ASCET V5.2

Referenzhandbuch

Copyright

Dokument EC010005 R5.2.2 DE

Inhalt

Die Modelliersprache

Referenzlisten

ASCET V5.2

Die Modelliersprache

1 Projekte

In ASCET wird ein eingebettetes Softwaresystem im Kontext eines Projekts definiert. Ein Projekt umfasst mindestens die folgenden Bestandteile:

- Eine Sammlung von Modulen
- Den Task-Schedule für das Echtzeit-Betriebssystem
- Die Definition der Interprozesskommunikation

Der zentrale Bestandteil eines Projekts ist die Definition des Task-Schedules des Betriebssystems. In ihm wird das dynamische Verhalten des Systems beschrieben. Abb. 1-1 veranschaulicht die Struktur eines Projekts.

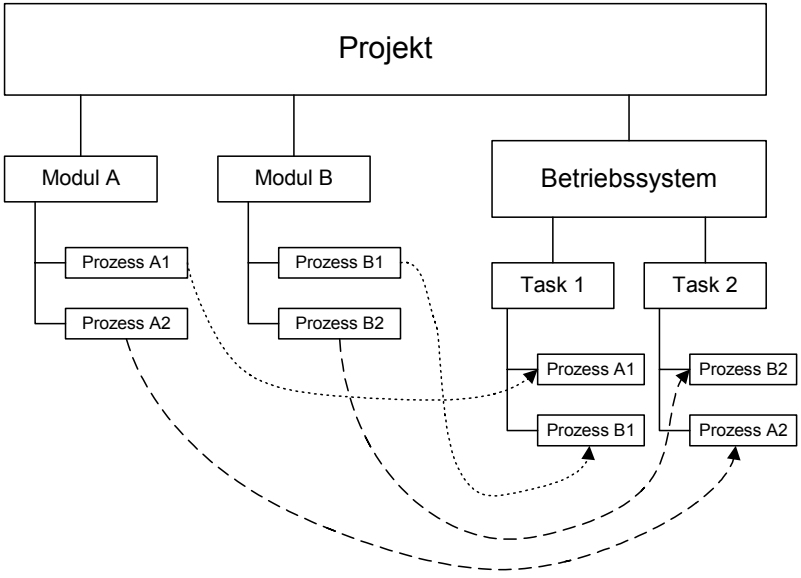


Abb. 1-1 Die Struktur eines Projekts

1.1 Der Task-Schedule für das Betriebssystem

Ein wesentlicher Bestandteil eines Embedded Control Systems ist das zugrundeliegende Echtzeit-Betriebssystem, das die Ausführung der verschiedenen Algorithmen und Berechnungen steuert. In ASCET wird die Spezifikation des Task-Schedules durch einen speziellen Editor unterstützt, mit dem alle relevanten Daten für das Scheduling des Betriebssystems vorgegeben werden können.

Die Spezifikation des Task-Schedules basiert auf dem Echtzeit-Betriebssystem für Kraftfahrzeuge ERCOS^{EK}. Um die große Zahl von parallelen Anforderungen an das Embedded Control System zu bedienen, z. B. Nockenwellen-Unterbrechungen oder Abtastung mit einer festen Frequenz, ist ein prioritätsbasiertes, kooperatives und präemptives Scheduling das Kernstück des Betriebssystems.

Dieses Scheduling steuert die Ausführung von Tasks in einer Multitasking-Umgebung. Eine Task ist als eine Liste von Prozessen definiert, die in einer vorgegebenen Reihenfolge auszuführen sind. Ein Prozess ist ein beliebiger Teil eines Regelalgorithmus, der mit einer vorgegebenen Frequenz oder als Reaktion auf eine externe Unterbrechung auszuführen ist.

Da ein Steuerungssystem eine Anzahl von Algorithmen enthält, kann die Zahl der Prozesse sehr groß sein. Andererseits weisen viele dieser Prozesse ein ähnliches dynamisches Verhalten auf. Die Zusammenfassung von Prozessen mit demselben dynamischen Verhalten zu Tasks verringert daher den Verwaltungsaufwand des Betriebssystems und strukturiert das dynamische Verhalten der Anwendung. Prozesse mit demselben dynamischen Verhalten werden deshalb zu einer Task zusammengefasst.

Die Definition eines Echtzeit-Task-Schedules besteht aus:

- Scheduling
- Tasks
- Prozessen
- Betriebsmodi (*Application Modes*)

1.1.1 Scheduling

Das Betriebssystem steuert den zeitlichen Ablauf der Ausführung der in den Modulen definierten Prozesse. Die Definition des Schedules besteht im Gruppieren von Prozessen zu Sequenzen, wobei jede Sequenz eine Task im Task-

Schedule des Betriebssystems definiert. Die Tasks werden durch das Betriebssystem in verschiedenen Modi aktiviert, zum Beispiel periodisch durch Timer, oder durch Software- oder externe Ereignisse.

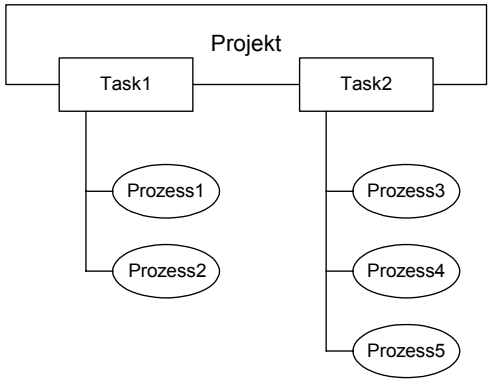


Abb. 1-2 Gruppieren von Prozessen zu Tasks

Abb. 1-2 zeigt zwei Tasks mit ihnen zugeordneten Prozessen. `Task1` wird alle 10 ms aktiviert und hat eine höhere Priorität als `Task2`, die alle 20 ms aktiviert wird. Die Laufzeiten der Prozesse sind folgende: `p1`= 2ms, `p2` = 1ms, `p3` = 2ms, `p4` = 1ms, `p5` = 1ms. Die Scheduling würde dann wie folgt aussehen:

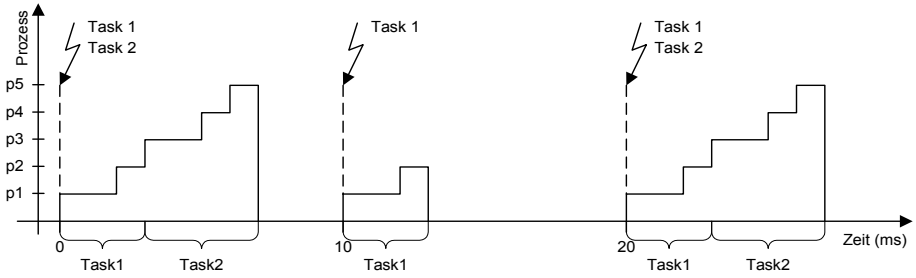


Abb. 1-3 Ein einfacher Task-Schedule

Das Betriebssystem kennt drei Arten von Scheduling. Beim *kooperativen Scheduling* wird der laufende Prozess nicht unterbrochen, wenn eine Task mit einer höheren Priorität aktiviert wird. Eine neue Task beginnt, nachdem der laufende Prozess beendet ist. Wenn in der laufenden Task (derjenigen, die unterbrochen wird) noch mehr Prozesse auszuführen sind, so wird sie unterbrochen, bis die unterbrechende Task abgeschlossen ist. Nachdem die unterbrechende Task abgeschlossen ist, wird die unterbrochene Task fortgesetzt.

Dieser Typ von Scheduling ist in Abb. 1-4 dargestellt, wobei die Laufzeiten der Prozesse folgende sind: p1= 2ms, p2 = 1ms, p3 = 5ms, p4 = 4ms und p5 = 2ms.

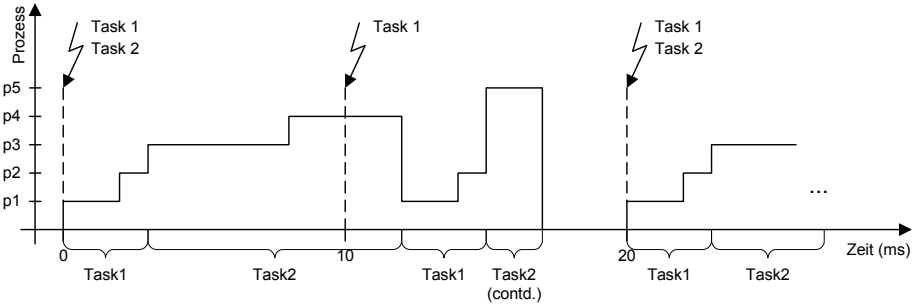


Abb. 1-4 Wiederaufnahme einer unterbrochenen Task bei kooperativem Scheduling

Beim *präemptiven Scheduling* wird der laufende Prozess direkt unterbrochen, sobald eine Task mit einer höheren Priorität aktiviert wird. Da grundsätzlich kooperative Tasks eine niedrigere Priorität haben als präemptive oder nicht unterbrechbare Tasks (s. Abb. 1-7), kann eine präemptive Task nicht von einer kooperativen Task unterbrochen werden. Nachdem die unterbrechende Task abgeschlossen ist, wird der Prozess fortgesetzt. Abb. 1-5 zeigt dasselbe Szenario wie oben (d.h. dieselben Prozesslaufzeiten) für ein präemptives Scheduling.

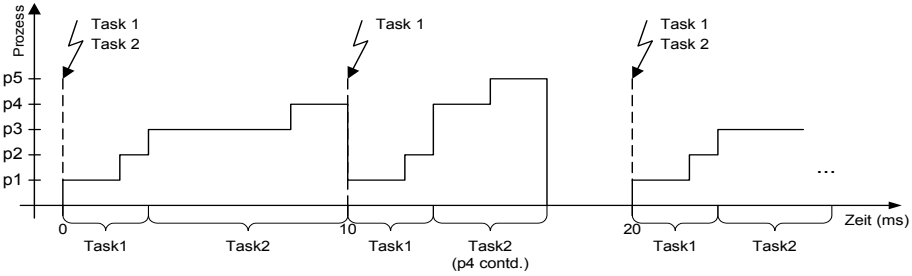


Abb. 1-5 Wiederaufnahme einer unterbrochenen Task bei präemptivem Scheduling

Beim *nicht-unterbrechbaren (non-preemptable) Scheduling* (nur in Verbindung mit einem Mikrocontroller-Target) wird weder der laufende Prozess noch die laufende Task unterbrochen, wenn eine Task mit höherer Priorität aktiviert

wird. Die neue Task wird erst abgearbeitet, wenn die nicht-unterbrechbare Task beendet ist. Abb. 1-6 zeigt dasselbe Szenario wie oben (d.h. dieselben Prozesslaufzeiten) für ein nicht-unterbrechbares Scheduling.

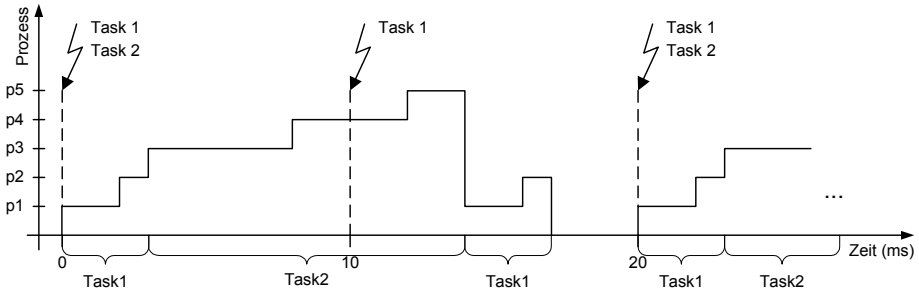


Abb. 1-6 Task-Schedule für eine nicht-unterbrechbare Task

Hinweis

Nicht-unterbrechbare Tasks wurden in ERCOS^{EK} eingeführt, um OSEK-Konformität zu gewährleisten. Ihre Verwendung wird jedoch ausdrücklich nicht empfohlen, da sich die meisten Aufgabenstellungen leichter durch ausschließliche Verwendung kooperativer und präemptiver Tasks lösen lassen und die Konfiguration von nicht-unterbrechbaren Tasks aufgrund verschiedener Randbedingungen kompliziert sein kann.

1.1.2 Tasks

Eine Task enthält eine Liste von Prozessen, die bei Aktivierung dieser Task ausgeführt werden. Die Reihenfolge der Ausführung der Prozesse ist feststehend. Die Art und Weise, wie der Ablauf einer Task durch den Scheduler des Betriebssystems gesteuert wird, wird durch die Einstellungen der Task festgelegt. Es gibt folgende verschiedene Taskmodi:

- *Alarm*-Tasks werden periodisch aktiviert. Die Aktivierungsfrequenz wird in Sekunden vorgegeben.
- *Timetable*-Tasks (nur Mikrocontroller-Targets) sind Alarm-Tasks, die in eine Zeitsteuertabelle geschrieben werden. Auf diese Weise kann Laufzeit eingespart werden (allerdings auf Kosten höheren Speicherbedarfs).
- *Interrupt*-Tasks werden durch ein externes Ereignis aktiviert. Für jeden Prozessor sind verschiedene Typen von Ereignissen verfügbar. Das zutreffende Ereignis kann aus einer Ereignisliste ausgewählt werden.
- *Software*-Tasks werden durch Aufrufen einer Routine des Betriebssystems aktiviert, d.h. sie werden direkt durch die Software aktiviert.

- *Init*-Tasks werden einmal vor dem Start des Betriebssystems aktiviert. Init-Tasks enthalten Code für die Initialisierung des Systems.

Jede Task wird ferner einer der drei Scheduling-Gruppen zugeordnet, der nicht-unterbrechbaren, der präemptiven oder der kooperativen, und innerhalb jeder Gruppe einer der verfügbaren Prioritätsebenen. Die Anzahl der Prioritätsebenen kann für jede Scheduling-Gruppe vom Benutzer definiert werden, und sie bestimmt den Speicherbedarf der Scheduler-Tabellen. Sie sollte für das endgültige System optimiert werden.

Tasks, die eine höhere Priorität als die laufende Task haben, können die laufende Task unterbrechen, es sei denn, die laufende Task ist nicht-unterbrechbar (*non-preemptable*). Wenn die unterbrechende Task der präemptiven Scheduling-Gruppe angehört, wird die laufende Task sofort unterbrochen, andernfalls erfolgt die Unterbrechung am Ende des laufenden Prozesses. Präemptive und nicht-unterbrechbare Tasks haben stets eine höhere Priorität als kooperative Tasks. Abb. 1-7 zeigt das verwendete Prioritätsschema. Welche Tasks tatsächlich verfügbar sind, hängt vom verwendeten Target ab.

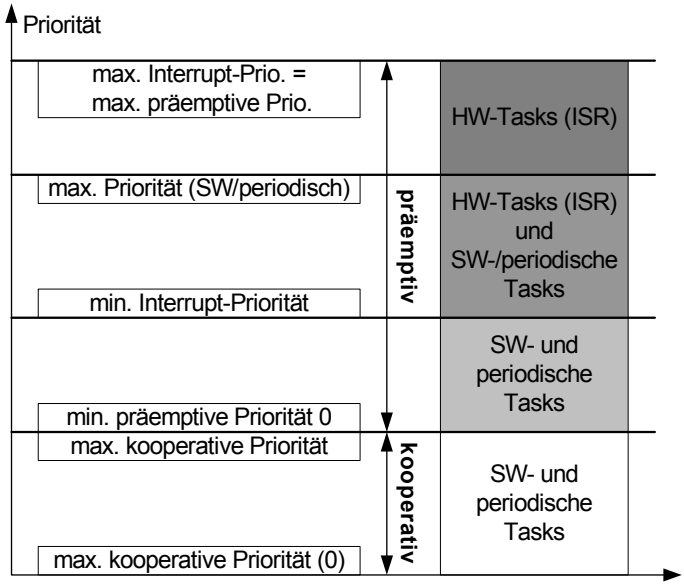


Abb. 1-7 Prioritätsschema

Bei jedem Aktivieren einer Task wird die seit der vorhergehenden Aktivierung vergangene Zeit in der globalen Variablen `dT` gespeichert. Diese Variable kann bei der Definition von Algorithmen verwendet werden, um die Regelalgorithmen unabhängig von ihrer Abtastfrequenz zu beschreiben.

1.1.3 Prozesse

Eine Task besteht aus einer Sequenz von *Prozessen*. Prozesse enthalten den Ausführungscode des Programms. Der Hauptteil eines Prozesses wird sequentiell ausgeführt. Da Tasks durch präemptive Tasks mit einer höheren Priorität unterbrochen werden können, können Prozesse mitten in ihrer Ausführung unterbrochen werden. Deshalb müssen Prozesse so entworfen werden, dass sie parallel ausgeführt werden können.

Wenn in einem präemptiven System gearbeitet wird, besteht das Hauptproblem in der Datenkonsistenz. Das Betriebssystem muss garantieren, dass das Ergebnis der Berechnung in einem Prozess allein vom Wert der Eingangsvariablen abhängt, nicht jedoch von der Reihenfolge der Ausführung im System.

Um dieses Problem zu lösen, wird in Prozessen das ERCOS^{EK}-Konzept der *Messages* unterstützt. Im Betriebssystem ERCOS^{EK} sind Messages geschützte globale Variable. Der Schutz wird dadurch erreicht, dass mit Kopien der globalen Variablen gearbeitet wird. Das System analysiert, ob eine Kopie benötigt wird, und sorgt für ein optimales Datenkonsistenzschema ohne nachteilige Auswirkungen auf den Kern der Laufzeit.

1.1.4 Betriebsmodi

Betriebsmodi (engl. *Application Modes*) sind ein spezielles Merkmal des Betriebssystems ERCOS^{EK}. Um die Laufzeitbelastung des Prozessors gering zu halten, kann das Betriebssystem in verschiedenen Modi betrieben werden. Typische Modi sind der Normalbetrieb, der EEPROM-Programmiermodus usw. Diese Modi schließen einander gegenseitig aus, d.h. zu einem gegebenen Zeitpunkt ist jeweils nur ein Modus aktiv. Daher müssen in jedem Modus nur die jeweils zutreffenden Tasks ausgeführt werden.

Jeder Task ist ein Betriebsmodus zugeordnet, in dem sie ausgeführt wird. Umschaltungen zwischen den Betriebsmodi werden durch die Software aktiviert. Beim Eintritt in einen neuen Betriebsmodus werden die diesem Betriebsmodus zugeordneten Init-Tasks aktiviert.

Hinweis

Die Umschaltung von einen in den anderen Betriebsmodus erfolgt durch Aufruf eines Betriebssystemdienstes. Näheres findet sich in der API-Beschreibung des ERCOS^{EK} Handbuches.

1.2 Module und Prozesse

Die den Tasks zugeordneten Prozesse werden im Kontext von Modulen definiert. Ein Modul umfasst eine Anzahl von miteinander zusammenhängenden Prozessen, z. B. Prozesse, die zu einer Lambdaregelungs-Funktion gehören. Die

in einem Modul beschriebene Funktionalität kann in verschiedene Prozesse aufgespalten werden, da verschiedene Teile eines Regelalgorithmus zu verschiedenen Zeiten verarbeitet werden können. Dies bewirkt eine beträchtliche Verkürzung der Ausführungszeit für die Regelalgorithmen, da nur die empfindlichsten Teile der Algorithmen mit der höchsten Frequenz verarbeitet zu werden brauchen. Andererseits werden die Beschreibungen der Algorithmen nicht verteilt, wodurch sie sich leichter entwickeln, pflegen und verstehen lassen.

Die Funktionalität einer komplexen Steuerungs-Task kann auf verschiedene Module verteilt werden, die hierarchisch modelliert werden können. Zur weiteren Verfeinerung können Klassen und Zustandsautomaten für Unteralgorithmen oder Serviceroutinen verwendet werden (z. B. Akkumulator, Einspritzregelung usw.).

Module werden ausschließlich von Projekten verwendet und sind die Komponenten der obersten Ebene innerhalb eines Projekts. Module werden gewöhnlich verwendet, um einen nur einmal vorhandenen Bestandteil eines Projekts zu beschreiben, z. B. eine Lambdaregelung. Deshalb können Module nur eine Instanz innerhalb eines Projekts besitzen, im Gegensatz zu anderen Komponenten, die eine beliebige Zahl von Instanzen haben können (z. B. Akkumulatoren).

Wie alle anderen Komponenten haben Module eine Schnittstelle. Die Schnittstelle eines Moduls besteht aus seinen Prozessen und den Messages, die für den Datenaustausch verwendet werden.

1.3 Interprozesskommunikation

Die Kommunikation zwischen Prozessen wird über Messages erzielt, die geschützte globale Variable in ERCOS^{EK} sind. Die Datenkonsistenz wird erreicht, indem immer dann, wenn eine Kopie benötigt wird, mit Kopien der eigentlichen Variablen gearbeitet wird.

Abb. 1-8 zeigt, wie Dateninkonsistenz in einem präemptiven System entstehen kann. Um diesen Konflikt zu vermeiden, wird die Interprozesskommunikation mit Messages modelliert. Zu Beginn eines Prozesses werden alle Receive-Messages (die Messages, die nur gelesen werden) durch den Prozess empfangen. Beim Empfang einer Message wird eine automatische temporäre Kopie hergestellt, mit der der Prozess arbeitet. Am Ende des Prozesses werden alle Messages, in die geschrieben wurde, zurück auf die eigentliche Message kopiert. Dieser Mechanismus garantiert, dass die Werte der Variablen innerhalb eines Prozesses unverändert bleiben, sofern nicht der Prozess selbst seinen Wert ändert.

Die Verwendung von geschützten globalen Variablen für die Interprozesskommunikation, d.h. die Verwendung von State-Messages, ist für Embedded Control Systeme geeignet. Es gibt keine Abhängigkeit zwischen dem Sender und dem Empfänger einer Message, so dass kein kompliziertes und viel Laufzeit verbrauchendes Synchronisationsschema benötigt wird. Zweitens existiert bei Verwendung von State-Messages keine Eins-zu-Eins-Beziehung zwischen einem Sender und dem Empfänger. Deshalb kann eine Message durch mehr als einen Prozess empfangen werden.

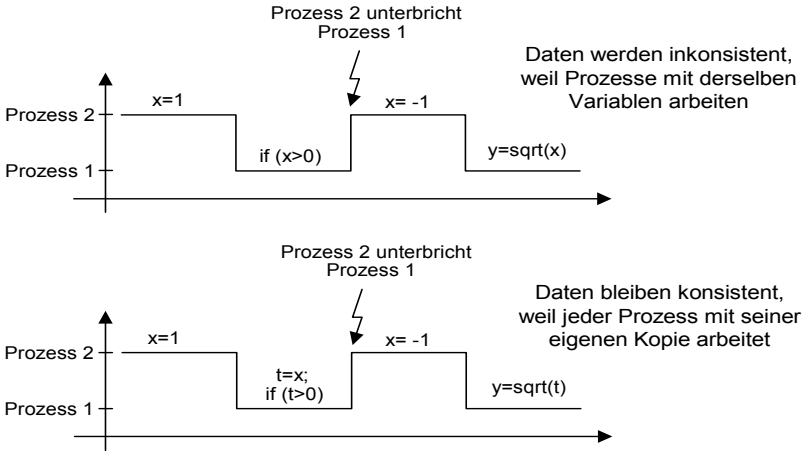


Abb. 1-8 Dateninkonsistenz in einem System mit unterbrechender Priorität

Der Message-Mechanismus beruht auf dem ERCOS^{EK} Message-Prinzip. Die ERCOS^{EK} Entwicklungsumgebung enthält ein Offline-Systemoptimierungsmerkmal, durch das die Message-Implementierung optimiert werden kann. Dabei werden Kopien nur dann eingeführt, wenn die Datenkonsistenz gefährdet ist, und Kopien werden nur am Anfang und am Ende einer Task hergestellt.

Die Interprozesskommunikation wird durch das Projekt aufgelöst. Messages mit demselben Namen sind miteinander verknüpft und repräsentieren dieselbe Message. Wenn zum Beispiel zwei Prozesse die Message `velocity` verwenden, so kommunizieren sie durch Schreiben in und Lesen aus dieser Variablen. Derselbe namensbasierte Auflösungsmechanismus wird bei anderen globalen Objekten ausgeführt, z. B. globalen Variablen oder globalen Parametern.

2 Komponenten

Ein Projekt befindet sich auf der obersten Ebene der Spezifikation eines Embedded Control Systems in ASCET. Hier wird das Grundgerüst einer Anwendung definiert, und hier wird ihre Ausführung gesteuert. Ein Projekt ist das Gehirn eines Embedded Control Systems.

Im Vergleich damit stellen die Komponenten den Körper dar. Sie werden verwendet, um die eigentlichen Steuerungsalgorithmen und andere verschiedenartige Verarbeitungs-Tasks vorzugeben, die in dem eingebetteten Steuerungssystem ausgeführt werden sollen.

Komponenten verfügen über eine klar definierte Schnittstelle, die beschreibt, wie und wann die in den Komponenten beschriebenen Algorithmen auszuführen sind und wie der Datenaustausch mit anderen Komponenten durchgeführt werden soll.

Es gibt zwei Typen von Komponenten: Module und Klassen. Ein zentraler Aspekt beim Entwurf beider Typen ist die Datenkapselung, wobei ASCET einer objektorientierten Herangehensweise folgt. Eine Komponente enthält eine Anzahl von Elementen, die von allen Prozessen oder Methoden verwendet werden können, die in dem betreffenden Modul oder der betreffenden Klasse definiert sind. Der Geltungsbereich dieser Elemente kann auf einen lokalen Bereich beschränkt sein. Selbst für Messages kann der Geltungsbereich auf Prozesse beschränkt sein, die nur innerhalb des betreffenden Moduls definiert sind.

Eine Komponenten-Spezifikation besteht aus:

- dem Inhalt der Komponente, d.h. Vereinbarungen der Variablen, Parameter usw., die die Komponente verwendet;
- der Schnittstelle der Komponente in Form von Prozessen oder Methoden. Diese Schnittstelle kann erweitert werden, indem der direkte Zugriff auf interne Variable (von Klassen) und Messages (die in Modulen verwendet werden) ermöglicht wird;
- den Algorithmen selbst, welche die Verarbeitung innerhalb eines Prozesses oder einer Methode vorgeben.

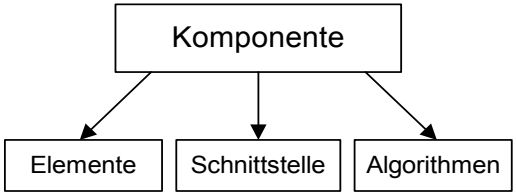


Abb. 2-1 Die Elemente einer Komponentenspezifikation

Im folgenden werden Module und Klassen allgemein erörtert. Anschließend wird die Struktur der Schnittstelle einer Komponente erläutert. Die verschiedenen Möglichkeiten, wie Algorithmen beschrieben werden können (Blockdiagramme, durch Text, C-Code) werden in den darauffolgenden Kapiteln erörtert. Der letzte Abschnitt dieses Kapitels ist einem speziellen Typ von Klassen gewidmet: den Zustandsautomaten. Dieser spezielle Typ von Klassen kann nur unter Verwendung von Blockdiagrammen beschrieben werden.

2.1 Gegenüberstellung von Modulen und Klassen

Wenn ein Embedded Control System spezifiziert wird, sind die Echtzeit-Anforderungen des Systems von entscheidender Bedeutung. Um diesen Anforderungen gerecht zu werden, können in ASCET spezielle Komponenten mit einer echtzeitfähigen Schnittstelle, nämlich Module, verwendet werden.

Ein Modul definiert eine Anzahl von Prozessen; zusätzlich können auch Methoden definiert werden. Ein Prozess enthält ein Stück Code, das sequentiell ausgeführt wird. Prozesse werden durch das Betriebssystem aktiviert, es können keine Parameter übergeben werden. Statt dessen verwenden Module Messages für den Datenaustausch, d.h. den direkten Zugriff auf einen globalen Variablenraum, woraus ein höchst effektiver Kommunikationsmechanismus resultiert.

Im Gegensatz zu Prozessen, die nur durch das Betriebssystem aktiviert werden können, sind Methoden wesentlich flexibler. Jede Methode kann eine beliebige (aber feste) Anzahl von Argumenten und einen einzigen Rückgabewert haben.

Das Verhalten von Modulen ist einzigartig innerhalb eines eingebetteten Steuerungssystems in dem Sinne, dass sie im Kontext eines Projekts nur einmal eine Instanzbildung ermöglichen. Um diese Einschränkung zu vermeiden, können Klassen verwendet werden. Klassen sind objektorientierte abstrakte Datentypen, die Daten kapseln und eine gut definierte Schnittstelle zur Verfügung stellen. Die Schnittstelle ist eine Sammlung von Methoden, welche von einer beliebigen Stelle innerhalb des Programms aus aufgerufen werden können.

Klassen ermöglichen mehr als eine Instanzbildung, z. B. kann in einem Projekt mehr als eine Akkumulatorklasse existieren. Jede Instanz einer Klasse hat ihren eigenen Datenraum (ihre eigenen Parameter und Variablen), jedoch haben alle Instanzen ein und dieselbe Spezifikation gemeinsam. In Klassen definierte globale Variablen sind für alle Instanzen einer Klasse dieselben (und können bei einer objektorientierten Betrachtungsweise als Klassenvariable betrachtet werden), doch auf sie kann auch durch andere Komponenten zugegriffen werden.

Klassen unterstützen jedoch nicht Echtzeit-Interprozesskommunikation über Messages. Dies hat zwei Gründe. Erstens können Klassen mehrere Instanzen haben, und das Datenkonsistenzschema von ERCOS^{EK} kann nicht mehrfache Instanzbildungen verwalten. Zweitens werden Prozesse statisch einer festen Task zugeordnet.

Jedesmal, wenn ein Prozess abläuft, erzeugt das Betriebssystem Kopien von dessen sämtlichen Messages. Diese Kopien sind nur für diejenige Instanz des Prozesses zugänglich, die sie erzeugt hat. Daher erhält, wenn dieselbe Message von verschiedenen Prozessen verwendet wird, jeder Prozess seine eigene Kopie der Message. Diese Strategie wird vom Echtzeit-Betriebssystem angewandt, um die Datenkonsistenz über mehrere Prozesse hinweg sicherzustellen.

Methoden hingegen können von verschiedenen Stellen im Programm aus beliebig aufgerufen werden, zum Beispiel von verschiedenen Prozessen in verschiedenen Tasks. Die Methode „kennt" die aufrufende Task nicht. Es kann also nicht entschieden werden, welche Kopie der Message bei welchem Methodenaufruf relevant ist.

Die Eigenschaften von Modulen und Klassen werden zusammengefasst in Tab. 2-1.

Eigenschaft	Modul	Klasse
Prozesse	x	
Methoden	x	x
Übergabe des Arguments		x
Messages	x	
Mehrfache Instanzen		x
Hierarchischer Entwurf	x	x

Tab. 2-1 Die Eigenschaften von Modulen und Klassen

Zustandsautomaten sind ein spezieller Typ von Klassen, die in ASCET zur Verfügung stehen. Ihr semantisches Verhalten ist dasselbe wie das von Klassen, doch die Darstellungsweisen sind unterschiedlich. Zustandsautomaten verfügen zum Beispiel über spezielle Methoden zur Berechnung der Bedingungen eines Zustandsübergangs.

Beim Spezifizieren von Komponenten, sowohl von Modulen als auch von Klassen, ist die Struktur oft hierarchisch, da andere zuvor definierte Klassen oder Module wiederverwendet werden sollen.

2.2 Definition und Instanzbildung von Komponenten

Eine Komponente beschreibt einen abstrakten Datentyp, sie stellt eine Schnittstelle zur Verfügung, über die sie mit ihrer Umgebung in Wechselwirkung steht. Bei Verwendung einer Komponente in einem Projekt muss jedes Element erzeugt werden, d.h. für jedes Element müssen echte Speicherzellen zugewiesen werden. Der Prozess der Erzeugung eines Objekts wird auch Instanzbildung genannt. Bei einer Instanzbildung wird die notwendige Datenstruktur aufgebaut und initialisiert.

Jede Instanz einer Komponente hat ihre eigene Menge von Elementen, „erbt" jedoch die Schnittstelle und die funktionale Beschreibung von der Komponente selbst.

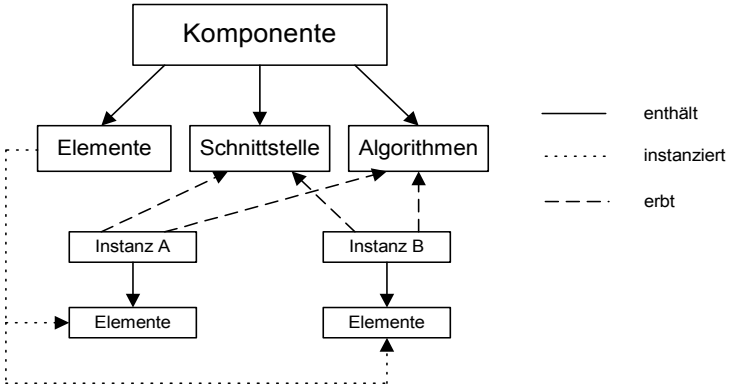


Abb. 2-2 Instanzbildung und Vererbung in Komponenten

Die Definition einer Komponente ist daher die Definition einer Schablone für die bei Instanzbildungen realisierten Komponenten. Der Unterschied zwischen Schablone und Instanz ist für Module nicht offensichtlich, da Module in einem

Projektkontext nur einmal auftreten, d.h. bei Modulen erfolgt nur eine Instanzbildung. Bei Modulen besteht eine Eins-zu-Eins-Beziehung zwischen der Schablone und der Instanz.

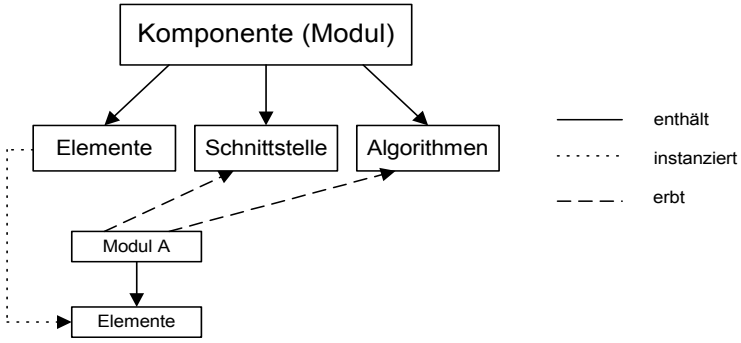


Abb. 2-3 Instanzbildung und Vererbung in Modulen

Klassen dagegen können mehrere Instanzen haben. Hier wird die Unterscheidung zwischen Definition und Instanzbildung offensichtlicher, da hier keine einfache Eins-zu-Eins-Beziehung zwischen Schablone und Instanz vorliegt. Die Beziehung kann 1:n sein. Die Definition einer Klasse ist daher die Definition eines wiederverwendbaren, benutzerdefinierten Modelltyps.

Die Instanzbildung einer Komponente erfolgt nur im Kontext eines Projekts. Daher wird, wenn nur mit Komponenten gearbeitet wird, automatisch ein Standardprojekt erzeugt, um den Kontext für die Instanzbildung der Komponenten zu liefern.

Wenn eine Klasse in einer anderen Komponente verwendet wird (siehe nachfolgender Abschnitt), erfolgt die Instanzbildung der Klasse im Kontext der betreffenden Komponente, wenn die Instanzbildung dieser Komponente erfolgt. Im Gegensatz hierzu erfolgt die Instanzbildung von Modulen in einem Projekt immer.

2.3 Die Schnittstelle von Komponenten

Die Schnittstelle einer Komponente besteht aus Methoden, Prozessen und dem Zugriff auf globale Variable. Module beispielsweise haben einen Zugriff auf Messages. Methoden und Prozesse sind auf dieselbe Weise strukturiert. Ihre Struktur ist unabhängig von der Art und Weise, wie die Methoden oder Prozesse beschrieben werden.

Jede Methode und jeder Prozess ist einem Diagramm zugeordnet, wobei jedes Diagramm entweder öffentlich oder privat sein kann. Methoden, die privaten Diagrammen zugeordnet sind, sind nur innerhalb der Komponente sichtbar und gehören nicht zu der öffentlichen Schnittstelle der Komponente, die für

andere Komponenten sichtbar ist. Alle Methoden, die einem Diagramm zugeordnet sind, werden in diesem Diagramm beschrieben (im Falle von Blockdiagrammen existiert ein gemeinsames Blockdiagramm für alle diese Methoden).

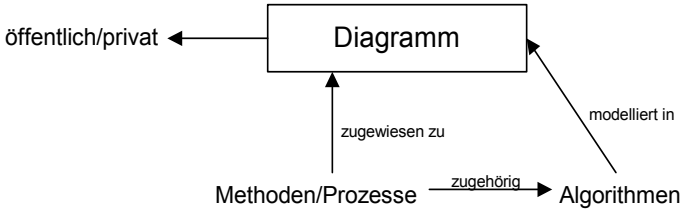


Abb. 2-4 Die Schnittstelle von Komponenten

2.3.1 Die Schnittstelle von Klassen

Die Schnittstelle einer Klasse besteht aus einer Anzahl von Methoden, die einem der Diagramme der Klasse zugeordnet sind. Die Schnittstelle jeder Methode besteht aus ihren Argumenten und einem Rückgabewert. Methoden sind Unterprogrammen ähnlich, die von einer beliebigen Stelle in der Software aus aufgerufen werden können. Jedoch bewirkt die Datenkapselung einer Klasse, d.h. der Zugriff auf dieselbe Menge von Instanzvariablen und –parametern, dass der Begriff der Methoden und Klassen wesentlich umfassender ist als der von Unterprogrammen. Methoden haben Zugriff auf alle Elemente, die in ihrer Klasse definiert sind.

Die Argumente und der Rückgabewert einer Methode können nur im Hauptteil der zugehörigen Methode verwendet werden. Außerdem verfügt jede Methode über eine Anzahl von methodenlokalen Variablen. Diese Variablen sind temporär und nicht statisch, und ebenso wie Argumente können sie nur im Hauptteil der zugehörigen Methode verwendet werden.

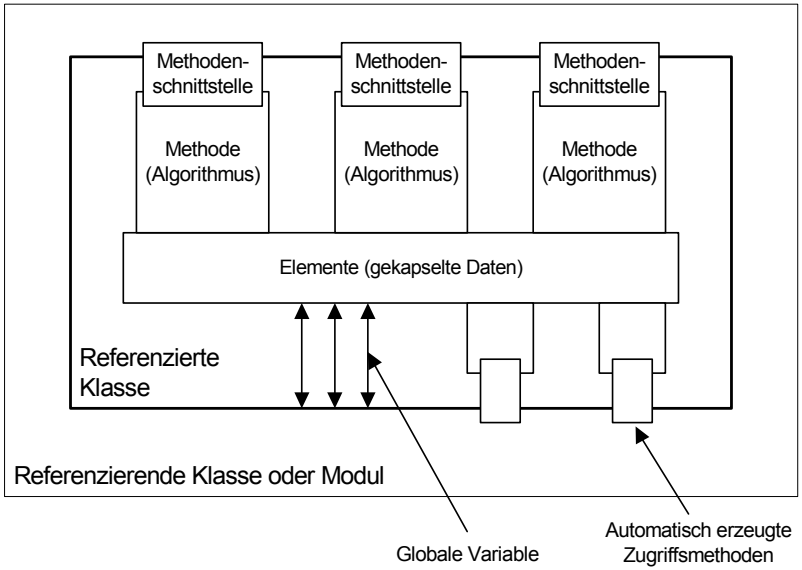


Abb. 2-5 Die Schnittstelle von Klassen

Für den direkten Zugriff auf die Instanzvariablen einer Klasse können zusätzliche Methoden verfügbar gemacht werden. Dieser Mechanismus ermöglicht es, Klassen als Datenbehälter zu verwenden (ähnlich wie Datensätze in C).

Die Wechselwirkung einer Klasse mit ihrer Umgebung besteht im Aufrufen der Methoden der Klasse. Wenn eine Methode aufgerufen wird, werden die Anweisungen im Hauptteil der Methode ausgeführt.

Die Methoden einer Klasse werden als öffentlich oder privat kategorisiert, indem ihnen ein öffentliches oder privates Diagramm zugeordnet wird. Öffentliche Methoden können von einer beliebigen Komponente aus aufgerufen werden, welche die betreffende Klasse benutzt. Private Methoden sind verborgen und können nur von Methoden derselben Klasse aufgerufen werden. Sie können als interne Unterprogramme verwendet werden.

2.3.2 Die Schnittstelle von Modulen

Die Schnittstelle eines Moduls besteht aus einer Anzahl von Prozessen und – optionalen – öffentlichen Methoden sowie den Messages, die in diesem Modul verwendet werden. Module treten auf zwei verschiedenen Ebenen in Wechselwirkung, da die Aktivierung von Prozessen und die Kommunikation über Messages getrennt erfolgen. Die Aktivierung des Prozesses wird vom Betriebssystem gesteuert (das Teil des Projekts ist).

Die Kommunikation zwischen Prozessen mittels Messages erfolgt asynchron bezüglich der Aktivierung der Prozesse, d.h. das Senden einer Message und das Empfangen derselben in einem Prozess erfolgen nicht zur gleichen Zeit. Dieses Konzept unterscheidet sich von der Übergabe von Parametern zwischen Methoden, welche synchron mit dem Aufrufen der Methode erfolgt.

Ebenso wie Methoden können Prozesse temporäre, prozesslokale Variable besitzen.

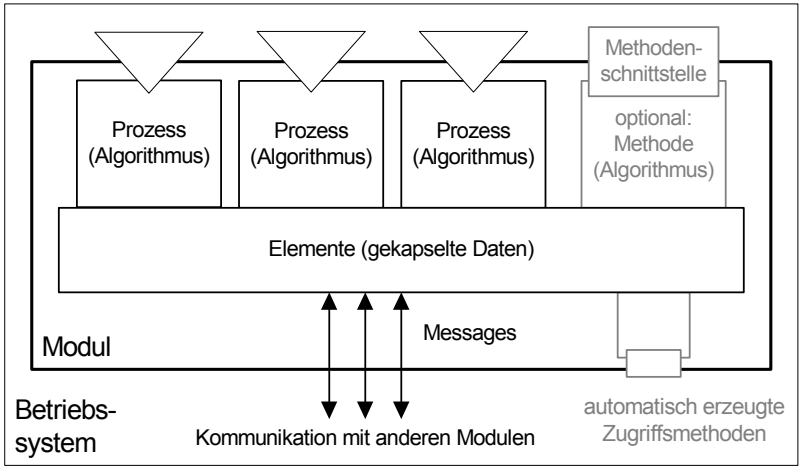


Abb. 2-6 Interprozesskommunikation (graue Teile sind optional)

2.4 Wiederverwendung von Komponenten

Beim Spezifizieren von Komponenten enthalten früher definierte Klassen oder Module Funktionalität, die wiederverwendet werden kann. Die Wiederverwendung von Komponenten führt zu einer hierarchischen, baumartigen Struktur einer Komponente. Die Blätter dieser Struktur sind Klassen oder Module, die keine anderen Klassen oder Module enthalten.

Die Struktur einer Komponente muss baumartig sein, d.h. zyklische Abhängigkeiten sind nicht gestattet. Das liegt daran, dass die Verwendungsbeziehung auch eine Einschlussbeziehung ist und eine zyklische Abhängigkeit nicht lösbar wäre.

Falls eine Klasse in einer anderen Komponente verwendet wird, so erfolgen die Instanzbildung und Initialisierung der Klasse automatisch, wenn die sie enthaltende Komponente initialisiert wird.

Es gibt jedoch eine Ausnahme. Bei Verwendung einer importierten Klasse, d. h. die Instanzbildung der Klasse erfolgte in einem bestimmten anderen Kontext, zum Beispiel direkt im Projekt, ist die Verwendungsbeziehung keine Einschluss- sondern eine Referenzbeziehung. In diesem Falle führt daher eine zyklische Abhängigkeit nicht zu einer unlösbaren Einschlussbeziehung.

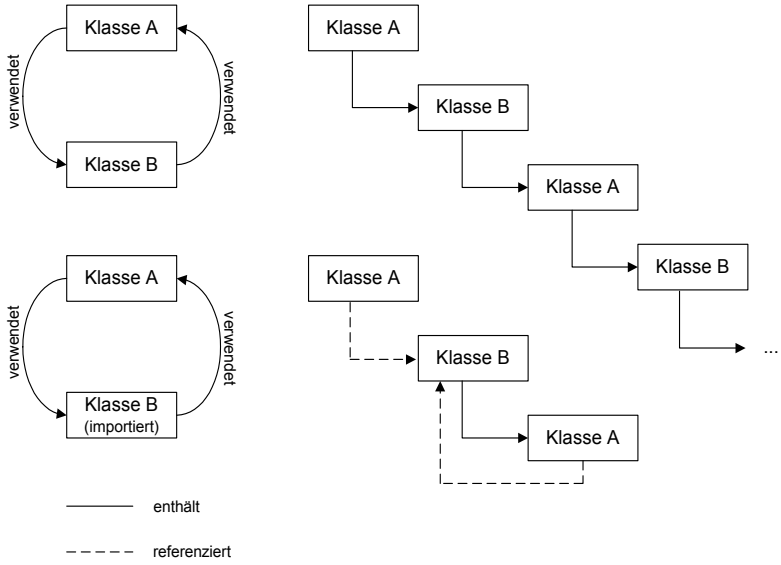


Abb. 2-7 Einschlussbeziehung und Referenz

Module sind die Komponente der obersten Ebene. Daher können Module nicht in Klassen enthalten sein. Klassen können jedoch in Modulen enthalten sein, ebenso wie in anderen Klassen. Es gelten die folgenden Beziehungen:

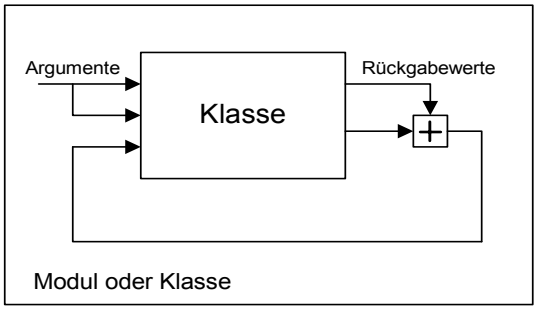
Einschlussbeziehungen	Klasse	Modul
Klasse	x	-
Modul	x	x

Tab. 2-2 Typologie von Beziehungen

Da die Schnittstellen von Modulen und Klassen unterschiedlich sind, ist die Bedeutung einer hierarchischen Modulstruktur und einer hierarchischen Klassenstruktur gleichfalls unterschiedlich.

2.4.1 Hierarchische Klassenstruktur

Wenn eine Klasse innerhalb einer anderen Komponente verwendet wird, können die Methoden der Klasse als Unterprogramme in der Komponente verwendet werden.



Methoden werden von der referenzierenden Komponente aktiviert

Abb. 2-8 Aufrufen von Methoden in einer verschachtelten Klasse

Die Methoden werden als Teil der Ausführung der Methoden oder des Prozesses der Komponente aufgerufen; diese Stelle in der Software kann von der Komponente selbst bestimmt werden. Wenn eine Methode aufgerufen wird, muss die Komponente der Methode aktuelle Parameter für die Argumente der Methode zur Verfügung stellen.

2.4.2 Hierarchische Modulstruktur

Wie bereits erwähnt, erfolgt bei Modulen eine Instanzbildung immer in einem Projekt. Das heißt, in einer hierarchischen Modulstruktur erfolgt bei einem Modul, das in einem anderen Modul verwendet wird, keine Instanzbildung innerhalb des dieses enthaltenden Moduls. Demzufolge befinden sich sämtliche Module, bei denen in einem Projekt eine Instanzbildung erfolgt, auf derselben Ebene, unabhängig von ihrer Position in der hierarchischen Struktur.

Die hierarchische Strukturierung von Modulen dient hauptsächlich zwei Zwecken. Eine hierarchische Struktur spiegelt die Natur eines Steuerungssystems wider. In einer Motorregelung beispielsweise können separate Module für Zündung, Einspritzung und Lambdaregelung vorhanden sein.

Außerdem kann die Kommunikationsstruktur in einem hierarchischen Modul viel transparenter gemacht werden, da der Datenfluss in Blockdiagrammen unmittelbar sichtbar ist.

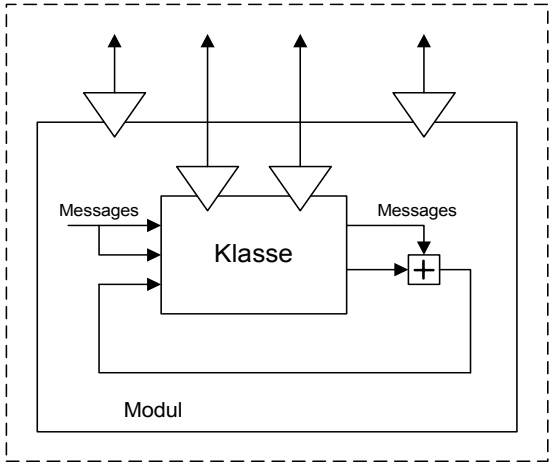


Abb. 2-9 Die Kommunikationsstruktur in einem hierarchischen Modul

Ein weiterer Vorteil einer hierarchischen Modulstruktur wird durch folgendes Beispiel klar: leichtere Instandhaltung. Wenn zum Beispiel der Name einer Message geändert wird, muss er in allen Modulen geändert werden, die diese Message verwenden. Wird statt dessen ein hierarchisches Modul verwendet, so betreffen die Änderungen nur ein Modul, da die auf dem Namen basierende Verknüpfung nicht explizit verwendet wird.

2.5 Zustandsautomaten

Ein Zustandsautomat ist ein spezieller Typ von Klassen, ein ereignisgesteuertes System, bei dem nicht Berechnungen, sondern der Kontrollfluss im Mittelpunkt steht. Daher beschreibt die Hauptebene der Beschreibung eines Zustandsautomaten, das Zustandsdiagramm, nicht, wie Daten übergeben werden, sondern wie die Kontrolle übergeben wird. Um den Kontrollfluss zu modellieren, besteht ein Zustandsautomat aus einer endlichen Anzahl von Zuständen und Übergängen zwischen diesen. Dazu kommt mindestens ein Trigger für die Steuerung des Zustandsautomaten. Bei einem Aufruf des Triggers wird ein Schritt des Zustandsautomaten abgearbeitet.

Weitere Informationen über die Theorie endlicher Zustandsautomaten finden Sie in

- Harel, David: „Statecharts: A Visual Formalism for Complex Systems“, *Science of Computer Programming 8*, 1987, S. 231-274

- Hatley, Derek J. & Imtiaz A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House Publishing Co., Inc., NY, 1988.

Die nachstehende Abbildung zeigt schematisch die Komponenten eines Zustandsautomaten.

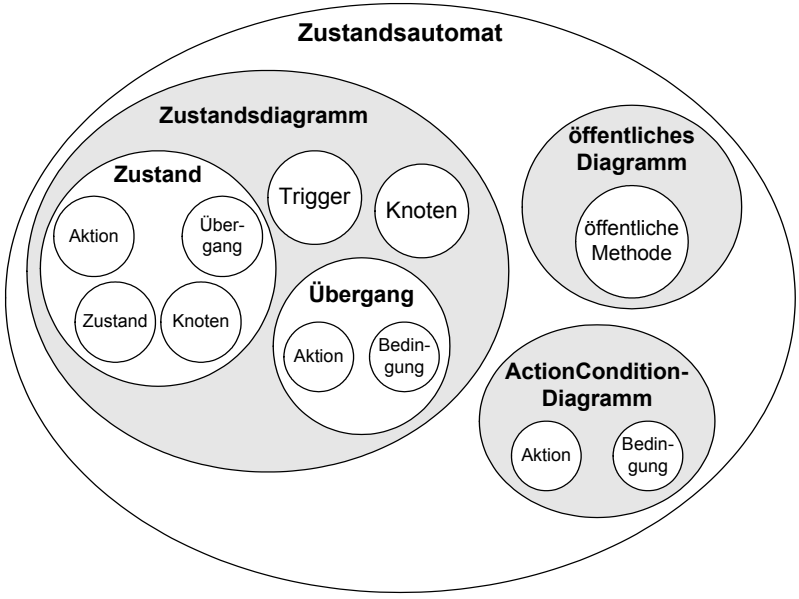


Abb. 2-10 Zustandsautomat – Schema

Das Spezifizieren eines Zustandsautomaten besteht im Festlegen der Zustände, in denen sich ein System befinden kann, im Definieren der Bedingungen, die erfüllt werden müssen, um von einem Zustand in einen anderen zu wechseln, und im Bestimmen der Aktionen, die während dieser Übergänge ausgeführt werden müssen.

Das Zustandsdiagramm ist ein spezielles Blockdiagramm zum Definieren eines Zustandsautomaten. Jeder Zustand wird grafisch als abgerundetes Rechteck dargestellt. Einer der Zustände muss stets als Anfangszustand gekennzeichnet sein; dies ist der Zustand, in dem sich der Automat zu Beginn befindet.

Die Übergänge sind gerichtete Bögen zwischen den Zuständen. Jeder Bogen steht für einen Übergang in einer Richtung, die durch eine Pfeilspitze an einem Ende angezeigt wird. Jedes Ende eines Übergangs ist mit einem Zustand verbunden. Der Zustand, in dem der Übergang beginnt, ist der Ausgangszustand, derjenige, in dem der Übergang endet, ist der Zielzustand. Um einen bidirektionalen Übergang zu modellieren, sind zwei Bögen erforderlich.

Das Beispieldiagramm enthält die wesentlichen grafischen Komponenten eines Zustandsautomaten.

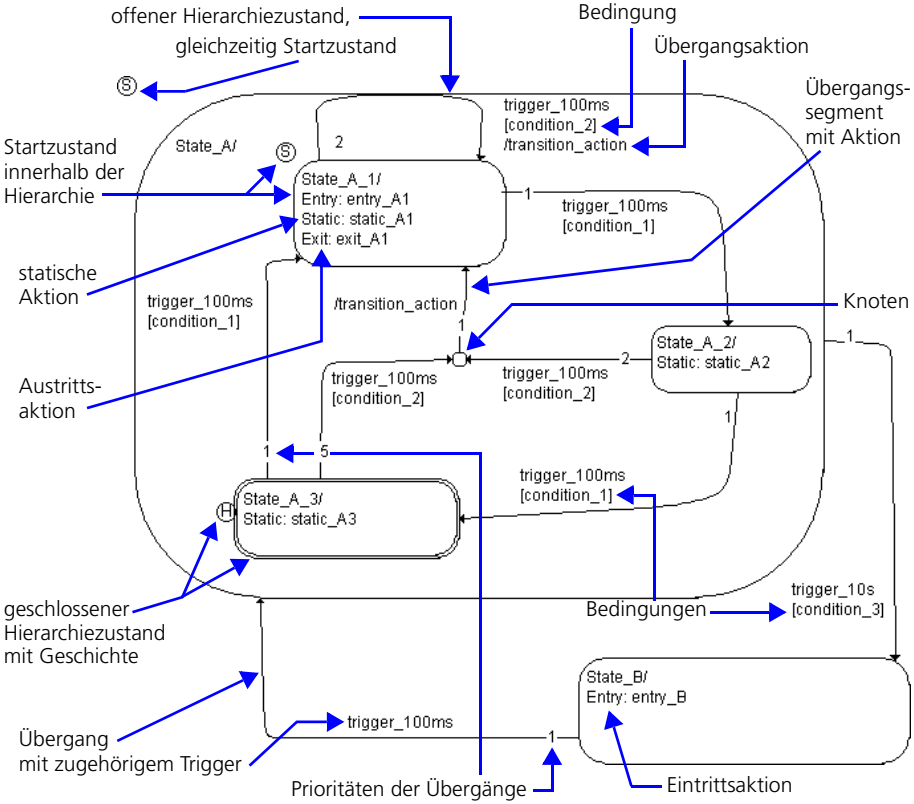


Abb. 2-11 Grafische Komponenten

2.5.1 Komponenten eines Zustandsautomaten

Zustände

Ein *Zustand* beschreibt einen Modus eines ereignisgetriebenen Systems. Die Aktivität oder Inaktivität der Zustände ändert sich dynamisch, abhängig von Triggerereignissen und Bedingungen.

Jeder Zustand hat einen übergeordneten Zustand (Hierarchiezustand, s. Seite 42). Für die Zustände auf der höchsten Ebene (`State_A` und `State_B` in Abb. 2-11) ist dies das Zustandsdiagramm selbst. Sie können Zustände innerhalb von anderen übergeordneten Zuständen platzieren; `State_A_1`,

`State_A_2` und `State_A_3` sind Unterzustände von `State_A`. Zustände, in denen keine weiteren Zustände enthalten sind, werden auch Basiszustände genannt (`State_A_1`, `State_A_2`, `State_B` in Abb. 2-11). Ein übergeordneter Zustand kann auch eine *Geschichte* (s. Seite 45) haben. Diese bietet ein geeignetes Mittel, zukünftige Aktivitäten auf vorherigen aufzubauen.

Die Zustände sind exklusiv, d.h. es können nicht mehrere Basiszustände gleichzeitig aktiv sein. Ist der aktive Basiszustand der Unterzustand einer Hierarchie, sind alle übergeordneten Zustände, die ihn enthalten, ebenfalls aktiv. Ist also der Zustand `State_A_2` in Abb. 2-11 aktiv, ist der Hierarchiezustand `State_A` ebenfalls aktiv. Wenn einer der (nicht sichtbaren) Unterzustände von `State_A_3` aktiv ist, sind `State_A_3` und `State_A` ebenfalls aktiv.

Jeder Zustand hat einen eindeutigen Namen. Identische Namen auch innerhalb verschiedener Hierarchien sind verboten; wenn Sie einen vorhandenen Namen erneut vergeben, wird das Kürzel `_n` angehängt. `n` ist dabei die kleinste für diesen Namen noch nicht vergebene Zahl (Zustände `State_A_1` bis `State_A_3` in Abb. 2-11). Ebenfalls verboten sind folgende Namen:

- Namen von Methoden, Prozessen, Elementen usw. im gesamten Projekt
- Namen aus dem C-Sprachumfang (z.B. `static`, `define`, usw.)

  Solche Namen für Zustände führen nicht immer zu einer Fehlermeldung, aber auf jeden Fall zu fehlerhaftem Code.

Neben den Namen beinhalten die Bezeichnungen der Zustände die verschiedenen *Aktionen* (s. Seite 46). Diese werden entsprechend ihrem Typ nacheinander abgearbeitet. Folgende Typen existieren: Eintrittsaktion, statische Aktion und Austrittsaktion. Alle Aktionen sind optional.

Übergänge

Ein *Übergang* ist ein grafisches Objekt, das zwei Zustände verbindet. Das eine Ende des Übergangs ist mit dem Ausgangszustand verbunden, dort beginnt der Übergang. Das andere Ende ist mit dem Zielzustand verbunden, hier endet der Übergang. Ein Übergang kann durch einen oder mehrere *Knoten* (Seite 39) unterbrochen und in Segmente geteilt werden.

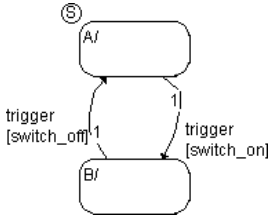
Jedem Übergang ist eine Priorität zugeordnet. Je höher der numerische Wert, desto höher ist die Priorität. Gehen von einem Zustand oder Knoten mehrere Übergange ab, werden diese in der Reihenfolge ihrer Priorität ausgewertet. Zwei Übergänge von demselben Zustand dürfen nicht dieselbe Priorität haben.

Die Bezeichnungen der Übergänge enthalten die Umstände, unter denen das System von einem Zustand in den nächsten übergeht. Ein Triggerereignis muss auftreten, damit ein Übergang stattfindet. Der Name des Triggers ist der erste Teil der Bezeichnung eines Übergangs. In Abb. 2-11 wird der Übergang von

`State_A_1` nach `State_A_2` durch den Trigger `trigger_100ms` angestoßen. Optional können die Übergänge auch eine *Bedingung* (s. Seite 45) und eine Aktion (Seite 46), die *Übergangsaktion*, enthalten. Diese werden im zweiten und dritten Teil der Bezeichnung angegeben. Bedingungen werden im Zustandsdiagramm in eckigen Klammern dargestellt, Übergangsaktionen mit einem vorangestellten „/". Wie Trigger, Bedingungen und Aktionen den Segmenten eines Übergangs mit Knoten zugeordnet werden, ist unter „Knoten" auf Seite 39 beschrieben.

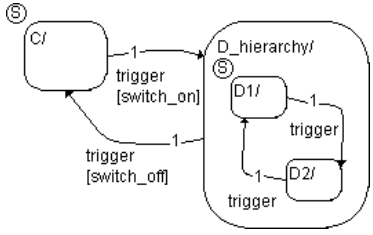
Ein Übergang ist gültig, wenn sein Anfangszustand aktiv und seine Bedingung – falls vorhanden – erfüllt ist. Es gibt mehrere Arten von Übergängen:

1. Übergänge zwischen Basiszuständen



   Der Übergang von Zustand `A` nach `B` ist gültig, wenn `A` aktiv ist, das Triggerereignis `trigger` eintritt und die Bedingung `[switch_on]` erfüllt ist.

2. Übergänge von und nach Hierarchiezuständen

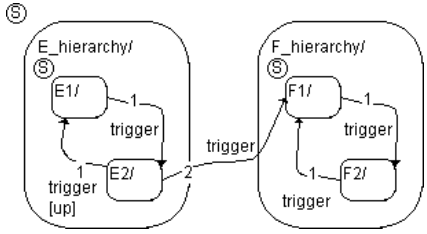


   Der Übergang von `C` in den Hierarchiezustand (s. Seite 42) `D_hierarchy` ist gültig, wenn `C` aktiv ist, das Triggerereignis `trigger` eintritt und die Bedingung `[switch_on]` erfüllt ist. Es handelt sich um einen expliziten Übergang in den Hierarchiezustand.

   Für einen gültigen Übergang in einen Hierarchiezustand muss ein Unterzustand implizit als Ziel definiert sein. Das ist hier dadurch gegeben, dass der Unterzustand `D1` als Startzustand (s. Seite 44) markiert ist. Ausgeführt wird also der Übergang von `C` nach `D1`.

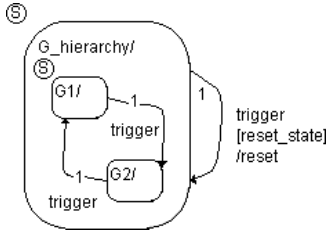
Der Übergang von `D_hierarchy` nach `C` ist gültig, wenn `D_hierarchy` aktiv ist, das Triggerereignis `trigger` eintritt und die Bedingung `[switch_off]` erfüllt ist, und zwar unabhängig davon, welcher der Unterzustände gerade aktiv ist.

3. Übergänge zwischen Unterzuständen verschiedener Hierarchien



Der Übergang vom Unterzustand `E2` im Hierarchiezustand `E_hierarchy` in den Unterzustand `F1` im Hierarchiezustand `F_hierarchy` ist gültig, wenn `E2` aktiv ist und das Triggerereignis `trigger` eintritt. Der Übergang definiert explizit das Verlassen des Unterzustands `E2` und implizit das Verlassen des Hierarchiezustands `E_hierarchy`. Ebenso definiert er implizit den Eintritt in `F_hierarchy` und explizit den Eintritt in `F1`.

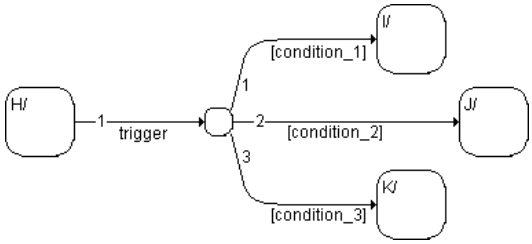
4. Schleifen



Die Schleife ist ein Übergang von einem Zustand in sich selbst. In der Abbildung ist der Übergang gültig, wenn einer der beiden Unterzustände von `G_hierarchy` aktiv ist, das Triggerereignis `trigger` eintritt und die Bedingung `[reset_state]` erfüllt ist. Das System verlässt den aktiven Unterzustand, den Zustand `G_hierarchy`, führt die Übergangsaktion aus, tritt wieder in `G_hierarchy` und schließlich in den Unterzustand `G1` ein.

5. Übergänge mit Knoten

   Alle Arten von Übergängen können Knoten (siehe folgender Abschnitt) beinhalten. Hier ist nur eins von vielen mögliche Beispielen dargestellt.



   Wenn der Zustand `H` aktiv ist und das Triggerereignis `trigger` eintritt, verlässt das System den Zustand `H`. Im Knoten werden die Bedingungen an den wegführenden Übergangssegmenten (`[condition_1]`, `[condition_2]`, `[condition_3]`) in der Reihenfolge ihrer Priorität geprüft. Ist z.B. die Bedingung `[condition_2]` erfüllt, findet der Übergang in den Zustand `J` statt. Ist keine der Bedingungen erfüllt, bleibt das System im Ausgangszustand `H`.

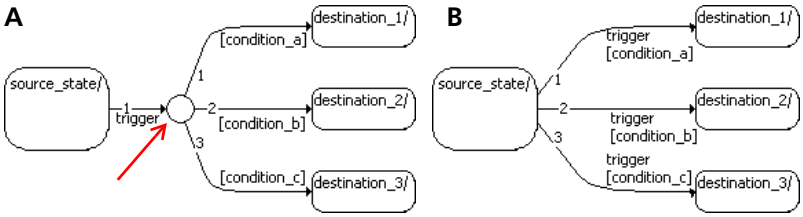
Knoten

Ein *Knoten* ist ein grafisches Objekt, das die Lesbarkeit von Zustandsdiagrammen deutlich verbessert und die Generierung von effizientem Code erleichtert. Knoten bilden weitere Möglichkeiten, das gewünschte Systemverhalten darzustellen.

Knoten sind keine Zustände, sie repräsentieren Verzweigungsstellen im Zustandsdiagramm. Knoten unterbrechen einen Übergang (s. Seite 36) und teilen ihn in Segmente. Ein Segment verbindet den Ausgangszustand mit dem Knoten, ein oder mehrere Segmente verbinden ggf. die unterbrechenden Knoten, und das letzte Segment verbindet den letzten Knoten mit dem Zielzustand. Knoten erleichtern so die Darstellung verschiedener Übergänge. Gleichzeitig ermöglichen sie die Wiederverwendung von Übergangssegmenten.

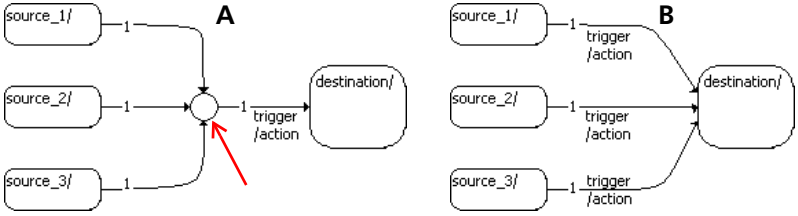
Folgendes ist bei der Verwendung von Knoten zu beachten:

- Übergänge von einem Ausgangszustand in mehrere Zielzustände werden übersichtlich dargestellt.



Dieselbe Funktionalität, die in Teil A der Abbildung mit einem Knoten modelliert wurde, kann auch durch direkte Übergänge vom Ausgangszustand `source_state` in die Zielzustände erreicht werden (Teil B der Abbildung). Die Verwendung des Knotens bringt jedoch einen Laufzeitvorteil, da zunächst nur das Übergangssegment zwischen Startzustand und Knoten ausgewertet wird. Ist dieses schon ungültig, kann kein Übergang stattfinden, und die vom Knoten wegführenden Segmente brauchen nicht betrachtet zu werden.

- Ebenso werden Übergänge von mehreren Ausgangszuständen in einen Zielzustand übersichtlich dargestellt.

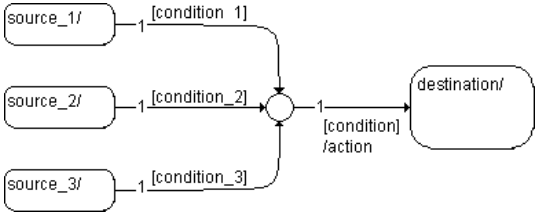


Auch in diesem Fall sind die beiden Schreibweisen gleichbedeutend. Eine allen drei Übergängen gemeinsame Aktion muss dem vom Knoten wegführenden Segment zugewiesen werden.

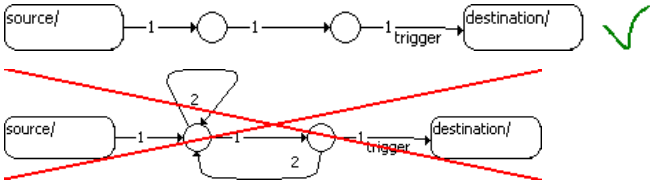
- Übergangssegmente von einem Knoten in einen Zustand können Aktionen enthalten (siehe vorige Abbildung, Teil A).

  Es ist nicht möglich, einem Übergangssegment, das in einem Knoten endet, eine Aktion zuzuweisen. Die komplexe Semantik solcher Übergangsaktionen hätte ineffizienten Code zur Folge.

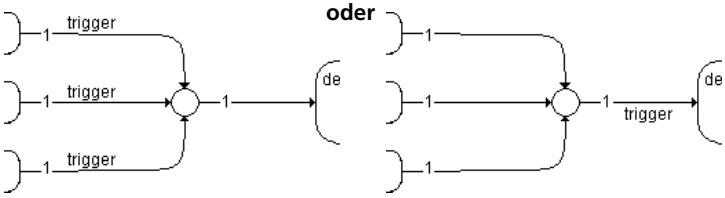
- Jedes Segment eines Übergangs kann eine Bedingung haben.



- Mehrere Knoten hintereinander (kaskadierende Knoten) sind erlaubt, Schleifen in jeder Form sind verboten.



- Nur ein Segment eines Übergangs hat einen Trigger. In der Regel wird ein Trigger entweder den in den ersten Knoten eines Übergangs führenden Segmenten oder den aus dem letzten Knoten wegführenden Segmenten zugewiesen, aber nicht allen.



Hinweis

Die Zuweisung von Triggern zu mehr als einem Segment desselben Übergangs ist nicht deaktiviert; allerdings gibt ASCET eine Fehlermeldung aus, wenn den Segmenten verschiedene Trigger zugewiesen sind.
Der Anwender ist daher für die korrekte Zuweisung von Triggern selbst verantwortlich.

- Wenn keins der möglichen Segmente, die in einen Zielzustand führen, gültig ist, findet kein Übergang statt, das System bleibt im Ausgangszustand.

Trigger

Trigger aktivieren die Ausführung des Zustandsautomaten: Jeder Triggeraufruf bewirkt, dass ein Schritt des Zustandsautomaten abgearbeitet wird. Sie sind öffentliche Methoden des Zustandsautomaten; jeder Trigger, der das Zustandsdiagramm beeinflusst, muss definiert werden. Für die Kommunikation mit anderen ASCET-Komponenten kann ein Trigger Argumente haben (siehe auch die Abschnitte „Zustandsautomaten als Klassen“ auf Seite 85 und „Der Zustandsautomaten-Editor“ im ASCET Benutzerhandbuch).

Ein Zustandsautomat kann einen oder mehrere Trigger haben. Jeder Übergang wird einem Trigger des Zustandsautomaten zugeordnet. Durch diese Zuordnung ist es möglich, mit Hilfe mehrerer Trigger verschiedene Unterzustandsautomaten zu definieren, die mit denselben Zuständen arbeiten. Alle Trigger können unabhängig voneinander aufgerufen werden. Bei jedem Aufruf eines Triggers wird der Zustandsautomat aktiviert: Alle Übergänge aus dem aktuellen Zustand, die dem aufgerufenen Trigger zugeordnet sind, werden in der Reihenfolge ihrer Priorität geprüft und gegebenenfalls wird ein Übergang ausgeführt.

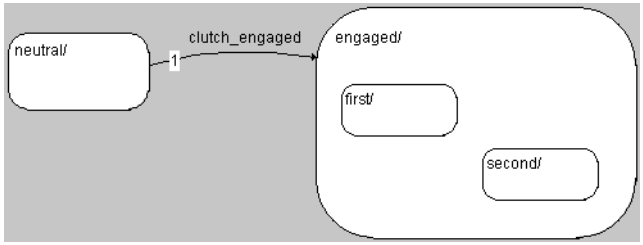
Hierarchie

Zustandsautomaten haben oft eine große Anzahl von Zuständen. Die *Hierarchie* erlaubt die Organisation komplexer Systeme, indem über- und untergeordnete Objektstrukturen definiert werden. Ein hierarchischer Aufbau reduziert üblicherweise die Anzahl der Übergänge und liefert strukturierte und übersichtliche Diagramme (siehe auch das Kapitel „Hierarchiezustände“ im ASCET Benutzerhandbuch).

ASCET unterstützt die hierarchische Organisation von Zuständen in Form von offenen und geschlossenen Hierarchien (`State_A` bzw. `State_A_3` in Abb. 2-11). Der Unterschied zwischen beiden besteht lediglich in der grafischen Darstellung.

Jeder Zustand kann seinerseits Zustände enthalten. Diese Zustände werden Hierarchiezustände genannt; Zustände, die keine anderen Zustände enthalten, werden Basiszustände genannt. Ein Zustand, der in einem Hierarchiezustand enthalten ist, wird Unterzustand des Hierarchiezustands genannt. Das System befindet sich immer in einem Basiszustand, und zusammen mit diesem Basiszustand auch in den zugehörigen Hierarchiezuständen.

Das hier dargestellte Zustandsdiagramm hat einen Hierarchiezustand, der zwei Unterzustände enthält. (Einige Übergänge sind der besseren Übersicht halber weggelassen.)



Der Hierarchiezustand `engaged` enthält die beiden Unterzustände `first` und `second`. Damit ist `engaged` der übergeordnete Zustand von `first` und `second`. Wenn das Triggerereignis `clutch_engaged` eintritt, geht das System vom Zustand `neutral` in den Hierarchiezustand `engaged` über.

Es sind aber auch sehr viel kompliziertere Strukturen möglich (s. Abb. 2-11). Hier folgt ein weiteres Beispiel eines hierarchischen Zustandsautomaten mit zwei hierarchischen Unterzuständen, von denen einer einen weiteren Hierarchiezustand enthält. Die Linien zwischen den Zuständen symbolisieren eine Einschlussbeziehung und dürfen nicht mit Übergängen verwechselt werden.

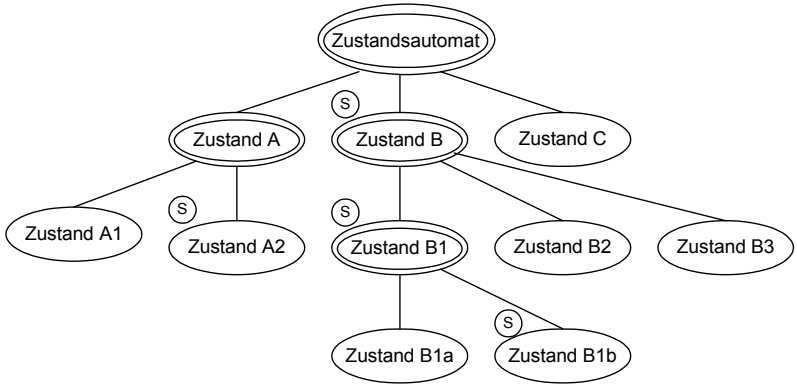


Abb. 2-12 Beziehungen innerhalb komplexer Zustandsautomaten

Innerhalb eines Hierarchiezustands bilden die Unterzustände ihrerseits einen Zustandsautomaten. Zum Beispiel bilden Zustand A1 und Zustand A2 einen Zustandsautomaten für sich. Zustände innerhalb eines Hierarchiezustands können Übergänge zu anderen Zuständen haben, welche sich nicht innerhalb desselben Hierarchiezustands befinden. Die Zustände sind durch Übergänge

verbunden; einer der Zustände ist als Startzustand im Hierarchiezustand gekennzeichnet. Zu Beginn befindet sich der hierarchische Zustandsautomat im Startzustand, und falls dieser Zustand ebenfalls hierarchisch ist, befindet er sich im Startzustand des Hierarchiezustands, usw.

In dem obigen Beispiel ist der Startzustand des Zustandsautomaten der Zustand B1b, da dies der Startzustand des Zustands B1 ist, welcher seinerseits der Startzustand von B ist. B wiederum ist der Startzustand auf der obersten Ebene.

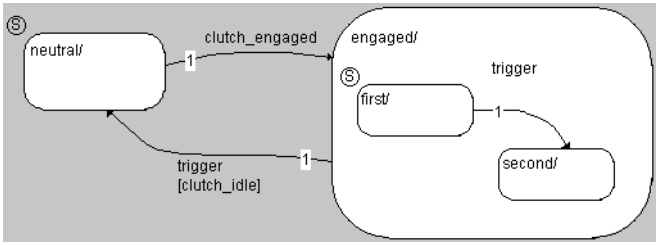
Ein Übergang heraus aus einem Hierarchiezustand beinhaltet automatisch das Verlassen des gerade aktiven Unterzustands. Ein Übergang von einem Unterzustand kann über die Grenzen von Hierarchiezuständen hinweg zu einem anderen Unterzustand führen. Wenn ein Unterzustand aktiv ist, ist dessen übergeordneter Hierarchiezustand ebenfalls aktiv.

Startzustand

Der *Startzustand* gibt an, welcher Zustand aktiviert wird, wenn es auf derselben hierarchischen Ebene mehrere Möglichkeiten gibt. Auf diese Weise wird der Anfangszustand des Zustandsautomaten insgesamt oder einer hierarchischen Ebene bestimmt.

Ein häufiger Fehler beim Spezifizieren von Zustandsautomaten ist, dass mehrere Zustände erzeugt werden, ohne dass einer als Startzustand markiert wird. In dem Fall gibt es keinerlei Hinweis, welcher der Zustände standardmäßig aktiviert wird. ASCET gibt daher bei der Erzeugung von Code eine entsprechende Fehlermeldung.

In dem hier abgebildeten Zustandsdiagramm ist der Zustand `neutral` als Startzustand des Gesamtsystems definiert, `first` als Startzustand innerhalb des Hierarchiezustands `engaged`.

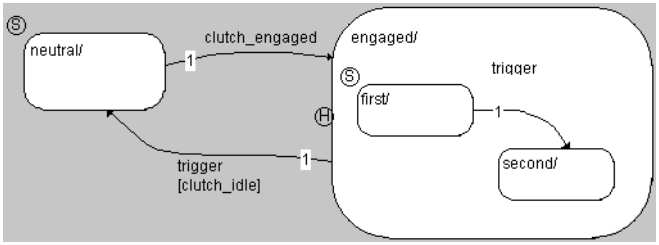


So wird bei der ersten Aktivierung des Zustandsautomaten der Zustand `neutral` aktiviert. Wäre kein Startzustand definiert, wäre unklar, ob nun `neutral` oder `engaged` aktiviert werden soll. Wenn ein Übergang von `neutral` nach `engaged` stattfindet, wird innerhalb des Hierarchiezustands der Unterzustand `first` aktiviert.

Geschichte

Die Option *Geschichte* bietet die Möglichkeit, das Ziel für einen Übergang in einen Hierarchiezustand anhand der vorangegangenen Aktivitäten festzulegen. Wenn ein Hierarchiezustand eine Geschichte hat, dann endet der Übergang in demjenigen Unterzustand, der zuletzt aktiv war.

Die Geschichte gehört zu dem Hierarchiezustand, in dem die Option gesetzt wurde. Sie hat Vorrang vor dem Startzustand innerhalb des Hierarchiezustands.

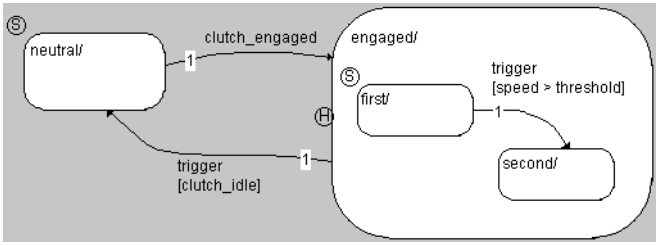


Das H in der Abbildung zeigt an, dass der Hierarchiezustand `engaged` eine Geschichte hat. Ob bei einem Übergang von `neutral` nach `engaged` der Zustand `first` oder `second` erreicht wird, hängt also davon ab, welcher von beiden zuletzt aktiv war.

Im generierten Code wird eine eigene Variable für die Geschichte angelegt, die *History-Variable*.

Bedingungen

Eine *Bedingung* ist ein Boolescher Ausdruck, der spezifiziert, dass ein Übergang stattfindet, wenn der Ausdruck wahr (`true`) ist. Jeder Übergang und jedes Übergangssegment kann eine Bedingung haben. In Abb. 2-11 repräsentiert die Bedingung `[condition_3]` einen Booleschen Ausdruck, der den Wert `true` haben muss, damit der Übergang von `State_A` nach `State_B` stattfinden kann.



Im dargestellten System findet der Übergang von `first` nach `second` statt, wenn die Boolesche Bedingung `[speed > threshold]` erfüllt ist.

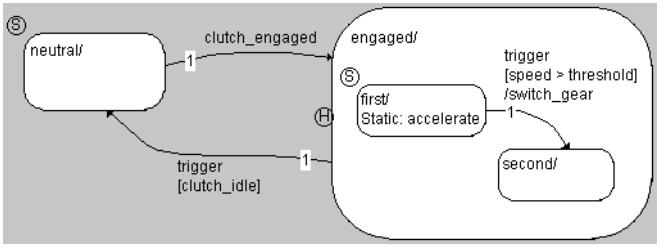
Bedingungen können als Blockdiagramme (in separaten Diagrammen) oder in ESDL (in separaten Diagrammen oder direkt am Übergang) spezifiziert werden. Mehr dazu finden Sie im Kapitel „Spezifizieren von Bedingungen und Aktionen“ im ASCET Benutzerhandbuch.

Für die Kommunikation mit anderen ASCET-Komponenten können Bedingungen auch Argumente haben. Mehr dazu finden Sie im Abschnitt „Zustandsautomaten als Klassen“ auf Seite 85 und im ASCET Benutzerhandbuch, Abschnitt „Kommunikation mit anderen Komponenten“.

Aktionen

Aktionen finden statt als Teil der Ausführung eines Zustandsdiagramms. Eine Aktion wird entweder als Teil eines Übergangs zwischen zwei Zuständen ausgeführt (z.B. `/transition_action` in Abb. 2-11) oder entsprechend der Aktivität eines Zustands (z.B. `static_A2` oder `exit_A1` in Abb. 2-11).

Übergänge sowie von einem Knoten wegführende Übergangssegmente können *Übergangsaktionen* haben, Zustände können *Eintritts-*, *statische* und *Austrittsaktionen* haben. Alle Aktionen sind optional. In Abb. 2-11 auf Seite 35 hat der Zustand `State_A_1` alle drei Aktionsarten, der Zustand `State_A_2` hingegen nur eine statische Aktion, aber weder Eintritts- noch Austrittsaktion. Der Übergang von `State_A_1` nach `State_A_2` hat keine Aktion.



Wenn im Beispiel hier der Zustand `first` aktiv ist und kein Übergang stattfindet, wird die statische Aktion `accelerate` ausgeführt. Beim Übergang von `first` nach `second` wird die Übergangsaktion `switch_gear` ausgeführt.

Wann welche Aktion ausgeführt wird, ist in den Abschnitten „Semantik: Einfache Zustandsautomaten“, „Semantik: Knoten in Zustandsautomaten“ und „Semantik: Hierarchische Zustandsautomaten“ im Detail beschrieben. Aktionen können als Blockdiagramme (in separaten Diagrammen) oder in ESDL (in separaten Diagrammen oder direkt an der Aktion/dem Übergang) spezifiziert werden. Mehr dazu finden Sie im ASCET Benutzerhandbuch, Kapitel 4.2.3 „Spezifizieren von Bedingungen und Aktionen“.

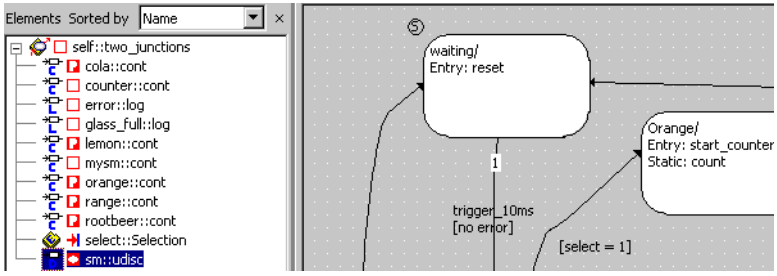
Für die Kommunikation mit anderen ASCET-Komponenten können Aktionen auch Argumente haben. Mehr dazu finden Sie im Abschnitt „Zustandsautomaten als Klassen“ auf Seite 85 und im ASCET Benutzerhandbuch, Abschnitt „Kommunikation mit anderen Komponenten“.

Daten

Datenobjekte werden verwendet, um numerische Werte im Zustandsdiagramm zu speichern und zu verarbeiten. Folgende Typen stehen zur Verfügung:

- Variable, Parameter, Konstanten (s. Seite 99)
- Enumerationen (s. Seite 98)
- Arrays, Matrizen (s. Seite 94, 94)
- Literale (s. Seite 99)
- temporäre Variablen (s. Seite 101)
- Kennlinien und -Felder (s. Seite 95)
- Eingänge für Daten von anderen ASCET Komponenten
- Ausgänge zu anderen ASCET Komponenten
- andere Klassen (z.B. Timer, Zähler, Komparatoren)

Zu den Daten gehört auch die Zustandsvariable `sm` vom Typ `unsigned discrete`, die in jedem Zustandsautomaten angelegt wird.



Diese Variable enthält die Nummer des gerade aktiven Zustands. Er kann im Zustandsautomaten-Editor nicht bearbeitet werden, lässt sich aber im Experiment messen. Wird für ein Projekt, das einen Zustandsautomaten beinhaltet, eine ASAM-MCD-2CM-Datei erzeugt, wird der Parameter `sm` mit in die Datei geschrieben.

2.5.2 Semantik von Zustandsautomaten

Ein Zustandsautomat besteht aus einer endlichen Anzahl von Zuständen. Jeder Zustand repräsentiert einen Zustand, in dem sich ein System befinden kann, zum Beispiel ob eine Tür verriegelt, geöffnet oder geschlossen ist. Unter bestimmten Umständen ändert sich der Zustand des Systems. Diese Zustandsänderungen werden durch Übergänge zwischen den verschiedenen Zuständen modelliert. Für jeden möglichen Übergang, der stattfinden soll, muss eine Bedingung erfüllt werden.

Ein Zustandsautomat wird durch ein äußeres Ereignis aktiviert, das Triggerereignis. Ein Trigger ist eine öffentliche Methode des Zustandsautomaten. Der Zustandsautomat muss sich immer in einem seiner Zustände befinden. Zu Beginn befindet sich ein Zustandsautomat in einem speziellen Zustand, dem Startzustand. Tritt ein Triggerereignis auf, so reagiert das System mit der Ausführung von Aktionen (z. B. Erzeugung eines Signals, Ändern eines Variablenwerts oder dem Übergehen in einen anderen Zustand).

Die *Eintrittsaktion* eines Zustands wird ausgeführt, wenn ein Übergang in diesen Zustand stattfindet. Der Zustand wird aktiviert, ehe die Ausführung der Eintrittsaktion gestartet wird.

Hinweis

Wenn der Zustandsautomat das erste Mal aufgerufen wird, wird die Eintrittsaktion des Startzustands ***nicht*** *ausgeführt.*

Die *statische Aktion* eines Zustands wird ausgeführt, wenn der Zustand aktiv ist und ein Triggerereignis eintritt, das keinen Übergang aus dem Zustand heraus zur Folge hat. Bei einem Übergang zwischen zwei Unterzuständen desselben Hierarchiezustands führt der Hierarchiezustand (der ja nicht verlassen wird) seine statische Aktion aus und beendet sie, nachdem der Ausgangszustand verlassen wurde, aber bevor die Übergangsaktion ausgeführt wird.

Die *Austrittsaktion* eines Zustands wird ausgeführt, wenn ein Übergang aus diesem Zustand heraus stattfindet. Der Zustand wird inaktiv, wenn die Ausführung der Austrittsaktion beendet ist.

Die *Übergangsaktion* eines Übergangs wird ausgeführt, nachdem der Ausgangszustand verlassen und deaktiviert wurde und bevor der Zielzustand aktiviert wird.

Die Semantik beschreibt, wie ein Zustandsdiagramm interpretiert und ausgeführt wird und in welcher Reihenfolge dabei die Aktionen durchgeführt werden. Die Kenntnis der Semantik von Zustandsdiagrammen ist für die Erstellung geeigneter Zustandsautomaten und die Generierung von effizientem Code unerlässlich. Verschiedene Realisierungsmöglichkeiten ergeben unterschiedliches Verhalten bei der Simulation und im ausführbaren Code.

Die Semantik von Zustandsautomaten enthält Regeln für die

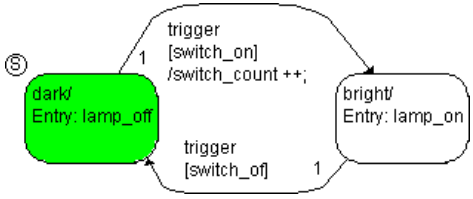
- Abarbeitung von Zuständen,
- Auswahl von Übergängen,
- Abarbeitung von Übergängen.

Die Semantik von Zustandsautomaten wird in den folgenden Abschnitten anhand von Beispielen erklärt. Diese decken einen großen Bereich möglicher Realisierungen und Kombinationen der verschiedenen Aktionen ab.

Eine Zusammenfassung der Regeln finden Sie im Abschnitt „Semantik: Zusammenfassung“ auf Seite 69.

2.5.3 Semantik: Einfache Zustandsautomaten

Beispiel 1: Übergang zwischen zwei Zuständen.



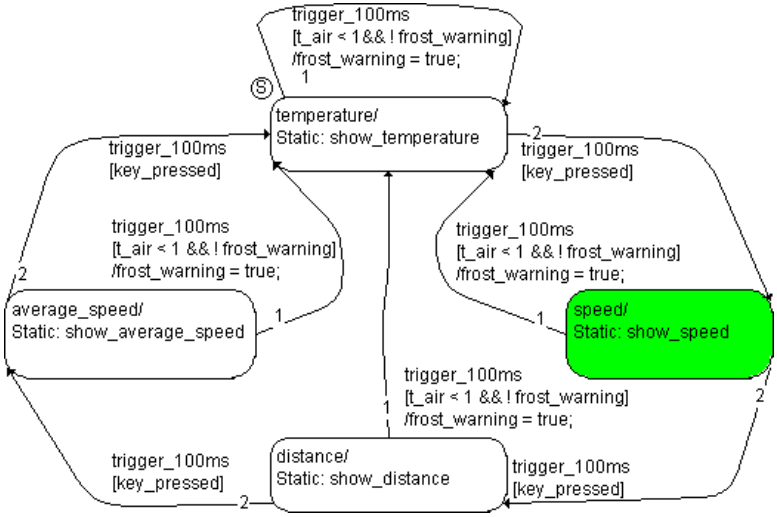
Dieser einfache Zustandsautomat modelliert einen Lichtschalter. Anfangs ist die Lampe aus, der Zustand `dark` ist aktiv. Das Triggerereignis `trigger` tritt ein und stößt die Auswertung des Zustandsautomaten an. Der Lichtschalter wird betätigt, sodass die Bedingung `switch_on` wahr ist. Folgende Schritte werden ausgeführt:

1. Das Zustandsdiagramm prüft, ob es einen gültigen Übergang gibt.
2. Der Zustand `dark` ist aktiv, also kommt nur der Übergang von `dark` nach `bright` in Frage. Da die Bedingung [`switch_on`] erfüllt ist, ist der Übergang gültig.
3. Der Zustand `dark` hat keine Austrittsaktion, die ausgeführt werden kann. Er wird deaktiviert.
4. Die Übergangsaktion wird durchgeführt, der Zähler `switch_count` wird um 1 erhöht.
5. Der Zustand `bright` wird aktiviert.
6. Die Eintrittsaktion `lamp_on` wird ausgeführt und beendet. Die Lampe ist eingeschaltet.

   Damit ist die Auswertung des Zustandsautomaten, die durch dieses Triggerereignis angestoßen wurde, beendet.

Jeder Zustand kann Übergänge zu mehr als einem anderen Zustand haben. Um das Verhalten des Zustandsautomaten deterministisch zu machen, muss jedem Übergang eine Priorität zugeordnet werden. Durch die Priorität wird die Reihenfolge festgelegt, in der die zu den Übergängen gehörenden Bedingungen geprüft werden. Sobald die Auswertung einer Bedingung den Wert `true` (wahr) ergibt, findet der zugehörige Übergang statt und alle anderen Bedingungen, die zu Übergängen mit niedrigeren Prioritäten gehören, werden nicht geprüft. Wenn die Auswertung bei keiner Bedingung „wahr" ergibt, bleibt der Zustand unverändert und es wird die statische Aktion ausgeführt.

Beispiel 2: Mehrere mögliche Übergänge aus einem Zustand



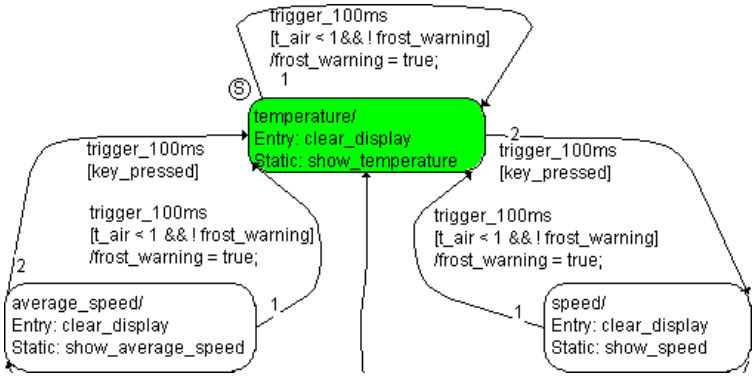
Dieser Zustandsautomat modelliert ein Display. Außentemperatur, Geschwindigkeit, Durchschnittsgeschwindigkeit oder gefahrene Strecke können wahlweise angezeigt werden. Daneben gibt es eine Taste, mit der die Display-Anzeige umgeschaltet wird. Fällt die Außentemperatur unter 1°C, findet ein Wechsel zur Temperaturanzeige statt und eine Eiswarnung wird angezeigt.

Der Zustandsautomat befindet sich im Zustand `speed`. Ein Triggerereignis `trigger_100ms` tritt ein; die Temperatur fällt von 1,5°C auf 0,5°C. Der Schalter wird nicht betätigt. Folgende Schritte werden ausgeführt:

1. Das System prüft, ob ein gültiger Übergang aus `speed` vorhanden ist.
2. Der Übergang von `speed` nach `distance` hat die höhere Priorität, wird also zuerst untersucht. Die Bedingung `[key_pressed]` ist jedoch nicht erfüllt, also ist der Übergang ungültig.

3. Der Übergang von `speed` nach `temperature` hat die Bedingung `[t_air < 1 && !frost_warning]`. Die Temperatur war anfangs oberhalb der Schwelle von 1°C, es lag also keine Notwendigkeit für eine Glatteiswarnung vor. Jetzt ist sie aber auf 0,5°C gefallen. Beide Teile der Bedingung sind also wahr, der Übergang ist gültig.
4. Der Zustand `speed` hat keine Austrittsaktion. Er wird deaktiviert.
5. Die Übergangsaktion `/frost_warning = true` wird ausgeführt und die Eiswarnung angezeigt.
6. Der Zustand `temperature` wird aktiviert.
7. Da dieser Zustand keine Eintrittsaktion hat, ist die Auswertung des Zustandsautomaten, die durch dieses Triggerereignis angestoßen wurde, damit beendet.

Beispiel 3: Schleife



Der Zustandsautomat ist derselbe wie in Beispiel 2, jedoch wurde den Zuständen die Eintrittsaktion `clear_display` hinzugefügt. Der Zustandsautomat befindet sich im Zustand `temperature`, ansonsten ist der Ausgangszustand derselbe wie im vorigen Beispiel. Ein Triggerereignis `trigger_100ms` tritt ein, der Schalter wird nicht gedrückt. Folgende Schritte werden ausgeführt:

1. Das System prüft, ob ein gültiger Übergang aus `temperature` vorhanden ist.
2. Der Übergang von `temperature` nach `speed` hat die höhere Priorität, aber die Bedingung ist nicht erfüllt. Der Übergang ist ungültig.
3. Der Übergang von `temperature` nach sich selbst hat die Bedingung `[t_air < 1 && !frost_warning]`. Diese ist erfüllt, der Übergang ist gültig.

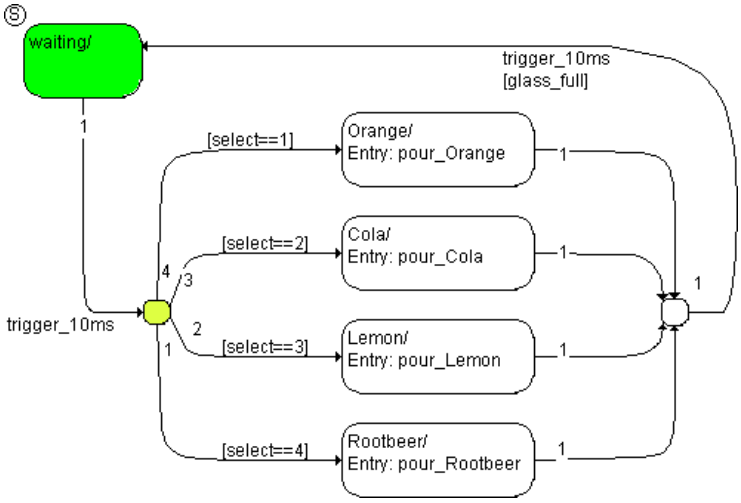
4. Der Zustand `temperature` hat keine Austrittsaktion. Er wird deaktiviert.
5. Die Übergangsaktion `/frost_warning = true` wird ausgeführt und die Eiswarnung angezeigt.
6. Der Zustand `temperature` wird aktiviert.
7. Die Eintrittsaktion `clear_display` des Unterzustands `temperature` wird ausgeführt und beendet.

   Damit ist die Auswertung des Zustandsautomaten, die durch das Triggerereignis `trigger_100ms` angestoßen wurde, beendet.

2.5.4 Semantik: Knoten in Zustandsautomaten

Knoten (s. Seite 39) dienen der Lesbarkeit der Zustandsdiagramme. Die Funktionalität aller Beispiele lässt sich ebenso mit direkten Übergängen zwischen den Zuständen beschreiben.

Beispiel 4: If…Then…Else-Konstruktion



Dieser Zustandsautomat modelliert einen einfachen Getränkeautomaten, der vier verschiedene Getränke anbietet. Der Zustandsautomat befindet sich im Zustand `waiting`. Ein Triggerereignis `trigger_10ms` tritt ein, jemand möchte Cola trinken und setzt die Auswahl `select` auf 2. Folgende Schritte werden ausgeführt:

1. Das System prüft, ob ein gültiger Übergang oder ein gültiges Segment aus `waiting` vorhanden ist.
   Das Übergangssegment von `waiting` in den linken Knoten ist gültig.

2. Die Übergangssegmente vom Knoten weg werden in der Reihenfolge ihrer Prioritäten untersucht, beginnend mit dem Segment vom Knoten in den Zustand `Orange`.
   Die Bedingung `[select==1]` ist nicht erfüllt, das Segment ist ungültig.
3. Als nächstes wird das Segment vom Knoten in den Zustand `Cola` geprüft.

   Die Bedingung `[select==2]` ist erfüllt, das Segment ist gültig. Damit ist ein vollständiger gültiger Übergang aus dem Zustand `waiting` vorhanden.
4. Erst jetzt findet der Übergang statt. Der Zustand `waiting` hat keine Austrittsaktion und wird deaktiviert.
5. Der Zustand `Cola` wird aktiviert.
6. Die Eintrittsaktion `pour_Cola` wird ausgeführt und beendet.

   Damit ist die Auswertung des Zustandsautomaten, die durch dieses Triggerereignis angestoßen wurde, beendet.

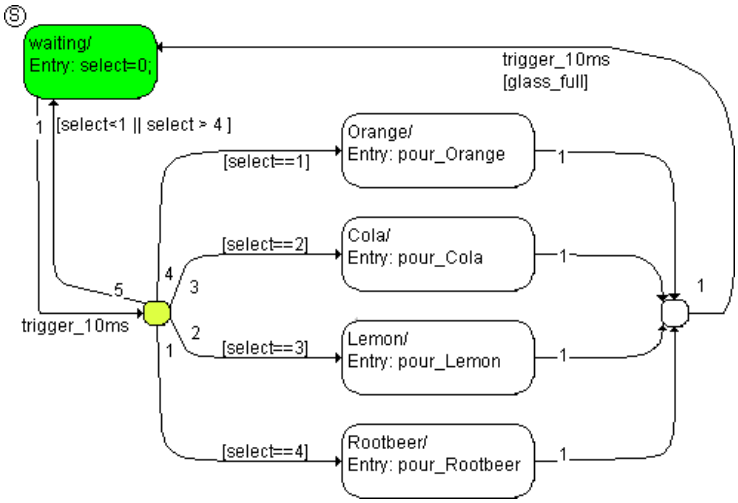
Beispiel 5: Kein Übergang

Der Zustandsautomat ist derselbe wie in Beispiel 4. Das System befindet sich im Zustand `waiting`. Ein Triggerereignis `trigger_10ms` tritt ein, die Auswahl `select` wird irrtümlich auf 5 gesetzt. Folgende Schritte werden ausgeführt:

1. Das System prüft, ob ein gültiger Übergang oder ein gültiges Segment aus `waiting` vorhanden ist.
   Das Übergangssegment von `waiting` in den linken Knoten ist gültig.
2. Die Übergangssegmente vom Knoten weg werden in der Reihenfolge ihrer Prioritäten untersucht.
   Da `select` auf 5 gesetzt wurde, ist keine der Bedingungen erfüllt, alle Segmente sind ungültig.
3. Es gibt keinen gültigen Übergang aus `waiting`, das System bleibt im Zustand `waiting`. Da der Zustand keine statische Aktion hat, passiert nichts.

   Damit ist die Auswertung des Zustandsautomaten, die durch dieses Triggerereignis angestoßen wurde, beendet.

Beispiel 6: Schleifenkonstruktion



Der Zustandsautomat ist derselbe wie in Beispiel 4. Neu hinzugekommen ist jetzt ein Übergangssegment vom Knoten zurück in den Zustand `waiting` sowie die Eintrittsaktion in `waiting`.

Der Zustandsautomat befindet sich im Zustand `waiting`; ein Triggerereignis `trigger_10ms` tritt ein. Irrtümlich wird die Auswahl `select` auf 5 gesetzt. Folgende Schritte werden ausgeführt:

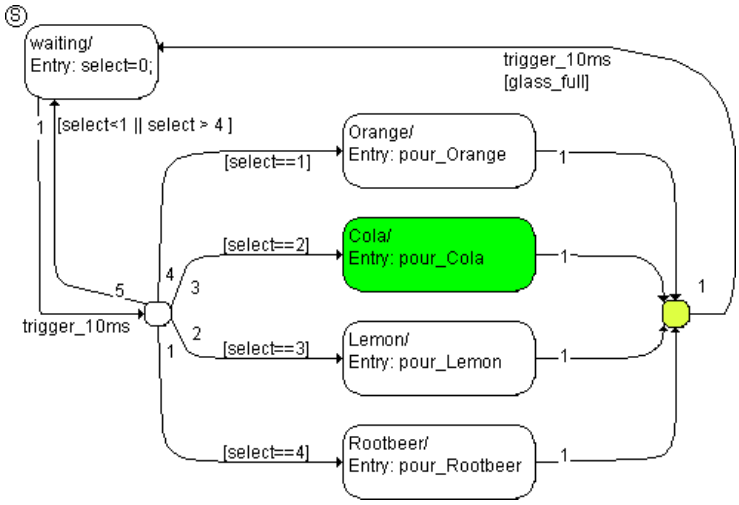
1. Das System prüft, ob ein gültiger Übergang oder ein gültiges Segment aus `waiting` vorhanden ist.
   Das Übergangssegment von `waiting` in den linken Knoten ist gültig.
2. Die Übergangssegmente vom Knoten weg werden in der Reihenfolge ihrer Prioritäten untersucht, beginnend mit dem Segment vom Knoten zurück in den Zustand `waiting`.

   Die Bedingung `[select<1 || select > 4]` ist erfüllt, das Segment ist gültig. Damit ist ein vollständiger gültiger Übergang aus dem Zustand `waiting` vorhanden.
3. Der Zustand `waiting` hat keine Austrittsaktion. Er wird deaktiviert.
4. Der Übergang von `waiting` in `waiting` hat keine Übergangsaktion, daher wird der Zustand `waiting` wieder aktiviert.
5. Die Eintrittsaktion `select=0;` von `waiting` wird ausgeführt und abgeschlossen.

   Damit ist die Auswertung des Zustandsautomaten, die durch dieses Triggerereignis angestoßen wurde, beendet.

Diese Schleifenkonstruktion entspricht dem direkten Übergang von einem Zustand in sich selbst aus Beispiel 3.

Beispiel 7: Übergänge von mehreren Startzuständen in einen Zielzustand (ein Trigger)

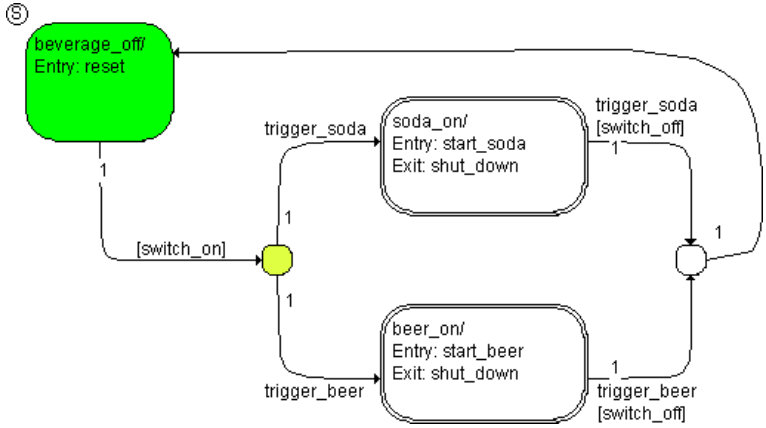


Der Zustandsautomat ist derselbe wie in Beispiel 6. Der Zustand `Cola` ist aktiv, das Glas ist bereits gefüllt und die logische Variable `glass_full` wurde auf `true` gesetzt. Ein Triggerereignis `trigger_10ms` tritt ein; folgende Schritte werden ausgeführt:

1. Das System prüft, ob ein gültiger Übergang oder ein gültiges Segment aus `Cola` vorhanden ist.
   Das Übergangssegment von `Cola` in den rechten Knoten ist gültig.
2. Das Übergangssegment vom Knoten in den Zustand `waiting` hat die Bedingung `[glass_full]`. Da `glass_full` auf `true` gesetzt wurde, ist auch dieses Segment gültig, und der Übergang kann stattfinden.
3. Der Zustand `Cola` hat keine Austrittsaktion. Er wird deaktiviert.
4. Der Übergang hat keine Übergangsaktion, daher wird als nächstes der Zustand `waiting` aktiviert.
5. Die Eintrittsaktion `select=0;` von `waiting` wird ausgeführt und abgeschlossen.

   Damit ist die Auswertung des Zustandsautomaten, die durch dieses Triggerereignis angestoßen wurde, beendet.

Beispiel 8: Übergänge aus einem Ausgangszustand in verschiedene Zielzustände (mehrere Trigger)



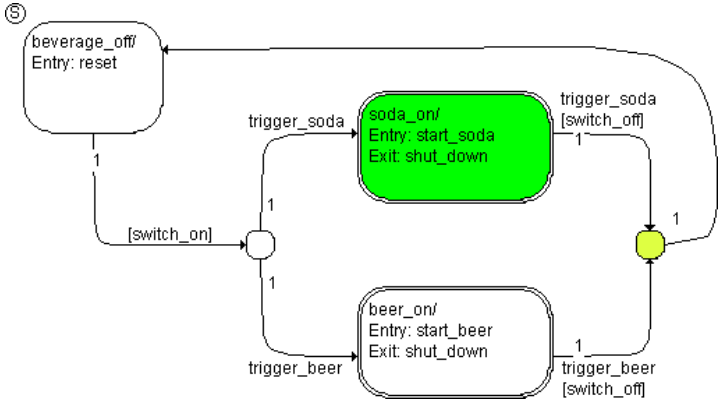
Dieser Zustandsautomat beschreibt einen Getränkeautomaten, der wahlweise verschiedene Limonaden- oder Biersorten anbietet. Die eigentliche Auswahl findet in den Hierarchiezuständen `soda_on` und `beer_on` statt; sie ist für das Beispiel irrelevant. Die Semantik von hierarchischen Zustandsautomaten wird im Abschnitt „Semantik: Hierarchische Zustandsautomaten" beschrieben.

Der Zustandsautomat befindet sich im Startzustand `beverage_off`. Ein Triggerereignis `trigger_soda` tritt ein und die Maschine wird eingeschaltet (`switch_on` ist `true`). Folgende Schritte werden ausgeführt:

1. Das System prüft, ob ein gültiger Übergang oder ein Segment aus `beverage_off` vorhanden ist.
2. Das Übergangssegment von `beverage_off` in den Knoten ist gültig, da die Bedingung `[switch_on]` erfüllt ist. Da das Triggerereignis `trigger_soda` eingetreten ist, ist auch das Segment vom Knoten in den Zustand `soda_on` gültig; der Übergang kann stattfinden.
3. Der Zustand `beverage_off` hat keine Austrittsaktion. Er wird deaktiviert.
4. Der Übergang von `beverage_off` in `soda_on` hat keine Übergangsaktion, also wird als nächstes der Zustand `soda_on` aktiviert.
5. Die Eintrittsaktion `start_soda` von `soda_on` wird ausgeführt und abgeschlossen.
6. Die notwendigen Schritte im Hierarchiezustand werden durchgeführt.

   Damit ist die Auswertung des Zustandsautomaten, die durch dieses Triggerereignis angestoßen wurde, beendet.

Beispiel 9: Übergänge aus verschiedenen Ausgangszuständen in denselben Zielzustand (mehrere Trigger)



Der Zustandsautomat ist derselbe wie in Beispiel 8. Das System befindet sich im Zustand `soda_on` (bzw. in einem der Unterzustände der Hierarchie). Ein Triggerereignis `trigger_soda` tritt ein, die Maschine wird abgeschaltet (`switch_off` ist `true`). Folgende Schritte werden ausgeführt:

1. Das System prüft, ob ein gültiger Übergang oder ein Segment aus `soda_on` vorhanden ist.
2. Das Übergangssegment von `soda_on` in den Knoten ist gültig, da die Bedingung `[switch_off]` erfüllt ist. Da das Triggerereignis `trigger_soda` eingetreten ist, ist auch das Segment vom Knoten in den Zustand `beverage_off` gültig; der Übergang kann stattfinden.
3. Die notwendigen Schritte im Hierarchiezustand werden durchgeführt.
4. Die Austrittsaktion `shut_down` des Zustands `soda_on` wird ausgeführt.
5. Der Übergang von `soda_on` in `beverage_off` hat keine Übergangsaktion, also wird als nächstes der Zustand `beverage_off` aktiviert.
6. Die Eintrittsaktion `reset` von `beverage_off` wird ausgeführt und abgeschlossen.

   Damit ist die Auswertung des Zustandsautomaten, die durch dieses Triggerereignis angestoßen wurde, beendet.

2.5.5 Semantik: Hierarchische Zustandsautomaten

Nach Aktivierung des Zustandsautomaten werden die Bedingungen der Übergänge geprüft. Die Priorität wird durch die hierarchische Reihenfolge festgelegt. Die höchste hierarchische Ebene hat die höchste Priorität, d.h. die

Bedingungen für die Übergänge auf den oberen hierarchischen Ebenen werden zuerst geprüft. Wenn ein Hierarchiezustand verlassen wird, werden die aktuellen Unterzustände gleichfalls verlassen. Der innerste Unterzustand wird zuerst verlassen, der äußerste Hierarchiezustand zuletzt. Beim Eintritt in einen Hierarchiezustand ist die Reihenfolge, in der die Eintrittsaktionen ausgeführt werden, vom äußersten Hierarchiezustand zum innersten (Unter-) Zustand, d.h. zuerst erfolgt der Eintritt in den äußersten Zustand und zuletzt der Eintritt in den innersten Zustand. Findet kein Übergang statt, werden die statischen Aktionen von innen nach außen ausgeführt, d.h. zuerst die statische Aktion des innersten Unterzustands und zuletzt die statische Aktion des äußersten Hierarchiezustands.

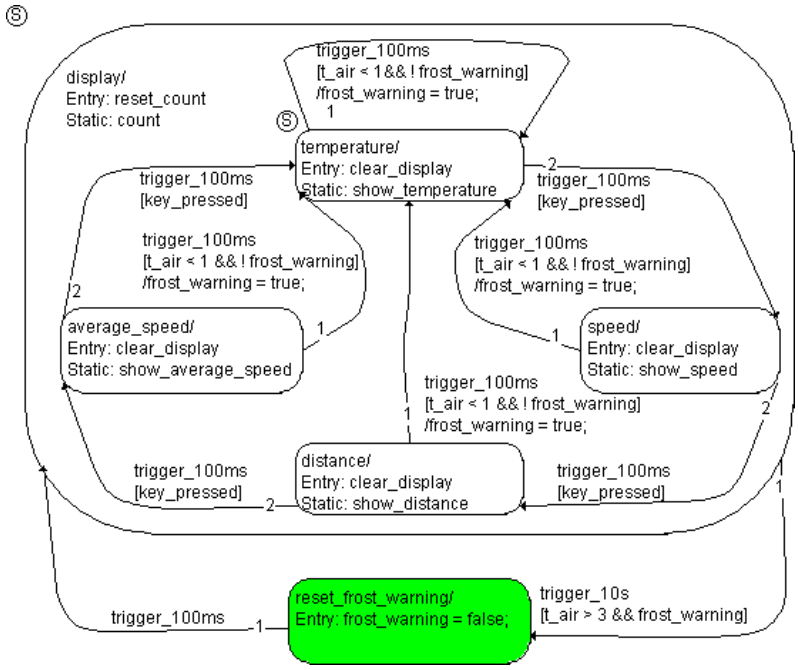
Hinweis

*Die Beispiele in diesem Kapitel gehen davon aus, dass **keine** Optimierung der statischen Aktionen von Hierarchiezuständen stattfindet. Ist diese Optimierung eingeschaltet, ändert sich die Semantik, siehe „Optimiert für Codegröße“ auf Seite 78.*

Beispiel 10: Übergang in einen Hierarchiezustand ohne Geschichte

Beim Eintritt in einen Hierarchiezustand gibt es zwei Möglichkeiten: Entweder erfolgt der Eintritt in den Anfangszustand des Hierarchiezustands (dieses Beispiel). In diesem Falle hat der Hierarchiezustand den Unterzustand ‘vergessen’, in dem er sich befand, als er verlassen wurde. Die andere Möglichkeit ist, dass der Eintritt in den letzten aktiven Unterzustand erfolgt. In diesem Falle hat der Hierarchiezustand eine Geschichte (Beispiel 11).

Für jeden Hierarchiezustand kann bestimmt werden, ob er eine Geschichte hat oder nicht. Beim erstmaligen Eintritt in einen Hierarchiezustand mit Geschichte erfolgt der Eintritt in den Anfangszustand dieses Hierarchiezustands.



Dieser hierarchische Zustandsautomat enthält im Zustand `display` die Anzeigefunktion aus Beispiel 3. `display` ist ein Hierarchiezustand. Sobald eine Temperatur von 3°C überschritten wird, soll die Eiswarnung abgeschaltet werden. Dazu dient der zweite Zustand auf der obersten hierarchischen Ebene, `reset_frost_warning`. Alle 10 Sekunden kann ein Wechsel von `display` in den Zustand `reset_frost_warning` passieren, in dem die Eiswarnung abgeschaltet wird.

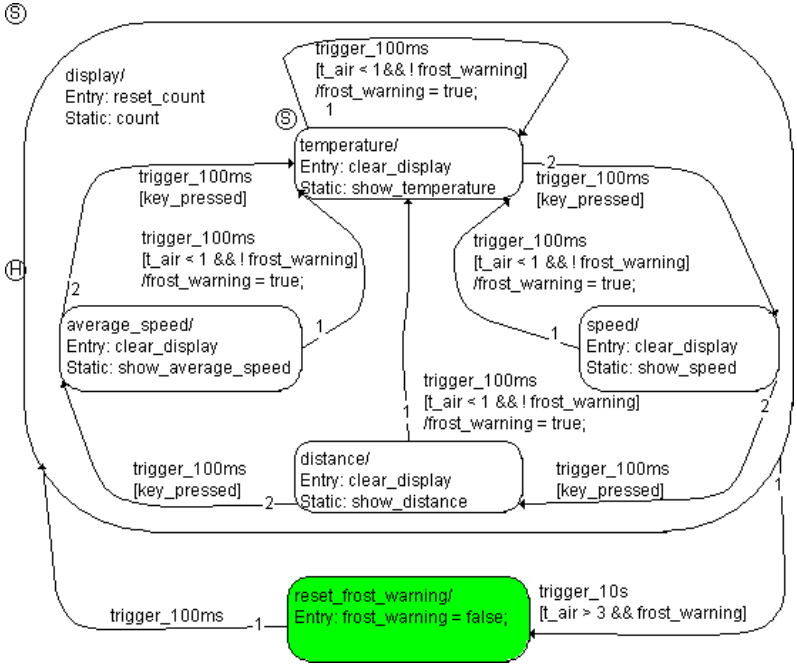
Nachdem die Eiswarnung gegeben wurde (`frost_warning` = `true`), wurde auf die Geschwindigkeitsanzeige geschaltet, das System befand sich im Zustand `speed`. Die Temperatur stieg auf 5°C, sodass beim Auftreten des Triggerereignisses `trigger_10s` der Übergang von `display` nach `reset_frost_warning` stattfand. Das System befindet sich jetzt im Zustand `reset_frost_warning`. Ein Triggerereignis `trigger_100ms` tritt ein; folgende Schritte werden ausgeführt:

1. Das System prüft, ob ein gültiger Übergang aus `reset_frost_warning` heraus vorhanden ist.
2. Der Übergang von `reset_frost_warning` nach `display` hat keine Bedingung, ist also bei jedem Triggerereignis `trigger_100ms` gültig.

3. Der Zustand `reset_frost_warning` hat keine Austrittsaktion. Er wird deaktiviert.
4. Der Übergang von `reset_frost_warning` nach `display` hat keine Übergangsaktion, also wird als nächstes der Hierarchiezustand `display` aktiviert.
5. Die Eintrittsaktion `reset_count` des Hierarchiezustands `display` wird ausgeführt und beendet.
6. Der Unterzustand `temperature` ist der Startzustand der Hierarchie. Er wird aktiviert.
7. Die Eintrittsaktion `clear_display` des Unterzustands `temperature` wird ausgeführt und beendet.

   Damit ist die Auswertung des Zustandsautomaten, die durch das Triggerereignis `trigger_100ms` angestoßen wurde, beendet.

Beispiel 11: Übergang in einen Hierarchiezustand mit Geschichte



Der Zustandsautomat ist derselbe wie in Beispiel 10, nur hat er jetzt eine Geschichte. Die Vorgeschichte und der Ausgangszustand sind dieselben wie im vorigen Beispiel.

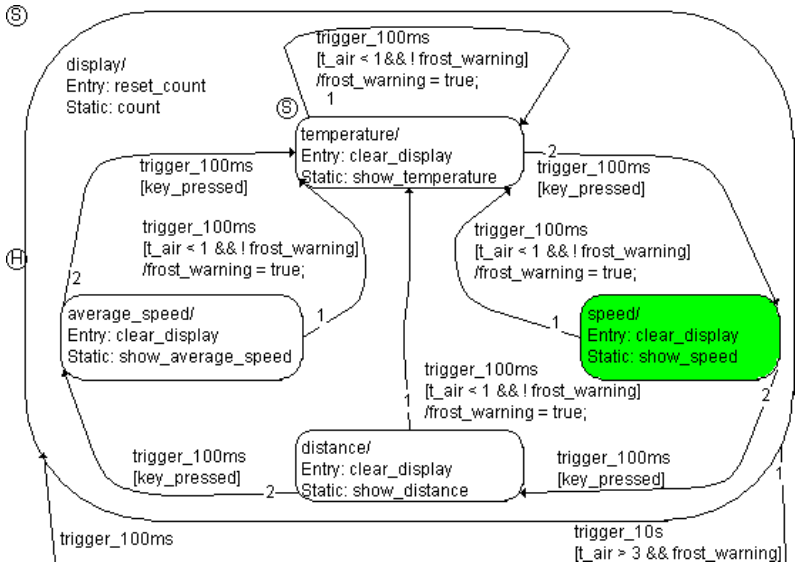
Das System befindet sich jetzt im Zustand `reset_frost_warning`. Ein Triggerereignis `trigger_100ms` tritt ein; folgende Schritte werden ausgeführt:

1. Das System prüft, ob ein gültiger Übergang aus `reset_frost_warning` heraus vorhanden ist.
2. Der Übergang von `reset_frost_warning` nach `display` hat keine Bedingung, ist also bei jedem Triggerereignis `trigger_100ms` gültig.
3. Der Zustand `reset_frost_warning` hat keine Austrittsaktion. Er wird deaktiviert.
4. Der Übergang von `reset_frost_warning` nach `display` hat keine Übergangsaktion, also wird als nächstes der Hierarchiezustand `display` aktiviert.
5. Die Eintrittsaktion `reset_count` des Hierarchiezustands `display` wird ausgeführt und beendet.
6. Da `display` eine Geschichte hat ('H' in der Abbildung), wird der Unterzustand `speed` aktiviert. Dieser war aktiv, als der Hierarchiezustand verlassen wurde.
7. Die Eintrittsaktion `clear_display` des Unterzustands `speed` wird ausgeführt und beendet.

   Damit ist die Auswertung des Zustandsautomaten, die durch das Triggerereignis `trigger_100ms` angestoßen wurde, beendet.

Beispiel 12: Übergang innerhalb eines Hierarchiezustands

Falls ein Übergang innerhalb eines Hierarchiezustands stattfindet, verbleibt der Zustandsautomat in diesem Hierarchiezustand. Daher wird auch die statische Aktion dieses Hierarchiezustands ausgeführt, ebenso wie die aller Hierarchiezustände, in denen der betreffende Zustand enthalten ist. Diese werden nach allen Austrittsaktionen und vor der Übergangsaktion in der Reihenfolge vom innersten zum äußersten Hierarchiezustand ausgeführt.

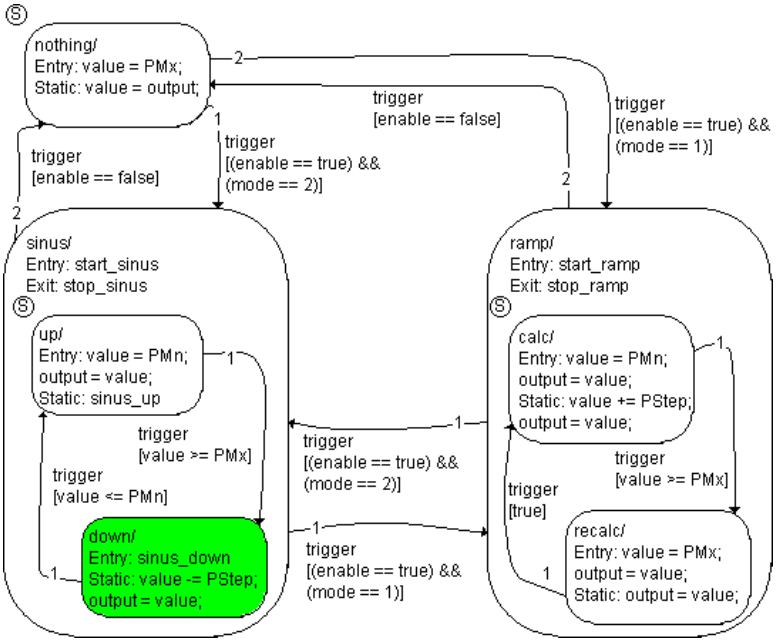


Der Zustandsautomat ist derselbe wie in Beispiel 11. Das System befindet sich im Zustand `speed`. Die Temperatur beträgt noch immer 5°C, die Eiswarnung ist abgestellt (`frost_warning` hat den Wert `false`). Ein Triggerereignis `trigger_100ms` tritt ein, der Schalter wird betätigt (`key_pressed` hat den Wert `true`). Folgende Schritte werden ausgeführt:

1. Das System prüft, ob ein gültiger Übergang vorhanden ist.
2. Der Übergang vom Hierarchiezustand `display` nach `reset_frost_warning` wird von einem anderen Trigger (`trigger_10s`) aufgerufen; er spielt hier keine Rolle.
3. Der Übergang von `speed` in den Unterzustand `distance` wird geprüft. Die Bedingung `[key_pressed]` ist erfüllt, der Übergang ist gültig.
4. Der Zustand `speed` hat keine Austrittsaktion. Er wird deaktiviert.
5. Der Hierarchiezustand `display` wird nicht verlassen. Also wird seine statische Aktion `count` ausgeführt und beendet.
6. Der Übergang von `speed` nach `distance` hat keine Übergangsaktion, also wird als nächstes der Unterzustand `distance` aktiviert.
7. Die Eintrittsaktion `clear_display` des Unterzustands `distance` wird ausgeführt und beendet.

   Damit ist die Auswertung des Zustandsautomaten, die durch das Triggerereignis `trigger_100ms` angestoßen wurde, beendet.

Beispiel 13: Übergang zwischen Hierarchiezuständen



Dieser Zustandsautomat fungiert als Datengenerator. Wenn `enable` den Wert `true` hat, wird ein Signal produziert, und zwar wahlweise eine Rampe (Zustand `ramp`, `mode = 1`) oder ein Sinus (Zustand `sinus`, `mode = 2`).

Der Unterzustand `down` im Hierarchiezustand `sinus` ist aktiv. Der Modus `mode` wird auf 1 gesetzt; `enable` bleibt `true`. Ein Triggerereignis tritt ein und folgende Schritte werden ausgeführt:

1. Das System prüft, ob ein gültiger Übergang vorhanden ist. Da die Übergänge aus dem Hierarchiezustand `sinus` heraus höhere Priorität haben als die Übergänge aus `down` heraus, werden sie zuerst untersucht.
2. Der Übergang von `sinus` nach `nothing` hat die höchste Priorität. Er ist jedoch ungültig, da die Bedingung `[enable == false]` nicht erfüllt ist.
3. Der Übergang von `sinus` nach `ramp` wird als nächster untersucht. Die Bedingung `[(enable ==true) && (mode == 1)]` ist wahr, also findet der Übergang statt.

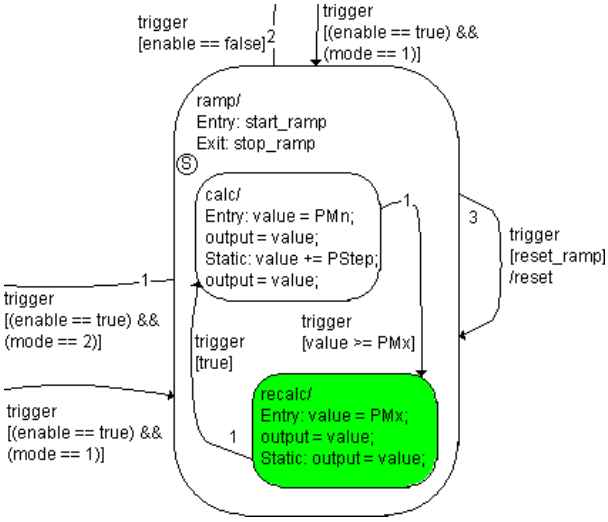
   Der Übergang vom Unterzustand `down` in den Unterzustand `up` hat eine niedrigere Priorität und wird nicht untersucht.

4. Der Unterzustand `down` hat keine Austrittsaktion, also wird er gleich deaktiviert.
5. Die Austrittsaktion `stop_sinus` des Hierarchiezustands `sinus` wird ausgeführt und abgeschlossen.
6. Der Hierarchiezustand `sinus` wird deaktiviert.
7. Der Übergang von `sinus` nach `ramp` hat keine Übergangsaktion, also wird als nächstes der Hierarchiezustand `ramp` aktiviert.
8. Die Eintrittsaktion `start_ramp` von `ramp` wird ausgeführt und abgeschlossen.
9. Der Unterzustand `calc` ist der Startzustand innerhalb der Hierarchie. Er wird aktiviert.
10. Die Eintrittsaktion `value = PMn; output = value;` von `calc` wird ausgeführt und abgeschlossen.

    Damit ist die Auswertung des Zustandsautomaten, die durch dieses Triggerereignis angestoßen wurde, beendet.

Beispiel 14: Schleife

Start- und Zielzustand eines Übergangs können identisch sein. Solche Schleifen werden häufig benutzt, um die Reset-Funktion eines Hierarchiezustandes zu spezifizieren.



Der Hierarchiezustand `ramp` aus dem Zustandsautomaten in Beispiel 13 hat hier eine Reset-Funktion in Form einer Schleife, also eines Übergangs von `ramp` nach sich selber. Der Rest des Zustandsdiagramms ist der Übersichtlichkeit halber weggelassen.

Der Unterzustand `recalc` im Hierarchiezustand `ramp` ist aktiv. Ein Triggerereignis tritt ein; der Reset-Knopf wird gedrückt (`reset_ramp = true`). `enable` und `mode` bleiben unverändert. Folgende Schritte werden ausgeführt:

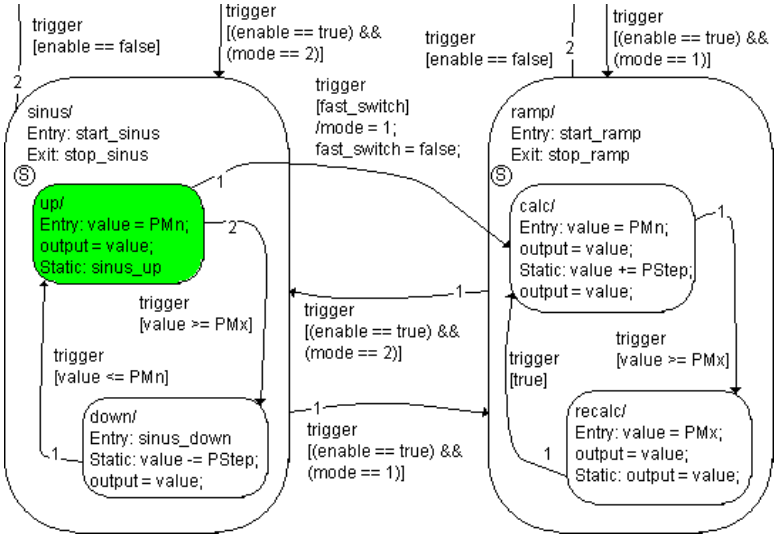
1. Das System prüft, ob ein gültiger Übergang vorhanden ist.
2. Die Schleife hat die höchste Priorität. Die Bedingung `[reset_ramp]` ist erfüllt, der Übergang ist gültig.

   Weitere Übergänge werden nicht untersucht.
3. Der Unterzustand `recalc` hat keine Austrittsaktion, also wird er gleich deaktiviert.
4. Die Austrittsaktion `stop_ramp` des Hierarchiezustands `ramp` wird ausgeführt und abgeschlossen.
5. Der Hierarchiezustand `ramp` wird deaktiviert.
6. Die Übergangsaktion `/reset` der Schleife wird ausgeführt und abgeschlossen.
7. Der Hierarchiezustand `ramp` wird wieder aktiviert.
8. Die Eintrittsaktion `start_ramp` von `ramp` wird ausgeführt und abgeschlossen.
9. Der Unterzustand `calc` ist der Startzustand innerhalb der Hierarchie. Er wird aktiviert.
10. Die Eintrittsaktion von `calc` wird ausgeführt und abgeschlossen.

    Damit ist die Auswertung des Zustandsautomaten, die durch dieses Triggerereignis angestoßen wurde, beendet.

Beispiel 15: Übergang zwischen Unterzuständen verschiedener Hierarchien

Übergänge können auch direkt vom Unterzustand eines Hierarchiezustands in den Unterzustand eines anderen Hierarchiezustands gehen.



Dieser Zustandsautomat ist derselbe wie in Beispiel 13, nur der Übergang vom Unterzustand `up` im Zustand `sinus` zum Unterzustand `calc` im Zustand `ramp` wurde hinzugefügt.

Der Unterzustand `up` im Hierarchiezustand `sinus` ist aktiv. Der Wert `value` ist kleiner als das Maximum `PMx`. Ein Triggerereignis tritt ein. `mode` bleibt auf 2, `enable` bleibt `true`, aber der Schnellschalter wird betätigt (`fast_switch = true`). Folgende Schritte werden ausgeführt:

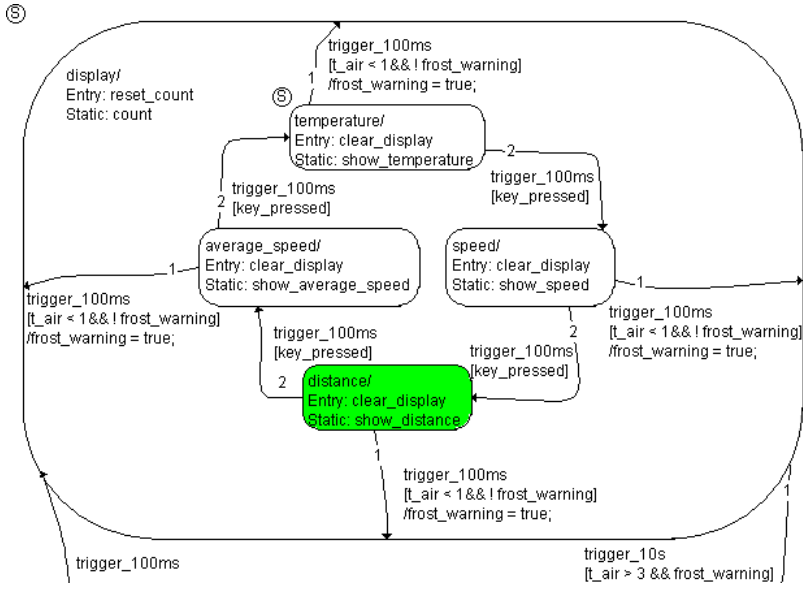
1. Das System prüft, ob ein gültiger Übergang vorhanden ist.
2. Die Übergänge von `sinus` nach `nothing` und von `sinus` nach `ramp` werden zuerst untersucht. Sie sind beide ungültig, da die zugehörigen Bedingungen nicht erfüllt sind.
3. Der Übergang vom Unterzustand `up` in den Unterzustand `down` wird als nächster untersucht. Auch er ist ungültig, da die Bedingung `[value >= PMx]` nicht erfüllt ist.
4. Der Übergang von `up` in den Unterzustand `calc` hat die niedrigste Priorität und wird als letzter untersucht. Die Bedingung `[fast_switch]` ist erfüllt; der Übergang findet statt.
5. Der Unterzustand `up` hat keine Austrittsaktion, also wird er gleich deaktiviert.

6. Die Austrittsaktion `stop_sinus` des Hierarchiezustands `sinus` wird ausgeführt und abgeschlossen.
7. Der Hierarchiezustand `sinus` wird deaktiviert.
8. Die Übergangsaktion (`/mode = 1; fast_switch = false;`) wird ausgeführt und abgeschlossen.
9. Der Hierarchiezustand `ramp` wird aktiviert.
10. Die Eintrittsaktion von `ramp` wird ausgeführt und abgeschlossen.
11. Der Unterzustand `calc` wird aktiviert.
12. Die Eintrittsaktion von `calc` wird ausgeführt und abgeschlossen.

    Damit ist die Auswertung des Zustandsautomaten, die durch dieses Triggerereignis angestoßen wurde, beendet.

Beispiel 16: Übergang von einem Unterzustand in einen Hierarchiezustand

Wenn der Übergang von einem Unterzustand nicht in einen anderen Unterzustand geht, sondern in den Hierarchiezustand, ist die Vorgehensweise fast dieselbe. Der Unterzustand wird verlassen, der Hierarchiezustand wird ebenfalls verlassen und gleich wieder betreten. Abhängig davon, ob der Hierarchiezustand eine Geschichte hat oder nicht, wird entweder der zuletzt aktive Unterzustand aktiviert oder der Startzustand innerhalb der Hierarchie. Auf diese Weise kann z.B. die Frostwarnung auch realisiert werden.



Der Zustandsautomat ist dem aus Beispiel 10 sehr ähnlich, nur ist hier die Eiswarnung durch Übergänge in den Hierarchiezustand realisiert. Er befindet sich im Zustand `distance`. `frost_warning` ist `false`. Ein Triggerereignis `trigger_100ms` tritt ein; die Temperatur fällt auf 0,5°C. Der Schalter wird nicht betätigt. Folgende Schritte werden ausgeführt:

1. Das System prüft, ob ein gültiger Übergang aus `distance` vorhanden ist.
2. Der Übergang von `display` nach `reset_frost_warning` wird von einem anderen Trigger aufgerufen; er spielt hier keine Rolle.
3. Der Übergang von `distance` nach `average_speed` wird geprüft. Die Bedingung `[key_pressed]` ist nicht erfüllt, der Übergang ist ungültig.
4. Der Übergang von `distance` nach `display` hat die Bedingung `[t_air < 1 && !frost_warning]`. Beide Teile der Bedingung sind wahr, der Übergang ist gültig.
5. Der Zustand `distance` hat keine Austrittsaktion. Er wird deaktiviert.
6. Der Hierarchiezustand `display` hat keine Austrittsaktion. Er wird deaktiviert.
7. Der Hierarchiezustand `display` wird wieder aktiviert
8. Die Eintrittsaktion `reset_count` von `display` wird ausgeführt und beendet.
9. Die Übergangsaktion `/frost_warning = true` wird ausgeführt und die Eiswarnung angezeigt.
10. Der Unterzustand `temperature` ist der Startzustand innerhalb der Hierarchie. Er wird aktiviert, da `display` keine Geschichte hat.
11. Die Eintrittsaktion `clear_display` des Unterzustands `temperature` wird ausgeführt und beendet.

    Damit ist die Auswertung des Zustandsautomaten, die durch das Triggerereignis `trigger_100ms` angestoßen wurde, beendet.

Beispiel 17: Kein Übergang

Der Zustandsautomat ist derselbe wie in Beispiel 16. Das System befindet sich nun im Zustand `temperature`. Die Temperatur ist unverändert. Ein Triggerereignis `trigger_100ms` tritt ein, der Schalter wird aber nicht betätigt (`key_pressed` ist `false`). Folgende Schritte werden ausgeführt:

1. Das System prüft, ob ein gültiger Übergang vorhanden ist.
2. Der Übergang von `display` nach `reset_frost_warning` wird von einem anderen Trigger aufgerufen; er spielt hier keine Rolle.

3. Der Übergang von `temperature` nach `speed` ist ungültig, da der Schalter nicht betätigt wurde.
4. Der Übergang von `temperature` nach `display` ist ungültig, da `frost_warning = true` und damit die Bedingung nicht erfüllt ist.

   Weitere mögliche Übergänge sind nicht vorhanden.
5. Die statische Aktion `show_temperature` im Unterzustand `temperature` wird ausgeführt und beendet.
6. Die statische Aktion `count` im Hierarchiezustand `display` wird ausgeführt und beendet.

   Damit ist die Auswertung des Zustandsautomaten, die durch das Triggerereignis `trigger_100ms` angestoßen wurde, beendet.

2.5.6 Semantik: Zusammenfassung

Initialisieren des Zustandsdiagramms: Der Startzustand des Systems wird aktiviert. Ist der Startzustand ein Hierarchiezustand, wird auch der Startzustand innerhalb der Hierarchie aktiviert. Dabei wird *keine* Eintrittsaktion ausgeführt.

In einen Zustand eintreten:

1. Wenn der Zustand einen nicht aktiven übergeordneten Zustand hat, werden zunächst für diesen die Schritte 1 – 4 ausgeführt.
2. Der Zustand wird aktiviert.
3. Die Eintrittsaktion wird ausgeführt.
4. Führe bei Bedarf implizite Eintrittsaktionen aus:

   4.1 Wenn der Zustand ein untergeordnetes Diagramm ohne Geschichte enthält, wird dessen Startzustand aktiviert und seine Eintrittsaktion ausgeführt.

   4.2 Wenn der Zustand ein untergeordnetes Diagramm mit Geschichte enthält, und wenn einer der Unterzustände nach der Initialisierung des Zustandsautomaten aktiv war, wird dieser Unterzustand aktiviert und seine Eintrittsaktion ausgeführt. Andernfalls wird wie unter 4.1verfahren.

Einen (Basis-)Zustand ausführen:

1. Die vom Zustand wegführenden Übergänge sowie Übergänge, die aus übergeordneten Zuständen wegführen, werden in der Reihenfolge ihrer Priorität ausgewertet.
2. Wird ein gültiger Übergang gefunden, wird dieser ausgeführt. Damit ist die Ausführung des Zustands beendet.

3. Ist kein gültiger Übergang aus dem Zustand vorhanden, wird die statische Aktion ausgeführt.
4. Wenn der Zustand übergeordnete Zustände hat, werden deren statische Aktionen ausgeführt.

Einen Zustand verlassen:

1. Wenn der Zustand aktive Unterzustände enthält, werden deren Austrittsaktionen ausgeführt. Die Austrittsaktion des innersten (Basis-)Zustands wird zuerst ausgeführt.
2. Die Austrittsaktion des Zustands wird ausgeführt.
3. Der Zustand wird deaktiviert.

Einen Übergang ausführen:

Übergänge werden in der Reihenfolge ihrer Priorität ausgewertet. Übergänge aus einem Hierarchiezustand haben immer eine höhere Priorität als Übergänge aus den Unterzuständen dieses Hierarchiezustands.

1. Ein Übergang oder Übergangssegment wird geprüft.
2. Ist der Übergang/das Segment ungültig, wird der Übergang/das Segment mit der nächstniedrigeren Priorität geprüft.
3. Ist der Übergang/das Segment gültig, hängt der nächste Schritt davon ab, wo der Übergang/das Segment endet.

   In einem Zustand:

   3.1 Keine weiteren Übergänge oder Übergangssegmente werden geprüft. Im Fall eines Übergangssegments aus einem Knoten wird das Segment in den fraglichen Knoten hinzugezogen, um einen vollständigen Übergang zu erhalten.

   3.2 Die Unterzustände des Ausgangszustandes werden verlassen (siehe „Einen Zustand verlassen“).

   3.3 Der Ausgangszustand wird verlassen.

   3.4 Die Übergangsaktion wird ausgeführt.

   3.5 Das System tritt in den Zielzustand ein (siehe „In einen Zustand eintreten“).

   In einem Knoten:

   3.1 Die vom Knoten wegführenden Übergangssegmente werden wie in den Schritten 1 – 3 beschrieben ausgewertet.

4. Wenn alle von einem Knoten wegführenden Übergangssegmente ungültig sind, geht das System zurück in den Ausgangszustand, von dem aus der Knoten erreicht wurde. Da das Segment in den Knoten zu keinem gültigen Übergang gehört, werden die Schritte 1 – 4 für den Übergang/das Segment mit der nächstniedrigeren Priorität ausgeführt.
5. Wenn alle von einem Zustand wegführenden Übergänge/Segmente ungültig sind, findet kein Übergang statt, das System bleibt in dem Zustand.

Der Ablauf ist in Abb. 2-13 schematisch dargestellt.

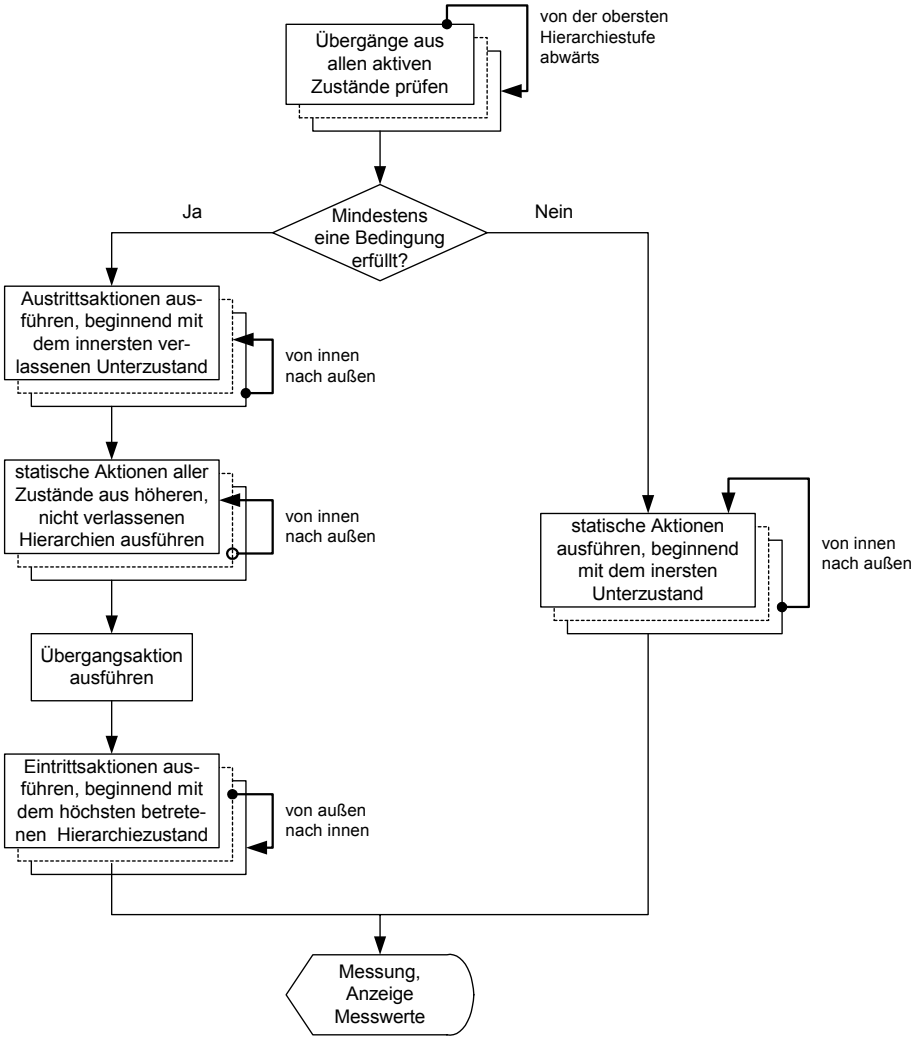
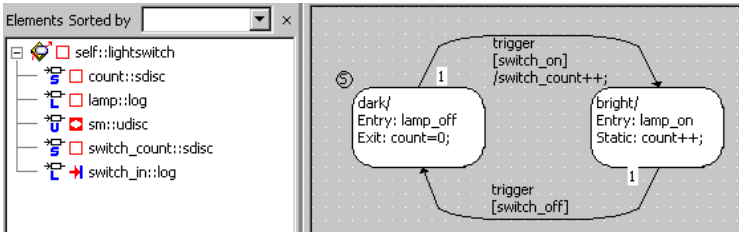


Abb. 2-13 Auswertung eines Zustandsautomaten

2.5.7 Einfaches Codebeispiel



Für diesen ganz einfachen Zustandsautomaten ist hier ein Ausschnitt aus dem generierten Code abgebildet.

```
/*
 *--------------------------------------------------------------------------
 *      Defines
 *--------------------------------------------------------------------------
 */

#define bright 1
#define dark 0
/*

/*****************************************************************************
 * BEGIN: Function definitions - Algorithms
 *---------------------------------------------------------------------------*/
void LIGHTSWITCH_IMPL_trigger(struct LIGHTSWITCH_IMPL_Obj *self)
{
   switch (self->sm->val)
   {
      default:
      case dark :
      {
         if (LIGHTSWITCH_IMPL_switch_on (self))
         {
            self->count->val = (sint32)0;
            self->switch_count->val++;
            self->lamp->val = (uint8)true;
            self->sm->val = bright;
            return;
         }
         return;
      }
      case bright :
      {
         if (LIGHTSWITCH_IMPL_switch_off (self))
         {
            self->lamp->val = (uint8)false;
            self->sm->val = dark;
            return;
         }
         self->count->val++;
         return;
      }
   }
}
uint8 LIGHTSWITCH_IMPL_getlamp(struct LIGHTSWITCH_IMPL_Obj *self)
{
   return (self->lamp->val);
}
uint8 LIGHTSWITCH_IMPL_setswitch_in(struct LIGHTSWITCH_IMPL_Obj *self,
   uint8 parm)
{
   return ((uint8)(self->switch_in->val = parm));
}
```

`#define`-Statements für die Zustände

Austrittsaktion von `dark`

Übergang von `dark` nach `bright`

Übergangsaktion

Eintrittsaktion von `bright`

Eintrittsaktion von `dark`

Übergang von `bright` nach `dark`

statische Aktion von `bright`

öffentliche Methode für den Ausgang `lamp`

öffentliche Methode für den Eingang `switch_in`

2.5.8 Den Zustandsautomaten optimieren

Es gibt meist mehrere Möglichkeiten, dieselbe Funktionalität zu spezifizieren oder die Codegenerierung/den Build-Prozess einzurichten.

Wenn für einen Zustandsautomaten Code generiert wird, werden Teile von Aktionen und Bedingungen, die am Zustand oder Übergang spezifiziert wurden, entweder an Ort und Stelle in den Code eingefügt (*Inlining*) oder – unter bestimmten Voraussetzungen – als eigene Methoden generiert (*Outlining*). Die Voraussetzungen für Outlining sind folgende:

1. In dem Projekt, zu dem der Zustandsautomat gehört, ist im Fenster „Project Properties", Knoten „Statemachine", die Option **Outline Generated Methods (may be changed locally)** aktiviert.

   Diese Option betrifft alle Zustandsautomaten im Projekt; sie gilt für alle Experimenttypen (physikalisch, quantisiert, implementiert).

2. Im Implementierungseditor für den Zustandsautomaten ist im Register „Settings" die Option **Outline automatically generated methods for State Machines** aktiviert.

Hinweis

*Wenn die erste Voraussetzung nicht erfüllt ist, findet **kein** Outlining für irgendeinen Zustandsautomaten im Projekt statt.*
*Wenn die erste Voraussetzung erfüllt ist, die zweite aber nicht, findet für **diesen** Zustandsautomaten **kein** Outlining statt.*

Wenn beide Voraussetzungen erfüllt sind, wird während der Codegenerierung die Codegröße mit und ohne Outlining geprüft. Ist der Code mit Outlining kleiner, wird diese Möglichkeit gewählt.

Wenn Aktionen und Bedingungen (oder Teile davon) in separaten Diagrammen spezifiziert werden, wird der Code dafür entweder in Form von eigenen privaten Funktionen generiert (Outlining) oder während der Codegenerierung automatisch an Ort und Stelle in den Code eingefügt (*Auto-Inlining*).

Folgende Voraussetzungen müssen erfüllt sein, damit Auto-Inlining stattfinden kann:

1. In dem Projekt, zu dem der Zustandsautomat gehört, ist im Fenster „Project Properties", Knoten „Statemachine", die Option **Auto-inline private methods (Smaller code size - may be changed locally)** aktiviert.

   Diese Option betrifft alle Zustandsautomaten im Projekt; sie gilt für alle Experimenttypen (physikalisch, quantisiert, implementiert).

2. Im Implementierungseditor für den Zustandsautomaten ist im Register „Settings“ die Option **Auto-inline private methods (Smaller code size)** aktiviert.

Hinweis

*Wenn die erste Voraussetzung nicht erfüllt ist, findet **kein** Auto-Inlining für irgendeinen Zustandsautomaten im Projekt statt.*
*Wenn die erste Voraussetzung erfüllt ist, die zweite aber nicht, findet für **diesen** Zustandsautomaten **kein** Auto-Inlining statt.*

Wenn beide Voraussetzungen erfüllt sind, wird bei der Codegenerierung die Größe mit und ohne Auto-Inlining überprüft. Ist der Code mit Auto-Inlining kleiner, wird diese Möglichkeit gewählt. Dies ist üblicherweise für kleine private Funktionen oder für solche mit wenigen Aufrufen der Fall. Die Prüfung findet für jede Funktion getrennt statt, sodass nur die Funktionen an Ort und Stelle eingefügt werden, deren Inlining Codegröße einspart.

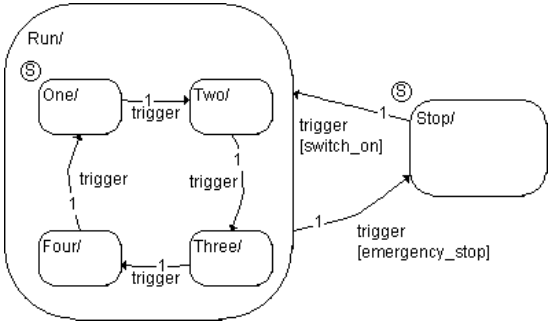
Je nachdem, für welche Möglichkeiten Sie sich entscheiden, können Sie einen Zustandsautomaten unter drei Gesichtspunkten optimieren:

- Antwortzeit auf ein Ereignis
- Laufzeit
- Codegröße

Optimiert für Antwortzeit

Wenn die Antwortzeit das wichtigste Kriterium ist, sollten Sie die hierarchische Struktur und die Prioritäten der Übergänge gezielt einsetzen. Zeitkritische Aktionen werden in die höchstmögliche hierarchische Ebene eingebaut, um effizienten Code und eine schnellstmögliche Reaktion zu erzeugen.

Das wird an einem Beispiel verdeutlicht:



Wenn der Notfallschalter betätigt wird (`emergency_stop = true`), sollte das System so schnell wie möglich anhalten, also in den Zustand `Stop` wechseln. Indem der entsprechende Übergang vom Hierarchiezustand `Run` zum Zustand `Stop` geht, hat er innerhalb der Hierarchie die höchste Priorität und wird als erster überprüft.

Wenn einer der Unterzustände bei Betätigen des Notschalters aktiv ist, wird immer der Übergang von `Run` zu `Stop` als erster ausgewertet und der Übergang findet statt.

Direkte Übergänge von jedem der Unterzustände nach `Stop` sind ebenso zeiteffektiv, erfordert aber mehr Wartungsaufwand, da vier Übergänge statt eines einzigen betroffen sind.

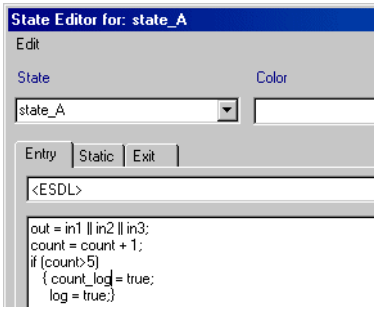
Ein eigener Trigger für zeitkritische Ereignisse (im Beispiel `emergency_stop = true`) optimiert ebenfalls die Antwortzeit. Nachteilig ist hierbei, dass für den eigenen Trigger zusätzlicher Programmcode generiert wird.

Optimiert für Laufzeit

Wenn die Gesamtlaufzeit das wichtigste Kriterium ist, gibt es verschiedene Optimierungsmöglichkeiten, um effizienten Code zu erzeugen. Die verschiedenen Möglichkeiten können einzeln oder in Kombination verwendet werden.

Aktionen/Bedingungen: Wenn *Aktionen* oder *Bedingungen* spezifiziert werden, die ganz oder teilweise dieselbe Funktionalität haben, kann dies wahlweise für Laufzeit oder Codegröße optimiert geschehen. *Laufzeitoptimiert* bedeutet dabei, dass der Code für jede Aktion/Bedingung bei der Codegenerierung an Ort und Stelle eingefügt wird. Ein zusätzlicher Funktionsaufruf ist hierbei nicht nötig. Der Nachteil ist mehrfach generierter Code und damit erhöhter Speicherbedarf.

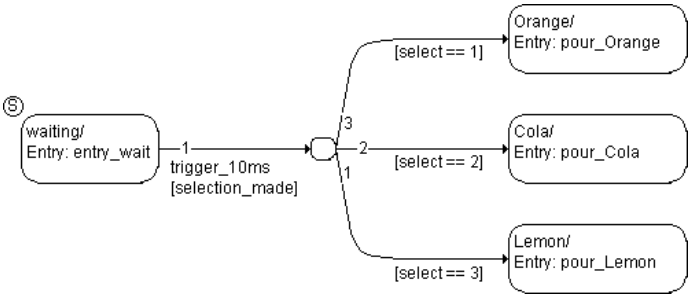
Dies kann erreicht werden, indem der Code explizit am Zustand oder Übergang eingegeben und die Optionen **Outline Generated Methods (may be changed locally)** und **Outline automatically generated methods for State Machines** deaktiviert werden (siehe die Voraussetzungen für Outlining auf Seite 74).



Noch effektiver wird die Optimierung, wenn Auto-Inlining (s. Seite 74) aktiviert ist; in diesem Fall werden gegebenenfalls auch solche Aktionen/Bedingungen an Ort und Stelle in den Code eingefügt, die in separaten Diagrammen spezifiziert wurden.

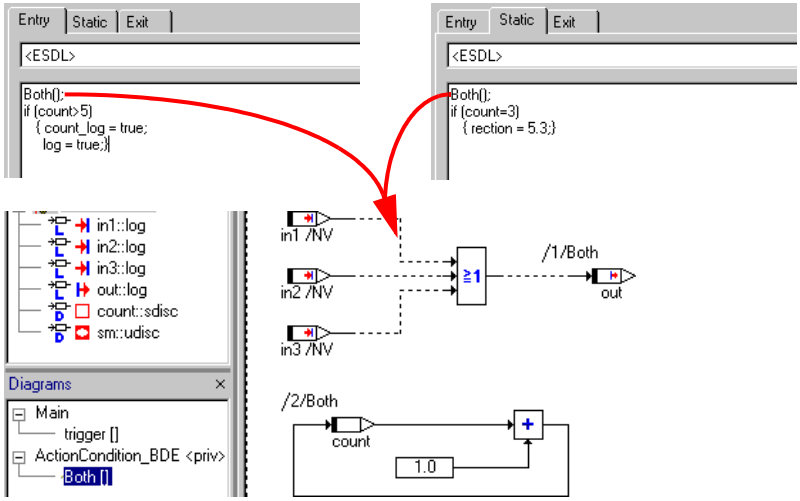
Inlining *erzwingen* können Sie mit der Option **Inline** im Implementierungseditor einer Aktion oder Bedingung, die in einem separaten Diagramm spezifiziert ist.

Knoten: Wenn von einem Zustand mehrere Übergänge mit teilweise identischen Bedingungen wegführen, kann die Verwendung von Knoten Laufzeitersparnis bringen. Gleiche Teile der Bedingungen werden dem Übergangssegment vom Ausgangszustand in den ersten Knoten zugewiesen; sind diese nicht erfüllt, brauchen die anderen Segmente nicht ausgewertet werden.



Aktionen/Bedingungen: Wenn Aktionen oder Bedingungen *für Codegröße optimiert* werden, heißt das, dass die gleichen Teile der Aktionen/Bedingungen im generierten Code als eigene private Funktionen enthalten sind, die bei Bedarf aufgerufen werden.

Das kann erreicht werden, indem die mehrfach genutzten Teile als Methoden in einem separaten Diagramm spezifiziert werden; diese Methoden werden dann in den Aktionen oder Bedingungen aufgerufen (siehe Abbildung).



Alternativ dazu können Sie den Code auch direkt am Zustand oder Übergang eingeben und das Outlining nutzen.

Für beide Alternativen wird der Code für die betreffenden Aktionen/Bedingungen nicht jedesmal generiert. Nachteilig hinsichtlich der Laufzeit sind zusätzliche Funktionsaufrufe.

In manchen Fällen (kleine private Funktionen, wenige Aufrufe) kann es bezüglich Codegöße günstiger sein, die Funktionen an Ort und Stelle einzufügen. Mit den Optionen **Auto-inline private methods (Smaller code size - may be changed locally)** und **Auto-inline private methods (Smaller code size)** aktivieren Sie Auto-Inlining (s. Seite 74); damist ist für eine möglichst effektive Optimierung von Aktionen und Bedingungen bezüglich Codegröße gesorgt.

Statische Aktionen von Hierarchiezuständen: Für statische Aktionen von Hierarchiezuständen gibt es eine weitere Optimierungsmöglichkeit für Codegröße.

Code für die statische Aktion eines Hierarchiezustands wird standardmäßig für jeden Übergang generiert, der nicht aus dem Hierarchiezustand hinausführt, sowie einmal für jeden Unterzustand der Hierarchie. Bei großen Hierarchien kann das einen merklichen Anteil am Gesamtcode geben.

Wenn Sie in dem Projekt, zu dem der Zustandsautomat gehört, die Codeoptimierungsoption **Optimize Static Actions (Restricted Modeling)** aktivieren, wird der Code für die statische Aktion des Hierarchiezustands nur noch einmal für jeden Unterzustand generiert. Auf diese Weise kann die Codegröße reduziert werden.

Nachteilig bei dieser Optimierung ist, dass sie nicht für alle Modelle funktioniert. Hat ein Unterzustand des Zustandsautomaten einen direkten Übergang aus seinem Hierarchiezustand heraus, muss dieser Übergang von allen Übergängen aus dem Unterzustand die höchste Priorität haben. Andernfalls bricht die Codegenerierung mit folgender Fehlermeldung ab:

```
ERROR(YSm72): higher priority transitions do not exit
hierarchy state "HState", but this transition does.
```

Durch die Änderungen bei der Codegenerierung ändert sich die Semantik des Zustandsautomaten wie folgt:

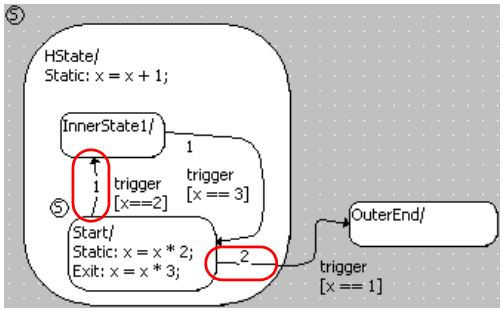
Hinweis

Die genannten Änderungen können das Verhalten des Zustandsautomaten beeinflussen. Prüfen Sie das Verhalten sorgfältig, wenn Sie die Option für einen vorhandenen Zustandsautomaten aktivieren.

- Die statische Aktion des Hierarchiezustands wird ausgeführt, *bevor* die Bedingungen an den Übergängen des Unterzustands geprüft werden.
- Die statische Aktion des Hierarchiezustands wird *vor* der statischen Aktion des Unterzustands ausgeführt, wenn kein Übergang stattfindet.
- Die statische Aktion des Hierarchiezustands wird *vor* der Austrittsaktion des Unterzustands ausgeführt, wenn ein Übergang stattfindet.

Zwei Beispiele sollen die Wirkung dieser Optimierung darstellen. Der Zustandsautomat besteht in beiden Beispielen aus dem Hierarchiezustand `HState`, der die Unterzustände `Start` und `InnerState1` enthält, sowie dem Basiszustand `OuterEnd`. Von `Start` gehen zwei Übergänge weg, einer davon (`Start` → `OuterEnd`) verlässt den Hierarchiezustand `HState`.

Im ersten Beispiel hat der Übergang von `Start` nach `OuterEnd` eine höhere Priorität als der Übergang von `Start` nach `InnerState1`. Das heißt, Code kann sowohl mit deaktivierter als auch mit aktivierter Option **Optimize Static Actions (Restricted Modeling)** generiert werden.



Die folgende Tabelle zeigt den generierten C-Code für die beiden Fälle. Code für die statische Aktion von `HState` ist fettgedruckt.

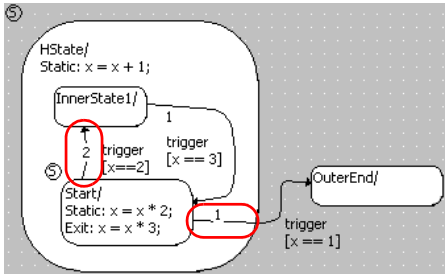
Option deaktiviert	**Option aktiviert**

`case Start :`
`{`
`  if (x == 1.0)`
`  {`
`  x = x * 3.0;`
`  sm = OuterEnd;`
`  return;`
`  }`
`  if (x == 2.0)`
`  {`
`  x = x * 3.0;`
`  **x = x + 1.0;**`
`  sm = InnerState1;`
`  return;`
`  }`
`  x = x * 2.0;`
`  **x = x + 1.0;**`
`  return;`
`}`
`case InnerState1 :`
`{`
`  if (x == 3.0)`
`  {`
	`case Start :`
	`{`
	`  if (x == 1.0)`
	`  {`
	`  x = x * 3.0;`
	`  sm = OuterEnd;`
	`  return;`
	`  }`
	`  **x = x + 1.0;**`
	`  if (x == 2.0)`
	`  {`
	`  x = x * 3.0;`
	`  sm = InnerState1;`
	`  return;`
	`  }`
	`  x = x * 2.0;`
	`  return;`
	`}`
	`case InnerState1 :`
	`{`
	`  **x = x + 1.0;**`
	`  if (x == 3.0)`
	`  {`

Option deaktiviert	Option aktiviert

x = x + 1.0;
sm = Start;
return;
}
x = x + 1.0;
return;
}
	sm = Start;
	return;
	}
	return;
	}

Im zweiten Beispiel hat der Übergang von `Start` nach `OuterEnd` eine niedrigere Priorität. Code kann mit aktivierter Option **Optimize Static Actions (Restricted Modeling)** nicht generiert werden.



Option deaktiviert	Option aktiviert

case Start :
{
if (x == 2.0)
{
x = x * 3.0;
x = x + 1.0;
sm = InnerState1;
return;
}
if (x == 1.0)
{
x = x * 3.0;
sm = OuterEnd;
return;
}
x = x * 2.0;
x = x + 1.0;
return;
}
	ERROR(YSm72): higher priority transitions do not exit hierarchy state "HState", but this transition does.

Option deaktiviert	Option aktiviert

`case InnerState1 :`
`{`
`  if (x == 3.0)`
`  {`
`  **x = x + 1.0;**`
`  sm = Start;`
`  return;`
`  }`
`  **x = x + 1.0;**`
`  return;`
`}`	

Hierarchische Codegenerierung: Bei der Codegenerierung für einen hierarchischen Zustandsautomaten gibt es zwei Möglichkeiten:

Mit *flacher Codegenerierung* wird die Hierarchie geglättet, d.h. es wurde eine einzige `switch`-Anweisung für alle (Basis-)Zustände und Übergänge erzeugt.

Mit *hierarchischen Codegenerierung* werden gemäß der Hierarchie geschachtelte `switch`-Anweisungen erzeugt. Um diese Art der Codegenerierung einzuschalten, müssen folgende Optionen aktiviert werden:

1. Projekteinstellungen, Knoten „Statemachine“: **Hierarchical Code-Generation (may be changed locally)**

   und

2. Implementierungseditor des Zustandsautomaten, Register „Settings“: **Hierarchical code generation for State Machines**

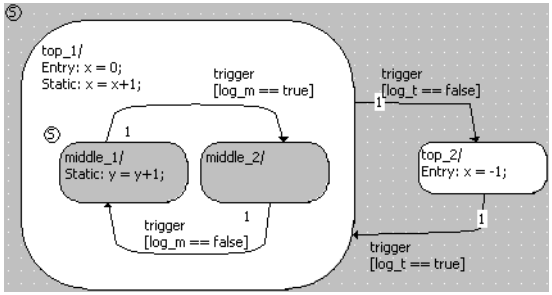
Ist die erste Option nicht aktiviert, findet keine hierarchische Codegenerierung statt. Ist die erste Option aktiviert, schaltet die zweite Option die hierarchische Codegenerierung für einen bestimmten Zustandsautomaten ein oder aus.

Bei der hierarchischen Codegenerierung werden die Übergänge von Hierarchiezuständen nur einmal generiert, nicht wie bei flacher Codegenerierung einmal für jeden betroffenen Basiszustand. Auf diese Weise wird die Codegröße reduziert. Die Reduktion kann beträchtlich sein (bis 30%). Im Experiment verhalten sich hierarchische und flache Codegenerierung für identische Zustandsautomaten gleich.

Hinweis

*Für Hierarchiezustände **ohne** Übergänge und/oder **ohne** statische Aktionen wird die Codegröße nicht reduziert, sondern geringfügig (1–2%) **erhöht**.*

Ein Beispiel soll den Unterschied im generierten Code zeigen.



Hinweis

Die reduzierte Codegröße zeigt sich nicht in der generierten C-Code-Datei, sondern im generierten ausführbaren Code.

Der Übergang von `top_1` nach `top_2` ist fettgedruckt.

hierarchische Codegenerierung

```
switch (self->                    ↵
      _ASCET_smLevel_0->val)
{
 case top_2 :
 {
  if (self->log_t->val)
  {
   self->x->val = 0.0;
   self->                         ↵
         _ASCET_smLevel_0->       ↵
         val = top_1;
   self->sm->val = middle_1;
   return;
  }
  return;
 }
 default:
 case top_1 :
 {
  if (!self->log_t->val)
  {
```

flache Codegenerierung

```
switch (self->sm->val)
{
 default:
 case middle_1 :
 {
  if (!self->log_t->val)
  {
   self->x->val = -1.0;
   self->sm->val = top_2;
   return;
  }
  if (self->log_m->val)
  {
   self->x->val = self->    ↵
                  x->val + 1.0;
   self->y->val = -1.0;
   self->sm->val = middle_2;
   return;
  }
  self->y->val = self->y->↵
                 val + 1.0;
```

```
 self->x->val = -1.0;
 self->                          ↵
      _ASCET_smLevel_0->↵
      val = top_2;
 self->sm->val = top_2;
 return;
}
switch (self->sm->val)
{
 default:
 case middle_1 :
 {
  if (self->log_m->val)
  {
   self->x->val = self-> ↵
              x->val + 1.0;
   self->y->val = -1.0;
   self->sm->val =       ↵
              middle_2;
   return;
  }
  self->y->val = self->  ↵
              y->val + 1.0;
  self->x->val = self->  ↵
              x->val + 1.0;
  return;
 }
 case middle_2 :
 {
  if (!self->log_m->val)
  {
   self->x->val = self-> ↵
             x->val + 1.0;
   self->sm->val =       ↵
              middle_1;
   return;
  }
  self->x->val = self->  ↵
              x->val + 1.0;
  return;
  self->x->val = self->x->↵
                 val + 1.0;
  return;
 }
 case middle_2 :
 {
  if (!self->log_t->val)
  {
   self->x->val = -1.0;
   self->sm->val = top_2;
   return;
  }
  if (!self->log_m->val)
  {
   self->x->val = self->    ↵
                 x->val + 1.0;
   self->sm->val = middle_1;
   return;
  }
  self->x->val = self->x->↵
                 val + 1.0;
  return;
 }
 case top_2 :
 {
  if (self->log_t->val)
  {
   self->x->val = 0.0;
   self->sm->val = middle_1;
   return;
  }
  return;
 }
}
```

```
    }
   }
  }
 }
```

Trigger und Triggerargumente: Werden *Triggerargumente* anstelle von Ein- und Ausgängen für die Kommunikation mit anderen ASCET-Komponenten verwendet, wird der statische RAM-Bedarf reduziert. Mehr dazu finden Sie im folgenden Kapitel.

2.5.9 Zustandsautomaten als Klassen

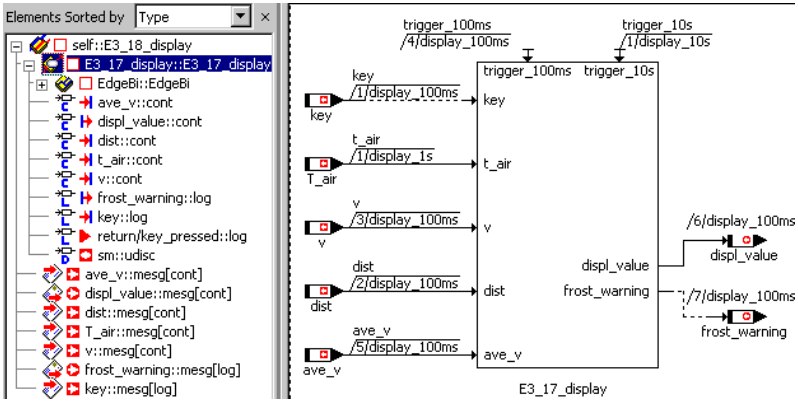
Ein Zustandsautomat ist eine Klasse mit speziellen Mitteln zur Beschreibung. Trigger, Bedingungen und Aktionen werden als spezielle Methoden modelliert:

- Ein Trigger ist eine öffentliche Methode ohne Rückgabewert. Der Zustandsautomat wird bei jedem Aufruf eines Triggers ausgeführt.
- Eine Bedingung ist eine private Methode mit einem Rückgabewert vom Typ logischer Wert.
- Eine Aktion ist eine private Methode. Sie hat standardmäßig keine Argumente und keinen Rückgabewert.

Bei Bedarf können für die Kommunikation mit anderen ASCET-Komponenten zu jeder dieser Methoden Argumente hinzugefügt werden.

Eingänge und Ausgänge dienen zur Integration des Zustandsautomaten in anderen Komponenten. Die Eingabewerte werden in internen Variablen zwischengespeichert und können daher in allen Berechnungen des Zustandsautomaten verwendet werden (im Gegensatz zu Argumenten einer Methode, die nur in der Methode selbst verwendet werden können). Die Ausgabewerte wer-

den ebenfalls zwischengespeichert, sodass sie gelesen werden können, ohne die Berechnung des Zustandsautomaten aufzurufen. Für jeden Eingang und Ausgang ist ein eigener Sequenzaufruf (s. Kapitel 6.3) erforderlich.



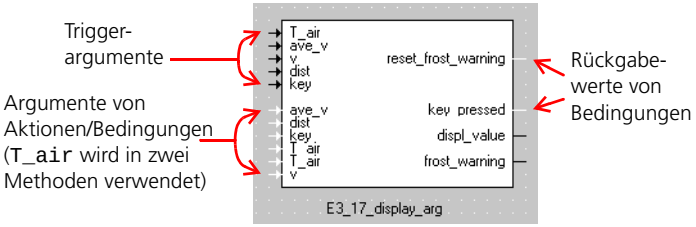
Diese Art der externen Kommunikation ist allerdings speicherintensiv, da für jeden Eingang und Ausgang eine Variable im RAM reserviert werden muss. Um den statischen RAM-Bedarf zu senken, können Sie Trigger (sowie Aktionen und Bedingungen, wenn diese in separaten Diagrammen spezifiziert sind) mit Argumenten versehen und diese für die externe Kommunikation verwenden. Für die Argumente einer C-Funktion werden Stack-Variablen angelegt, die den statischen RAM nicht belasten. Der dynamische RAM-Bereich wird allerdings temporär höher belastet.

Grundsätzlich ist folgendes zu beachten:

- Trigger sind öffentliche Methoden. Ihre Argumente können von außerhalb des Zustandsautomaten beschrieben werden. Im Layout-Editor sind die Triggerargumente als schwarze Argumentanschlüsse dargestellt.

- Argumente und Bedingungen sind private Methoden. Ihre Argumente sind daher außerhalb des Zustandsautomaten nicht verfügbar; sie werden im Layout-Editor als weiße Argumentanschlüsse dargestellt.

  Soll nun ein Triggerargument in einer als Blockdiagramm spezifizierten Aktion oder Bedingung verwendet werden, muss zu jeder entsprechenden Methode ein Argument vom gleichen Typ und mit dem gleichen Namen wie das Triggerargument hinzugefügt werden.



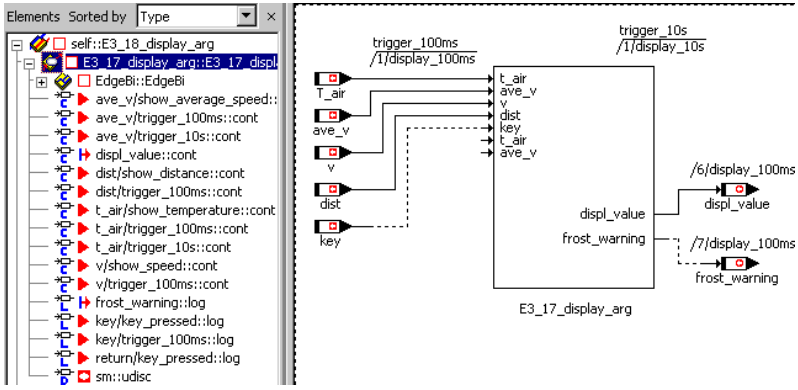
  Die Argumente werden entsprechend ihrer Namen und ihres Typs aufeinander abgebildet. Wenn im Trigger und der Aktion/Bedingung gleichnamige Argumente mit verschiedenen Typen vorhanden sind, wird eine Warnung ausgegeben. Ist das Argument nur in einer Aktion oder Bedingung, nicht aber im aufrufenden Trigger definiert, wird eine Fehlermeldung ausgegeben.

Darüber hinaus gibt es folgende Regeln für die Verwendung von Triggerargumenten in Aktionen und Bedingungen:

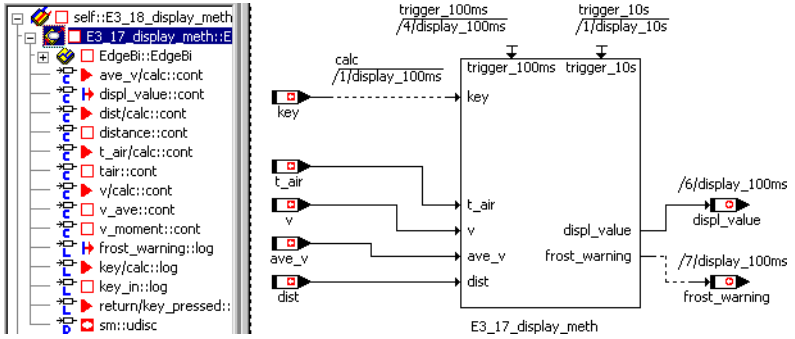
- Alle Triggerargumente, die in der *Eintrittsaktion* eines Zustands benutzt werden sollen, müssen in der Aktion sowie in jedem Trigger definiert sein, der zu einem in den Zustand hineinführenden Übergang gehört, da sie von diesen Triggern aufgerufen wird.
- Alle Triggerargumente, die in der *Austrittsaktion* eines Zustands benutzt werden sollen, müssen in der Aktion sowie in jedem Trigger definiert sein, der zu einem von dem Zustand wegführenden Übergang gehört, da sie von diesen Triggern aufgerufen wird.
- Alle Triggerargumente, die in der *statischen Aktion* eines Zustands benutzt werden sollen, müssen in der Aktion sowie in jedem Trigger des Zustandsautomaten definiert sein. Jedes Triggerereignis, das keinen Übergang aus dem aktiven Zustand herbeiführt, bewirkt die Ausführung von dessen statischer Aktion.
- Alle Triggerargumente, die in der *Bedingung* oder *Übergangsaktion* eines Übergangs benutzt werden sollen, müssen in der Aktion/Bedingung sowie in dem zum Übergang gehörigen Trigger definiert sein, da sie von diesem Trigger aufgerufen werden.

Wird eine dieser Regeln verletzt, wird eine Fehlermeldung ausgegeben.

Nach der Integration in eine andere Komponente können den Triggerargumenten Werte zugewiesen werden. Anders als bei Ein- und Ausgängen wird hier nur ein Sequenzaufruf (s. Kapitel 6.3) für alle Argumente benötigt.



Außerdem können 'normale' öffentliche Methoden für einen Zustandsautomaten genauso wie bei allen anderen Klassen definiert werden. Diese öffentlichen Methoden bieten einige zusätzliche Möglichkeiten. Sie können z.B. sowohl von außerhalb des Zustandsautomaten aufgerufen werden als auch direkt in den Zuständen und Übergängen. Ihre Argumente und Rückgabewerte können anstelle von Ein- und Ausgängen für die Kommunikation mit anderen Komponenten verwendet werden. Auch diesem Fall wird nur ein einziger Sequenzaufruf für die ganze Methode benötigt (was allerdings keine Laufzeitersparnis bringt). Sie haben so außerdem die Möglichkeit, die Eingangswerte aufzubereiten, ehe sie im Zustandsautomat verwendet werden.



Weitere Anwendungen für diese öffentliche Methoden sind z.B. Reset-Funktionen, die von innerhalb und außerhalb des Zustandsautomaten aufgerufen werden können, oder Zähler, die Ereignisse innerhalb und außerhalb des Zustandsautomaten erfassen müssen. Teile des Zustandsautomaten, integrierte Klassen, können mit Hilfe der öffentlichen Methoden in einem anderen Zeitraster gerechnet werden. Ein zweiter Zustandsautomat kann in den ersten integriert werden und - ohne Verwendung eines zusätzlichen Triggers - ebenfalls in einem anderen Zeitraster gerechnet werden, indem er über eine öffentliche Methode aufgerufen wird.

3 Typen und Elemente

Jeder Algorithmus in einer Komponente arbeitet mit *Elementen*. Ein Element enthält eine Datenangabe und stellt eine Schnittstelle für den Zugriff auf seine Daten oder die Rückgabe des Wertes einer Berechnung (z. B. Interpolation einer Kennlinie) zur Verfügung. Elemente sind streng typisiert, d.h. jedes Element ist von einem festen Typ. Da mehr als nur ein einziges Element von einem gegebenen Typ existieren kann, wird ein Element als eine *Instanz* eines gegebenen Typs bezeichnet.

ASCET weist eine Anzahl von Basistypen auf, die direkt verwendet werden können, wie etwa diskrete oder stetige Variable, Arrays, Matrizen oder Kennlinien und -felder. Neue, benutzerdefinierte Typen können in Form von Klassen zum System hinzugefügt werden. Klassen sind komplexe Typen, sie haben eine komplexe Struktur, da sie gewöhnlich aus anderen Typen aufgebaut sind (aus Basistypen sowie anderen komplexen Typen). Die Typen können wie im folgenden Diagramm angegeben klassifiziert werden:

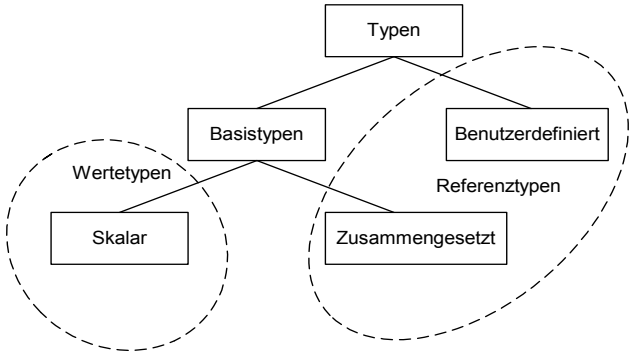


Abb. 3-1 Klassifikation von Datentypen in ASCET

Da die Modellierung in ASCET auf der physikalischen Ebene stattfindet, sind die Typen gleichfalls 'physikalische' Typen. Elemente sind nur während der Implementierungsphase, die von der Modellierungsphase unabhängig ist, auf einen spezifischen Datentyp festgelegt (z. B. `unsigned int8`).

Die physikalische Definition eines Elements muss die folgenden Informationen enthalten:

- Name des Elements
- Modelltyp
- Art des Elements
- Geltungsbereich des Elements

Die Optionen, die für jede der vorgenannten Kategorien zur Verfügung stehen, werden in den folgenden Abschnitten im einzelnen beschrieben.

Beim Definieren eines Elements können zusätzliche Informationen über die Baueinheit und ein Kommentar hinzugefügt werden, um eine aussagefähige Dokumentation des Modells zu generieren. Diese Informationen haben keine Auswirkung auf das physikalische Modell.

3.1 Basismodelltypen

In ASCET gibt es zwei Kategorien von Basismodelltypen: skalare Typen und zusammengesetzte Typen.

3.1.1 Skalare Typen

Die wichtigsten unter den Basismodelltypen sind die skalaren Typen. ASCET unterstützt vier skalare Basistypen, die in den verschiedenen Fenstern von ASCET durch die jeweiligen nebenstehenden Symbole dargestellt werden:

- *Continuous* wird für stetige physikalische Werte verwendet, die unendlich groß sein können und eine beliebig feine Auflösung haben. Dieser Typ ist für die Modellierung von Variablen wie Temperatur, Geschwindigkeit usw. geeignet; er wird als Modelltyp `cont` bezeichnet.

- *Signed discrete* wird verwendet, um ganze Zahlen von beliebiger Größe zu modellieren; dieser Typ wird als Modelltyp `sdisc` bezeichnet.

- *Unsigned discrete* wird verwendet, um nichtnegative ganze Zahlen von beliebiger Größe zu modellieren; dieser Typ ist zur Modellierung von Dingen geeignet, wie der Zylinderzahl eines Motors; er wird als Modelltyp `udisc` bezeichnet.

- *Logical* wird verwendet, um logische Informationen zu modellieren, z. B. ob ein bestimmtes System aktiv ist oder nicht; dieser Typ wird als Modelltyp `log` bezeichnet.

Die vier skalaren Basistypen sind Wertetypen. Bei jeder Verwendung eines Elements eines solchen Typs wird nicht das Element selbst als Objekt verwendet, sondern sein Wert. Falls erforderlich, wird eine automatische Typenumwandlung zwischen den arithmetischen Typen `cont`, `sdisc` und `udisc` vorgenommen.

Ebenso wie komplexe Typen (Klassen) verfügt jeder Basistyp über eine Schnittstelle, d.h. über Methoden, um auf ihn zuzugreifen. Für die Basismodelltypen sind diese Methoden fest, die Schnittstelle kann nicht geändert werden.

Skalare Typen verfügen über zwei einfache Zugriffsmethoden zum Zugriff auf den Wert, der in einem Element des skalaren Basistyps gespeichert ist, d.h. zum Schreiben eines neuen Werts in das Element und zum Lesen des aktuellen Werts aus dem Element:

- `set (type a)`: Diese Methode nimmt einen Wert, z. B. den Wert `a`, und überschreibt den Wert des Elements mit diesem Wert. Falls der Typ des Werts nicht zum Typ des Elements passt, wird automatisch eine Typumsetzung durchgeführt.
- `get()`: Diese Methode gibt den aktuellen Wert des Elements zurück. Der zurückgegebene Wert ist von demselben Typ wie das Element selbst.

Zugriffsmethoden bei Basistypen werden automatisch aufgerufen, wenn in einem Ausdruck ein Elementname verwendet oder wenn eine Zuweisung vorgenommen wird. Sie müssen nicht explizit codiert werden.

3.1.2 Zusammengesetzte Typen

Zusammengesetzte Typen sind Basistypen, die aus skalaren Basistypen aufgebaut sind. In ASCET stehen die folgenden zusammengesetzten Typen zur Verfügung:

- Array ()
- Matrix ()
- Kennlinie ()
- Kennfeld ()
- Verteilung ()

Zusammengesetzte Typen bestehen aus skalaren Basistypen. Arrays und Matrizen können aus allen vier skalaren Typen bestehen, Kennlinien, Kennfelder und Verteilungen nur aus den drei arithmetischen Typen. Im Unterschied zu den skalaren Basistypen sind zusammengesetzte Typen *Referenztypen*. Wenn zwei Variable eines Referenztyps einander zugewiesen werden, werden nicht die Werte zugewiesen (und kopiert), sondern die Referenzen werden der Variablen zugewiesen.

Alle Referenztypen haben Zugriffsmethoden für ihre Elemente:

- `set (reference type a)`: Dies ist eine Zuweisung der Referenz zum Referenztyp a. Nach einer solchen Zuweisung sind beide Elemente (das zugewiesene ebenso wie das zuweisende) dasselbe Element!
- `get`: Dies bewirkt die Rückgabe einer Referenz an das Element eines zusammengesetzten Typs.

Die Übergabe von Parametern in Aufrufen von Methoden funktioniert auf dieselbe Weise wie Zuweisungen. Eine Referenz wird an das Element übergeben. Infolgedessen spiegelt sich eine Änderung des Parameters, zum Beispiel indem ihm ein Wert zugewiesen wird, auch außerhalb der Methode wieder. Dieser Mechanismus ist äquivalent zu einem „Aufruf über eine Referenz“ in Programmiersprachen wie C.

Array

Ein Array ist ein Basistyp, der eine Anzahl skalarer Werte von demselben skalaren Basistyp, z. B. `continuous` oder `logical`, enthält. Die Position eines skalaren Wertes innerhalb eines Array wird durch seinen zugehörigen Indexwert angegeben, der vom Modelltyp `unsigned discrete` sein muss. Die Größe eines Array ist auf 2048 begrenzt und muss statisch definiert werden. Der Array-Index nimmt Werte zwischen `0` und `Größe-1` an.

Die Schnittstelle eines Array besteht aus den folgenden Methoden:

- `void setAt(scalar type a, udisc i)`: Die Zuweisung des skalaren Wertes `a` zur Position `i` im Array.
- scalar type `getAt(udisc i)`: Gibt den Wert an der Position `i` des Array zurück.

Arrays aus nichtskalaren Basistypen oder komplexen (benutzerdefinierten) Typen stehen nicht zur Verfügung.

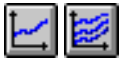
Matrix

Eine Matrix ist einem Array ähnlich. Eine Matrix ist jedoch zweidimensional, es sind zwei Indizes erforderlich. Der Typ der Indizes ist derselbe wie bei einem Array (`udisc`). Die Größe ist für jede Dimension auf `63` begrenzt, d.h. die Indizes nehmen Werte zwischen `0` und (maximal) `62` an.

Die Schnittstelle einer Matrix besteht aus den folgenden Methoden:

- `void setAt(`scalar type `a`, `udisc i`, `udisc j`): Die Zuweisung des skalaren Wertes `a` zur Position `(i,j)` in der Matrix.
- type `getAt(udisc i, udisc j)`: Gibt den Wert an der Position `(i,j)` der Matrix zurück.

Matrizen aus nichtskalaren Basistypen oder benutzerdefinierten Typen stehen nicht zur Verfügung.



Um nichtlineare Steuerungstechnik zu unterstützen, stehen in ASCET ein- und zweidimensionale *Kennfelder* zur Verfügung. Erstere werden auch Kennlinien genannt. Kennlinien und -felder werden verwendet, um einen Wert in Abhängigkeit von einem oder zwei anderen Werten zu beschreiben, wenn entweder die funktionale Abhängigkeit nicht exakt bekannt ist oder eine Berechnung der Funktion mit einem hohen Rechenaufwand verbunden wäre.

Ein Beispiel für eine Kennlinie ist der Durchlass einer Diode in Abhängigkeit von der Eingangsspannung. Dieses charakteristische Verhalten wird durch eine Kurve beschrieben. Die Kurve ist als eine Tabelle von Abtastpunkten dargestellt, von denen jeder mit einem Abtastwert verknüpft ist. Die Abtastpunkte stellen die x-Achse eines Funktionsgrafen dar, und die Abtastwerte stellen die beschriebene Kurve dar.

Entsprechend wird ein Kennfeld durch eine zweidimensionale Tabelle von Abtastpunkten für Paare von Eingangswerten dargestellt, wobei mit jedem Paar von Abtastpunkten ein Abtastwert verknüpft ist. Die Größe von Kennfeldern ist auf 2048 Abtastpunkte für Kennlinien oder 63 Abtastpunkte auf jeder Achse für Kennfelder begrenzt. Kennlinien und -felder sind stets Parameter, d.h. sie können nur vom Inneren des Modells aus gelesen werden.

Jedes Kennfeld ist auch mit einer Interpolations- und Extrapolationsroutine verknüpft. Diese Routinen legen fest, wie der Ausgabewert eines Kennfeldes aus dem (den) Eingabewert(en) bestimmt wird.

ASCET stellt zwei verschiedene Interpolationsarten bereit: Bei gerundeter Interpolation wird der Wert zwischen zwei Abtastpunkten auf den Abtastwert am unteren (linken) Abtastpunkt gesetzt, und bei linearer Interpolation wird der Wert von einer Geraden zwischen den Abtastwerten abgeleitet.

Bei Anwendungen in Steuereinheiten ist Interpolation ein sehr zeitaufwendiger Vorgang. Sie besteht aus zwei Operationen: aus der Suche nach dem richtigen Intervall von Abtastpunkten und der Berechnung der Interpolationsfaktoren, und zweitens aus der Berechnung des Ausgabewertes aus den Interpolationsfaktoren.

Die Berechnung von Interpolationsfaktoren kann in ASCET unter Verwendung von zwei speziellen Typen von Kennfeldern optimiert werden: Gruppentabellen und feste Tabellen. *Gruppentabellen* enthalten keine Abtastpunktverteilung, sondern referenzieren eine Verteilung von Abtastpunkten. Verteilungen können von vielen Gruppentabellen gemeinsam benutzt werden. Die Berech-

nung der Interpolationsfaktoren wird für die Verteilung nur einmal durchgeführt, und nur die Berechnung des Ausgabewertes erfolgt für jede Gruppentabelle gesondert.

Eine *Verteilung* (engl. distribution) ist immer eine eindimensionale Tabelle von Abtastpunkten. Zweidimensionale Gruppentabellen referenzieren daher zwei Verteilungen.

Feste Tabellen haben eine äquidistante Verteilung, d.h. die Abtastpunkte haben einen konstanten Abstand voneinander. Dies macht die Berechnung von Interpolationsfaktoren wesentlich schneller. Der Speicherbedarf ist gleichfalls geringer, da anstelle einer Liste von Abtastpunkten nur ein Versatz und ein Abstand gespeichert werden müssen. Es gibt jedoch keine Kombination von festen Tabellen und Gruppentabellen.

Die Schnittstelle eines Kennfeldes hängt von seiner Dimension ab und davon, ob es sich um eine normale, feste oder Gruppentabelle handelt. Es gibt im wesentlichen drei Methoden:

- `void search (`arithmetic type `a)`: Diese Methode wird auf die Verteilung einer Kennlinie angewandt. Dabei werden die richtigen Stützpunkte gesucht, und es werden die Interpolationsfaktoren berechnet. Für zweidimensionale Tabellen (Kennfelder) sind zwei Parameter vorhanden, d.h. `void search (`arithmetic type `a`, arithmetic type `b)`.
- arithmetic type `interpolate`(): Diese Methode interpoliert den Wert der Kennlinie oder des Kennfeldes mittels der Interpolationsfaktoren und der Werte an den zugehörigen Stützpunkten.
- arithmetic type `getAt`(arithmetic type a) ist die Kombination der Methoden search und interpolate. Für zweidimensionale Tabellen sind zwei Parameter vorhanden, d.h. void `getAt` (arithmetic type a, arithmetic type b).

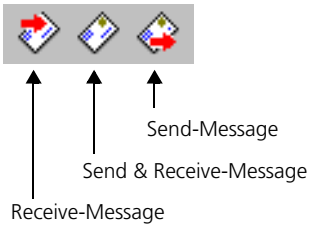
Für eine Tabelle müssen der Parameter und der Ausgabewert von demselben arithmetischen Typ sein, z. B. gibt es kein Kennfeld, bei dem stetige und diskrete Typen gemischt werden können. Die Teilung der Methode `getAt` in die Methoden `search` und `interpolate` ist nur für Gruppentabellen sinnvoll.

Eine Verteilung hat nur die Methode `search`. Eine Gruppentabelle hat nur die Methode `interpolate`. Ein reguläres oder festes Kennfeld hat alle drei Methoden.

3.1.3 Echtzeit-Sprachkonstrukte

ASCET stellt eine Reihe von Sprachkonstrukten für Echtzeit-Anwendungen bei der Beschreibung von Komponenten zur Verfügung.

Messages



Messages bilden die Eingangs- und Ausgangsvariablen von Prozessen und werden auf dieselbe Weise für die Interprozesskommunikation verwendet wie skalare Basistypen. Im Gegensatz zu globalen Variablen sind Messages *geschützte Variable* bei präemptivem Scheduling. Wenn zwei konkurrierende Prozesse auf dieselbe Message zugreifen, ist die Datenkonsistenz sichergestellt, da jeder Prozess mit seiner eigenen Kopie arbeitet. Messages stehen nur in Modulen zur Verfügung. In Abhängigkeit von der Verwendung sind drei verschiedene Typen von Messages zu unterscheiden:

-  *Receive-Messages* können nur gelesen werden. Receive-Messages werden als Eingaben für ein Modul verwendet.
-  In *Send-Messages* kann nur geschrieben werden. Sie werden für die Ergebnisse der Berechnungen eines Moduls verwendet.
-  *Send&Receive-Messages* können gelesen werden, und in sie kann geschrieben werden.

Ressourcen

Eine Ressource (Typensymbol ) stellt einen Teil einer Anwendung dar, der nur ausschließlich verwendet werden kann, z. B. Timer oder spezielle Vorrichtungen. Es existieren zwei Methoden, um auf eine Ressource zuzugreifen:

- `void reserve()`: Die Ressource wird reserviert, das heißt, der Zugriff auf die Ressource wird gesperrt.
- `void release()`: Die Ressource wird freigegeben, das heißt, der Zugriff auf die Ressource wird wieder ermöglicht.

Durch Ausführung der Methode `reserve` wird der Zugriff auf die Ressource gesperrt, und es wird ein ausschließlicher Zugriff in einer präemptiven Umgebung garantiert, d.h., wenn der laufende Prozess unterbrochen wird und ein anderer Prozess auf die Ressource zugreifen möchte, wird der Zugriff verweigert.

Wenn der Zugriff auf die Ressource nicht mehr erforderlich ist, kann die Ressource mit Hilfe der Methode `release` freigegeben werden. Dadurch wird die Ressource wieder für andere Komponenten zugänglich. Um Blockierungen oder Prioritätsumkehrungen zu vermeiden, ist die Reservierung einer Ressource mit der Obergrenze der Priorität des entsprechenden Prozesses verknüpft. Ressourcen sind immer globale Elemente.

Der Parameter `dT`

Bei Anwendungen in der Steuer- und Regeltechnik hängt das Ergebnis der Berechnungen innerhalb einer Komponente oft vom Wert der Abtastfrequenz ab. Um die Algorithmen für alle Abtastfrequenzen einheitlich zu beschreiben, stellt ASCET den Systemparameter `dT` (Typensymbol) bereit. Der Wert dieses Parameters wird durch das Betriebssystem zur Verfügung gestellt und stellt die Zeitdifferenz seit der letzten Aktivierung der zur Zeit aktiven Task dar.

Hinweis

Der Name `dT` *ist für den Systemparameter reserviert. Sie können kein anderes Element mit dem Namen dT erzeugen; da bei reservierten Schlüsselwörtern zwischen Groß- und Kleinschreibung generell nicht unterschieden wird, sind auch die Namen* `DT`*,* `dt` *und* `Dt` *reserviert.*

3.1.4 Spezielle Typen

Neben den bisher genannten Typen gibt es noch eine Reihe weiterer, die in diesem Abschnitt beschrieben werden.

Enumeration

Enumerationen (Typensymbol) sind eindeutige Typen mit Werten aus einer Menge von bekannten Konstanten, die Aufzählungskonstanten (Enumeratoren) genannt werden.

Literale

Literale sind Zeichenketten, die einen festen Wert eines skalaren Basistyps darstellen, der in einem beliebigen Ausdruck verwendet werden kann. Der Wert eines Literals ist entweder eine Zahl (`discrete` oder `continuous`), ein logischer Wert (`true`, `false`) oder eine Zeichenkette (`String`). Im Blockdiagrammeditor sind die Werte `string`, `true`, `false`, `0.0` und `1.0` vordefiniert.

3.2 Die Art von Elementen

Jedes Element hat eine Art (engl. *kind*). Die Art eines Elements beschreibt, wie das Element verwendet wird, als Variable, Parameter, Konstante oder Systemkonstante. Als weitere Art gibt es Implementation-Casts.

- *Variablen* speichern Werte, die von innerhalb des Modells aus gelesen und geschrieben werden können, d.h., auf ihnen kann eine Lese- und eine Schreiboperation ausgeführt werden.

  Im Steuergerät können sie sowohl im flüchtigen (*volatile*) als auch im nicht-flüchtigen (*non-volatile*) Speicher abgelegt werden; bei neu angelegten Variablen ist *volatile* voreingestellt.

- *Parameter* speichern Werte, die von innerhalb des Modells aus nur gelesen werden können. Parameter können auch kalibriert werden, d.h. in sie kann von außerhalb des Modells aus geschrieben werden. Gegebenenfalls sind dafür spezielle Voraussetzungen erforderlich, z.B. der Anschluss an ein Kalibrierwerkzeug.

  Parameter (auch Kennlinien/-felder) haben automatisch das Attribut *non-volatile*; sie werden im nicht-flüchtigen Speicher des Steuergeräts abgelegt.

- *Konstanten* speichern Werte, die von innerhalb des Modells aus nur gelesen werden können. Im Gegensatz zu Parametern können Konstanten nicht von außerhalb des Modells aus geändert werden, sondern werden zum Zeitpunkt der Spezifikation festgelegt. Ebensowenig können Konstanten implementiert werden.
  Konstanten werden im generierten C-Code als `define`-Anweisung angelegt, tauchen aber nicht notwendigerweise explizit erkennbar im Code auf. Wenn beispielsweise Konstanten mit Umquantisierungen verrechnet werden, tauchen sie nicht explizit auf.

- *Systemkonstanten* werden wie Konstanten verwendet und ebenfalls als `define`-Anweisung generiert. Anders als Konstanten können Systemkonstanten implementiert werden. Sie sind immer explizit im Code vorhanden.

  Systemkonstanten können mit dem Befehl **Extras → Convert System Constants to Constants** im Komponentenmanager in normale Konstanten umgewandelt werden.

  Tab. 3-1 fasst die Unterschiede hinsichtlich der Verwendung von Variablen, Parametern und Konstanten zusammen.

- *Implementation-Casts* (siehe Kapitel 4.2.4) geben dem Anwender die Möglichkeit, an frei wählbaren Stellen einer Berechnung oder eines Datenflusses die Implementierung gezielt vorzugeben. Anders als Variablen und Parameter belegen Implementation-Casts dabei keinen Speicherplatz, haben also auch keine speichernde Wirkung im Modell und können nicht kalibriert werden.

  Implementation-Casts haben keine Daten; sie haben immer den Modelltyp `cont`, eine skalare Dimension und den Geltungsbereich *lokal* (siehe Kapitel 3.3). Anders als bei den anderen Elementen können die Eigenschaften von Implementation-Casts nicht bearbeitet werden.

	Modell	**Experiment / Kalibrierwerkzeug**	**Implementierung**
Variable	r-w	r-w	ja
Parameter	r	r-w	ja
Systemkonstante	r	r	ja
Konstante	r	r	nein
Implementation-Cast	—	—	ja

Tab. 3-1 Übersicht: Variable, Parameter, Systemkonstante, Konstante, Implementation-Cast

Die Arten von Elementen werden in den verschiedenen ASCET-Fenstern (z.B. im Feld „3 Contents“ des Komponentenmanagers) durch Symbole dargestellt.

	Geltungsbereich				
	importiert	**exportiert**	**lokal**	**abhängig**	**virtuell**
Variablen[a]					 [b]
Messages					
Parameter[c]					*[d]
(System-)Konstanten					
Implementation-Casts					
dT					

a: auch Array, Matrix, Enumeration
b: unabhängig vom Geltungsbereich; siehe Seite 102
c: auch Kennlinie/-feld, Verteilung
d: Symbol wird von sonstigen Einstellungen (Geltungsbereich usw.) übernommen

Tab. 3-2 Symbole für die verschiedenen Arten und Geltungsbereiche

Temporäre Variable

Um eine mehrfache Ausführung innerhalb derselben Methode oder desselben Prozesses zu vermeiden, können für jeden Operator oder Methodenaufruf temporäre Variablen spezifiziert werden. In diesem Fall wird der Wert des betreffenden Ausdrucks nur einmal für jede Methode oder Prozess, in dem er verwendet wird, in einer temporären Variablen gespeichert. Bei erneuter Verwendung des Ausdrucks in der betreffenden Methode oder Prozess wird er nicht neu ausgewertet, sondern die temporäre Variable wird wiederverwendet.

Jeder Spezifikationseditor kann eine temporäre Variable erzeugen. Eine temporäre Variable verfügt über keinen Anfangswert, sie erhält ihren Wert erst durch die Zuweisung eines Ausdrucks. ASCET verwaltet intern die temporären Variablen und sorgt für die eindeutige Zuweisung (z.B. in den Ausführungspfaden bei einer `IF`-Anweisung), damit beim späteren Gebrauch der temporären Variablen keine undefinierbaren Werte verwendet werden. Der Wert bleibt solange gültig, bis eine neue Zuweisung auf die temporäre Variable erfolgt.

Das Beispiel zeigt die temporäre Variable `t`, die den Wert aus der Addition von `a + b` speichert und wiederverwertet:

```
t = a + b;
c = t;
d = t;
```

Virtuelle Variablen / Parameter

Virtuelle Variablen / Parameter sind nur für die Spezifizierungsplattform verfügbar, d.h. für die Generierung des Codes finden sie keine Beachtung. Sie werden vielmehr zum besseren Verständnis der Bedeutung von Modellelementen in der Spezifikation verwendet.

Virtuelle Variablen sind immer abhängig von anderen virtuellen oder nicht-virtuellen Variablen. Sie sind nichts weiter als ein Alias für nicht-virtuelle Variablen. Mathematische Abhängigkeiten wie etwa Formeln sind nicht erlaubt; beim Bearbeiten der Daten von virtuellen Variablen ist daher die Identität (`var_virtual = var_real`) fest vorgegeben.

Parameter hingegen, die als virtuell deklariert sind, sind nicht automatisch abhängig von anderen Parametern.

Abhängige Parameter

Modellparameter können über eine mathematische Abhängigkeit mit anderen System- oder Modellparametern verbunden sein. Das Verstellen von Parametern kann somit zu Inkonsistenzen führen.

Um mögliche Inkonsistenzen durch das Verstellen von Parametern zu vermeiden, ist es in ASCET möglich, die Abhängigkeit eines Parameters in den Spezifikationseditoren zu spezifizieren. Die Abhängigkeit des Parameters wird in Form einer mathematischen Formel dargestellt.

Hinweis

*Es gibt **keine** abhängigen Variablen.*

3.3 Der Geltungsbereich von Elementen

Manche Elemente werden für den Datenaustausch zwischen verschiedenen Komponenten verwendet. Um dies zu realisieren, können Elemente aus einer Komponente (oder aus einem Projekt) exportiert und in eine andere Komponente importiert werden. Dabei erfolgt die Anpassung über Namen. Der Geltungsbereich (engl. *scope*) jedes Elements kann als einer der folgenden definiert werden:

- *Lokale* Elemente können nur innerhalb der Komponente verwendet werden, die sie definiert, d.h. in allen Methoden oder Prozessen dieser Komponente.

- *Importierte* Elemente sind in irgendeiner anderen Komponente oder einem anderen Projekt definiert, können jedoch in der Komponente verwendet werden, die sie importiert. Die Eigenschaften eines importierten Elements können nur im Kontext der Komponente geändert werden, die das Element definiert und exportiert.
- *Exportierte* Elemente sind in einer Komponente definiert, und auf sie kann durch alle anderen Komponenten durch Importieren dieses Elements zugegriffen werden.
- *Methoden-/prozesslokale* Elemente können nur in der Methode/dem Prozess verwendet werden, die/der sie definiert. Methoden-/prozesslokale Elemente sind nicht statisch und haben keinen Datensatz.

3.4 Benutzerdefinierte Modelltypen

Elemente können auch benutzerdefinierte Modelltypen sein, d.h. Module oder Klassen. Benutzerdefinierte Modelltypen sind immer Referenztypen. Die Schnittstelle ist durch die Schnittstelle dieser Komponente definiert.

Der Geltungsbereich eines benutzerdefinierten Typs kann derselbe sein wie der der Basistypen, nämlich importiert, exportiert, lokal und methodenlokal. Ebenso wie Argumente erfolgt für methoden-/prozesslokale Elemente von einem Referenztyp keine Instanzbildung, sondern es wird eine Referenz auf sie festgelegt. Das bedeutet, dass bei Verwendung eines methoden-/prozesslokalen Elements von einem Referenztyp eine Zuweisung zu diesem Element jeder weiteren Verwendung dieses Elements vorausgehen muss.

Die Art eines Elements ist für benutzerdefinierte Modelltypen unwesentlich. Benutzerdefinierte Modelltypen werden immer als Variable behandelt, d.h. es gibt keine Einschränkung der Schnittstelle von innerhalb des Modell aus.

4 Daten und Implementierungen

Im vorigen Kapitel wurden als die Bestandteile einer Komponente die Menge von Elementen, die Schnittstelle der Komponente und die funktionale Beschreibung der Methoden oder Prozesse in Form von Algorithmen genannt.

Im vorliegenden Kapitel werden zwei zusätzliche Bestandteile einer Komponenten-Spezifikation vorgestellt: Daten und Implementierung. Sowohl Daten als auch Implementierung gehören zu den Elementen in einer Komponente, d. h. beide beschreiben Eigenschaften der Elemente.

Die Methode der Verwendung separater Beschreibungen für Daten und Implementierungen ist in Standard-Programmiersprachen gewöhnlich nicht zu finden. Dort ist die Datenzuordnung von Variablen Bestandteil der Funktionsspezifikation, d.h. des Programmcodes.

Die Daten einer Komponente beschreiben die physikalischen Werte, mit denen die Elemente der Komponenten initialisiert werden. Daten enthalten physikalische Informationen und sind daher Bestandteil der physikalischen Spezifikation der Komponente.

Auch unterscheiden Standard-Programmiersprachen gewöhnlich nicht zwischen der Implementierung einer Funktionsspezifikation und der Funktionsspezifikation selbst. Die Funktionsspezifikation ist gewöhnlich mit ihrer Implementierung identisch.

4.1 Daten

Die Daten einer Komponente beschreiben, wie die Elemente einer Komponente initialisiert werden müssen. Somit beziehen sich Daten auf die Elemente einer Komponente.

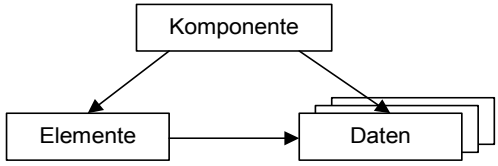


Abb. 4-1 Eine Komponente mit mehreren Datensätzen

Die Daten werden von den Elementen getrennt gehalten, weil eine Komponente mehrere Instanzen in einem Projekt haben kann, wobei die verschiedenen Instanzen auf verschiedene Datensätze für ihre Elemente zugreifen können. (Die Datensätze sind jedoch nicht Teil der jeweiligen Instanz.)

Ein Beispiel wäre ein P-Regelfilter. Jede Instanz dieses P-Regelfilters hat ihren eigenen Wert für den P-Faktor. Dies wird erreicht, indem dem P-Regler verschiedene Datensätze zugeordnet werden.

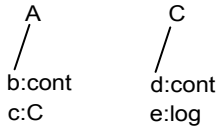
Die Spezifikation der Daten ist Teil der Spezifikation der Komponente selbst und nicht der verschiedenen Instanzen. Dies kann zu einer großen Anzahl unterschiedlicher Datensätze für eine Komponente führen, doch wenn jede Instanz ihre eigenen Daten enthielte, würde dies den Verlust eines modularen Systemkonzepts zur Folge haben.

Die Organisation der Daten für die einzelnen Elemente hängt davon ab, ob es sich um ein Grundelement oder ein komplexes Element handelt. Da Grundelemente stets innerhalb komplexer Objekte verwendet werden und niemals getrennt von diesen betrachtet werden, haben Grundelemente keine expliziten Datensätze. Die Daten für die Grundelemente sind daher Bestandteil des Datensatzes des komplexen Elements, in dem sie enthalten sind.

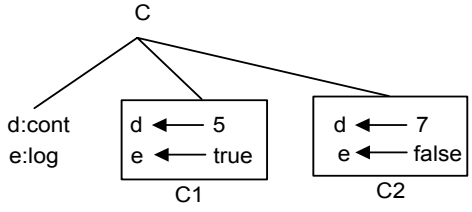
Komplexe Elemente sind die Komponenten, die vom Benutzer spezifiziert werden. Jedes komplexe Element hat seinen eigenen Datensatz. Wird ein komplexes Element in einer Komponente verwendet, so wird der Datensatz des komplexen Elements durch die Komponente referenziert. Folglich haben die Daten einer Komponente dieselbe hierarchische Struktur wie die Komponente selbst.

Datensätze haben eine Objektkennung (Objekt-ID), die verwendet wird, um die Daten einer Komponente zu referenzieren. Ebenso wie Referenzen auf benutzerdefinierte Typen ist diese Referenz nicht namensbasiert.

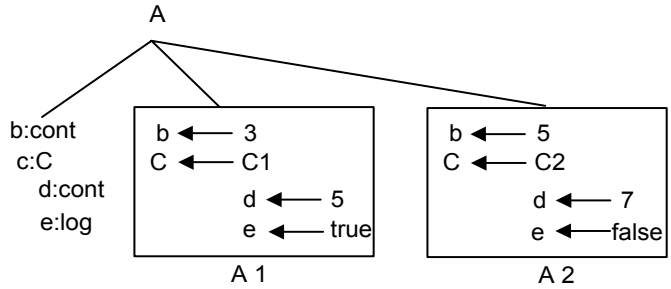
Es wird das folgende Beispiel mit den Typen A und C betrachtet:



Der Typ C hat die folgenden Datensätze:



Eine Datendeklaration für den Typ A unter Verwendung der Datensätze von C würde die folgenden Ergebnisse haben:



Die Daten für die Basistypen können direkt spezifiziert werden. Für die skalaren Typen bestehen die Daten aus einem Wert. Für zusammengesetzte Typen, wie Arrays oder Kennlinien, bestehen die Daten aus einer Wertetabelle oder einer Tabelle von Abtastpunkten und Abtastwerten.

4.2 Implementierungen

Implementierungen beschreiben, wie die Elemente einer Komponente in Code zu realisieren sind. Hierbei kommt dasselbe Schema zur Anwendung wie bei Daten:

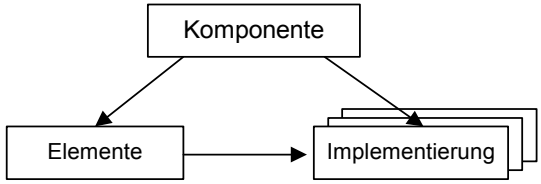


Abb. 4-2 Eine Komponente mit mehreren Implementierungen

Bei Implementierungen kommt dasselbe Referenzschema zur Anwendung wie bei Basistypen und komplexen Typen. Die Auswirkung von Implementierungen ist wesentlich stärker als die von Datensätzen. Die Implementierung eines Elements, z. B. ob ein Element vom Typ `cont` als ein Datentyp `float` oder `signed int` dargestellt wird, hat unmittelbaren Einfluss auf den Code, der durch die funktionale Beschreibung für eine Methode oder einen Prozess generiert wird.

4.2.1 Implementierungen für skalare Typen

Die Implementierung beschreibt, wie ein Element eines Basistyps in dem generierten C-Code realisiert wird. Die Implementierungsspezifikation für Elemente vom Typ `logical` ist sehr einfach, da ein logisches Element nur zwei Werte

hat, entweder wahr oder falsch. Die Implementierungsspezifikation besteht nur aus dem Datentyp. Für logische Elemente können entweder `byte`, `word` oder `long` gewählt werden.

Die Implementierungsspezifikation für die arithmetischen Typen ist wesentlich komplexer. Sie beschreibt unter anderem den Implementierungstyp, der selbst für Elemente vom Typ `continuous` ein ganzzahliger Typ sein kann. Die Implementierungsspezifikation enthält daher eine komplexe Umwandlung aus der physikalischen Domäne in die Implementierungsdomäne, wobei sich diese Domänen sehr stark unterscheiden können.

Die Unterschiede zwischen der physikalischen Domäne (z. B. Modelltyp `continuous`) und der Implementierungsdomäne sind der unendliche Bereich der physikalischen Domäne von –unendlich bis +unendlich sowie ihre beliebig feine Auflösung. In der Implementierungsdomäne dagegen ist der Bereich durch die Wortlänge begrenzt, und die Auflösung ist nicht beliebig fein, sondern mit 1 festgelegt.

Um eine Umwandlung zwischen der physikalischen Domäne und der Implementierungsdomäne zu ermöglichen, muss der Bereich der physikalischen Domäne begrenzt werden. Daher muss jedem Element ein Intervall für die betreffenden physikalischen Werte zugeordnet werden. Die Auflösung muss ebenfalls begrenzt werden. Daher muss jedes Element eine feste Auflösung erhalten, die Quantisierung.

Sei zum Beispiel A ein Wertebereich in der physikalischen Domäne, A = [-1, 0,5], und es werde eine Quantisierung von q = 0,2 angenommen.

Das Ergebnis der Begrenzung des Bereichs auf ein Intervall sowie der Quantisierung ist eine Beschränkung der Werte eines Elements auf eine endliche Menge von äquidistanten Werten.

A_q = {-1, -0,8, -0.6, -0.4, -0.2, 0, 0.2, 0.4}

Diese endliche Wertemenge kann nun auf einen ganzzahligen Bereich abgebildet werden:

A_{int} = {-5, -4, -3, -2, -1, 0, 1, 2}

Dies entspricht einer linearen Umrechnungsformel aus der physikalischen Domäne in die Implementierungsdomäne von der Art `impl = 5 * phys`. Der Datentyp für die ganzzahlige Variable wird aus dem ganzzahligen Bereich automatisch bestimmt. Im vorliegenden Beispiel würde der Datentyp `signed int8` gewählt.

Wenn der Bereich des physikalischen Elements einen Versatz aufweist, der größer als null ist, kann das zugehörige ganzzahlige Intervall nur wenige Werte enthalten, doch es muss ein großer Datentyp verwendet werden.

Es werde zum Beispiel der Bereich A = [120, 130] der physikalischen Domäne und eine Quantisierung von q = 0,5 betrachtet. Eine lineare Umsetzung würde einen ganzzahligen Bereich A_{int} = {240, … , 260} liefern.

Der Typ für die ganzzahlige Variable ist in diesem Falle `unsigned int16`, obwohl die Anzahl der Werte auch in eine Variable vom Typ `int8` passen würde.

Um dies zu implementieren, kann eine allgemeine lineare Umrechnungsformel mit einem Versatz spezifiziert werden. Im obigen Beispiel würde eine Umrechnungsformel des Typs

```
impl = 2 * phys - 240
```

zu einem ganzzahligen Intervall von {0,…,20} führen, und eine Variable vom Datentyp `unsigned int8` wäre ausreichend.

Die Umrechnungsformeln werden nicht im Kontext einer Komponente spezifiziert, sondern im Kontext eines Projekts. Dies erleichtert die Verwendung ein und derselben Umrechnungsformeln für verschiedene Komponenten. Ferner genügt dies den Bedingungen des ASAM-MCD-2MC-Standards.

4.2.2 Die Implementierung von zusammengesetzten Typen

Für zusammengesetzte Typen wie Arrays, Matrizen oder Kenntabellen wird die Implementierung für die Schnittstellenelemente der zusammengesetzten Typen spezifiziert, die ihrerseits von skalarem Typ sind.

Beispielsweise muss für Arrays die Implementierung für die in dem Array enthaltenen Elemente vorgegeben werden. Diese Implementierung ist sowohl für den Eingang als auch für den Ausgang des Arrays gültig. Die Implementierung für den Index ist fest, da der Index ein diskreter Modelltyp ist.

Für Kenntabellen können die Implementierung der x- und y-Punkte und der Werte der Tabelle getrennt voneinander spezifiziert werden.

4.2.3 Die Implementierung von benutzerdefinierten Typen

Die Implementierung von benutzerdefinierten Typen besteht aus den Implementierungen aller Elemente, die in der betreffenden Komponente verwendet werden.

Im Falle von Klassen müssen die Argumente und Rückgabewerte auch eine Implementierung besitzen, da der Wert eines konkreten und der eines formalen Arguments korrekt aneinander angepasst werden müssen. Dies geschieht für Argumente von einem skalaren Typ automatisch.

Diese automatische Anpassung funktioniert bei Argumenten von zusammengesetzten oder komplexen Typen nicht. Wenn solche Argumente verwendet werden, müssen die Implementierung des formalen Arguments und des konkreten Arguments übereinstimmen. Hier ist keine automatische Anpassung möglich, da diese Elemente als Referenzen übergeben werden.

Temporäre Elemente haben keine explizite Implementierung, sondern ihnen wird durch den Codegenerierungs-Algorithmus automatisch eine Implementierung zugewiesen. Es ist wichtig, dass eine Zuweisung zu dieser Variablen (z. B. eine Initialisierung) jeder anderen Verwendung von ihr vorangeht.

Methoden- und prozesslokale Elemente können wie temporäre Elemente automatisch implementiert werden, sie können aber auch explizit implementiert werden (s. ASCET Benutzerhandbuch, Kapitel „Implementierung von methoden-/prozesslokalen Variablen"). Die Implementierung bleibt innerhalb der Methode/des Prozesses erhalten.

4.2.4 Implementation-Casts

Mit ASCET 5.0 wurde ein neuer primitiver Elementtyp eingeführt – der *Implementation-Cast*. Implementation-Casts geben dem Anwender die Möglichkeit, Einfluss auf die Implementierung von Zwischenergebnissen in arithmetischen Rechenketten zu nehmen. Auf diese Weise kann der Anwender Wissen über bestimmte physikalische Zusammenhänge (dass z.B. an einer definierten Stelle im Modell ein bestimmter Wertebereich nicht überschritten wird) im Modell darstellen, ohne dass dabei notwendigerweise physikalisch Speicherplatz belegt wird.

Hinweis

*Implementation-Casts können **nicht** in Verbindung mit logischen Elementen verwendet werden.*

Ein kleines Beispiel soll die Idee dieser Funktionalität nochmals verdeutlichen.

In einer einfachen arithmetischen Spezifikation werden zwei Variablen a und b addiert, das Ergebnis der Addition mit dem Literal 2 multipliziert und das Ergebnis der Multiplikation einer Variablen c zugewiesen.

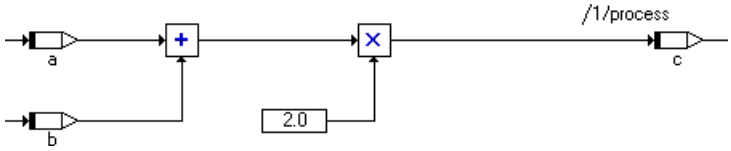


Abb. 4-3 Einfache Berechnung ohne Implementation-Cast

Den Variablen a, b und c wurde bei Implementierung der Typ `int16` zugewiesen; alle drei schöpfen den gesamten möglichen Wertebereich aus. Im obigen Beispiel würde der Codegenerator daher eine 32 Bit breite temporäre Variable anlegen und diese vor der Zuweisung auf c durch eine Rechtsverschiebung auf einen für `int16` geeigneten Wertebereich umquantisieren.

Wenn nun der Anwender die Information besitzt, dass die Summe aus a und b maximal ein 16 Bit breites Ergebnis ist und nur den halben möglichen Wertebereich ausschöpft (z.B. aufgrund physikalischer Randbedingungen oder weil bestimmte Zusammenhänge im Modell es erzwingen), kann er dies mit Hilfe eines Implementation-Casts (siehe Abb. 4-4) angeben.

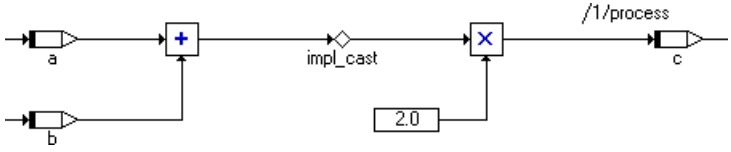


Abb. 4-4 Einfache Berechnung mit Implementation-Cast

Wenn der Anwender den Implementation-Cast mit dem Typ `int16` und dem Wertebereich `[-16384..16383]` implementiert und die Optionen **Limit Overflows** und **Limit Assignments** deaktiviert, sichert er dem Codegenerator gewisse Eigenschaften des Zwischenergebnisses zu. Damit wird die im Beispiel in Abb. 4-3 erforderliche Umquantisierung vermieden.

Ein weiterer Anwendungsfall für Implementation-Casts ist die gezielte Vergabe von Implementierungen an den Ein- und Ausgängen von Operatoren. Auf diese Weise können gezielt Arithmetische Dienste (siehe Kapitel 4.14 „Arithmetische Dienste“ im ASCET Benutzerhandbuch) für bestimmte Operatoren ausgewählt werden. In diesem Zusammenhang ersetzen Implementation-Casts die bisherigen Operatorimplementierungen.

Implementation-Casts haben, wie der Name verdeutlicht, nur Auswirkungen auf die Implementierung. Konkret heißt das, dass nur für die Codegenerierung von Experimenten (siehe Kapitel 4.8.8 im ASCET Benutzerhandbuch) vom Typ

- `Implementation Experiment` und
- `Object Based Controller Implementation`

Implementation-Casts berücksichtigt werden, während sie für

- `Physical Experiment` und
- `Quantized Physical Experiment`

ignoriert werden.

Abhängig von den Codegenerierungsoptionen (s. „Anpassen der Projekteinstellungen:“ im ASCET-Benutzerhandbuch) für die Implementierungsexperimente haben Implementation-Casts folgende Eigenschaften:

- Wenn die für das Projekt definierte maximale Bitgröße kleiner als 32 Bit ist, erlaubt die Codegenerierung für Implementation-Casts die Verwendung einer größeren Bitgröße. Wird jedoch im Code eine Variable benötigt, die die erlaubte Bitgröße überschreitet, wird eine Fehlermeldung angezeigt.

  Auf diese Weise können Implementation-Casts in rechenketten dazu verwendet werden, um Zwischenergebnisse zu spezifizieren, die über die ursprüngliche maximale Bitgröße des Controllers hinausgehen.

- Wenn am Zähler-Eingang eines Divisionsoperators ein Implementation-Cast anliegt, überschreibt dessen Implementierung die Option **Allow Double bit Size for Division Numerators**.

Eine weitere wesentliche Eigenschaft eines Implementation-Casts ist, dass für seine Realisierung bei der Codegeneration weder permanenter Speicher noch temporärer Speicher belegt wird, da Implementation-Casts weder als globales Element noch als lokale Funktionsvariable erzeugt werden. Einzig bei Implementation-Casts, die in Kombination mit einer Wertbeschränkung verwendet werden, kann es vorkommen, dass eine lokale, temporäre Funktionsvariable benötigt wird, um das Berechnungsergebnis vor der Bereichsüberprüfung zwischenzuspeichern.

Der Einsatz von Implementation-Casts ist auf den Blockdiagrammeditor und den ESDL-Editor beschränkt. Weiterhin werden diese Elemente nur für Module und Klassen (jedoch nicht für CT-Blöcke, Boolesche Tabellen oder Bedingungstabellen) sowie für die Spezifikation von Bedingungen und Aktionen in Zustandsautomaten angeboten.

4.3 Codegenerierung mit Implementierungen

Bei Wahl einer Implementierung wird der Code in Festkomma-Arithmetik generiert. Diese Festkomma-Arithmetik basiert auf ganzzahliger Arithmetik. Die Informationen der Implementierung werden auf Elemente einer Komponente angewandt. Diese Informationen bilden zusammen mit einer funktiona-

len Beschreibung, d.h. den Informationen darüber, wie die Elemente miteinander in Wechselwirkung stehen, die Grundlage für die ganzzahlige Codegenerierung.

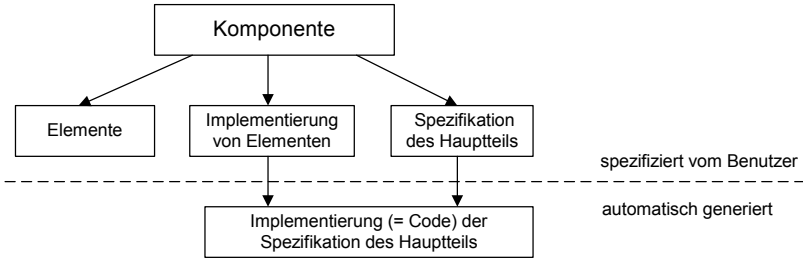


Abb. 4-5 Codegenerierung mit Implementierungen

Um das Prinzip der ganzzahligen Codegenerierung verständlicher zu machen, wird nachfolgend ein einfaches Beispiel angegeben.

Ein Beispiel – Codegenerierung für eine Addition

Es wird das folgende einfache Beispiel betrachtet:

```
c = a + b;
```

wobei `a`, `b` und `c` Modellvariable vom Typ `continuous` sind.

Die Implementierungs-Umwandlung ist linear ohne einen Versatz. Es werden die folgenden Quantisierungen verwendet: 0,01 für `a`, 0,04 für `b` und 0,05 für `c`. A, B, und C sind die entsprechenden Implementierungsvariablen für die Elemente in dem generierten C-Code.

Wenn für das obige Beispiel Code generiert wird, müssen die Quantisierungen berücksichtigt werden. Für die Werte `a` = 1, `b` = 0,6 und demzufolge `c` = 1,6 wäre das Ergebnis mit den obigen Quantisierungen A = 100, B = 15 und C = 32. Eine direkte Umsetzung des Modells auf die Implementierungsebene würde zu einem falschen Ergebnis führen (A+B = 100 + 15 = 115, was nicht mit C = 32 übereinstimmt).

Der Grund hierfür ist, dass die Quantisierung nicht berücksichtigt wird. Die obige Modellgleichung muss entsprechend der Implementierungsumwandlung umgewandelt werden. Dabei müssen die Quantisierungen von A und B angepasst werden, bevor die Addition erfolgt, und das Ergebnis dieser Addition muss an die Quantisierung von C angepasst werden. Dies führt zu dem folgenden Stück C-Code für das obige Modell:

```
C = (A + 4 * B) / 5;
```

Die Multiplikation von B mit 4 entspricht der Anpassung der Quantisierung 0,04 an 0,01, und die Division durch 5 entspricht der Anpassung der Quantisierung von 0,01 an 0,05.

4.3.1 Umwandlung von Daten bei einer Implementierung

Die Daten, die zusammen mit einem Element gespeichert werden, enthalten immer die „Modelldaten", d.h. die physikalischen Werte, doch die Implementierung muss sich auch in den Daten widerspiegeln. Im obigen Beispiel war 1 die physikalische Datenangabe (Modell-Datenangabe) für die Variable a, die Datenangabe für die Implementierungsvariable A war jedoch 100.

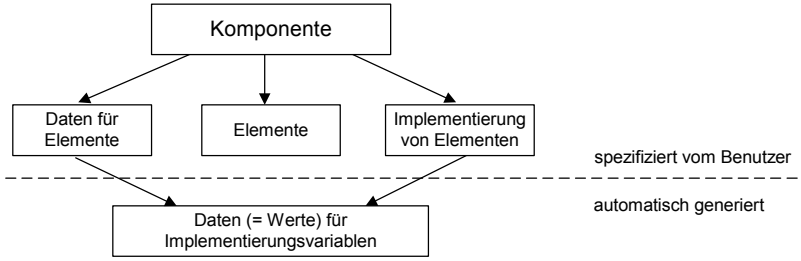


Abb. 4-6 Umwandlung von Daten

4.3.2 Allgemeine Regeln für die Implementierungsumwandlung

Die Implementierungsumwandlung betrifft arithmetische Werte. Die Werte werden in allen arithmetischen Ausdrücken angepasst, so dass die entsprechenden arithmetischen Operationen ausgeführt werden können:

Addition und Subtraktion:

Die Argumente dieser Operationen werden an eine Quantisierung angepasst. Diese Quantisierung wird durch die internen Codegenerierungs-Algorithmen bestimmt und minimiert die Anzahl der Umquantisierungen. Der konstante Versatz wird für das Ergebnis aus den Quantisierungen und dem Versatz der Argumente berechnet.

Multiplikation und Division:

Die Argumente dieser Operationen werden zuerst versatzfrei gemacht, bevor die Multiplikation oder Division erfolgen kann. Die Quantisierung muss nicht angepasst werden, sondern wird aus dem Ergebnis der Multiplikation oder Division bestimmt. Um jedoch einen Überlauf oder einen Verlust an Genauigkeit zu vermeiden, kann die Quantisierung der Argumente mit einer Zweierpotenz multipliziert werden (Verschiebungsoperationen). Diese wird ebenfalls automatisch durch den internen Codegenerierungs-Algorithmus bestimmt.

Vergleich, Minimum und Maximum:

Ähnlich wie bei der Addition werden die Argumente aneinander angepasst (sowohl hinsichtlich der Quantisierung als auch hinsichtlich des Versatzes). Minimum- und Maximum-Operator funktionieren wie der Additions-Operator.

Zuweisung:

Der Wert, der einer Variablen zugewiesen ist, wird umquantisiert, und der Versatz wird korrigiert, bevor eine Zuweisung vorgenommen wird. Dies gilt auch für die Übergabe von Argumenten.

4.4 Die Implementierung von Methoden und Prozessen

Die Einrichtungen zur Verwendung von Implementierungen (verbessert in ASCET 5.0) ermöglichen, dass Implementierungen von Methoden spezifiziert werden können. Methoden- und Prozessimplementierungen sind sowohl in ESDL als auch in Blockdiagrammen verfügbar.

Die Implementierung einer Methode oder eines Prozesses enthält Informationen über den zu verwendenden Speicher für die Ausführung einer Methode oder eines Prozesses sowie darüber, ob er während der Codegenerierung vollständig expandiert werden soll.

Im allgemeinen werden Algorithmen, die eine kurze Ansprechzeit haben sollten oder häufiger verwendet werden, im internen Speicher ausgeführt, während andere Algorithmen, die nicht sehr häufig verwendet werden, wie etwa Initialisierungsalgorithmen, im externen Speicher ausgeführt werden.

Außerdem können Methoden- und Prozessaufrufe entweder als Funktionsaufrufe dargestellt werden, oder vollständig expandiert im generierten Code (*inlining*).

5 Spezifikation in ESDL

Das vorliegende Kapitel beschreibt die allgemeinen Merkmale von ESDL, die bei der Beschreibung von Klassen und Modulen verwendet werden. Die Beschreibung ist in drei Hauptteile gegliedert.

Der erste Abschnitt enthält eine kurze Beschreibung von allgemeinen charakteristischen Eigenschaften von ESDL. In den nachfolgenden Abschnitten wird eine umfassende Beschreibung sowohl der Syntax als auch der Elemente von ESDL gegeben.

Die Unterschiede zwischen ESDL und Blockdiagrammen sowie diejenigen zwischen ESDL und den Programmiersprachen C und Java werden am Ende dieses Kapitels zusammengefasst.

Es wird vorausgesetzt, dass der Leser mit der Programmiersprache C oder Java (oder mit beiden) vertraut ist. Wenn Sie zusätzliche Informationen über C oder Java benötigen, können Sie eines der Standard-Handbücher für diese Sprachen verwenden.

Es folgt eine Liste einiger häufig verwendeter Handbücher für Java und C:

- Arnold, Ken, Gosling, James, *The Java Programming Language* (Reading, Mass.: Addison Wesley, 1996)
- Flanagan, David, *Java in a Nusthell* (Cambridge, Mass.: O'Reilly, ^{2}1997).
- Kernighan, Brian W., Ritchie, Dennis M., *The C Programming Language* (Englewood Cliffs: Prentice-Hall, ^{2}1988).

5.1 ESDL als Modelliersprache

ESDL wurde speziell als Modelliersprache für die Kraftfahrzeug-Umgebung entwickelt. In ASCET wird ESDL verwendet, um die Hauptteile von Methoden oder Prozessen innerhalb von Klassen oder Modulen zu spezifizieren. Der Einfachheit halber werden Klassen und Module im vorliegenden Abschnitt unter der Bezeichnung Klassen zusammengefasst.

In ESDL basieren sowohl die Syntax als auch die Elemente auf der Programmiersprache Java, wodurch eine leichte Erlernbarkeit sichergestellt ist. Beim Arbeiten mit ESDL ist jedoch zu bedenken, dass sich ESDL grundlegend von anderen Sprachen unterscheidet.

Die Hauptmerkmale, durch die sich ESDL in gewissem Maße von anderen Sprachen unterscheidet, sind folgende:

- ESDL ist eine *Modelliersprache* und keine Programmiersprache. Es ist eine Modelliersprache, die auf derselben abstrakten, physikalischen Beschreibungsebene operiert wie die in ASCET gewöhnlich verwende-

ten Blockdiagramme. Konzepte, die mit der Implementierung zusammenhängen oder von ihr abhängig sind, wie etwa Zeiger oder Schiebeoperatoren, stehen nicht zur Verfügung.

- ESDL wird für Systeme verwendet, die in einer *Echtzeit-Umgebung* funktionieren. Daher muss die Sprache den Anforderungen eines Echtzeitbetriebs genügen. Infolgedessen ist ESDL so objektorientiert, wie es diese Parameter gestatten. Die Modellstruktur kann auf Klassen und Module abgebildet werden, doch die Instanzbildung ist statisch, und es gibt keine Vererbung.
- ESDL wird verwendet, um *Kraftfahrzeug-Software* herzustellen. Während Benutzer in ESDL komplexe Softwaremodelle aufbauen können, sind Konzepte, die für eingebettete Systeme gegenwärtig nicht relevant sind, wie etwa Zeichenketten-Operationen, nicht implementiert.
- ESDL lässt sich nahtlos in die Entwicklungsumgebung von ASCET einbinden. Die Sprache wird auf derselben Ebene verwendet wie Blockdiagramme, das heißt zur Beschreibung der Funktionen, die in *Hauptteilen von Methoden oder Prozessen* enthalten sind. Der Import von Elementen und die Deklaration von Variablen werden unter Verwendung der entsprechenden Werkzeuge im ESDL-Editor realisiert.

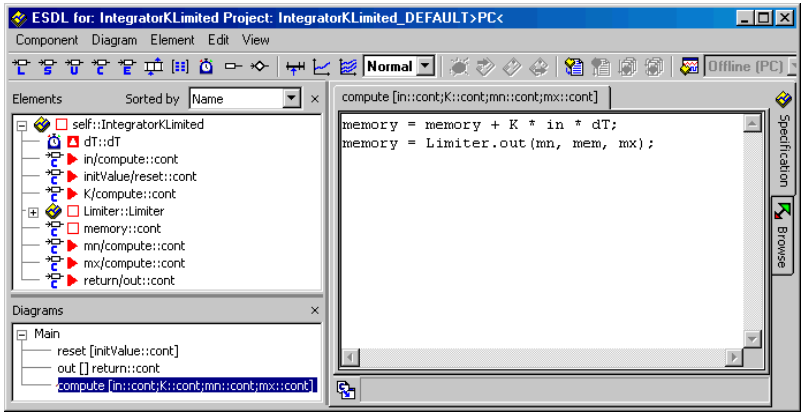
Diese vier Hauptmerkmale von ESDL bestimmen den Anwendungsbereich und die Art und Weise der Verwendung der Sprache. Ansonsten kann ESDL – mehr oder weniger – als eine hochspezialisierte Variante der Programmiersprache Java betrachtet werden.

5.2 Basiselemente

5.2.1 Arbeiten mit Methoden und Prozessen

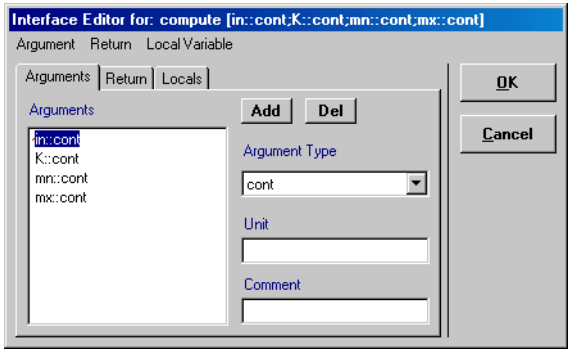
Die Basiselemente einer funktionalen Beschreibung in ESDL sind Methoden und Prozesse. Eine Methode besteht aus einem Methoden-Header, der als Kennung dient, und dem Methodenhauptteil, der die auszuführenden Operationen beschreibt.

Der Methoden-Header besteht aus dem Methodennamen, einer Liste von Argumenten und einem Rückgabewert. Methodennamen werden zugewiesen, wenn zur Methodenliste des ESDL-Editor (Feld „Diagrams“) ein neues Element hinzugefügt wird. Sie können durch Umbenennen des Elements in der Liste geändert werden.



Methodennamen müssen in ESDL eindeutig sein. Ein *Überladen* von Methoden wird nicht unterstützt, d.h. es ist nicht möglich, dass sich zwei Methoden nur in der Anzahl der Parameter und/oder in den Parametertypen voneinander unterscheiden.

Die Argumente und der Rückgabewert sind optionale Elemente der Schnittstelle der Methode. Der Methoden-Header und die Schnittstelle können mit Hilfe des Schnittstelleneditors im ESDL-Editorfenster geändert werden. Der Schnittstelleneditor wird verwendet, um Parameter sowie den Rückgabewert nach Bedarf hinzuzufügen oder zu ändern.



Die funktionale Beschreibung eines Modells ist im Methoden-Hauptteil enthalten, der im Textfeld des ESDL-Editors editiert werden kann.

5.2.2 ESDL-Syntax

Die Syntax von ESDL ist völlig dieselbe wie die der Programmiersprache Java. Jede Anweisung in ESDL wird durch ein Semikolon (;) abgeschlossen.

```
Timer.calculate();
x = a + b;
tmp = Timer.out();
```

Zusammengesetzte Anweisungen oder Blöcke werden in geschweifte Klammern `{ … }` gesetzt.

```
if (x > 0) {
   y = f(x);
   z =1; }
```

Parameter von Methoden und Ausdrücke werden in runde Klammern `( … )` gesetzt.

```
while (z > 4) {
   z--;}

Integrator.reset(15);
Limiter.out(0, 15, 100);
```

Das Gleichheitszeichen (=) wird für Zuweisungen verwendet.

```
low = -1;
xVar = a * (b-5);
tmp = xVar.max(15);
```

5.2.3 Variablennamen

In ESDL werden Variablennamen aus Buchstaben und Ziffern gebildet. Das erste Element eines Variablennamens muss ein Buchstabe sein. Das Unterstreichungszeichen gilt als Buchstabe. Variablennamen dürfen keine Leerzeichen enthalten.

Beispiele für gültige Variablennamen in ESDL:

```
i, j2a, aVar, a_Var
```

Die Namen aller Variablen müssen innerhalb des Geltungsbereiches des betreffenden Elements eindeutig sein. Diese Einschränkung ist wichtig, wenn mit importierten Klassen oder Modulen gearbeitet wird. ESDL ist im gegenwärtigen Stadium nicht in der Lage, Namenskonflikte zu lösen.

Reservierte Schlüsselwörter:

Die folgenden Schlüsselwörter sind reserviert und dürfen nicht als Variablennamen verwendet werden:

```
auto, break, case, char, cond, const, continue,
default, df, do, double, dt, else, enum, exit, extern,
false, float, for, get, getat, getatat, goto, header,
if, inactive, int, interpolate, long, monitorprocess,
normal, null, receive, register, return, search,
self, send, set, setat, setatat, short, signed,
sizeof, static, struct, switch, true, typedef, undef,
union, unsigned, void, volatile, while.
```

Da nicht zwischen Groß- und Kleinschreibung unterschieden wird, ist *jede* Schreibweise der obigen Namen reserviert.

5.2.4 Datentypen

ESDL ist streng typisiert, und Variable müssen deklariert werden. Die Vorgehensweise ist dabei dieselbe wie beim Editieren von Blockdiagrammen. Variable werden zur Elementeliste hinzugefügt und können dann je nach Erfordernis editiert werden.

In ESDL stehen vier Datentypen zur Verfügung, nämlich `udisc, sdisc, cont` und `log`. Sie können zu einer Klasse oder einem Modul hinzugefügt werden, indem das entsprechende Element aus der Symbolleiste des Editors ausgewählt wird.

Der Hauptteil einer Methode oder eines Prozesses in ESDL enthält selbst keine Deklarationen von Variablen. Nur wenn eine Variable lokal bezüglich der vorliegenden Methode/des Prozesses ist, kann sie im Hauptteil der Methode deklariert und initialisiert werden, wobei eine Anweisung der folgenden Art verwendet wird:

```
cont set = 12.34;
cont temp = 0.78e4;
udisc i = 3, j, k;
sdisc aVar = -12;
log trigger = true;
```

5.2.5 Typenumwandlung

Jedesmal, wenn ein Grundrechenoperator wie `+, -, *, /` Operanden von unterschiedlichen Typen hat, wird das Ergebnis automatisch in das mit dem stärksten Typ, der im Ausdruck verwendet wird, umgewandelt.

Die Reihenfolge der Typen ist (nach zunehmender Stärke geordnet): `sdisc`, `udisc`, `cont`.

```
cont result = varUdisc + varCont;
```

Wenn einer Variablen ein Wert zugewiesen wird, müssen die Datentypen zusammenpassen. Es gibt keine explizite Typenbildung. Nur für die arithmetischen Basistypen signed discrete, unsigned discrete und continuous führt ESDL eine implizite Umwandlung durch.

```
cont tmp = 2;
```

Eine Umwandlung von Booleschen und arithmetischen Typen ist *nicht* möglich.

5.2.6 Einfache Methoden

Jeder arithmetische Typ hat eine vordefinierte Schnittstelle, die eine Reihe mathematischer Grundfunktionen abdeckt. Die folgenden Messages stehen für alle arithmetischen Typen zur Verfügung:

Methode	Empfänger	Rückgaben	Benutzung
`val.abs()`	arithmet.	arithmet.	absoluter Wert von `val`
`val1.max(val2)`	arithmet.	arithmet.	der größere der beiden Werte
`val1.min(val2)`	arithmet.	arithmet.	der kleinere der beiden Werte
`var.between(val1, val2)`	arithmet.	log.	`var` zwischen `val1` und `val2`

Tab. 5-1 Einfache Methoden für arithmetische Typen

Die Methode `var.between(val1, val2)` entspricht dem Element `between:And:` in Blockdiagrammen.

5.2.7 Literale und Konstanten

Literale sind Werte wie `12`, `6.1e4` oder `true`. Jeder Grundtyp (Boolesch und arithmetisch) kann in einer ESDL-Methode als Literal auftreten. Der Datentyp von Literalen ist implizit.

Konstanten sind Werte, die einen Namen haben, wie etwa `g = 9.81`. Sie werden auf dieselbe Weise wie Variable zu einer Klasse hinzugefügt und deklariert. Der Elementeditor kann verwendet werden, um einen Wert zuzuweisen und eine Variable als Konstante zu kennzeichnen.

Einige Anwendungsbeispiele:

- `x = g.abs();`

  Der Variablen `x` wird der Absolutbetrag der Konstanten `g` zugewiesen.

- `out1 = myvar.max(g);` oder `out1 = g.max(myvar);`

  Der Variablen `out1` wird der größere der beiden Werte `myvar` (eine Variable) und `g` zugewiesen.

- `out2 = myvar.min(.04);` oder `out2 = (.04).min(myvar);`

  Der Variablen `out2` wird der kleinere der beiden Werte `myvar` und `0.04` zugewiesen.

5.2.8 Kommentare

Ein Kommentar erläutert den Zweck eines konkreten Stücks ESDL-Code. Es gibt zwei Typen von Kommentaren, die im allgemeinen als einzeilige bzw. mehrzeilige Kommentare bezeichnet werden.

Vor einzeiligen Kommentaren steht ein doppelter Schrägstrich (`//`). Der nachfolgende Text bis zum Ende der aktuellen Zeile wird ignoriert. Mehrzeilige Kommentare werden durch `/*` und `*/` begrenzt.

Die in einer ESDL-Beschreibung verwendeten Kommentare werden nicht in den C-Code übertragen, der aus dieser Beschreibung generiert wird.

5.2.9 Operatoren

In ESDL haben Methodenaufrufe Vorrang vor allen anderen Operatoren. Die Vorrang-Reihenfolge kann beeinflusst werden, indem in einem Ausdruck runde Klammern verwendet werden.

Unäre Operatoren:

Die unären Operatoren sind `+`, `-` und `!` (nicht); wobei das letztgenannte für Boolesche Typen verwendet wird. Außerdem stehen der Inkrement-Operator `++` und der Dekrement-Operator `--` zur Verfügung. Sie können als Präfix- oder Postfix-Operatoren verwendet werden.

Unäre Operatoren haben Vorrang vor allen anderen Operatoren. Sie werden von rechts nach links verknüpft.

Arithmetische Operatoren:

In ESDL können die vier arithmetischen Operatoren `+`, `-` , `*` und `/` verwendet werden. Der Modulo-Operator `%`, der den Rest bei einer ganzzahligen Division berechnet, steht gleichfalls zur Verfügung.

Die Operatoren `*`, `/` und `%` haben Vorrang vor den binären Operatoren `+` und `-`. Arithmetische Operatoren werden von links nach rechts verknüpft.

Vergleichs- und Gleichheitsoperatoren:

Die Vergleichsoperatoren sind `>`, `>=`, `<` und `<=`. Sie werden auf arithmetische Typen angewandt und haben in dieser Gruppe Vorrang.

Die Gleichheitsoperatoren == und !=, die sowohl auf Werte- als auch auf Referenztypen angewandt werden können, kommen in der Vorrang-Reihenfolge an nächster Stelle.

Vergleichs- und Gleichheitsoperatoren sind binär. Sie werden von links nach rechts verknüpft.

Logische Operatoren:

Die logischen Operatoren && und || (UND und ODER) folgen in der Vorrang-Reihenfolge an nächster Stelle, wobei der Operator UND Vorrang vor einem ODER hat.

Logische Ausdrücke werden nur so weit ausgewertet, bis bestimmt worden ist, ob der gesamte Ausdruck wahr oder falsch ist. Wenn zum Beispiel der Ausdruck `a && b` ausgewertet wird, und die Auswertung von `a` ergibt `false`, so ist es überflüssig, den restlichen Teil des Ausdrucks auszuwerten. Die Auswertung von hat keinen Einfluss auf das Ergebnis.

Logische Operatoren sind binär. Sie werden von links nach rechts verknüpft.

Bedingter Operator (MUX):

Der bedingte Operator `?:` entspricht dem Operator MUX im Blockdiagrammeditor. Der Operator hat die allgemeine Form `(a ? n : m)`, wobei `a` Boolesch ist und `n` und `m` von demselben Typ sein müssen. Sie können von einem beliebigen Grundtyp sein, Boolesch oder arithmetisch.

Der Wert eines bedingten Ausdrucks hängt vom Wert von `a` ab. Wenn `a` im obigen Beispiel `true` (wahr) ist, so ist der Wert des Ausdrucks `n`, andernfalls ist er `m`.

Der bedingte Operator ist ternär. Hinsichtlich des Vorrangs rangiert er hinter allen binären Operatoren. Die Verknüpfung erfolgt von rechts nach links.

Kurzzuweisungsoperatoren:

In ESDL können gebräuchliche Kurzzuweisungen verwendet werden, wie etwa += oder *=. Die Operation `a += 4` ist eine Kurzform für die Zuweisungsoperation `a = a + 4`. Eine Kurzschreibweise steht für die folgenden Operatoren zur Verfügung:

```
*=, /=, %=, +=, -=.
```

Kurzzuweisungs-Operatoren kommen in der Vorrang-Reihenfolge an letzter Stelle. Die assoziative Verknüpfung erfolgt von rechts nach links.

Zusammenfassung: Vorrang und Assoziativität von Operatoren:

In der folgenden Tabelle werden der Vorrang und die Assoziativität von Operatoren in ESDL, die in diesem Abschnitt beschrieben wurden, zusammengefasst.

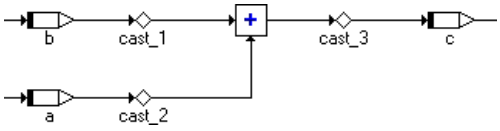
Operator	Assoziativität
`++ --`	von rechts nach links
`+ - (unary)`	von rechts nach links
`!`	von rechts nach links
`* / %`	von rechts nach links
`+ - (binary)`	von rechts nach links
`< <=`	von rechts nach links
`> >=`	von rechts nach links
`==`	von rechts nach links
`!=`	von rechts nach links
`&&`	von rechts nach links
`||`	von rechts nach links
`?:`	von rechts nach links
`=`	von rechts nach links
`*= /= %= += -=`	von rechts nach links

Tab. 5-2 Vorrang und Assoziativität von Operatoren

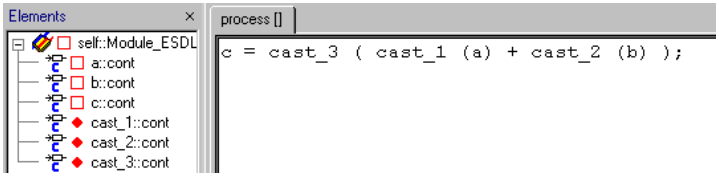
5.3 Implementation-Casts in ESDL

Implementation-Casts (siehe Kapitel 4.2.4) sind in ESDL für Module und Klassen (ausgenommen CT-Blöcke) verfügbar.

In der Spezifikation einer Operation in ESDL müssen die Implementation-Casts durch ihren Namen repräsentiert werden. Eine Addition mit Implementation-Casts, die im BDE so aussieht,



wird als Funktion in ESDL auf folgende Weise repräsentiert:



Wichtig an dieser Stelle ist, dass ein Implementation-Cast wie ein Methodenaufruf geschrieben wird: Er steht immer *vor* dem Element, auf das er sich bezieht; das Element steht, wie ein Methodenargument, in Klammern. Soll der Implementation-Cast auf das Ergebnis einer Operation angewendet werden, so muss die *gesamte* Operation durch Klammern eingeschlossen sein.

Im obigen Beispiel bezieht sich `cast_1` also auf die Variable `a`, `cast_2` auf `b` und `cast_3` auf das Ergebnis der Operation `a + b`.

Sollen Teilergebnisse von arithmetischen Operationen über einen Implementation-Cast manipuliert werden, so müssen die entsprechenden Teilergebnisse geklammert sein.

In der Anweisung

```
x = cast_1 ( (cast_2 ( (a + b) * c - d ) ) / e );
```

bezieht sich `cast_2` also auf das Zwischenergebnis der Operation,

```
(a + b) * c - d
```

während `cast_1` das Gesamtergebnis der Operation

```
((a + b) * c - d) / e
```

verändert.

Hinweis

Ein Implementation-Cast in ESDL bezieht sich immer auf den Wert, der im Code direkt auf den Implementation-Cast folgt.

Wichtig ist an dieser Stelle ebenfalls, dass die Verwendung der obigen Syntax auf Implementation-Casts beschränkt ist. In den Klammern muss ein existierender Implementation-Cast stehen; wenn Sie einen Standardtyp eingeben, etwa `uint8 (a)`, wird eine Fehlermeldung angezeigt.

Achten Sie bei der Verwendung von Implementation-Casts darauf, dass diese für logische Variablen nicht zur Verfügung stehen. Wird dennoch ein Implementation-Cast auf eine logische Variable angewandt, erzeugt die Codegenerierung eine Fehlermeldung.

5.4 Kontrollfluss

Die Kontrollflusselemente können verwendet werden, um die Reihenfolge und die Bedingungen festzulegen, unter denen eine ESDL-Funktion oder ESDL-Anweisung ausgeführt wird. Die gebräuchlichsten Typen sind bedingte Strukturen und Schleifen.

Es gibt zwei Typen von bedingten Anweisungen, `if…else` und `switch…case…default`, und zwei Typen von Schleifen-Anweisungen, `while` und `for`.

Außerdem steht eine Unterbrechungsanweisung `break` zur Verfügung.

Die Kontrollflusskonstruktionen in ESDL werden in den folgenden Unterabschnitten ausführlicher beschrieben.

5.4.1 If…Else

Die Anweisung `if…else` kann für einfache bedingte Konstruktionen verwendet werden. Sie hat die allgemeine Form

```
if (expressionLog){
   statementTrue;}
else {
   statementFalse;}
```

Der `else`-Block kann weggelassen werden. Wenn `expressionLog` ausgewertet wird, entscheidet das Programm, ob der Block `statementTrue` ausgeführt wird. Wenn nicht, führt das Programm entweder einen vorhandenen `statementFalse`-Block aus, oder es wird fortgesetzt, ohne irgendetwas zu tun.

Der Ausdruck `expressionLog`, der die Entscheidung steuert, muss explizit vom Typ `log` sein. Eine Arithmetik mit einem Wert von eins oder null wird *nicht* akzeptiert.

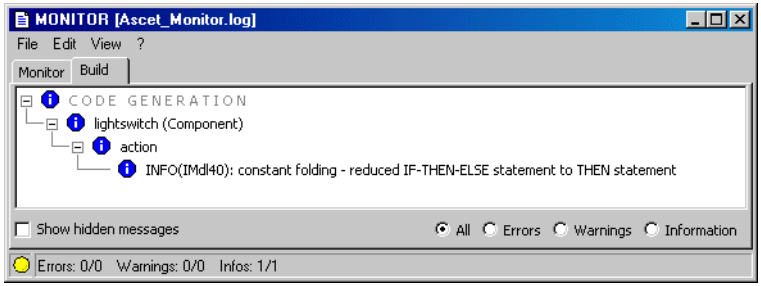
Wenn direkt an der If-Anweisung entschieden werden kann, dass der Ausdruck immer `true` ist, wird die Konstruktion im generierten Code optimiert. Ein Beispiel: Aus

```
if (true || testlog_a) {
   cont=1; }
else {
   cont=0; }
```

wird einfach:

```
cont=1;
```

Wenn eine solche Optimierung stattfindet, wird im ASCET-Monitorfenster eine Information ausgegeben.



Im generierten C-Code hingegen wird *kein* Hinweis gegeben.

Hinweis

Die Entscheidung über die Optimierung wird lokal an der `If`*-Anweisung getroffen. Wenn vorangegangene Programmteile berücksichtigt werden müssten, um die Entscheidung zu treffen, findet* ***keine*** *Optimierung statt.*

5.4.2 Switch…Case…Default

Die Anweisung `switch…case…default`, oder kurz die `switch`-Anweisung, kann für komplexere bedingte Konstruktionen verwendet werden. Sie hat die allgemeine Form

```
switch (expressionsDisc) {
   case sDiscM: {
      statementM }
   …
   case sDiscN: {
      statementN }
   default: {
      statementDefault }
}
```

Die `switch`-Anweisung ist eine Mehrweg-Entscheidung, die prüft, ob das Argument `expressionsDisc` mit einem der konstanten Werte von `sDiscM` bis `sDiscN` übereinstimmt, und den entsprechenden Zweig ausführt.

Jeder Zweig ist mit einem konstanten Ausdruck gekennzeichnet, der vom Typ `sdisc` sein muss. Der entsprechende Block wird ausgeführt, wenn der Ausdruck `expressionsDisc` mit dem Wert des konstanten Ausdrucks übereinstimmt. Der (optionale) Fall `default` wird ausgeführt, wenn kein andere Übereinstimmung gefunden werden kann.

Wenn der Fall `default` nicht zur Verfügung steht und keine Übereinstimmung gefunden wird, bewirkt die `switch`-Anweisung nichts, und die Steuerung kehrt zum restlichen Teil des Software-Modells zurück.

Das nachfolgende Beispiel setzt den Wert einer Variablen `scont` in Abhängigkeit vom Wert des Arguments `argSdisc`.

```
switch(sdiscArg) {
   case 1 : {
      scont = 1.123;
      break; }
   case -1: {
      scont = 0;
      break; }
   default: {
      scont = -1; }
}
```

In diesem Beispiel wird jeder Block mit einer `break`-Anweisung beendet. Dies bewirkt, dass die switch-Anweisung beendet wird, unmittelbar nachdem der Block ausgeführt worden ist.

Bei nicht expliziter Beendigung der `case`-Blöcke würde die Ausführung fortgesetzt, unmittelbar nachdem eine Übereinstimmung gefunden wurde. Im obigen Beispiel bedeutet dies, dass für `sdisc=-1` der Wert von `scont` zuerst durch den entsprechenden Block auf 0 gesetzt würde und anschließend durch den `default`-Block auf –1 gesetzt würde, wenn die `break`-Anweisung fehlen würde.

Diese Erscheinung wird gewöhnlich als *Hindurchfallen* bezeichnet. Der restliche Teil einer `switch`-Anweisung wird immer ausgeführt, wenn ein Block nicht beendet ist. Obwohl dies für eine mehrschichtige Filterung von Nutzen sein kann, wird es im allgemeinen als schlechter Stil betrachtet und sollte vermieden werden, indem jede `case`-Anweisung mit einem `break` beendet wird.

5.4.3 While

Die `while`-Schleife wird verwendet, um eine einfache Schleife zu modellieren. Sie hat die allgemeine Form:

```
while (expressionLog) {
```

```
   loopStatement; }
```

Die Schleifenbedingung `expressionLog` wird ausgewertet. Wenn sie `true` (wahr) ist, wird der Block `loopStatement` ausgeführt, und `expressionLog` wird erneut ausgewertet. Die Schleife wird verlassen, wenn die Auswertung von `expressionLog` das Ergebnis `false` liefert.

In ESDL muss die Schleifenbedingung `expressionLog` vom Typ `logical` sein.

5.4.4 For

Die `for`-Schleife ist eines der hervorstechenden Modellierungsmerkmale, die nur in ESDL zur Verfügung stehen. In Blockdiagrammen gibt es kein Äquivalent dazu.

Die for-Schleife hat die allgemeine Form

```
for ( initExpression; expressionLog; incrExpression )
{
   loopStatement; }
```

Diese ist äquivalent zu

```
initExpression;
while (expressionLog) {
   loopStatement;
   incrExpression; }
```

In der `for`-Schleife ist jeder Bestandteil des Kopfteils der Schleife, `initExpression`, `expressionLog` und `incrExpression`, optional. Die Schleifenbedingung `expressionLog` muss vom Typ `logical` sein. Sie wird auf `true` gesetzt, wenn sie weggelassen wird, was zu einer Endlosschleife führt.

Hinweis

In ESDL müssen die Bestandteile des Kopfteils der Schleife einfache Ausdrücke sein; durch Komma getrennte Listen von Ausdrücken, wie etwa `i=0, j=1`*; oder* `i++, j--;`*, sind nicht zulässig. Anders ausgedrückt, es ist nicht möglich, im Ausdruck* `initExpression` *oder im Ausdruck* `incrExpression` *mehr als eine einzige Anweisung zu verwenden*

Das folgende Beispiel ist eine einfache Kombination einer `if`…`else`-Anweisung und einer `for`-Schleife:

```
if (log) {
   for (index=0; index < array.length(); index++) {
      array[index] = index * index; }
   }
else {
```

```
for (index=0; index < array.length(); index++) {
   array[index] = index; }
}
```

Im Beispiel werden Werte in ein `array` geschrieben. Die `log`-Bedingung in der `if…else`-Anweisung legt fest, welche der beiden Schleifen verwendet wird, um Werte in das `array` zu schreiben.

Jede der Schleifen iteriert über das gesamte `array` und weist jedem Element einen Wert zu. Der Wert ist entweder das Ergebnis von `index * index` oder der Wert von `index`.

5.4.5 Break

Die `break`-Anweisung (Unterbrechungsanweisung) kann verwendet werden, um von jedem beliebigen der oben aufgelisteten Steuerungselemente aus unmittelbar die Schleife zu verlassen und zu einer anderen, die Schleife einschließenden Anweisung oder zum restlichen Teil des Modells zurückzukehren.

Da ESDL keine Sprungmarken in der Modellbeschreibung unterstützt, gibt es keine mit Sprungmarken versehene `break`-Anweisung, welche die Rückkehr der Steuerung zu einer Sprungmarke bewirkt.

5.5 Methoden

Die funktionale Beschreibung eines Softwaremodells in ESDL ist in Methoden enthalten. Die Methoden führen Berechnungen aus und verarbeiten Daten. Sie werden als Operationen an Objekten aufgerufen.

Ein Methodenaufruf hat die allgemeine Form

```
receiverClassName.doSomething(parameterList)
```

wobei `receiverClassName` der Name des Empfänger-Objekts ist, das die Methode `doSomething` 'ausführt'. Parameter können entweder als eine durch Kommas getrennte Liste oder als ein einzelner Parameter in der `parameterList` übergeben werden. Jeder Ausdruck kann ein Parameter sein, einschließlich von Methodenaufrufen.

Beispiele gültiger Methodenaufrufe in ESDL:

```
loader.resolve(false, 1.76);
//do not use characteristic, calculate value for 1.76


numbers.setAt(10*index, index);
//set array numbers to 10*index at index
```

```
(12.4).between(valA, valB);
//check if 12.4 is between valA and valB

array.length();
//return array length
```

Hinweis

Falls eine Methode keine Parameter hat, müssen die runden Klammern am Ende des Methodennamens trotzdem geschrieben werden, damit die Anweisung als Methodenaufruf interpretiert wird.

Ein Methodenaufruf kann einen Wert zurückgeben, der wiederum einer Variablen in dem Methodenaufruf zugewiesen werden kann. Die Variable muss von demselben Typ sein wie der Rückgabewert.

```
aNumber = anArray.getAt(index);
//assign value from index position

anOffset = loader.resolve(true, 2.14);
//assign value for 2.14, calulate using characteris-
tic
```

Falls eine Methode einen Rückgabewert hat, muss der Hauptteil der Methode mit einer `return`-Anweisung beendet werden. Der `return`-Anweisung kann ein beliebiger Ausdruck folgen, dessen Auswertung den Rückgabetyp der Methode ergibt.

```
return in.between(ub, lb);
// returns a logical value

return intVar;
//returns the value of intVar
```

Ein Methodenaufruf kann nur einen einzigen Wert zurückgeben. Wenn mehr als ein Wert zwischen Modulen oder Objekten übergeben werden muss, kann ein Objekt verwendet werden, um diese Werte festzuhalten (siehe Abschnitt „Strukturen“ auf Seite 142).

Methodenaufrufe können in ESDL nicht verschachtelt werden. Die folgende Anweisung ist unzulässig:

```
loader.resolve(true, 2.14).sqrt();
```

Sie muss durch die folgende zulässige Anweisung ersetzt werden:

```
aNumber = loader.resolve(true, 2.14);

aNumber.sqrt();
```

Nur wenn direkte Zugriffsmethoden für den Zugriff auf die Variable eines Objekts freigegeben sind, kann ein Methodenaufruf verschachtelt werden. Daher ist die folgende verschachtelte Anweisung zulässig, wenn `aa` eine Variable ist, die in `anObject` definiert ist:

```
anObject.aa().sqrt()
```

5.5.1 This

Die Pseudo-Kennung `this` kann in ESDL verwendet werden, um eine Methode an der aktuellen Komponente aufzurufen. Wenn Sie zum Beispiel die private Methode `initCounter` am aktuellen Objekt aufrufen möchten, können Sie die folgende Anweisung verwenden:

```
this.initCounter();
```

Wenn die Methode `initCounter` einen Rückgabewert hat, können Sie ihn wie folgt zuweisen:

```
aValue = this.initCounter();
```

Die Referenz auf das aktuelle Objekt mit Hilfe der Kennung `this` ist in diesen beiden Fällen optional, da sie im Kontext impliziert ist. Daher können die obigen Anweisungen wie folgt geschrieben werden:

```
initCounter;
aValue = initCounter();
```

Nur wenn das aktuelle Objekt als Parameter an eine andere Methode weitergegeben werden soll, wird die Referenz unter Verwendung von `this` benötigt.

```
OtherObject.evaluate(this);
```

Hier übergibt die Kennung `this` eine Referenz auf das aktuelle Objekt.

Hinweis

Obwohl ESDL sowohl die Kennung `self` *als auch die Kennung* `this` *akzeptiert, wird die Verwendung von* `this` *empfohlen, um die Kompatibilität mit der Syntax von Java sicherzustellen.*

5.5.2 Zugriffskontrolle

In ESDL können sowohl die Methoden als auch die Variablen einer Klasse entweder als öffentlich oder als privat deklariert werden, um den Zugriff zu diesen Elementen zu kontrollieren und ihre Implementierung vor anderen Objekten zu verbergen.

Private Methoden können nur von innerhalb des aktuellen Objekts aus aufgerufen, private Variable nur aus dem aktuellen Objekts heraus gehandhabt werden. Dagegen sind der Aufruf von öffentlichen Methoden und der Zugriff auf öffentliche Variable sowohl von innerhalb als auch von außerhalb des aktuellen Objekts möglich.

Methoden werden als öffentlich oder privat deklariert, indem sie einem entsprechenden Diagramm im ESDL-Editor zugewiesen werden. Neue Objekte haben standardmäßig ein einziges öffentliches Diagramm `Main`, das die Methode `calc` enthält.

Benutzer können zusätzliche öffentliche Methoden in demselben Diagramm erzeugen oder ein neues Diagramm hinzufügen. Private Methoden müssen als Teil eines privaten Diagramms erzeugt werden. Die Zugriffsrechte auf eine Methode können geändert werden, indem sie von einem Diagramm zu einem anderen bewegt wird.

Ein Objekt `Caller` kann auf die öffentliche Schnittstelle eines anderen Objekts `Receiver` zugreifen, wenn das letztere durch Hinzufügen zur Elementeliste für `Caller` importiert worden ist.

Neue Variable werden als privat erzeugt, wenn sie zur Elementeliste im ESDL-Editor hinzugefügt werden. Auf sie kann nicht von außerhalb des aktuellen Objekts aus zugegriffen werden.

Der Status einer Variablen kann nur im Elementeditor für das betreffende Objekt geändert werden (siehe ASCET Benutzerhandbuch, Kapitel „Bearbeiten von Eigenschaften von Elementen“).

5.5.3 Direkte Zugriffsmethoden

Jede öffentliche Variable fügt zur Schnittstelle des aktuellen Objekts automatisch zwei Methoden hinzu, die als direkte Zugriffsmethoden bezeichnet werden. Eine direkte Zugriffsmethode kann aufgerufen werden, um auf die Daten in einer öffentlichen Variablen zuzugreifen. Sie kann sowohl für den Lesezugriff als auch für den Schreibzugriff auf die betreffende Variable verwendet werden.

Im folgenden Beispiel wird angenommen, dass das Objekt `VisibleObject` zwei öffentliche Variable hat, die den Namen `free` bzw. `all` haben. Methodenaufrufe von außerhalb können folgende Form haben:

```
sdisc tmp = VisibleObject.all();
VisibleObject.free(120);
```

Direkte Zugriffsmethoden werden jedesmal automatisch generiert und der öffentlichen Schnittstelle eines Objekts hinzugefügt, wenn eine Variable als öffentlich deklariert wird. Diese Methoden brauchen nicht explizit codiert zu werden.

5.6 Zusammengesetzte Datentypen

ESDL stellt zwei Gruppen von zusammengesetzten Datentypen bereit. Die erste Gruppe von zusammengesetzten Typen umfasst gewöhnliche Arrays und Matrizen, und die zweite Gruppe, die eindimensionale Tabellen, zweidimensionale Tabellen und Verteilungen enthält, wird für Kennlinien und -felder verwendet.

In den folgenden Unterabschnitten werden die zusammengesetzten Datentypen erläutert, und zwar zuerst Arrays und Matrizen und anschließend Verteilungen.

5.6.1 Arrays

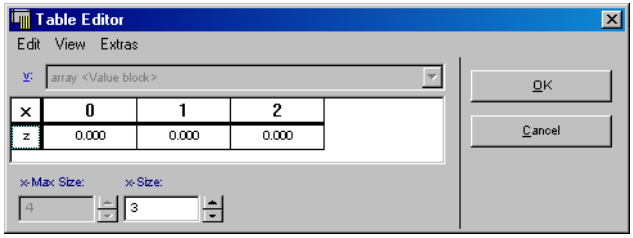
Ein Array ist eine eindimensionale, indizierte Menge von Variablen, die von demselben Datentyp sind. In ESDL stehen Arrays für alle Basisdatentypen zur Verfügung. Der Zugriff auf die Variablen erfolgt über den Array-Index, die erste Indexposition ist 0.

Ein Array kann zu einem Modul hinzugefügt werden, indem es zur Liste „Elements“ im ESDL-Editor hinzugefügt wird. Der Array-Typ kann im Elementeditor als ein beliebiger Grundtyp spezifiziert werden.

Die Größe eines Array und seine Daten können mittels des Tabelleneditors editiert werden, das automatisch geöffnet wird, wenn die Daten eines Array editiert werden sollen. Sie können sowohl die aktuelle als auch die maximale Größe des Array im Tabelleneditor vorgeben.

Die Größe eines Arrays kann während der Laufzeit nicht geändert werden. Die maximale Größe für Arrays beträgt 1024 Elemente.

Die Daten eines Array können entweder im Tabelleneditor editiert werden, oder sie können aus einer durch Tabs getrennten ASCII-Datei der Reihe nach übernommen werden (siehe ASCET Benutzerhandbuch, Kapitel „Bearbeiten von Daten“, Abschnitt „Array-Editor“).



In ESDL ist es unter Verwendung der folgenden Syntax möglich, Elemente eines Array zu lesen und in sie zu schreiben:

```
val = myArray[index];
myArray[index] = val;
```

Die erste Anweisung bewirkt das Lesen des Wertes des Array-Elements an der Position `index` und weist diesen der Variablen `val` zu, die von demselben Typ wie das Array sein muss. Da die Zählung des Array-Index bei 0 beginnt, gibt `myArray[3]` das vierte Element eines Array zurück.

Die zweite Anweisung setzt den Wert des Array-Elements in der Position `index` auf `val`, wobei dieses von demselben Datentyp wie das Array sein muss.

Öffentliche Schnittstelle:

Tab. 5-3 fasst die öffentlichen Methoden zusammen, die bei Arrays zur Verfügung stehen.

Methode	Rückgabe	Verwendung
`length()`	`udisc`	Anzahl der Array-Elemente bestimmen
`getAt(index)`	Array-Type	Array-Element an der Position `index` bestimmen
`setAt(val, index)`	`void`	Array-Element an Position `index` auf `val` setzen

Tab. 5-3 Die öffentliche Schnittstelle von Arrays

5.6.2 Matrizen

Eine Matrix ist eine zweidimensionale, indizierte Menge von Variablen, die von demselben Datentyp sind. In ESDL stehen Matrizen für alle Basisdatentypen zur Verfügung. Der Zugriff auf die Variablen erfolgt über die Array-Indizes x und y, die erste Indexposition ist 0.

Eine Matrix kann im ESDL-Editor auf genau dieselbe Weise hinzugefügt und gehandhabt werden wie ein Array. Die Größe einer Matrix kann während der Laufzeit nicht geändert werden. Die maximale Größe von Matrizen beträgt 64 Elemente pro Dimension.

In ESDL ist es unter Verwendung der folgenden Syntax möglich, Elemente einer Matrix zu lesen und in sie zu schreiben:

```
val = matrix[indX][indY];
matrix[indX][indY] = val;
```

Die erste Anweisung weist den Wert des Elements von `matrix` an der Position Spalte `indX` und Zeile `indY` der Variablen `val` zu, die von demselben Typ wie die Matrix sein muss. Da die Zählung des Index bei 0 beginnt, gibt `myMatrix[2,3]` das dritte Element in der vierten Zeile einer Matrix zurück.

Die zweite Anweisung setzt den Wert des Elements von `matrix` an der Position Spalte `indX` und Zeile `indY` auf `val`, wobei dieses von demselben Datentyp wie die Matrix sein muss.

Öffentliche Schnittstelle:

Tab. 5-4 fasst die öffentlichen Methoden zusammen, die bei Matrizen zur Verfügung stehen.

Methode	Rückgabe	Verwendung
`xLength()`	`udisc`	Spaltenanzahl in Matrix bestimmen
`yLength()`	`udisc`	Zeilenanzahl in Matrix bestimmen
`getAt(indX, indY)`	Matrizen-Typ	Matrizen-Element in Position `indX`, `indY` bestimmen
`setAt(val, indX, indY)`	`void`	Matrizen-Element in Position `indX`, `indY` auf `val` setzen

Tab. 5-4 Die öffentliche Schnittstelle von Matrizen

5.6.3 Eindimensionale Tabellen

Ein eindimensionale Tabelle wird verwendet, um Kennlinien zu modellieren, die Parameterwerte nicht unter Verwendung eines Algorithmus, sondern in Abhängigkeit von einer gegebenen Menge von Abtastpunkten beschreiben.

Für jeden Abtastpunkt x_n in der Tabelle existiert ein Parameterwert y_n, welcher aus der eindimensionalen Tabelle abgerufen werden kann. Außerdem kann die Tabelle den gesamten Wertebereich zwischen Abtastpunkten abdecken, indem entweder lineare oder gerundete Interpolation angewandt wird.

Ein eindimensionale Tabelle kann zu einem Modul hinzugefügt werden, indem sie zur Elementliste im ESDL-Editor hinzugefügt wird. Der Datentyp kann im Elementeditor als ein beliebiger arithmetischer Typ spezifiziert werden.

Die maximale Größe für eindimensionale Tabellen beträgt 1024 Paare Abtastpunkt : Wert. Im Gegensatz zu Arrays und Matrizen werden Tabellen in ASCET als Parameter verwendet, das heißt, in die Abtastpunkte und Werte kann nicht von innerhalb des Modells aus geschrieben werden.

Die Tabellendaten können entweder im Tabelleneditor editiert werden, oder sie können aus einer durch Tabs getrennten ASCII-Datei der Reihe nach übernommen werden (siehe ASCET Benutzerhandbuch, Kapitel „Bearbeiten von Daten“, Abschnitt „1D-Tabelleneditor“).

Der Interpolationsmodus für Abtastpunkte kann gleichfalls im Tabelleneditor als gerundet oder linear vorgegeben werden. Bei gerundeter Interpolation wird der Wert für einen Punkt vom unteren (linken) Abtastpunkt übernommen, während bei linearer Interpolation der Wert von einer Geraden zwischen den Abtastwerten abgeleitet wird.

Öffentliche Schnittstelle:

In ESDL kann auf Tabellen nur unter Verwendung ihrer öffentlichen Schnittstelle zugegriffen werden. Tab. 5-5 fasst die öffentlichen Methoden zusammen, die bei eindimensionalen Tabellen zur Verfügung stehen.

Methode	Rückgabe	Verwendung
`search(index)`	`void`	den Abtastpunkt der Tabelle auf `index` setzen oder Interpolationsfaktor für `index` berechnen
`interpolate()`	Tabellen-Typ	den Wert für den aktuellen Abtastpunkt ermitteln oder ihn aus der Tabelle interpolieren
`getAt(index)`	Tabellen-Typ	den Abtastpunkt auf `index` setzen und den entsprechenden Wert ermitteln oder Interpolationsfaktor für `index` berechnen und den Wert interpolieren

Tab. 5-5 Die öffentliche Schnittstelle von eindimensionalen Tabellen

Lineare Interpolation:

Das folgende Beispiel veranschaulicht die lineare Interpolation in eindimensionalen Tabellen. Darin wird eine Tabelle `LLpr` verwendet, die die folgenden Werte hat:

`0.0`	`1000.0`	`2000.0`	`3000.0`	`4000.0`	`5000.0`	`6000.0`
`0.0`	`0.8`	`1.1`	`1.5`	`1.8`	`2.0`	`2.2`

Die Methode `getAt(index)` reicht im Allgemeinen völlig für die Auswertung einer Kennlinie. Die lineare Interpolation wird für dieses Beispiel wie folgt durchgeführt:

```
tmpVal = LLpr.getAt(3000);
// assigns 1.5 to tmpVal

tmpVal = LLpr.getAt(2280);
// calculates interpolation factor for 2280
// interpolates value for 2280 as 1.212 and
// assigns it to tmpVal
```

```
tmpVal = LLpr.getAt(9000);
// calculates extrapolation factor for 9000
// extrapolates value for 9000 as 2.2 and
// assigns it to tmpVal
```

Das Trennen der Such- und Interpolationsschritte in Tabellen kann jedoch in Einzelfällen effizienter sein, wenn z.B. Code für experimentelle Targets generiert wird. In diesem Fall wird die lineare Interpolation folgendermaßen ausgeführt:

```
LLpr.search(1000);
// sets sample point to 1000

tmpVal = LLpr.interpolate();
// assigns 0.8 to tmpVal

LLpr.search(2780);
// calculates interpolation factor for 2780

tmpVal = LLpr.interpolate();
// interpolates value for 2780 as 1.412 and
// assigns it to tmpVal
```

5.6.4 Zweidimensionale Tabellen

Eine zweidimensionale Tabelle wird verwendet, um Kennfelder zu modellieren, die Parameterwerte nicht unter Verwendung eines Algorithmus, sondern in Abhängigkeit von einer gegebenen Menge von Paaren von Abtastpunkten beschreiben.

Für jedes Paar von Abtastpunkten (x_n : y_n) in der Tabelle existiert ein Parameterwert z_n, der aus der zweidimensionalen Tabelle abgerufen werden kann. Außerdem kann die Tabelle den gesamten Wertebereich zwischen Abtastpunkten abdecken, indem entweder lineare oder gerundete Interpolation angewandt wird.

Eine zweidimensionale Tabelle kann im ESDL-Editor auf dieselbe Weise hinzugefügt und gehandhabt werden wie eine eindimensionale Tabelle. Die maximale Größe für zweidimensionale Tabellen beträgt 64 Paare von Abtastpunkten und entsprechenden Werten.

Öffentliche Schnittstelle:

In ESDL kann auf Tabellen nur unter Verwendung ihrer öffentlichen Schnittstelle zugegriffen werden. Tab. 5-6 fasst die öffentlichen Methoden zusammen, die bei zweidimensionalen Tabellen zur Verfügung stehen.

Methode	Rückgabe	Verwendung
`search(indX, indY)`	`void`	die Abtastpunkte der Tabelle auf `indX` und `indY` setzen oder Interpolationsfaktor für `indX` und `indY` berechnen
`interpolate()`	Tabellen-Typ	den Wert für den aktuellen Abtastpunkt ermitteln oder ihn aus der Tabelle interpolieren
`getAt(indX, indY)`	Tabellen-Typ	den Abtastpunkt auf `indX` und `indY` setzen und den entsprechenden Wert ermitteln oder Interpolationsfaktor für `indX` und `indY` berechnen und den Wert aus der Tabelle interpolieren

Tab. 5-6 Die öffentliche Schnittstelle von zweidimensionalen Tabellen

Lineare Interpolation:

Das folgende Beispiel veranschaulicht die lineare Interpolation in zweidimensionalen Tabellen. Darin wird eine Tabelle `LLpr2` verwendet, die die folgenden Werte hat:

```
y \ x     0.0       1.0       8.0       15.0
1.0       -5.0      -3.0      0.0       1.0
3.0       0.0       1.0       4.0       6.0
5.0       8.0       5.0       4.0       4.0
```

Wie bei den Kennlinien beinhaltet die Methode `getAt(indX,indY)` alles, was für die Auswertung eines Kennfeldes nötig ist. Die lineare Interpolation wird für dieses Beispiel wie folgt durchgeführt:

```
tmpVal = LLpr2.getAt(8,5);
// assigns 4.0 to tmpVal

tmpVal = LLpr2.getAt(0.5,1.5);
// calculates interpolation factor for
// x=0.5 and y=1.5
// interpolates value for (0.5,1.5) as -2.875 and
// assigns it to tmpVal
```

```
tmpVal = LLpr2.getAt(20,10);
// calculates extrapolation factor for x=20, y=10
// extrapolates value for (20,10) as 5.0 and
// assigns it to tmpVal
```

Auch bei Kennfeldern kann es Fälle geben, wo die Trennung der Such- und Interpolationsschritte effizienter ist. In diesem Fall wird die lineare Interpolation folgendermaßen ausgeführt:

```
LLpr2.search(1,3);
// sets x sample point to 1 and y sample point to 3

tmpVal = LLpr2.interpolate();
// assigns 1.0 to tmpVal

LLpr2.search(4,4);
// calculates interpolation factor for x=4, y=4

tmpVal = LLpr2.interpolate();
// interpolates value for (4,4) as 3.143 and
// assigns it to tmpVal
```

5.6.5 Verteilungen und Gruppentabellen

Kennlinien und -felder können unter Verwendung derselben Menge von Abtastpunkten miteinander in Verbindung gebracht werden. In ASCET wird eine solche gemeinsam verwendete Menge von Abtastpunkten als eine Verteilung modelliert; Tabellen, die die Abtastpunkte in einer Verteilung verwenden, werden als Gruppentabellen bezeichnet.

Eine Verteilung ist ein Array von Abtastpunkten. Die Folge muss streng monoton wachsend sein. Verteilungen können für beide Tabellentypen (eindimensional und zweidimensional) verwendet werden. Bei zweidimensionalen Tabellen ist für jede Dimension eine Verteilung erforderlich.

Die Verwendung von Verteilungen und Gruppentabellen kann den Zeit- und Speicherplatzbedarf für Berechnungen erheblich verringern, da Interpolationsfaktoren nur einmal berechnet werden und innerhalb einer Menge von Tabellen wiederverwendet werden können.

Das Hinzufügen einer Gruppentabelle im ESDL-Editor besteht darin, zunächst eine Verteilung und anschließend eine Gruppentabelle hinzuzufügen. Wenn die Gruppentabelle hinzugefügt ist, fordert das System die entsprechende Verteilung an. Da der ESDL-Editor nicht verwendet werden kann, um Verteilungen wieder existierenden Gruppentabellen zuzuweisen, sollten Verteilungen stets erzeugt werden, bevor die Tabellen hinzugefügt werden.

Der Tabelleneditor kann verwendet werden, um die Daten sowohl von Verteilungen als auch von Tabellen zu editieren. Er akzeptiert keine Verteilungen, bei denen die strenge Monotonie verletzt ist. Daten können auch aus durch Tabs getrennten ASCII-Datei der Reihe nach übernommen werden.

Öffentliche Schnittstelle:

Im Gegensatz zu einfachen Tabellen haben Gruppentabellen keine Methode `getAt`. Statt dessen ist die öffentliche Schnittstelle zwischen der Verteilung, die eine Methode `search` (suchen) hat, und der Gruppentabelle, die eine Methode `interpolate` (Interpolieren) hat, „aufgespalten".

Bei einer zweidimensionalen Tabelle ist es erforderlich, dass die Abtastpunkte für beide Verteilungen gesetzt werden, bevor der entsprechende Wert interpoliert werden kann.

Tab. 5-7 zeigt die öffentliche Schnittstelle von Verteilungen in ESDL.

Methode	Rückgabe	Verwendung
`search(index)`	`void`	die Abtastpunkte der Verteilung auf `index` setzen oder den Interpolationsfaktor für `index` berechnen

Tab. 5-7 Die öffentliche Schnittstelle von Verteilungen

Tab. 5-8 zeigt die öffentliche Schnittstelle von Gruppentabellen in ESDL.

Methode	Rückgabe	Verwendung
`interpolate()`	Tabellen-Typ	den Wert für den aktuellen Abtastpunkt ermitteln oder ihn aus der Tabelle interpolieren

Tab. 5-8 Die öffentliche Schnittstelle von Gruppentabellen

5.7 Strukturen

In ESDL werden Strukturen (oder Datensätze) unter Verwendung von Klassen modelliert. Eine Klasse kann als ein komplexes Behälterelement verwendet werden, das eine beliebige Anzahl von Variablen enthält. Wenn eine Variable in einer Klasse öffentlich ist, kann von ESDL mit Hilfe direkter Zugriffsmethoden diese Variable gelesen und in sie geschrieben werden.

Auf Klassen, die als Behälterelemente verwendet werden, wird auf dieselbe Weise zugegriffen wie auf andere Klassen in ESDL. Der erste Schritt besteht immer darin, die Klasse zur Elementeliste des ESDL-Editors hinzuzufügen, um sie im Kontext der aktuellen Klasse verfügbar zu machen. Variable können im Layout-Editor für das übergeordnete Objekt als öffentlich deklariert werden.

Auf die Variablen kann dann aus ESDL heraus unter Verwendung der einfachen Syntax der direkten Zugriffsmethoden zugegriffen werden:

```
theVar = VisibleObject.aVar()
VisibleObject.aVar(5.12);
// read/write access to primitive variable


theVar = VisibleObject.anArray().getAt(2)
VisibleObject.anArray().setAt(2.14, 3);
// read/write access to array variables


VisibleObject.aDistribX().search(6);
VisibleObject.aDistribY().search(2);
theVar =
     VisibleObject.a2dGroupTable().interpolate();
// reading from a two-dimensional table
```

In ESDL können Klassen verschachtelt werden, um selbstreferierende Strukturen zu modellieren.

Hinweis

Eine komplexe Zuweisung, wie etwa `VisibleObject.anArray(myArray)`*, ist in ESDL nicht zulässig; sie weist nicht die Werte in dem Parameter* `myArray` *dem Element* `anArray` *zu. Komplexe Anweisungen können jedoch verwendet werden, um eine Referenz an ein anderes Objekt zu übergeben.*

5.8 Messages

In ASCET wird eine zusätzliches Konzept von Messages als Echtzeit-Sprachkonstrukten für die Interprozesskommunikation verwendet. Messages werden in diesem Sinne als geschützte globale Variable in der Echtzeit-Umgebung verwendet.

Messages stehen nur in Modulen zur Verfügung. Aus einem Modul heraus ist eine Message lediglich eine Variable, die gelesen oder in die geschrieben werden kann, oder beides. Jedesmal, wenn ein Prozess abläuft, erzeugt das Betriebssystem Kopien von dessen sämtlichen Messages. Diese Kopien sind nur für diejenige Instanz des Prozesses zugänglich, die sie erzeugt hat.

Daher erhält, wenn dieselbe Message von verschiedenen Prozessen verwendet wird, jeder Prozess seine eigene Kopie der Message. Diese Strategie wird von dem Echtzeit-Betriebssystem angewandt, um die Datenkonsistenz über mehrere Prozesse hinweg sicherzustellen.

Messages werden in ESDL voll unterstützt; sie können in allen Modulen verwendet werden. Eine Message wird wie alle anderen Elemente der Elementeliste hinzugefügt, indem die entsprechende Symbolgrafik aus der Werkzeugleiste des ESDL-Editors ausgewählt wird. Messages können hinzugefügt werden als

- Send-Messages—das aktuelle Modul kann in diese Variable schreiben,
- Receive-Messages—das aktuelle Modul kann diese Variable lesen, oder
- Send&Receive-Messages—das aktuelle Modul kann diese Variable lesen und in sie schreiben.

In ESDL wird auf Messages über Zuweisungsanweisungen zugegriffen:

```
theVar = receiveMsg + 1.24;
sendMsg = 12;
theMessage = 3 * tmpVar;
```

Öffentliche Schnittstelle:

Tab. 5-9 fasst die öffentlichen Methoden zusammen, die für Messages zur Verfügung stehen.

Methode	Rückgabe	Verwendung
`receive()`	void	Message lesen
`send()`	void	Message schreiben

Tab. 5-9 Die öffentliche Schnittstelle von Messages

5.9 Resourcen

Ressourcen sind wie Messages nur in Modulen verfügbar. Wie in Kapitel 3.1.3 beschrieben, besitzen sie zwei Zugriffsmethoden, `reserve` und `release`. Das folgende Beispiel zeigt, wie diese Methoden in ESDL verwendet werden können:

```
resource1.reserve();
do_something();
resource1.release();
```

Tab. 5-10 fasst die öffentlichen Methoden zusammen, die für Ressourcen verfügbar sind.

Method	Returns	Usage
`reserve()`	void	reserve a resource
`release()`	void	release a resource

Tab. 5-10 Die öffentliche Schnittstelle von Ressourcen

5.10 Mathematische Funktionen

ASCET enthält eine umfangreiche Bibliothek von vordefinierten Elementen. Sie können als Bausteine für neue Module und Klassen verwendet werden.

Für Modellbeschreibungen in ESDL werden in der Systembibliothek zusätzliche mathematische Funktionen bereitgestellt. Die mathematischen Funktionen sind in der Klasse `Etas_Systemlib_CT\Classes\MathFcn` definiert, und auf sie kann zugegriffen werden, nachdem diese Klasse zur Elementeliste des ESDL-Editors hinzugefügt worden ist.

Die folgenden Beispiele zeigen, wie von einer ESDL-Modellbeschreibung aus auf mathematische Funktionen zugegriffen wird.

```
// calculate sine of x
x = x + MathFcn.pi()/2;
y = MathFcn.sin(x);

// calculate square root of arg
if (arg > 0) return MathFcn.sqrt(arg);

// typecast continuous arg to logical
return (MathFcn.Sign(arg) = 0 ? false : true);

// fill array at x-1 with x^1/x
udisc x
cont tmp, y;
for (x = 1; x < array.length() + 1; x++) {
   tmp = x;
   array[x-1] = MathFcn.pow(tmp, 1/tmp); }
```

In Tab. 5-11 sind die Funktionen zusammengefasst, die in der Klasse `MathFcn` zur Verfügung stehen. Die Rückgabe- und Parametertypen sind für alle mathematischen Funktionen dieselben, sie akzeptieren Variable vom Typ `continuous` als Parameter, und der Rückgabetyp ist gleichfalls `continuous`.

Methode	Operation
`pi()`	Gibt 3.141592654 zurück
`sin(x)`	sin von x
`cos(x)`	cos von x
`tan(x)`	tan von x
`asin(x)`	$\sin^{-1}(x)$ (Arkussinus)
`acos(x)`	$\cos^{-1}(x)$ (Arkuskosinus)
`atan(x)`	$\tan^{-1}(x)$ (Arkustangens)
`sinh(x)`	Hyperbelsinus von x
`cosh(x)`	Hyperbelkosinus von x
`tanh(x)`	Hyperbeltangens von x
`sch(x)`	Hyperbelsekante von x
`csch(x)`	Hyperbelkosekans von x
`coth(x)`	Hyperbelkontangens von x
`exp(x)`	Exponentialfunktion e^x
`log(x)`	natürlicher Logarithmus mit der Basis e: $\log_e(x)$, `x > 0`
`log10(x)`	Logarithmus mit der Basis 10: $\log_{10}(x)$, `x > 0`
`pow(x,y)`	x^y
`sqrt(x)`	Quadratwurzel von x
`abs(x)`	Absolutwert `|x|`
`sign(x)`	Vorzeichenfunktion gibt zurück: `-1` wenn `x < 0`; `0` wenn `x = 0`; `1` wenn `x > 0`
`limit(m,x n)`	Begrenzer gibt zurück: `m` wenn `x <= m`; `x` wenn `m < x < n`; `n` wenn `x => n`
`max(x,y)`	Gibt den jeweils größeren Wert von `x` und `y` zurück
`min(x,y)`	Gibt den jeweils kleineren Wert von `x` und `y` zurück
`fmod(x,y)`	Gleitkommarest von `x/y`, gleiches Vorzeichen wie `x`
`ceil(x)`	Gibt den kleinsten ganzzahligen Wert nicht kleiner als `x` zurück.
`floor(x)`	Gibt den größten ganzzahligen Wert nicht größer als `x` zurück.

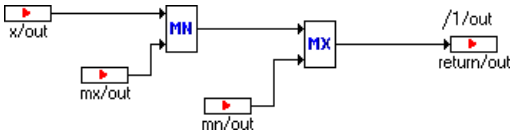
Tab. 5-11 Mathematische Funktionen in ESDL

5.11 Zugriff auf Blockdiagramme aus ESDL

Im vorliegenden Abschnitt wird der Aufbau eines einfachen begrenzten Integrators in ESDL beschrieben. Der Integrator verwendet ein Begrenzungselement aus dem Ordner `SystemLib_ETAS`, um die Bandbreite des Ausgangssignals zu bestimmen.

Das Begrenzungselement verfügt über eine einzige Methode `out` mit drei Parametern `mn`, `x`, `mx`. Die Methode `out` gibt entweder `mn` zurück, falls `x < mn`, oder `x`, falls `mn <= x <= mx`, oder `mx`, falls `x > mx`.

Das Blockdiagramm für das Begrenzungselement ist nachfolgend abgebildet.



Aufbauen des Integrator-Elements

- Erzeugen Sie im Komponentenmanager eine neue ESDL-Klasse und benennen Sie sie um in `IntegratorLimit`.
- Öffnen Sie einen ESDL-Editor für `IntegratorLimit`.
- Fügen Sie in der Elementeliste eine Variable vom Typ `continuous` mit dem Namen `mem` hinzu. Der Speicher des Integrators speichert den Wert des Ausgangssignals.
- Fügen Sie in der Liste „Elements“ die Variable `K` vom Typ `continuous` und den Parameter `dT` hinzu.
- Fügen Sie in der Elementeliste das Begrenzungsmodul aus dem folgenden Ordner hinzu: `SystemLib_ETAS\Nonlinears\Limiter`.
- Fügen Sie in der Methodenliste die Methoden `out`, `reset` und `compute` hinzu.

  Sie können die Standardmethode `calc` entweder in `compute` umbenennen oder löschen.

- Verwenden Sie den Schnittstelleneditor, um die entsprechenden Methoden-Schnittstellen wie folgt zu editieren:

Methode	Argumente	Rückgaben

`compute`	`cont mx`
	`cont in`
	`cont mn`
		`void`

`out`	`void`	`cont`
`reset`	`cont initVal`	`void`

- Geben Sie für jede Methode den ESDL-Code ein und sichern Sie die Methode. Der ESDL-Code für die einzelnen Methoden ist nachfolgend aufgelistet.

```
reset(initVal)
mem = initVal;
```

```
cont out()
return mem;
```

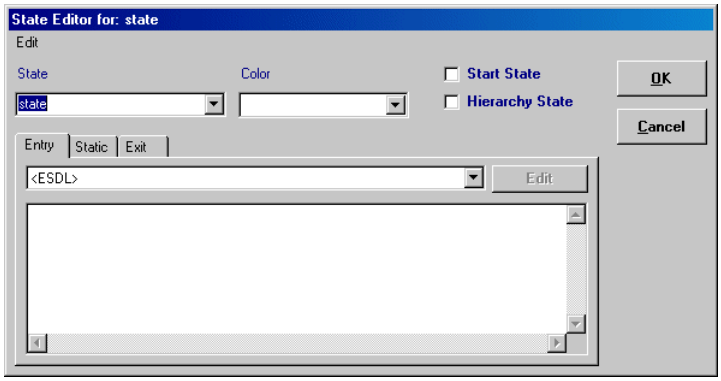
```
compute(mn, in, mx)
mem = mem + K * in * dT;
mem = Limiter.out(mn, mem, mx);
```

Das Beispiel zeigt, wie ein vorhandenes Modul als Baustein für ein neues Modul verwendet werden kann. Die zweite Anweisung in der Methode `compute` begrenzt das Integratorsignal. Die Methode `out` des Begrenzers gibt den Signalwert oder die untere oder obere Schranke zurück, die dem Speicher des Integrators zugewiesen wird.

5.12 Verwendung von ESDL in Zustandsautomaten

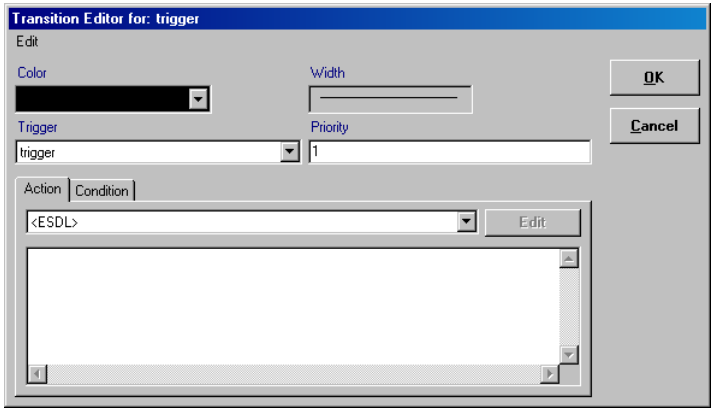
Bei der Modellierung von Zustandsautomaten in ASCET ist die Beschreibung in ESDL oft kompakter als Blockdiagramme. ESDL kann verwendet werden, um sowohl Zustände als auch Übergänge zwischen Zuständen zu beschreiben.

Ein Zustand kann bis zu drei verschiedene Aktionen haben, die als Eintrittsaktion, statische Aktion und Austrittsaktion bezeichnet werden. Sie werden ausgeführt, wenn in den Zustand eingetreten wird, während er aktiv ist, und wenn der Zustand beendet wird.



Die Aktionen in einem Zustand können im Zustandseditor editiert werden. Sie können in ESDL spezifiziert werden, wenn die Option **<ESDL>** für die entsprechende Aktion gewählt wird. Dadurch wird das Textfeld für die Aktion aktiviert, das ein einfacher ESDL-Editor ist. Von diesem Editor aus kann auf die Ausgangs- und Eingangsvariablen des Zustandsautomaten und alle anderen Elemente in der Elementeliste des Zustandsautomaten zugegriffen werden.

Ein Übergang zwischen Zuständen ist gewöhnlich mit einer Bedingung verknüpft, die den Übergang in einen anderen Zustand auslöst; er kann auch mit einer Aktion verknüpft sein, die ausgeführt wird, wenn der Übergang vollzogen wird.



Die Übergänge zwischen Zuständen können im Übergangseditor editiert werden. Auch hier können Bedingungen und Aktionen im Textfeld in ESDL spezifiziert werden, nachdem die Option `<ESDL>` aktiviert worden ist, und es kann auf alle Elemente in der Elementeliste zugegriffen werden.

In allen Feldern beider Editoren wird Standard-ESDL-Code verwendet wie in den obigen Beispielen. Der einzige wesentliche Punkt, der in der Syntax von ESDL zu beachten ist, besteht darin, dass der im Register „Condition" eingegebene Ausdruck einen Booleschen Wert zurückgibt und nicht mit einem Semikolon beendet wird. Mehr über das Bearbeiten von Bedingungen und Aktionen in ESDL finden Sie im ASCET Benutzerhandbuch, Kapitel Kapitel 4.2, Abschnitt „Bedingungen und Aktionen in ESDL".

5.13 Übersicht: Merkmale von ESDL im Vergleich

Gegenüberstellung ESDL – Blockdiagramme

Die folgende Tabelle gibt eine Übersicht über die Unterschiede in den Modellbeschreibungen bei Verwendung von ESDL und von Blockdiagrammen.

	ESDL	**Blockdiagramme**
`this`	x	o
`self`	x	x
`%` operator	x	o
`++`, `--` operator	x	o
`for` Anweisung	x	o
atomare Folgen	o	x

Tab. 5-12 Zusammenfassung: Gegenüberstellung ESDL – Blockdiagramme

Übersicht: Gegenüberstellung ESDL – ANSI C

Die folgende Tabelle gibt eine Übersicht über die Hauptunterschiede zwischen der Modelliersprache ESDL und der Programmiersprache ANSI C.

	ESDL	ANSI C
Bit-Datentyp, Schiebeoperationen	o	x
String-Datentyp, Zeichenketten-Operationen	o	x
Anweisung `continue`	o	x
Zeigerarithmetik	o	x
Präprozessor	o	x

Tab. 5-13 Zusammenfassung: Gegenüberstellung ESDL - ANSI C

Übersicht: Gegenüberstellung ESDL – Java

Die folgende Tabelle gibt eine Übersicht über die Hauptunterschiede zwischen der Modelliersprache ESDL und der Programmiersprache Java.

	ESDL	Java
Vererbung	o	x
dynamische Instanzbildung	o	x
Polymorphie	o	x
Methodenüberladung	o	x
Explizite Typenbildung	o	x
Fehlerbehandlung	o	x
Speicherbereinigung	o	x

Tab. 5-14 Zusammenfassung: Gegenüberstellung ESDL – Java

6 Spezifikation mit Blockdiagrammen

Mit der blockorientierten Beschreibungssprache von ASCET können eingebettete Steuerungssysteme grafisch spezifiziert werden. Dies ist das grafische Gegenstück zur Sprache ESDL, die zum textuellen Spezifizieren von Steuerungssystemen verwendet wird.

Im vorliegenden Abschnitt wird beschrieben, wie Softwaremodi unter Verwendung von Blockdiagrammen in ASCET zu spezifizieren sind. Der folgende Abschnitt beginnt mit einer kurzen Einführung in die grafische Beschreibung von Komponenten, der eine Übersicht über die grafische Modelliersprache folgt.

In dem Übersichtsabschnitt werden die sprachlichen Mittel vorgestellt, die in Blockdiagrammen zur Verfügung stehen:

- Elemente
- Ausdrücke
- Anweisungen

Blockdiagramme und ESDL sind in ASCET größtenteils funktional äquivalent. Die Unterschiede zwischen Blockdiagrammen und ESDL sind im Abschnitt „Gegenüberstellung ESDL – Blockdiagramme“ auf Seite 150 zusammengefasst.

6.1 Grafische Beschreibung von Elementen

Jedes in einer Komponente verwendete Element und jeder in einer Komponente verwendete Operator wird grafisch durch ein Diagrammelement in Form eines Rechtecks dargestellt. Die Wechselwirkung zwischen diesen Elementen wird durch Linien dargestellt, welche die entsprechenden Diagrammelemente verbinden.

Die Schnittstelle eines Elements wird grafisch durch Anschlüsse (Abb. 6-1) dargestellt. Jedes Argument einer Methode wird durch einen Argumentanschluss (mit einer kleinen, auf den Block zeigenden Pfeilspitze) am Blockrahmen dargestellt. Die Rückgabewerte werden durch einen Rückgabeanschluss dargestellt. Der Aufruf einer Methode ist mit deren Rückgabeanschluss verknüpft. Methoden ohne Argumente oder Rückgabewerte werden durch einen Methodenanschluss dargestellt.

↦ Methodenanschluss

→ Argumentanschluss

— Rückgabeanschluss

Abb. 6-1 Die Darstellung von Anschlüssen in grafischen Blöcken

Der Name des Elements wird unter dem Rechteck angegeben. Es kann eine Symbolgrafik verwendet werden, um die Funktionalität eines Elements zu veranschaulichen. Die Position der Anschlüsse kann vom Benutzer verändert werden.

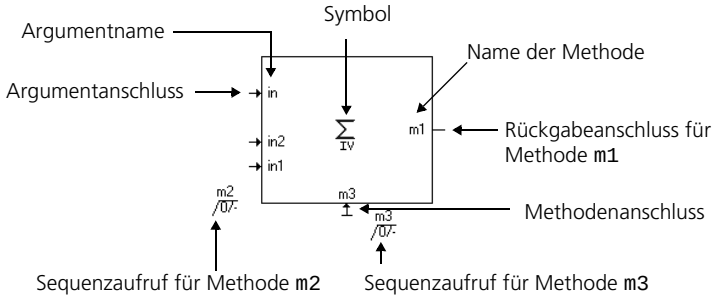


Abb. 6-2 Der grafische Block für ein komplexes Element.

Das Beispiel in Abb. 6-2 zeigt ein komplexes Element mit drei Methoden. Die Methode `m1` hat ein Argument und einen Rückgabewert. Die Methode `m2` hat keinen Rückgabewert und wird durch ihre Argumente dargestellt, und die Methode `m3` hat weder Argumente noch einen Rückgabewert und wird durch einen Methodenanschluss dargestellt.

6.1.1 Basiselemente

Elemente werden als rechteckige Blöcke mit den als Anschlüsse dargestellten Argumenten und Rückgabewerten dargestellt. Jedes Element hat einen Namen, der standardmäßig unter dem Block angegeben wird, wobei diese Position jedoch geändert werden kann.

Alle Basistelemente haben eine feste Schnittstelle, und ihre grafische Darstellung ist ebenfalls fest.

Skalare Basiselemente

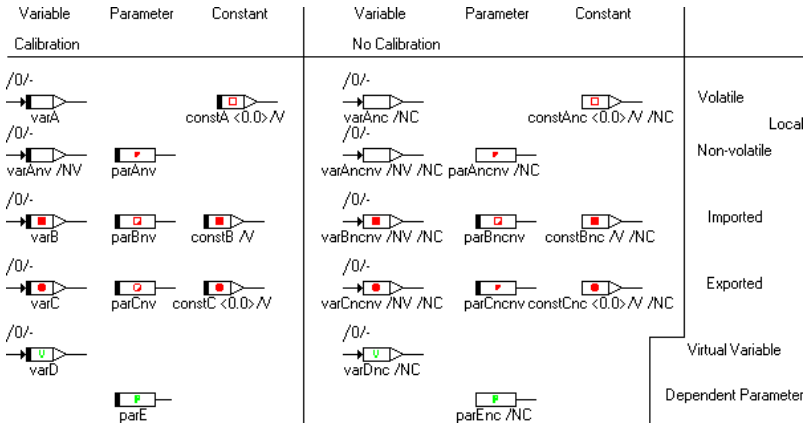
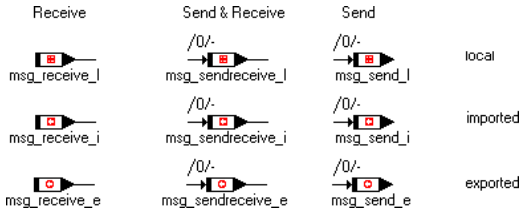


Abb. 6-3 Die grafische Darstellung von Basiselementen

Skalare Basiselemente haben einen Argumentanschluss zum Einstellen eines neuen Wertes (wenn ihr Wert eingestellt werden kann) und einen Rückgabeanschluss zum Lesen des aktuellen Wertes. Die Symbolgrafik innerhalb des Blocks stellt die Art des Elements dar. Der Geltungsbereich eines Elements wird ebenfalls durch die Symbolgrafik angegeben: ein ausgefülltes rotes Quadrat stellt ein importiertes Element dar, ein ausgefüllter Kreis ein exportiertes. Falls die Art des skalaren Basiselements ein Schreiben auf dieses Element nicht gestattet (z. B. Parameter), so fehlt der entsprechende Anschluss. Elemente, die kalibriert werden können, sind durch ein kleines schwarzes Kästchen auf der linken Seite gekennzeichnet.

Messages

Messages sind die Ein- und Ausgangsvariablen von Prozessen. Je nach Message-Typ werden sie mit einem oder zwei Anschlüssen dargestellt. Die Abbildung zeigt Messages mit den Eigenschaften *Calibration* und *volatile*; werden eine oder beide Eigenchaften geändert, ändert sich die Darstellung wie in Abb. 6-3 für Variablen gezeigt.

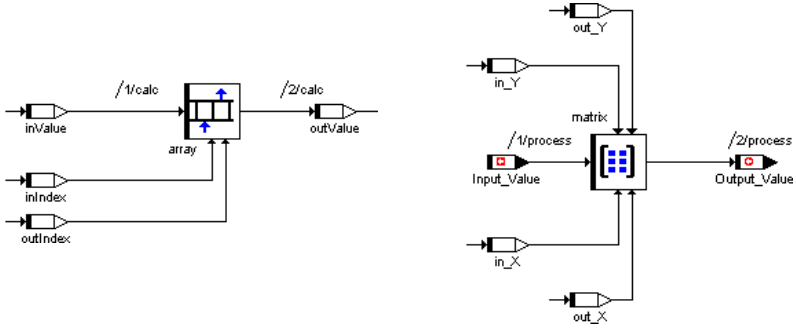


Literale

Literale werden durch kleine Blöcke dargestellt, wobei der Wert des Literals im Inneren des Blocks angegeben ist:

Arrays und Matrizen

Ein Array oder eine Matrix hat zwei Methoden, eine zur Einstellung des Inhalts eines spezifischen Elements und eine zum Abrufen desselben. Die Lese- und Schreiboperationen können unabhängig voneinander stattfinden. Der in das Array zu schreibende Wert wird durch den linken (Argument-)Anschluss dargestellt, der entsprechende Index durch den linken unteren Anschluss. Das Ergebnis des Lesens aus dem Array wird durch den Rückgabeanschluss dargestellt und der Index durch den rechten unteren Argument-Anschluss.



Matrizen werden auf ähnliche Weise dargestellt, wobei jedoch für jede Methode zwei Indexargumente benötigt werden. Der x-Index wird durch den linken unteren Anschluss dargestellt, und zwar wie der Index eines Array. Der y-Index wird durch die Anschlüsse oben am Block dargestellt, wobei der linke obere Anschluss der Index zum Schreiben in die Matrix und der rechte obere Anschluss der Index beim Lesen aus der Matrix ist.

Wenn ein Array oder eine Matrix als Methodenargument übergeben oder als Rückgabewert zurückgegeben werden soll, geschieht dies mit Hilfe der *Get- und Set-Ports*. Diese werden im Zeichenbereich über das Pop-up-Menü **Get/ Set Ports** verfügbar gemacht.

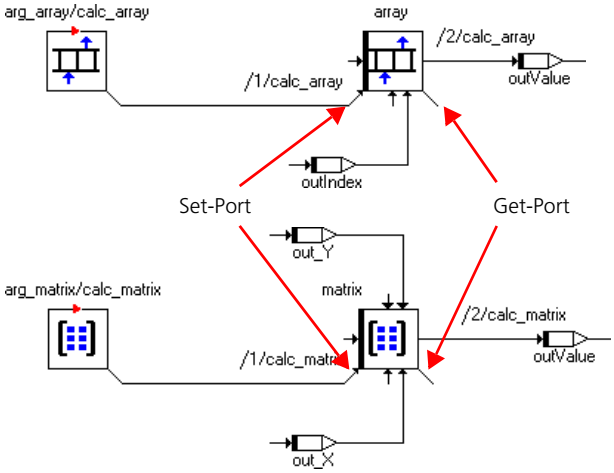


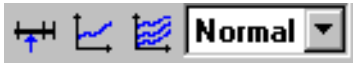
Abb. 6-4 Get- und Set-Ports bei Arrays und Matrizen

Der Get-Port liefert einen Zeiger auf den gesamten Dateninhalt des betreffenden Elements, während ein Element über den Set-Port angewiesen wird, auf einen bestimmten Speicherbereich zuzugreifen.

Hierzu ein Beispiel: In Abb. 6-4 liest der Array `array` aus dem Speicherbereich des Arrays `arg_array`, die Matrix `matrix` aus demjenigen von `arg_matrix`. Die Zeiger auf die jeweiligen Speicherbereiche werden über Get- und Set-Ports übergeben. Wichtig ist dabei, dass das Schreiben in den jeweiligen Set-Port als erster Rechenschritt der Methode ausgeführt wird, andernfalls kommt es zu Inkonsistenzen.

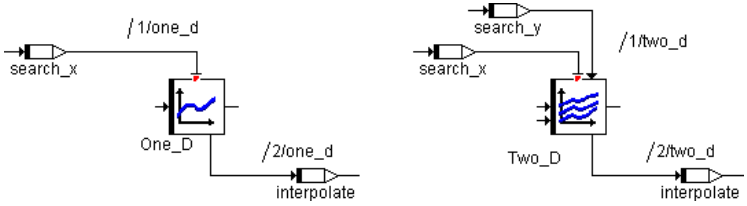
Hinweis

Der gleiche Mechanismus wird auch für die Übergabe von Klassen verwendet.

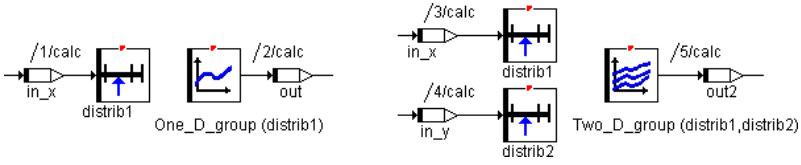


In Abhängigkeit von der Dimension haben Kennlinien einen und Kennfelder zwei Argumentanschlüsse auf der linken Seite, wo die Abtastwerte bereitgestellt werden, und einen Rückgabeanschluss, wo der Wert der Interpolation geliefert wird. Entsprechendes gilt für Festkennlinien und -felder.

Diese Form der Darstellung entspricht der Verwendung der Methode `getAt` in ESDL (s. die entsprechenden Abschnitte auf Seite 137 bzw. Seite 139). Wie bei ESDL können auch im Blockdiagrammeditor die Such- und Interpolationsschritte getrennt werden. Hierzu wird im Zeichenbereich über das Pop-up-Menü **Extended Interface** die erweiterte Schnittstelle der Tabelle verfügbar gemacht.



Eine Verteilung hat einen Argumentanschluss für den Abtastwert auf der linken Seite der Verteilung. Die Gruppentabelle hat einen Rückgabeanschluss auf der rechten Seite. Sie enthält keine eigene Abtastpunktverteilung, sondern referenziert eine oder zwei Verteilungen. Gruppentabellen und Verteilungen haben keine erweiterte Schnittstelle.



Wie bei Arrays und Matrizen können auch bei Kennlinien und -feldern über das Pop-up-Menü **Get/Set Port** die Get- und Set-Ports verfügbar gemacht werden.

Hinweis

Wenn Sie Kennlinien/-felder als Methodenargumente übergeben wollen, müssen Sie diese in eine Klasse einbetten und die Klasse per Get-Port übergeben.

Ressourcen

Ressourcen werden durch einen Block mit den zwei Methoden `reserve` und `release` an der Oberseite dargestellt. Beide Methoden haben weder Argumente noch Rückgabewerte und werden als Methodenanschlüsse dargestellt:

Implementation-Casts

Implementation-Casts (siehe Kapitel 4.2.4) werden durch eine kleine Raute mit zwei Anschlüssen dargestellt:

6.1.2 Elemente von einem benutzerdefinierten Typ

Die Methoden, Argumente und Rückgabewerte von Elementen eines benutzerdefinierten Typs werden durch Argument- oder Rückgabeanschlüsse am grafischen Block dargestellt. Der Benutzer kann für jeden benutzerdefinierten Typ die Gestaltung der Darstellung definieren. Auch für diese Elemente können Get-/Set-Ports verfügbar gemacht werden.

6.2 Ausdrücke

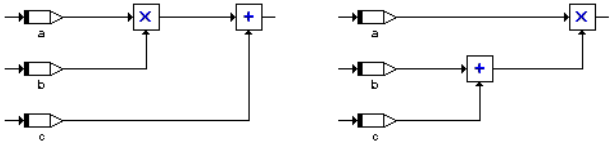
Ausdrücke werden in Blockdiagrammen durch Verbinden von Elementen oder anderen Ausdrücken mit Operatoren gebildet. Wie in ESDL werden Ausdrücke rekursiv aufgebaut, und zwar wie folgt:

- Ein Element ist ein Ausdruck.
- Das Ergebnis eines Operators ist ein Ausdruck (die Operanden selbst sind Ausdrücke).
- Der Rückgabewert eines Methodenaufrufs ist ein Ausdruck. Wenn der Methode Argumente zur Verfügung gestellt werden, so gehören diese Argumente ebenfalls zum Ausdruck.

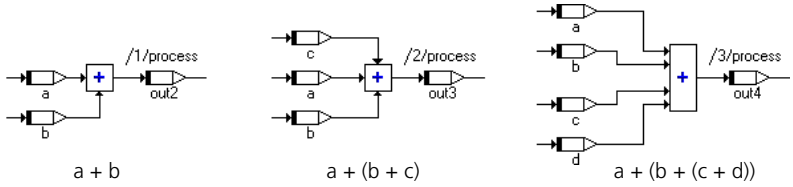
Der Bereich eines Ausdrucks ist daher durch die Basisausdrücke in diesem Ausdruck begrenzt, die entweder Elemente oder Rückgabewerte von Methoden ohne Argumente sind.

Ausdrücke werden grafisch durch Verbinden der Rückgabeanschlüsse von Elementen oder Operatoren mit den Argumentanschlüssen von Methoden oder anderen Operatoren aufgebaut.

Es existieren keine Präzedenzregeln für Operatoren in den Blockdiagramm-Ausdrücken, da die Ausdrücke durch die Art und Weise, wie die Linien und Operatoren verbunden sind, „mit Klammer versehen" sind. Das folgende Beispiel zeigt den Unterschied zwischen den Ausdrücken `(a*b)+c` und `a*(b+c)` in der grafischen Darstellung.



Die Reihenfolge der Auswertung der Argumente von Operatoren ist manchmal sehr wichtig. In der grafischen Darstellung ist dies stets die Reihenfolge von oben nach unten, mit Ausnahme der vier arithmetischen Grundoperatoren mit höchstens drei Eingängen. Die Reihenfolge der Auswertung wird im nachfolgenden Diagramm veranschaulicht:



a + b a + (b + c) a + (b + (c + d))

In Blockdiagrammen ist die Anzahl der Argumente für die Operatoren oft auf maximal 10 oder 20 Eingänge begrenzt. Die Auswertungsreihenfolge von Argumenten von Methoden hängt von der Reihenfolge ab, in der sie definiert sind. Da das Layout eines Elements geändert werden kann, muss die Reihenfolge im Layout nicht mit der in der Definition übereinstimmen.

6.2.1 Arithmetische Operatoren

Die Bedeutung der Operatoren ist dieselbe wie in ESDL. Folgende Operatoren stehen zur Verfügung: Addition, Subtraktion, Multiplikation, Division und Modulo. Die Additions- und Multiplikationsoperatoren können zwischen 2 und 10 Argumenten haben. Die Subtraktions- und Divisionsoperatoren haben nur zwei Argumente.

6.2.2 Vergleichsoperatoren

Die Vergleichsoperatoren sind mit ihren Gegenstücken in der Textdarstellung mit ESDL identisch. Folgende Vergleichsoperatoren stehen zur Verfügung:

- Größer als
- Kleiner als
- Kleiner oder gleich
- Größer oder gleich
- Gleich
- Ungleich

Die Operatoren Gleich und Ungleich können auch auf nichtarithmetische Elemente angewandt werden.

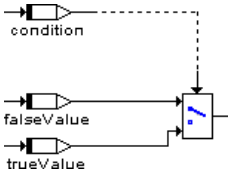
6.2.3 Logische Operatoren

Die Bedeutung der logischen Operatoren Und, Oder und Nicht ist mit deren Bedeutung in ESDL identisch. Die Operatoren Und und Oder können auf mehr als zwei Operanden angewandt werden.

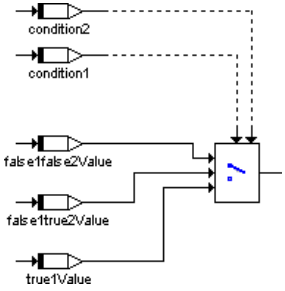
6.2.4 Bedingte Operatoren

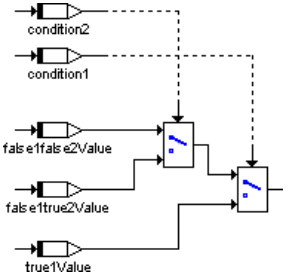
Multiplexoperator (MUX)

Der Bedingte Operator `( ? : )` wird in der grafischen Darstellung Multiplexoperator (kurz: MUX) genannt. Die grafische Darstellung von `(condition ? trueValue : falseValue)` ist folgende:



Der Multiplexoperator kann auch direkt mit mehreren Argumenten verwendet werden (linkes Bild); das rechte Bild zeigt zur Veranschaulichung dieselbe Funktion als Kaskade aus mehreren MUX-Operatoren:

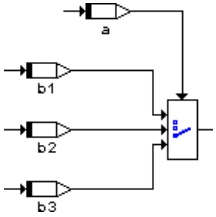




Das obige Beispiel ist äquivalent zu `(condition1 ? (true1value : condition2 ? (false1true2value : false1false2value ))`, d.h. das erste Argument hat Priorität gegenüber den anderen. Ein in Kaskadenschaltung angeordneter MUX-Operator mit n logischen Bedingungs-Argumenten kann zwischen n+1 Argumenten auswählen, zwischen denen er umschaltet. Der Typ der Argumente ist beliebig, jedoch müssen alle Argumente von einem kompatiblen Typ sein.

Falloperator (Case)

Der Falloperator ist ein Spezialfall des bedingten Operators. Er nimmt keinen logischen Wert an, sondern einen Schalterwert vom Typ `unsigned` discrete. Der Falloperator wählt in Abhängigkeit vom Schalterwert eines der Argumente aus. Wenn der Schalterwert 1 ist, wird das erste Argument gewählt, wenn er 2 ist, wird das zweite zurückgegeben, und so weiter. Wenn der Schalterwert außerhalb des Bereiches liegt, wird das letzte Argument gewählt.



Das obige Beispiel ist äquivalent zu `((a=1) ? b1 : ((a=2) ? b2 : b3))`.

6.2.5 Weitere Operatoren

Neben den bisher beschriebenen stehen noch folgende weitere Operatoren zur Verfügung:

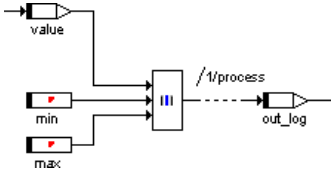
- Max und Min
- Between
- Abs
- Negation

Max- und Min-Operatoren

Die Max- und Min-Operatoren geben das Maximum bzw. Minimum der Argumente zurück. Beide Operatoren können 2 bis 20 Argumente haben. Sie können nur auf arithmetische Elemente angewandt werden.

Between-Operator

Der Between-Operator untersucht, ob das Argument `value` zwischen den Begrenzungen `min` und `max` liegt. Ist dies der Fall, wird der logische Rückgabewert `out_log` auf `true` gesetzt, andernfalls ist er `false`.



Die grafische Darstellung ist äquivalent zu `out_log = (( value >= min ) && ( value <= max ))`. Das Argument und die beiden Begrenzungen müssen alle entweder vom Typ `cont` oder `discrete` sein.

Betragsoperator (Abs)

Dieser Operator gibt den Betrag des Arguments zurück. Argument und Ausgabe müssen beide entweder vom Typ `cont` oder `discrete` sein.

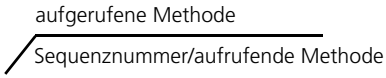
Negations-Operator

Der Negationsoperator gibt den Negativwert des Arguments zurück. Argument und Ausgabe können vom Typ `cont` oder `discrete` sein; wenn das Argument vom Typ `cont` ist, sollte die Ausgabe denselben Typ haben.

6.3 Anweisungen

Grafische Spezifikationen von Komponenten können hierarchisch über verschiedene Diagramme verteilt sein. In einem Diagramm können eine oder mehrere Methoden oder Prozesse beschrieben werden, die unabhängig voneinander ausgeführt werden können. Die Reihenfolge, in der Berechnungen ausgeführt werden, sowie die spezielle Methode oder der Prozess, zu dem eine Berechnung gehört, wird durch Sequenzaufrufe festgelegt.

Für jede Anweisung eines Blockdiagramms existiert ein Sequenzaufruf, der sie einem Prozess oder einer Methode zuweist. Die Reihenfolge innerhalb eines Prozesses oder einer Methode wird durch die Sequenznummer festgelegt, die Bestandteil des Sequenzaufrufs ist. Ein Sequenzaufruf wird wie folgt grafisch dargestellt:



Mit Hilfe der Sequenznummern kann die Reihenfolge der Operationen, die zu einer Methode oder einem Prozess gehören, durch den Benutzer bestimmt werden. Ein eingebauter Sequenzierungsalgorithmus kann verwendet werden, um Sequenznummern zuzuweisen, die der Auswertungsreihenfolge von Standard-Blockdiagrammen entsprechen.

Ein Sequenzaufruf besteht im allgemeinen aus drei Feldern:

- Dem Namen der aufgerufenen Methode
- Dem Namen der aufrufenden Methode oder des aufrufenden Prozesses
- Der Sequenznummer, welche die Position der aufgerufenen Methode in der aufrufenden Methode oder dem aufrufenden Prozess festlegt.

Im Falle von skalaren Elementen wird der Name der aufgerufenen Methode leer gelassen, da dies immer die Zuweisung eines neuen Wertes ist.

Es gibt drei Arten von Anweisungen:

- Zuweisungsanweisungen
- Methodenaufrufe
- Kontrollfluss-Anweisungen, z. B. `if`…`then`…`else`, `while`

6.3.1 Zuweisung

Eine Zuweisungsanweisung ist die Zuweisung des Wertes eines Ausdrucks zu einem Element. Im Falle einer Zuweisung zu einem komplexen Element kann nur ein Element desselben Typs zugewiesen werden. Die Zuweisung ist dann nicht die Zuweisung eines Wertes, sondern die einer Referenz.

Die Zuweisung eines Wertes zum Rückgabewert einer Methode ist ein Sonderfall. Der zugehörige Sequenzaufruf muss der letzte Sequenzaufruf dieser Methode sein.

6.3.2 Die Unterbrechungsanweisung

Die Rückkehr von einer Methode oder einem Prozess kann auch durch die Unterbrechungsanweisung festgelegt werden. Diese Anweisung muss nicht die letzte einer Methode oder eines Prozesses sein.

6.3.3 Methodenaufruf

Eine Zuweisung ist ein Spezialfall eines Methodenaufrufs. Wenn eine Methode in einem Blockdiagramm aufgerufen wird, muss der entsprechende Sequenzaufruf korrekt eingetragen werden, und die Argumente für die Methode sind zur Verfügung zu stellen.

6.3.4 Kontrollfluss

In Blockdiagrammen stehen die folgenden Kontrollflussanweisungen zur Verfügung:

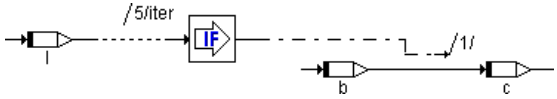
- `If…Then`
- `If…Then…Else`
- `Switch`
- `While`

Alle Kontrollflussanweisungen werten einen logischen Ausdruck aus und aktivieren je nach Ergebnis einen Zweig des Kontrollflusses, der mehrere Anweisungen enthalten kann. Die durch Sequenzaufrufe dargestellten Anweisungen sind durch Konnektoren mit dem Kontrollfluss verbunden.

Die Sequenznummer des Sequenzaufrufs legt die Reihenfolge der mit dem aktivierten Kontrollflusszweig verbundenen Anweisungen fest.

If…Then

Die Anweisung `If…Then` wertet einen logischen Ausdruck aus und aktiviert einen Kontrollflusszweig, wenn das Ergebnis `True` ist. Der Ausgang des Kontrollflusses ist mit einem oder mehreren Sequenzaufrufen verbunden, die jedesmal dann ausgelöst werden, wenn der Kontrollflusszweig aktiviert wird. Jedesmal, wenn die Auswertung des Eingangs-Ausdrucks `True` ergibt, werden die mit ihm verbundenen Sequenzaufrufe ausgeführt.

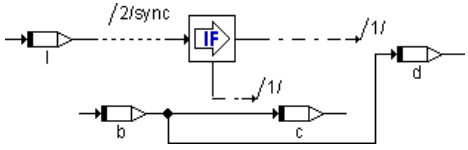


Das obige Beispiel ist äquivalent zu

```
if (l) {
   c = b
};
```

If...Then...Else

`If`…`Then`…`Else` ist der Anweisung `If`…`Then` ähnlich, hat jedoch zwei Kontrollflusszweige. In Abhängigkeit vom Wert des logischen Ausdrucks wird der obere oder der untere Zweig ausgeführt; der obere Zweig wird ausgeführt, wenn der Wert `True` ist, der untere, wenn der Wert `False` ist.



Das obige Beispiel ist äquivalent zu

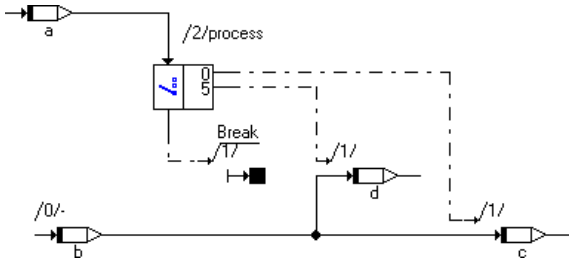
```
if (l) {
   d = b}
else {
   c = b
};
```

Wie bei der `if`…`else`-Anweisung in ESDL wird der generierte Code optimiert, wenn bei `If`…`Then` oder `If`…`Then`…`Else` immer `true` anliegt. Wie die Optimierung genau aussieht, ist im Abschnitt „If…Else“ auf Seite 127 beschrieben.

Switch

Das Konstrukt `Switch` (Schalter) ist dem Falloperator ähnlich. Ein Schalter wertet einen Wert vom Typ `signed discrete` oder `unsigned discrete` aus und aktiviert in Abhängigkeit von diesem Wert verschiedene Kontrollflusszweige. Diese Zweige sind voneinander getrennt, so dass ein „Hindurchfallen“ wie im `Switch`-Konstrukt in C nicht möglich ist.

Für jede Alternative kann der Wert für den Zweig vom Benutzer definiert werden. Der letzte Zweig ganz unten ist der Default-Zweig, der ausgeführt wird, wenn der Eingabewert mit keinem der Werte an den Zweigen übereinstimmt.



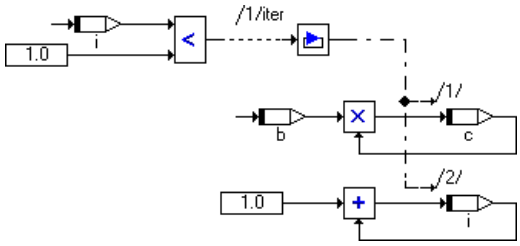
Das obige Beispiel ist äquivalent zu

```
switch (a) {
  case 0: c = b; break;
  case 5: d = b; break;
  default: break;
};
```

While

Das einzige Schleifenkonstrukt, das in Blockdiagrammen zur Verfügung steht, ist die `While`-Schleife. Es ist Vorsicht geboten, um unendliche Schleifen oder für Echtzeitanwendungen ungeeignete Schleifen zu vermeiden.

Ähnlich wie bei der Anweisung `If…Then` wird der Kontrollfluss aktiviert, wenn der Wert des logischen Ausdrucks `True` ist. Die Operation wird so lange ausgeführt, wie der Wert des logischen Eingangs `True` bleibt. Deshalb muss der Wert des logischen Ausdrucks in der `While`-Schleife verarbeitet werden. Um unendliche Schleifen zu vermeiden, kann die Anzahl der maximalen Schleifeniterationen auf eine feste, durch den Benutzer definierbare Anzahl begrenzt werden.



Das obige Beispiel ist äquivalent zu

```
while (i<10) {
   c = b * c;
   i = 1 + i;
};
```

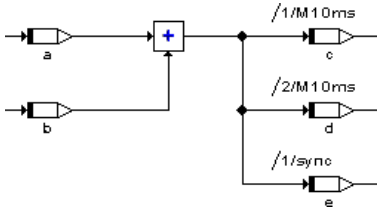
6.4 Die Semantik von Blockdiagrammen

Jeder Teil eines Blockdiagramms ist einem Prozess oder einer Methode zugeordnet. Die Reihenfolge der Ausführung wird durch die Sequenznummern in den Sequenzaufrufen festgelegt. Wenn ein Prozess oder eine Methode aktiviert wird, werden alle Anweisungen, deren Sequenzaufrufe mit diesem Prozess oder dieser Methode gekoppelt sind, in der durch die Sequenznummern vorgegebenen Reihenfolge ausgeführt.

Anders als bei Standardblockdiagrammen wird eine Operation nur auf Anforderung ausgeführt, d.h. wenn ihr Sequenzaufruf aktiviert wird. Die Reihenfolge der Ausführung ist ähnlich dem Von-links-nach-rechts-Prinzip von Standardblockdiagrammen: Bevor eine Operation, zum Beispiel eine Addition, ausgeführt werden kann, müssen die Werte für alle ihre Elemente berechnet werden.

Die Auswertungsreihenfolge der Argumente von Methoden von benutzerdefinierten Komponenten ist durch die Reihenfolge ihrer Deklaration gegeben. Diese Reihenfolge stimmt jedoch möglicherweise nicht mit der durch das Diagramm implizierten Reihenfolge überein, da die Argumentanschlüsse am Blockrahmen beliebig angeordnet werden können.

Die Auswertung von Operanden usw. ist direkt mit den Anweisungen verknüpft, welche die Ergebnisse verwenden. Dies kann mehrfache Auswertungen eines Ausdrucks zur Folge haben.



In diesem Beispiel wird die Addition dreimal ausgeführt, für jede der Zuweisungen zu den Variablen `c`, `d` und `e` einmal. Die Addition wird in Zuweisungen in zwei verschiedenen Prozessen verwendet. Ohne eine mehrfache Ausführung wäre es nicht klar, in welchem der Prozesse die Addition ausgeführt werden soll. Der Ausdruck `a + b` wird in dem Prozess `10ms` zweimal ausgewertet.

6.4.1 Grafische Hierarchien

Um eine grafische Spezifikation zu strukturieren, können grafische Hierarchien verwendet werden. Grafische Hierarchien haben keinen Einfluss auf die Semantik eines Blockdiagramms, sondern werden nur zur Strukturierung verwendet. Eine Hierarchie enthält einen Teil des Blockdiagramms. Die Linien, die den Rand der Hierarchie kreuzen, d.h. Elemente innerhalb der Hierarchie mit Elementen außerhalb verbinden, werden durch Anschlüsse dargestellt. In ASCET 5.2 kann im Blockdiagrammeditor Hierarchien eine Symbolgrafik zugeordnet werden.

7 Spezifikation in C

Die Spezifikation des Hauptteils von Methoden und Prozessen kann, ebenso wie in Form von Blockdiagrammen und in ESDL, auch in C-Code implementiert werden. Wie bei den anderen Spezifikationsmethoden muss nur der Hauptteil einer Methode oder eines Prozesses spezifiziert werden. Die Deklaration der Methode, der Funktionskopfteil und Rahmen sowie die Instanzbildung und Initialisierung der Daten werden automatisch generiert.

Im Gegensatz zu Spezifikationen in ESDL oder mit Blockdiagrammen werden Komponenten in C-Code auf der Implementierungsebene spezifiziert und nicht auf der Modellebene.

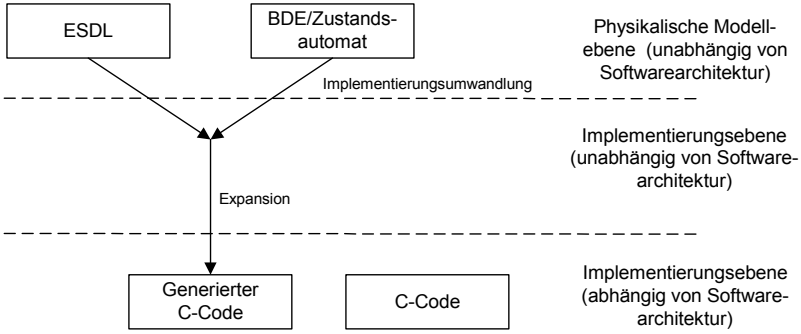


Abb. 7-1 Vom physikalischen Modell zur Implementierung

Dies hat mehrere wesentliche Konsequenzen:

- Es gibt keine Umwandlung von der Modellebene auf die Implementierungsebene.
- Für jede Codevariante (anderes Target, andere Spezifikationsebene, andere Implementierungen) kann der C-Code anders sein. Infolgedessen muss der Code für jede Variante gesondert spezifiziert werden.
- Bei Verwendung benutzerdefinierter Typen muss der C-Code an die Software-Architektur des von ASCET generierten Codes angepasst werden. Dies ist erforderlich, weil die Schnittstelle generiert wird und die exakte Namensvereinbarung für die generierten C-Funktionen von der Erweiterung abhängt und für den Benutzer möglicherweise nicht durchschaubar ist. In der vorliegenden Erweiterung wird die Kennung der Klasse im Namen für die generierten Funktionen verwendet, um einen einheitlichen Namensraum zu garantieren.

7.1 Struktur

Eine mit C-Code beschriebene Komponente hat dieselbe Struktur wie im Falle einer Beschreibung mit ESDL oder als Blockdiagramm. C-Code beschreibt den Hauptteil von Methoden oder Prozessen. Jede Codevariante wird gesondert gespeichert.

Die Spezifikation einer Komponente in C-Code ist abhängig:

- Vom Target, z. B. davon, ob der C-Code für den PC, PPC oder die CPU einer speziellen Steuereinheit bestimmt ist. Hierbei kann der Code unterschiedlich sein, da zum Beispiel die CPU einer Steuereinheit spezielle Register aufweist, die direkt adressiert werden müssen, oder weil das Endian-Format unterschiedlich ist.
- Von der Spezifikationsebene. Es kann beabsichtigt sein, mit dem C-Code die physikalische Ebene darzustellen. In diesem Falle stimmt die Implementierungsebene weitestgehend mit der physikalischen Ebene überein, z. B. wird der Typ `continuous` durch eine 64-bit-Gleitkommazahl dargestellt. Der C-Code kann sich aber auch auf der Implementierungsebene von Festkomma-Arithmetik befinden.
- Von der gewählten Implementierung, wenn sich C-Code auf der Implementierungsebene befindet, da C-Code von den Implementierungen der Variablen abhängt, insbesondere von ihren Quantisierungen.

7.1.1 Methoden und Prozesse

Für jede Methode oder jeden Prozess wird eine C-Funktion generiert. Der Funktionskopfteil wird automatisch generiert, C-Code wird nur im Hauptteil der Funktion selbst verwendet.

Beispiel:

Der Hauptteil der Methode `calc()`

```
a = b + d;
c = a * c;
```

könnte den folgenden generierten Code zum Ergebnis haben (einschließlich Funktionskopfteil), je nach der für das experimentelle Target benötigten Softwarearchitektur:

```
void QX040H28HJ8HAMDJ870S4G7MDIBQQLSM_calc (struct
QX040H28HJ8HAMDJ870S4G7MDIBQQLSM_Obj *self) {
...
   /* BEGIN handwritten code */
   /* calc 1  */a = b + d;
```

```
   /* calc 2  */c = a * c;
   /*  END  handwritten code */
...
}
```

Die Namen der für die Methoden und Prozesse von Komponenten generierten Funktionen hängen von der Codeerweiterung und von der Softwarearchitektur des generierten Codes ab. Der Benutzer hat keinen Einfluss auf diese Namen. Je nach Codeerweiterung wird ein einheitlicher Namensraum erzielt, d.h. Methoden, die zu verschiedenen Klassen gehören, können denselben Namen haben, ohne dass irgendwelche Konflikte bei der Namensgebung auftreten. Im obigen Beispiel wird die Kennung für die Komponente verwendet, um für die Methode `calc` den eindeutigen Namen `QX040H28HJ8HAMDJ870S4G7MDIBQQLSM_calc` zu generieren.

7.1.2 Variable und Funktionsparameter

Die Variablen einer Komponente werden in einer Datenstruktur aufbewahrt, die wie die Funktionskopfteile automatisch generiert wird. Der Benutzer hat keinen Einfluss auf diese Datenstruktur. Ein Teil dieser Datenstruktur besteht aus den Instanzvariablen der Komponente, die in jeder beliebigen Methode verwendet werden können. Daher müssen sie in alle generierten Funktionen übergeben werden. Diese Datenstruktur hängt auch von der Codeerweiterung ab, und die exakte Namensgebung bleibt dem Benutzer daher verborgen.

Im obigen Beispiel hat die Komponente ihre eigene Datenstruktur, welche der generierten Funktion für die Methode calc übergeben wird. Die Datenstruktur könnte wie folgt aussehen:

```
struct QX040H28HJ8HAMDJ870S4G7MDIBQQLSM_Obj  {
   ASDObjectHeader objectHeader;
   real64_Obj *a;
   real64_Obj *b;
   real64_Obj *c;
   real64_Obj *d;
};
```

Die Namen der Elemente müssen zulässige ANSI C-Kennungen sein. Zusätzlich zu den reservierten Schlüsselwörtern von C sind die Namen `self` und `this` reserviert.

Hinweis

Beim Spezifizieren von Komponenten in C-Code muss der Benutzer sicherstellen, dass die Namen von Funktionen, die im Hauptteil der Methode aufgerufen werden, nicht mit den Namen von Variablen kollidieren, die in der Schnittstelle derselben Komponente definiert sind.

Zugriff auf Elemente:

Um einen leichten Zugriff auf die Elemente der Komponente zu ermöglichen, wird für jedes Element automatisch ein Makro definiert. Auf jedes Element kann dann einfach über seinen Elementnamen zugegriffen werden.

Auf die in anderen Komponenten definierten öffentlichen Elemente kann von innerhalb von C-Funktionen aus unter Verwendung der Notation `Defining-Object.PublicElement` zugegriffen werden. Der Zugriff ist auf Basiselemente, Arrays und Matrizen beschränkt. Auf die öffentliche Schnittstelle von komplexen Elementen, die in anderen Komponenten definiert sind, z. B. unter Verwendung der Methoden `getAt`, `setAt` oder `search` und `interpolate` wie in ESDL, kann von C-Funktionen aus nicht zugegriffen werden.

Automatisch generierte `define`-Statements für Schnittstellenvariablen:

```
#define a self->a->val
#define b self->b->val
#define d self->d->val
#define c self->c->val
/* BEGIN handwritten code */
/* calc 1  */a = b + d;
/* calc 2  */c = a * c;
/*  END  handwritten code */
#undef a
#undef b
#undef d
#undef c
```

Arbeit mit Basiselementen :

Für Basistypen können die Methodennamen dieser Typen verwendet werden, wie in Kapitel 3 „Typen und Elemente“ auf Seite 91 erläutert wurde. Beim Zugreifen auf Arrays oder Matrizen kann der Index-Operator '`[]`' auf eine ähnliche Art und Weise wie bei C verwendet werden.

Da die Methodennamen eines benutzerdefinierten Typs von der Erweiterung abhängen, kann die Methode von benutzerdefinierten Typen nur bei Kenntnis des exakten generierten Funktionsnamens für die betreffende Methode aufgerufen werden. Im obigen Beispiel wird für die Methode `calc` der Funktionsname `QX040H28HJ8HAMDJ870S4G7MDIBQQLSM_calc` generiert.

Bei Verwendung von Elementen, die in ASCET definiert sind, sind diese Elemente von einem Modelltyp (entweder Basistyp oder benutzerdefiniert). Basistypen haben die folgende Standard-Implementierung, welche auf der physikalischen Ebene realisiert wird:

- `continuous = real64`
- `udisc = unsigned int32`
- `sdisc = signed int32`
- `log = int16`.

Hinweis

Elemente vom Typ `logical` *sollten nicht als Zahlen im C-Code verwendet werden, da dies von der Standard-Implementierung abhängt, welche in künftigen Versionen von ASCET geändert werden könnte.*

Die Standard-Implementierung wird durch die benutzerdefinierte Implementierung ersetzt, wenn die Spezifikationsebene umgeschaltet wird (z. B. Festkomma-Code). Elemente vom Modelltyp `logical` können z. B. als ein Bit dargestellt werden und können daher im C-Code nicht als Zahl verwendet werden.

Messages:

Messages sind Teil des Konzepts für Intratask-Kommunikation (Interprozesskommunikation), das in ASCET-Modellen verwendet wird (siehe Kapitel 1.3). Um Datenkonsistenz zu erreichen, muss die ASCET-Codegenerierung zusätzliche Message-Kopien anlegen.

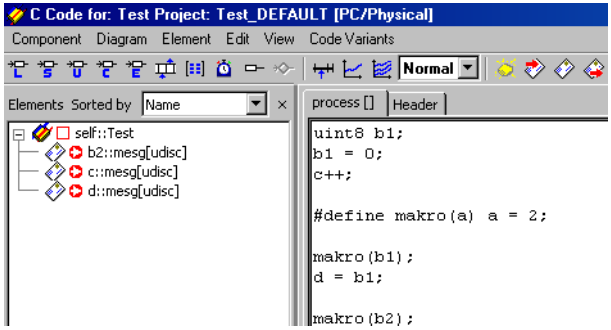
Wenn Messages im funktionalen Code verwendet werden (Lese-/Schreibzugriff), ist zusätzlicher Code erforderlich, um einen sicheren Kopiervorgang von den originalen Messages zu den lokalen Kopien zu gewährleisten. Im Hauptteil des Prozesses werden nur die lokalen Kopien verwendet. Am Ende müssen alle lokalen Kopien, deren Wert sich im Hauptteil des Prozesses geändert haben konnte, zurück in die Original-Messages geschrieben werden.

In ESDL- und Blockdiagrammkomponenten erkennt ASCET üblicherweise sehr gut, welche Messages sich in einem Prozess geändert haben. Diese Funktionalität ist jedoch nur eingeschränkt verfügbar, wenn C-Code für die Spezifikation verwendet wurde. In diesem Fall muss der Benutzer selbst auf Datenkonsistenz achten.

ASCET ist generell nicht in der Lage festzustellen, wo und wann im benutzerdefinierten C-Code in eine Variable geschrieben wird. Nur einige wenige Spezialfälle werden von ASCET erkannt, etwa wenn dem Variablennamen ein = folgt oder wenn Zuweisungsoperatoren wie ++ verwendet werden. Wenn eine Variable in einem Makro, einer extern-Funktion oder mit Hilfe von Adressoperatoren und Zeigerarithmetik geändert wird, erkennt ASCET das *nicht*.

Bei der Verwendung von Messages führt das dazu, dass die Message-Kopien zwar anfangs erzeugt, aber unter Umständen am Ende nicht zurückgeschrieben werden.

Ein einfaches Beispiel soll das verdeutlichen. Das abgebildete Modul enthält die Messages `b2`, `c` und `d`. Die Messages `c` und `d` werden direkt beschrieben, `b2` wird innerhalb eines Makros verwendet.



Im generierten Code werden anfangs für alle drei Messages Kopien angelegt (1). Da aber nur auf c und d auf eine Weise zugegriffen wird, die ASCET erkennt, werden am Ende nur diese Message-Kopien zurückgeschrieben (2). Die Änderung der Message b2, die im Makro erfolgt, wird nicht erkannt und geht verloren.

```
/*
 *-------------------------------------------------------------------------------
 *     Function definitions - Algorithms
 *-------------------------------------------------------------------------------
 */

void initClass_TEST (TEST_Class *class)
{
   class->b2 = v3_initInstance_uint32 (0, ASD_VARIABLE);
   class->d = v3_initInstance_uint32 (0, ASD_VARIABLE);
   class->c = v3_initInstance_uint32 (0, ASD_VARIABLE);
}

/*public*/
void TEST_process (void)
{
   uint32 _t1uint32;
   uint32 _t2uint32;
   uint32 _t3uint32;
   SUSPEND_HS_INTERRUPTS
   _t1uint32 = TEST_ClassObj.b2->val;
   _t2uint32 = TEST_ClassObj.c->val;
   _t3uint32 = TEST_ClassObj.d->val;
   RESUME_INTERRUPTS
   {
/* process 1    */       uint8 b1;
/* process 2    */       b1 = 0;
/* process 3    */       _t2uint32 ++;
/* process 4    */
#define makro(a) a = 2;
/* process 6    */
/* process 7    */       makro(b1);
/* process 8    */       _t3uint32  = b1;
/* process 9    */
/* process 10   */       makro(_t1uint32 );
/* process 11   */
/* process 12   */
   }
   SUSPEND_HS_INTERRUPTS
   TEST_ClassObj.c->val = _t2uint32;
   TEST_ClassObj.d->val = _t3uint32;
   RESUME_INTERRUPTS
}
```

(1)

(2)

Argumente:

Argumente von Methoden werden auf Funktionsparameter in der Parameterliste der für die Methode generierten Funktion abgebildet. Auf diese wird auch durch den Namen des Arguments zugegriffen.

Lokale Variable:

Entsprechend den allgemeinen Regeln von C können funktionslokale Variable im Hauptteil der Methode deklariert werden. Hier können nur Variable von einem C-Datentyp deklariert werden, nicht jedoch von einem ASCET Modelltyp. Insbesondere können keine lokalen Variablen von einem benutzerdefinierten Typ in Komponenten innerhalb von Spezifikationen in C verwendet werden.

```
real64 i;
for (i=0; i < 10; i++)
{
   sum = sum + a[i];
}
```

Da für jede Implementierungsvariante eine Codevariante existiert, kann der Benutzer die lokalen Variablen und ihre Datentypen im Hinblick auf die Implementierungsvariante definieren.

Kennlinien und Kennfelder:

In der Komponente definierte Kennlinien und -felder können mit Hilfe von je drei Unterprogrammen ausgewertet werden (ähnlich wie in ESDL).

Als Beispiel für eine Kennlinie wird wieder die Tabelle `LLpr` aus dem Abschnitt „Eindimensionale Tabellen“ auf Seite 137 verwendet.

0.0	1000.0	2000.0	3000.0	4000.0	5000.0	6000.0
0.0	0.8	1.1	1.5	1.8	2.0	2.2

Das Unterprogramm `CharTable1_getAt_real64_real64(charline, index)` reicht im Allgemeinen völlig für die Auswertung einer Kennlinie. Die lineare Interpolation wird für das Beispiel wie folgt durchgeführt:

```
tmpVal = CharTable1_getAt_real64_real64(LLpr,3000);
// assigns 1.5 to tmpVal

tmpVal = CharTable1_getAt_real64_real64(LLpr,2280);
// calculates interpolation factor for 2280
// interpolates value for 2280 as 1.212 and
// assigns it to tmpVal

tmpVal = CharTable1_getAt_real64_real64(LLpr,9000);
// calculates interpolation factor for 9000
// interpolates value for 9000 as 2.2 and
// assigns it to tmpVal
```

Das Trennen der Such- und Interpolationsschritte in Tabellen kann jedoch in Einzelfällen effizienter sein. In diesem Fall werden die Unterprogramme `CharTable1_search_real64(charline, index)` und `CharTable1_interpol_real64_real64(charline)` verwendet:

```
CharTable1_search_real64(LLpr, 1000);
// sets sample point to 1000

tmpVal = CharTable1_interpol_real64_real64(LLpr);
// assigns 0.8 to tmpVal
```

```
CharTable1_search_real64(LLpr, 2780);
// calculates interpolation factor for 2780
tmpVal = CharTable1_interpol_real64_real64(LLpr);
// interpolates value for 2780 as 1.412 and
// assigns it to tmpVal
```

Als Beispiel für ein Kennfeld wird wieder die Tabelle `LLpr2` aus dem Abschnitt „Zweidimensionale Tabellen“ auf Seite 139 verwendet:

y \ x	0.0	1.0	8.0	15.0
1.0	-5.0	-3.0	0.0	1.0
3.0	0.0	1.0	4.0	6.0
5.0	8.0	5.0	4.0	4.0

Für die Auswertung einer Kennlinie reicht im Allgemeinen das Unterprogramm `CharTable2_getAt_real64_real64_real64(charmap, indX,indY)`. Die lineare Interpolation wird für das Beispiel wie folgt durchgeführt:

```
tmpVal =
     CharTable2_getAt_real64_real64_real64(LLpr2,8,5);
// assigns 4.0 to tmpVal
tmpVal =
     CharTable2_getAt_real64_real64_real64(LLpr2,2,2);
// calculates interpolation factor for x=2 and y=2
// interpolates value for (2,2) as -0.571 and
// assigns it to tmpVal
tmpVal =
     CharTable2_getAt_real64_real64_real64(LLpr2,20,9);
// calculates extrapolation factor for x=20, y=10
// extrapolates value for (20,10) as 5.0 and
// assigns it to tmpVal
```

Das Trennen der Such- und Interpolationsschritte in Tabellen kann jedoch in Einzelfällen effizienter sein. In diesem Fall werden die Unterprogramme `CharTable2_search_real64_real64(charmap, indX,indY)` und `CharTable2_interpol_real64_real64_real64(charmap)` verwendet:

```
CharTable2_search_real64_real64(LLpr2, 1, 3);
// sets x sample point to 1 and y sample point to 3
tmpVal =
     CharTable2_interpol_real64_real64_real64(LLpr2);
// assigns 1.0 to tmpVal
```

```
CharTable2_search_real64_real64(LLpr2,4,4);
// calculates interpolation factor for x=4, y=4
tmpVal =
     CharTable2_interpol_real64_real64_real64(LLpr2);
// interpolates value for (4,4) as 3.143 and
// assigns it to tmpVal
```

7.1.3 Header

Neben der Beschreibung der Methoden in Form von C-Code kann ein Header für Makros und für enthaltene Dateien definiert werden. Dieser Header hat einen lokalen Geltungsbereich, der auf die Komponente beschränkt ist. Daher wird keine zusätzliche Header-Datei generiert, sondern die Definitionen werden in die generierte Datei in C-Code kopiert.

7.2 Externer Quellcode

Existierender C-Code kann integriert werden, indem externe C-Code-Quelldateien importiert werden. Zu diesem Zweck kann eine C-Code-Datei mit einer entsprechenden Header-Datei mit jeder Codevariante einer Komponente verknüpft werden. Die C-Code-Datei enthält standardmäßige Definitionen von C-Funktionen; die Header-Datei enthält die entsprechenden Funktionsdeklarationen und Strukturdefinitionen. Die definierten Funktionen können über die standardmäßigen Konventionen von C aufgerufen werden. Es ist möglich, Zeiger zu übergeben und definierte Strukturen zwischen Methoden oder Prozessen der Komponente und den Funktionen in dem mit ihr verknüpften C-Code zu teilen.

Als Alternative zur Verwendung einer C-Code-Datei kann eine Objektdatei mit einer entsprechenden Header-Datei mit einer Komponente verknüpft werden. Wie die Header-Dateien der Komponente selbst ist der Bereich der Header-Dateien der mit ihr verknüpften Quellen lokal, d.h. sie werden in den generierten C-Code kopiert.

Die angefügte C-Datei wird gesondert kompiliert und mit den anderen (generierten und kompilierten) C-Quelldateien verbunden. Infolgedessen existiert diese kompilierte Einheit innerhalb eines beliebigen gegebenen Kontexts nur einmal. Falls der Code und die darin enthaltenen Daten von mehreren Instanzen derselben Komponente gemeinsam verwendet werden, teilen alle Instanzen dieselbe kompilierte Einheit.

Hinweis

Die Daten in einer angefügten C-Datei werden von mehreren Instanzen der Komponente geteilt, und es erfolgt nicht für jede dieser Instanzen eine Instanzbildung dieser Daten

Zusätzlich ist es möglich, `include`–Anweisungen im C-Code zu haben. Die include-Dateien werden jedoch nicht in der Datenbank gespeichert, sondern sie werden auf dem Dateisystem gespeichert. Die `include`-Anweisung muss den Dateipfad zu diesen `include`-Dateien enthalten. Der C-Code hängt daher nicht nur von Elementen in der Datenbank ab, sondern auch von der Dateistruktur der aktuellen Installation. Daher ist beim Datenaustausch Vorsicht geboten, da diese Dateien dem ASCET System nicht bekannt sind.

Hinweis

Wenn include-Dateien verwendet werden, muss der Benutzer selbst auf die korrekten Referenzen auf diese Dateien achten.

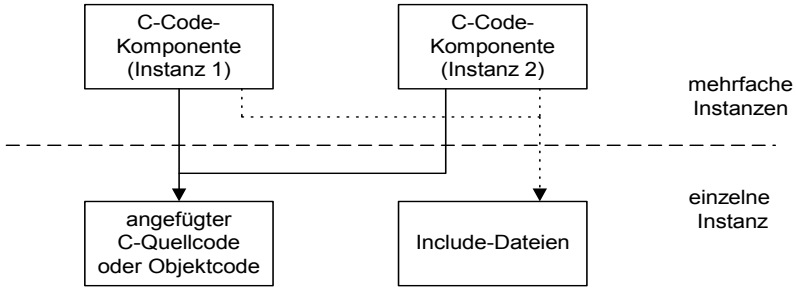


Abb. 7-2 Verwendung von externem Quellcode

7.3 Modellschnittstelle

In früheren ASCET-Versionen konnten die Namen von Klassen und Methoden nur dann im C-Code verwendet werden, wenn sie durch ein Fluchtsymbol („@") gekennzeichnet waren. Durch diesen Mechanismus wurde die sogenannte Modellschnittstelle (oder Programming Model Interface, PMI) aktiviert. In ASCET 4x und 5.x werden Klassen- und Methodennamen automatisch erkannt, d.h. es ist kein Fluchtsymbol mehr erforderlich und das PMI wird standardmäßig verwendet (siehe Beschreibung der Codegenerierungsoptionen im ASCET-Handbuch für weitere Details). Das Fluchtsymbol ist obsolet und sollte in C-Code-Modellen nicht mehr verwendet werden. Für die Kompatibilität mit älteren Versionen kann das Fluchtsymbol noch verwendet werden, indem die entsprechend Codegenerierungsoption geändert wird.

7.4 Zugriffsmakros

Analog zu den Zugriffsmethoden in ESDL, bietet ASCET für C-Code Komponenten Zugriffsmakros an. Mit Hilfe dieser Makros kann der Anwender vordefinierte Operationen im C-Code verwenden. Die Makros sind im folgenden dargestellt.

Direktzugriff

Auf Elemente von Klassen, die in C-Code Komponenten eingebettet sind, kann unter Verwendung der Makros

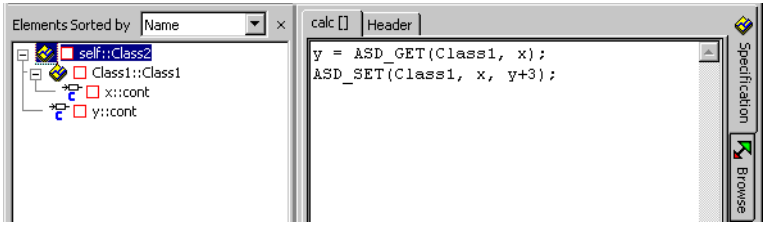
```
ASD_GET(receiver, variable);
```

und

```
ASD_SET(receiver, variable, value);
```

direkt zugegriffen werden.

Beispiel:



Länge von Arrays

Die aktuelle Länge eines Arrays kann mittels des Makros

```
ASD_LENGTH (receiver);
```

ermittelt werden.

Beispiel:

```
z = ASD_LENGTH(array);
```

Zugriff auf Ressourcen

Ressourcen können mithilfe der Makros

```
ASD_RESERVE(resource);
```

und

```
ASD_RELEASE(resource);
```

gesperrt und wieder freigegeben werden.

Zugriff auf private Methoden

Private Methoden können mittels

```
self.
```

angesprochen werden, z.B.:

```
y = self.method_private(x);
```

Bereitstellen von Arrays für die Verwendung in externem C-Code

Unter Verwendung des Makros

```
ASD_USE_ARRAY_EXTERNAL(array)
```

können Array-Zugriffe von der ASCET-internen Darstellung in die übliche C-Code darstellung konvertiert werden. Das Makro steht synonym für:

```
&array[0]
```

Beispiel:

```
y = c_function(ASD_USE_ARRAY_EXTERNAL(array));
```

8 Zeitkontinuierliche Systeme

Bei der Funktionsentwicklung von Steuergeräte-Software und zur Simulation von Steuergeräten werden die umfassenden Möglichkeiten von ASCET zur Modellierung diskreter Systeme eingesetzt. Dagegen stellt die zum Steuergerät gehörige Regelstrecke ein zeitkontinuierliches physikalisches System dar, das durch Differentialgleichungen beschrieben wird.

Beispiele für zeitkontinuierliche Systeme sind der Antriebsstrang oder die Radaufhängung von Fahrzeugen (mechanisches System), der Verbrennungsvorgang im Zylinderraum (thermodynamisches System), der Bremskreis eines Fahrzeugs (hydraulisches oder pneumatisches System), die Fahrzeugbatterie (elektrisches oder elektrochemisches System). Außerdem kommen natürlich zunehmend auch mechatronische Systeme vor, bei denen z. B. die Mechanik eines Stellers mit einer lokalen elektronischen Steuerung verbunden ist oder ein intelligenter Sensor das physikalische Signal elektronisch aufbereitet.

ASCET unterstützt die Modellierung und Simulation solcher zeitkontinuierlicher Systeme durch die sogenannten CT-Blöcke. CT steht für *Continuous Time* und bezeichnet Elemente, die quasi zeitkontinuierlich modelliert bzw. berechnet werden. Dabei basiert das zeitkontinuierliche Modellieren in ASCET auf der Zustandsraum-Darstellungsform, der Standardbeschreibungsform beim Design eines zeitkontinuierlichen Systems. Anhand dieser Darstellung lassen sich CT-Basisblöcke mit nichtlinearen gewöhnlichen Differentialgleichungen (engl. ordinary differential equation - ODE) erster Ordnung und nichtlinearen Ausgabegleichungen beschreiben. ASCET stellt mehrere echtzeitfähige Integrationsverfahren zur Verfügung, um diese Differentialgleichungen optimal zu lösen.

Das zeitkontinuierliche Modell kann auf modulare und hierarchische Art in Blöcke strukturiert werden. Zeitkontinuierliche Modelle können mit ASCET Controller-Spezifikationen kombiniert werden, um kombinierte Modelle, sog. hybride Projekte zu bilden. Diese hybriden Projekte können verwendet werden, um eine Controller-Spezifikation anhand eines Modells des tatsächlichen zu steuernden technischen Prozesses zu erproben.

Modell und Simulations-Experiment sind streng getrennt; ein Modell enthält die modulare und hierarchische Systembeschreibung, ein Experiment enthält die gewählte Datenmenge, den Integrationsalgorithmus und die gewählte Visualisierungskonfiguration samt Eingabemöglichkeit für Parameter. Die Resultate sind akkurate, wiederverwendbare Modelle und hohe Flexibilität. Auf Experimentierebene kann jede Modellvariable flexibel geändert und gemessen werden. Die gewählte Integrationsschrittgröße sowie der Integrationsalgorithmus können während der Simulation verändert werden - ohne zeitaufwendige Neukompilierung des Modells oder des aktuellen Experiments.

8.1 Strukturierung der zeitkontinuierlichen Modelle

Im folgenden werden die verschiedenen Strukturierungsmöglichkeiten eines Modells mit Basisblöcken, Strukturblöcken und grafischen Hierarchien beschrieben.

8.1.1 Modellierung mit Basisblöcken und Strukturblöcken

Modelle zeitkontinuierlicher Systeme können modular und hierarchisch strukturiert werden. Grundelement ist der Continuous Time Basisblock, kurz CT-Basisblock, in dem mit den Hochsprachen ESDL oder C[1] das Teilmodell in Form von Differentialgleichungen, algebraische Gleichungen, Formeln und Zuweisungen beschrieben wird.

Zeitkontinuierliche Blöcke (CT-Blöcke) bestehen aus Eingängen, Ausgängen, Parametern und diskreten sowie kontinuierlichen Zuständen mit mehreren Dimensionen, Bereichen und Datentypen. Außerdem werden zeitkontinuierliche und diskrete Gleichungen und Ausgabegleichungen sowie eine Initialisierungs- und Abschlusssequenz unterstützt. Zustandsereignisse, Software- und Hardwareereignisse (Interrupts) können ebenfalls behandelt werden.

Komplexere zeitkontinuierliche Modelle können mit Hilfe des Block Diagramm Editors (BDE) zu CT-Strukturblöcken aufgebaut werden. Zu einer CT-Struktur können mehrere CT-Basisblöcke und/oder CT-Strukturblöcke in einem Blockdiagramm mit dem Blockdiagrammeditor zusammengesetzt und verbunden werden. Abb. 8-1 zeigt einen einfachen CT-Strukturblock, der aus zwei CT-Basisblöcken aufgebaut ist.

1. C sollte nur in zwingenden Ausnahmen verwendet werden, da ASCET automatische Kontrollfunktionen (Semantikprüfung, Berechnungsreihenfolge) nur für ESDL bereitstellt.

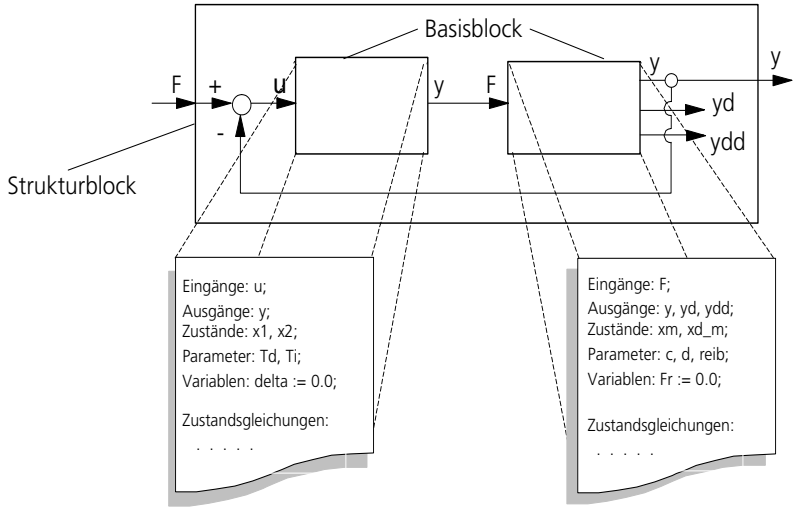


Abb. 8-1 CT-Strukturblock - aufgebaut aus zwei CT-Basisblöcken

Mehrere CT-Strukturblöcke und CT-Basisblöcke können wiederum zu einem neuen CT-Strukturblock zusammengefasst werden. Abb. 8-2 zeigt die Strukturierungsmöglichkeiten mit CT-Strukturen.

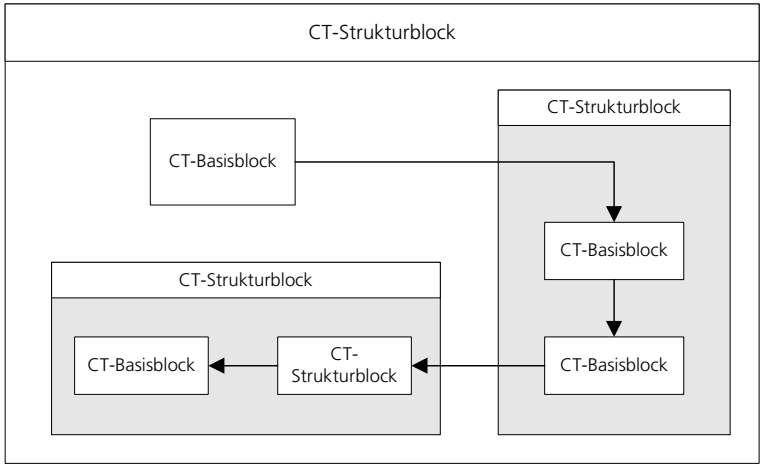


Abb. 8-2 Modellierung mit CT-Strukturblöcken.

Der korrekte Auswertungsablauf der CT-Blöcke wird automatisch ermittelt. CT-Basisblöcke und/oder CT-Strukturblöcke können zusammen mit Standard-ASCET Strukturen in hybriden Projekten zusammengefasst werden.

CT-Basisblöcke werden zum Beschreiben kleiner physischer Komponenten wie Bremsen, Räder usw. benutzt. Mit CT-Strukturblöcken werden komplexere Einheiten beschrieben, z. B. eine Kraftübertragungseinrichtung oder ein komplettes Fahrzeugmodell. CT-Basisblöcke und CT-Strukturblöcke werden jeweils in der Datenbank gespeichert und stehen anderen Modellen zur Verfügung. So kann einfach eine Modellbibliothek aufgebaut werden. Änderungen in Blöcken oder Strukturen werden automatisch von allen Modellen innerhalb einer Datenbank übernommen. Dies hat den Vorteil, dass die Basiselemente nur an einer Stelle gepflegt werden müssen und Korrekturen von allen Modellen, die sich in der selben Datenbank befinden, automatisch übernommen werden. Auf der anderen Seite muss natürlich sichergestellt werden, dass die Basiselemente kompatibel bleiben.

8.1.2 Modellieren mit grafischen Hierarchien

Ein CT-Strukturblock, der aus vielen CT-Basisblöcken und/oder CT-Strukturblöcken aufgebaut ist, kann übersichtlicher gestaltet werden, wenn mehrere zusammengehörige CT-Blöcke zu einer grafischen Hierarchie zusammengefasst werden (siehe Abb. 8-3). Grafische Hierarchien und CT-Strukturen können in neuen Hierarchien zusammengefasst werden - die Abarbeitungsreihenfolge wird durch grafische Hierarchien nicht beeinflusst. Im Blockdiagrammeditor werden grafische Hierarchien durch einen doppelten Rahmen gekennzeichnet.

Grafische Hierarchien werden insbesondere dann verwendet, wenn die einzelnen CT-Blöcke eng gekoppelt sind und eine feste Rechenreihenfolge innerhalb eines Integrationsschrittes haben müssen. Durch grafische Hierarchien können algebraische Schleifen (siehe Abschnitt „Algebraische Schleife“ auf Seite 219 und Abschnitt „Unterschied: Grafische Hierarchie - CT-Strukturblock“ auf Seite 220), die durch CT-Strukturblöcke entstehen können, vermieden werden. Die richtige Rechenreihenfolge wird durch eine automatische Sequenzierung sichergestellt. Grafische Hierarchien können nicht einzeln abgespeichert werden, sondern nur zusammen mit dem entsprechenden Strukturblock.

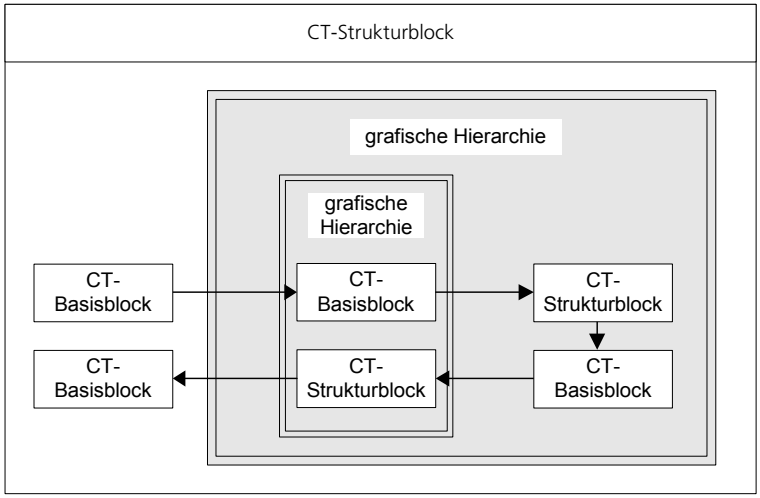


Abb. 8-3 Grafische Hierarchie

8.1.3 Experiment

Basis- und Strukturblöcke können in einem Simulationsexperiment ausgewertet werden. Im Experiment wird die Integrationsmethode, die Anregung des Modells und die Visualisierung der Ergebnisse ausgewählt und spezifiziert. Zu einem (Teil-) Modell können verschiedene Experimentiereinstellungen gespeichert werden.

8.1.4 Projekt und hybrides Projekt

Das Echtzeit-Experiment wird im Projekt definiert. Im Projekt können sowohl Basisblöcke als auch Strukturblöcke direkt genutzt werden. Außerdem kann nur im Projekt jedem integrierten Basisblock bzw. Strukturblock eine eigene Integrationsmethode mit Schrittweite zugewiesen werden. So kann, wenn die Rechenkapazität begrenzt ist, dem Modellteil mit hoher Dynamik mehr Rechenzeit zugeteilt werden als einem anderen, weniger dynamischen Modellteil.

Im Falle eines Modells, in dem Controller und Streckenmodell kombiniert werden soll, kann ein hybrides Projekt definiert werden, d.h. ein Projekt, das sowohl CT-Blöcke als auch Standard-ASCET Komponenten beinhaltet. Im hybriden Projekt kann also Strecke und Steuergerät in einem Modell simuliert werden (hybride Simulation).

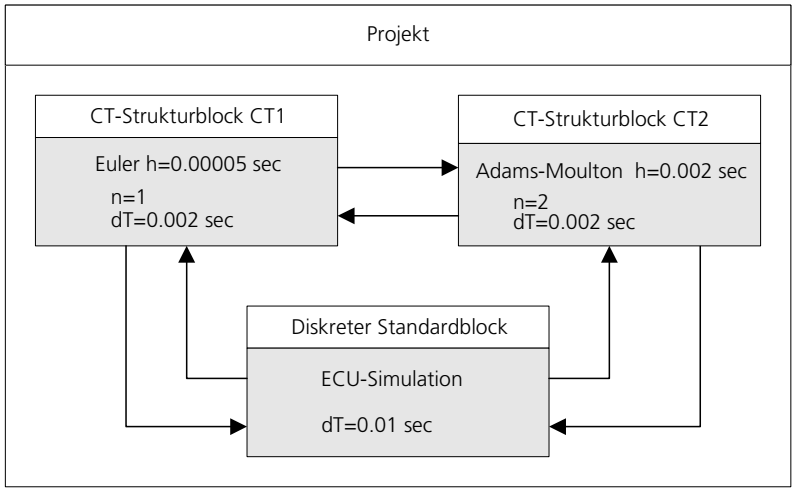


Abb. 8-4 Projekt

Die Kommunikation zwischen CT-Blöcken einerseits und Controller-Modulen andererseits erfolgt durch explizites Verbinden von Ein- und Ausgängen im Blockdiagrammeditor (Einzelheiten zu Projekten siehe Kapitel „Projekte“ auf Seite 13).

Das Experiment kann online auf der echtzeitfähigen Simulations-Hardware oder offline auf dem PC durchgeführt werden (sofern keine spezielle Hardware für das Experiment verfügbar bzw. eingebunden sein muss).

8.2 Lösen von Differentialgleichungen - Integrationsalgorithmen

Wegen der Komplexität der Gleichungen in zeitkontinuierlichen Modellen und der häufig auftretenden Nichtlinearitäten ist ein analytisches Lösen im allgemeinen nicht möglich. Daher muss das System von Differentialgleichungen mit einem Integrationsalgorithmus numerisch gelöst werden.

Wenn nur CT-Blöcke in einer CT-Struktur simuliert werden, nutzt ASCET einen globalen Integrationsalgorithmus. Die Kombination mit diskreten Controller-Modellen ist nur auf der Projektebene möglich (kombiniertes Modellieren in einem hybriden Projekt). In Projekten wird auch das Modellieren mit mehreren CT-Strukturen mit unterschiedlichen Integrationsverfahren unterstützt.

Um bei Modellierung und Simulation hohe Flexibilität und kurze Iterationszyklen zu sichern, kann die Konfiguration der Integrationsmethode, d.h. die Integrationsmethode selbst und die zugehörige Integrationsschrittweite, beim Experimentieren interaktiv gewählt und geändert werden.

Keine der Integrationsmethoden ist für alle Modellarten ideal. Die Geschwindigkeit und Genauigkeit der verschiedenen Algorithmen variieren bei unterschiedlichen Modelleigenschaften, z. B. Nichtlinearitäten, Unstetigkeiten und dynamischem Verhalten. Eine pauschale Aussage zur Geschwindigkeit der Verfahren kann nicht gemacht werden, da je nach Modell und Integrationsverfahren die Schrittweite angepasst wird. Im folgenden werden einige Hinweise zur Wahl einer geeigneten Integrationsmethode gegeben. Detailliertere Informationen siehe z.B.

> Addison, C. A.; Enright, W. H.; *et al.,* A Decision Tree for the Numerical Solution of Initial Value Ordinary Differential Equations. *ACM Transitions on Mathematical Software 17, 1, March 1991,* Kapitel „Continuous Time Integration Algorithms".

ASCET bietet die folgenden echtzeitfähigen Integrationsmethoden mit fester Schrittweite an:

- Euler
- Mulstep 2
- Heun
- Adams-Moulton 2
- Runge-Kutta 4

Für komplexere oder steife Differentialgleichungssysteme, die genauere Berechnung benötigen, stellt ASCET die folgenden schrittweitengesteuerten iterativen Integrationsmethoden zur Verfügung:

- Dormand/Prince RK5
- Calvo 6(5)
- Dormand/Prince RK8
- Implicit RK2
- Implicit RK4
- Implicit Gear 1

- Implicit Gear 2

Diese Integrationsmethoden arbeiten nicht mit einer vorgegebenen Schrittweite, sondern passen die Schrittweite während der Berechnung iterativ an, um eine bestimmte Genauigkeit zu erreichen. Daher sind sie *nicht* echtzeitfähig.

Die impliziten Integrationsmethoden können aus technischen Gründen nur mit neueren Borland- und Microsoft-Compilern verwendet werden, nicht jedoch mit dem mitgelieferten Borland C 4.5. Sie sind der GNU Scientific Library entnommen. Die in früheren ASCET-Versionen angebotene Integrationsmethode Gear 4 steht nicht mehr zur Verfügung.

8.2.1 Übersicht über die verschiedenen Integrationsverfahren

Die Differentialgleichung liegt in der Zustandsform

$x'(t) = f(x,t);$ mit $x(t=0) = x_0$

vor.

In der folgenden Tabelle sind einige Eigenschaften der implementierten Integrationsverfahren aufgeführt:

- Die globale Fehlerordnung p des Diskretionsfehlers, der proportional zu h^p ist, wobei h die Integrationsschrittweite ist.
- Die Anzahl der Funktionsauswertungen pro Integrationsschritt. Dazu werden jeweils die lokalen Variablen zurückgesetzt und die Methoden `nondirectOutputs`, `directOutputs`, `derivatives` ausgeführt. Daraus kann zusammen mit der Integrationsschrittweite die Geschwindigkeit des Verfahrens abgeschätzt werden.
- Einschritt-/Mehrschrittverfahren (ESV/MSV): Einschrittverfahren verwenden für den nächsten Schritt nur den letzten Schätzwert. Mehrschrittverfahren dagegen berücksichtigen die letzten n Schätzwerte.
- Bei einem Prädiktor-Korrektorverfahren (P-K) wird zuerst mit einem Integrationsverfahren ein Schätzwert berechnet, der dann mit einem zweiten Verfahren korrigiert wird.
- Feste oder variable Schrittweite.

Die folgende Tabelle fasst die Eigenschaften der Integrationsverfahren mit fester Schrittweite zusammen (in Klammern ist der Zeitpunkt der Funktionsauswertung bzw. der berücksichtigten Stützstellen bei MSV angegeben).

Integrations-verfahren	Fehler-ordnung	Funktions-auswertungen /Schritt	ESV/MSV	P-K	Schritt-weite
Euler	1	1 (t)	ESV	nein	fest
Mulstep 2	2	1 (t)	MSV (t-h, t)	nein	fest
Heun	2	2 (t, t+h)	ESV	ja	fest
Adams-Moulton	2	2 (t, t+h)	MSV(t-h, t)	ja	fest
Runge-Kutta 4	4	4 (t, t+h/2, t+h/2, t+h)	ESV	nein	fest

Damit die Integrationsverfahren in Echtzeit angewandt werden können, ist das jeweilige Verfahren mit relativ wenigen Funktionsauswertungen pro Integrationsschritt und entsprechend niedriger Fehlerordnung implementiert.

Euler

Die Euler-Integrationsmethode ist die einfachste verfügbare Integrationsmethode. Als Einschrittverfahren, mit nur einer Funktionsauswertung pro Integrationsschritt, ist die Zykluszeit am kleinsten, d.h. das Verfahren ist relativ schnell und wird deshalb oft zur Echtzeitsimulation verwendet.

Mathematische Formel
`x(t+h)=x(t)+h*f(x,t)`

Der Stabilitätsbereich ist groß, jedoch ist der Diskretionsfehler – bei gleicher Schrittgröße – typischerweise größer als bei den anderen Verfahren (niedrigste Ordnung).

Mulstep

Die Mulstep-Integrationsmethode ist ein Mehrschrittverfahren und wird für Modelle ohne stark variierende Eigenwerte verwendet. Die Zykluszeit für einen Integrationsschritt ist nur etwas größer als beim Euler-Verfahren, da pro Integrationsschritt nur eine Funktionsauswertung erfolgt. Die Fehlerordnung ist jedoch 2.

Mathematische Formel
`x(t+h)=x(t)+h(3/2*f(x,t)-1/2f(x,t-h))`

Heun

Die Heun-Integrationsmethode wird für Modelle ohne stark variierende Eigenwerte verwendet. Die Zykluszeit ist doppelt so groß wie beim Euler-Verfahren.

Mathematische Formel

Prädiktor: $\bar{x}(t+h)=x(t)+h*f(x,t)$ (Euler)
Korrektor: $x(t+h)=x(t)+h/2*(f(x,t)+f(\bar{x},t+h))$

Adams-Moulton

Auch die Adams-Moulton-Integrationsmethode ist für Modelle ohne stark variierende Eigenwerte geeignet. Im Gegensatz zu den vorherigen Algorithmen sollte das Modell ein glattes Verhalten haben. Die Zykluszeiten beim Adams-Moulton und Heun-Algorithmus stimmen nahezu überein

Mathematische Formel

Prädiktor: $\bar{x}(t+h)=x(t)+h/2(3f(x,t)-f(x,t-h))$ (Adams-Bashforth)
Korrektor: $x(t+h)=x(t)+h/2(f(x,t)+f(\bar{x},t+h))$

Runge-Kutta 4

Die Runge-Kutta-Integrationsmethode ist am besten geeignet für Modelle ohne stark variierende Eigenwerte. Für diese Modellarten ist die Robustheit dieser Integrationsmethode sehr gut. Langsamstes, aber auch genauestes Verfahren bei vergleichbarer Schrittweite. Deshalb kann die Schrittweite deutlich vergrößert werden.

Mathematische Formel

$x(t+h)=x(t)+h/6(K_1+2K_2+2K_3+K_4)$
mit
$K_1 = f(x,t)$
$K_2 = f(x + K_1*h/2,\ t + h/2)$
$K_3 = f(x + K_2*h/2,\ t + h/2)$
$K_4 = f(x + K_3*h,\ t + h)$

Integrationsmethoden mit variabler Schrittweite

Wird eine hohe Genauigkeit benötigt, so muss die Schrittweite teilweise sehr stark herabgesetzt werden, um die gewünschten Ergebnisse zu erzielen. Dies kann die Laufzeit der Berechnung erheblich verlängern. Liegt ein steifes Differenzialgleichungsmodell vor, ist die Berechnung teilweise mit Verfahren mit fester Schrittweite kaum mehr möglich. Adaptive Verfahren arbeiten fehlerto-

leranzgesteuert, sodass die Schrittweite lediglich an den kritischen Stellen herabgesetzt werden muss. Aufgrund der variablen Schrittweite sind diese Integrationsmethoden nicht echtzeitigfähig.

Falls während der Berechnung die gewünschte Genauigkeit aufgrund anderer Parameter nicht erreicht werden kann, wird vom Experiment eine entsprechende Meldung im Monitorfenster ausgegeben. Dies geschieht wenn die maximale Integrationstiefe zu niedrig oder die minimale Schrittweite zu hoch angesetzt wurde.

9 Zeitkontinuierliche Basisblöcke

Zeitkontinuierliche Basisblöcke (CT-Basisblöcke, CT steht für *continuous time*) werden im allgemeinen verwendet, um kleine und unabhängige physikalische Komponenten zu beschreiben, die in mehreren Modellzusammenhängen verwendbar sind. Basisblöcke können im CT-Block-Editor spezifiziert werden. Die Blockschnittstelle wird interaktiv spezifiziert und die Dynamik der physikalischen Komponente wird durch Differential- und algebraische Gleichungen beschrieben.

9.1 Grundlagen

Zeitkontinuierliche Basisblöcke werden entweder im C-Code- oder ESDL-Editor spezifiziert. Beide Editoren erscheinen für die Spezifikation von CT-Blöcken leicht verändert. Die Blockinterna, d.h. die Differential- und algebraischen Gleichungen sowie die Steuerstrukturen werden innerhalb vordefinierter Methoden beschrieben. Die korrekte Auswertungsreihenfolge, die zum korrekten, zeitkontinuierlichen Modellieren erforderlich ist, wird automatisch abgeleitet (Sequenzierung). Die vordefinierte Methodenstruktur kann vom Benutzer nicht verändert werden.

In Basisblöcken können Modelle mittels nichtlinearer, gewöhnlicher Differentialgleichungen (ODE) erster Ordnung und nichtlinearer Ausgabe-Gleichungen beschrieben werden. Um ein System höherer Ordnung zu beschreiben, muss eine Umwandlung in mehrere Differentialgleichungen erster Ordnung erfolgen. Die nachstehende Tabelle veranschaulicht die Transformation eines Systems zweiter Ordnung in die Darstellung im Zustandsraum:

Eine Differentialgleichung 2. Ordnung	**Zwei Differentialgleichungen 1. Ordnung**
$T^2*x'' + 2.0*d*T*x' + x = K*in;$	$x' = xp;$
	$xp' = (K*in - (2.0*d*T*xp) - x) / T^2;$

Tab. 9-1 Auflösung einer Differentialgleichung zweiter Ordnung

Die Gleichungen können in ESDL oder C geschrieben werden. Die Benutzung von ESDL garantiert eine targetunabhängige Spezifikation und fortgeschrittene Semantiküberprüfungen. Bei Verwendung von C steht der gesamte Funktionsumfang der Programmiersprache C zur Verfügung. Der Nachteil bei C ist, dass keine semantische Analyse möglich ist. Solange mit ANSI C gearbeitet wird, ist noch weitgehend targetunabhängiges Modellieren möglich, jedoch nicht mehr, wenn spezielle Sprachdialekte, z.B. für spezielle Hardwareoptimierung verwendet werden soll. Außerdem muss bei C das Verhalten des Blockes als direkt oder nicht-direkt spezifiziert werden.

9.2 Verfügbare Elemente und Methoden

Zeitkontinuierliche Basisblöcke unterscheiden sich in einigen Elementen von diskreten Modulen oder Klassen. Es sind folgende Elemente vorhanden:

- Eingänge (Inputs)
- Ausgänge (Outputs)
- Kontinuierlicher Zustand (Continuous State)
- Diskreter Zustand (Discrete State)
- Schrittlokale Variablen (Steplocal Variables)
- Parameter (Parameters)
- Abhängige Parameter (Dependent Parameters)
- Konstanten (Constants)
- Kennfelder (OneD / TwoD Table Parameters)

Jeder Elementtyp kann unterschiedliche Dimensionen, Bereiche und Datentypen haben (vgl. Abschnitt „Blockschnittstellen“ auf Seite 199). Die Abbildung zeigt die verschiedenen Datentypen (mit ihrer jeweiligen Schaltfläche) und die zur Verfügung stehenden vordefinierten Methoden, die mit diesen Daten arbeiten.

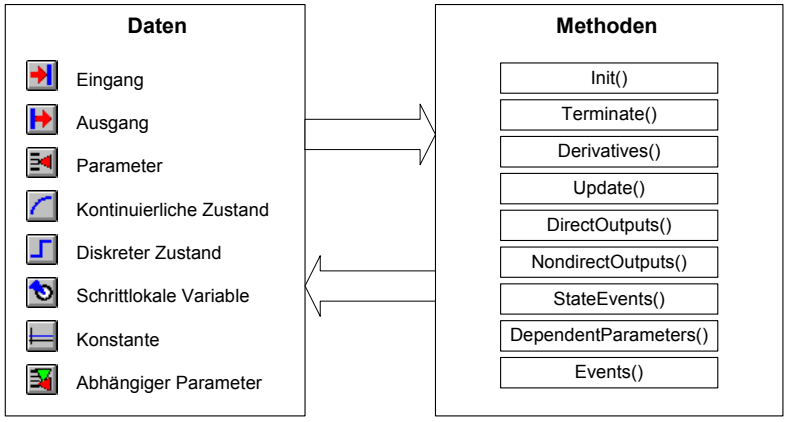


Abb. 9-1 Verschiedene Datentypen und Methoden

9.2.1 Modellieren mit zeitkontinuierlichen Basisblöcken

Innerhalb eines zeitkontinuierlichen Basisblocks können die Interna des zu modellierenden Systems mit der Modellbeschreibungssprache ESDL oder direkt in C beschrieben werden. Bei Verwendung der targetunabhängigen Modelliersprache ESDL stellt eine fortgeschrittene, semantische Überprüfung ein korrektes Modell sicher. Daher sollten nur targetabhängige Echtzeitblöcke direkt in C modelliert werden. Allgemein wird die Verwendung von ESDL empfohlen.

Das Verhalten des Blocks wird in einem festen Rahmen beschrieben, d.h. mit einer festen Anzahl von Methoden. Jede Methode hat eine spezifische Bedeutung, z. B. die Berechnung von Ableitungen oder Ausgängen. Im Gegensatz zu Standard-ASCET-Modellen ist die Ausführungsreihenfolge festgelegt (siehe Abschnitt „Auswertungsablauf" auf Seite 202), und die Methoden werden automatisch eingeplant.

9.3 Blockschnittstellen

Zur Modellierung mit zeitkontinuierlichen Basisblöcken stehen etwas andere Elemente (Schnittstellen, Speicherelemente) als bei diskreten Modulen oder Klassen zur Verfügung. Im folgenden werden die verfügbaren Elementarten beschrieben.

Eingänge (Input): Blockeingaben müssen mit *Eingängen* beschrieben werden. In jedem Auswertungsschritt werden alle Eingabevariablen eingelesen.

Ausgänge (Output): Blockausgaben müssen mit *Ausgängen* beschrieben werden. In jedem Auswertungsschritt werden alle Ausgabevariablen aktualisiert.

Kontinuierlicher Zustand (Continuous State): Für die Beschreibung gewöhnlicher Differentialgleichungen werden Zustandsvariablen benötigt. Jede Zustandsvariable arbeitet als „Speicherelement"; ein Beispiel seien Entfernung und Geschwindigkeit eines sich bewegenden Massepunktes. Kontinuierliche Zustandsvariablen werden nur vom Differentialoperator `ddt` verwendet.

Diskreter Zustand (Discrete State): Eine diskrete Zustandsvariable ist ein Speicherelement. Sie kann benutzt werden, um einen Variablenwert von einem Berechnungsschritt zum nächsten zu halten, z. B. den Wert eines Zählers. Diskrete Zustandsvariablen sind zu den Variablen in diskreten Klassen oder Modulen äquivalent. Diskrete Zustandsvariable können nicht vom Differentialoperator `ddt` verwendet werden.

Schrittlokale Variable (Steplocal Variable): Schrittlokale Variablen werden benutzt, um Zwischenwerte während der Berechnung eines Auswertungsschrittes zu speichern. Diese Variablen sind in allen Blockmethoden sichtbar. Der Wert einer schrittlokalen Variable ist nur in einem Auswertungszyklus gültig; die Variable wird am Anfang jedes Iterationsschrittes neu initialisiert. Soll

der Wert in einer anderen Methode ausgewertet werden, muss die Abarbeitungsreihenfolge der Methoden berücksichtigt werden (Schreiben vor Lesen sicherstellen).

Parameter (Parameter): Parameter werden zum Erstellen eines physikalischen Modells verwendet. Normalerweise entspricht ein Parameter einer charakteristischen Eigenschaft eines realen Systems, z. B. Masse, Länge oder Dämpfungskonstante. Durch effiziente Nutzung der Parametrisierung kann eine generische Modellbibliothek systematisch aufgebaut werden. Parameter können während der Simulation in der Experimentierumgebung variiert werden (im Verstellfenster).

Abhängiger Parameter (Dependent Parameter): Wenn ein Parameter von anderen abhängt, z. B. in unterschiedlichen Koordinatensystemen beschriebene Parameter, sollte er nur dann neu berechnet werden, wenn sich ein Parameter, von dem er abhängt, geändert hat. Diese Art von Parameter-Verhalten kann durch abhängige Parameter beschrieben werden. Sie werden nur bei Änderungen asynchron in der Methode *dependentParameters* neu berechnet.

Beispiel: `m_Fahrzeug = m_Leer + m_Zuladung`.

Wird die Zuladung im Experiment geändert, wird in der Methode `dependentParameters` die Fahrzeugmasse neu berechnet.

Konstante (Constant): Konstanten beschreiben systemweite Größen, die sich während eines Experimentes nicht ändern, z. B. die Gravitationskonstante.

Dimensionen, Geltungsbereiche und Datentypen: Zu jeder verfügbaren Elementart gibt es verschiedene Dimensionen, Geltungsbereiche und Datentypen. Die möglichen Kombinationen sind hier aufgeführt:

Kombinationen / Elemente	Dimension			Geltungsbereich		Datentyp			
	scalar	array	record	local	global	logic	sdisc	udisc	cont
input	x	x	x	x		x	x	x	x
output	x	x	x	x		x	x	x	x
discrete state	x	x		x		x	x	x	x
continuous state	x	x		x					x
steplocal variable	x	x		x		x	x	x	x
parameter	x	x		x	x	x	x	x	x
dependent parameter	x	x		x		x	x	x	x
constant	x	x		x	x	x	x	x	x

Abb. 9-2 Dimensionen, Geltungsbereiche und Datentypen

9.4 Block-Methoden

Die in CT-Basisblöcken zur Verfügung stehenden Methoden (Typ und Anzahl) sind vorgegeben und können nicht vom Benutzer verändert werden. Jede Methode hat einen spezifischen Zweck, wie beispielsweise die Berechnung von Ableitungen oder Ausgängen. Die Auswertungsreihenfolge der Methoden ist festgelegt; die Methoden werden daher automatisch abgearbeitet. Nicht jede Methode in einem CT-Basisblock muss verwendet werden.

Folgende Methoden stehen in CT-Basisblöcken zur Verfügung:

`init():` Die `init()`-Methode wird einmal am Start oder Neustart eines Experiments aufgerufen. In der Methode `init()` kann Code für die Initialisierung des Blockes spezifiziert werden, z. B. zum Modellieren des Hochfahrens eines Modells oder Initialisieren von Zustandsgrößen (z.B. `resetContinuousState(x,5.3)`). Wenn Initialisierungswerte von Ausdrücken abgeleitet werden, die zu berechnen sind, ist eine explizite Zuweisung in der `init()`-Methode nötig.

`terminate():` Die `terminate`-Methode wird am Ende eines Experiments ausgeführt. In der Methode `terminate()` kann Code für die Beendigung eines Blocks angegeben werden, um z. B. ein Herunterfahren des Systems zu modellieren.

`derivatives():` Gewöhnliche Differentialgleichungen (ODE) müssen in der `derivatives()`-Methode spezifiziert werden. Wenn sich die Modellstruktur während der Simulation ändert (z.B. in einem Modell mit bewegten Massen, bei dem Haft- und Gleitreibung simuliert werden soll), kann der Strukturwechsel mit den üblichen Kontrollstrukturen (`if(...) then ... else ...`) gesteuert werden.

`update():` `update()` wird im Raster der externen Kommunikationsschrittweite dT ausgeführt. Werte, die nur in diesem Raster benötigt werden (auch Kommunikation mit der Experimentierumgebung), können in dieser Methode berechnet werden.

`directOutputs():` Die Methode `directOutputs()` beinhaltet alle Ausgabe-Gleichungen mit direktem Durchgang, die direkt von Eingängen abhängen. Da sie direkt von Eingängen abhängen, die wiederum von `nondirectOutputs()` abhängen können, wird diese Methode nach den `nondirectOutputs()` abgearbeitet.

`nondirectOutputs():` Die Methode `nondirectOutputs()` beinhaltet alle Ausgabe-Gleichungen mit nichtdirektem Ausgang (d.h. solche, die nicht direkt von Eingängen abhängen).

`dependentParameters():` Innerhalb der Methode `dependentParameters()` werden Gleichungen für Parameter, die von anderen Parametern abhängen, spezifiziert. Diese Methode wird immer nur dann aufgerufen, wenn ein Parameter während des Simulationsexperimentes geändert wird (asynchroner Aufruf bei Änderung). Dies reduziert die Rechenzeit.

Beispiel: `m_Fahrzeug = m_Leer + m_Zuladung`.

Nur wenn die Zuladung im Experiment geändert wird, wird in der Methode `dependentParameters` die Fahrzeugmasse neu berechnet.

`stateEvents():` Innerhalb der Methode `stateEvents()` ist das Modellieren von zustands- und zeitabhängigen Unstetigkeiten möglich. Diese Methode wird am Ende eines jeden konsistenten Integrationsschrittes ausgewertet. Diskrete Zustandsgleichungen sind in der `stateEvents()`-Methode zu spezifizieren.

`events():` Mit der Methode `events()` können asynchrone Software- und Hardware-Interrupts behandelt werden. Diese Methode wird nicht zeitsynchron, sondern asynchron ausgewertet, wenn das entsprechende Ereignis aufgetreten ist.

9.5 Auswertungsablauf

Während der Ausführung einer Simulation werden die in einem CT-Block enthaltenen Methoden in verschiedenen Zyklen ausgelöst. Es kann zwischen drei allgemeinen Zykluszeiten unterschieden werden:

- Die externe Kommunikationsschrittweite dT
- Die Integrationsschrittweite h
- Die von der internen Integrationsmethode abhängige Schrittweite h/n

Externe Kommunikationsschrittweite **dT**

Die Kommunikationsschrittweite ist nicht Teil des Modells, sondern wird erst bei der Simulation gewählt. Im dT-Zyklus findet die folgende Kommunikation statt:

- Kommunikation zwischen CT-Blöcken und der Experimentierumgebung, z.B. Anregung und Visualisierung.
- Kommunikation zwischen CT-Blöcken und Controller-Modulen innerhalb eines hybriden Projekts.
- Kommunikation zwischen mehreren CT-(Struktur-)Blöcken innerhalb eines hybriden Projekts, wenn mehrere Integrationsverfahren verwendet werden.
- Aufruf der Methode `update()`.

Integrationsschrittweite **h**

Die Integrationsschrittweite ist nicht Teil des Modells, sondern wird erst bei der Simulation gewählt. Im h-Zyklus findet Kommunikation zwischen mehreren zeitkontinuierlichen Blöcken innerhalb eines zeitkontinuierlichen Strukturblocks statt. Nachdem der Integrationsschritt über alle Blöcke durchgeführt worden ist, wird die Methode `stateEvents()` ausgeführt.

Jeder übertragene Wert wird numerisch bestätigt und ist unabhängig von der gewählten Integrationsmethode. Beim Simulieren eines sehr dynamischen Modells, bei dem h sehr klein gewählt werden muss, kann die Geschwindigkeit wesentlich erhöht werden, indem dT wesentlich größer als h gewählt wird.

Schrittweite abhängig von der internen Integrationsmethode - **h/n**

Im Gegensatz zum `h`-Zyklus hängt der `h/n`-Zyklus von der gewählten Integrationsmethode ab, z. B. benutzt die Euler-Integrationsmethode die Zykluszeit h/1, die Heun-Integrationsmethode h/2.

Im h/n-Zyklus werden die Zwischenschritte der Integration berechnet. Wie beim h-Zyklus findet Kommunikation zwischen zeitkontinuierlichen Blöcken eines zeitkontinuierlichen Strukturblocks statt. Die Zwischenschritte der Integration können nicht nach außen kommuniziert werden.

Aus numerischer Sicht können in diesem Zyklus auch keine Unstetigkeiten behandelt werden, da die `stateEvents()`-Methode in diesem Zyklus nicht aufgerufen wird.

Zwischen den unterschiedlichen Schrittweiten gilt folgende Beziehung:

```
dT >= h >= h/n.
```

Der gesamte Zyklus der verschiedenen Methodenaufrufe ist in Abb. 9-3 dargestellt:

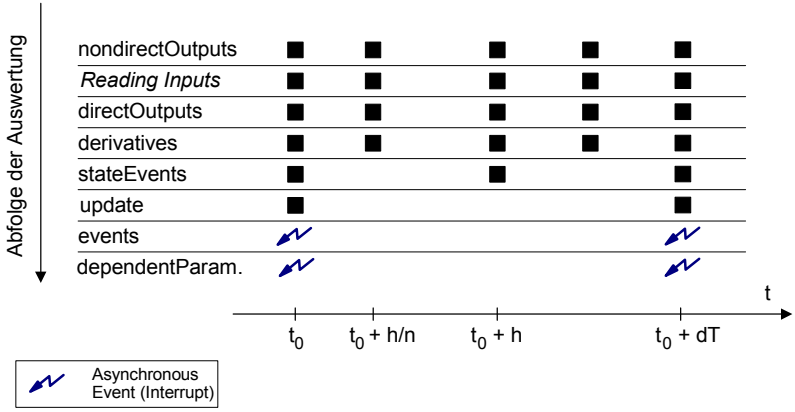


Abb. 9-3 Der Zyklus von Methodenaufrufen in einem zeitkontinuierlichen Block

Die Methoden `events()` und `dependentParameters()` werden nur bei einem expliziten, asynchronen Ereignis ausgeführt und zwar insbesondere nicht innerhalb eines `dT`-Zyklus.

Abb. 9-4 zeigt die Abarbeitungsreihenfolge aller Methoden vom Start bis zum Ende der Simulation.

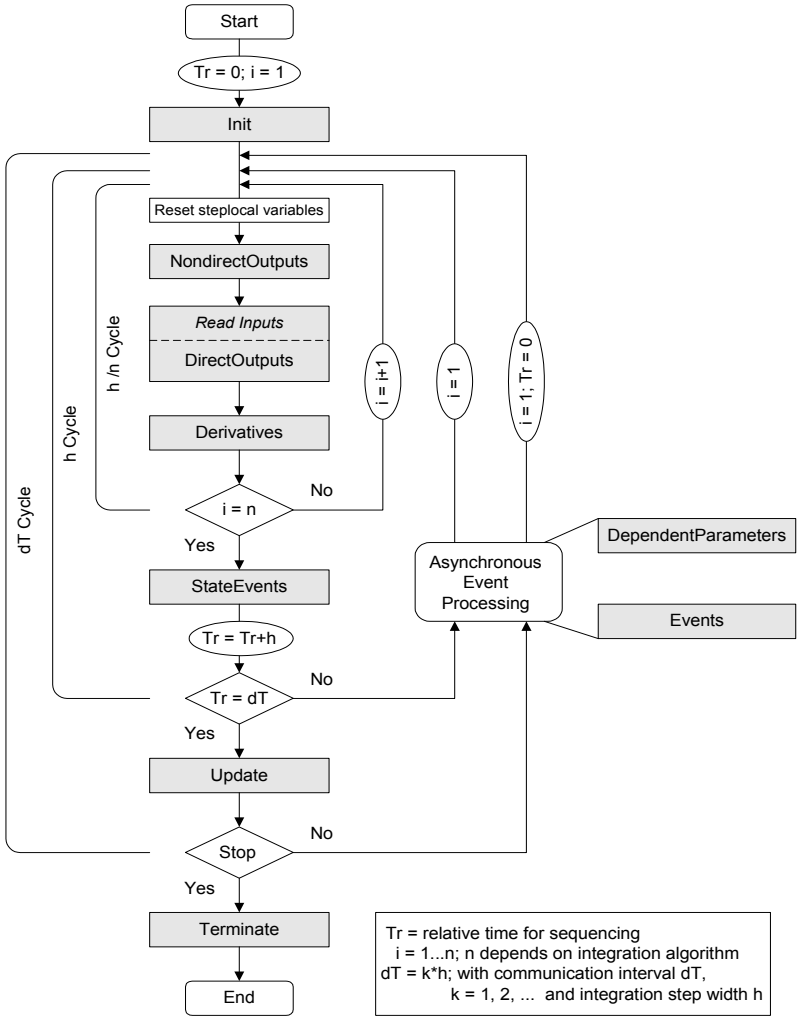


Abb. 9-4 Abarbeitungsreihenfolge der Methoden im CT-Block.

Die Reihenfolge, in der die Methoden eines Basisblocks ausgeführt werden, wird durch die folgenden Beispiele veranschaulicht.

Der Auswertungsablauf für synchrone Aufrufe, z. B. wenn `n = 1` (Euler) und `h = dT` ist:

- zum Zeitpunkt t = *dT*: `nondirectOutputs` - (Lesen der Eingänge) - `directOutputs` - `derivatives`
- zum Zeitpunkt t = *dT*: `stateEvents`
- zum Zeitpunkt t = *dT*: `update`

Bei einer komplexeren Integrationsmethode, z. B. wenn *n* = 2 (Adams-Moulton) und *h* = *dT* gilt, ist der Ablauf:

- zum Zeitpunkt t = *dT*/2: `nondirectOutputs` - (Lesen der Eingänge) - `directOutputs` - `derivatives`
- zum Zeitpunkt t = *dT*: `nondirectOutputs` - (Lesen der Eingänge) - `directOutputs` - `derivatives`
- zum Zeitpunkt t = *dT*: `stateEvents`
- zum Zeitpunkt t = *dT*: `update`

Der Auswertungsablauf bei *n* = 1 und *h* = *dT*/2 ist:

- zum Zeitpunkt t = *dT*/2: `nondirectOutputs` - (Lesen der Eingänge) - `directOutputs` - `derivatives`
- zum Zeitpunkt t = *dT*/2: `stateEvents`
- zum Zeitpunkt t = *dT*: `nondirectOutputs` - (Lesen der Eingänge) - `directOutputs` - `derivatives`
- zum Zeitpunkt t = *dT*: `stateEvents`
- zum Zeitpunkt t = *dT*: `update`

Das Verständnis des Auswertungsablaufs und damit des Verhaltens zeitkontinuierlicher Basisblöcke ist für eine korrekte Verwendung solcher Blöcke unbedingt erforderlich. Beim Benutzen von ESDL zur Spezifizierung stellt eine automatische Analysephase ein konsistentes Modellieren bei der Kopplung mehrerer CT-Blöcke sicher. Die Berechnungsreihenfolge ist insbesondere bei Blöcken mit direktem Ausgang (directOutputs) wichtig, da an den entsprechenden Eingängen gültige Werte anliegen müssen, die aus dem gleichen Iterationszyklus stammen.

9.6 Modellieren mit ESDL

Zur Spezifizierung von zeitkontinuierlichen Basisblöcken steht der gesamte Sprachumfang von ESDL zur Verfügung. Außerdem werden eine Semantiküberprüfung und einige weitere Bibliotheksfunktionen zur Beschreibung von Differentialgleichungen angeboten. Diese werden im folgenden erläutert.

9.6.1 Differentialgleichungen in ESDL

In ESDL unterstützt jede kontinuierliche Zustandsvariable den Ableitungsoperator `ddt`. Differentialgleichungen können mit dem Operator `ddt` beschrieben werden.

Ein Beispiel sei ein PT2-System mit den kontinuierlichen Zustandsvariablen `x` und `xp`, der Eingabe `in` und den Parametern `d`, `T`, `K`. Die mathematische Formulierung des Systems sieht so aus:

```
x' = xp;
xp' = (K*in - (2.0*d*T*xp) - x) / (T*T);
```

Beim Modellieren dieses PT2-Systems mit ESDL werden die Ableitungen mittels der `ddt`-Methode spezifiziert:

```
x.ddt(xp);
xp.ddt( (K*in - (2.0*d*T*x.ddt()) - x) / (T*T) );
```

Auf die Ableitungen auf der linken Seite einer Differentialgleichung (d.h. im Argument einer Ableitungsmethode) kann nicht zugegriffen werden. Wenn ein Zugriff erforderlich ist, muss das System umformuliert werden.

Der `ddt`-Operator kann nur in der `derivatives()`-Methode verwendet werden.

9.6.2 Semantische Überprüfungen in ESDL

Beim Benutzen von ESDL innerhalb einer zeitkontinuierlichen Methode ist ein semantisches Überprüfen möglich. Die Überprüfungspunkte unterstützen die Übereinstimmung eines Modells mit dem grundlegenden zeitkontinuierlichen Simulationsrahmen. Beispielsweise ist das direkte Ändern des Wertes einer Zustandsvariablen nicht erlaubt (statt dessen muss die `Funktion resetContinuousState()` mit einem internen Rücksetzen des Integrationsalgorithmus verwendet werden). Einen Überblick über Zugriffsrechte auf Elemente gibt Abb. 9-5. Jede Verletzung dieser Rechte wird bei einer semantischen Überprüfung abgefangen.

Methode \ Element	input	output	discrete state	continuous state	ddt-operator	steplocal variable	local variable	parameter	dependent parameter	constant
init	r -	rw	rw	r -	- -	rw	rw	r -	r -	r -
derivatives	r -	- -	r -	r -	rw	rw	rw	r -	r -	r -
update	r -	- -	rw	- -	- -	rw	rw	r -	r -	r -
directOutputs	r -	-w	r -	r -	- -	rw	rw	r -	r -	r -
nondirectOutputs	r -	-w	r -	r -	- -	rw	rw	r -	r -	r -
terminate	r -	-w	rw	r -	- -	rw	rw	r -	r -	r -
events	r -	-w	rw	r -	- -	rw	rw	r -	r -	r -
dependentParameters	- -	- -	- -	- -	- -	- -	rw	r -	rw	r -
stateEvents	r -	- -	rw	r -	- -	rw	rw	r -	r -	r -

r = lesen
w = schreiben

Abb. 9-5 Zugriffsrechte auf Elemente

Der Ableitungsoperator `ddt` unterstützt nur die erste Ableitung. Die Ausgabe-Gleichungen der `nondirectOutputs()`-Methode werden analysiert, um eine direkte Abhängigkeit von einer Eingabe festzustellen. In einem solchen Fall wird eine Warnung ausgegeben.

9.6.3 Zusätzliche Bibliotheksfunktionen

Zum fortgeschrittenen zeitkontinuierlichen Modellieren mit ESDL bietet die Systembibliothek eine zusätzliche Menge von Bibliotheksfunktionen an:

- `getTime()`
- `getdT()`
- `getIntegrationStepsize()`
- `resetContinuousState()`
- `resetCTSolver()`

Im folgenden wird der Gebrauch der Bibliotheksfunktionen näher erläutert. Der Zugriff auf die Funktionen in den verschiedenen Methoden wird in Abb. 9-6 gezeigt.

Methode \ Zusätzliche Bibliotheksfunktionen	getTime	getdT	getIntegrationStepsize	resetContinuousState	resetCTSolver
init	+	+	+	+	+
derivatives	+	+	+	-	-
update	+	+	+	+	+
directOutputs	+	+	+	-	-
nondirectOutputs	+	+	+	-	-
terminate	+	+	+	-	-
events	+	+	+	-	-
dependentParameters	+	+	+	-	-
stateEvents	+	+	+	+	+

+ vorhanden

- nicht vorhanden

Abb. 9-6 Zugriff auf Funktionen in den Methoden eines zeitkontinuierlichen Blocks

`getTime( ):` In einigen Fällen ist die aktuelle Simulationszeit wichtig. In Online-Experimenten ist dies die tatsächlich vergangene Zeit. Diesen Wert kann man über die Bibliotheksfunktion `getTime` erhalten:

```
t = getTime ( );
```

Die `getTime`-Funktion kann in jeder Methode verwendet werden.

`getdT( ):` Mit der Bibliotheksfunktion `getdT` wird die aktuelle Schrittweite für externe Kommunikation zur Verfügung gestellt:

```
step = getdT ( );
```

`getIntegrationStepsize( ):` Die Bibliotheksfunktion `getIntegrationStepsize()` gibt die aktuelle Integrationsschrittweite zurück:

```
h = getIntegrationStepsize ( );
```

`resetContinuousState( state, new value):` Beim Modellieren von zeit- oder zustandsabhängigen Unstetigkeiten ist oft ein Zurücksetzen der kontinuierlichen Zustandsvariable nötig. Um eine korrekte numerische Auswertung zu gewährleisten, muss die Integrationsmethode intern neu initialisiert werden. Dies erfolgt mit der `resetContinuousState`-Funktion:

```
resetContinuousState (x, 0.0 );
```

In diesem Falle wird der Zustand `x` auf `0.0` gesetzt und nötigenfalls die Integrationsmethode neu initialisiert. Die Verwendung der Bibliotheksfunktion `resetContinuousState` ist nur in den Methoden `init` und `stateEvents` erlaubt und sinnvoll. In der Methode `update` ist die Verwendung zwar erlaubt, aber sinnlos, da die Methode keinen Lesezugriff auf kontinuierliche Zustände hat. Nach `resetContinuousState(x,y)` wird automatisch `resetCTSolver()` ausgeführt.

`resetCTSolver( ):` Das explizite Rücksetzen der Integrationsmethode ist mit `resetCTSolver` möglich:

```
resetCTSolver ( );
```

Die Verwendung der Bibliotheksfunktion `resetCTSolver` ist nur in den Methoden `init`, `update` und `stateEvents` gestattet. Nach `resetContinuousState(x,y)` wird `resetCTSolver()` automatisch ausgeführt.

9.7 Modellieren in C

Modellieren mit C bietet alle Möglichkeiten der Sprache C, jedoch keine Semantiküberprüfung. In C spezifizierte zeitkontinuierliche Basisblöcke können hardwareabhängig sein. Wird mit ANSI-C programmiert, kann auch in C hardwareunabhängig modelliert werden. Dies ist notwendig, wenn Zeiger oder C-Unterprogramme verwendet werden sollen. C-Basisblöcke können zum Modellieren hardwareabhängiger Blöcke verwendet werden, sind jedoch auch wie ESDL-Basisblöcke zu verwenden. Bei C-Basisblöcken muss durch Auswahl von `direct` oder `nondirect` im *„Block Behavior"*-Kombikästchen explizit festgelegt werden, ob es sich um einen Block mit direktem Durchgang

(Ausgang hängt direkt vom Eingang ab) oder nichtdirektem Durchgang handelt. Dadurch wird die automatische Festlegung der Bearbeitungsreihenfolge beeinflusst.

Hinweis

Beim Modellieren in C gibt es keine semantischen Überprüfungen, die konsistentes Modellieren sicherstellen (wie bei ESDL). Konsistenz muss durch den Benutzer gewährleistet werden.
Es wird empfohlen, C zum Modellieren zeitkontinuierlicher Systeme nur zu verwenden, wenn dies absolut notwendig ist, z. B. zum Modellieren controllerabhängiger Systemteile oder wenn C-Zeiger oder C-Unterprogramme verwendet werden müssen.

9.7.1 Differentialgleichungen in C

In C wird für jede kontinuierliche Zustandsvariable eine interne Ableitungsvariable erzeugt. Der Name dieser Variablen entsteht aus dem Zustandsvariablennamen mit dem Präfix `ddt`.

Beispiele seien die kontinuierlichen Zustandsvariablen `x` und `xp`; die automatisch erzeugten Ableitungsvariablen lauten `ddtx` und `ddtxp`. Sie sind in allen Methoden sichtbar.

Ein vollständiges Beispiel ist ein PT2-System mit den kontinuierlichen Zustandsvariablen `x` und `xp`, der Eingabe `in` und den Parametern `d`, `T`, `K`.

```
x' = xp;
xp' = (K*in - (2.0*d*T*xp) - x) / (T*T);
```

Das obige PT2-System lässt sich im CT-Block als C-Code wie folgt formulieren:

```
ddtx = xp;
ddtxp = (K*in - (2.0*d*T*ddtx) - x) / (T*T);
```

9.7.2 Weitere C-Routinen

Beim Modellieren mit C sind zusätzliche C-Routinen verfügbar. Zur generischen Verwendung muss die interne Datenstruktur des aktuellen Blocks in der Routinenschnittstelle stehen. In jeder Methode sind die Variablen `CTBlock` und `self` sichtbar.

Es gibt die folgenden Routinen:

- `getTime`
- `getdT`
- `getIntegrationStepsize`
- `resetCTSolver`

- sizeU
- sizeY
- sizeV
- sizeX
- sizeXK

Mit den Routinen `get` und `reset` stehen zusätzliche ESDL-Bibliotheksroutinen zur Verfügung; die Routine `size` sichert ein generisches Modellieren, wenn die Anzahl oder Arraylänge von Instanzenvariablen geändert werden muss.

Im folgenden wird die Verwendung der zusätzlichen C-Routinen genauer erklärt. Es gibt keine semantische Überprüfung und keine Nutzungsbeschränkungen bei diesen Routinen. Der Benutzer muss hier deren korrekte Verwendung selbst sicherstellen.

`real64 getTime(CTSimExperiment *):`

Die Funktion `getTime` liefert die aktuelle Simulationszeit:

```
t = getTime (CTBlock);
```

`real64 getdT ():`

Die Funktion `getdT` liefert die aktuelle Schrittweite für externe Kommunikation:

```
step = getdT ();
```

`real64 getIntegrationStepsize(CTSimExperiment *):`

Die Funktion `getIntegrationStepsize` liefert die aktuelle Integrationsschrittweite:

```
h = getIntegrationStepsize (CTBlock);
```

`void resetCTSolver(CTSimExperiment *):`

Ein explizites Zurücksetzen des Integrationsalgorithmus ist mit der Routine `resetCTSolver` möglich. Ein Beispiel ist die Verwendung nach dem Rücksetzen eines zeitkontinuierlichen Zustands:

```
x = 0.0;
resetCTSolver (CTBlock);
```

Nach jedem expliziten Setzen einer oder mehrerer zeitkontinuierlicher Zustände ist ein abschließendes Rücksetzen der internen Strukturen nötig. Es ist zu beachten, dass das Kommando `resetCTSolver` immer aufgerufen werden sollte, nachdem einem zeitkontinuierlichen Zustand ein Wert zugewiesen wurde.

int_32 sizeU (CTSimExperiment *):

Die Funktion `sizeU` gibt die Anzahl der Blockeingaben zurück:

```
sizeU = sizeU (CTBlock);
```

Wenn einige Eingänge Arrays sind, wird die Gesamtzahl der skalaren Elemente zurückgegeben. Komplexere Eingänge, z. B. Records, Strukturen oder Klassen zählen als ein Element.

int_32 sizeY (CTSimExperiment *):

Die Funktion `sizeY` gibt die Anzahl der Blockausgaben zurück:

```
sizeY = sizeY (CTBlock);
```

Wenn einige Ausgänge Arrays sind, wird die Gesamtzahl der skalaren Elemente zurückgegeben. Komplexere Ausgänge, z. B. Records, Strukturen oder Klassen zählen als ein Element.

int_32 sizeV (CTSimExperiment *):

Die Funktion `sizeV` liefert die Anzahl an Blockparametern (Parameter und abhängige Parameter):

```
sizeV = sizeV (CTBlock);
```

Wenn einige Parameterzustände Arrays sind, wird die Gesamtzahl der skalaren Elemente zurückgegeben.

int_32 sizeX (CTSimExperiment *): Die Funktion `sizeX` gibt die Anzahl der kontinuierlichen Zustände zurück:

```
sizeX = sizeX (CTBlock);
```

Wenn einige kontinuierliche Zustände Arrays sind, wird die Gesamtzahl der skalaren Elemente zurückgegeben.

int_32 sizeXK (CTSimExperiment *):

Die Funktion `sizeXK` gibt die Anzahl der diskreten Zustände zurück:

```
nofX = sizeXK (CTBlock);
```

Wenn einige kontinuierliche Zustände Arrays sind, wird die Gesamtzahl der skalaren Elemente zurückgegeben.

10 Zeitkontinuierliche Strukturblöcke und grafische Hierarchien

Mit zeitkontinuierlichen Strukturblöcken (CT-Strukturblöcke) können komplexe Modelle grafisch aufgebaut werden, indem andere CT-Struktur- und CT-Basisblöcke in einem Blockdiagramm zu einem neuen CT-Strukturblock zusammengesetzt und verbunden werden. Zum Spezifizieren zeitkontinuierlicher Strukturblöcke wird ein leicht modifizierter Blockdiagrammeditor (BDE) verwendet. (siehe auch Abb. 8-2 auf Seite 187). Die entsprechenden Ein- und Ausgänge werden im BDE grafisch miteinander verbunden.

Ein zeitkontinuierlicher Strukturblock wird als Blockdiagramm mit einer festen Anzahl von Methoden modelliert. Im Prinzip werden die Methoden der CT-Basisblöcke automatisch übernommen und können im BDE nicht mehr geändert werden. Die funktionale Beschreibung ist an ein einziges Diagramm gebunden. Der korrekte Auswertungsablauf bzw. -reihenfolge wird ebenfalls automatisch ermittelt und kann nicht direkt beeinflusst werden.

Ein einfaches Beispiel, das die Anwendung von CT-Basisblöcken, ihrer Methoden und CT-Strukturblöcken bis hin zur Simulation in der Experimentierumgebung veranschaulicht, finden Sie im Tutorial im ASCET Schnelleinstieg, Kapitel „Modellierung eines zeitkontinuierlichen Systems“.

10.1 Strukturblöcke wiederverwenden

CT-Strukturblöcke werden wie CT-Basisblöcke in der aktuellen Datenbank gespeichert und stehen damit wieder anderen CT-Strukturblöcken zur Verfügung. So kann eine Modellbibliothek für eine modulare und hierarchische Modellstruktur aufgebaut werden. Wird ein CT-Basisblock oder CT-Strukturblock in der Datenbank geändert, wird die Änderung in allen Modellen der Datenbank automatisch übernommen. Die CT-Blöcke müssen also nur an einer Stelle gepflegt werden.

Abgeschlossene Modelle, bei denen die verwendeten CT-Blöcke durch neuere Versionen nicht ersetzt werden sollen, müssen in einer eigenen Datenbank gespeichert werden.

10.2 Elemente eines zeitkontinuierlichen Strukturblocks

In zeitkontinuierlichen Strukturblöcken sind nicht alle Variablen erforderlich, die in Basisblöcken verwendet werden. Die folgenden Elemente sind vorhanden:

- Eingaben (Inputs)
- Ausgaben (Outputs)
- Globale Parameter (Global parameters)

- Konstanten (Constants)
- Kennfelder (OneD and TwoD table parameters)

Für jeden Elementtyp gibt es verschiedene Dimensionen, Bereiche und Datentypen.

Es steht ein Additions- und ein Subtraktionsoperator zur Verfügung, bei denen die Anzahl der Eingänge jeweils individuell gewählt werden können.

10.3 Blockschnittstellen

In den folgenden Abschnitten werden die in zeitkontinuierlichen Strukturblöcken verfügbaren Elemente beschrieben.

Eingänge: Blockeingaben werden durch Eingänge beschrieben. In jedem Auswertungsschritt werden alle Eingabevariablen gelesen.

Ausgänge: Blockausgaben werden durch Ausgänge beschrieben. In jedem Auswertungsschritt werden alle Ausgabevariablen aktualisiert.

Globale Parameter: Globale Parameter werden zur Beschreibung von Parametern verwendet, die im gesamten Modell sichtbar sind. Normalerweise entspricht ein globaler Parameter einer globalen charakteristischen Eigenschaft des realen Systems. Durch effiziente Nutzung von globalen Parametern kann eine Reduzierung der Komplexität und des Wartungsaufwandes erzielt werden.

Konstanten: Konstanten werden für Größen benutzt, die sich während eines Experimentes nicht ändern, z. B. die Gravitationskonstante.

Dimensionen, Geltungsbereich und Typ: Jede Art von Element hat eine bestimmte Dimension, einen Geltungsbereich und einen Typ. Die möglichen Kombinationen werden in der folgenden Tabelle veranschaulicht.

Kombinationen / Elemente	Dimension			Geltungsbereich		Datentyp			
	scalar	array	record	local	global	logic	sdisc	udisc	cont
Eingang	x	x	x	x		x	x	x	x
Ausgang	x	x	x	x		x	x	x	x
Globaler Parameter	x	x			x	x	x	x	x
Konstanten	x	x		x	x	x	x	x	x

Abb. 10-1 Dimension, Geltungsbereich und Datentyp der Elemente

10.4 Operatoren

Aus Sicht der Systemtheorie sind zur Beschreibung von Strukturblöcken nur lineare Operatoren erforderlich. Nichtlineare Elemente werden in Basisblöcke eingeschlossen. Somit sind nur der Additions- und der Subtraktionsoperator verfügbar.

10.5 Algebraische Schleifen

Im folgenden Gleichungssystem

$x = f_1(z)$
$y = f_2(x)$
$z = f_3(\texttt{Input a})$ (`Input a` sei gültig)

hängt - außer f_3 - jede Gleichung von einer anderen ab. Damit das System von oben nach unten korrekt berechnet werden kann, müssen die Gleichungen umgestellt werden:

$z = f_3(\texttt{Input a})$
$x = f_1(z)$
$y = f_2(x)$

In dieser Reihenfolge kann das System - auch von üblichen PC-Programmen - problemlos berechnet werden.

Eine algebraische Schleife liegt dann vor, wenn gilt:

$y=f_1(x);$
$x=f_2(y);$

d.h. wenn zwei Funktionen direkt voneinander abhängen. Um x zu berechnen, wird y benötigt und um y zu berechnen, wird x benötigt.

10.6 Direkter und nichtdirekter Ausgang

ASCET sortiert CT-Blöcke bzw. Methoden in gekoppelten CT-Blöcken, die direkt voneinander abhängen, automatisch in die richtigen Reihenfolge (automatische Sequenzierung). Tritt im Modell eine algebraische Schleife auf, bricht ASCET bei der Bestimmung der Berechnungsreihenfolge mit einer entsprechenden Fehlermeldung ab. Dies passiert beispielsweise, wenn zwei oder mehrere CT-Blöcke mit direkten Ausgängen eine Rückkopplungsschleife bilden.

Damit die Berechnungsreihenfolge kontrolliert und automatisch bestimmt werden kann, müssen die Ausgänge entsprechend ihrer Eigenschaft spezifiziert werden. Ausgänge, die direkt von Eingängen abhängen, müssen in der Methode `directOutputs` spezifiziert bzw. geschrieben werden. Ein derartiger CT-Basisblock hat einen *direkten Ausgang* oder einen *direkten Durchgriff*.

Dagegen werden Ausgänge, die nicht direkt von Eingängen abhängen, in der Methode `nondirectOutputs` spezifiziert. Der CT-Basisblock hat einen *nichtdirekten Ausgang* oder einen *nichtdirekten Durchgriff*.

Falsch deklarierte Ausgaben werden erkannt (z.B. direkte Ausgabe in die Methode `nondirectOutputs`), wenn die Modellierungssprache ESDL verwendet wird. Bei CT-Blöcken, die in der Programmiersprache C geschrieben sind, wird die Eigenschaft *nondirect* oder *direct* vom Modellentwickler festgelegt.

Die Methoden `nondirectOutputs` und `directOutputs` bestimmen wesentlich das Verhalten der CT-Basisblöcke und die Abarbeitungsreihenfolge in einem CT-Strukturblock. Im folgenden Beispiel wird dies noch einmal erläutert.

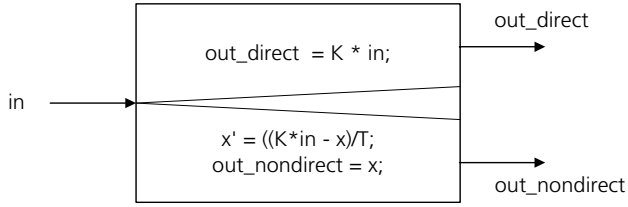


Abb. 10-2 Beispiel: Direkter und nichtdirekter Ausgang

Im allgemeinen hat eine Ausgang ein direktes Durchgangsverhalten, wenn er unmittelbar von einem Eingang abhängt. Beispielsweise wird ein Verstärkerblock (P-Verhalten) durch die Funktion

```
out = K * in
```

beschrieben. Der Ausgang hängt direkt vom Eingang ab. Folglich müssen zuerst die Eingaben gelesen wurden und dann der Ausgang berechnet werden. Die Funktion muss in die Methode `directOutputs` geschrieben werden.

Wenn ein Ausgang nicht von einem Eingang abhängt, wenn z. B. der Ausgang von einem kontinuierlichen Zustand oder einer Parameterbedingung abhängt, hat dieser Ausgang kein direktes Durchgriffsverhalten. Nichtdirekte Ausgänge werden aus den Werten des vorherigen Schrittes berechnet. Ein CT-Block mit direktem Ausgang bricht eine bestehende Schleife ab. Ein Beispiel ist das sogenannte PT_1-Verhalten:

```
x' = ((K*in - x)/T);
out = x;
```

Die Differentialgleichung wird mit einem Integrationsverfahren gelöst, das zur Berechnung des aktuellen Ausgabewertes `x` den letzten Ausgabewert und Eingangswert `in` benötigt. Die Zuweisung `out=x` muss in die Methode `nondirectOutputs` geschrieben werden (die Differentialgleichung wird in der Methode `derivatives` behandelt).

Zwei einfache Beispiele zeigen eine richtige und falsche Kopplung zweier CT-Basisblöcke mit direkten und nichtdirekten Ausgang innerhalb eines CT-Strukturblocks. Wesentlich dabei ist, dass ein direkter Ausgang die Eingangsdaten des aktuellen Zeitschritts benötigt. Ein nichtdirekter Ausgang kann ohne die Eingangsinformation des aktuellen Zeitschritts berechnet und gesendet werden. Daher werden die direkten Ausgaben nach den nichtdirekten berechnet.

Abb. 10-3 zeigt eine Kombination aus zwei CT-Blöcken mit direktem und nichtdirektem Durchgangsverhalten, die nicht zu einer algebraischen Schleife führt.

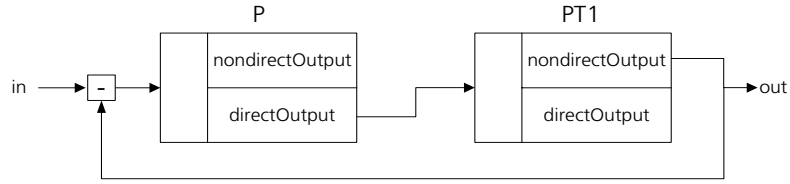


Abb. 10-3 Schleife aus CT-Blöcken mit direktem und nichtdirektem Ausgang

Der P-Block benötigt zur Berechnung einen gültigen Eingangswert. Also muss zuerst im PT_1-Block die Methode `nondirectOutputs` berechnet werden und danach `directOutputs` im P-Block.

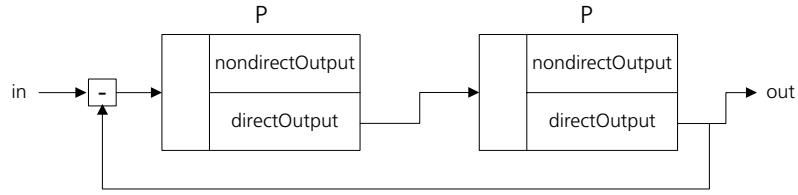


Abb. 10-4 Algebraische Schleife

In Abb. 10-4 sind zwei CT-Blöcke mit direkten Ausgang hintereinander gekoppelt. Dies führt zu einer algebraischen Schleife. Jeder Block benötigt den gültigen Ausgangswert des anderen Blocks. ASCET meldet diesen Fehler.

Direkte Durchgangsschleifen müssen vermieden werden. Eine automatische, implizite Auflösung von algebraischen Schleifen ist aber nicht möglich, da eine implizite Auflösung eine iterative Methode voraussetzt, was unter Echtzeit-Bedingungen nicht akzeptabel ist.

Ein Vorteil gegenüber der automatischen Auftrennung einer algebraischen Schleife ist außerdem, dass der Anwender, der sein Modell kennt, an der geeignetesten Stelle einen Block ohne direkten Ausgang einfügen kann, wodurch die nachfolgenden Blöcke im nächsten Iterationsschritt berechnet werden.

Prinzipiell gibt es zwei Möglichkeiten algebraische Schleifen zu vermeiden:

1. Einfügen eines Blocks ohne direkten Ausgang. Da dies einem Speicherelement entspricht, muss, damit die Dynamik des Modells nicht zu stark verfälscht wird, gegebenenfalls die Integrationsschrittweite verkleinert werden.
2. Änderung des Modellaufbaus, so dass eine algebraische Schleife nicht mehr auftritt. Umformulieren der Gleichungen in den CT-Basisblöcken, ändern der Struktur.

10.7 Unterschied: Grafische Hierarchie - CT-Strukturblock

CT-Strukturblöcke verhalten sich nach aussen bezüglich der Abarbeitungsreihenfolge wie CT-Basisblöcke. Im CT-Strukturblock wird die Abarbeitungsreihenfolge festgelegt. Je nachdem, ob Ausgänge des Strukturblocks direkt oder nichtdirekt von Eingängen abhängen, verhält sich die Struktur wie ein Block mit direktem oder nichtdirektem Ausgang.

Hierarchien dagegen sind rein symbolischer Natur, um einen CT-Strukturblock übersichtlicher zu gestalten und haben keinen Einfluss auf die Simulation. Abb. 10-5 (linke Hälfte) zeigt ein Beispiel, in dem ein CT-Strukturblock einen direkten Ausgang in einen CT-Basisblock und dieser CT-Basisblock gleichzeitig einen direkten Ausgang in den selben CT-Strukturblock hat. Dies führt zu einer algebraischen Schleife, weil die beiden Blöcke innerhalb des Strukturblocks

direkt nacheinander (praktisch gleichzeitig) ausgewertet werden. Der zweite CT-Block innerhalb des Strukturblocks benötigt aber einen gültigen Ausgang des externen CT-Blocks.

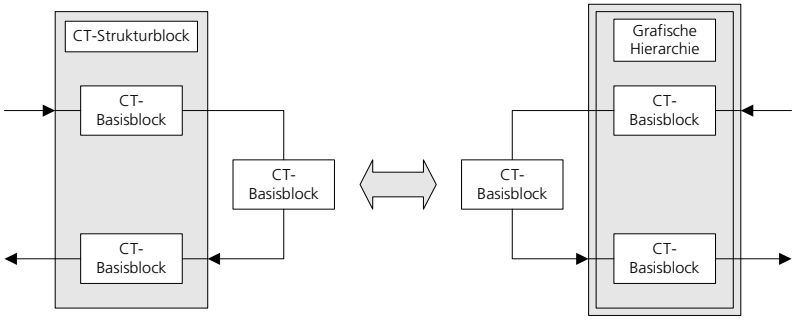


Abb. 10-5 Strukturieren mit CT-Strukturblock oder mit grafischer Hierarchie

Diese algebraische Schleife kann dadurch vermieden werden, dass der Strukturblock aufgelöst und durch eine grafische Hierarchie ersetzt wird (Abb. 10-5, rechte Hälfte), die offensichtlich zusammengehörige Modellteile zusammenfaßt. Nachteil der Hierarchie ist, dass sie nicht getrennt abgespeichert werden kann, sondern nur zusammen mit dem Strukturblock, in dem sie sich befindet.

10.8 Abarbeitungsreihenfolge der Methoden innerhalb einer Struktur

Die Abarbeitungsreihenfolge in einem CT-Strukturblock wird wesentlich von der Abarbeitungsreihenfolge der Methoden innerhalb eines CT-Basisblocks bestimmt, die im Kapitel 9.5 beschrieben wurde (siehe Abb. 9-4 auf Seite 205). Sie hängt hauptsächlich vom Integrationsverfahren und den gewählten Zeit- bzw. Kommunikationsintervallen ab.

Prinzipiell werden die Methoden im Strukturblock in der gleichen Reihenfolge wie im Basisblock berechnet (`init`, `nondirectOutputs`, `directOutputs`,...), wobei immer die gleiche Methode in allen Basisblöcken des Strukturblocks ausgeführt wird, bevor zur nächsten Methode gewechselt wird. Es wird also in allen Blöcken zuerst die Methode `init` ausgeführt, bevor in irgendeinem Basisblock mit der Methode `nondirectOutputs` begonnen wird.

Solange in keinem CT-Basisblock die Methode `directOutputs` verwendet wird, wird die Reihenfolge ausschließlich von den CT-Basisblöcken bestimmt. Dabei ist die Reihenfolge beliebig, in der die gleiche Methode der einzelnen CT-Basisblöcke berechnet wird. Es werden also zuerst alle Methoden `init`, dann alle Methoden `nondirectOutputs` usw. in beliebiger Reihenfolge abgearbeitet.

Wird die Methode `directOutputs` in mehreren Blöcken verwendet, spielt die Berechnungsreihenfolge eine wichtige Rolle, da an den entsprechenden Eingänge der Methoden `directOutputs` gültige Werte von anderen Ausgängen anliegen müssen. Ist der Eingang an einen Ausgang der Methode `nondirectOutputs` angeschlossen, liegt immer ein gültiger Wert vor, da diese Methode in allen CT-Blöcken berechnet ist, bevor die Methode `directOutputs` an der Reihe ist. Hängt der Eingang dagegen von dem Ausgang einer anderen Methode `directOutputs` ab, muss diese schon vorher berechnet worden sein.

Beispiel Abarbeitungsreihenfolge

Die Abb. 10-6 zeigt die Abarbeitungsreihenfolge in einem kleinen CT-Strukturblock mit gekoppelten CT-Basisblöcken. `readInputs` ist keine eigene Methode, sondern gehört zu `directOutputs`, und soll verdeutlichen, dass zum Auswerten der `directOutputs` gültige Werte gelesen werden müssen. Dementsprechend wird die Abarbeitungsreihenfolge vom automatischen Sequenzierungsalgorithmus festgelegt. Die Nummern geben die Reihenfolge an. Mehrfach vergebene Nummern bedeuten, dass die Ausführungsreihenfolge beliebig ist.

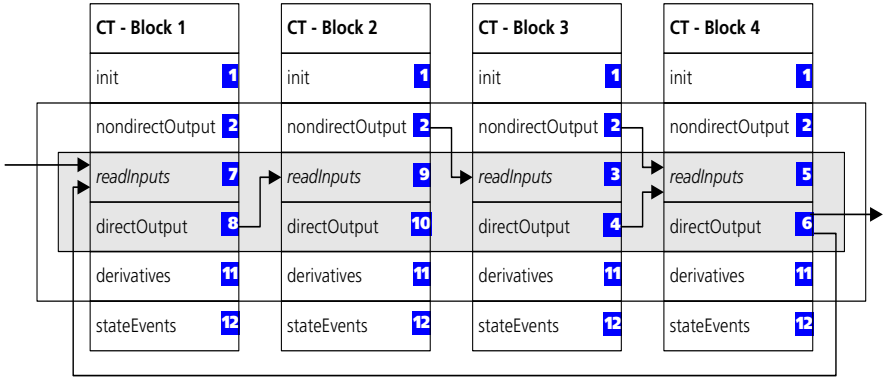


Abb. 10-6 Berechnungsreihenfolge der Methoden bei gekoppelten CT-Blöcken

Die Berechnungsreihenfolge ist insbesondere bei CT-Blöcken mit direkten Ausgängen (Methode `directOutputs`) wichtig, da an den entsprechenden Eingängen gültige Werte anliegen müssen, die aus dem gleichen Iterationszyklus stammen (dunkel hinterlegter Bereich in Abb. 10-6).

Die CT-Blöcke werden von oben nach unten abgearbeitet. Außerdem wird immer eine Methode nach der anderen abgearbeitet. `init` wird nur einmal beim Start der Simulation berechnet.

Innerhalb der Integrationsschleife (Methoden `nondirectOutputs` bis `derivatives`) werden immer alle `nondirectOutputs` berechnet. Die Reihenfolge liegt nicht fest. Da die Methode `directOutputs` direkt vom jeweiligen Eingang abhängt, sucht ASCET solange die Methoden `directOutputs` ab, bis der entsprechende `readInputs` nicht mehr von einer anderen Methode `directOutputs` abhängt (dunkel unterlegter Bereich in Abb. 10-6). Dies ist in Abb. 10-6 im CT -Basisblock 3 der Fall. Daraus ergibt sich für das Lesen der Eingänge und Abarbeitung der Methode `directOutputs` folgende Reihenfolge:

1. `readInputs` (CT - Block 3), `directOutputs` (CT - Block 3)
2. `readInputs` (CT - Block 4), `directOutputs` (CT - Block 4)
3. `readInputs` (CT - Block 1), `directOutputs` (CT - Block 1)
4. `readInputs` (CT - Block 2), `directOutputs` (CT - Block 2)

Danach werden die Methoden `derivatives` 1-4 in beliebiger Reihenfolge abgearbeitet. Bei einem einstufigen Integrationsverfahren folgen jetzt die Methoden `stateEvents` jeweils von CT - Block 1-4 in beliebiger Reihenfolge. Danach wieder zurück zu `nondirectOutputs`.

Bei n-stufigen Integrationsverfahren wird n-mal `nondirectOutputs` - `directOutputs` (wie oben beschrieben in der richtigen Reihenfolge) und `derivatives` von CT-Block 1 - 4 abgearbeitet, bevor die `stateEvents` ausgeführt werden (siehe auch Abb. 9-4 auf Seite 205).

Die Kommunikation bei kombinierten CT-Basisblöcken und/oder CT-Strukturblöcken innerhalb einer Struktur erfolgt also auch bei den Zwischenschritten des Integrationsverfahrens. Es werden jedesmal die Methoden `nondirectOutputs` bis `derivatives` ausgewertet (einfach eingerahmt).

Die Methode `update` wird nach `stateEvents` nur im Raster des Kommunikationsintervalls `dT` abgearbeitet und `terminate` nur am Ende der Simulation. Bei beiden ist die Berechnungsreihenfolge innerhalb des Strukturblocks wieder beliebig.

Typischerweise gibt es also mehrere äquivalente Abarbeitungsreihenfolgen, um einen Strukturblock zu lösen. Der Sequenzierungsalgorithmus von ASCET wählt automatisch eine Reihenfolge aus.

Beispiel: Abarbeitung nicht möglich

Liegt eine algebraische Schleife vor, kann die Berechnungsreihenfolge nicht automatisch bestimmt werden. Diese Situation ist in Abb. 10-7 gezeigt. Jeder Eingang eines `directOutputs` hängt von einem anderen `directOutputs` ab und die Schleife ist von CT-Block 4 nach CT-Block 1 geschlossen. Es kommt zu einer entsprechenden Fehlermeldung.

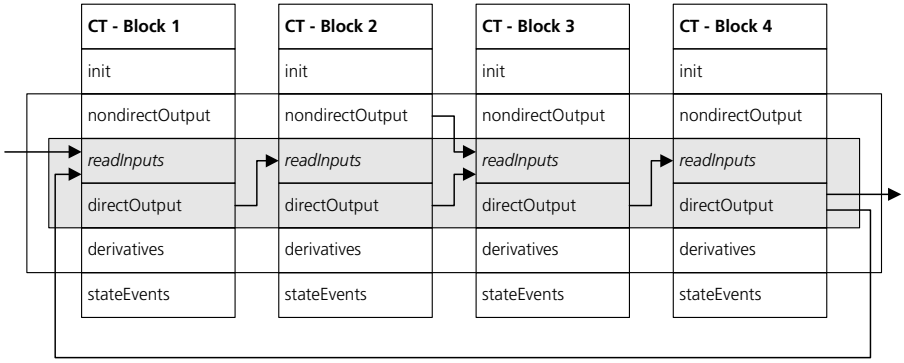


Abb. 10-7 Algebraische Schleife

11 Projekte und hybride Projekte

Projekte werden verwendet zur:

- Online-Simulation (Hardware-in-the-Loop) von CT-Blöcken und Standard-ASCET Blöcken
- Zeitkontinuierlichen Modellierung mehreren CT-Strukturen mit verschiedenen Integrationsalgorithmen und Schrittweiten einem Projekt
- Simulation von CT-Blöcken in Echtzeit

Ein Projekt kann aus Standard- oder/und zeitkontinuierlichen Modulen bzw. Strukturen bestehen. Ein hybrides Projekt ist ein Projekt, das sowohl Standard-ASCET Blöcke als auch zeitkontinuierliche Komponenten enthält. Beispielsweise wird bei der Hardware-in-the-Loop Simulation das Senden, Empfangen und Verarbeiten von Signalen vom realen Prozess (der in zeitkontinuierlichen Strukturen simuliert wird) gewöhnlich von Standard-Modulen behandelt.

Bei der Modellierung und Simulation eines Systems mit sehr schnellen und sehr langsamen Komponenten, z. B. hydraulischen und mechanischen Komponenten, kann die Verwendung verschiedener Integrationsmethoden oder verschiedener Integrationsschritte die Rechenzeit verringern. Dazu müssen sich die entsprechenden Modellteile jeweils in einem CT-Basis- oder CT-Strukturblock befinden.

Die verschiedenen CT-Modellteile werden in ein Projekt geladen und im Blockdiagrammeditor miteinander verbunden. Dabei kann in einem Projekt jeder CT-Modellteil (CT-Basisblock oder CT-Strukturblock) als unabhängiger Prozess mit eigener Integrationsmethode und eigener Integrationsschrittgröße berechnet werden. Es muss allerdings berücksichtigt werden, dass die verschiedenen Blöcke in verschiedene Tasks eingebunden werden, die nur in festen, wählbaren Zeitrastern `dT` miteinander kommunizieren.

Es werden keine Werte bei Zwischenwerten des Integrationsverfahrens, wie bei der Kopplung von CT-Blöcken zu CT-Strukturen üblich, ausgetauscht. Es gibt auch keinen automatische Semantikprüfung wie bei CT-Strukturen, die die Rechenreihenfolge bei der Integration bestimmt. Das oben Gesagte gilt nur für CT-Blöcke/Strukturen auf Projektebene. CT-Blöcke und CT-Strukturen innerhalb der CT-Strukturen kommunizieren natürlich auch in Projekten im Raster der Integrationsschrittweite.

Um numerische Stabilität sicherzustellen, sollten stark gekoppelte Systeme deshalb nicht auf Projektebene, sondern innerhalb von CT-Strukturen gekoppelt werden. Dagegen können schwach gekoppelte Systeme in Projekten strukturiert werden. Vorteil bei schwach gekoppelten Systemen mit großem Dynamikunterschied ist, dass das Integrationsverfahren und die Integrationsschrittweite individuell festgelegt werden kann und so Rechenzeit optimal eingesetzt werden kann.

Abb. 11-1 zeigt schematisch ein Projekt, das aus einem diskret rechnenden Standardblock und zwei verschiedenen CT-Strukturblöcken aufgebaut ist.

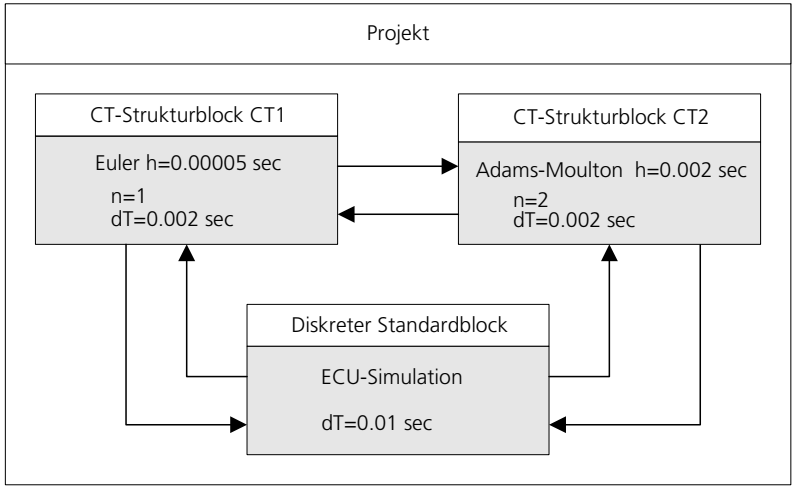


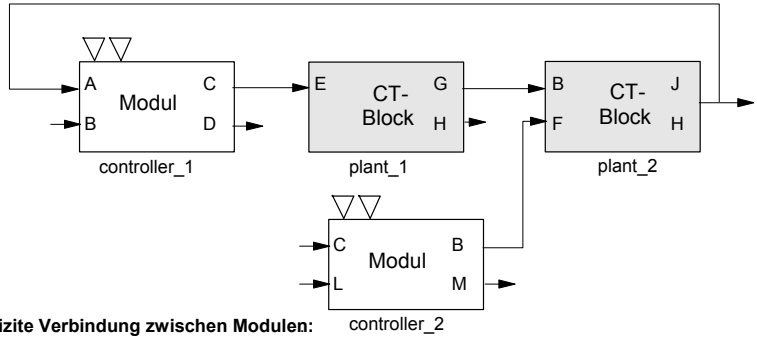
Abb. 11-1 Projekt mit zwei zeitkontinuierlichen und einem diskreten Block

Ein Steuerungsmodell (Standard-ASCET Block) und ein Streckenmodell (zeitkontinuierliche CT-Strukturblöcke) werden zusammen in einem hybriden Projekt simuliert, z.B. zur ECU-Test-Automatisierung. Der zeitkontinuierliche Modellteil ist wiederum aufgeteilt in zwei CT-Strukturblöcke mit unterschiedlichen Integrationsverfahren und unterschiedlicher Schrittweite. Die Kommunikation zwischen den CT-Blöcken findet im 2 Millisekundentakt statt, während die Kommunikation zwischen den CT-Blöcken und dem diskreten Standardblock alle 10 Millisekunden stattfindet.

11.1 Kombinieren zeitkontinuierlicher Blöcke mit Modulen

Diskrete Module in einem Projekt kommunizieren über Messages (globale Variablen in ASCET Blöcken). Bei ihnen existieren keine expliziten Verbindungen (Verbindungslinien) zwischen Send- und Receive-Message; sie werden durch ihre Namen einander zugeordnet.

Dagegen kommunizieren zeitkontinuierliche Blöcke unter sich und mit Modulen über grafisch spezifizierte Verbindungen. Die Verbindungen werden nach der gleichen Methode aufgebaut wie in Blockdiagrammen.



implizite Verbindung zwischen Modulen:
controller_1 - B und controller_2 - B
controller_1 - C und controller_2 - C

keine implizite Verbindung zwischen Modulen und CT- Blöcken

keine implizite Verbindung zwischen CT- Blöcken

Abb. 11-2 Kombinieren zeitkontinuierlicher Blöcke mit Modulen

Für diskrete Module muss der Benutzer explizit die Tasks definieren und angeben, zu welchen die im Moduleditor definierten Prozesse zugewiesen werden.

Für CT-Blöcke müssen keine Tasks explizit definiert werden - sie werden bei Bedarf automatisch definiert. Zu jedem CT-Block werden eine `simulate`-Task und eine `event`-Task generiert. Zusätzlich werden eine gemeinsame `init`-Task und eine gemeinsame `terminate`-Task für alle CT-Blöcke in einem Projekt erzeugt. Zum obigen Beispiel werden die folgenden Tasks automatisch erstellt:

- `simulate_CT1 (plant_1)`
- `simulate_CT2 (plant_2)`
- `event_CT1 (plant_1)`
- `event_CT2 (plant_2)`
- `initialize_CT (plant_1 ... plant_n)`
- `terminate_CT (plant_1 ... plant_n)`

Diese vordefinierten Tasks sind statisch. Sie werden alle als kooperative Tasks definiert. In den folgenden Abschnitten ist die Bedeutung dieser Tasks genauer beschrieben.

`simulate_CTn`-Tasks:

Bei den `simulate_CTn`-Tasks wird ein Simulationsschritt berechnet; die Schrittgröße ist dT. Die Schrittgröße kann für jede `simulate_CTn`-Task explizit angegeben werden; dadurch sind mehrere Integrationsverfahren für verschiedene CT-Strukturblöcke in einem Projekt möglich. Die Integrationsschrittgröße kann während eines Experiments interaktiv eingestellt werden. Normalerweise ist der Triggermodus einer Simulations-Task *Timer*.

`event_CTn`-Tasks:

Bei Aufruf der `event_CTn`-Task werden die `event`-Methoden der zugrundeliegenden CT-Blöcke ausgeführt. Normalerweise werden `event`-Methoden asynchron aufgerufen, daher sollte der Triggermodus der `event_CTn`-Task *Software* oder *Event* sein.

`initialize_CT`-Task:

Bei Aufruf der `initialize_CT`-Task werden die `init`-Methoden der zugrundeliegenden CT-Blöcke ausgeführt. Normalerweise werden `init`-Methoden zu Beginn einer Simulation berechnet; daher sollte der Triggermodus der `initialize_CTn`-Task *Init* sein.

`terminate_CT`-Task:

Bei Aufruf der `terminate_CT`-Task werden die `terminate`-Methoden der zugrundeliegenden CT-Blöcke ausgeführt. Die `terminate`-Task wird automatisch ausgeführt, wenn ein Experiment endet.

ASCET V5.2

Referenzlisten

12 ASCET-Systembibliothek

12.1 Bitoperatoren

12.1.1 and

`and` führt ein bitweises UND der beiden Eingangswerte durch.

Methoden	Argumente	Rückgabewert
`and`	`bitArray1:: unsigned discrete` `bitArray2:: unsigned discrete`	`unsigned discrete`

Beim Aktivieren der Methode

`and:` Das Ergebnis der bitweisen UND-Verknüpfung der beiden Argumente wird zurückgegeben.

12.1.2 clearBit

`clearBit` setzt das Bit an der übergebenen Position auf 0. Die Position des niederwertigsten Bits ist 0.

Methoden	Argumente	Rückgabewert
`clearBit`	`bitArray:: unsigned discrete` `position:: unsigned discrete`	`unsigned discrete`

Beim Aktivieren der Methode

`clearBit:` Das übergebene Argument wird mit nullwertigem Bit an der Position `position` zurückgegeben.

12.1.3 getBit

`getBit` gibt den Wert des Bits an der übergebenen Position als logische Größe zurück.

Methoden	Argumente	Rückgabewert
`getBit`	`bitArray:: unsigned discrete` `position:: unsigned discrete`	`logical`

Beim Aktivieren der Methode

`getBit:` `TRUE` wird zurückgegeben, wenn das Bit an der Position `position` gleich 1 ist, sonst wird `FALSE` zurückgegeben.

12.1.4 or

`or` führt ein bitweises ODER der beiden Eingangswerte durch.

Methoden	Argumente	Rückgabewert
`or`	`bitArray1:: unsigned discrete` `bitArray2:: unsigned discrete`	`unsigned discrete`

Beim Aktivieren der Methode

`or:` Das Ergebnis der bitweisen ODER-Verknüpfung der beiden Argumente wird zurückgegeben.

12.1.5 rotate

`rotate` schiebt die Bits des übergebenen Arguments um die angegebene Anzahl an Positionen nach links. Die links „herausfallenden" Bits werden rechts wieder eingeschoben.

Methoden	Argumente	Rückgabewert
`rotate`	`bitArray:: unsigned discrete` `k::unsigned discrete`	`unsigned discrete`

Beim Aktivieren der Methode

`rotate:` Das Ergebnis der Linksrotation von `bitArray` um `k` Positionen wird zurückgegeben.

12.1.6 setBit

`setBit` setzt das Bit an der übergebenen Position auf 1. Die Position des niederwertigsten Bits ist 0.

Methoden	Argumente	Rückgabewert
`setBit`	`bitArray:: unsigned discrete` `position:: unsigned discrete`	`unsigned discrete`

Beim Aktivieren der Methode

`setBit:` Das übergebene Argument wird mit einswertigem Bit an der Position `position` zurückgegeben.

12.1.7 shiftLeft

`shiftLeft` schiebt alle Bits der Eingangsgröße nach links. Es werden stets Nullen nachgeschoben.

Methoden	Argumente	Rückgabewert
`shiftLeft`	`bitArray:: unsigned discrete` `k::unsigned discrete`	`unsigned discrete`

Beim Aktivieren der Methode

`shiftLeft:` Das Ergebnis der bitweisen Linksverschiebung um `k` Positionen wird zurückgegeben. Die Verschiebung um eine Position entspricht der Multiplikation mit zwei.

12.1.8 shiftRight

`shiftRight` schiebt alle Bits der Eingangsgröße nach rechts. Da es sich um eine vorzeichenfreie (unsigned) Größe handelt, werden stets Nullen nachgeschoben.

Methoden	Argumente	Rückgabewert
`shiftRight`	`bitArray:: unsigned discrete` `k::unsigned discrete`	`unsigned discrete`

Beim Aktivieren der Methode

`shiftRight:` Das Ergebnis der bitweisen Rechtsverschiebung um `k` Positionen wird zurückgegeben.

12.1.9 toggleBit

`toggleBit` ersetzt das Bit an der übergebenen Position durch sein Einerkomplement.

Methoden	Argumente	Rückgabewert
`toggleBit`	`bitArray:: unsigned discrete` `position:: unsigned discrete`	`unsigned discrete`

Beim Aktivieren der Methode

`toggleBit:` Das übergebene Argument wird mit invertiertem Bit an der Position `position` zurückgegeben.

12.1.10 writeBit

`writeBit` schreibt den Wert der angelegten logischen Größe an die übergebene Position.

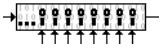
Methoden	Argumente	Rückgabewert

`writeBit`	`bitArray:: unsigned discrete`
	`aBool::logical`
	`position:: unsigned discrete`
		`unsigned discrete`

Beim Aktivieren der Methode

`writeBit` Für `aBool = FALSE` wird das Argument mit nullwertigem Bit an der Position `position` zurückgegeben, für `aBool = TRUE` wird das Argument mit einswertigem Bit an der Position `position` zurückgegeben.

12.1.11 writeByte



`writeByte` schreibt die Werte der acht angelegten logischen Größen an die acht niederwertigsten Bitpositionen des übergebenen Arguments.

Methoden	Argumente	Rückgabewert
`writeByte`	`bitArray:: unsigned discrete`	`unsigned discrete`
	`b0::logical`	
	`b1::logical`	
	`b2::logical`	
	`b3::logical`	
	`b4::logical`	
	`b5::logical`	
	`b6::logical`	
	`b7::logical`	

Beim Aktivieren der Methode

`writeByte:` Die Werte von `b0` bis `b7` werden an die Bitpositionen 0 bis 7 des Arguments geschrieben und das Argument wird zurückgegeben. Dabei ist die Bitposition 0 die Position des niederwertigsten Bits und die Werte `TRUE` und `FALSE` werden auf die Bitwerte 1 bzw. 0 abgebildet.

12.1.12 xor

`xor` führt ein bitweises exklusives ODER der Eingangswerte durch.

Methoden	Argumente	Rückgabewert
`xor`	`bitArray1:: unsigned discrete`	`unsigned discrete`
	`bitArray2:: unsigned discrete`	

Beim Aktivieren der Methode

`xor:` Das Ergebnis der bitweisen exklusiven ODER-Verknüpfung der beiden Argumente wird zurückgegeben.

12.2 Komparatoren

12.2.1 `ClosedInterval`

`ClosedInterval` gibt TRUE zurück, wenn der Wert x im abgeschlossenen Intervall durch A und B definiert ist.

Methoden	Argumente	Rückgabewert
`out`	`x::continuous`	`logical`
	`A::continuous`	
	`B::continuous`	

Beim Aktivieren der Methode

`out:` TRUE wird zurückgegeben, wenn A <= x <= B. Andernfalls wird FALSE zurückgegeben.

12.2.2 `LeftOpenInterval`

`LeftOpenInterval` gibt TRUE zurück, wenn der Wert x im links offenen Intervall durch A und B definiert ist.

Methoden	Argumente	Rückgabewert
`out`	`x::continuous`	`logical`
	`A::continuous`	
	`B::continuous`	

Beim Aktivieren der Methode

`out:` TRUE wird zurückgegeben, wenn A < x <= B. Andernfalls wird FALSE zurückgegeben.

12.2.3 OpenInterval

OpenInterval gibt TRUE zurück, wenn der Wert x im offenen Intervall durch A und B definiert ist.

Methoden	Argument	Rückgabewert
out	x::continuous	logical
	A::continuous	
	B::continuous	

Beim Aktivieren der Methode

out: TRUE wird zurückgegeben, wenn A < x < B. Andernfalls wird FALSE zurückgegeben.

12.2.4 RightOpenInterval

RightOpenInterval gibt TRUE zurück, wenn der Wert x im rechts offenen Intervall durch A und B definiert ist.

Methoden	Argumente	Rückgabewert
out	x::continuous	logical
	A::continuous	
	B::continuous	

Beim Aktivieren der Methode

out: TRUE wird zurückgegeben, wenn A <= x < B. Andernfalls wird FALSE zurückgegeben.

12.2.5 GreaterZero

GreaterZero gibt TRUE zurück, wenn der Wert x größer Null ist.

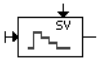
Methoden	Argumente	Rückgabewert
out	x::continuous	logical

Beim Aktivieren der Methode

out:	TRUE wird zurückgegeben, wenn x > 0.0. Andernfalls wird FALSE zurückgegeben.

12.3 Zähler und Timer

12.3.1 CountDown



CountDown dekrementiert den Zähler und signalisiert, wenn der Zähler Null erreicht hat.

Methoden	Argumente	Rückgabewert
start	startValue:: unsigned discrete	Keiner
compute	Keine	Keiner
out	Keine	logical

Beim Aktivieren der Methode

start:	Der Zähler wird auf den Ausgangswert gestellt.
compute:	Der Zähler wird um Eins dekrementiert.
out:	TRUE wird zurückgegeben, wenn der Zähler größer Null ist. Andernfalls wird FALSE zurückgegeben.

12.3.2 CountDownEnabled

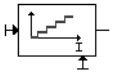
CountDownEnabled dekrementiert den Zähler und signalisiert, wenn der Zähler Null erreicht hat. Diese Zähler muss explizit aktiviert werden.

Methoden	Argumente	Rückgabewert
start	startValue:: unsigned discrete	Keiner
compute	enable::logical	Keiner
out	Keine	logical

Beim Aktivieren der Methode

start:	Der Zähler wird auf den Ausgangswert gestellt.
compute:	Wenn enable TRUE ist, wird der Zähler um Eins dekrementiert.
out:	TRUE wird zurückgegeben, wenn der Zähler größer Null ist. Andernfalls wird FALSE zurückgegeben.

12.3.3 Counter



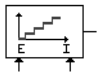
Counter inkrementiert den Zähler um Eins.

Methoden	Argumente	Rückgabewert
reset	Keine	Keiner
compute	Keine	Keiner
out	Keine	unsigned discrete

Beim Aktivieren der Methode

reset:	Der Zähler wird auf Null gestellt.
compute:	Der Zähler wird um Eins inkrementiert.
out:	Es wird der Zählerwert zurückgegeben.

12.3.4 CounterEnabled



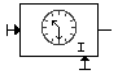
Counter inkrementiert den Zähler um Eins. Dieser Zähler muss explizit aktiviert werden.

Methoden	Argumente	Rückgabewert
reset	initEnable::logical	Keiner
compute	enable::logical	Keiner
out	Keine	unsigned discrete

Beim Aktivieren der Methode

reset:	Wenn initEnable TRUE ist, wird der Zähler auf Null gestellt.
compute:	Wenn enable TRUE ist, wird der Zähler um Eins inkrementiert.
out:	Es wird der Zählerwert zurückgegeben.

12.3.5 StopWatch



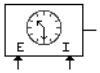
StopWatch inkrementiert den Zeitzähler um ein dT.

Methoden	Argumente	Rückgabewert
reset	Keine	Keiner
compute	Keine	Keiner
out	Keine	continuous

Beim Aktivieren der Methode

reset:	Der Zeitzähler wird auf Null gestellt.
compute:	Der Zeitzähler wir um dT inkrementiert.
out:	Der Zeitzählerwert, d. h. die seit dem letzten Start vergangene Zeit, wird zurückgegeben.

12.3.6 StopWatchEnabled



`StopWatchEnabled` inkrementiert den Zeitzähler um ein `dT`. Dieser Zeitzähler muss explizit aktiviert werden.

Methoden	Argumente	Rückgabewert
`reset`	`initEnable::logical`	`Keiner`
`compute`	`enable::logical`	`Keiner`
`out`	`Keine`	`continuous`

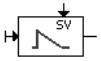
Beim Aktivieren der Methode

`reset:` Wenn `initEnable` `TRUE` ist, wird der interne Zeitzähler auf Null gestellt.

`compute:` Wenn `enable` `TRUE` ist, wird der Zeitzähler um `dT` inkrementiert.

`out:` Der Zeitzähler-Wert, d. h. die seit dem letzten Start und unter der Bedingung, dass `enabled` `TRUE` war, vergangene Zeit wird zurückgegeben.

12.3.7 Timer



`Timer` dekrementiert den Zeitzähler um `dT` und signalisiert, wenn der Zeitzähler Null erreicht hat. Er kann nicht erneut ausgelöst werden.

Methoden	Argumente	Rückgabewert
`start`	`startTime::continuous`	`Keiner`
`compute`	`Keine`	`Keiner`
`out`	`Keine`	`logical`

Beim Aktivieren der Methode

`start`: Der Zeitzähler wird auf `startTime` gestellt, wenn der Zeitzählerwert vorher kleiner/gleich Null war.

`compute:` Der Zeitzähler wird um `dT` dekrementiert.

`out:` TRUE wird zurückgegeben, wenn der Zeitzähler größer Null ist. Andernfalls wird FALSE zurückgegeben.

12.3.8 TimerEnabled

`TimerEnabled` dekrementiert den Zeitzähler um `dT` und signalisiert, wenn der Zeitzähler Null erreicht hat. Er muss explizit aktiviert werden.

Methoden	Argumente	Rückgabewert

`compute`	`enable::logical`
	`in::logical`
	`startTime::continuous`
		`Keiner`

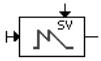
`out`	`Keine`	`logical`

Beim Aktivieren der Methode

`compute:` Wenn `enable` TRUE ist, `in` eine ansteigende Flanke hat und der Zeitzählerwert kleiner/gleich Null ist, wird der Zeitzähler gestartet, d. h. sein Zählerwert wird auf die Startzeit gesetzt. Andernfalls wird der Zeitzähler um `dT` dekrementiert. Wenn `enable` FALSE ist, geschieht gar nichts.

`out:` TRUE wird zurückgegeben, wenn der Zeitzähler größer Null ist. Andernfalls wird FALSE zurückgegeben.

12.3.9 TimerRetrigger



`TimerRetrigger` dekrementiert den Zeitzähler um `dT` und signalisiert, wenn der Zeitzähler Null erreicht hat. Er kann erneut ausgelöst werden.

Methoden	Argumente	Rückgabewert
`start`	`startTime::continuous`	`Keiner`
`compute`	`Keine`	`Keiner`
`out`	`Keine`	`logical`

Beim Aktivieren der Methode

`start`: Der Zeitzähler wird auf den Startwert gesetzt.

`compute`: Der Zeitzähler wird um `dT` dekrementiert.

`out`: `TRUE` wird zurückgegeben, wenn der Zeitzähler größer Null ist. Andernfalls wird `FALSE` zurückgegeben.

12.3.10 `TimerRetriggerEnabled`

`TimerRetriggerEnabled` dekrementiert den Zeitzähler um `dT` und signalisiert, wenn der Zeitzähler Null erreicht hat. Er kann erneut ausgelöst und muss explizit aktiviert werden.

Methoden	Argumente	Rückgabewert

`compute`	`enable::logical`
	`in::logical`
	`startValue::continuous`
		`Keiner`

`out`	`Keine`	`logical`

Beim Aktivieren der Methode

`compute`: Wenn `enable` `TRUE` ist und `in` eine ansteigende Flanke hat, wird der Zeitzähler gestartet, d. h. sein Zählerwert wird auf den Startwert gesetzt. Andernfalls wird der Zeitzähler um `dT` (den Zeitrahmen) dekrementiert. Wenn `enable` `FALSE` ist, geschieht gar nichts.

`out`: `TRUE` wird zurückgegeben, wenn der Zeitzählerwert größer Null ist. Andernfalls wird `FALSE` zurückgegeben.

12.4 Verzögerung

12.4.1 `DelaySignal`

`DelaySignal` verzögert sein Eingabesignal um einen Bewertungsschritt.

Methoden	Argumente	Rückgabewert
`compute`	`signal::logical`	`Keiner`
`out`	`Keine`	`logical`

Beim Aktivieren der Methode

`compute:`	Das Eingabesignal wird zwischengespeichert.
`out:`	Es wird das zwischengespeicherte Signal zurückgegeben; somit wird das Eingabesignal um einen Schritt verzögert.

12.4.2 DelaySignalEnabled

`DelaySignalEnabled` verzögert sein Eingabesignal um einen Auswertungsschritt. Es muss explizit aktiviert werden.

Methoden	Argumente	Rückgabewert
`reset`	`initEnable::logical` `initValue::logical`	`Keiner`
`compute`	`signal::logical` `enable::logical`	`Keiner`
`out`	`Keine`	`logical`

Beim Aktivieren der Methode

`reset:`	Wenn `initEnable` TRUE ist, wird `initValue` zwischengespeichert.
`compute:`	Wenn `enable` TRUE ist, wird das Eingabesignal zwischengespeichert.
`out:`	Es wird das zwischengespeicherte Signal zurückgegeben; somit wird das Eingabesignal um einen Schritt verzögert.

12.4.3 DelayValue

`DelayValue` verzögert seinen Eingabewert um einen Bewertungsschritt.

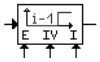
Methoden	Argumente	Rückgabewert
`compute`	`value::continuous`	`Keiner`
`out`	`Keine`	`continuous`

Beim Aktivieren der Methode

`compute:`	Der Eingabewert `value` wird zwischengespeichert.

`out:` Es wird der zwischengespeicherte Wert zurückgegeben; somit wird der Eingabewert um einen Schritt verzögert.

12.4.4 DelayValueEnabled



`DelayValueEnabled` verzögert seinen Eingabewert um einen Auswertungsschritt. Es muss explizit aktiviert werden.

Methoden	Argumente	Rückgabewert
`reset`	`initEnable::logical` `initValue::continuous`	`Keiner`
`compute`	`value::continuous` `enable::logical`	`Keiner`
`out`	`Keine`	`logical`

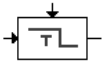
Beim Aktivieren der Methode

`reset:` Wenn `initEnable` TRUE ist, wird der initValue zwischengespeichert.

`compute:` Wenn `enable` TRUE ist, wird der Eingabewert zwischengespeichert.

`out:` Es wird der zwischengespeicherte Wert zurückgegeben; somit wird der Eingabewert um einen Schritt verzögert.

12.4.5 TurnOffDelay



`TurnOffDelay` verzögert eine abfallende Flanke des Eingangssignals.

Methoden	Argumente	Rückgabewert
`compute`	`signal::logical` `delayTime::continuous`	`Keiner`
`out`	`Keine`	`logical`

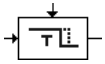
Beim Aktivieren der Methode

`compute:` Eine abfallende Flanke des Eingangssignals wird verzögert. Springt das Signal von TRUE zu FALSE, wird ein Zeitglied gestartet. Im

Zustand FALSE wird das Zeitglied um dT inkrementiert und mit der delayTime verglichen. Ist das Eingangssignal TRUE, wird das Zeitglied rückgestellt.

`out:` TRUE wird zurückgegeben, wenn das Eingangssignal TRUE ist oder das Zeitglied die delayTime nicht überschritten hat. Andernfalls wird FALSE zurückgegeben.

12.4.6 TurnOffDelayVariable



TurnOffDelayVariable verzögert eine abfallende Flanke des Eingangssignals. Die Dauer der Verzögerung kann über die Variable Time zur Laufzeit geändert werden.

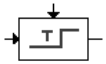
Methoden	Argumente	Rückgabewert
compute	signal::logical delayTime::continuous	Keiner
out	Keine	logical

Beim Aktivieren der Methode

`compute:` Eine abfallende Flanke des Eingangssignals wird verzögert. Springt das Signal von TRUE zu FALSE, wird ein Zeitglied gestartet. Im Zustand FALSE wird das Zeitglied um dT inkrementiert und mit der delayTime verglichen. Ist das Eingangssignal TRUE, wird das Zeitglied rückgestellt.

`out:` TRUE wird zurückgegeben, wenn das Eingangssignal TRUE ist oder das Zeitglied die delayTime nicht überschritten hat. Andernfalls wird FALSE zurückgegeben.

12.4.7 TurnOnDelay



TurnOnDelay verzögert eine ansteigende Flanke des Eingangssignals.

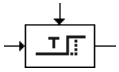
Methoden	Argumente	Rückgabewert
compute	signal::logical delayTime::continuous	Keiner
out	Keine	logical

Beim Aktivieren der Methode

compute: Eine ansteigende Flanke des Eingangssignals wird verzögert. Springt das Signal von FALSE zu TRUE, wird ein Zeitglied gestartet. Im Zustand TRUE wird das Zeitglied um dT inkrementiert und mit der delayTime verglichen. Ist das Eingangssignal FALSE, wird das Zeitglied rückgestellt.

out: FALSE wird zurückgegeben, wenn das Eingangssignal FALSE ist oder das Zeitglied die delayTime nicht überschritten hat. Andernfalls wird TRUE zurückgegeben.

12.4.8 TurnOnDelayVariable



TurnOnDelayVariable verzögert eine ansteigende Flanke des Eingangssignals. Die Dauer der Verzögerung kann über die Variable Time zur Laufzeit geändert werden.

Methoden	Argumente	Rückgabewert
compute	signal::logical delayTime::continuous	Keiner
out	Keine	logical

Beim Aktivieren der Methode

compute: Eine ansteigende Flanke des Eingangssignals wird verzögert. Springt das Signal von FALSE zu TRUE, wird ein Zeitglied gestartet. Im Zustand TRUE wird das Zeitglied um dT inkre-

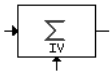
mentiert und mit der `delayTime` verglichen. Ist das Eingangssignal FALSE, wird das Zeitglied rückgestellt.

`out:` FALSE wird zurückgegeben, wenn das Eingangssignal FALSE ist oder das Zeitglied die `delayTime` nicht überschritten hat. Andernfalls wird TRUE zurückgegeben.

12.5 Speicher

12.5.1 Accumulator

`Accumulator` summiert seinen Eingabewert.



Methoden	Argumente	Rückgabewert
`reset`	`initValue::continuous`	`Keiner`
`compute`	`value::continuous`	`Keiner`
`out`	`Keine`	`continuous`

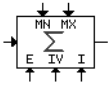
Beim Aktivieren der Methode

`reset:` Der Akkumulatorwert wird auf den `initValue` gesetzt.

`compute:` Der Akkumulator wird um den Eingabewert inkrementiert, d. h. `accumulator (new) = accumulator (old) + input value`.

`out:` Es wird der Akkumulatorwert zurückgegeben.

12.5.2 AccumulatorEnabled



`AccumulatorEnabled` summiert seinen Eingabewert. Es muss explizit aktiviert werden, und sein Akkumulatorwert kann begrenzt werden.

Methoden	Argumente	Rückgabewert
`reset`	`initValue::continuous` `initEnable::logical`	`Keiner`

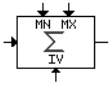
`compute`	`value::continuous`
	`mn::continuous`
	`mx::continuous`
	`enable::logical`
		`Keiner`

`out`	`Keine`	`continuous`

Beim Aktivieren der Methode

`reset:`	Wenn `initEnable` `TRUE` ist, wird der Akkumulatorwert auf `initValue` gesetzt.
`compute:`	Wenn `enable` `TRUE` ist, wird der Akkumulator um den Eingabewert inkrementiert, d. h. `accumulator(new) = accumulator(old) + input value`. Darüber hinaus wird der Akkumulatorwert durch `mn` und `mx` begrenzt.
`out:`	Es wird der Akkumulatorwert zurückgegeben.

12.5.3 AccumulatorLimited



`AccumulatorLimited` summiert seinen Eingabewert. Sein Akkumulatorwert kann begrenzt werden.

Methoden	Argumente	Rückgabewert
`reset`	`initValue::continuous`	`Keiner`

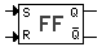
`compute`	`value::continuous`
	`mn::continuous`
	`mx::continuous`
		`Keiner`

`out`	`Keine`	`continuous`

Beim Aktivieren der Methode

`reset:`	Der Akkumulatorwert wird auf `initValue` gesetzt.
`compute:`	Der Akkumulator wird um den Eingabewert inkrementiert, d. h. `accumulator(new) = accumulator(old) +input value`. Darüber hinaus wird der Akkumulator-Wert durch `mn` und `mx` begrenzt.
`out:`	Es wird der Akkumulatorwert zurückgegeben.

12.5.4 RSFlipFlop



`RSFlipFlop` ist ein Flipflop mit einem Reset-Eingang und einem Set-Eingang, wobei der Reset-Eingang den Set-Eingang dominiert.

Methoden	Argumente	Rückgabewert
`compute`	`r::logical` `s::logical`	`Keiner`
`q`	`Keine`	`logical`
`nq`	`Keine`	`logical`

Beim Aktivieren der Methode

`compute:`	Wenn `r` TRUE ist, wird der Zustand des Flipflops auf FALSE gesetzt. Andernfalls wird, wenn `s` TRUE ist, der Zustand auf TRUE gesetzt. Sind sowohl `r` als auch `s` FALSE, bleibt der Zustand unverändert.
`q`:	Es wird der Zustand des Flipflops zurückgegeben.
`nq`:	Es wird der negierte Zustand zurückgegeben.

12.6 Sonstiges

12.6.1 DeltaOneStep

`DeltaOneStep` gibt die Differenz zwischen dem aktuellen Eingabewert und dem letzten Eingabewert zurück.

Methoden	Argumente	Rückgabewert
`compute`	`value::continuous`	`Keiner`
`out`	`Keine`	`continuous`

Beim Aktivieren der Methode

`compute:` Der vorherige Eingabewert wird vom Eingabewert subtrahiert.

`out:` Es wird die Differenz zurückgegeben.

12.6.2 DifferenceQuotient

`DifferenceQuotient` berechnet den Differenzquotienten des Eingabewerts.

Methoden	Argumente	Rückgabewert
`compute`	`value::continuous`	`Keiner`
`out`	`Keine`	`continuous`

Beim Aktivieren der Methode

`compute:` Es wird der Differenzquotient `(value - previous value) / dT` berechnet.

`out:` Es wird der Differenzquotient zurückgegeben.

12.6.3 EdgeBi

EdgeBi stellt eine bidirektionale Flanke des logischen Eingangssignals fest.

Methoden	Argumente	Rückgabewert
compute	signal::logical	Keiner
out	Keine	logical

Beim Aktivieren der Methode

compute:	Das Eingangssignal wird mit dem vorherigen Eingangssignal verglichen.
out:	TRUE wird zurückgegeben, wenn sich das Eingangssignal und das vorherige Eingangssignal unterscheiden. Andernfalls wird FALSE zurückgegeben.

12.6.4 EdgeFalling

EdgeFalling stellt die abfallende Flanke des logischen Eingangssignals fest.

Methoden	Argumente	Rückgabewert
compute	signal::logical	Keiner
out	Keine	logical

Beim Aktivieren der Methode

compute:	Das Eingangssignal wird mit dem vorherigen Eingangssignal verglichen.
out:	TRUE wird zurückgegeben, wenn das Eingangssignal low ist und das vorherige Eingangssignal high war. Andernfalls wird FALSE zurückgegeben.

12.6.5 EdgeRising

EdgeRising stellt eine ansteigende Flanke des logischen Eingangssignals fest.

Methoden	Argumente	Rückgabewert
compute	signal::logical	Keiner
out	Keine	logical

Beim Aktivieren der Methode

compute:	Das Eingangssignal wird mit dem vorherigen Eingangssignal verglichen.
out:	TRUE wird zurückgegeben, wenn das Eingangssignal high ist und das vorherige Eingangssignal low war. Andernfalls wird FALSE zurückgegeben.

12.6.6 Mux1of4

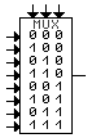
Mux1of4 schaltet zwischen den vier Eingabewerten s0,...,s3 in der Binärdarstellung von deren Index um.

Methoden	Argumente	Rückgabewert
out	b0::logical b1::logical s0::continuous s1::continuous s2::continuous s3::continuous	continuous

Beim Aktivieren der Methode

out:	Es wird der Eingabewert si (index i) zurückgegeben, wobei i = b0 + 2*b1 und FALSE als 0 und TRUE als 1 interpretiert werden.

12.6.7 Mux1of8



`Mux1of8` schaltet zwischen den acht Eingabewerten `s0,...,s7` in der Binärdarstellung von deren Index um.

Methoden	Argumente	Rückgabewert

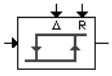
`out`	`b0::logical`
	`b1::logical`
	`b2::logical`
	`s0::continuous`
	`s1::continuous`
	`s2::continuous`
	`s3::continuous`
	`s4::continuous`
	`s5::continuous`
	`s6::continuous`
	`s7::continuous`
		`continuous`

Beim Aktivieren der Methode

`out:` Es wird der Eingabewert `si(index i)` zurückgegeben, wobei `i = b0 + 2*b1+ 4*b2` und FALSE als 0 und TRUE als 1 interpretiert werden.

12.7 Nichtlineare

12.7.1 Hysteresis-Delta-RSP



`Hysteresis-Delta-RSP` ist eine Hysterese mit einem rechten Schaltpunkt und einem Delta-Offset.

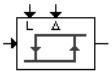
Methoden	Argumente	Rückgabewert

`out`	`x::continuous`
	`delta::continuous`
	`rsp::continuous`
		`logical`

Beim Aktivieren der Methode

`out:` TRUE wird zurückgegeben, wenn `x > rsp`. FALSE wird zurückgegeben, wenn `x < (rsp - delta)`. Der Rückgabewert bleibt unverändert, wenn `x` innerhalb des offenen Intervalls `](rsp - delta), rsp[` liegt.

12.7.2 Hysteresis-LSP-Delta



`Hysteresis-LSP-Delta` ist eine Hysterese mit einem linken Schaltpunkt und einem Delta-Offset.

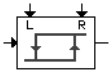
Methoden	Argumente	Rückgabewert

`out`	`x::continuous`
	`lsp::continuous`
	`delta::continuous`
		`logical`

Beim Aktivieren der Methode

`out:` TRUE wird zurückgegeben, wenn `x > (lsp + delta)`. FALSE wird zurückgegeben, wenn `x < lsp`. Der Rückgabewert bleibt unverändert, wenn `x` innerhalb des offenen Intervalls `]lsp, (lsp + delta)[` liegt.

12.7.3 Hysteresis-LSP-RSP



`Hysteresis-LSP-RSP` ist eine Hysterese mit einem linken und einem rechten Schaltpunkt.

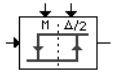
Methoden	Argumente	Rückgabewert

`out`	`x::continuous`
	`lsp::continuous`
	`rsp::continuous`
		`logical`

Beim Aktivieren der Methode

`out:` TRUE wird zurückgegeben, wenn `x > rsp`. FALSE wird zurückgegeben, wenn `x < lsp`. Der Rückgabewert bleibt unverändert, wenn `x` innerhalb des offenen Intervalls `]lsp, rsp[` liegt.

12.7.4 Hysteresis-MSP-DeltaHalf



`Hysteresis-MSP-DeltaHalf` ist eine Hysterese mit einem Mittelschaltpunkt und einem Offset von Delta/2.

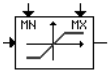
Methoden	Argumente	Rückgabewert

`out`	`x::continuous`
	`msp::continuous`
	`deltahalf::continuous`
		`logical`

Beim Aktivieren der Methode

`out:` TRUE wird zurückgegeben, wenn `x > (msp + deltahalf)`. FALSE wird zurückgegeben, wenn `x < (msp - deltahalf)`. Der Rückgabewert bleibt unverändert, wenn die Eingabe x im offenen Intervall `](msp - deltahalf), (msp + deltahalf)[` liegt.

12.7.5 Limiter



`Limiter` gibt die Eingabe `x` begrenzt durch `mn` und `mx` zurück.

Methoden	Argumente	Rückgabewert

`out`	`x::continuous`
	`mn::continuous`
	`mx::continuous`
		`continuous`

Beim Aktivieren der Methode

`out:` Die Eingabe `x` wird durch `mn` und `mx` begrenzt und wird zurückgegeben, d. h. `max( min(x, mx), mn)`. Es wird nicht geprüft, ob `mn <= mx`.

12.7.6 Signum

`Signum` gibt das Vorzeichen der Eingabe zurück.

Methoden	Argumente	Rückgabewert
`out`	`x::continuous`	`continuous`

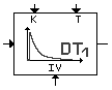
Beim Aktivieren der Methode

`out:` Es wird 1.0 zurückgegeben, wenn `x > 0.0`, 0.0 wird zurückgegeben, wenn `x = 0.0`, -1.0 wird zurückgegeben, wenn `x < 0.0`.

12.8 Übertragungsfunktion

12.8.1 Regelung

dT1



`dT1` ist eine zeitdiskrete Differenz-Übertragungsfunktion mit einer Zeitkonstanten `T` und einer Verstärkungskonstanten `K`.

Methoden	Argumente	Rückgabewert

`compute`	`in::continuous`
	`T::continuous`
	`K::continuous`
		`Keiner`

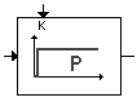
`out`	`Keine`	`continuous`

Beim Aktivieren der Methode

`compute:` Der Differenzierungswert wird über eine P-Funktion und eine I-Funktion, die rückgekoppelt wird, berechnet.

`out:` Es wird der Differenzierungswert zurückgegeben.

P



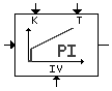
P ist eine zeitdiskrete proportionale Übertragungsfunktion mit Verstärkungskonstante K.

Methoden	Argumente	Rückgabewert
`out`	`in::continuous` `K::continuous`	`continuous`

Beim Aktivieren der Methode

`out:` Es wird der **Rückgabewert** out = in * K berechnet.

PI



`PI` ist ein zeitdiskreter proportionaler Integrator mit einer Zeitkonstanten T und Verstärkungskonstante K.

Methoden	Argumente	Rückgabewert
`reset`	`initValue::continuous`	`Keiner`

`compute`	`in::continuous`
	`T::continuous`
	`K::continuous`
		`Keiner`

`out`	`Keine`	`continuous`

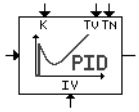
Beim Aktivieren der Methode

`reset:` Der Integratorwert wird auf den `initValue` gesetzt.

`compute:` Der Wert der PI-Funktion wird als Summe einer P-Funktion und einer I-Funktion berechnet.

`out:` Es wird der Wert der PI-Funktion zurückgegeben.

PID



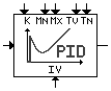
PID ist ein zeitdiskreter proportionaler Integrator mit Differentialanteil mit Zeitkonstanten Tv und Tn sowie Verstärkungskonstante K.

Methoden	Argumente	Rückgabewert
reset	initValue::continuous	Keiner
compute	in::continuous Tv::continuous Tn::continuous K::continuous	Keiner
out	Keine	continuous

Beim Aktivieren der Methode

reset:	Der Integratorwert wird auf den initValue gesetzt.
compute:	Der Wert der PID-Funktion wird als Summe einer P-Funktion, einer D-Funktion und einer I-Funktion berechnet.
out:	Es wird der Wert der PID-Funktion zurückgegeben.

PIDLimited



`PIDLimited` ist ein zeitdiskreter proportionaler Integrator mit Differentialanteil mit Zeitkonstanten Tv und Tn sowie Verstärkungskonstante K. Der Wert des Integrators ist begrenzt.

Methoden	Argumente	Rückgabewert
`reset`	`initValue::continuous`	`Keiner`

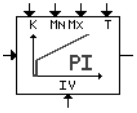
`compute`	`in::continuous`
	`Tv::continuous`
	`Tn::continuous`
	`K::continuous`
	`mn::continuous`
	`mx::continuous`
		`Keiner`

`out`	`Keine`	`continuous`

Beim Aktivieren der Methode

`reset:`	Der Integratorwert wird auf den `initValue` gesetzt.
`compute:`	Der Wert der PID-Funktion wird als Summe einer P-Funktion, einer D-Funktion und einer I-Funktion berechnet, wobei der Integratorwert der I-Funktion durch `mn` und `mx` begrenzt ist.
`out:`	Es wird der Wert der PID-Funktion zurückgegeben.

PILimited



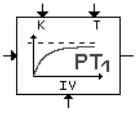
PILimited ist eine zeitdiskreter proportionaler Integrator mit Zeitkonstante T und Verstärkungskonstante K. Der Wert des Integrators ist begrenzt.

Methoden	Argumente	Rückgabewert
reset	initValue::continuous	Keiner
compute	in::continuous T::continuous K::continuous mn::continuous mx::continuous	Keiner
out	Keine	continuous

Beim Aktivieren der Methode

reset:	Der Integratorwert wird auf den initValue gesetzt.
compute:	Der Wert der PI-Funktion wird als Summe einer P-Funktion und einer I-Funktion berechnet, wobei der Integratorwert der I-Funktion durch mn und mx begrenzt ist.
out:	Es wird der Wert der PI-Funktion zurückgegeben.

PT1



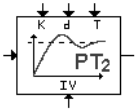
PT1 ist ein zeitdiskreter Tiefpass mit Zeitkonstante T und Verstärkungskonstante K.

Methoden	Argumente	Rückgabewert
reset	initValue::continuous	Keiner
compute	in::continuous T::continuous K::continuous	Keiner
out	Keine	continuous

Beim Aktivieren der Methode

`reset:`	Der Wert des Integrators wird auf `initValue` gesetzt.
`compute:`	Der Wert der PT1-Funktion wird über eine I-Funktion und eine P-Funktion, die rückgekoppelt wird, berechnet.
`out:`	Es wird der Wert der PT1-Funktion zurückgegeben.

PT2



PT2 ist eine zeitdiskrete Verzögerungsfunktion mit Zeitkonstante T, Verstärkungskonstante K und Dämpfung d..

Methoden	Argumente	Rückgabewert
`reset`	`initValue::continuous`	`Keiner`

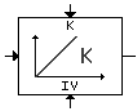
compute	`in::continuous`
	`T::continuous`
	`K::continuous`
	`d::continuous`
		`Keiner`

`out`	`Keine`	`continuous`

Beim Aktivieren der Methode

`reset:`	Die beiden Integratorwerte werden auf den `initValue` gesetzt.
`compute:`	Der Wert der PT2-Funktion wird über zwei I-Funktionen in Reihe berechnet, die über eine Kaskade von zwei P-Funktionen rückgekoppelt werden.
`out:`	Es wird der Wert der PT2-Funktion zurückgegeben.

IntegratorK



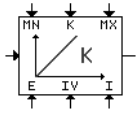
IntegratorK ist ein zeitdiskreter Integrator mit Verstärkungskonstante K.

Methoden	Argumente	Rückgabewert
reset	initValue::continuous	Keiner
compute	in::continuous K::continuous	Keiner
out	Keine	continuous

Beim Aktivieren der Methode

reset:	Der Integratorwert wird auf den initValue gesetzt.
compute:	Der Integrator-Wert wird über Integrator (new) = Integrator (old) + in * dT* K berechnet.
out:	Es wird der Integratorwert zurückgegeben.

IntegratorKEnabled



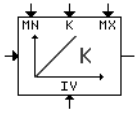
IntegratorKEnabled ist ein zeitdiskreter Integrator mit Verstärkungskonstante K. Er muss explizit aktiviert werden, und sein Integratorwert kann begrenzt werden.

Methoden	Argumente	Rückgabewert
reset	initValue::continuous initEnable::logical	Keiner
compute	in::continuous K::continuous mn::continuous mx::continuous enable::logical	Keiner
out	Keine	continuous

Beim Aktivieren der Methode

`reset:`	Wenn `initEnable` TRUE ist, wird der Integratorwert auf `initValue` gesetzt.
`compute:`	Wenn `enable` TRUE ist, wird der Integratorwert über `Integrator(new) = Integrator(old) + in * dT * K` (begrenzt durch `mn` und `mx`) berechnet.
`out:`	Es wird der Integratorwert zurückgegeben.

`IntegratorKLimited`



`IntegratorKLimited` ist ein zeitdiskreter Integrator mit Verstärkungskonstante `K`. Sein Integratorwert kann begrenzt werden.

Methoden	Argumente	Rückgabewert
`reset`	`initValue::continuous`	`Keiner`

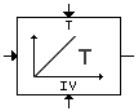
`compute`	`in::continuous`
	`K::continuous`
	`mn::continuous`
	`mx::continuous`
		`Keiner`

`out`	`Keine`	`continuous`

Beim Aktivieren der Methode

`reset:`	Der Integratorwert wird auf den initValue gesetzt.
`compute:`	Der Integratorwert wird über `Integrator (new) = Integrator (old) + in * dT * K` (begrenzt durch `mn` und `mx`) berechnet.
`out:`	Es wird der Integratorwert zurückgegeben.

IntegratorT



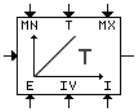
`IntegratorT` ist ein zeitdiskreter Integrator mit Zeitkonstante `T`.

Methoden	Argumente	Rückgabewert
`reset`	`initValue::continuous`	`Keiner`
`compute`	`in::continuous` `T::continuous`	`Keiner`
`out`	`Keine`	`continuous`

Beim Aktivieren der Methode

`reset:`	Der Integratorwert wird auf den `initValue` gesetzt.
`compute:`	Der Integratorwert wird über `Integrator (new) = Integrator (old) + in * dT / T` berechnet.
`out:`	Es wird der Integratorwert zurückgegeben.

IntegratorTEnabled



`IntegratorTEnabled` ist ein zeitdiskreter Integrator mit Zeitkonstante `T`. Er muss explizit aktiviert werden, und sein Integratorwert kann begrenzt werden.

Methoden	Argumente	Rückgabewert
`reset`	`initValue::continuous` `initEnable::logical`	`Keiner`

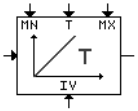
`compute`	`in::continuous`
	`T::continuous`
	`mn::continuous`
	`mx::continuous`
	`enable::logical`
		`Keiner`

`out`	`Keine`	`continuous`

Beim Aktivieren der Methode

`reset:`	Wenn `initEnable` TRUE ist, wird der Integratorwert auf den `initValue` gesetzt.
`compute:`	Wenn `enable` TRUE ist, wird der Integrator-Wert über `Integrator(new) = Integrator(old) + in * dT / T` (begrenzt durch `mn` und `mx`) berechnet.
`out:`	Es wird der Integratorwert zurückgegeben.

`IntegratorTLimited`



`IntegratorTLimited` ist ein zeitdiskreter Integrator mit Zeitkonstante `T`. Sein Integratorwert kann begrenzt werden.

Methoden	Argumente	Rückgabewert
`reset`	`initValue::continuous`	`Keiner`

`compute`	`in::continuous`
	`T::continuous`
	`mn::continuous`
	`mx::continuous`
		`Keiner`

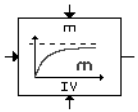
`out`	`Keine`	`continuous`

Beim Aktivieren der Methode

`reset:`	Der Integratorwert wird auf den `initValue` gesetzt.
`compute:`	Der Integratorwert wird über `Integrator (new) = Integrator (old) + in * dT / T` (begrenzt durch `mn` und `mx`) berechnet.
`out:`	Es wird der Integratorwert zurückgegeben.

12.8.3 Lowpass

DigitalLowpass



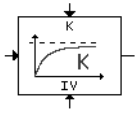
`DigitalLowpass` berechnet rekursiv den Mittelwert des Eingabewertes.

Methoden	Argumente	Rückgabewert
`reset`	`initValue::continuous`	`Keiner`
`compute`	`in::continuous` `m::continuous`	`Keiner`
`out`	`Keine`	`continuous`

Beim Aktivieren der Methode

`reset:`	Der Mittelwert wird auf den `initValue` gesetzt.
`compute:`	Der Mittelwert wird über `Mittelwert(new) = Mittelwert(old) + m *(in - Mittelwert(old) )` berechnet.
`out:`	Es wird der Mittelwert zurückgegeben.

LowpassK



`LowpassK` ist eine vereinfachte PT1-Funktion mit Verstärkungskonstante `K` (Tiefpassfilter).

Methoden	Argumente	Rückgabewert
`reset`	`initValue::continuous`	`Keiner`
`compute`	`in::continuous` `K::continuous`	`Keiner`
`out`	`Keine`	`continuous`

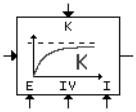
Beim Aktivieren der Methode

`reset:`	Der Lowpass-Wert wird auf den `initValue` gesetzt.

`compute:` Der Lowpass wird über `Lowpass(new) = Lowpass(old)+ (in - Lowpass(old) ) * dT* K` berechnet.

`out:` Es wird der Lowpass-Wert zurückgegeben.

LowpassKEnabled



`LowpassKEnabled` ist eine vereinfachte PT1-Funktion mit Verstärkungskonstante `K` (Tiefpassfilter). Sie muss explizit aktiviert werden.

Methoden	Argumente	Rückgabewert
`reset`	`initValue::continuous` `initEnable::logical`	`Keiner`

`compute`	`in::continuous`
	`K::continuous`
	`enable::logical`
		`Keiner`

`out`	`Keine`	`continuous`

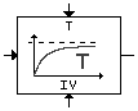
Beim Aktivieren der Methode

`reset:` Wenn `initEnable` `TRUE` ist, wird der Lowpass-Wert auf den `initValue` gesetzt.

`compute:` Wenn `enable` `TRUE` ist, wird der Lowpass über `Lowpass (new) = Lowpass (old)+ (in - Lowpass (old) ) * dT* K` berechnet.

`out:` Es wird der Lowpass-Wert zurückgegeben.

LowpassT



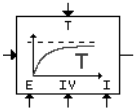
LowpassT ist eine vereinfachte PT1-Funktion mit Zeitkonstante T (Tiefpassfilter).

Methoden	Argumente	Rückgabewert
reset	initValue::continuous	Keiner
compute	in::continuous T::continuous	Keiner
out	Keine	continuous

Beim Aktivieren der Methode

reset:	Der Lowpass-Wert wird auf den initValue gesetzt.
compute:	Der Lowpass wird über Lowpass(new) = Lowpass(old) + (in - Lowpass(old)) * dT/ T berechnet.
out:	Es wird der Lowpass-Wert zurückgegeben.

LowpassTEnabled



LowpassTEnabled ist eine vereinfachte PT1-Funktion mit Zeitkonstante T (Tiefpassfilter). Sie muss explizit aktiviert werden.

Methoden	Argumente	Rückgabewert
reset	initValue::continuous initEnable::logical	Keiner
compute	in::continuous T::continuous enable::logical	Keiner
out	Keine	continuous

Beim Aktivieren der Methode

reset:	Wenn initEnable TRUE ist, wird der Lowpass-Wert auf den initValue gesetzt.

`compute:`	Wenn `enable` `TRUE` ist, wird der Lowpass über `Lowpass (new) = Lowpass (old) + (in - Lowpass (old) ) * dT / T` berechnet.
`out:`	Es wird der Lowpass-Wert zurückgegeben.

13 Fehlersuche und Störungsbeseitigung

In diesem Kapitel werden potentielle Probleme beim Arbeiten mit ASCET erörtert und Hinweise zum Lösen dieser Probleme gegeben. Wenn Sie auf Probleme stoßen, die im vorliegenden Kapitel nicht angesprochen werden, informieren Sie bitte ETAS, damit dieser Abschnitt erweitert werden kann.

Allgemein kann es sich bei jedem von ASCET angezeigten Systemfehler um einen schwerwiegenden Fehler handeln, d.h. es ist ratsam, nach einem Systemfehler alle Daten in einer Datenbank zu speichern. Kommt es nach einem Systemfehler zu einem unerwarteten Systemverhalten, hat der Systemfehler zu einer Unregelmäßigkeit im laufenden System geführt. In diesem Fall sollten Sie ASCET schließen und Windows erneut booten.

13.1 Allgemeine Hinweise

Grenze der Größe einer Datenbank: Die Größe einer ASCET-Datenbank ist auf 4 GByte begrenzt, einzelne Datenbankobjekte auf 128 MByte. Sie sollten beim Arbeiten mit einer großen Datenbank darauf achten, dass diese Grenze nicht erreicht wird, denn im Falle einer Überschreitung dieser Grenze wird die Datenbank zerstört. Nutzen Sie ggf. die Datenbankwerkzeuge zum Komprimieren Ihrer Datenbank.

Umwandeln von Datenbanken: Datenbanken, die mit den Vorgängerversionen ASCET-SD V4.1 oder V4.2 oder ASCET V5.0 entwickelt wurden, werden automatisch in ASCET 5.2 umgewandelt. Dabei ist zu beachten, dass die umgewandelte Datenbank nicht mit älteren Versionen von ASCET verwendet werden kann. Beim Umwandeln wird automatisch eine Sicherheitskopie der alten Datenbank erstellt.

Datenbanken, die mit ASCET-SD V4.0 oder noch früheren Versionen erstellt wurden, können mit ASCET 5.2 *nicht* geöffnet werden.

ASCET unterstützt nur Namen, die ANSI C entsprechen. Zum Gewährleisten von Kompatibilität müssen Sie unter Verwendung des integrierten Umwandlungswerkzeugs die Namen aller Einträge in der Datenbank entsprechend anpassen. Wählen Sie im Komponentenmanager **Tools** → **Database** → **Convert** → **All Names to ANSI C** zum Umbenennen aller Einträge.

Probleme mit Grafikkarten: Beim Auftreten von Problemen mit der Anzeige von ASCET Fenstern liegt wahrscheinlich eine Unverträglichkeit zwischen ASCET, der Grafikkarte und dem Grafikkartentreiber vor. Probieren Sie beim Auftreten solcher Probleme den neuesten Treiber für Ihre Grafikkarte (dieser steht gewöhnlich vom Kartenhersteller über das Internet zur Verfügung) oder versuchen Sie es mit einer anderen Auflösung Ihrer Karte. Alle üblichen VGA- und SVGA-Modi müssten im allgemeinen funktionieren.

Beim Offline-Experiment kommt es zu einer Zeitüberschreitung: Die Zeit (dT) für Offline-Experimente ist auf ca. 3 Tage (in Einheiten von dT) begrenzt, d.h. bei sehr hoher Einstellung von dT (beispielsweise 1000 Sekunden) stürzt das Offline-Experiment nach wenigen Minuten ab.

Unvorhersehbare Auswirkungen bei Verwendung komplexer Zuweisungen: Bei Ausführung komplexer Zuweisungen im Modell kommt es zu unvorhersehbaren Auswirkungen beim Messen komplexer Elemente. Eine komplexe Zuweisung wird durch eine Zuweisung der entsprechenden Zeiger der komplexen Elemente dargestellt, d.h. beide Objekte sind hinterher identisch, und ein Objekt ist „verlorengegangen". Beispielsweise wird in der Zuweisung A=B aus dem Element A das Element B. Das Mess- und Applikationssystem verweist jedoch noch auf beide als separate Objekte. Sie können zwar das „verlorengegangene" Objekt (in diesem Fall Objekt A) messen und kalibrieren, dies hat aber keine Auswirkung und berücksichtigt nicht das Objekt, das nach den Zuweisungen das komplexe Element (d.h. Objekt B) darstellt.

Die Schriftarten werden nicht ordnungsgemäß angezeigt: Die Schriftartfamilie Arial wird unter Win9x/WinNT nicht ordnungsgemäß angezeigt; folglich sind bestimmte Einträge kaum sichtbar. Verwenden Sie stattdessen die Schriftart SansSerif von Microsoft im Schriftgrad 10. Mit dieser Schriftart gibt es keine Anzeigeprobleme.

Probleme mit externen experimentellen Zielsystemen: Eine potentielle Fehlerquelle beim Einsatz des Centronics-Verbindungskabels besteht darin, dass die Geschwindigkeit der parallelen Schnittstelle für das Centronics-Verbindungskabel u. U. zu hoch ist (insbesondere bei Einsatz eines Pentium 200 oder höher). In diesem Fall ist es angeraten, die parallele Schnittstelle beim Einrichten des BIOS des Rechners neu zu konfigurieren.

ASCET arbeitet: Während ASCET arbeitet (z. B. generieren von Code, Übernahme in die Datenbank), sollten Sie nicht versuchen, andere Funktionen in ASCET auszulösen; stattdessen sollten Sie warten, bis die aktuelle Aktion von ASCET abgeschlossen ist. Andernfalls kann das Systemverhalten von ASCET zu unerwarteten Fehlern (z. B. Systemfehlern) führen.

13.2 Probleme mit ASCET

Bestimmte ASCET Experimente werden nicht zum Abschluss gebracht oder laufen nicht ordnungsgemäß ab: Hier liegt das Problem häufig am C-Code, der in ein ASCET Modell integriert wurde. Potentielle Fehler sind falsche Übergabe von Parametern (beim Umwandeln des ASCET Typs `continuous` ist der C-type `double float` zu wählen) sowie unendliche Schleifen

im C-Code. Unendliche Schleifen können auch in rekursiven Objektstrukturen auftreten. Eine Möglichkeit zur Fehlersuche besteht darin, die C-Code-Komponenten auszuschließen.

Unter Umständen läuft der generierte Code nicht im vorgesehenen Zeitrahmen, d.h. seine Ausführungszeit ist zu lang. In diesem Fall muss entweder die Spezifikation geändert oder ein Zeitrahmen mit einem längeren Intervall zugewiesen werden.

Nicht ordnungsgemäß gesetzte oder einfach vergessene Sequenzaufrufe stellen eine weitere Fehlerquelle in diesem Bereich dar.

Die Kompilierung gibt unerklärbare Fehlermeldungen zurück oder kommt nicht zum Abschluss: Wenn Sie während des Kompilierens in ein anderes Fenster klicken, wird die Priorität für das DOS-Feld, in dem die Kompilierung stattfindet, drastisch reduziert, so dass die Kompilierung fast vollständig abgebrochen wird. In diesem Fall können Sie das DOS-Feld durch Doppelklick auf sein Symbol aktivieren.

Außerdem sollten Sie die folgenden Schlüsselwörter vermeiden, die vom Fehlermanagementsystem verwendet werden, um Compilerfehler zum ASCET Modell zurückzuverfolgen: Error, ERROR, Serious, Fatal, illegal, Failed, failed, warning, known format.

ASCET rechnet bei Verwendung temporärer Variabler nicht ordnungsgemäß: Es können automatische temporäre Variable verwendet werden, wenn das Ergebnis eines Ausdrucks in verschiedenen Zweigen zur Anwendung kommen soll. Diese temporären Variablen werden nur einmal berechnet (nach Auswertung des ersten Zweigs). Werden die Zweige unter Verwendung der temporären Variable nur bedingt berechnet (weil sie z. B. in einen Schalter oder einen MUX-Operator eingegeben werden), lässt sich der Wert dieser temporären Variable u. U. nicht korrekt berechnen. Deshalb sollten automatische temporäre Variable nicht verwendet werden, wenn Zweige, die von einer temporären Variable wegführen, in einen bedingten Operator eingegeben werden.

Während Online-Experimenten treten häufig L1-Kommunikationsfehler auf: In diesem Fall ist die Priorität des Kommunikationsprozesses zu gering. Die Priorität dieses Prozesses lässt sich für das Target in der Datei `es1130cp.inv`, `es1130cp_gnu.inv` oder `es1135cp_gnu.inv` im entsprechenden Targetverzeichnis erhöhen. Welche Datei Sie bearbeiten müssen, hängt von der verwendeten Kombination aus Target und Compiler ab.

Diese Datei kommt in der Konfiguration des Compilers zur Anwendung. Hier können Sie die Priorität des Kommunikationsprozesses erhöhen, indem Sie den Parameter `__L1_Prio =` auf die gewünschte Priorität setzen (er hat standardmäßig die niedrigste Priorität, d.h. 0).

Die Dokumentationserzeugung im `.rtf`-Format funktioniert nicht ordnungsgemäß. Bei der Anzeige von `.rtf`-Dateien kann Word für Windows die integrierten bitmap-Bilddateien u. U. nicht anzeigen. In dem Fall müssen Sie u. U. alle Links zu (externen) `*.gif`-Dateien aktualisieren, um sich die Bilder ansehen zu können.

14 Meldungen bei der Codegenerierung

Dieses Kapitel enthält die Warnungen und Fehlermeldungen, die während einer ASCET Codegenerierung auftreten können, zusammen mit Hinweisen und Erläuterungen, wie entstandene Fehler abgestellt werden können. Fehlermeldungen weisen auf schwerwiegende Fehler in der Spezifikation hin, die zu einem Abbruch des Codegenerierungsprozesses führen können. Warnungen weisen auf weniger schwerwiegende Fehler hin. Der Codegenerierungsprozess kann zwar erfolgreich sein, aber u. U. funktioniert der sich ergebende Code nicht wunschgemäß.

14.1 Komponenten

14.1.1 Fehlermeldungen

`method <method_name> must be defined; need a return value`

Beschreibung: In der Komponente wurde eine Methode mit Rückgabewert deklariert, aber mit dem Rückgabewert ist kein Sequenzaufruf verknüpft. Dieser ist erforderlich, da die Methode durch andere Komponenten aufgerufen werden könnte.

Lösung: Bearbeiten Sie den Sequenzaufruf und wählen Sie die Methode, zu der der Rückgabewert gehört, als Sequenzname. Die Sequenznummer muss die höchste mit der betreffenden Methode verknüpfte Nummer sein.

`<method_name> has no argument <argument_name>`

Beschreibung: Eine mit der Methode `method_name` verknüpfte Operation benutzt ein zu einer anderen Methode gehörendes Argument. Eine Methode kann nur die lokalen und globalen Elemente und deren Argumente nutzen, jedoch nicht die Argumente anderer Methoden.

Lösung: Ändern Sie den Sequenzaufruf oder ersetzen Sie das Argument durch ein anderes Element.

`missing argument connection for method <method_name> at block <block_name>`

Beschreibung: Beim Block `block_name` wird zwar die Methode `method_name` aufgerufen, aber es sind nicht alle Argumente verbunden, d. h. es fehlt eins der Argumente. Bei einem Operator bleibt der Methodenname frei.

Lösung: Verbinden Sie die fehlenden Argumente bzw. bei einem Operator wählen sie einen Operator mit der entsprechenden Anzahl von Argumenten.

double sequence number <sequence_number> **for** <name>

Beschreibung: Mit dem Prozess, der Methode, der Aktion oder der Bedingung `name` sind zwei Sequenzaufrufe mit der gleichen Sequenznummer `sequence_number` verknüpft.

Lösung: Ändern Sie eine der Sequenznummern in eine in `name` noch nicht verwendete Sequenznummer.

return value does not belong to <name>

Beschreibung: Ein Rückgabewert einer bestimmten Methode oder einer Bedingung wird einem Sequenzaufruf zugewiesen, der zu einer Methode oder Aktion `name` gehört, die keinen Rückgabewert hat. Der Sequenzaufruf eines Rückgabewerts muss stets der Methode oder der Bedingung zu gewiesen werden, die den betreffenden Rückgabewert definiert.

Lösung: Ändern Sie den Sequenznamen des Sequenzaufrufs des Rückgabewerts in den Namen der Bedingung oder Methode, zu welcher der Rückgabewert gehört.

delay-free loop detected at <block_name> **block**

Beschreibung: Es wird eine Schleife ohne eine Operation in der betreffenden Schleife erzeugt, beispielsweise wird der Rückgabewert eines Operators direkt als Eingabe in den betreffenden Operator eingegeben.

Lösung: Fügen Sie ein Element in die Schleife ein.

type mismatch: expected <type_A>, **got** <type_B>

Beschreibung: Ein Argument des `type_B` wird da verwendet, wo ein Argument des `type_A` erforderlich ist, und der `type_B` lässt sich nicht in den `type_A` umformen. Es wird z. B. ein Argument des Typs `cont` in einen logischen Booleschen Operator eingegeben. Vermutlich ist die Verbindung falsch.

Lösung: Stellen Sie ein Argument des korrekten Typs bereit.

type mismatch: expected <type_A> [<name_A>], **got** <type_B> [<name_B>]

Beschreibung: Ein Element mit dem Namen `name_B` vom Typ `type_B` wird einer Variablen mit dem Namen `name_A` des Typs `type_A` zugewiesen, wo sich `type_B` nicht in `type_A`. umwandeln lässt. Es wird z. B. ein Element des Typs `cont` einer Variablen des Typs `logical` zugewiesen. Vermutlich ist die Verbindung falsch.

Lösung: Ändern Sie den Typ des Elements oder stellen Sie eine korrekte Verbindung her.

`return must be the last operation of` <name>

Beschreibung: Eine Methode mit einem Rückgabewert oder einer Bedingung `name` hat eine Rückgabeanweisung, in deren Sequenzaufruf mit der Methode oder der Bedingung nicht die höchste Sequenznummer bei Sequenzaufrufen verknüpft ist.

Lösung: Ändern Sie die Sequenznummer im Sequenzaufruf in die höchste Nummer bei allen Sequenzaufrufen, die zur Methode oder Bedingung `name` gehören.

<then> **`part of IF block must be specified`**

Beschreibung: Es wird ein `If`-Block verwendet, dessen `Then`-Teil nicht benutzt wird.

Lösung: Spezifizieren Sie den `Then`-Teil. Es muss mindestens ein Sequenzaufruf durch einen Verbinder mit dem `Then`-Teil verknüpft sein.

`state machine needs start state`

Beschreibung: Der Zustandsautomat hat keinen Startzustand.

Lösung: Spezifizieren Sie einen der Zustände des Zustandsautomaten als dessen Startzustand.

`multiple prio` <priority_number> **`for trigger`** <trigger_name> **`in state`** <state_name>

Beschreibung: Der Zustandsautomat enthält zwei Übergänge, die vom Zustand `state_name` wegführen, der mit dem gleichen Trigger `trigger_name` mit der gleichen Priorität `priority_number` verknüpft ist. Dies ist nicht zulässig, da der Übergang nicht eindeutig ist.

Lösung: Ändern Sie eine der Prioritäten so, dass sich alle Prioritäten, die vom gleichen Zustand wegführen und dem gleichen Trigger zugewiesen sind, voneinander unterscheiden.

`unbalanced number of start/stop atomic in` <name>

Beschreibung: Die Methode, der Prozess, die Bedingung oder die Aktion `name` hat Sequenzaufrufe mit verknüpften kleinen Markierungen. Es gibt jedoch eine unabgeglichene Anzahl von Start- und Stopp-Markierungen.

Lösung: Nehmen Sie ein Einfügen oder Löschen von einigen Start- bzw. Stopp-Markierungen vor, so dass deren Anzahl und Aussehen ausgeglichen sind.

Expected consistent datamodel for <element_name> **in** <Class_name>**. Element needs GET/SET direct access - please change attributes OR restore diagram**

oder – für ESDL/C-Code

method <Element_name>/<function_name> **not defined as public in class** <Class_name>

Beschreibung: Das Element oder die Funktion in der Klasse <Class_name> wurde nicht für direkten Zugriff freigegeben/öffentlich gemacht.

Lösung: Aktivieren Sie den direkten Zugriff (Get/Set-Funktionalität) für das Element oder machen Sie die Funktion öffentlich.

14.1.2 Warnungen

<name> **not defined**

Beschreibung: Die Methode, der Prozess oder die Aktion wurde zwar deklariert, aber nicht definiert. Es gibt keinen Sequenzaufruf mit dem Sequenznamen name. Dies betrifft nur Methoden ohne Rückgabewerte.

Lösung: Definieren Sie die Methode, den Prozess oder die Aktion oder löschen Sie deren bzw. dessen Deklaration aus der Komponenten-Schnittstelle.

type mismatch with casting from <type_B> [<name_B>] **to** <type_A>

Beschreibung: Ein Element mit dem Namen name_B des Typs type_B wurde einer Variablen des Typs type_A zugewiesen, wo eine Typenumformung von type_B in type_A erfolgt. Es wird beispielsweise ein Element des Typs cont einer Variablen des Typs sdisc zugewiesen.

Lösung: Ändern Sie den Typ des Elements oder stellen Sie eine korrekte Verbindung her.

argument <argument_name> **of method** <method_name> **not used**

Beschreibung: In der Definition der Methode method_name wird das Argument argument_name der Methode nicht verwendet.

Lösung: Verwenden Sie das Argument `argument_name` in der Definition der Methode oder löschen Sie dieses aus der Definition der Methode.

`unreachable state` `<state_name>`

Beschreibung: Der Zustandsautomat enthält einen Zustand mit dem Namen `state_name`, der aus dem Startzustand nicht erreicht werden kann, d.h. es führt kein Übergang zu dem betreffenden Zustand.

Lösung: Löschen Sie den Zustand oder gestalten Sie den Zustand so, dass er aus dem Startzustand erreichbar ist.

`literal value` `<value>` **`does not fit type`** `<type>` **`- limited to`** `<range_value>`

Beschreibung: Der Wert des Literals ist für die Variable des Typs `type` zu groß. Der Wert des Literals für diese Zuweisung wird automatisch auf den Wert `range_value` begrenzt. Dies gilt nicht für Ausdrücke, die nur aus Literalen bestehen. Der Typ `type` ist entweder `udisc` oder `sdisc`, die einen Bereich einer ganzen Zahl von 32 Bit (`unsigned` oder `signed`) haben.

14.2 Projekte

14.2.1 Fehlermeldungen

`need binding for imported element` `<element_name>`

Beschreibung: Das importierte Element oder die importierte Message `element_name` ist nicht mit einem globalen Element oder einer globalen Message verknüpft.

Lösung: Stellen Sie die Verknüpfung ein (entweder automatisch oder manuell).

`application modes missing for task` `<task_name>`

Beschreibung: Der Task `task_name` wurde kein Betriebsmodus (Application Mode) zugewiesen.

Lösung: Weisen Sie einen Betriebsmodus zu oder löschen Sie die Task `task_name`. Um bestimmte Tasks von der Ausführung auszuschließen, spezifizieren Sie einfach einen zusätzlichen Betriebsmodus mit dem Namen `unused` und weisen Sie diesen den auszuschließenden Tasks zu.

14.2.2 Warnungen

`no start application mode specified - using`

<opmode_name>

Beschreibung: Es wurde kein Betriebsmodus als Startmodus definiert. Der Betriebsmodus `opmode_name` wird automatisch als Startmodus definiert.

Lösung: Legen Sie einen Modus als Startmodus fest, falls nicht schon der richtige Modus als standardmäßige Einstellung gewählt wurde.

`missing trigger event`

Beschreibung: Einer der im Betriebssystem spezifizierten Ereignis-Tasks wurde kein Triggerereignis zugewiesen.

Lösung: Ändern Sie den Modus der betreffenden Task oder weisen Sie der betreffen Task eines der Triggerereignisse zu.

14.3 Festkommacode-Generierung

14.3.1 Fehlermeldungen

`Integer interval [a,b] of variable <name> too large for implementation type`

Beschreibung: Das vom Modellintervall abgeleitete ganzzahlige Intervall [a,b] ist für den gewählten Implementierungstyp zu groß. Vermutlich wurde die Implementierung für dieses Element nicht bearbeitet, oder der Implementierungstyp wurde nicht als ganzzahliger Typ eingerichtet.

Lösung: Bearbeiten Sie die Implementierung für das Element `name`.

`Cannot generate fixed point code for the non-linear formula <formula_name> of variable <name>`

Beschreibung: Die nichtlineare Formel `formula_name` wurde `name` zugewiesen. Die Festkommacode-Generierung unterstützt nur lineare Formeln.

Lösung: Ändern Sie die `name` zugewiesene Formel oder ändern Sie die Formel `formula_name` so, dass es sich um eine lineare Formel handelt.

`Physical interval [a,b] of divisor contains zero`

Beschreibung: Festkommacode kann nicht erzeugt werden, da es zu einer Division durch Null kommen könnte. Dies würde zu einem Implementierungsintervall unendlicher Größe führen.

Lösung: Fügen Sie eine Variable für den Teiler ein und spezifizieren Sie eine sinnvolle Implementierung dafür (das physikalische Intervall sollte keine Null enthalten).

`formula in implementation for <name> not known in current project - using default`

Beschreibung: In der Implementierung für das Element `name` ist die Formel im Rahmen des aktuellen Projekts nicht bekannt. Vermutlich wurde keine Formel zugewiesen. Statt dessen wird die Identitätsformel verwendet.

Lösung: Verwenden Sie eine gültig Formel aus dem Rahmen des aktuellen Projekts für die Implementierung für das Element `name`.

`Interval mismatch in assignment of <variable_name>: [a,b] := [c,d] (will be limited)`

Beschreibung: Der Festkommacode-Generator hat festgestellt, dass es in der Zuweisung der Variablen `variable_name` einen möglichen Konflikt gibt. Der Wert des der Variablen zugewiesenen Ausdrucks liegt innerhalb des Intervalls [c,d]. Dieses Intervall wird über Intervallarithmetik aus den für die Elemente in dem betreffenden Ausdruck spezifizierten Intervallen berechnet. Das Intervall [a,b] für die Variable `variable_name` schließt jedoch das Intervall [c,d] nicht mit ein, so dass es zu einem Überlauf kommen könnte. Um dies zu vermeiden, der Wert des Ausdrucks automatisch auf den Wertebereich der Variablen `variable_name` begrenzt, bevor die Zuweisung ausgeführt wird. Dabei ist zu beachten, dass diese Warnung nicht vermieden werden kann, wenn arithmetische Schleifen vorliegen.

Index

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z