut)	x	kg/kg
Luftfeuchte (relativ)	phi	kg/kg

Tabelle 2.8: Indizes für Variablennamen

Kennzeichnung	Index
hinein	in
hinaus	out
innen	ins
aussen	outs
kurzwellig	sw
langwellig	lw
konduktiv	cd
konvektiv	cv
laminar	la
turbulent	tu
Umgebung	env
Verlust	loss
gesamt	total
minimal	min
maximal	max
oben	top
unten	down
kumuliert	cum

2.5 Kommentare und Dokumentation

2.6 Kommentare im Quellcode

Die einzeiligen Kommentare der Form

```
// dies ist
// ein Kommentar
```

sollten denen der Form

```
/* dies ist auch
   ein Kommentar */
```

vorgezogen werden. Die zweite Schreibweise ist unübersichtlicher und fehleranfälliger bei geschachtelten Kommentaren. Sie sollte nur für mehrzeilige Erläuterungen (wie z.B. den Header) genutzt werden, oder zum zeitweiligen Deaktivieren von Code, etwa zu Testzwecken. Gute Editoren (wie z.B. **EMACS**) bieten Funktionen zum komfortablen Setzen und Entfernen von Kommentaren, auch für größere Codestücke.

2.7 Headerangaben im Quellcode

[...noch in Bearbeitung...sorry!...]

2.8 Modelldokumentation

Dokumente zur Modelldokumentation werden grundsätzlich in Adobes PDF zur Verfügung gestellt[3]. PDF wird auf allen gängigen Plattformen gut unterstützt und bietet Möglichkeiten zur Vernetzung von Dokumenten sowie eine sehr gute Darstellungsqualität.

Es wird empfohlen, Dokumentationen mit **PDFLaTeX** zu erstellen. Die Alternative mittels LaTeX, dvips und ps2pdf ist nicht zu empfehlen, da die Qualität durch die Rasterung dann erbärmlich wird!. Es gibt allerdings im Netz andere Werkzeuge, um Postscript in PDF ohne Qualitätsverluste zu übersetzen.

[...noch in Bearbeitung...sorry!...]

[3] Was manchen auch unter dem etwas unpassenden Namen Acrobat-Format bekannt sein dürfte.

3 SmileDefines - grundlegende Einstellungen

Die Datei `SmileDefines.h` ist die zentrale Steuerdatei für SMILE-Modelle. Sie enthält grundlegende Einstellungen, definiert Macros und Funktionen von vielfältigem Nutzen und sollte in jede Modelldatei eingebunden werden.

Im obersten Bereich der Datei können die zu nutzenden Eigenschaften durch Setzen von `#define XYZ` ausgewählt werden. Folgende grundlegenden Eigenschaften sind wählbar:

- `USE_COMMON_VALUES`: Dieser Schalter definiert verschiedene Werte, die von allen Modellen gemeinsam genutzt werden sollen.
- `USE_FUNCTIONS`: Dieser Schalter stellt die Definitionen von allgemeinen Hilfsfunktionen zur Verfügung.
- `USE_MACROS`: Dieser Schalter stellt die Definitionen von allgemeinen Hilfsmacros (funktionsähnlichen Konstrukten) zur Verfügung.
- `USE_FSTREAM_DEFINES`: Dieser Schalter stellt gemeinsame Definitionen und Macros für Fluid-Streams zur Verfügung.
- `PARANOID`: Dieser Schalter bewirkt in manchen Hilfsfunktionen und Modellen, daß besondere Fehlerabfragen aktiviert werden.
- `LOCAL_DEFINES`: Wird `LOCAL_DEFINES` mit dem Namen einer Headerdatei belegt, stehen die dortigen Einstellungen allen Modellen zur Verfügung. Auf diese Art können modellspezifische Einstellungen überschrieben werden, ohne die Modelldateien zu ändern.

3.1 Symbolische Konstanten

Das Setzen von `USE_COMMON_VALUES` bewirkt die Definition einiger Konstanten, die in verschiedenen Modellen genutzt werden. Dazu gehören Initialisierungswerte für Starttemperaturen, Einstellungen für die Berechnung von Fluideigenschaften und minimale Massenströme, die noch nicht als “numerisches Rauschen” angesehen werden. Diese Werte werden dann in allen Modellen konsistent eingesetzt, können aber natürlich durch explizite Initialisierung geändert werden. Momentan sind für diesen Zweck die in Tabelle 3.1 aufgeführten symbolischen Konstanten gesetzt.

Name	Bedeutung	Wert
MDOT_MIN	minimaler Massenstrom, der als ungleich Null angesehen wird	1.0e-5 kg/s
T_INIT	Starttemperatur für dynamische Modelle oder Anfangswert für implizite Temperaturgleichungen	293.15 K
CALC_PROP	Schalter für konstante (0) oder temperaturabhängige Fluideigenschaften (1)	0 (konstant)
T_PROP	Temperatur für konstante Fluideigenschaften	293.15 K

Tabelle 3.1: Symbolische Konstanten

3.2 Streams für Fluidströme

In vielen Modellen werden Eigenschaften von Fluidströmen zu Feldvariablen zusammengefaßt, die das Verknüpfen der Ein- und Ausgangsströme erleichtern. Der Aufbau dieser Streams ist weitgehend variabel. Die Gesamtgröße des Feldes, die genutzte Größe, die Positionen der einzelnen Größen innerhalb des Feldes und die Initialisierung der nicht genutzen Feldbausteine können über symbolische Konstanten festgelegt werden. Damit genügt oft die Änderung einer einzelnen Datei, um diese Streams voll kompatibel zu machen mit den Streams der Komponenten anderer Bibliotheken. Voreinstellung ist Kompatibilität mit dem an der TU-Berlin genutzten Schema (Positionen 0 bis 2 sind Fluid-Nummer, Massenstrom und Temperatur, Positionen 3 und 4 die oft ungenutzten Größen Druck und Feuchte).

Die vorgestellte Stream-Funktionalität wird genutzt, wenn USE_FSTREAM_DEFINES gesetzt ist. Die in Tabelle 3.2 genannten Konstanten und Macros stehen derzeit zur Verfügung.

Name	Bedeutung	Wert
FSTR_MAX	Größe des Stream-Feldes	5
FSTR_USED	Genutzte Größe des Stream-Feldes	3
FSTR_FID	Position der Fluid-Nummer im Stream	0
FSTR_MDOT	Position des Massenstromes im Stream	1
FSTR_T	Position der Temperatur im Stream	2
FSTR_P	Position des Druckes im Stream	3
FSTR_X	Position der Feuchte im Stream	4
FSTR_DEF	Default-Wert für ungenutzte Positionen	1.0
FSTR_INIT()	Macro zur Initialisierung ungenutzter Positionen	—

Tabelle 3.2: Einstellungen für Fluid-Streams

Es kann bei Anschluß an Komponenten mit anderen Stream-Konventionen notwendig sein, das Macro FSTR_INIT() umzudefinieren, mit

```
#define FSTR_INIT(X) { X[2] = 0.0; X[5] = 1.0; }
```

werden beispielsweise nur die Positionen 2 und 5 initialisert. Ein Macro zur Verknüpfung von

Streambestandteilen (`connect` - wenn z.B. der Druck vom Eingang zum Ausgang weitergereicht werden soll) existiert noch nicht!

☞ **Beispiel:**
Ein Stream wird mit `FSTR_MAX` als Feldgröße im Interface-Block festgelegt:

```
double in[FSTR_MAX] [doc: "input stream", unit: "*"];
```

im Init-Block eventuell ungenutzte Elemente über ein Macro initialisiert:

```
FSTR_INIT(in)
```

und im connect-Block mit den anderen Größen des Modells verknüpft:

```
@connect {
  in[FSTR_FID]  = fluid_id;
  in[FSTR_T]    = T_in;
  in[FSTR_MDOT] = mdot;
}
```

Das Verknüpfen der Fluidströme unterschiedlicher Modelle sieht dann zum Beispiel so aus:

```
@connect {
  pump.out[i]   = heater.in[i];
  heater.out[i] = tube1.in[i];
  tube1.out[i]  = house.in[i];
  house.out[i]  = tube2.in[i];
  tube2.out[i]  = pump.in[i];
}
```

Die Größe des Feldes und die Positionen der Feldelemente spielen keine Rolle mehr, sie wurden global gesetzt. Zudem sind die symbolischen Konstanten als Positionsbezeichner anschaulicher als reine Zahlen[1].

3.3 Macros - Funktionsähnliche Definitionen

Wird `USE_MACROS` gesetzt, stehen Macros für oft gebrauchte Operationen zur Verfügung. Für diese Macros stehen aber meist sauberer implementierte Funktionen oder Modellklassen mit ähnlicher Funktionalität zur Verfügung.

Achtung: Macros haben gegenüber Funktionen den Vorteil, daß oft weniger Datenübergaben, -konvertierungen etc. nötig sind und die Ausführung dadurch meist schneller ist.
Dadurch können aber auch Fehler entstehen:

- Macros testen (im Gegensatz zu sauber implementierten und genutzten Funktionen) nicht den Datentyp von Argumenten und erkennen keine Fehler, die durch Operationen mit inkompatiblen Typen entstehen.
- In Macros können Argumente mehrmals ausgewertet werden, was zu unerwünschten Seiteneffekten führen kann.
  ☞ **Beispiel:**

[1] In der Experimentshell stehen diese Defines nicht zur Verfügung, hier muß für Initialisierung und Observer die Nummer eingesetzt oder —falls vorhanden— die entsprechende Klartextvariable (wie z.B. `mdot`) genutzt werden.

Wird das Macro
```
#define POS(X) {((X>=0.0)?X:0.0)}
```
im Programmcode z.B. in der Form
```
y = POS(i++);
```
aufgerufen, wird `i` u.U. zweifach inkrementiert! Die hier vorgestellten Macros sind i.d.R. nicht seiteneffektfrei implementiert!

3.3.1 INTCOMP() - vergleiche diskrete Werte

`Y = INTCOMP(X0, X1)`		
Symbol	Typ	Bedeutung
`Y`	int	Rückgabewert
`X0`	int/long/float/double	Argument
`X1`	int/long/float/double	Vergleichswert

`INTCOMP()` führt einen Wertevergleich unter Berücksichtigung eines Toleranzbereiches durch. Es ist als Ersatz für Formulierungen wie
```
if ((int)z == 1) {...}  oder
if (y == 2.0) {...}
```
gedacht. Diese Beispiele können als
`if INTCOMP(z, 1) {...}` oder `if INTCOMP(z, 1.0) {...}` bzw.
`if INTCOMP(y, 2) {...}` oder `if INTCOMP(z, 2.0) {...}`
geschrieben werden. Das zweite Argument kann als ganze oder Fließkommazahl angegeben werden. Der Vergleich funktioniert auch bei kleinen Abweichungen, voreingestellt ist eine Toleranz von $+/-0.5$. Dieser Wert kann durch die symbolische Konstante `INTCOMP_TOL` eingestellt werden.

3.3.2 MIXVAL() - stetiges Mischen zweier Werte

`Y = MIXVAL(A, X0, X1)`		
Symbol	Typ	Bedeutung
`Y`	double	Rückgabewert
`A`	double	Mischungsverhältnis
`X0`	double	Wert für A = 1
`X1`	double	Wert für A = 0

`MIXVAL()` berechnet den Rückgabewert über die Mischungsfunktion $y = a \cdot x_0 + (1.0 - a) \cdot x_1$. Der Haupteinsatzbereich ist jedoch bei Formulierungen wie:
```
if (a == 1.0) return x_0;
if (a == 0.0) return x_1;
```
die vereinfachend als
```
return MIXVAL(a, x_0, x_1)};
```
geschrieben werden können. Ein großer Vorteil dieser Formulierung liegt darin, daß ein Ergebnis definiert und stetig ist auch bei unerwarteten Werten von `a`.

Die ursprüngliche Formulierung (`if`-Abfrage) erzwingt die Entscheidung, ob die Gleichung diskret oder kontinuierlich definiert wird. Ersteres gibt Probleme[2], falls `x_0` und/oder `x_1` kontinuierlich bestimmt werden, also weder konstant noch diskret springend sind. Der zweite Fall (kontinuierlich) funktioniert nur, wenn `a` konstant ist oder durch eine diskrete Gleichung genau geschaltet wird. Beide Einschränkungen können durch den Einsatz von `MIXVAL()` fallengelassen werden! Allerdings sollte die Gleichung hier kontinuierliche definiert werden, sie funktioniert mit diskreten und kontinuierlichen Argumenten!

3.3.3 MIN() und MAX()

`Y = MIN(X0, X1)` `Y = MAX(X0, X1)`		
Symbol	Typ	Bedeutung
`Y`	double	Rückgabewert
`X0`	double	erstes Argument
`X1`	double	zweites Argument

`MIN()` gibt den kleineren, `MAX()` den größeren Wert der beiden Argumente wieder.

3.3.4 CUTRANGE() - Wertebereichsbegrenzung

`Y = CUTRANGE(X, X_LOW, X_HIGH)`		
Symbol	Typ	Bedeutung
`Y`	double	Rückgabewert
`X`	double	Argument
`X_LOW`	double	unterer Grenzwert
`X_HIGH`	double	oberer Grenzwert

`CUTRANGE()` ergibt den Wert von `X`, solange `X_LOW <= X <= X_HIGH` ist, sonst den entsprechenden Grenzwert. Der Verlauf von `Y` knickt an den Grenzwerten, der Einsatz in kontinuierlichen Gleichungen sollte also genau überlegt werden! Siehe auch die Funktion `softcut()`.

3.3.5 SAFEDIV() - keine Division durch Null

`Y = SAFEDIV(Z, N, Y_ZERO)`		
Symbol	Typ	Bedeutung
`Y`	double	Rückgabewert
`Z`	double	Zähler
`N`	double	Nenner
`Y_ZERO`	double	Rückgabewert für `N = 0`

[2]Der Löser wird die kleinste zulässige Schrittweite wählen!

`SAFEDIV()` vermeidet zwar die nicht erlaubte Division durch Null, die Funktion ist jedoch keineswegs stetig!

3.4 SmileFunctions - oft gebrauchte Hilfsfunktionen

Durch Aktivieren des Schalters `USE_FUNCTIONS` in der Datei `SmileDefines.h` oder explizite Einbindung von `SmileFunctions.h` in der entsprechenden Modelldatei über

```
#include "SmileFunctions.h"
```

oder[3]

```
#include <SmileFunctions.h>
```

stehen weitere Hilfsfunktionen zur Verfügung. In beiden Fällen muß im Makefile die Datei `SmileFunctions.c` unter `USER_C_SOURCES` angegeben werden.

Wer in Modellgleichungen die Funktionen `fabs()`, `pow()`, `sqrt()`, `tanh()` benutzt oder eine Division vornimmt, deren Nenner Null oder sehr klein werden kann, sollte unbedingt den Abschnitt über SmileFunctions lesen!

3.5 Bedingte Übersetzung von Programmteilen

Der Schalter `PARANOID` bewirkt, daß bestimmte Sicherheitsabfragen in manchen Modellen aktiviert werden. Dies kann hilfreich bei der Fehlersuche sein, führt aber auch zu längeren Übersetzungs- und Laufzeiten sowie teilweise unsinnigen und eventuell unüberschaubar vielen Meldungen.

3.6 Zentrale Einstellungen setzen

Viele Modelle benötigen modellspezifische symbolische Konstanten, z.B. zur Festlegung maximaler Feldgrößen. Deren Definitionen sind oft auf diverse Modell- oder Headerdateien verteilt. Alle dieser Konstanten sollten im Modell so gesetzt werden, daß vorher gesetzte Werte nicht überschrieben werden! Eine entsprechende Voreinstellung in der Modelldatei oder einer modellspezifischen Headerdatei kann z.B. so aussehen:

```
#ifndef LOGIC_N_INPUTS
  #define LOGIC_N_INPUTS 8
#endif
```

Dies macht es möglich, von den Standards abweichende Einstellungen in einer einzelnen Datei vorzunehmen, ohne sämtliche Headerdateien zu ändern. Ziel dieser Maßnahme ist, daß niemand zur Anpassung von Feldgrößen u.Ä. in den Modelldateien irgendetwas ändern muß, diese sollten als *Read-Only* angesehen werden! Lediglich in `SmileDefines.h` und eventuell eigenen Konfigurationsdateien sollen Änderungen nötig sein.

[3] Im ersten Fall sollte sich die Datei im Modellpfad befinden, im zweiten Fall in einem Systempfad, z.B. im SMILE-Includeverzeichnis.

Symbolische Konstanten mit modellspezifischer Bedeutung sollten in ihrem Namen eindeutig sein und nicht mit denen anderer Modelle in Konflikt stehen. Dies erreicht man am besten, indem dem eigentlichen Namen ein aus dem Namen des Modells, der Modelldatei oder einer Bibliothek gebildeter Prefix vorangestellt wird. Im obigen Beispiel ist dies `LOGIC`, eine Größe `N_INPUTS` wird mit hoher Wahrscheinlichkeit auch in anderen Modellen eine Bedeutung haben.

Setzt man `LOCAL_DEFINES` in der Datei `SmileDefines.h` auf einen Dateinamen, wird diese Datei eingelesen und die darin enthaltenen Werte gesetzt.[4]

☞ **Beispiel:**
Mit der Zeile
```
#define LOCAL_DEFINES "MeineWerte.h"
```
in der Datei `SmileDefines.h` und den Zeilen
```
#define LOGIC_N_INPUTS 4 // Zahl der Eingänge in MultiAND/OR
#define TUBE_N_ELEM    5 // Zahl der Rohrelemente von Tube
#define HEX_N_ELEM    10 // Zahl der Elemente in HeatExchanger
```
in der Datei `MeineWerte.h` werden diese Voreinstellungen für maximale Feldgrößen verschiedener Modelle in einer Datei zentral geändert.

Es ist natürlich auch möglich, hier eigene Macros zu definieren.

Anmerkung: Es ist nicht möglich, für einzelne Instanzen einer Klasse unterschiedliche Feldgrößen zu setzen. Durch Einführen einer Modellgröße, die die Zahl der aktuell genutzten Elemente angibt, kann ein solches Verhalten allerdings emuliert werden. Dies ist jedoch mit erhöhtem Verbrauch an Arbeitsspeicher und Rechenzeit verbunden.

Und weil es so komplex ist, noch ein Beispiel: Bei einem diskretisiertes Rohrmodell ist die Zahl der zu nutzenden Einzelelemente wählbar. Am Anfang der Modelldatei, die das Rohr enthält, steht folgendes:
```
#include "SmileDefines.h"
#define SMILE_HEADER
  #ifndef TUBE_N_ELEMENTS
    #define TUBE_N_ELEMENTS 6
  #endif
#undef SMILE_HEADER
```
Damit hat der Programmierer festgelegt, daß das Rohr in sechs Elemente aufgespalten wird, falls der Anwender keine Angabe dazu macht. Wenn für eine Simulation von diesem Wert abgewichen werden soll, muß der Anwender eine Datei erstellen, die z.B. `MeineEinstellungen.h` heißt. In dieser Datei sollte dann mindestens eine Zeile der Form:
```
#define TUBE_N_ELEMENTS 4
```
stehen. Damit diese Einstellung auch beim Modell ankommt, wird in `SmileDefines.h` der Wert von `LOCAL_DEFINES` auf den Namen der Datei gesetzt:
```
#define LOCAL_DEFINES "MeineEinstellungen.h"
```
Es würde theoretisch auch funktionieren, die `define`-Anweisung in der Modelldatei zu ändern oder eine entsprechende Anweisung z.B. in `SmileDefines.h` zu setzen. Dies sollte aber keinesfalls getan werden, da auf diese Art die Änderungen über viele Dateien verteilt werden und die Nachvollziehbarkeit der Änderungen und Austauschbarkeit der Modelldateien nicht mehr gegeben ist!

[4] Setzt man den Wert auf etwas anderes als den Namen einer gültigen, lesbaren Datei, gibt's einen Fehler!

4 SmileFunctions - wichtige Hilfsfunktionen

In der Datei `SmileFunctions.c` sind Funktionen definiert, die in Modellgleichungen genutzt werden können. Dazu muß entweder die Datei `SmileDefines.h` in die Modelldatei über eine `#include`-Anweisung eingebunden werden (und dort der Schalter `USE_FUNCTIONS` gesetzt sein), oder die Headerdatei `SmileFunctions.h` muss direkt eingebunden werden.

Hier werden drei Klassen von Funktionen definiert: allgemeine Funktionen, approximierende Funktionen und solche, die einen zulässigen Definitionsbereich erweitern.

4.1 Approximierende Funktionen zur Glättung

4.1.1 softcut() - Knickfreie Wertebereichsbeschränkung

`y = softcut(x, x_low, x_high, r)`
`y = softcut_upper(x, x_high, r)`
`y = softcut_lower(x, x_low, r)`		

Symbol	Typ	Bedeutung
`y`	double	Rückgabewert
`x`	double	Argument
`r`	double	Glättungsradius
`x_high`	double	oberer Grenzwert
`x_low`	double	unterer Grenzwert

Alle `softcut*()`-Funktionen geben prinzipiell die Funktion $y = x$ wieder. Es werden aber aber untere und obere Limits angegeben werden, die y keinesfalls überschreitet. In der Nähe dieser Limits wird die Funktion durch einen Kreisabschnitt mit wählbarem Radius approximiert[1]. Die Funktion selber und ihre ersten Ableitungen sind dabei stetig! Eine Abweichung durch die Approximation tritt nur in einem meist kleinen, vom gewählten Radius abhängigen Bereich auf!

☞ **Beispiel:**

```
a = softcut(x, 0.0, 1.0, 0.05);   // 0 <= a <= 1 mit r = 0.05}
b = softcut_upper(x, 100.0, 1.0); // b <= 100 mit r = 1.0}
c = softcut_lower(x, -PI, 0.5);   // c >= - PI mit  r =  0.5}
```

[1] Auf eine genaue Funktionsbeschreibung wird hier verzichtet, Interessierte lesen den Quelltext.

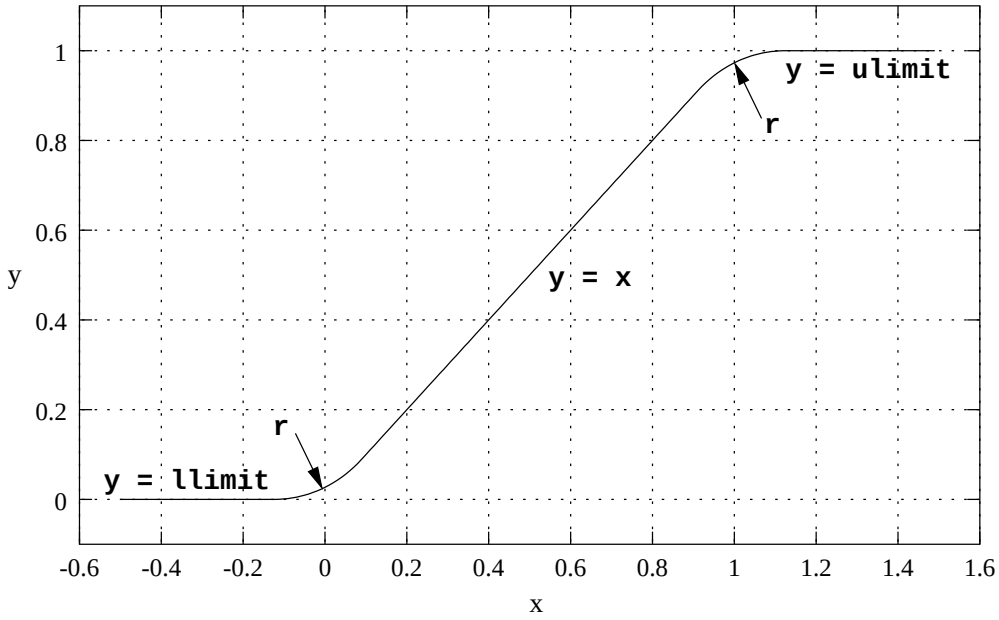

Abbildung 4.1: Die `softcut()`-Funktion

Anmerkung: Ähnliche Funktionen stehen als `smabove()` und `smbelow()` bereits in Smile zur Verfügung. Die obigen Varianten besitzt allerdings den Vorteil, dass Abweichungen vom linearen Verlauf nur in einem definierten Bereich auftreten und damit die Grenzwerte nicht erst im Unendlichen erreicht werden.

4.1.2 softswitch() - stetiger Schalterersatz

`y = softswitch(x, x_switch, y0, y1, f)`		
Symbol	Typ	Bedeutung
`y`	double	Rückgabewert
`x`	double	Argument
`x_switch`	double	Umschaltpunkt
`y0`	double	Rückgabewert für `x < x_switch`
`y1`	double	Rückgabewert für `x > x_switch`
`f`	double	Glättungsfaktor

Schalten mit `discrete`-Gleichungen kann nicht nur ineffizient sein, sondern auch zum Simulationsabbruch führen, wenn nach dem Schaltvorgang keine konsistenten Werte gefunden werden können. Ein schneller, aber stetiger Übergang zwischen zwei Zuständen kann hier Abhilfe schaffen.

Die Funktion `softswitch()` benutzt die `tanh()`-Funktion, um diesen Übergang zu berechnen. Das bedeutet, daß theoretisch nur in unendlichem Abstand vom Umschaltpunkt der Rückgabewert mit einem der beiden Zustände übereinstimmt. Durch einen kleinen Wert für den

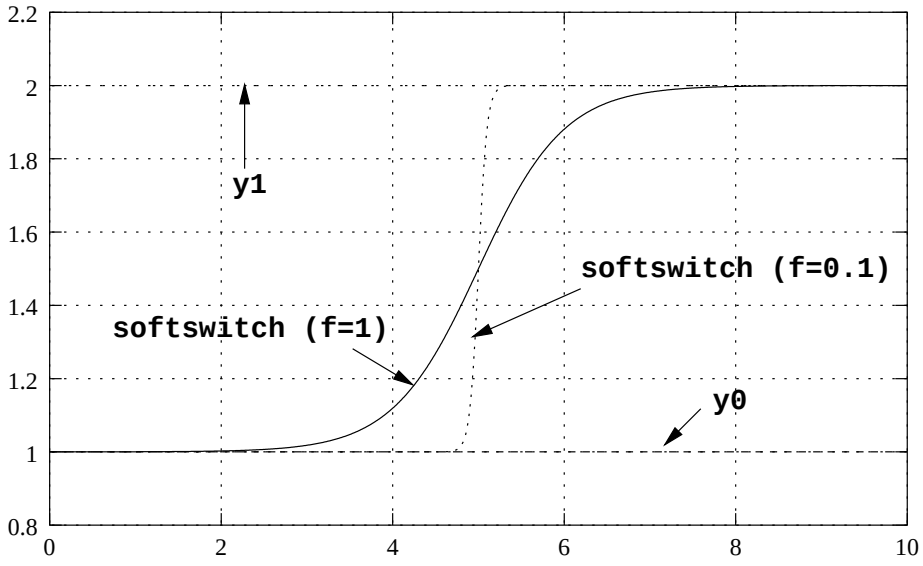

Abbildung 4.2: Die `softswitch()`-Funktion

Glättungsfaktor `f` kann das Verhalten aber nahe an das eines Schalters gebracht werden, so daß die tolerierbare Abweichung für bestimmte Anwendungsfälle angepaßt werden kann (siehe Bild).

4.1.3 softfabs() - Knickfreie Betragsbestimmung

`y = softfabs(x, delta_y)`		
Symbol	Typ	Bedeutung
`y`	double	Rückgabewert
`x`	double	Argument
`delta_y`	double	max. erlaubte Differenz zu `fabs()`

Die `softfabs()`-Funktion stellt eine knickfreie Approximation der `fabs()`-Funktion dar. Sie ist implementiert als $y = x \cdot \tanh(x \cdot \frac{0.28}{\Delta y})$. Durch den Faktor 0.28 kann man mit Δy direkt die maximale Abweichung der Funktion von der ursprünglichen `fabs()`-Funktion festlegen. Geringe Werte von Δy bedeuten geringere Abweichungen, aber auch ein knick-ähnlicheres Verhalten.

☞ **Beispiel:**

```
y = softfabs(x, 0.01);   // max. Abweichung 0.01 von fabs()
```

Anmerkung: Eine ähnliche Funktion steht als `smabs()` bereits in Smile zur Verfügung. Die obige Variante besitzt allerdings den Vorteil, dass sie bei $x = 0$ auch wirklich den Wert 0 liefert und der zu tolerierende Fehler explizit abgegeben werden kann.

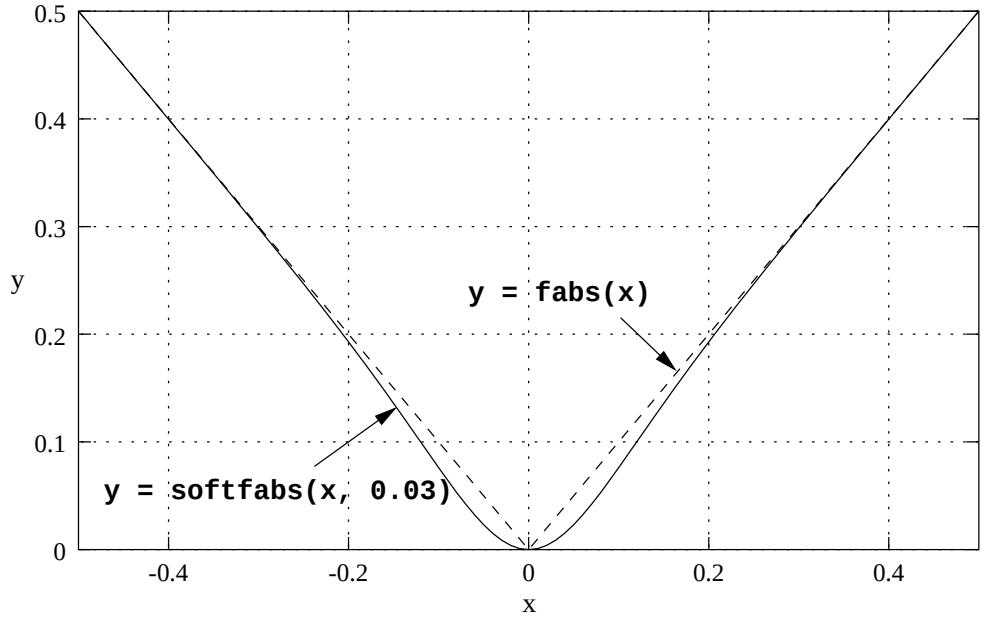

Abbildung 4.3: Die `softfabs()`-Funktion

4.2 Funktionen zur Erweiterung des Definitionsbereiches

4.2.1 softpow() und safepow()

`y = softpow(x, e, r)` `y = safepow(x, e)`		
Symbol	Typ	Bedeutung
`y`	double	Rückgabewert
`x`	double	Argument
`e`	double	Exponent
`r`	double	Glättungsradius bei `x = 0`

Für Argumente kleiner als Null liefert die `pow()`-Funktion `NaN` (Not a Number). `safepow()` verlängert die Funktion stetig für $x < 0$, indem es den Verlauf von $x > 0$ um die Gerade $y = -x$ spiegelt. `softpow()` approximiert für sehr kleine Argumente und negative Argumente einen sanften Übergang auf einen Rückgabewert von 0. Dabei werden die Argumente mit der `softcut()`-Funktion (und dem Radius r) auf $x >= 0$ begrenzt.

☞ **Beispiel:**

```
y = safepow(x, 1.5);         // pow(x, 1.5) mit Umklappen bei x<=0}
y = softpow(x, 0.66, 0.01); // pow(x, 0.66) mit softcut-Radius 0.01}
```

Anmerkung: `safepow()` kann negative Werte zurückgeben! Da dies niemand erwartet, können bei einem einfachen Austausch von `pow()` andere Probleme auftreten! `softpow()` tut dies nicht,

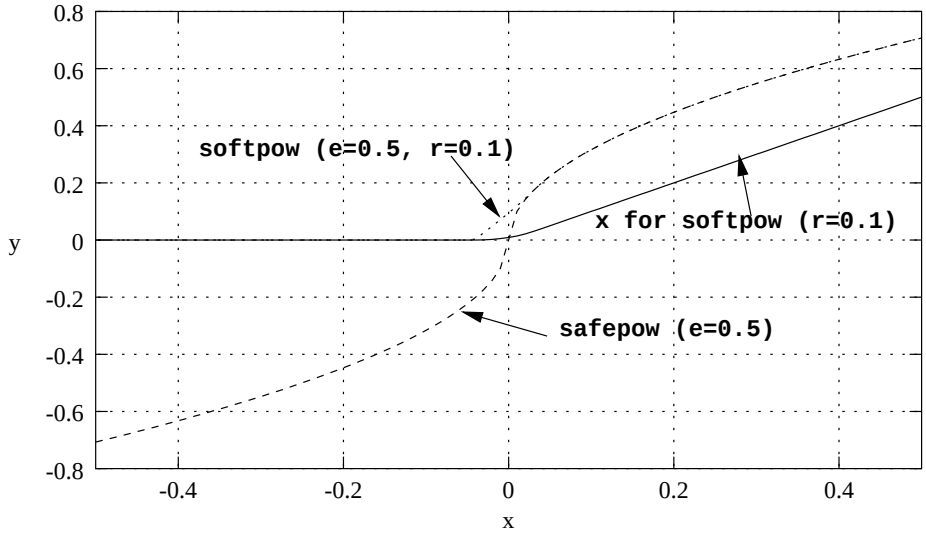

Abbildung 4.4: Die `softpow()`- und die `safepow()`-Funktion

diese Funktion ist allerdings für kleine Argumente fehlerbehaftet und kann bei kleinen Glättungsradien numerische Probleme durch den knickähnlichen Verlauf verursachen!

4.2.2 softsqrt() und safesqrt()

`y = softsqrt(x, r)` `y = safesqrt(x)`		
Symbol	Typ	Bedeutung
`y`	double	Rückgabewert
`x`	double	Argument
`r`	double	Glättungsradius bei `x = 0`

Diese Funktionen sind analog zu `softpow()` und `safepow()`, es gilt das dort gesagte!

4.2.3 safetanh() - tanh() auch für große Argumente

`y = safetanh(x)`		
Symbol	Typ	Bedeutung
`y`	double	Rückgabewert
`x`	double	Argument

Auf manchen Systemen[2] scheint die `tanh()`-Funktion für betragsmäßig große Argumente `NaN` statt 1.0 und -1.0 zurückzugeben. `safetanh()` fängt diese Fälle ab und liefert auch für sehr große Argumente ein sinnvolles Ergebnis.

[2] Es wäre schön, den Fehler genau eingrenzen zu können, bitte Bugreports schicken!

4.3 Weitere Hilfsfunktionen

4.3.1 sign() - Vorzeichen eines Wertes

y = sign(x)		
Symbol	Typ	Bedeutung
y	double	Rückgabewert
x	double	Argument

`sign()` liefert -1.0 für negative Argumente, 1.0 für positive Argumente, bei 0.0 wird auch 0.0 zurückgegeben.

Anmerkung: Es versteht sich, daß beim Einsatz in anderen als `discrete`-Gleichungen Vorsicht geboten ist.

4.3.2 periodic() - periodische Umrechnung

y = periodic(x, start, length)		
Symbol	Typ	Bedeutung
y	double	Rückgabewert
x	double	Argument
start	double	Startwert des zulässigen Intervalls
length	double	Länge des Intervalls

`periodic()` verschiebt Werte in ein definiertes Intervall und schneidet alle vollen Intervalle dieses Wertes ab.

☞ **Beispiel:**

```
y = periodic(x, 0.0, 2.0*PI);}
y = periodic(x, 0.0, 3600.0);}
```

Im ersten Fall wird z.B. der fortgeschriebene Drehwinkel eines rotierenden Körpers (pro ganzer Umdrehung $2.0 \cdot \pi$) in den Bereich $0..2 \cdot \pi$ verschoben. der Rückgabewert gibt also den aktuellen Drehwinkel zum Nullpunkt an, alle vollen Umdrehungen werden abgeschnitten.
Die zweite Anwendung ergibt immer den Anteil an der aktuellen Stunde, wenn das erste Argument z.B. die Zeit (in s) ist. Jede volle Stunde beginnt die Zählung neu.

Anmerkung: Da diese Funktion weder die Anforderungen an `discrete`-Gleichungen erfüllt noch denen der anderen Gleichungstypen gerecht wird, ist höchste Vorsicht geboten. Sie kann u.U. zur Berechnung von Hilfsgrößen benutzt werden, ihr Ergebnis sollte aber niemals direkt durch `return` als Ergebnis einer Gleichung zurückgegeben werden.

4.3.3 validate() - ersetze NaN, INF etc.

y = validate(x)		
Symbol	Typ	Bedeutung
y	double	Rückgabewert
x	double	Argument

Diese Funktion ersetzt `NaN`, `inf` und `-inf` durch endliche Werte[3]. Sie sollte nur in Ausnahmefällen Anwendung finden. Es ist meist sinnvoller, unzulässige Operationen direkt abzufangen als die Ergebnisse der Operationen zu manipulieren!

[3] Interessierte lesen den Quellcode.

5 Allgemeine Konzepte

5.1 Ausgabe von Meldungen

Es ist zwar möglich, die Standardbibliotheksfunktionen von C und beliebige weiter Bibliotheksfunktionen zu nutzen, also auch `printf(), fprintf()` und ihre Kollegen. SMILEbringt allerdings ein eigenes Konzept der Ausgabe mit, welches unbedingt Vorrang erhalten sollte! Grafische Oberflächen machen es erforderlich, den Ausgabestrom eventuell umzulenken und zu analysieren. Wenn nun jedes Modell und jede Bibliothek ihre eigenen Routinen benutzt wird dies stark erschwert! Außerdem bietet SMILEbereits die Möglichkeit, Ausgaben je nach Wichtigkeit und gewünschtem Detaillierungsgrad zu filtern. Zum Mitschreiben von Meldungen stehen folgende Funktionen zur Verfügung:

```
void log_if(bool condition, Sm_Facility, Sm_LogLevel,
            const char *fmt, ...);
void log_msg(Sm_Facility, Sm_LogLevel, const char *fmt, ...);
void log_cond(Sm_Facility, Sm_LogLevel);
void log_printf(const char *fmt, ...);
void log_iprintf(const char *fmt, ...);
```

Die `Sm_Facility` sind Auswahlparameter `SMF_Model0` bis `SMF_Model7` sowie `SMF_Model`.

Die `Sm_LogLevel` sind `LOG_NOTHING`, `LOG_ERROR`, `LOG_WARNING`, `LOG_NOTICE`, `LOG_INFO` und `LOG_DEBUG`.

Aktuelle Loglevel und Facility können im Experimentskript oder per Kommandozeilenoption gesetzt werden. Es werden jeweils die passenden und die schlimmeren Meldungen ausgegeben, bei gesetztem Loglevel `LOG_NOTICE` also Hinweise, Warnungen und Fehler.

Für das selektive Ein- und Ausschalten von Debugging-Code stehen ähnliche Funktionen und Macros zur Verfügung. Zur Dokumentation siehe das Referenzdokument.

Beispiel: [...noch in Bearbeitung...sorry!...]

5.2 Externe Funktionen:

Es war üblich, die Dateien mit externen Funktionen als Objective-C-Dateien zu kennzeichnen (*.m), obwohl sie in der Regel reinen ANSI-C-Code enthielten. Als **neue Konvention** sollten solche Dateien als reine ANSI-C-Dateien behandelt und deshalb mit dem Suffix **.c** versehen

werden. Der Suffix .m ist nun den von SMILEaus den Modelldateien generierten echten Objective-C-Dateien vorbehalten. Dies vereinfacht die automatische, Makefile-freie Kompilation, schafft mehr Klarheit und erleichtert die Wiederverwendbarkeit der Funktionsdateien.

6 Equations - Grundgleichungen

Prinzipiell verfügen alle hier definierten Modellklassen über einzelne oder mehrere Eingänge (`x`) und Ausgänge (`y`). Die Werte der Ausgangsgrößen werden in Abhängigkeit von den Eingangsgrößen (und eventuell weiteren Größen) berechnet. Manche der Modelle stellen lediglich die Funktionalität der in SmileFunctions enthaltenen Hilfsfunktionen in Klassenform zur Verfügung.

6.1 Lineare Funktion

In der Klasse *EquationLinear* wird der Ausgang `y` in Abhängigkeit vom Eingang `x` über die Funktion $y = a \cdot x + b$ bestimmt.

Die davon abgeleitete Klasse *EquationLinearAuto* berechnet darüberhinaus die Parameter `a` und `b` automatisch aus den x-y-Koordinaten von zwei auf der Geraden liegenden Punkten (`x_points[0]` und `y_points[0]` sowie `x_points[1]` und `y_points[1]`).

6.2 Polynome

[...noch in Bearbeitung...sorry!...]

6.3 Bequeme Sinusfunktion

EquationSin stellt ein flexibel skalierbares Sinussignal auf dem Ausgang `y` in Abhängigkeit vom Eingang `x` zur Verfügung. `y_min` und `y_max` stecken den Bereich ab, in dem das Signal sich bewegt. Die Periodendauer wird durch `x_periode` bestimmt. Der Wert von `x_start` verschiebt das Signal entlang der x-Achse derart, daß an dieser Stelle die Periode beginnt[1].

6.4 Mischungsfunktion

Die Klasse *EquationMixer* werden die zwei Eingänge `x[0]` und `x[1]` gewichtet mit den Verhältnis `a` zu einem Ausgang `y` zusammengefaßt (gemischt). Dies geschieht über die Beziehung $y =$

[1] Das bedeutet, hier ist $y = 0.5 \cdot (y_{min} + y_{max})$ und anteigend.

$(1 - a) \cdot y_0 + a \cdot y_1$. Genaugenommen stellt diese Klasse nur das bereits eingeführte Macro `MIXVAL()` als SMILE-Modellklasse zur Verfügung.

6.5 Wertebereichsfilter

Die Modellklasse *EquationRangeCutter* dient dem Beschneiden von Werten auf einen erlaubten Bereich. Solange der Wert des Einganges `x` weit genug innerhalb des Bereiches `y_min` bis `y_max` liegt, wird er an den Ausgang `y` durchgereicht ($x = y$), sonst auf direse Werte begrenzt. Dies geschieht über die bereits beschriebene Funktion `softcut()`, der Glättungsradius wird in dieser Klasse als `radius` bezeichnet.

7 Switches - Schalter mit und ohne Hysterese

7.1 Hystereseschalter

Der Schalter *HysteresisSwitch* stellt einen allgemeinen Schalter dar. In Abhängigkeit vom Eingang `x` wird der Ausgang `y` auf die über `y_value[0]` und `y_value[1]` bestimmten Werte gesetzt. **Achtung: Die Werte für `y_value` müssen konstant oder diskret geschaltet sein!** Es stehen eine Hysterese (durch Setzen von `x_value[0]` und `x_value[1]`), eine Totzeit (`deadtime` - minimal erlaubte Zeit zwischen Schaltvorgängen) und ein Kanal zum Ein- und Auschalten des Schalters (`active` - Erlauben von Schaltvorgängen, initialisiert auf 1) zur Verfügung.

In *HysteresisSwitchSymmetric* wird die über `hysteresis` definierte Hysterese symmetrisch um einen Punkt `x_switch` verteilt.

7.2 Einfache Schalter

Die Klasse *SimpleSwitch* stellt einen hysterese- und totzeitfreien Schalter dar[1].

Der *ZeroOneSwitch* entspricht exakt dem *SimpleSwitch*, er ist nur auf einen speziellen Anwendungsfall initialisiert. Bei `x` kleiner 0.5 liefert er 0.0, anderenfalls 1.0.

7.3 Stetiger Pseudo-Schalter

Die Klasse *SoftSwitch* ist von den Schnittstellen baugleich zum *SimpleSwitch*, besitzt aber eine zusätzliche Größe `smoothing`. Wie der Name vermuten läßt, handelt es sich nicht um einen echten Schalter mit diskreten Zuständen, sondern eine stetige und stetig differenzierbare Approximation davon mit einstellbarer Glättung. Das heißt, daß `y` auch Werte zwischen `y_value[0]` und `y_value[1]` annehmen kann! Intern wird die Funktion `softswitch()` eingesetzt, zu weiteren Informationen siehe dort.

[1] Es ist bei der Nutzung darauf zu achten, daß die fehlende Hysterese und Totzeit zu inkonsistentem Schalten und "Modellflattern" führen können!

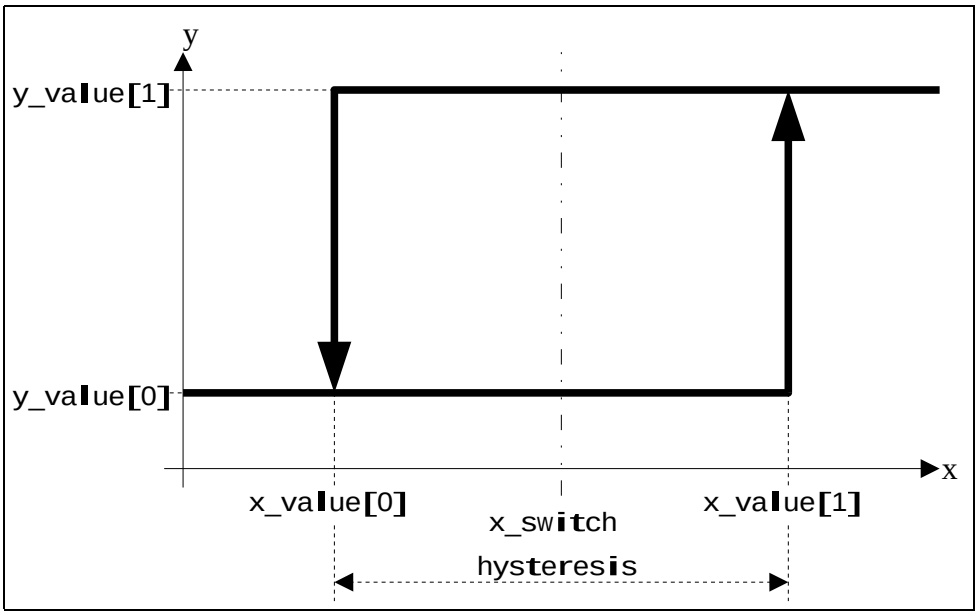

Abbildung 7.1: Hystereseschalter - Funktionsschema (`x_switch` und `hysteresis` nur in der Klasse *HysteresisSwitchSymmetric*)

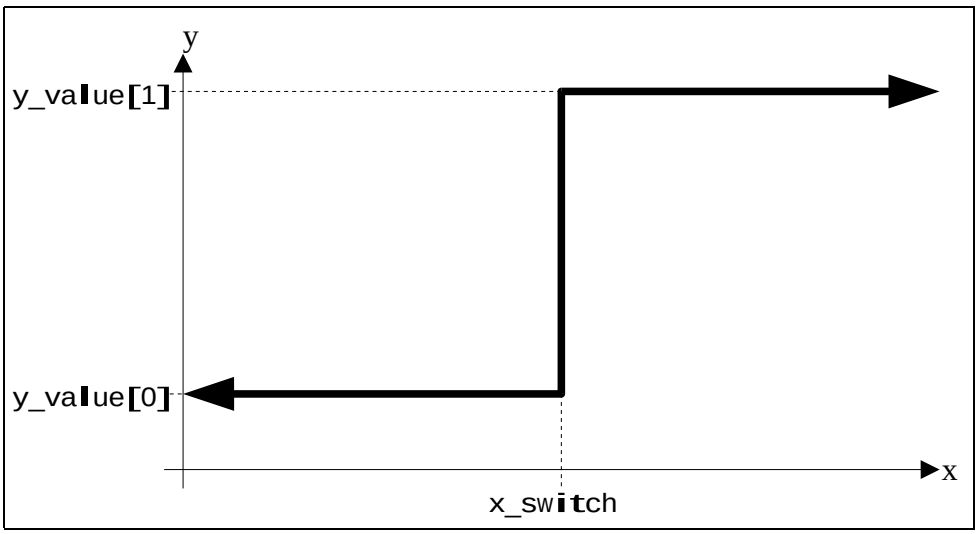

Abbildung 7.2: Einfache Schalter - Funktionsschema

7.4 Anwendung der Schalter

Alle Schalter bis auf den *SoftSwitch* funktionieren nur mit konstanten oder diskret geschalteten Werten für `y_value[0]` und `y_value[1]`[2]! Wenn diese Größen kontinuierlich bestimmt werden, muß entweder der *SoftSwitch* genutzt werden, oder z.B. eine Kombination des Schalters mit *EquationMixer*. Der Schalter (mit oder ohne Hysterese) könnte dann zwischen 0 und 1 schalten, diesen Wert an den *EquationMixer* als Mischsignal (`a`) weitergeben. Der Mixer errechnet dann aus diesem Wert und den beiden kontinuierlichen Eingangssignalen ein kontinuierliches (und manchmal durch den Schalter springendes) Ausgangsignal.

[2]Das selbe Problem wurde bereits im Abschnitt über das Macro `MIXVAL()` erörtert.

8 Logic - Binäre Schaltungen

Im Logic-Paket sind Bausteine der binären Logik enthalten. Binär heißt, dass nur die Werte 0 und 1 (oder FALSE und TRUE) ausgewertet werden. Alle Eingabewerte werden intern auf die ganzen Zahlen 0 und 1 umgerechnet. Vereinfachend wird jeder Wert, der unterhalb von 0.5 liegt, als 0 oder FALSE angenommen, jeder darüberliegende Wert als 1 oder TRUE. Dieses Verhalten kann durch das Macro `BIN()` am Anfang der Datei geändert werden.

Alle Namen von Modellklassen dieses Pakets sind mit dem Prefix *Logic_* versehen. Für Bausteine mit variabler Anzahl von Ein- und Ausgängen kann über die symbolische Konstante `LOGIC_N_INPUTS` die maximal verfügbare Größe festgelegt werden.

Im Folgenden ist mit 'Setzen' oder 'Aktivieren' eines Einganges das Belegen der entsprechenden Größe mit einem Wert von 1 (TRUE) gemeint.

8.1 AND, OR und NOT - Grundbausteine

Die Modellklassen *Logic_AND*, *Logic_OR* und *Logic_NOT* bilden die Grundbausteine der binären Logic ab.

Die ersten beiden verfügen über die gleichwertigen Eingänge `input_0` und `input_1` und den Ausgang `output`. Zusätzlich kann durch Aktivieren der Größen `inv_input_0` und `inv_input_1` der Wert des jeweiligen Einganges negiert werden, es wird damit quasi ein NOT vor den Eingang geschaltet. Der Ausgang `output_inv` stellt den negierten Wert von `output` dar.

Logic_NOT besitzt nur `input` und `output`.

☞ **Beispiel:**
Wird bei *Logic_AND* `inv_input_1` auf gesetzt, stellt das an `output_inv` anliegende Signal die folgende Funktion dar:

```
NOT(input_0 AND NOT(input_1))
```

Dies bedeutet im Klartext, daß dieser Ausgang nur dann auf FALSE liegt, wenn `input_0` TRUE ist **und** `input_1` FALSE ist. Bei allen anderen Kombinationen steht `output_inv` auf TRUE.

8.2 MultiAND, MultiOR und MultiNOT - beliebig vielen Eingängen

Die Modellklassen *Logic_MultiAND*, *Logic_MultiOR* und *Logic_MultiNOT* funktionieren prinzipiell genauso wie die oben beschriebenen Bausteine.

Die ersten beiden stellen AND- und OR-Verknüpfungen beliebig vieler Eingänge dar, die Eingänge heißen hier `input` und die jeweiligen Invertierungsschalter `inv_input`. Beide Größen sind eindimensionale Felder der Ausdehnung `LOGIC_N_INPUTS`. Die Ausgänge heißen analog zu den Bausteinen mit nur zwei Eingängen.

Logic_MultiNOT ist lediglich ein Feld von NOT-Schaltungen. Die Eingänge `input` und Ausgänge sind jeweils ein Feld der Ausdehnung `LOGIC_N_INPUTS`. Diese Komponente macht nur Sinn, wenn viele unabhängige NOT-Bausteine gebraucht werden, die als einzelne Komponenten unübersichtlich sind.

8.3 FlipFlop - bistabiler Flip-Flop

[...noch in Bearbeitung...sorry!...]

8.4 SignalHolder - monostabiler Flip-Flop

Logic_SignalHolder dient dazu, ein Eingangssignal beliebiger Länge für eine definierte Zeit zu halten. Der Eingang heißt hier `on`, wird er aktiviert (0-1-Übergang), schaltet der Ausgang `output` auf TRUE und hält diesen Zustand genau die durch `holdtime` definierte Zeitspanne, danach schaltet er wieder auf FALSE. Dies geschieht unabhängig davon, wie lange der Eingang aktiviert ist und wie er sich über diese Zeitspanne verhält. Erst nach dem automatischen Zurückschalten des Ausganges ist der Baustein bereit, beim nächsten 0-1-Übergang an `on` erneut zu schalten.

8.5 SignalCounter - Impulszähler

Logic_SignalCounter gibt in der Größe `count` die Anzahl der TRUE-Zustände[1] auf dem Eingang `input` wieder.

[1] Genauer gesagt werden die 0-1-Übergänge am Eingang gezählt.

9 Selectors - Auswahl aus Feldgrößen

Die Datei `Selector.smi` enthält Modelle, die jeweils einen der Werte der Feldgröße `input` auf dem Ausgang `output` zurückgeben, es also immer genau ein Eingangselement auf den Ausgang "durchgeschaltet". Das Feld hat durch die symbolische Konstante `SELECTOR_N_INPUTS` gegebene Maximalgröße, beachtet werden jedoch nur die ersten `n_inputs` Elemente.

9.1 Selector

Die Größe `index` legt den Index des Rückgabewertes aus dem Feld fest. Liegt `index` außerhalb des Bereiches 0 bis `n_inputs`, wird stattdessen der Wert der Größe `default` eingesetzt.

9.2 MinimumSelector und MaximumSelector

`output` erhält den Wert des kleinsten bzw. größten Elementes des Feldes `input`. `index` enthält den Index des entsprechenden Elementes. Haben mehrere Feldelemente den selben Wert, wird der Index des ersten entsprechenden Elementes eingesetzt.

9.3 BinarySelector - Logische Auswahl

Die Klasse besitzt eine zusätzliche Feldgröße namens `bin_input` mit der selben Ausdehnung wie `input`. Der Index des ersten gefundenen Elementes dieses Feldes, daß im logischen Sinne wahr ist (Kriterien siehe `Logic.smi`), wählt das entsprechende Element von `input` aus, das auf `output` wiedergegeben wird. Sind alle Elemente von `bin_input` logisch falsch, wird `default` wiedergegeben.